PRACTICA 3

Introducción a la programación en OpenGL

Parte opcional

Adrián David Morillas Marco y Guillermo Meléndez Morales
Diseño y Desarrollo de Videojuegos + Ingeniería de Computadoras

Filtro Anisotrópico

"El filtrado anisotrópico es un método para mejorar la calidad de una textura en una superficie que está vista desde un ángulo oblicuo con respecto al ángulo de proyección de la textura sobre una superficie. "

Para realizar el filtrado anisotrópico lo primera que se deberá realizar es comprobar que la tarjeta gráfica lo soporta, para ello en el método main de la clase main, introduciremos las siguientes líneas, imprimiendo por consola si es soportado o no:

```
if (glewIsSupported("GL_EXT_texture_filter_anisotropic")){
        std::cout << "soportado ";
}
else{
        std::cout << " no soportado ";
}
```

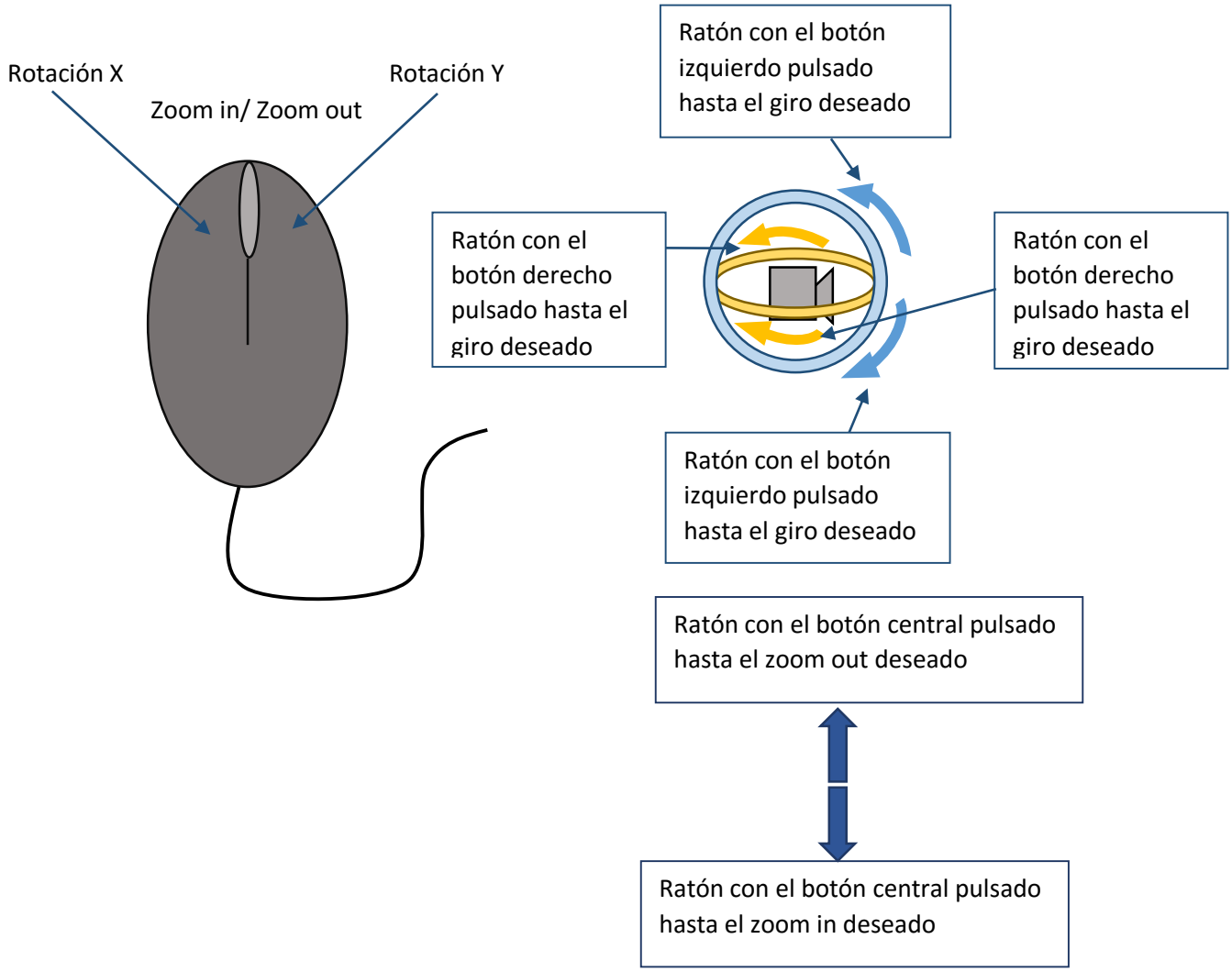
A continuación nos dirigimos al método loadTex, donde implementaremos el filtro anisotrópico, para ello, comentaremos el filtro anterior, y colocamos el siguiente código:

```
// 2.1 activamos el filtro anisotropico
GLfloat largest_supported_anisotropy;
// soportamos el máximo filtro anisotrópico posible
glGetFloatv(GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT, &largest_supported_anisotropy);
// aplicamos la máxima cantidad de filtro anisotrópico posible
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT,
        largest_supported_anisotropy);
```

