
Universidad de Huelva

# Tema 10

# Generación de sombras

# Índice

# Índice

- Una subrutina es un mecanismo que permite seleccionar una función (entre un conjunto de funciones posibles) basado en el valor de una variable uniforme.
- Las subrutinas proporcionan una forma de seleccionar implementaciones alternativas en tiempo de ejecución sin necesidad de cambiar el programa GLSL ni utilizar instrucciones if.
- En muchos sentidos es similar a los punteros a funciones de C.
- La variable uniforme sirve como puntero a la función y se utiliza para invocarla.
- Las funciones que implementan una subrutina utilizan distintos nombres, pero debe tener el mismo número y tipo de parámetros y el mismo tipo de retorno.

- Para declarar una subrutina en un shader se utiliza la instrucción ***subroutine*** seguida de la declaración del tipo de función. Por ejemplo,

```
subroutine vec3 shadeModelType( vec4 position, vec3 normal);
```

- Esta declaración define el nombre de la subrutina, los argumentos de la función y el tipo de dato devuelto.
- Para declarar una variable uniforme que contenga un puntero a una subrutina de utiliza la instrucción ***subroutine uniform***.

```
subroutine uniform shadeModelType shadeModel;
```

- Para declarar una función que desarrolla una subrutina se utiliza el modificador ***subroutine( nombre )*** seguida de la declaración de la función.

```
subroutine( shadeModelType )
vec3 phongModel( vec4 position, vec3 norm )
{
    …
}

subroutine( shadeModelType )
vec3 diffuseOnly( vec4 position, vec3 norm )
{
    …
}
```

- Una misma función puede desarrollar varias subrutinas al mismo tiempo. En ese caso aparece una lista de nombres entre paréntesis.

- Para usar una subrutina se utiliza la variable uniforme como si fuera una función.

```
void main()
{
    …
    LightIntensity = shadeModel(eyePosition,eyeNorm);
    …
}
```

- La selección de la subrutina en el programa OpenGL se realiza asignando el valor a la variable uniforme. Para ello en primer lugar hay que obtener la posición de las funciones que desarrollan la subrutina:

```
GLuint phongModelIndex = glGetSubroutineIndex(
          programHandle,
          GL_VERTEX_SHADER,
          "phongModel" );

GLuint diffuseOnlyIndex = glGetSubroutineIndex(
          programHandle,
          GL_VERTEX_SHADER,
          "diffuseOnly");
```

- Por último, una vez conocida la posición de cada función (el "puntero" a la función), se asigna el valor deseado a la variable uniforme.

```
glUniformSubroutinesuiv( GL_VERTEX_SHADER,
                         1,
                        &phongModelIndex);
```

- Esta función asigna una lista de subrutinas. El segundo argumento indica el número de subrutinas a asignar. El tercer argumento es un array a las posiciones de las funciones. En nuestro caso solo se asigna una función así que se puede utilizar la dirección de la variable *phongModelIndex* como si fuera el comienzo del array.

- El orden en el que se asignan las subrutinas a las variables uniformes es el mismo en el que se han definido en el programa. El array debe contener la asignación de funciones en el orden adecuado.
- Cuando el programa incluye varias variables uniformes de tipo subrutina, se puede consultar la posición de cada una con el comando *glGetSubroutineUniformLocation()* y de esa forma ordenar el array de manera correcta.

# Índice

- Como hemos visto hasta ahora, el proceso de renderizado genera el contenido de la imagen en una estructura que conocemos como *ColorBuffer*.
- Si activamos la opción de test de profundidad, además de información sobre el color de cada pixel también se almacena información sobre la coordenada *z* (la profundidad) asociada a cada pixel. Esta información se almacena en una estructura paralela conocidad como *DepthBuffer*.
- Existe otro buffer, denominado *StencilBuffer* que se utiliza como buffer auxiliar para almacenar otra información configurable. Este buffer se utiliza para programar algunos efectos (como reflejos, por ejemplo).

- OpenGL permite dirigir la salida del proceso de renderizado hacia otras estructuras que no estén vinculadas directamente con el contenido a mostrar en la pantalla. De esta forma, se puede generar una imagen y almacenarla en lugar de mostrarla. Esta imagen almacenada se puede utilizar posteriormente.
- Se denomina *Framebuffer Object* (FBO) a la estructura de datos que describe la salida del proceso de renderizado. Este FBO contiene enlaces al *ColorBuffer*, *DepthBuffer* y *StencilBuffer* utilizados en el renderizado. Estos buffers pueden ser objetos textura u objetos *RenderBuffer*.
- OpenGL crea un *Framebuffer* por defecto que asocia a la pantalla. Este FBO tiene preasignado el identificador 0.

