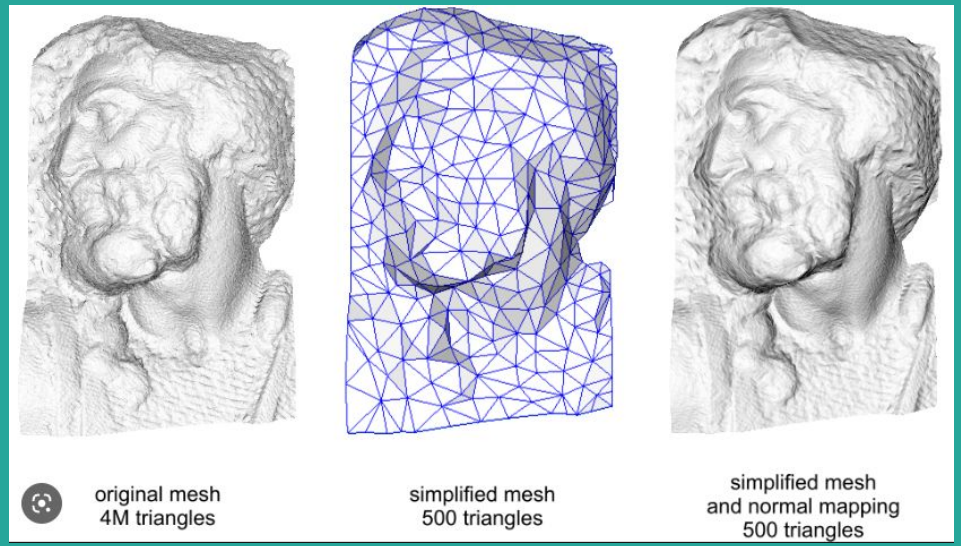
Normal Mapping

Carreon, Bruno
Recalde, Rene

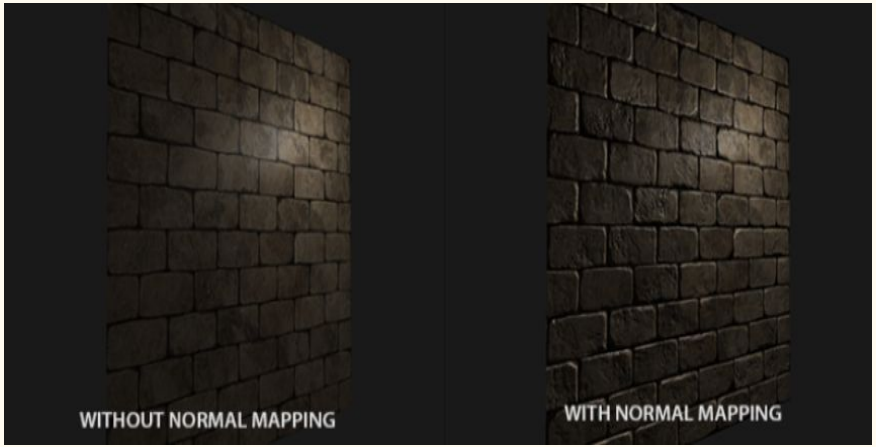
¿Qué es y cuándo se usa?

Normal mapping es una técnica 3D que permite dar una iluminación y relieve mucho más detallado a la superficie de un objeto



Usos

1. Cuando queremos agregar realismo.
2. Cuando queremos agregar niveles de detalle que son muy costosos utilizando técnicas geométricas.



