CG – Normal Mapping

COMPUTAÇÃO GRÁFICA

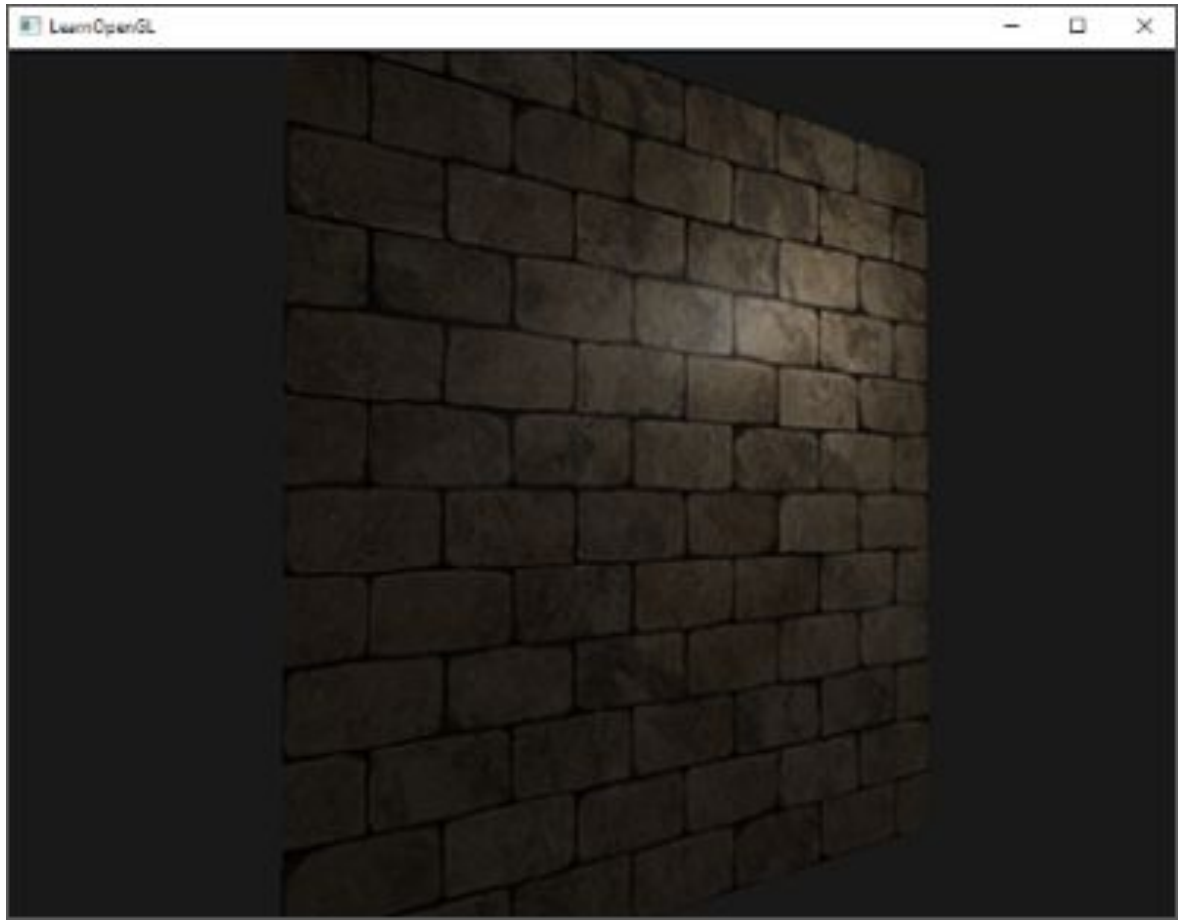
Normal Mapping

Todas as nossas cenas são cheias de malhas, cada uma consistindo de centenas ou talvez milhares de triângulos. Aumentamos o realismo envolvendo texturas 2D nesses triângulos planos, escondendo o fato de que os polígonos são apenas pequenos triângulos planos. As texturas ajudam, mas quando você dá uma boa olhada nas malhas, ainda é muito fácil ver as superfícies planas subjacentes. A maioria das superfícies da vida real não são planas e exibem muitos detalhes (esburacados).

