Implementação da técnica de Displacement Mapping em Hardware Gráfico

Daniel Menin Tortelli
Universidade Federal de Pernambuco

Marcelo Walter
Universidade Federal de Pernambuco

Resumo

A busca por maior realismo e qualidade das cenas, aumentando a sensação de imersão para o jogador, sempre foi um fator constante na história dos jogos eletrônicos. Devido a constante evolução dos dispositivos de hardware gráfico, a implementação de técnicas cada vez mais complexas e custosas computacionalmente tornou-se possível em aplicações em tempo real. *Displacement Mapping* é umas dessas técnicas que pode agora ser implementada em tempo real, devido ao advento da nova geração de hardware gráfico. O presente artigo tem por objetivo demonstrar a forma de implementação da técnica de *Displacement Mapping* no hardware gráfico atual, e como ela pode melhorar a qualidade visual dos objetos que compõe as cenas em jogos eletrônicos.

1 Introdução

A técnica de *Displacement Mapping* [Cook 1984], é uma técnica alternativa em contraste com técnicas como *Texture Mapping* [Catmull 1974] e *Bump Mapping* [Blinn 1978]. Essa técnica usa uma textura ou um *displacement map* que, aplicada sobre uma geometria base, causa um efeito onde a posição atual dos vértices dessa geometria é deslocada em alguma direção (geralmente a direção da normal da geometria), seguindo parâmetros fornecidos pela própria textura. Esse efeito proporciona à superfície um grande senso de profundidade, aumentando o detalhamento e a complexidade da geometria, permitindo sombras e silhuetas mais definidas e precisas.

Segundo Schein [Schein et al. 2005], vários esquemas de computação gráfica têm sido desenvolvidos para adicionar detalhes em superfícies de geometrias, dentre eles, *Texture Mapping* e *Bump Mapping* são os mais comuns. Quando um deles é usado para dar a impressão de uma superfície áspera, por exemplo, a técnica de *Bump Mapping* é mais eficaz. Entretanto, não ocorre nenhuma alteração na geometria do objeto, o que é facilmente exposto quando se aplicam efeitos de sombra sobre essa geometria. Ou seja, o efeito é apenas visual, não alterando a geometria do objeto. Segundo Cook, a técnica de *Displacement Mapping* é considerada mais do que uma técnica de textura, mas um tipo de modelagem de geometria.
Por vários anos, a técnica de *Displacement Mapping* ficou restrita apenas à sistemas de renderização de alto desempenho, como o *PhotoRealistic RenderMan* [Pixar 2000] da Pixar, não sendo viável sua implementação em sistemas de renderização em tempo real.
Um dos motivos dessa restrição, foi que a proposta inicial da implementação da técnica necessita de um sistema de criação de primitivas adaptativo da superfície da geometria para obter novos polígonos, cujo tamanho combinassem com o tamanho dos *pixels* da tela. Esse sistema de criação de primitivas era muito caro em termos de processamento, principalmente em aplicações em tempo real.
Felizmente, com a evolução dos dispositivos de hardware gráfico, APIs como OpenGL e Direct3D podem agora prover essa técnica cara em tempo real, criando assim novas possibilidades da sua utilização, principalmente em aplicações como jogos de computador. A figura 1 mostra a arquitetura do pipeline gráfico da nova geração de hardwares gráficos, com os seus três estágios programáveis: *Vertex Shader*, *Geometry Shader* e *Pixel Shader*.

Os estágios programáveis do pipeline permitem que um desenvolvedor crie o algoritmo usado pelo estágio por meio da escrita de *shaders*. Os *shaders* são um conjunto de comandos que aceitam recursos gráficos de entrada, executam uma série de instruções sobre esses recursos e apresentam o resultado. Os *shaders* são escritos em linguagens específicas, tais como: HLSL (*High Level Shading Language*) para Direct3D, ou, GLSL (*OpenGL Shading Language*) para OpenGL, que permitem a programação de *shaders* em nível de algoritmos. A implementação de *shaders* em uma aplicação gráfica em tempo real, permite maior flexibilidade ao programador para tratar os dados dentro do pipeline, e criar novos efeitos, aumentando a qualidade visual das cenas, proporcionando uma experiência de imersão com muito mais realismo.
A partir do modelo de shaders 3.0, um recurso denominado *Vertex Texture Fetching*, tornou possível o acesso de dados de texturas no estágio de *Vertex Shader* (antes possível apenas no estágio de *Pixel Shader*), possibilitando o uso da técnica de *Displacement Mapping* em tempo real.
Explorar todo o potencial da combinação da técnica de *Displacement Mapping* e de algoritmos de *shader* em hardware gráfico, tornou-se um campo possível, vasto e excitante, principalmente na área de desenvolvimento de jogos de computador.

