5
Detalhamento de superfícies com tesselação em GPU

Este capítulo descreve o procedimento realizado para a geração de detalhes em GPU usando tesselação em *hardware*. Inicialmente, será dada uma visão geral de cada método e, em seguida, serão apresentados detalhes de suas implementações, as quais têm como objetivo a geração e aplicação de detalhes estruturais sobre superfícies paramétricas, construídas a partir de um único *quad* enviado à GPU na forma de um *patch*. Para aprimorar a qualidade da imagem gerada por meio deste procedimento, também é aplicada a técnica de *normal mapping* sobre a superfície. Os resultados de cada método exposto neste capítulo são apresentados e analisados no Capítulo 6, tanto em termos de qualidade visual quanto desempenho.

5.1
Definição das superfícies paramétricas

Para os experimentos descritos neste capítulo, foram utilizados três tipos de superfícies: esfera, cilindro e *torus*, cujas coordenadas são computadas de acordo com as equações 5-1 a 5-9, as quais recebem como parâmetros as coordenadas *s* e *t*, definidas no domínio [0.0, 1.0].

$$x_{sphere} = r\sin(s)\cos(\mathrm{t}) \tag{5-1}$$

$$y_{sphere} = r\sin(s)\sin(t) \tag{5-2}$$

$$z_{sphere} = r\cos(s) \tag{5-3}$$

onde $s \in [0, \pi], t \in [0, 2\pi]$ e *r* é o raio da esfera.

$$x_{cylinder} = r\cos(s),\ s \in [0, 2\pi], t \in [0, 1] \tag{5-4}$$

$$y_{cylinder} = r\sin(s),\ s \in [0, 2\pi], t \in [0, 1] \tag{5-5}$$

$$z_{cylinder} = h\,t - (0.5h),\ t \in [0, 1] \tag{5-6}$$

onde $s \in [0, 2\pi], t \in [0, 1]$, *r* é o raio da base e *h* é a altura do cilindro.

$$x_{torus} = (R + r\cos(t))\cos(\mathrm{s}),\ s \in [0, 2\pi], t \in [0, 2\pi] \quad (5\text{-}7)$$

$$y_{torus} = (R + r\cos(t))\sin(\mathrm{s}),\ s \in [0, 2\pi], t \in [0, 2\pi] \quad (5\text{-}8)$$

$$z_{torus} = r\sin(\mathrm{t}),\ s \in [0, 2\pi], t \in [0, 2\pi] \quad (5\text{-}9)$$

onde $s \in [0, 2\pi], t \in [0, 2\pi]$, r é o raio do tubo e R é o raio do centro do torus ao centro do tubo.

5.2
Detalhamento utilizando mapas pré-computados

5.2.1.
Visão geral

A geração de detalhes por meio de mapas pré-computados obedece a um princípio similar ao do *displacement mapping* (Cook, 1984), no sentido de que utiliza informações codificadas em um mapa de altura para realizar diretamente o deslocamento da geometria do modelo. Neste caso, no entanto, os vértices da superfície deslocada são gerados na GPU através da subdivisão do *patch* de entrada. Uma vez subdividido, esse *patch* é transformado através de equações paramétricas, como as apresentadas na Seção 5.1, para gerar a topologia da superfície de interesse.

Além do *patch*, são enviadas para o *hardware* gráfico duas texturas: a primeira contém as informações de cor da superfície e a segunda codifica os mapas de normais (utilizado apenas no *fragment shader* para melhorar a qualidade da imagem renderizada) e altura, o qual constitui um escalar que é armazenado diretamente no canal alfa do mapa de normais. Exemplos dos mapas utilizados para este procedimento são mostrados na Figura 14.

