MAPEAMENTO DE TEXTURAS

PGP-OpenGL

INTRODUÇÃO

Até agora, adicionamos detalhes em nossa cena com geometria, cores de vértices e iluminação. Mas geralmente isso não é o bastante para conseguirmos a aparência que nós queremos. Através da técnica de mapeamento de texturas, podemos "pintar" detalhes adicionais em nossa cena sem a necessidade de geometria adicional.

Neste tutorial, são abordados conceitos básicos de mapeamento de texturas utilizando PGP-OpenGL. O repositório PGP possui três exemplos que demonstram estas funcionalidades básicas.

O QUE É MAPEAMENTO DE TEXTURAS?

No caso mais básico, mapeamento de texturas é um método para adicionar detalhes à geometria a ser renderizada e isto é feito ao mostrar uma imagem na superfície descrita pela geometria. Observe a Figura 1, que mostra um cubo com uma imagem mapeada em sua superfície.

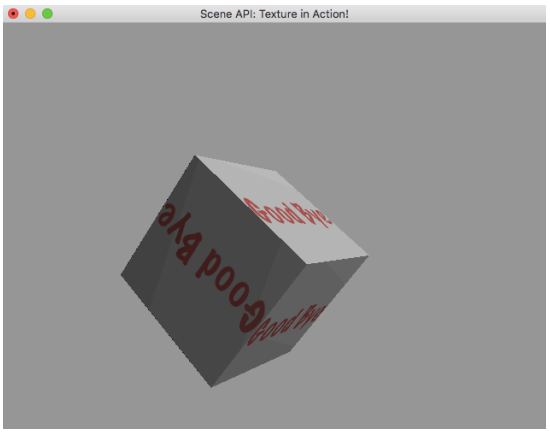

FIGURA 1. CUBO COM TEXTURA.

Apenas utilizando técnicas que manipulam geometria e topologia da malha, esta cena relativamente simples deveria ser muito difícil de ser construída e desnecessariamente complexa. A palavra *Good Bye* deveria ser construída cuidadosamente de muitos triângulos com as cores apropriadas. Certamente que essa abordagem é possível, mas a geometria adicional necessária deveria torna-la impraticável para ser utilizada mesmo em cenas marginalmente complexas.

Mapeamento de textura, por outro lado, faz a cena mostrada na Figura 1 incrivelmente simples. Tudo que é requerido é uma imagem com a palavra *Good Bye* em um formato apropriado, um atributo adicional de vértice na malha e algumas poucas adições ao nosso código de *shaders*.

Neste tutorial, será mostrado como isso pode ser feito utilizando as facilidades da PGP para OpenGL 2.1. Para isso, mostraremos um exemplo.

PRÉ-REQUISITOS

Precisamos de alguma biblioteca para carregar imagens e precisamos baixar algumas imagens para mapear em nossas malhas. Para carregar imagens, PGP incorporou o código do respositório *lodepng* [1]. Contudo, você não precisa se preocupar com lodepng, provemos uma função no cabeçalho *textureutils.hpp* (veja o sitio da PGP no *GitHub*, no diretório *includes*) que facilita o carregamento de uma imagem.

O conteúdo de uma imagem é armazenado em um vetor de bytes sem sinal (*unsigned char* em C++). A função

```
getTextureFromImage("texture.png", s, t);
```

retorna um objeto do tipo `std::vector<unsigned char>`. Este objeto armazena o conteúdo da imagem. O argumento *"texture.png"* é o nome da imagem, que deve estar no formato *png* (único formato suportado). Os argumentos *s* e *t* são do tipo *size_t* e retornam o tamanho da textura convertida para potência de 2. Toda textura é convertida pela função *getTextureFromImage* em potência de 2. Observe que, dessa forma, temos uma função que facilita o carregamento de uma imagem *png* como uma textura.

OS SHADERS

Tanto o *vertex shader* quanto o *fragmente shader* devem ser preparados para aplicarem a textura sobre a superfície do objeto. Antes de mostrar quais alterações devem ser feitas nos *shaders*, é preciso mostrar o relacionamento básico entre vértices e coordenadas de texturas. A Figura 2 mostra a imagem da textura utilizada na superfície do cubo exibido na Figura 1.

FIGURA 2. IMAGEM DE TEXTURA.