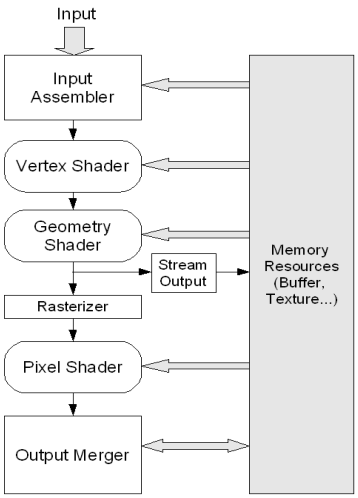

Figura 1: *Módulos do pipeline gráfico*

2 Trabalhos Relacionados

Displacement Mapping foi introduzido inicialmente por Cook e foi usada tradicionalmente em métodos baseados em Software, usando *raytracing* ou micro polígonos. Pharr e Hanrahan [Pharr and Hanrahan 1996] usaram geometrias em caching para acelerar a técnica. Wang [Wang et al. 2003] desenvolveu a *View-Dependent Displacement Mapping*. Esse método aplica *Displacement Mapping* baseado na direção da visualização da câmera. Diferentemente do método tradicional, esse método permite uma renderização de sombras e silhuetas sem incrementar complexidade na superfície base dos objetos.
Takahashi [Takahashi and Miyata 2005] descreve um modelo de deformação de superfícies base de objetos, por meio de *Displacement Mapping*, usando *vertex textures*. Também, no mesmo artigo, o autor descreve um método de cálculo de colisão que pode ser feito diretamente na GPU.
Schein [Schein et al. 2005] desenvolveu a chamada (DDM - *Deformation Displacement Maps*), que é um método de deformação de geometrias em tempo real tanto para superfícies racionais paramétricas, quanto para superfícies poligonais usando o Hardware Gráfico, sendo este um dos trabalhos mais recentes envolvendo esta técnica.
A implementação da técnica de *Displacement Mapping* proposta neste artigo segue o modelo proposto por Takahashi. Ou seja, a extração dos dados relativos as novas posições dos vértices da superfície base é feita através de um *displacement map* utilizando *Ver-*

tex Texture Fetching, dentro do estágio de *Vertex Shader*.

3 Pré-requisitos para Implementação da Técnica de Displacement Mapping

Sendo a proposta deste artigo a implementação da técnica de *Displacement Mapping* no hardware gráfico atual, realizou-se um estudo detalhado sobre a técnica de *Displacement Mapping*, com foco na sua implementação usando a nova API gráfica do DirectX (Direct3D 10) e a linguagem de escrita de shaders para essa API (HLSL), usando o modelo de shaders 4.0.
Para a implementação da técnica de *Displacement Mapping*, é necessário ter em mãos três coisas: uma **Superfície Base**, o chamado **Mapa de Deslocamento**, ou *Displacement Map* e uma **Função de Deslocamento**.
As subseções que seguem descrevem os pré-requisitos necessários para a implementação da técnica, citados acima.

3.1 Superfície Base

A superfície base pode ser definida como o objeto que sofrerá as alterações em sua geometria. Essa superfície base pode ser uma malha de polígonos planar ou mesmo um objeto que já possua uma geometria mais complexa. A figura 2 mostra as duas superfícies bases usadas na implementação da técnica. A primeira superfície corresponde a uma esfera e a segunda a uma malha plana texturizada.

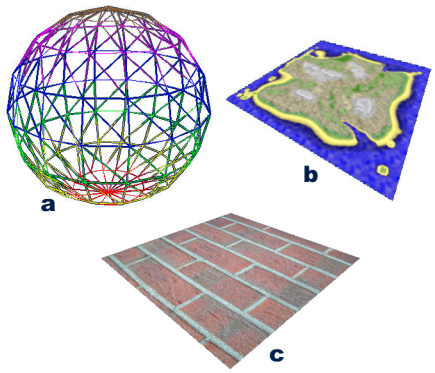

Figura 2: *Superfícies Base*

3.2 Mapa de Deslocamento (*Displacement Map*)

O mapa de deslocamento consiste de uma imagem em tons de cinza, como mostra a figura 3. Os valores da intensidade de cada pixel que compõe o *displacement map* são usados pela função de deslocamento para calcular o fator de deslocamento dos vértices da superfície base. Esses valores de intensidade geralmente variam dentro do intervalo de 0 a 255, onde 0 corresponde a cor preta e 255 corresponde a cor branca. Quanto maior o valor da intensidade, maior será o fator de deslocamento do vértice.