Normal Mapping

Por exemplo, pegue uma superfície de tijolos. Uma superfície de tijolos é uma superfície bastante áspera e obviamente não completamente plana: contém listras de cimento afundadas e muitos pequenos orifícios e rachaduras detalhados. Se vermos tal superfície de tijolos em uma cena iluminada, a imersão é facilmente quebrada. A seguir podemos ver uma textura de tijolo aplicada a uma superfície plana iluminada por uma luz pontual.

Normal Mapping



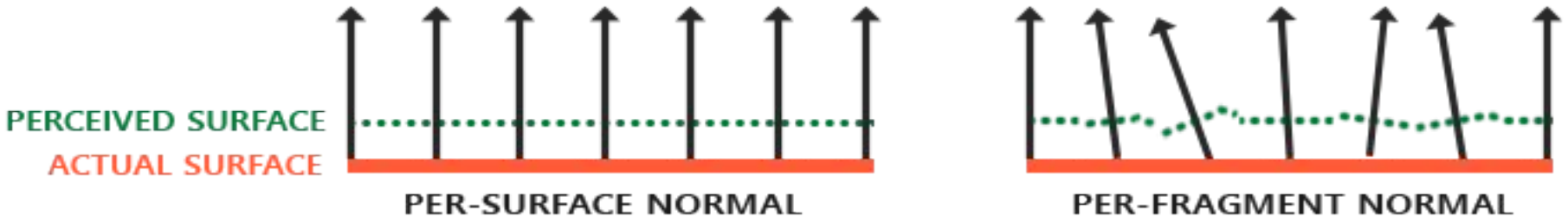
Normal Mapping

A imagem ignora completamente as listras profundas entre os tijolos; a superfície parece perfeitamente plana. Podemos corrigir parcialmente a aparência plana usando um mapa especular para fingir que algumas superfícies estão menos iluminadas devido à profundidade ou outros detalhes, mas isso é mais um truque do que uma solução real. O que precisamos é de alguma forma de informar o sistema de iluminação sobre todos os pequenos detalhes de profundidade da superfície.

Normal Mapping

Se pensarmos sobre isso da perspectiva de uma luz: como é que a superfície é iluminada como uma superfície completamente plana? A resposta é o vetor normal da superfície. Do ponto de vista da técnica de iluminação, a única maneira de determinar a forma de um objeto é pelo seu vetor normal perpendicular. A superfície do tijolo tem apenas um único vetor normal e, como resultado, a superfície é uniformemente iluminada com base na direção desse vetor normal. E se nós, em vez de uma normal por superfície que é a mesma para cada fragmento, usarmos uma normal por fragmento que é diferente para cada fragmento? Desta forma podemos desviar um pouco o vetor normal com base nos pequenos detalhes de uma superfície; isso dá a ilusão de que a superfície é muito mais complexa:

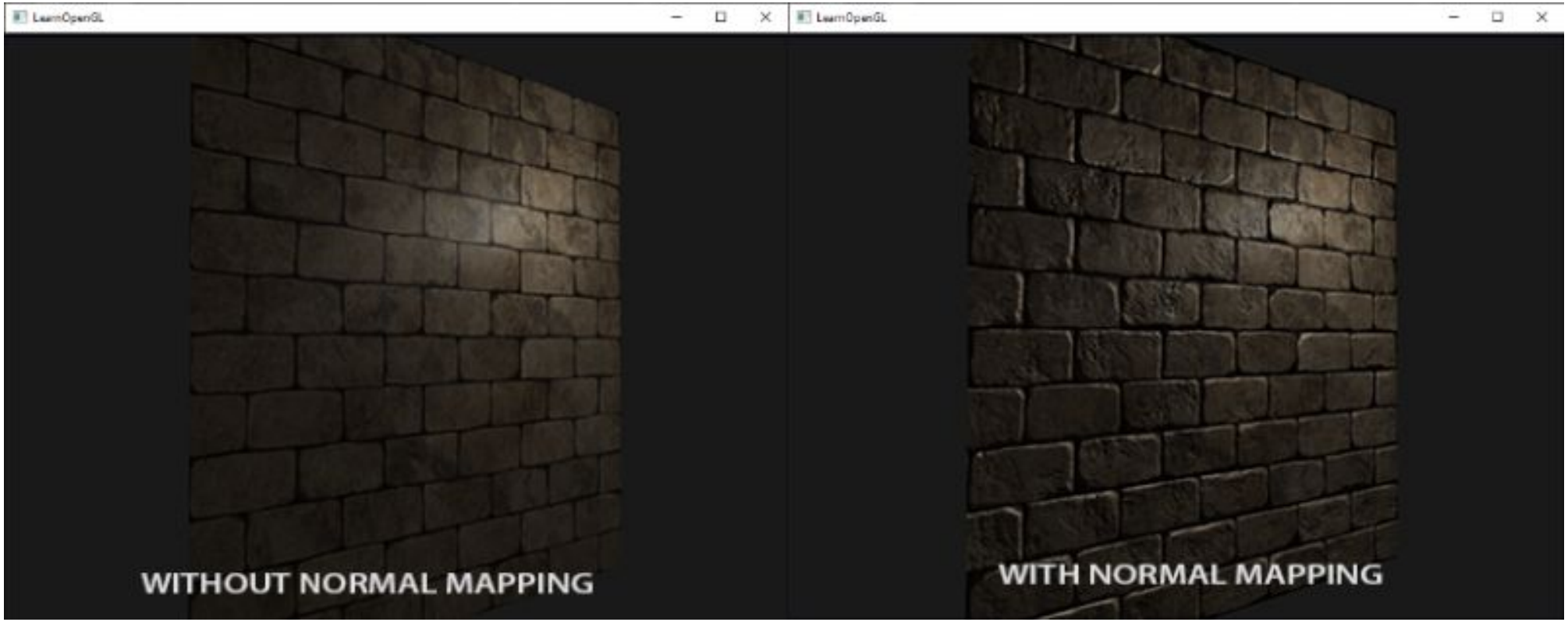
Normal Mapping



Normal Mapping

Usando normais por fragmento, podemos enganar a iluminação para acreditar que uma superfície consiste em pequenos planos (perpendiculares aos vetores normais), dando à superfície um enorme aumento de detalhes. Essa técnica para usar normais por fragmento em comparação com normais por superfície é chamada de mapeamento normal ou normal mapping. Aplicado ao plano de tijolos, parece um pouco com isso:

Normal Mapping



Normal Mapping

Como podemos ver, dá um enorme aumento em detalhes e por um custo relativamente baixo. Como alteramos apenas os vetores normais por fragmento, não há necessidade de alterar a equação de iluminação. Passamos agora uma normal por fragmento, em vez de uma normal de superfície interpolada, para o algoritmo de iluminação. A iluminação então faz o resto.

Aplicação

Para fazer o mapeamento normal funcionar, vamos precisar de uma normal por fragmento. Podemos usar uma textura 2D para armazenar dados normais por fragmento. Dessa forma, podemos amostrar uma textura 2D para obter um vetor normal para esse fragmento específico.

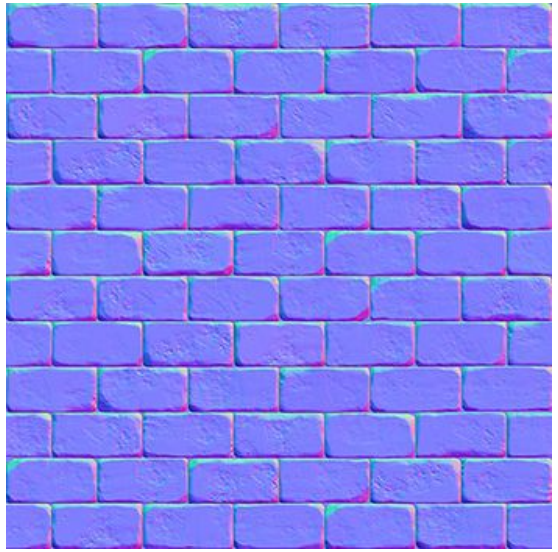
Embora os vetores normais sejam entidades geométricas e as texturas geralmente sejam usadas apenas para informações de cores, o armazenamento de vetores normais em uma textura pode não ser imediatamente óbvio. Se você pensar em vetores de cor em uma textura, eles são representados como um vetor 3D com um componente r, g e b. Da mesma forma, podemos armazenar os componentes x, y e z de um vetor normal nos respectivos componentes de cor. Os vetores normais variam entre -1 e 1, então eles são mapeados primeiro para [0,1]:

Aplicação

```
vec3 rgb_normal = normal * 0.5 + 0.5;
// transforma de [-1,1] para [0,1]
```

Aplicação

Com vetores normais transformados em um componente de cor RGB como este, podemos armazenar um fragmento normal derivado da forma de uma superfície em uma textura 2D. Um exemplo do mapa normal da superfície do tijolo fica assim:



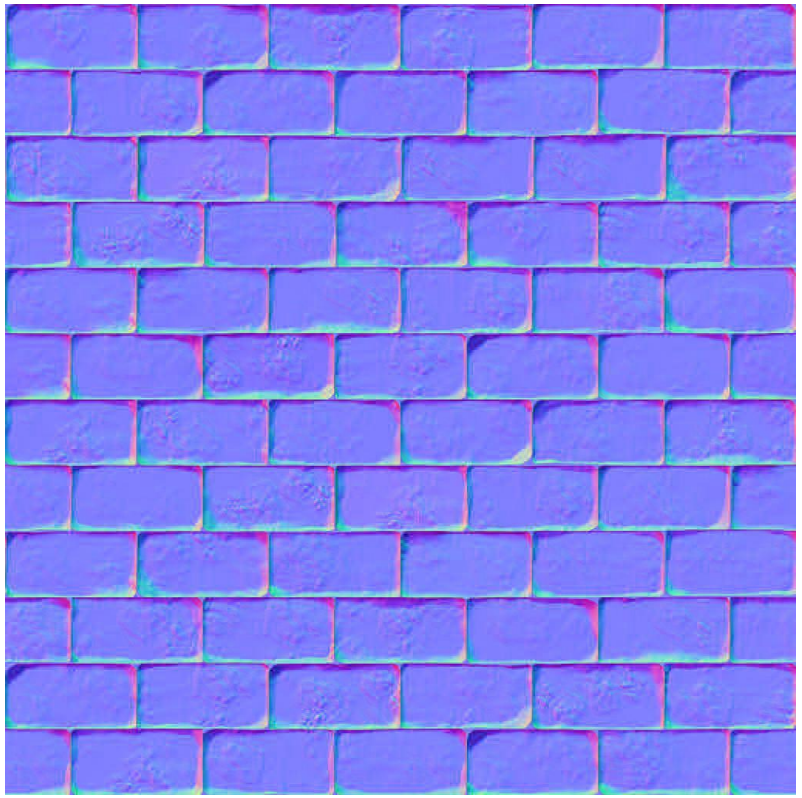
Aplicação

Esse (e quase todos os mapas normais que você encontrar online) terão um tom azulado. Isso ocorre porque as normais estão todas apontando para fora em direção ao eixo z positivo (0,0,1): uma cor azulada. Os desvios na cor representam vetores normais que são ligeiramente deslocados da direção z positiva geral, dando uma sensação de profundidade à textura. Por exemplo, você pode ver que no topo de cada tijolo a cor tende a ser mais esverdeada, o que faz sentido já que o lado superior de um tijolo teria normal apontando mais no sentido positivo e (0,1,0) o que acontece ser a cor verde!

Aplicação

Com um plano simples, olhando para o eixo z positivo, podemos pegar uma textura difusa e um mapa normal para renderizar uma imagem. Cuidado só que no OpenGL as coordenadas de textura y (ou v) são invertidas de como as texturas são geralmente criadas. O mapa normal vinculado, portanto, tem seu componente y (ou verde) invertido; se você não levar isso em consideração, a iluminação será incorreta.

Aplicação



Carregar Texturas

Mas antes de fazer o Normal Mapping, precisamos aprender a carregar texturas normais por shaders. Para isso, vamos aproveitar o código de exemplo de carregamento de bmp e utilizar os bmps do material didático.

Vamos aproveitar as partes de carregamento e ativação das texturas do código antigo, mas vamos continuar o código novo, fazendo algumas alterações.