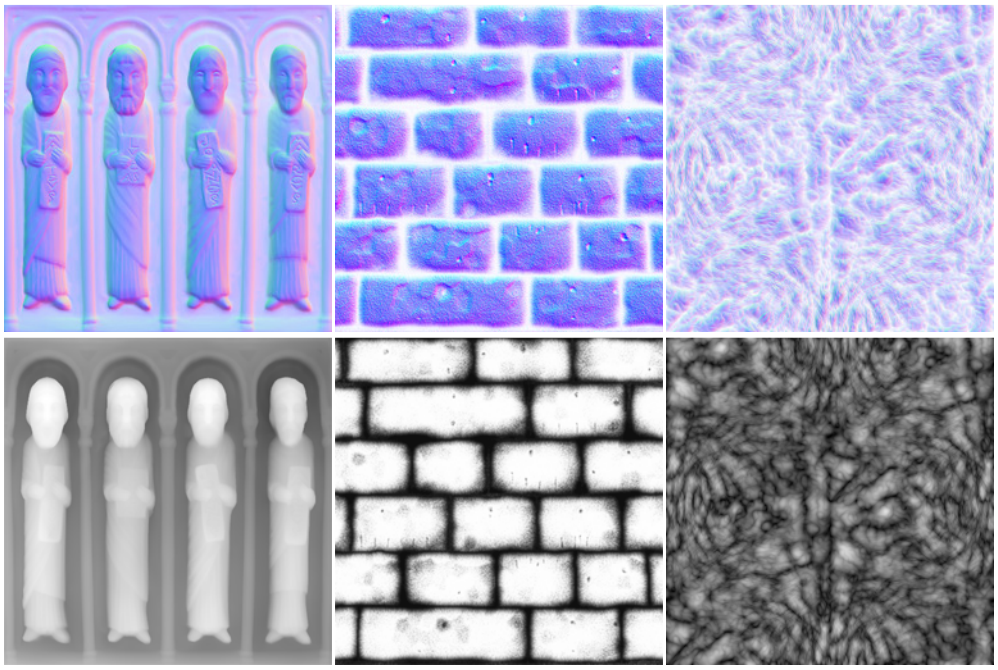


Figura 14: Mapas de normais (linha superior) e altura (linha inferior) pré-computados.

O mapa de altura é utilizado após a etapa de tesselação para deslocar os vértices da superfície. Esse processo é efetuado no espaço do objeto para cada vértice produzido e, caso esse mapa não esteja alinhado com a superfície, necessita do vetor normal deste vértice para definir a correta direção de deslocamento. Esse vetor, computado durante a etapa de subdivisão, é definido por meio do produto vetorial entre os vetores tangente e binormal associados ao vértice sendo transformado, os quais são determinados a partir dos gradientes das equações paramétricas apresentadas na Seção 5.1. Por fim, junto à textura de cor, que é mapeada diretamente sobre o objeto, o mapa de normais enviado à GPU é usado durante os cálculos de iluminação para perturbar as normais da superfície por *pixel*, processo que consiste no tradicional *normal mapping*, utilizado unicamente para melhorar o detalhamento na área interna à superfície renderizada.

5.2.2. Implementação

Na implementação deste trabalho, todas as transformações sobre vértices são calculadas no *tessellation evaluation shader*, após o estágio de geração de primitivas, quando a topologia da malha é definida. Dessa forma, o *vertex shader*

é responsável apenas pela propagação dos pontos de controle para o próximo estágio.

Como visto na Seção 3.1.2, o *tessellation control shader* recebe da aplicação os níveis de tesselação interno e externo. Visto que a primitiva de interesse é um *quad*, os dois valores correspondentes ao nível de tesselação interno e os quatro correspondentes ao nível externo são atribuídos. Também é definido nesta etapa o qualificador de *layout* do *patch* de saída, mostrado na Listagem 1, especificado através do identificador `vertices`, que recebe o número de vértices que constituem a primitiva a ser subdividida.

```
layout(vertices = 4) out;
```

Listagem 1: Declaração do layout de saída no tessellation control shader.