Implementación de las funcionalidades de las Practicas 1 y 2

Giro de la cámara con el ratón

Para poder controlar la rotación, emplearemos el mismo método que empleamos en el apartado de la parte opcional: control de la cámara con el teclado. Es decir modificaremos la matriz view, mediante una matriz auxiliar:



Esto en código de programación se traduce a:

```
// 2.1.a giro de la cámara utilizando el ratón.
// variables del método mouseFunc
// diferenciax sera la variable que calcule la diferencia de la coordenada x original menos la final
float diferenciax;
// diferenciax sera la variable que calcule la diferencia de la coordenada y original menos la final
float diferenciay;
// actualx será la variable encargada de guardar la coordenada x incial
float actualx = 0.0f;
// actualy será la variable encargada de guardar la coordenada y incial
float actualy = 0.0f;
```

```
// angulo será la variable encargada de incrementar/ decrementar el angulo de rotacion de x e y
float angulo = 0.1f;
void mouseFunc(int button, int state, int x, int y){
// se crea una matrix auxiliar que se empleara para trasladar y rotar
glm::mat4 aux(1.0f);
// si se pulsa el raton...
if (state == 0){
		// se muestra por consola que se ha pulsado un botón del raton
		std::cout << "Se ha pulsado el botón ";
		// se recoge la coordenada x inicial
		actualx = x;
		// se recoge la coordenada y inicial
		actualy = y;
}
// si se suelta el botón una vez pulsada...
else
		// se muestra por consola que se ha soltado el botón del ratón
		std::cout << "Se ha soltado el botón ";
		// se resta la coordenada x inicial por la actual
		diferenciax = actualx - x;
		// se resta la coordenada y inicial por la actual
		diferenciay = actualy - y;

// si se pulsa el boton izquierdo del ratón...
if (button == 0){
		// se muestra por consola que el botón se ha pulsado
		std::cout << "de la izquierda del ratón " << std::endl;
		// se marca un margen para expresar el movimiento
		if (diferenciax <= 50){
		// se configura la matriz para rotar en el eje X
		aux = glm::rotate(aux, angulo*(diferenciay / 100.0f), glm::vec3(1.0f, 0.0f, 0.0f));
		// se multiplica la matriz aux por la matriz view
		view = aux*view;
		}
}
// si se pulsa el boton central del ratón...
if (button == 1) {
		// se muestra por consola que el botón se ha pulsado
		std::cout << "central del ratón " << std::endl;
		// se configura la matriz para hacer zoom in u out
		aux = glm::translate(aux, glm::vec3(0.0f, 0.0f, diferenciay / 100.0f));
		// se multiplica la matriz aux por la matriz view
		view = aux*view;
}
// si se pulsa el boton derecho del ratón....
if (button == 2){
		// se muestra por consola que el botón se ha pulsado
		std::cout << "de la derecha del ratón " << std::endl;
		// se marca un margen para expresar el movimiento
		if (diferenciay <= 50){
		// se configura la matriz para rotar en el eje Y
		aux = glm::rotate(aux, angulo*(diferenciax / 100.0f), glm::vec3(0.0f, 1.0f, 0.0f));
		// se multiplica la matriz aux por la matriz view
		view = aux*view;
		}
}
```

```
// se muestra por pantalla que acción se ha realizado
std::cout << "en la posición " << x << " " << y << std::endl << std::endl;
}
```

Crear un tercer cubo que orbite con una curva de Bézier

Para crear el tercer cubo se tomará el mismo procedimiento empleado en el segundo apartado de la parte obligatoria:

Creamos una nueva model:

```
glm::mat4 model3 = glm::mat4(1.0f);
```

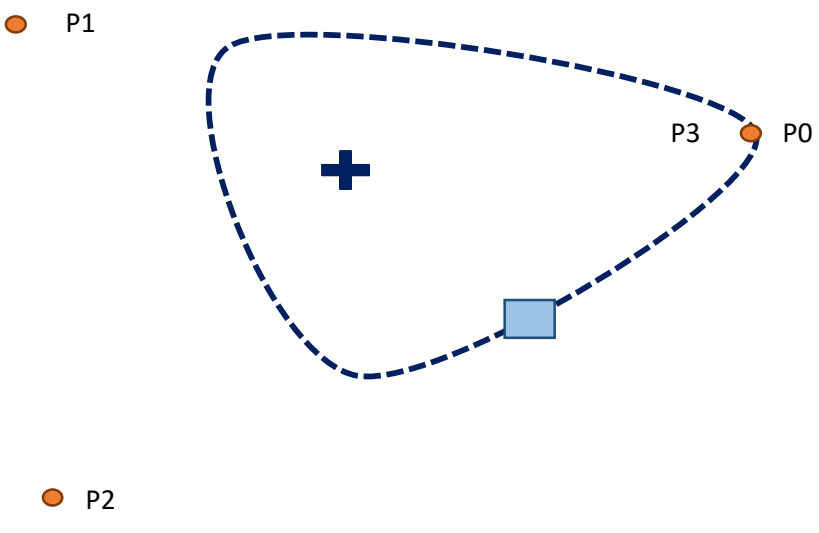
Se inicializa la model del objeto:

```
// 2.2.b inicializamos la matriz model del cubo de Bezier
model3 = glm::mat4(1.0f);
```

Se pinta el cubo con el mismo vao con el que se pintó el primer cubo:

```
modelView = view * model3;
modelViewProj = proj * view * model3;
normal = glm::transpose(glm::inverse(modelView));

if (uModelViewMat != -1)
        glUniformMatrix4fv(uModelViewMat, 1, GL_FALSE,
        &(modelView[0][0]));

if (uModelViewProjMat != -1)
        glUniformMatrix4fv(uModelViewProjMat, 1, GL_FALSE,
        &(modelViewProj[0][0]));

if (uNormalMat != -1)
        glUniformMatrix4fv(uNormalMat, 1, GL_FALSE,
        &(normal[0][0]));

glBindVertexArray(vao);
glDrawElements(GL_TRIANGLES, cubeNTriangleIndex * 3,
        GL_UNSIGNED_INT, (void*)0);
```

A continuación en la función Idle () se describe la curva de Bézier:



Lo mencionado anteriormente, en código de programación se traduce a:

```
// 2.2.b definimos la curva de Bezier del cubo de Bezier el cual rotará alrededor del primer cubo
        // xt y zt serán las variables que almacenaran los valores de Bezier
        float xt, zt;
        // t será la variable encargada de mover el cubo por la curva
        static float t = 0.0f;
        // t siempre mantendra un valor entre 0 y 1
        t = (t <= 1.0f) ? t + 0.0001f : 0.0f;
        // P0 (8, 0, -6)
        glm::vec3 P0 = glm::vec3(8, 0, -6);
        // P1 (-20, 0, -20)
        glm::vec3 P1 = glm::vec3(-20, 0, -20);
        // P2 (20, 0, 100)
        glm::vec3 P2 = glm::vec3(20, 0, 100);
        // P3 (8, 0, -6)
        glm::vec3 P3 = glm::vec3(8, 0, -6);;
        // x(t) = .....
        xt = pow((1 - t), 3)*P0.x + 3 * pow((1 - t), 2)*t*P1.x + (1 - t)*(pow(t, 2))*P2.x + pow(t, 3)*P3.x;
        // z(t) = ....
        zt = pow((1 - t), 3)*P0.z + 3 * pow((1 - t), 2)*t*P1.z + (1 - t)*(pow(t, 2))*P2.z + pow(t, 3)*P3.z;
        model3 = glm::mat4(1.0f);
        model3 = glm::translate(model3, glm::vec3(xt, 0.0f, zt));
        model3 = glm::scale(model3, glm::vec3(0.5f, 0.5f, 0.5f));
```