- Para dirigir el proceso de renderizado a memoria (en vez de a la pantalla) hay que crear un nuevo FBO, crear los buffers necesarios, vincularlos al FBO y activar el FBO. De esta forma el renderizado se dirigirá a las estructuras enlazadas al FBO.

```
GLuint fboHandle; // The handle to the FBO

// Generate and bind the framebuffer
glGenFramebuffers(1, &fboHandle);
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);
```

- El identificador de un FBO es un valor de tipo *GLuint*. Para crear un nuevo FBO se utiliza el comando *glGenFramebuffers()*. Para activarlo se utiliza el comando *glBindFramebuffer()*.

- Para crear el *ColorBuffer* y asociarlo al FBO se utiliza una textura 2D.

```
// Create the texture object
GLuint renderTex;
glGenTextures(1, &renderTex);
glActiveTexture(GL_TEXTURE0); // Use texture unit 0
glBindTexture(GL_TEXTURE_2D, renderTex);

glTexImage2D(GL_TEXTURE_2D,0,GL_RGBA,512,512,
             0,GL_RGBA,GL_UNSIGNED_BYTE,NULL);

glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Bind the texture to the FBO
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D, renderTex, 0);
```

- En este caso el contenido de la textura no se lee desde un fichero, sino que se generará en el proceso de renderizado. El comando *glTexImage2D()* se utiliza para configurar las propiedades de la textura (tamaño y formato de almacenamiento) pero el último argumento (puntero al contenido) se deja a nulo.
- El comando *glFramebufferTexture2D()* permite enlazar el *ColorBuffer* del FBO al buffer de textura que hemos definido. El último parámetro indica el nivel de *mipmap* de la textura que estamos enlazando al *Framebuffer*.

- Para crear el *DepthBuffer* y asociarlo al FBO se utiliza un *RenderBuffer*.

```
// Create the depth buffer
GLuint depthBuf;
glGenRenderbuffers(1, &depthBuf);
glBindRenderbuffer(GL_RENDERBUFFER, depthBuf);
glRenderbufferStorage(GL_RENDERBUFFER,
                      GL_DEPTH_COMPONENT,
                      512, 512);

// Bind the depth buffer to the FBO
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
                          GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, depthBuf);
```

- El tamaño del DepthBuffer debe coincidir con el del ColorBuffer.

- También se puede crear el *DepthBuffer* por medio de una textura.

```
// Create the texture object
GLuint depthTex;
glGenTextures(1, &depthTex);
glActiveTexture(GL_TEXTURE0); // Use texture unit 0
glBindTexture(GL_TEXTURE_2D, depthTex);

glTexImage2D(GL_TEXTURE_2D,0,
             GL_DEPTH_COMPONENT,512,512, 0,
           GL_DEPTH_COMPONENT,GL_UNSIGNED_BYTE,NULL);

glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,
                GL_TEXTURE_MAG_FILTER, GL_NEAREST);
// Bind the texture to the FBO
glFramebufferTexture2D(GL_FRAMEBUFFER,
                       GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, depthTex, 0);
```

- En este caso el formato de la textura no es GL_RGBA sino GL_DEPTH_COMPONENT.
- El comando *glFramebufferTexture2D()* utiliza el parámetro GL_DEPTH_ATTACHMENT para enlazar el *DepthBuffer* del FBO al buffer de textura que hemos definido.
- Para terminar, hay que indicar cual va a ser el buffer de dibujo.

```
// Set the target for the fragment shader outputs
GLenum drawBufs[] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, drawBufs);

// Revert to default framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- El último comando vuelve a activar el FBO por defecto, dirigiendo de nuevo la imagen a la pantalla.

- Una vez configurado el FBO alternativo, para activarlo se utiliza el siguiente código.

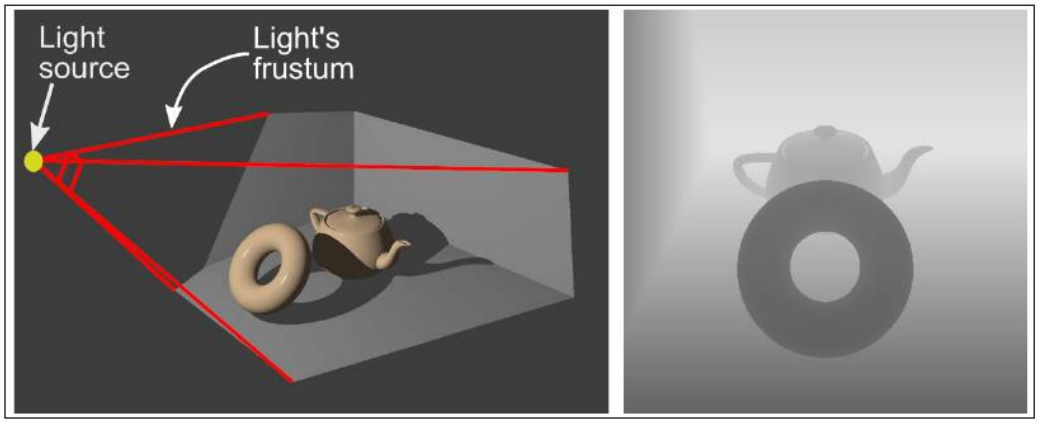
```
// Bind to texture's FBO
glBindFramebuffer(GL_FRAMEBUFFER, fboHandle);