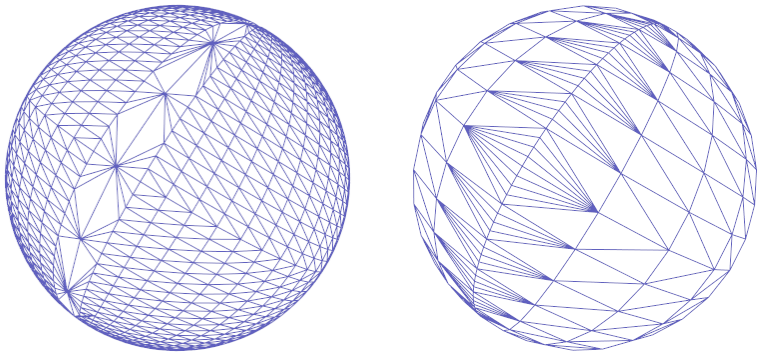
Nesta etapa, é aplicada uma pequena simplificação quando da atribuição dos níveis de tesselação, que é realizada apenas uma vez para o primeiro vértice de cada *patch*. Este controle é realizado por meio da variável interna `gl_InvocationID`, que armazena o índice no *patch* de entrada do vértice sendo processado atualmente, como mostra a Listagem 2. Na verdade, em virtude do número de *patches* utilizado aqui, esta modificação não possui impacto significativo sobre o desempenho, mas em casos de malhas mais complexas, pode evitar algumas operações redundantes.

```
if(gl_InvocationID == 0)
{
      // Atribuição dos níveis de tesselação.
      gl_TessLevelInner[0] = tessInner;
      gl_TessLevelInner[1] = tessInner;
      gl_TessLevelOuter[0] = tessOuter;
      gl_TessLevelOuter[1] = tessOuter;
      gl_TessLevelOuter[2] = tessOuter;
      gl_TessLevelOuter[3] = tessOuter;
}
```

Listagem 2: Atribuição dos níveis de tesselação no tessellation control shader.

Como a subdivisão é realizada sobre apenas um *patch*, a influência da variação do nível externo sobre a topologia da superfície não é tão significante quanto a do nível interno. Isto ocorre porque as mudanças no nível externo modificam apenas as arestas que “fecham” a primitiva (aquelas localizadas na borda do *patch* subdividido). Sobre as superfícies paramétricas apresentadas aqui, este efeito é perceptível apenas nos pontos pertencentes às arestas que delimitam a superfície, como mostra a Figura 15(a), que mostra uma superfície com níveis de

tesselação interno maior que o de tesselação externo. O efeito da variação do nível interno, por outro lado, tem influência mais direta sobre o nível de detalhamento desejado, como visto na Figura 15(b), onde o nível de tesselação interno é menor que o externo.



(a) (b)

Figura 15: Níveis de tesselação sobre superfície paramétrica.

O *tessellation evaluation shader* recebe a primitiva subdividida e é responsável pela computação das posições e atributos finais dos vértices antes do processo de iluminação. Conforme exposto na Seção 3.1.4, a declaração do *layout* de entrada utiliza três identificadores para definir o tipo de subdivisão, espaçamento e orientação da nova primitiva, os quais, para o *shader* de tesselação apresentado aqui, são: `quads`, `equal_spacing` e `ccw`, respectivamente. A declaração do layout utilizando estes identificadores pode ser vista na Listagem 3.

```
layout(quads, equal_spacing, ccw) in;
```

Listagem 3: Declaração do layout de entrada no tessellation evaluation shader.

As posições dos vértices da superfície são computadas por meio de um conjunto de funções que mapeiam as equações 5-1 a 5-9. Estas funções recebem como parâmetros de entrada as posições dos vértices que pertencem ao *patch* subdivido (as quais são interpoladas por uma função auxiliar) vindo do *primitive generator* e retornam a posição do vértice na superfície, bem como os vetores tangente e binormal associados (retornados como variáveis de saída da função paramétrica).