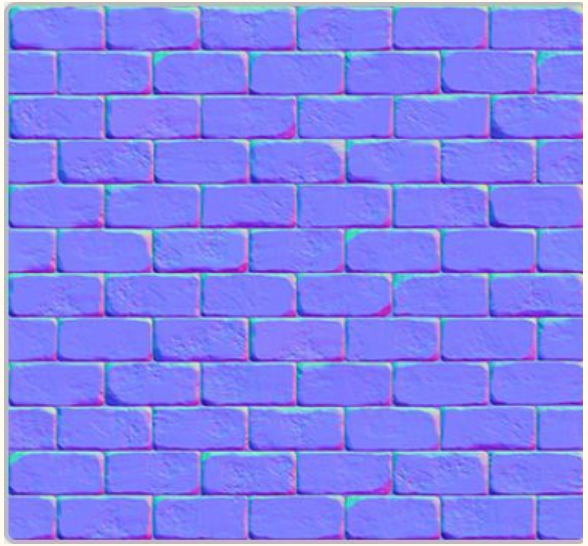
Normal Mapping

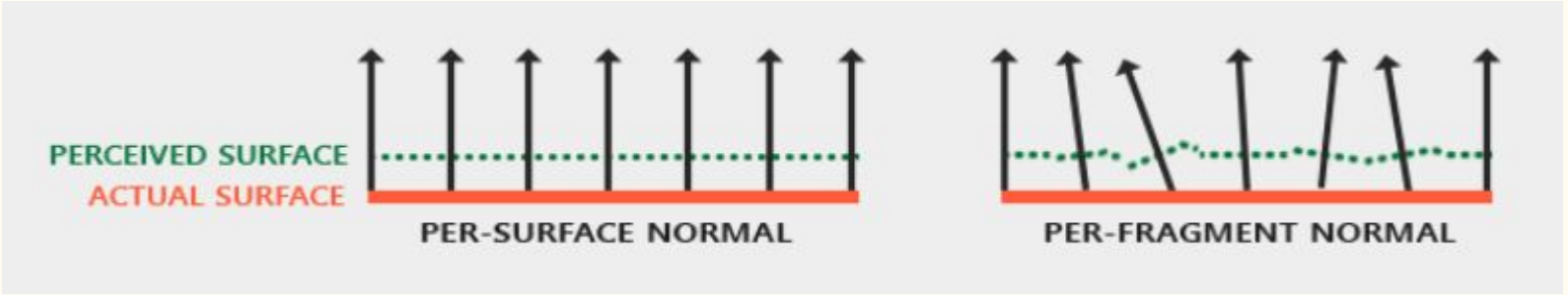
Ladrillos

1. La importancia del vector normal para el caso de superficies planas.

2. ¿Cómo se forma la textura de normales?

3. Vector -> (X, Y, Z) = Color -> (R,G,B)





Pros

Normal mapping

- No es necesario modificar la geometría para añadir realismo
- Creado y customizado por artistas
- Muy rápido
- Más flexibilidad y nivel de detalles

Contras

Normal mapping

- Memoria extra para texturas
- Memoria extra para datos de fragment y vertex shader
- Todos los detalles son planos.
- Puede ser usado para detalles de medio a bajo nivel.
- La performance baja para todos los objetos. Aunque no usen el normal mapping

Implementación

Caso simple

Las normales apuntan al z+

En el main mandamos las texturas

```
shader_texture = Shader ("shaders/texture");
shader_phong = Shader("shaders/phong");

// cargamos las nuevas texturas (la comun y la que tiene las normales)
// --------------------
Texture textura("models/floor.png"); // comun
Texture texturaNormal("models/floor_normal.png"); // con normales
textura.bind(4);
texturaNormal.bind(5);

// le pasamos las texturas al shader_texture
// --------------------
shader_texture.use();

shader_texture.setUniform("colorTexture", 4);
shader_texture.setUniform("myNormal", 5);
```