// Viewport for the texture
glViewport(0,0,512,512);

// Render the scene to the FBO
…
```

- Para reactivar el modo pantalla sería

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0,0,width,height);
// Render the scene to the window
…
```

- Para añadir sombras a un dibujo hay que saber si cada pixel se encuentra en una zona de sombra o no. Si un pixel está en una zona de sombra solo se ilumina con luz ambiental mientras que si está en una zona iluminada hay que colorearlo con luz ambiental, difusa y especular.
- Para añadir la sombra hay que modificar el *FragmentShader* para estudiar si el pixel está en zona iluminada o sombreada.

- El algoritmo básico para generar las sombras se conoce como *ShadowMap* y consiste en generar en primer lugar la imagen desde el punto de vista del foco de luz y almacenar la información de profundidad en un buffer de textura.
- Si estamos iluminando con un foco puntual, hay que configurar la proyección como una proyección en perspectiva (*frustum*) mientras que para las luces direccionales la matriz de proyección se calcula como una proyeción ortográfica (*ortho*).
- Para almacenar la imagen generada desde la posición de la luz hay que definir un framebuffer y configurarlo para que la información de profundidad se almacene en forma de textura.



- A la hora de generar la imagen hay que considerar las siguientes matrices de transformación:
  - *Model*: matriz de transformación entre coordenadas del objeto y coordenadas del modelo (escenario).
  - *LightView*: matriz de transformación entre coordenadas del modelo y coordenadas del foco de luz.
  - *LightProjection*: matriz de transformación entre coordenadas del foco de luz y el ClippingVolume.
  - *View*: matriz de transformación entre coordenadas del modelo y coordenadas de la cámara.
  - *Projection*: matriz de transformación entre coordenadas de la cámara y el ClippingVolume.

- Para generar la imagen desde el punto de vista de la luz hay que considerar

$$LightMVP = LightProjection \cdot LightView \cdot Model$$

  Además hay que modificar el Viewport para que la imagen tenga el tamaño de la textura.

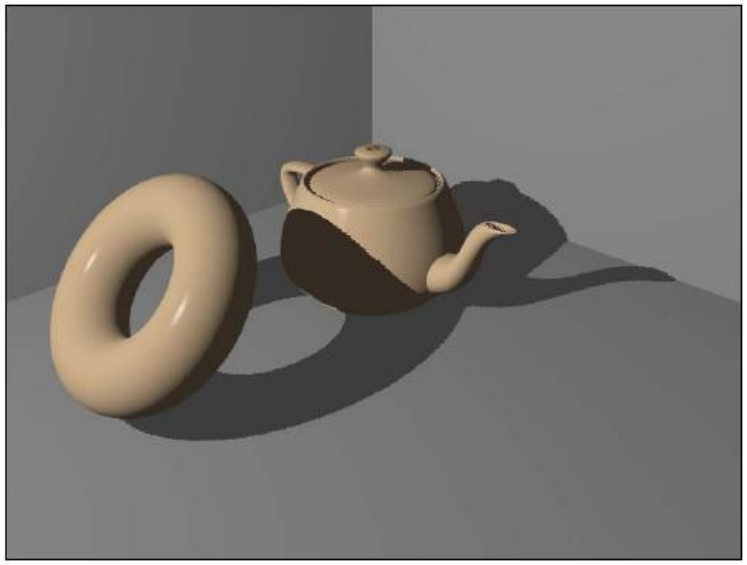
- Para generar la imagen desde el punto de vista de la cámara hay que considerar

$$MVP = Projection \cdot View \cdot Model$$

  y modificar el Viewport para que la imagen tenga el tamaño de la ventana.

- Dada una coordenadas asociadas a un pixel ($p$), se puede multiplicar por la tranformación de la luz ($lightMVP$) para obtener las coordenadas clip del pixel en el sistema de la luz. Para obtener las coordenadas de textura hay que convertir las coordenadas clip (entre -1 y 1) en valores de coordenadas de textura (entre 0 y 1).