No caso de um ou mais *patches* de entrada com maior nível de complexidade, a interpolação garante que os vértices estarão em suas posições corretas no interior de cada nova primitiva e, consequentemente, da malha inteira. No caso de apenas um *quad*, também seria possível utilizar diretamente as

coordenadas de tesselação (*u, v*), obtidas através da variável interna `gl_TessCoord`, já que essas coordenadas são definidas no mesmo domínio da primitiva subdividida, isto é, no intervalo [0.0, 1.0].

A quantidade de deslocamento necessária para adequar o vértice à curvatura da superfície é obtida por meio de um acesso a textura, que obtém apenas a componente alfa do *texel* do mapa de altura nas coordenadas (*u, v*). Como este valor é escalar, a direção de deslocamento é determinada através da multiplicação da quantidade de deslocamento com o vetor normal do respectivo vértice (calculado durante a etapa de subdivisão) no espaço do objeto. Finalmente, o vetor contendo o valor de deslocamento devidamente ajustado é adicionado às coordenadas do vértice após a definição da topologia básica da superfície. Este processo gera as posições dos vértices no espaço do objeto. Assim, a etapa final consiste em realizar a transformação para as coordenadas de *clip* através do produto entre as coordenadas do vértice e as matrizes *model*, *view* e *projection*. A seção principal do código do *tessellation evaluation shader* para definição da topologia da superfície com deslocamento geométrico é mostrado na Listagem 4.

```
uniform mat4 modelView;
uniform mat4 projection;
uniform int primitive_type;
uniform sampler2D normalDepthMap;

// Função auxiliar para interpolação
vec4 interpolate(
            in vec4 v0, in vec4 v1,
            in vec4 v2, in vec4 v3)
{
      vec4 a = mix(v0, v1, gl_TessCoord.x);
      vec4 b = mix(v3, v2, gl_TessCoord.x);
      return mix(a, b, gl_TessCoord.y);
}

void main()
{
      vec4 pos;
      vec3 normal, tangent, binormal;
      float disp;

      pos = interpolate(
      gl_in[0].gl_Position, gl_in[1].gl_Position,
      gl_in[2].gl_Position, gl_in[3].gl_Position);

      // Definição da topologia da superfície.
      switch(primitive_type)
      {
            case SPHERE:
                  pos = sphere(pos.xy, tangent, binormal);
                  break;
            case CYLINDER:
```

```
                pos = cylinder(pos.xy, tangent, binormal);
                break;
          case TORUS:
                pos = torus(pos.xy, tangent, binormal);
                break;
    }

    tangent = normalize((tangent));
    binormal = normalize((binormal));
    normal = normalize(cross(tangent, binormal));

    // Ajuste da direção de deslocamento.
    vec3 adj =
    normal * texture(normalDepthMap, gl_TessCoord.xy).a;

    // Aplicação do deslocamento geométrico.
    pos.xyz = pos.xyz + adj;

    // Transformação para o espaço de clip.
    gl_Position = (projection * modelView) * pos;
}
```

Listagem 4: Deslocamento geométrico no tessellation evaluation shader.

Além disso, é necessário propagar para o próximo estágio as coordenadas de textura e os vetores de luz (*light*) e visualização (*view*) derivados durante esta etapa para cada vértice gerado durante a tesselação, onde a iluminação será calculada. Como os domínios de tesselação e parametrização são coincidentes, os valores das coordenadas de textura correspondem às coordenadas de tesselação especificadas pela variável `gl_TessCoord`. Os vetores *light* e *view* são propagados no espaço tangente (ou espaço da textura), onde são realizadas as computações do *normal mapping* no *fragment shader*, o que implica na necessidade de transformá-los por meio do produto com a matriz de mudança de base definida na Equação 5-10, a qual é construída com os vetores tangente (*t*), binormal (*b*) e normal (*n*), calculados neste estágio.