Crea un nuevo modelo y añádelo a la escena

Para añadir el nuevo modelo, será necesario crear un nuevo vao, donde almacenaremos las cualidades del objeto. Para ello, inicialmente importaremos la pirámide, la model del objeto y se crearan las variables globales que permitirán configurar el objeto:

```
#include "PIRAMID.h"

glm::mat4 model4 = glm::mat4(1.0f);

// VAO de la piramide
unsigned int vao2;
// VBOs que forman parte de la piramide
unsigned int posVBO2;
unsigned int colorVBO2;
unsigned int normalVBO2;
unsigned int tangentVBO2;
unsigned int texCoordVBO2;
unsigned int triangleIndexVBO2;
```

A continuación en la función initObj () creamos e inicializamos el vao de la pirámide y configuramos sus atributos:

```
// 2.2.c inicializamos la piramide
glGenVertexArrays(1, &vao2);
```

```
glBindVertexArray(vao2);



if (inPos != -1)
{
	glGenBuffers(1, &posVBO2);
	glBindBuffer(GL_ARRAY_BUFFER, posVBO2);
	glBufferData(GL_ARRAY_BUFFER, piramidNVertex*sizeof(float)* 3,
		piramidVertexPos, GL_STATIC_DRAW);

	glVertexAttribPointer(inPos, 3, GL_FLOAT, GL_FALSE, 0, 0);
	glEnableVertexAttribArray(inPos);
}
if (inColor != -1)
{
	glGenBuffers(1, &colorVBO);
	glBindBuffer(GL_ARRAY_BUFFER, colorVBO);
	glBufferData(GL_ARRAY_BUFFER, piramidNVertex*sizeof(float)* 3,
		piramidVertexColor, GL_STATIC_DRAW);

	glVertexAttribPointer(inColor, 3, GL_FLOAT, GL_FALSE, 0, 0);
	glEnableVertexAttribArray(inColor);
}
if (inNormal != -1)
{
	glGenBuffers(1, &normalVBO);
	glBindBuffer(GL_ARRAY_BUFFER, normalVBO);
	glBufferData(GL_ARRAY_BUFFER, piramidNVertex*sizeof(float)* 3,
		piramidVertexNormal, GL_STATIC_DRAW);

	glVertexAttribPointer(inNormal, 3, GL_FLOAT, GL_FALSE, 0, 0);
	glEnableVertexAttribArray(inNormal);
}
if (inTexCoord != -1)
{
	glGenBuffers(1, &texCoordVBO);
	glBindBuffer(GL_ARRAY_BUFFER, texCoordVBO);
	glBufferData(GL_ARRAY_BUFFER, piramidNVertex*sizeof(float)* 2,
		piramidVertexTexCoord, GL_STATIC_DRAW);

	glVertexAttribPointer(inTexCoord, 2, GL_FLOAT, GL_FALSE, 0, 0);
	glEnableVertexAttribArray(inTexCoord);
}
if(inTangent != -1)
{
	glGenBuffers(1, &tangentVBO2);
	glBindBuffer(GL_ARRAY_BUFFER, tangentVBO2);
	glBufferData(GL_ARRAY_BUFFER, cubeNVertex*sizeof(float)* 3,
		cubeVertexTangent, GL_STATIC_DRAW);

	glVertexAttribPointer(inTangent, 3, GL_FLOAT, GL_FALSE, 0, 0);
	glEnableVertexAttribArray(inTangent);
}

glGenBuffers(1, &triangleIndexVBO);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, triangleIndexVBO);
```

```
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
        piramidNTriangleIndex*sizeof(unsigned int)* 3, piramidTriangleIndex,
        GL_STATIC_DRAW);
```

Inicializamos la matriz model de la pirámide:

```
Model4 = glm::mat4(1.0f);
```

Liberamos los recursos empleados en la función distroy ():

```
glDeleteVertexArrays(1, &vao2);
```

Finalmente para añadir el cubo a la escena, es necesario pintarlo, para ello nos dirigimos a la función renderFunc ():

```
        modelView = view * model4;
        modelViewProj = proj * view * model4;
        normal = glm::transpose(glm::inverse(modelView));

        if (uModelViewMat != -1)
                glUniformMatrix4fv(uModelViewMat, 1, GL_FALSE,
                &(modelView[0][0]));

        if (uModelViewProjMat != -1)
                glUniformMatrix4fv(uModelViewProjMat, 1, GL_FALSE,
                &(modelViewProj[0][0]));

        if (uNormalMat != -1)
                glUniformMatrix4fv(uNormalMat, 1, GL_FALSE,
                &(normal[0][0]));

        glBindVertexArray(vao2);
        glDrawElements(GL_TRIANGLES, cubeNTriangleIndex * 3,
                GL_UNSIGNED_INT, (void*)0);
```