3.3 A Função de Deslocamento

O objetivo da função de deslocamento é deslocar a posição dos vértices da superfície base, de acordo com os valores de intensidade extraídos do mapa de deslocamento (*displacement map*). Geralmente, o deslocamento dos vértices é feito na direção da coordenada normal do próprio vértice. A figura 4 ilustra a função de deslocamento e sua relação com o vértice e a sua coordenada normal, usada no deslocamento.

Onde **P0** é a posição original do vértice na superfície base, **P1** é a posição do vértice depois do deslocamento, **N** é o vetor normal do vértice, **fd** é o fator de deslocamento (extraído do *displacement map*) e **fu** um fator de escala definido pelo usuário.

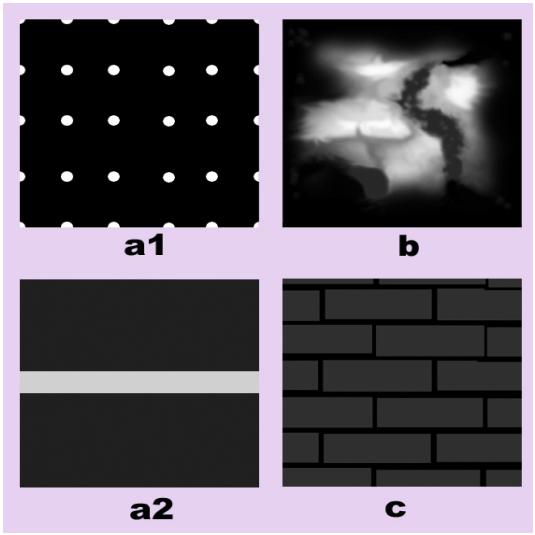

Figura 3: *Mapas de Deslocamento (Displacement Maps)*

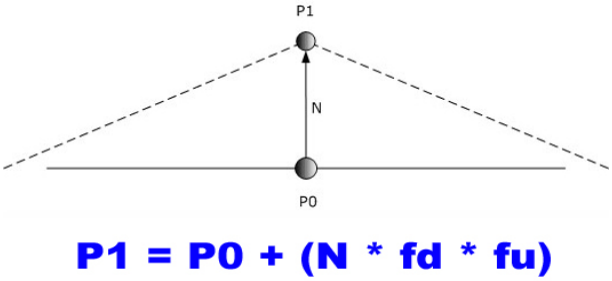

Figura 4: *Função de Deslocamento (Displacement Mapping Function)*

4 A Implementação da Técnica

Com os pré-requisitos em mãos, o procedimento para criar a deformação da superfície base, usando *Displacement Mapping*, se torna um procedimento simples.
Primeiramente a superfície base deve ser carregada (em tempo de inicialização) pela aplicação, e passada para o pipeline como um *buffer* de vértices com seus respectivos índices. É de fundamental importância que os vértices sejam passados para o pipeline com 3 propriedades: posição, vetor normal e coordenadas de textura.
O(s) mapa(s) de deslocamento *(displacement maps)* também deve(em) ser previamente carregado(s) e referenciado(s), através de variáveis especiais de comunicação entre a aplicação e os shaders.
Ao entrarem no estágio de *Vertex Shader*, os vértices tem suas posições deslocadas, em tempo de execução, através da função de deslocamento. Antes disso, entretanto, o fator de deslocamento para cada vértice é extraído do *displacement map*, via *Vertex Texture Fetching*.
Depois de concluídas as alterações, os vértices passam pela transformação de coordenadas locais para coordenadas de Universo, Visão e Projeção. Isso acontece também com seus respectivos vetores normais e as coordenadas de textura são apenas repassadas para o próximo estágio do pipeline.
A figura 8 exibe o trecho de código em HLSL para o estágio de *Vertex Shader*, demonstrando o procedimento descrito acima.