$$\begin{bmatrix} t_x & t_y & t_z \\ b_x & b_y & b_z \\ n_x & n_y & n_z \end{bmatrix} \tag{5-10}$$

Por fim, o *fragment shader* realiza os cálculos de iluminação utilizando uma implementação simples do modelo de Phong, que produz boa qualidade visual a um baixo custo computacional, e da técnica de *normal mapping*. Este procedimento é codificado na função mostrada na Listagem 5, a qual recebe as coordenadas de textura e os vetores *light* e *view* normalizados no espaço tangente. As normais utilizadas neste cálculo são aquelas obtidas do mapa enviado à GPU

(já codificado no espaço tangente), amostradas com as coordenadas de textura recebidas do *tessellation evaluation shader*.

```
uniform vec4 matAmbientColor;
uniform vec4 matDiffuseColor;
uniform vec4 matSpecularColor;

uniform sampler2D colorTexture;
uniform sampler2D normalDepthMap;

vec4 computeIllumination(
            vec2 texCoord, vec3 light_ts, vec3 view_ts)
{
      // Amostragem do mapa de normais e textura de cor.
      vec3 normal_ts = normalize(
            texture(normalDepthMap, texCoord) * 2 - 1).rgb;

      vec4 baseColor = texture(colorTexture, texCoord);

      // Computação da componente difusa:
      vec3 light_tsAdj =
            vec3(light_ts.x, -light_ts.y, light_ts.z);

      vec4 diffuse =
            clamp(dot(normal_ts, light_tsAdj), 0.0, 1.0) *
            matDiffuseColor;

      // Computação da componente especular:
      vec4 specular = vec4(0.0, 0.0, 0.0, 0.0);
      vec3 half = normalize(view_ts + light_ts);
      float spec = clamp(dot(normal_ts, half), 0.0, 1.0);

      specular = pow(spec, 75) * matSpecularColor;

      // Composição da cor final:
      vec4 finalColor = ((matAmbientColor + diffuse) * baseColor +
specular);

      return finalColor;
}
```

Listagem 5: Método para iluminação da superfície no fragment shader.

5.3
Detalhamento procedimental

5.3.1.
Visão geral

Conforme discutido na Seção 3.2, a geração de texturas procedimentais permite a codificação de uma variedade de efeitos para enriquecimento de ambientes virtuais. Uma das aplicações de texturas procedimentais baseadas em ruído no detalhamento de superfícies é o chamado *bump mapping* procedimental,

onde os mapas de normais utilizados para perturbar a superfície são gerados procedimentalmente. Esta aplicação, no entanto, também constitui apenas um efeito de iluminação que apresenta as mesmas limitações das técnicas baseadas em imagens apresentadas no Capítulo 4.

O uso do recurso de tesselação associado a técnicas de texturas procedimentais permite a simulação de diferentes efeitos de caráter aleatório sobre superfícies. Como na abordagem anterior, os vértices são gerados diretamente na GPU a partir de um *quad*, mas agora deslocados de acordo com um efeito procedimental baseado em ruído, o qual também é calculado na GPU.

É possível enviar uma textura gerada na CPU codificando o efeito desejado, substituindo, assim, o amplo volume de operações necessárias para computar os efeitos de ruído na GPU por algumas instruções de acesso à textura. No entanto, este experimento tem como objetivo avaliar o impacto da geração de detalhamento geométrico quando implementado inteiramente no hardware gráfico.

Para realizar o deslocamento dos vértices, foram desenvolvidos três efeitos procedimentais baseados em funções de ruído como *stripes*, *turbulence* e *fractal* (Ebert, et al., 2002): *marble*, *lumpy* e *ridge*. A parametrização da superfície ocorre da mesma forma daquela exposta na Seção 5.2, utilizando as mesmas funções para o cálculo da posição do vértice e computação dos vetores binormal e tangente. Em seguida, a quantidade de deslocamento é calculada através da chamada a uma função que codifica o efeito de detalhamento desejado, sendo então adicionada à posição do vértice após a devida correção da direção de deslocamento ao longo da normal.