A continuación lo reescalaremos y le aplicaremos una rotación, para ello nos dirigimos a la función Idle ():

```
// 2.2.c definimos el movimiento de la pirámide
model4 = glm::mat4(1.0f);
model4 = glm::translate(model4, glm::vec3(3.0f, 0.0f, 0.0f));
model4 = glm::scale(model4, glm::vec3(0.5f, 0.5f, 0.5f));
```

Iluminar el objeto con diferentes luces e implementar la atenuación

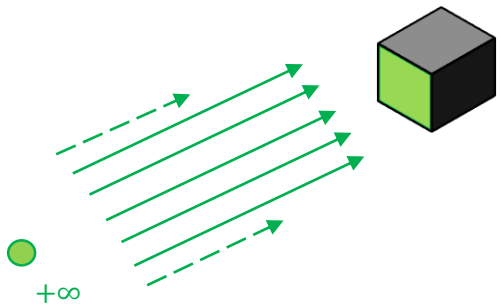
En este apartado se ha optado por realizar una luz difusa (ya definida), una luz direccional y una luz focal. Todas ellas han sido definidas en el shader de fragmentos. La atenuación es el desgaste de su iluminación a medida que su punto de luz se aleja. Se obtiene complementando la constante de atenuación al cálculo de la luz. La constante de atenuación tiene la siguiente formula:

$$fatt = \frac{1}{c_x + c_y * d + c_z * d^2}$$

En lenguaje de programación se traduce a:

```
// k1 valdra 1, k2 es la atenuacion lineal y k3 es la atenuacion cuadrática

vec3 C = vec3 (1.0, 0.0, 0.0);
```

Se comenzará por implementar la luz direccional, para ello se definirá una posición y una iluminación. En el caso implementado, nuestra posición e iluminación han sido declaradas de forma que puedan ser modificadas con el teclado:



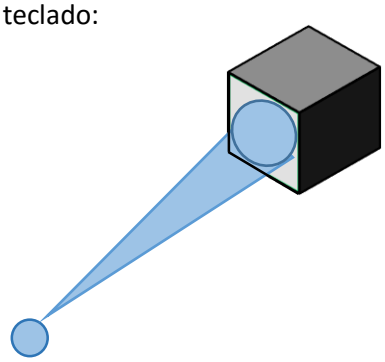

```
uniform vec3 lpos2;
uniform vec3 Id2;
```

A continuación se implementa en el método shade () la luz direccional, empleando la metodología de la segunda práctica:

"Para implementar la luz direccional simplemente habrá que crear una nueva iluminación difusa y una posición, y a continuación habrá que cambiar la luz de un punto a un vector, para ello tenemos la coordenada homogénea, la cual cambiaremos de un 1.0 a un 0.0."

```
float d2 = length (lpos2);                          // distancia desde el vértice hasta su incidencia
float fatt2 = 1/ (C.x + C.y*d2 + C.z*pow(d2, 2));   // hallamos la atenuación
vec3 L2 = normalize(vec3(View*vec4(lpos2, 0.0)));   // hallamos el vector L
vec3 directional = fatt2 *Id2*Kd*dot(N,L2);;
c += clamp(directional, 0.0, 1.0);
```

A continuación se implementará la luz focal, para ello se definirá la posición y la intensidad de la luz. Al igual que con la luz difusa, se ha definido la posición y la intensidad de forma que puedan ser modificadas con el teclado:



```
uniform vec3 lpos3;
uniform vec3 Id3;
```

A continuación se implementa en el método shade () la luz focal, empleando la metodología de la segunda práctica:

"Para implementarla emplearemos la siguiente formula:

$$si\ \vec{D} * (\overrightarrow{-L}) > cos\varepsilon \rightarrow I_{focal} = \left(\frac{(\vec{D} * (\overrightarrow{-L})) - cos\varepsilon}{1 - cos\varepsilon} \right)^{n} * I_{diff} \quad siendo\ I_{diff} = I_l * Kd$$

$$si\ \vec{D} * (\overrightarrow{-L}) < cos\varepsilon \rightarrow I_{focal} = 0$$