Na linha 1, o parâmetro de entrada da função **VS**, que executa o código HLSL no estágio de *Vertex Shader*, é uma estrutura denominada *VSINPUT*. Essa estrutura é composta de dados configurados para cada vértice da superfície base, na aplicação. São eles: a posições locais **x,y,z** de cada vértice, a sua coordenada Normal e as coordenadas de textura **u,v**.
Na linha 2 é inicializada a estrutura de retorno da função de *Vertex Shader*, que será repassada para o estágio seguinte do pipeline gráfico, depois de devidamente configurada.

As linhas 3 e 4 tratam da aplicação da técnica de *Vertex Texture Feching*, onde é extraído o valor de um determinado *texel* do mapa de deslocamento, armazenado-o como fator de deslocamento na variável **fd**, usada posteriormente pela função de deslocamento.
Nas linhas 5 e 6 são extraídos os outros dois elementos necessários para o cálculo da função de deslocamento. São eles: a posição do vértice da superfície base que será deslocado e a sua coordenada normal. Esses dados são armazenados nas variáveis **P0** e **N** respectivamente.
Com todas as variáveis devidamente configuradas, é aplicada na linha 7, a função de deslocamento. Essa função, como descrita na seção 4.3, faz o deslocamento da posição do vértice que está sendo processado atualmente no estágio de *Vertex Shader*. A variável **fu** da função de deslocamento é uma variável atualizada na aplicação pelo usuário. Ela é repassada para dentro do *Vertex Shader* através do uso de funções especiais de HLSL.
Depois de ter sua posição modificada, cada vértice precisa ser transformado de coordenadas espaciais para coordenadas de universo, visão e projeção. Esse procedimento, bem como a transformação da coordenada normal do vértice, é feito da linha 8 a 11.
Como a textura da superfície base não é usada no cálculo do *Displacement Mapping*, ela apenas passa para o próximo estágio do pipeline sem sofrer alterações no estágio de *Vertex Shader*. Isso é feito na linha 12.
Finalmente, na linha 13, a estrutura de saída com os valores transformados para o vértice é passada como retorno da função de *Vertex Shader* para os próximos estágios do pipeline.

5 Resultados Obtidos

Aplicando *Displacement Mapping* nas superfícies bases mostradas na figura 2, com seus respectivos mapas de deslocamento *(displacement maps)* mostrados na figura 3 e utilizando a função de deslocamento mostrada na figura 4, os resultados obtidos são mostrados na figura 5. As figuras 6 e 7 mostram o resultado da superfície base da figura 2, **b** e **c**, pela modificação em tempo de execução da variável de controle do usuário **fu**. O Hardware utilizado na implementação foi um *Pentium Core Duo* 2.80 GHz, 2046 MB RAM, NVidia GeForce 8500 GT de 256 MB, rodando no sistema Operacional *Windows Vista*, necessário para a implementação com *DirectX* 10 e *shader model* 4.

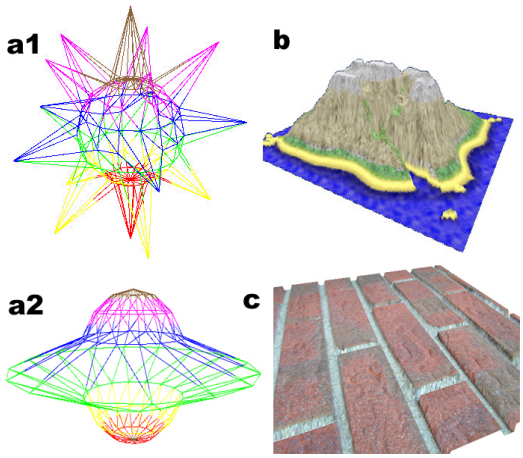

Figura 5: *Superfícies base depois do Displacement Mapping*

6 Conclusões e Trabalhos Futuros

A implementação de *Displacement Mapping* evidencia suas vantagens em relação a outras técnicas de mapeamento, como *Bump Mapping,* no sentido de adicionar detalhes físicos à geometria da superfície base, criando silhuetas e detalhes reais.
Como a deformação pode ser feita em tempo real, pode-se criar diversas formas para uma superfície base apenas modificando o mapa de deslocamento. Essa característica é muito útil e pode ser usada para descartar a necessidade de diversos modelos para representar o mesmo objeto, evitando o uso de *LODs (Level Of Detail)*.
Entretanto, deve-se tomar certos cuidados na aplicação da técnica, principalmente no grau de deformação desejado. Se o grau de deformação for muito elevado, pode ocorrer perda de definição da geometria deformada, criando faces poligonais indesejadas.