Carregar Texturas

Primeiro vamos mudar nosso vetor de vértices, além da cor e posição, vamos colocar as informações de textura, também vamos atualizar para um quadrado.

Carregar Texturas

```
float vertices[] =
{ // posicoes          //cores              //coordenadas textura
0.5f, 0.5f, 0.0f,     1.0f, 0.0f, 0.0f,      1.0f, 1.0f,
0.5f, -0.5f, 0.0f,    0.0f, 1.0f, 0.0f,      1.0f, 0.0f,
-0.5f, -0.5f, 0.0f,   0.0f, 0.0f, 1.0f,      0.0f, 0.0f,
-0.5f, 0.5f, 0.0f,    1.0f, 1.0f, 0.0f,      0.0f, 1.0f
};


glVertexAttribPointer(2, 2, GL_FLOAT, GL_FALSE, 8 * sizeof(float),
(void*)(6 * sizeof(float)));
glEnableVertexAttribArray(2);
```

Carregar Texturas

Depois vamos modificar os shaders, o vertex shader tem como entrada as informações de coordenadas de textura para cada vértice e como saída ele vai ter que repassar isso para o fragment shader.

O fragment shader por sua vez, vai pegar as informações de coordenadas do vertex shader, e a textura por um uniform e aplicar isso para cada fragmento com a função texture.

Desde que a textura que carregamos seja a última coisa no bindtexture, ela será carregada para o modelo.

Carregar Texturas

```
#version 330 core


layout (location = 0) in vec3 aPos;

layout (location = 1) in vec3 aCor;

layout (location = 2) in vec2 aTexCoord;

out vec3 Cor;

out vec2 TexCoord;

void main()

{

    gl_Position = vec4(aPos, 1.0);

    Cor = aCor;

    TexCoord = aTexCoord;

}
```

Carregar Texturas

```
#version 330 core

uniform sampler2D textura;


out vec4 fragCor;


in vec3 Cor;

in vec2 TexCoord;


void main()

{

    fragCor = texture(textura, TexCoord) * vec4(Cor.xyzx);

}
```

Iluminação - Ambiente

Além disso, precisamos fazer a iluminação por shaders, hoje vamos ver duas delas: A ambiente e a difusa.

A luz ambiente é bem simples de se resolver, precisamos somente de uma constante multiplicada por cada fragmento que determina a intensidade da cor, junto com uma possível cor.

Iluminação

```
float intAmbiente = 0.1;
vec3 luzAmbiente = intAmbiente * corLuz;
vec3 resultado = luzAmbiente * <cor_fragmento>;
FragColor = vec4(resultado, 1.0);
```

Iluminação - Difusa

A difusa é um pouco mais complicada, precisamos trabalhar com as normais, então precisamos primeiro adiciona-las como variáveis no nosso vetor de vértices, junto com a posição cores e texturas.

Iluminação - Difusa

```
layout (location = 3) in vec3 aNormal;


glVertexAttribPointer(3, 3, GL_FLOAT, GL_FALSE, 11 *
sizeof(float), (void*)(8 * sizeof(float)));
glEnableVertexAttribArray(3);
```

Iluminação - Difusa

Depois precisamos passar essas normais do vertex para o fragmente shader, junto com a normal precisamos passar a posição de cada fragmento, isso cria duas variáveis de saída do vertex shader e de entrada para o fragment shader.

Além disso, podemos criar uma variável uniform para o posicionamento da luz.

Iluminação - Difusa

```
out vec3 FragPos;
out vec3 Normal;


in vec3 FragPos;
in vec3 Normal;
uniform vec3 posLuz;
```

Iluminação - Difusa

A partir disso podemos fazer os cálculos necessários para criar o efeito de luz difusa junto com a luz ambiente.

Iluminação - Difusa

```
vec3 norm = normalize(Normal);
vec3 dirLuz = normalize(posLuz - FragPos);

float diff = max(dot(norm, dirLuz), 0.0);
vec3 difusa = diff * luzCor;

vec3 resultado = (ambiente + difusa) * <cor_fragmento>;
FragColor = vec4(resultado, 1.0);
```