Uma vez definida a topologia da malha, é realizado o processo de iluminação da superfície. Visto que não foi enviada à GPU uma textura contendo o mapa de normais, o cálculo destes vetores também é efetuado por meio de funções de ruído. O cálculo das normais *on the fly* também proporciona a liberdade de escolha do espaço onde elas podem ser computadas, por não serem recebidas pré-codificadas no espaço tangente. Dessa forma, optou-se por realizar os cálculos de iluminação diretamente no espaço do olho, evitando transformações desnecessárias entre espaços de coordenadas.

5.3.2. Implementação

Nesta abordagem, todos os efeitos procedimentais foram gerados inteiramente na GPU por meio do *tessellation evaluation shader* e *fragment shader*. O primeiro é responsável pelo deslocamento da geometria e atributos de iluminação de acordo com o efeito desejado, enquanto o segundo é responsável pelo cálculo da iluminação propriamente dita com base no efeito procedimental aplicado no estágio anterior. Portanto, os códigos para *vertex shader* e *tessellation control shader* seguem a ideia apresentada na Seção 5.2

No *tessellation evaluation shader*, as instruções de acesso à textura para obtenção da quantidade de deslocamento foram substituídas por uma chamada à função responsável pela geração do efeito de deslocamento desejado, as quais são mostradas nas Listagem 6. Observa-se que todas culminam com uma ou mais chamadas à função de ruído `noise()`, que segue a implementação apresentada por Perlin (Perlin, 2004).

```
// Funções para computação do efeito MARBLE
float marble(in vec3 pos)
{
      return (0.04f *
              stripes(pos.x + 2 *
              turbulence(pos.x,pos.y,pos.z), 1.6));
}

float turbulence(in float x, in float y, in float z) {
   float t = -0.5;
   float i = 128.0/12.0;
   for(float f = 1.0; f <= i; f *= 2)
      t += abs(noise(vec3(f*x,f*y,f*z)) / f);
   return t;
}

float stripes(in float x, in float f) {
   float t = 0.5 + 0.5 * sin(f * x * 2 * PI);
   return (t * t - 0.5);
}

// Função para computação do efeito LUMPY
float lumpy(in vec3 pos)
{
      return 0.2 * noise(2.0 * pos);
}

// Funções para computação do efeito RIDGE
float ridge(in vec3 pos)
{
      return ridgedmf(pos, 8);
}
```

```
float ridgedmf(in vec3 p, in int octaves)
{
      float lacunarity = 2.0;
      float gain = 0.5;
      float offset = 1.0;

      float sum = 0.0;
      float freq = 2.0, amp = 0.5;
      float prev = 1.0;

      for(int i = 0; i < octaves; i++)
      {
            float n = ridgeoff(noise(p*freq), offset);
            sum += n * amp * prev;
            prev = n;
            freq *= lacunarity;
            amp *= gain;
      }
      return sum;
}

float ridgeoff(in float h, in float offset)
{
    h = abs(h);
    h = offset - h;
    h = h * h;
    return h;
}
```

Listagem 6: Funções para cálculo de dos efeitos de deslocamento.

Uma vez obtida a quantidade de deslocamento, sua direção é ajustada de acordo com o vetor normal e adicionada à posição do vértice, que é multiplicada pelas matrizes *model*, *view* e *projection* para definir a topologia da superfície. Por fim, são propagadas as variáveis necessárias para a iluminação no *fragment shader*: os vetores normal, *light* e *view* (os dois últimos definidos no espaço do olho).