Umas das melhorias para a implementação da técnica de *Displacement Mapping* em hardware gráfico, evitando o problema de deformação excessiva da superfície base, seria a utilização do estágio programável de *Geometry Shader*.
Na nova arquitetura do pipeline gráfico, como mostra a figura 2, a função do estágio programável de *Geometry Shader* é a criação de novas primitivas (triângulos) com base nos vértices transformados que partem do estágio de *Vertex Shader*. A criação de novos triângulos pode evitar a deformação excessiva da malha de polígonos da superfície base. A deformação ocorre quando não há um limite previamente definido para o fator de deslocamento, aplicado a um determinado vértice na hora em que ele for deslocado. Sem esse limite, o triangulo pode esticar demais, deformando não só a superfície base, como também as coordenadas de textura da mesma. Dessa forma, quando o grau de deformação atingir um certo limiar, novos triângulos podem ser criados para preservar a integridade da forma da geometria da superfície base, e preservar a texturização correta da mesma.

Referências

BLINN, J. F. 1978. Simulation of wrinkled surfaces. *Proceedings of SIGGRAPH 78, Comput. Graph. 12* (August), 286 292.

BLYTHE, D. 2006. The direct3d 10 system. *ACM SIGGRAPH 2006*, 724 – 734.

CATMULL, E. E. 1974. A subdivision algorithm for computer display of curved surfaces. *Ph.D. thesis, Department of Computer Science, University of Utah, Salt Lake City, UT*.

COOK, R. L. 1984. Shade trees. *Proceedings of SIGGRAPH 84, Comput. Graph. 18* (July), 223 – 231.

ELBER, G. 2002. Geometric deformation-displacement maps. *In: 10th Pacific Graphics*, 156 – 165.

ENGEL, W. 2004. *ShaderX 3: Advanced Rendering with DirectX and OpenGL*. Charles River Media.

ENGEL, W. 2006. *ShaderX4: Advanced Rendering Techniques*. Charles River Media.

GERASIMOV, P., FERNANDO, R., AND GREEN, S., 2004. Shader model 3.0: using vertex texture. nvidia corporation, white paper. `download.nvidia.com/.../Direct3D9/src/VertexTextureFetchWater/docs/VertexTextureFetchWater_UserGuide.pdf`.

HIRCHE, J., EHLERT, A., GUTHE, S., AND DOGGETT, M. 2004. Hardware accelerated per-pixel displacement mapping. *Proceedings of Graphics Interface 2004*, 153 – 158.

LUNA, F. D. 2003. *Introduction to 3D Game Programming with DirectX 9*. Wordware Publishing Inc.

PATTERSON, J. W., HOGGAR, S. G., AND LOGIE, J. R. 1991. Inverse displacement mapping. *Computer Graphics Forum 10*, 129 – 139.

PEDERSEN, H. K. 1994. Displacement mapping using flow fields. *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, 279 – 286.

PHARR, M., AND HANRAHAN, P. 1996. Geometry caching for ray-tracing displacement maps. *7th Eurographics Rendering Workshop* (June), 31 – 40.

PHARR, M. 2005. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley Professional.

PIXAR. 2000. The renderman interface version 3.2. *Pixar, Inc.*.

ROST, R. J. 2006. *OpenGL Shading Language 2nd Edition*. Addison-Wesley Professional.

SCHEIN, S., KARPEN, E., AND ELBER, G. 2005. Real-time geometric deformation displacementmaps using programmable hardware. *The Visual Computer* (September), 791 – 800.

ST-LAURENT, S. 2004. *The COMPLETE Effect and HLSL Guide*. Paradoxal Press.

ST-LAURENT, S. 2004. *Shaders for Game Programmers and Artists*. Course Technology PTR.

STROUSTRUP, B. 1997. *The C++ Programming Language*. AT and T Labs.

TAKAHASHI, M., AND MIYATA, K. 2005. Gpu based interactive displacement mapping. *International Workshop on Advanced Image Technology 2005*, 105 – 108.

WANG, L., WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2003. View-dependent displacement mapping. *ACM SIGGRAPH 2003*, 334 – 339.

WANG, X., TONG, X., LIN, S., HU, S., GUO, B., AND SHUM, H.-Y. 2004. Generalized displacement maps. *Eurographics Symposium on Rendering*.