Em OpenGL, uma textura possui um espaço de coordenadas próprio, cujos valores variam entre 0 (zero) e 1 (um), tanto na horizontal qual na vertical. Assim, o canto superior esquerdo possui coordenadas (0, 0), enquanto o canto inferior direito possui coordenadas (1, 1), conforme ilustra a Figura 3. As coordenadas referentes à textura devem ser associadas aos vértices de cada face. Portanto, o *vertex shader* deve conter um atributo indicando qual a coordenada de textura do vértice. As coordenadas de texturas são da forma *(s, t)*. Em OpenGL é costume utilizar as letras *s* e *t* para indicar essas coordenadas.

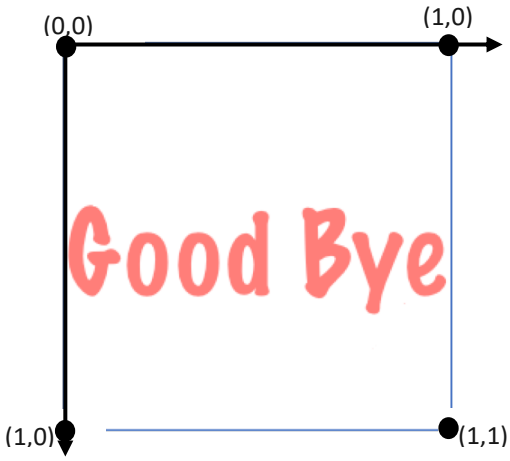

FIGURA 3. COORDENADAS DE TEXTURAS

O exemplo *cap5_ex1* tem o seguinte *vertex shader*:

```
1  #version 120
2  attribute vec3 coord3d; //coordenadas do vertice
3  attribute vec2 texcoord; //coordenadas de textura mapeadas para o vertice
4  uniform mat4 model; //matriz de transformacao do modelo
5  uniform mat4 view; //matriz de posicao e orientacao da camera
6  uniform mat4 projection; //matriz de projecao
7  varying vec2 ftexcoord; //coordenadas de textura interpolada para cada pixel
8  void  main(void) {
9      ftexcoord = texcoord;
10     gl_Position =  projection * view * model * vec4(coord3d, 1.0);
11 }
```

FIGURA 4. *VERTEX SHADER.*

Observe na linha 3 que o atributo *texcoord* é do tipo *vec2*. Observe que informamos apenas as coordenadas de textura dos vértices e não o mapeamento direto de cada pixel da imagem da textura para cada ponto da face do cubo. Esse mapeamento vai ser feito automaticamente quando passarmos as coordenadas dos vértices para o *fragmente shader* (FS) por meio de uma variável qualificada como *varying*. Observe que isso vai fazer com que os valores no FS sejam interpolações dos valores dos vértices. A variável *varying* ftexcoord conterá essa interpolação e será transmitida ao FS. Para isso, precisamos declarar *ftexcoord* no FS (linha 3):

```
1  #version 120
2
3  varying vec2 ftexcoord;
4  uniform sampler2D tex;
5
6  void main(void){
7      gl_FragColor = texture2D(tex, ftexcoord);
8  }
```

FIGURA 5. *FRAGMENT SHADER.*

Em OpenGL, podemos trabalhar com várias texturas. Para isso, precisamos indicar um código para cada textura que iremos trabalhar. Esse código, no FS, é do tipo *sampler2D*, como mostra a Figura 5 (linha 4). Cada textura receberá um valor numérico, assim, a variável *tex* receberá um valor numérico que indica qual textura estamos nos referindo.

A função *texture2D* (utilizada na linha 7 do FS) retorna o valor de pixel relativo à textura indicada por *tex* e que esteja na posição indicada pela variável *ftexcoord*. Aqui, para cada fragmento, estamos atribuindo somente a cor de um pixel da textura. Podemos combinar a cor desse pixel com a cor de outras texturas ou mesmo com os efeitos de um modelo de iluminação. A Figura 1 foi obtida com uma textura combinada com um modelo de iluminação difuso. O código completo do exemplo mostrado na Figura 1 pode ser encontrado em [6].