Siendo D la dirección del foco, el ángulo, el ángulo de apertura y n el coeficiente de atenuación del foco. Comenzaremos por crear una nueva luz y una nueva posición, y a continuación crearemos L, D, n y el ángulo."

```
vec3 L3 = normalize(vec3(view*vec4(lpos3, 1.0)-pos));

vec3 D = normalize(vec3(-lpos3));

float Ip;

float beta = 3.14159/17;

float no = 0.5;

float calculo1 = dot ( D, -L3 );

float calculo2 = cos(beta);

float calculo = (calculo1 - calculo2) / (1 - calculo2)

if ( calculo1 > calculo2 ){

        Ip = pow (calculo , no);

}else{

        Ip = 0;

}

vec3 idiff = vec3 (Id3*Kd);

focal = Ip*idiff;

c += clamp(focal, 0.0, 1.0);
```

Implementación del Bump mappibg

“Técnica que, gracias al uso de la luz, permite simular relieve en una textura, dándole sensación de profundidad o rugosidad a un objeto que en origen no era más que un color plano. “

Para implementar el bump mapping, se necesitan como entradas en el shader de vértices, las normales y las tangentes del objeto. En el shader de vértices tenemos las normales, pero carecemos de las tangentes, por lo que será necesario implementar la variable uniform inTangent, para adquirir las tangentes del objeto. Para ello:

Declaramos la variable global que permitirá acceder al atributo:

```
// 2.2.f creamos la variable inTangente, atributo del shader de vértices que proporcionará las tangentes del objeto
int inTangent;
```

Lo añadimos a los VBOs que forman parte del objeto:

```
unsigned int tangentVBO;
```

Añadimos el mapa de normales:

```
// 2.2.f se añade la textura de normales
unsigned int normalTexId;
```

Se crea la variable uniform:

```
int uNormalTex;
```

Declaramos el atributo inTangent del shader de vértices en initShader:

```
glBindAttribLocation(program, 4, "inTangent");
```

Creamos el identificador del mapa de normales:

```
uNormalTex = glGetUniformLocation(program, "normalTex");
```

Creamos el identificador de inTangent:

```
inTangent = glGetAttribLocation(program, "inTangent");
```

Configuramos el atributo para el cubo en initObj ():

```
if (inTangent != -1){
        glGenBuffers(1, &tangentVBO);
        glBindBuffer(GL_ARRAY_BUFFER, tangentVBO);
        glBufferData(GL_ARRAY_BUFFER, cubeNVertex*sizeof(float)* 3,
                cubeVertexTangent, GL_STATIC_DRAW);
        glVertexAttribPointer(inTangent, 3, GL_FLOAT, GL_FALSE, 0, 0);
```

```
        glEnableVertexAttribArray(inTangent);
}
```

Configuramos el atributo para la pirámide:

```
if (inTangent != -1){
        glGenBuffers(1, &tangentVBO2);
        glBindBuffer(GL_ARRAY_BUFFER, tangentVBO2);
        glBufferData(GL_ARRAY_BUFFER, cubeNVertex*sizeof(float)* 3,
                piramidVertexTangent, GL_STATIC_DRAW);
        glVertexAttribPointer(inTangent, 3, GL_FLOAT, GL_FALSE, 0, 0);
        glEnableVertexAttribArray(inTangent);
}
```

Cargamos el mapa de normales:

```
normalTexId = loadTex("../img/normal.png");
```

Definimos el atributo inTangent en la función de renderizado renderFunc ():

```
if (uNormalTex != -1)
{
        glActiveTexture(GL_TEXTURE0 + 2);
        glBindTexture(GL_TEXTURE_2D, normalTexId);
        glUniform1i(uNormalTex, 2);
}
```

Liberamos los recursos empleados en el metodo destroy ():

```
if (inTangent != -1) glDeleteBuffers(1, &tangentVBO);
glDeleteTextures(1, &normalTexId);
```