Las recibimos en el shader

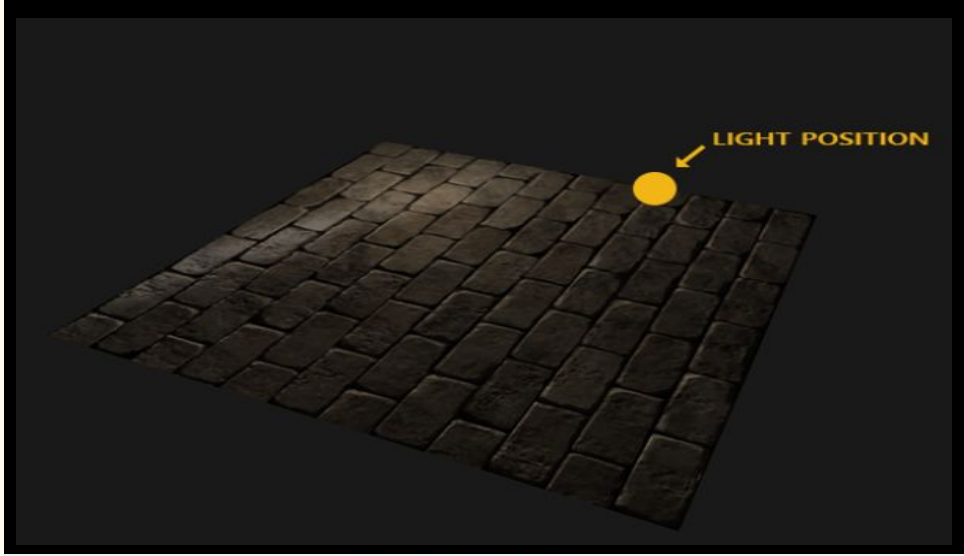
```
out vec4 fragColor;
out vec3 normal; //esta la retorno para usarla en phong

uniform sampler2D myNormal; // obtengo la textura con normales
uniform sampler2D colorTexture; // obtengo la textura
uniform float bump;

#include "funcs/calcPhong.frag"

void main() {
    // obtener normal del mapa normal en el rango [0,1]
    vec3 normal = texture(myNormal, fragTexCoords).rgb;

    // transformar vector normal a rango [-1,1]
    normal = normalize(normal * 2.0 - 1.0);

    if(bump < 1.f) { normal = fragNormal; }

    vec3 tex = texture(colorTexture,fragTexCoords).rgb;

    vec3 phong = calcPhong(lightVSPosition, lightColor,
                           mix(ambientColor,ambientColor*vec3(tex),1.f),
                           mix(diffuseColor,diffuseColor*vec3(tex),1.f),
                           specularColor, shininess,normal);
    fragColor = vec4(phong,opacity);
}
```

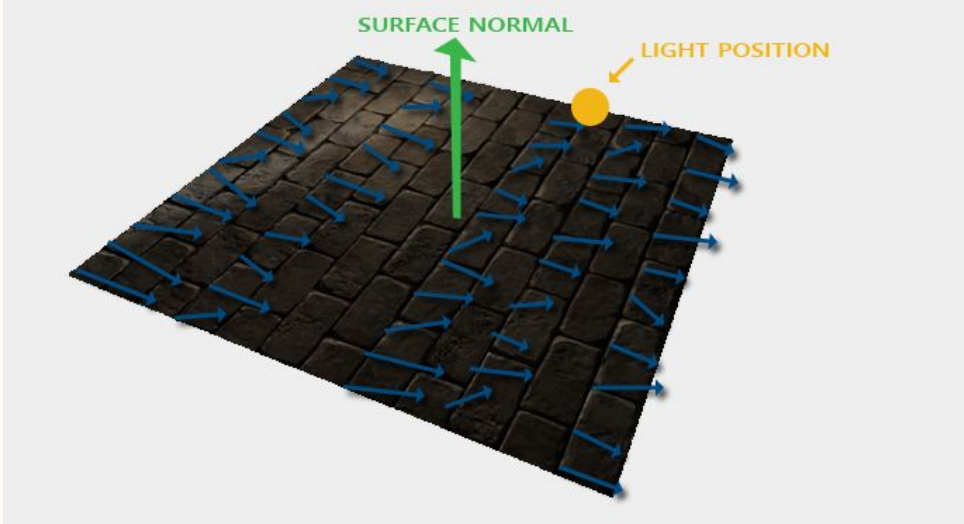
Aplicamos phong con las nuevas normales

```
vec3 calcPhong(vec4 lightPosition, vec3 lightColor,
               vec3 ambientColor, vec3 diffuseColor,
               vec3 specularColor, float shininess,vec3 normalTrucha)
{
    // ambiente
    vec3 ambient = ambientColor * lightColor * ambientStrength;

    // difusa
    vec3 viewDir = normalize(-fragPosition);
    vec3 norm = normalize(dot(viewDir,normalTrucha)<0? -normalTrucha : normalTrucha);
    vec3 lightDir = normalize(vec3(lightPosition)-fragPosition);
    vec3 diffuse = diffuseColor * max(dot(norm,lightDir),0.f) * lightColor;

    // especular
//  vec3 viewDir = normalize(-fragPosition);
    vec3 halfw = normalize(viewDir+lightDir);
    vec3 specular = specularColor * pow(max(dot(norm,halfw),0.f),shininess) * lightColor;

    // result
    return ambient+diffuse+specular;
}
```