- Dado un pixel, *p*, las coordenadas clip de la luz indican la distancia de ese punto al foco de luz (coordenada *z*). A partir de las coordenadas de textura de luz de ese pixel se puede obtener la profundidad asociada a ese punto en la imagen generada desde el foco de luz.
- Si la profundidad de la textura es menor que la profundidad del pixel, entonces es que hay un objeto más cercano a la luz que se interpone y, por tanto, el pixel está en zona de sombra.



- Creación del *FrameBuffer* en OpenGL

```
GLfloat border[] = { 1.0f, 1.0f, 1.0f, 1.0f };
GLsizei shadowMapWidth = 1024, shadowMapHeight = 1024;

glGenFramebuffers(1, &shadowFBO);
glBindFramebuffer(GL_FRAMEBUFFER, shadowFBO);

glGenTextures(1, &depthTex);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, depthTex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT24,
             shadowMapWidth, shadowMapHeight, 0,
             GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_BORDER);
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, border);

glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthTex, 0);

glDrawBuffer(GL_NONE);
glBindFramebuffer(GL_FRAMEBUFFER, 0);
```

- Genera el ShadowMap en OpenGL

```
// Asigna las matrices View y Projection de la luz.
glm::mat4 lightViewMatrix = GetLightViewMatrix();
glm::mat4 lightPerspective = GetLightProjection();
glm::mat4 lightMVP = lightPerspective*lightViewMatrix;

// Activa el framebuffer de la sombra
glBindFramebuffer(GL_FRAMEBUFFER, shadowFBO);

// Limpia la información de profundidad
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Selecciona la subrutina recordDepth
program->SetFragmentShaderUniformSubroutine("recordDepth");

// Activa front-face culling
glCullFace(GL_FRONT);

//Asigna el viewport
glViewport(0,0, shadowMapWidth, shadowMapHeight);

// Dibuja la escena
scene->Draw(program, lightPerspective, lightViewMatrix, lightMVP);
```

- Genera la imagen (con la sombra) en OpenGL

```
// Activa el framebuffer de la imagen
glBindFramebuffer(GL_FRAMEBUFFER, 0);

// Limpia el framebuffer
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// Activa back-face culling
glCullFace(GL_BACK);

// Selecciona la subrutina shadeWithShadow
program->SetFragmentShaderUniformSubroutine("shadeWithShadow");
program->SetUniformI("ShadowMap", 0);

// Asigna el viewport
glViewport(0,0,wndWidth,wndHeight);

// Dibuja la escena
viewMatrix = camera->PlaceCamera();
scene->Draw(program, projectionMatrix, viewMatrix, lightMVP);
```

- Vertex Shader

```
#version 400

layout(location = 0) in vec3 VertexPosition;
layout(location = 1) in vec3 VertexNormal;

uniform mat4 MVP;
uniform mat4 ViewMatrix;
uniform mat4 ModelViewMatrix;
uniform mat4 ShadowMatrix;

out vec3 Position;
out vec3 Normal;
out vec4 ShadowCoord;

void main()
{
  vec4 n4 = ModelViewMatrix*vec4(VertexNormal, 0.0);
  vec4 v4 = ModelViewMatrix*vec4(VertexPosition, 1.0);
  Normal = vec3(n4);
  Position = vec3(v4);
  ShadowCoord = ShadowMatrix * vec4(VertexPosition, 1.0);
  gl_Position = MVP * vec4(VertexPosition, 1.0);
}
```