A continuación nos dirigimos al shader de vértices donde declararemos la matriz TBN, matriz de cambio de base entre coordenadas de las normales a coordenadas de las tangentes, a partir del atributo inNormal el cual nos proporciona las normales del objeto y el atributo inTangent el cual nos proporciona las tangentes del objeto:

```
in vec3 inNormal;
in vec3 inTangent;

vec3 Tangente = normalize (vec3 (inTangent));
vec3 Normal = normalize (vec3 (inNormal));
vec3 Binormal = normalize (vec3 (cross (Normal, Tangente)));

TBN [0].x = Tangente.x;
TBN [0].y = Binormal.x;
TBN [0].z = Normal.x;
TBN [1].x = Tangente.y;
TBN [1].y = Binormal.y;
TBN [1].z = Normal.z;
```

```
TBN [2].x = Tangente.z;
TBN [2].y = Binormal.z;
TBN [2].z = Normal.z;
```

Declaramos como salida del shader de vértices la matriz de cambio de base TBN:

```
out mat3 TBN;
```

En el shader de fragmentos recibimos como entrada la matriz de cambio de base TBN y el mapa de normales:

```
in mat3 TBN;

uniform sampler2D normalTex;
```

Asignamos el mapa de normales:

```
vec4 Normal;

Normal = texture(normalTex, texcoord);
```

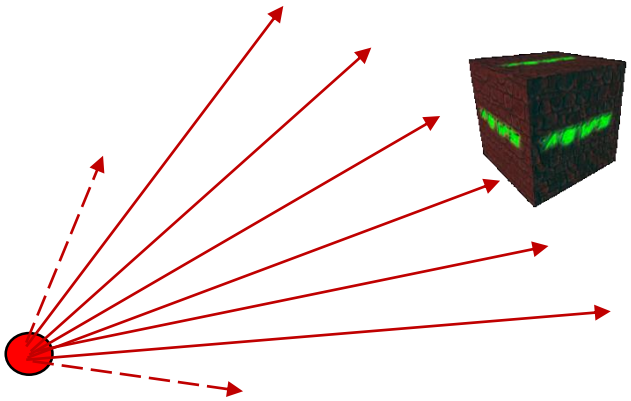
A continuación en el main (), extraemos las normales del mapa de normales:

```
vec3 coordlocal = 2.0* Normal.rgb - vec3(1.0);
```

Para realizar el cambio de base, pasaremos primero a coordenadas del objeto (TBN) y posteriormente pasaremos a coordenadas de la cámara (modelView):

```
N = normalize (TBN*coordlocal*mat3(modelView));
```

Obteniendo finalmente el bump mapping:



Añadir textura especular

Para añadir la textura especular será necesario emplear los mismos pasos que se emplearon en el apartado anterior con el mapa de normales, para ello:

Añadimos la textura especular:

```
unsigned int specularTexId;
```

Se crea la variable uniform:

```
int uSpecularTex;
```

Creamos el identificador de la textura especular:

```
uSpecularTex = glGetUniformLocation(program, "specularTex");
```

Liberamos los recursos empleados en el metodo destroy ():

```
glDeleteTextures(1, &specularTexId);
```

Asignamos la textura especular:

```
specularTexId = loadTex("../img/specmap.png");
```

Configuramos la textura en la función de renderizado:

```
if (uSpecularTex != -1)
{
	glActiveTexture(GL_TEXTURE0 + 3);
	glBindTexture(GL_TEXTURE_2D, specularTexId);
	glUniform1i(uSpecularTex, 3);
}
```

A continuación en el shader de fragmentos, se coloca como entrada:

```
uniform sampler2D specularTex;
```

Asignamos la textura especular:

```
Ks = texture(specularTex, texcoord).rgb;
```

Finalmente aplicamos el reflejo especular a las luces implementadas anteriormente:

```
vec3 V = normalize (-pos);
	// especular difusa
	vec3 R = normalize (reflect (-L,N));
	float factor = max (dot (R,V), 0.01);
	vec3 specular = Is*Ks*pow(factor,alpha);
	c += clamp(specular, 0.0, 1.0);

	// especular direccional
	vec3 R2 = normalize (reflect (-L2,N));
	float factor2 = max (dot (R2,V), 0.01);
	vec3 specular2 = Is*Ks*pow(factor2,alpha);
	c += clamp(specular2, 0.0, 1.0);

	// especular focal
	vec3 R3 = normalize (reflect (-L3,N));
	float factor3 = max (dot (R3,V), 0.01);
	vec3 specular3 = Is*Ks*pow(factor3,alpha);
	c += clamp(specular3, 0.0, 1.0);
```