PGP E OS SHADERS

Agora que o nosso *shader* está pronto, vamos visualizar como enviar dados para o nosso *shader*. Temos um objeto da variável *proxy*, que é do tipo *ShaderProxy*, que mapeará os dados do objeto 3D que queremos pintar para os atributos e uniformes em nossos *shaders*. A primeira coisa que precisamos fazer é enviar os dados de atributos para as variáveis *coord3d* e *texcoord*. Além disso, precisamos definir o valor do *uniform tex* e os índices do nosso cubo. A Figura 6 mostra estas configurações. Estou supondo aqui que as variáveis *cube_vertices*, *cube_textures* e *cube_indices* já foram inicializadas com valores adequados. Vejam no exemplo [2] quais são estes valores.

```
165     try {
166         proxy->useProgram();
167         proxy->setAttribute("coord3d", cube_vertices, sizeof(cube_vertices));
168         proxy->setAttribute("texcoord", cube_textures, sizeof(cube_textures), 2);
169         proxy->setUniform1i("tex", 0);
170         proxy->setElementPrimitive(cube_indices, sizeof(cube_indices));
171     } catch(string error) {
172         cout<<error<<endl;
173         return 0;
174     }
```

FIGURA 6. CONFIGURAÇÃO ATRIBUTOS E OUTROS DADOS DE NOSSOS SHADERS.

É necessário capturar a imagem da textura. Para isso, usamos a função *getTextureFromImage*, como mostra a Figura 7:

```
177     std::vector<unsigned char> image;
178     size_t s, t;
179
180     try {
181         image = getTextureFromImage("texture.png", s, t);
182     } catch(string e) {
183         cout<<e<<endl;
184         return 0;
185     }
```

FIGURA 7. LENDO UMA IMAGEM COMO UMA TEXTURA.

Observe na Figura 7 que os dados da imagem foram armazenados na variável *image*. Agora precisamos associar essa imagem às informações de *shader* do cubo, que são representadas pela variável *proxy*. Isso é mostrado na Figura 8.

Quando o tamanho da textura for menor do que o tamanho de tela da superfície sobre o qual será aplicada, o tamanho da imagem deve ser aumentado. Neste caso, temos um precesso de *magnification* da imagem. Se o tamanho da textura for maior do que o tamanho de tela da superfície, temos um caso de *minification*. Em OpenGL utilizamos filtros para definir como o a imagem será aumentada (*magnification*) ou diminuída (*minimization*). Neste trabalho, para o processo de *magnification* e *minimization*, usamos o filtro *GL_LINEAR*, que irá gerar novos pixels por meio de uma combinação linear de *pixels* na vizinhança.

```
187     try {
188         proxy->setTexture(&image[0], s, t);
189         glActiveTexture(GL_TEXTURE0);
190         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
191         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
192         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
193         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
194 
195         camera->update();
196         cubo->update();
197     } catch(string error) {
198         cout<<error<<endl;
199         return 0;
200     }
```

FIGURA 8. DEFININDO A TEXTURA E ESTABELECENDO COMO SEUS DADOS DEVEM SER TRATADOS.

Após definirmos qual imagem queremos associar à malha do cubo, devemos definir como essa textura irá se comportar em algumas situações.

Como vimos, as coordenadas de textura devem estar no intervalo *[0,1]*. Mas é possível prover valores maiores ou menores do que 1. Neste caso, *deve-se* definir como a superfície deve ser coberta pela textura. Isso se chama de *wrapping*. Uma abordagem comum é utilizar a opção *GL_REPEAT* que repetirá o padrão da textura proporcional ao valor de suas coordenadas.

É importante observar que todas as propriedades discutidas no parágrafo anterior são configuradas por meio da função *glTexParameteri(target, propriedade, valor)*, onde *target* é o tipo de textura que estamos trabalhando (*GL_TEXTURE_2D* é o valor que utilizamos neste tutorial), *propriedade* é o tipo de propriedade que queremos alterar, por exemplo, *GL_TEXTURE_MAG_FILTER* indica que vamos alterar o comportamento de *magnification* da textura. E um valor para essa propriedade é *GL_LINEAR*.

O exemplo completo estudado neste tutorial pode ser encontrado em [2]. Exemplos de código com OpenGL 2.1 e texturas podem ser vistos em [3]. Outros textos sobre o assunto podem ser encontrados em [4], [5] e [6]. Outros dois exemplos interessantes da PGP são [7] e [8]. Destaca-se que o exemplo [7] mostra como combinar duas texturas para se obter um efeito desejado.

REFERÊNCIAS

[1] http://lodev.org/lodepng.

[2] https://github.com/professorgilzamir/pgp/tree/master/src/cap5_ex1

[3] https://www.khronos.org/opengl/wiki/Texturing_a_Sphere

[4] http://www.opengl-tutorial.org/beginners-tutorials/tutorial-5-a-textured-cube/

[5]http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-2.1:-Buffers-and-Textures.html

[6] https://github.com/professorgilzamir/pgp/tree/master/src/cap5_ex2

[7] https://github.com/professorgilzamir/pgp/tree/master/src/cap5_ex3