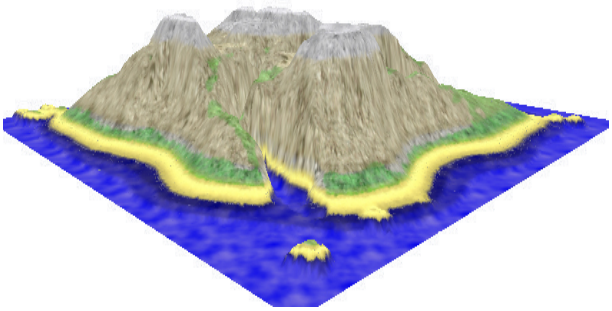

Figura 6: *Ilha*

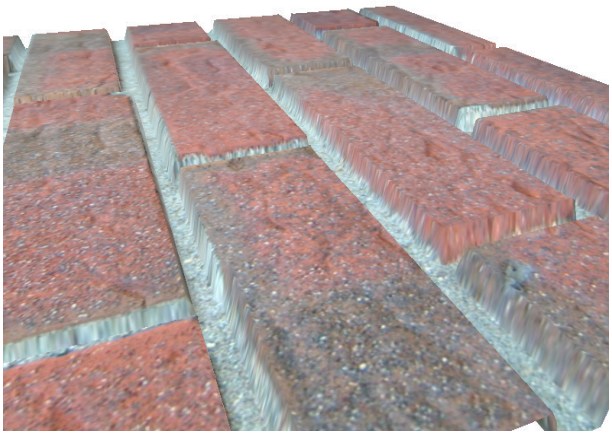

Figura 7: *Muro*

```
//--------------------------------------------------------------------------------------
// Constant Buffer Variables
//--------------------------------------------------------------------------------------
Texture2D g_txDiffuse;
Texture2D txDispMap;

SamplerState samPoint
{
   Filter = MIN_MAG_MIP_POINT;
   AddressU = Clamp;
   AddressV = Clamp;
};

SamplerState samLinear
{
   Filter = MIN_MAG_MIP_LINEAR;
   AddressU = Wrap;
   AddressV = Wrap;
};


cbuffer cbChangesOnce
{
   matrix View;
};

cbuffer cbChangeOnResize
{
   matrix Projection;
};

cbuffer cbChangesEveryFrame
{
   matrix World;
   float fu;
};


struct VS_INPUT                    // estrutura de entrada para o Vertex Shader
{
   float3 Pos       : POSITION;    // posição
   float3 Norm      : NORMAL;      // normal
   float2 Tex       : TEXCOORD0;   // coordenadas de textura
};

struct PS_INPUT                    // estrutura de entrada para o Pixel Shader
{
   float4 Pos : SV_POSITION;
   float3 Norm : TEXCOORD0;
   float2 Tex : TEXCOORD1;
};


//--------------------------------------------------------------------------------------
// Vertex Shader
//--------------------------------------------------------------------------------------
1  PS_INPUT VS( VS_INPUT input )
   {
2     PS_INPUT output = (PS_INPUT)0;

3     float fd = 0.0f;
4     fd = txDispMap.SampleLevel( samPoint, float2(input.Tex.xy), 0.0f );

5     float3 N = input.Norm;
6     float4 P0 = float4(input.Pos, 1.0f);

7     float4 P1 = float4(N * fd * fu, 0.0f) + P0;

8     output.Pos = mul( P1, World );
9     output.Pos = mul( output.Pos, View );
10    output.Pos = mul( output.Pos, Projection );

11    output.Norm = mul( input.Norm, World );

12    output.Tex = input.Tex;

13    return output;
   }


//--------------------------------------------------------------------------------------
// Pixel Shader
//--------------------------------------------------------------------------------------
float4 PS( PS_INPUT input) : SV_Target
{
   float4 outputColor = g_txDiffuse.Sample( samLinear, input.Tex );
   outputColor.a = 1.0f;

   return outputColor;
}


//--------------------------------------------------------------------------------------
// Tecnica D3D10 shader model 4.0. Uma única passada...
//--------------------------------------------------------------------------------------
technique10 Render
{
   pass P0
   {
      SetVertexShader( CompileShader( vs_4_0, VS() ) );
      SetGeometryShader( NULL );
      SetPixelShader( CompileShader( ps_4_0, PS() ) );
   }
}
```

Figura 8: *Código em HLSL para Displacement Mapping*