Em uma primeira abordagem, os vetores tangente, binormal e normal foram calculados para cada vértice no *tessellation evaluation shader* e propagados para o seguinte. Entretanto, apenas a interpolação realizada pela GPU não foi capaz de fornecer o detalhamento desejado e a qualidade visual dos efeitos resultantes não foi satisfatória, resultando em imagens borradas que pouco se assemelhavam ao efeito que se desejava obter, e, por esta razão, optou-se pelo cálculo desses vetores diretamente no *fragment shader*. Para isto, foram propagadas as coordenadas de tesselação obtidas no *tessellation evaluation shader* para realizar uma nova computação dos valores de deslocamento, que agora seriam utilizados para computar as normais em nível de *pixel*.

Inicialmente, foram calculados os vetores tangente e binormal a partir de três diferentes valores de deslocamento, em um procedimento análogo ao realizado no estágio anterior para definir a topologia da superfície. Os parâmetros de entrada nas três ocorrências, no entanto, correspondem às coordenadas de tesselação originais inalteradas, perturbadas na componente *x* e perturbadas na componente *y* com base em um fator constante. Calculados os três deslocamentos, os vetores tangente e binormal foram definidos como a diferença entre o valor de deslocamento com coordenadas inalteradas e as perturbadas nos eixos *x* e *y*, respectivamente. Em seguida, calculou-se a normal por meio do produto vetorial entre os dois vetores computados para, por fim, realizar a iluminação da superfície em um procedimento semelhante ao apresentado na Seção 5.2. A Listagem 7 mostra um excerto do código do *fragment shader*, que realiza as computações apresentadas para uma esfera detalhada com o efeito *marble*.

```
void main()
{
      vec4 matAmbientColor = vec4(0.125, 0.125, 0.25, 1.0);
      vec4 matDiffuseColor = vec4(0.25, 0.25, 0.5, 1.0);
      vec4 matSpecularColor = vec4(1.0, 1.0, 1.0, 1.0);

      float u0 = In.TexCoords.x;
      float v0 = In.TexCoords.y;

      // Perturbação das coordenadas de tesselação em x e y
      float u1 = (In.TexCoords.x + 0.0001);
      float v1 = (In.TexCoords.y + 0.0001);

      // Computação dos valores de deslocamento
      vec3 frag, frag_dx, frag_dy;
      frag    = sphere(vec2(u0, v0));
      frag_dx = sphere(vec2(u1, v0));
      frag_dy = sphere(vec2(u0, v1));

      float disp, disp_dx, disp_dy;
      disp    = marble(frag);
      disp_dx = marble(frag_dx);
      disp_dy = marble(frag_dy);

      // Computação dos valores base para tangente e binormal
      frag    = frag    + normalize(frag)    * disp;
      frag_dx = frag_dx + normalize(frag_dx) * disp_dx;
      frag_dy = frag_dy + normalize(frag_dy) * disp_dy;

      // Cálculo do vetor normal
      vec3 tangent  = normalize(frag_dx - frag);
      vec3 binormal = normalize(frag_dy - frag);
      vec3 normal   = normalize(cross(tangent, binormal));

      // Vetores light & view no espaço do olho
      vec3 light_es = normalize(In.Light);
      vec3 view_es  = normalize(In.View);
```

```
      // Componentes difusa e especular
      float diff =
            max(clamp(dot(normal, light_es), 0.0, 1.0), 0.0);
      float spec = 0.0;
      if(diff > 0.0)
      {
            vec3 half = normalize(view_es + light_es);
            spec = max(clamp(dot(normal, half), 0.0, 1.0), 0.0);
            spec = pow(spec,200);
      }

      // Cor final do fragmento
      FragColor =
            matAmbientColor +
            diff * matDiffuseColor +
            spec * matSpecularColor;
}
```

Listagem 7: Fragment shader para iluminação de superfície com base em deslocamento procedimental.

Os resultados obtidos com as implementações apresentadas aqui serão apresentados no Capítulo 6. Serão apontados os critérios utilizados durante os testes e feitas inferências acerca da qualidade visual e do desempenho de cada abordagem por meio de imagens, tabelas e gráficos, examinando a viabilidade de aplicação desses métodos em aplicações que requerem alto desempenho.