Caso complejo

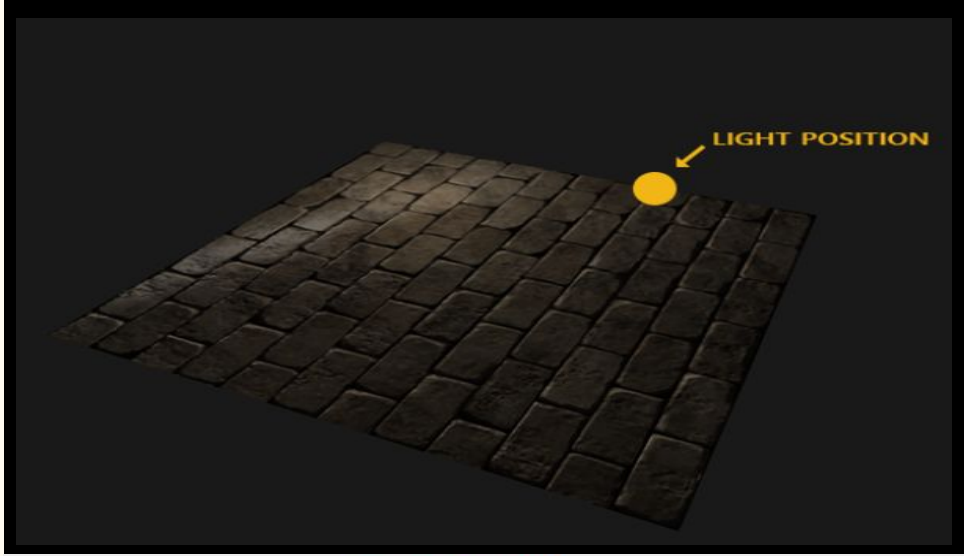
Las normales no apuntan hacia el z+

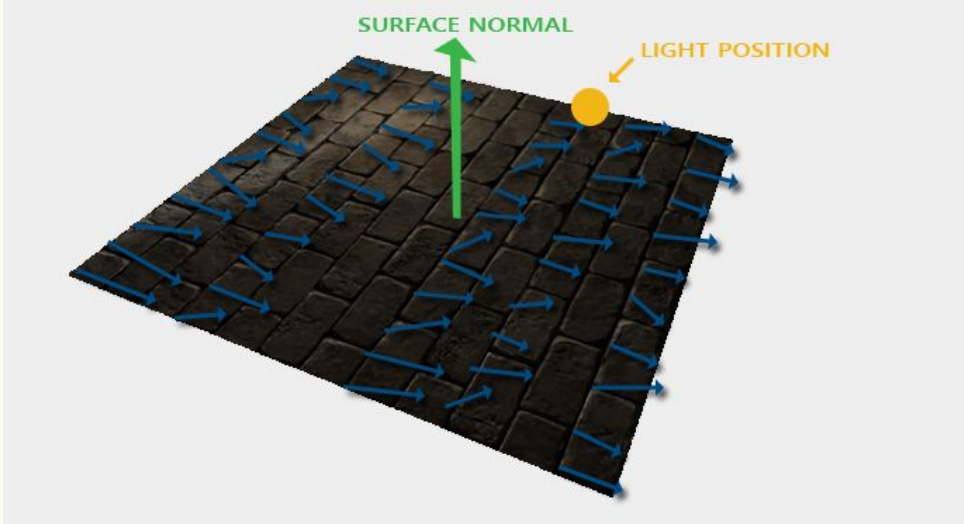




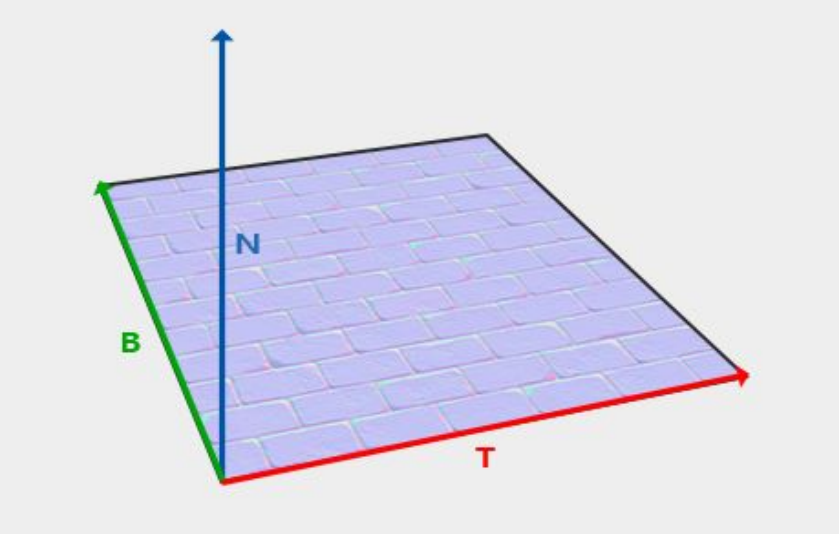
Caso complejo

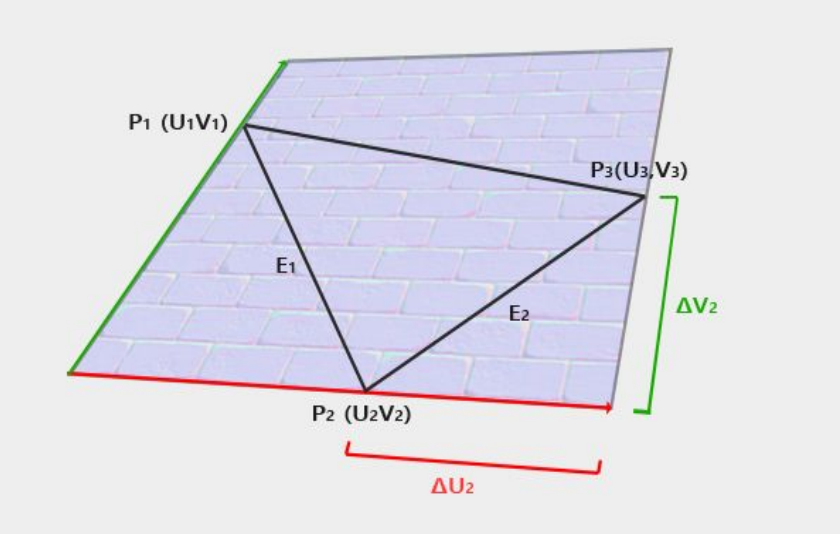
Las normales no apuntan hacia el z+





Matriz tangente, bitangente y normal





Cálculo de la matriz TBN

$$E_1 = \Delta U_1 T + \Delta V_1 B$$
$$E_2 = \Delta U_2 T + \Delta V_2 B$$

$$(E_{1x}, E_{1y}, E_{1z}) = \Delta U_1 (T_x, T_y, T_z) + \Delta V_1 (B_x, B_y, B_z)$$
$$(E_{2x}, E_{2y}, E_{2z}) = \Delta U_2 (T_x, T_y, T_z) + \Delta V_2 (B_x, B_y, B_z)$$

$$\begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix} \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

$$\begin{bmatrix} \Delta U_1 & \Delta V_1 \\ \Delta U_2 & \Delta V_2 \end{bmatrix}^{-1} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix} = \begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix}$$

$$\begin{bmatrix} T_x & T_y & T_z \\ B_x & B_y & B_z \end{bmatrix} = \frac{1}{\Delta U_1 \Delta V_2 - \Delta U_2 \Delta V_1} \begin{bmatrix} \Delta V_2 & -\Delta V_1 \\ -\Delta U_2 & \Delta U_1 \end{bmatrix} \begin{bmatrix} E_{1x} & E_{1y} & E_{1z} \\ E_{2x} & E_{2y} & E_{2z} \end{bmatrix}$$

Optimización

Cuando los vectores tangentes son calculados para mayas grandes con una gran cantidad de vértices el vector tangente es promediado obteniendo un buen resultado(suave).

El problema de este enfoque es que los tres vectores TBN podrian terminar no siendo perpendiculares. Por lo tanto TBN no sería ortogonal.

Este problema se resuelve usando un metodo llamado Gram-Schmit, para re-orthogonalizar.

Optimización

```
vec3 T = normalize(vec3(model * vec4(aTangent, 0.0)));

vec3 N = normalize(vec3(model * vec4(aNormal, 0.0)));

// re-orthogonalize T with respect to N

T = normalize(T - dot(T, N) * N);

// then retrieve perpendicular vector B with the cross product of T and N

vec3 B = cross(N, T); mat3 TBN = mat3(T, B, N)
```

Conclusión

Es una técnica no muy cara que nos ofrece un extra de realismo considerable para ya sea un videojuego o un film animado sin tener que sobrecargar la geometría de los modelos.