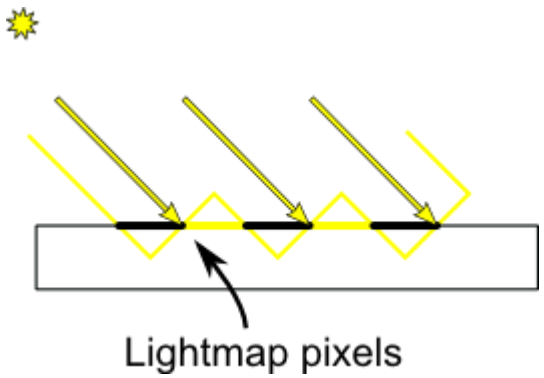
- Fragment Shader

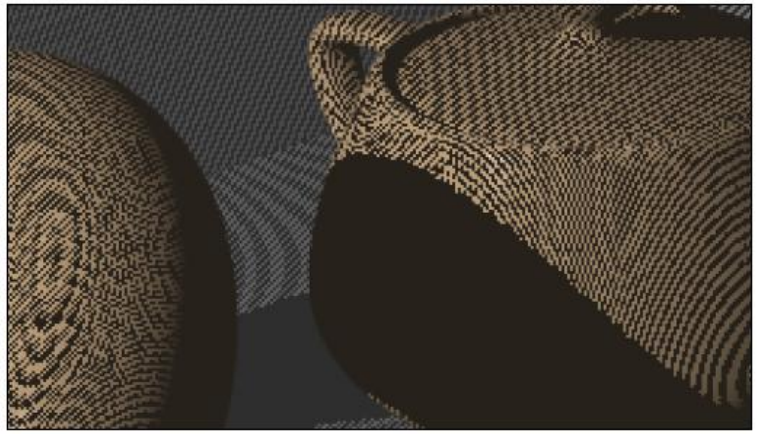
```
#version 400

in vec3 Position;
in vec3 Normal;
in vec4 ShadowCoord;

uniform sampler2D ShadowMap;
uniform mat4 ViewMatrix;

struct LightInfo { … };
uniform LightInfo Light;

struct MaterialInfo{ … };
uniform MaterialInfo Material;

out vec4 FragColor;

vec3 DiffusseAndSpecular() { … }

subroutine void RenderPassType();
subroutine uniform RenderPassType RenderPass;
```

- Fragment Shader

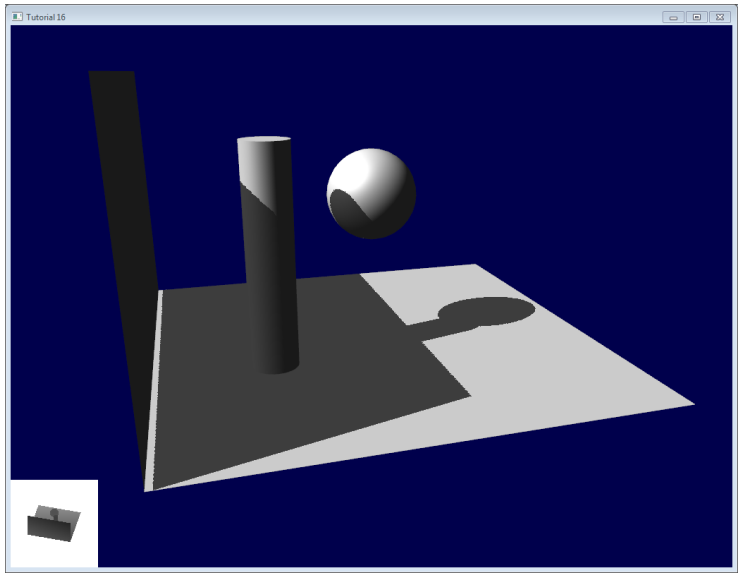
```
subroutine (RenderPassType)
void shadeWithShadow() {
  vec3 ambient = Light.La * Material.Ka;
  vec3 diffAndSpec = DiffusseAndSpecular();
  float ShadowCoordX = (ShadowCoord.x/ShadowCoord.w) * 0.5 + 0.5;
  float ShadowCoordY = (ShadowCoord.y/ShadowCoord.w) * 0.5 + 0.5;
  float ShadowCoordZ = (ShadowCoord.z/ShadowCoord.w) * 0.5 + 0.5;
  float shadowDepth = (texture(ShadowMap,
                               vec2(ShadowCoordX,ShadowCoordY))).z;
  float shadow = 1.0;
  if(shadowDepth < ShadowCoordZ) shadow = 0;
  FragColor = vec4(shadow * diffAndSpec + ambient, 1.0);
}

subroutine (RenderPassType)
void recordDepth() {
}

void main() {
  RenderPass();
}
```

- Uno de los problemas que aparecen en este algoritmo básico es que los píxeles de las zonas iluminadas pueden aparecer alternativamente en luz o en sombra (*shadow acne*). Esto se debe a la comparación entre la profundidad del pixel y la profundidad de la textura. Aunque teóricamente el valor debería ser el mismo, debido al pixelado el valor puede ser ligeramente superior o inferior (*Z-fighting*). Para evitarlo el ShadowMap se debe generar dibujando las caras de atrás de los objetos en vez de la de delante (*front-face culling*). Esto evita gran parte del problema, aunque puede seguir apareciendo en las zonas limítrofes entre luz y sombra.



- Otra opción para evitar el *shadow acne* consiste en añadir un *offset* al valor de la profundidad antes de compararlo con el valor almacenado como textura. Esto se puede hacer automáticamente con el comando *glPolygonOffset()*. Esta técnica puede ayudar a resolver el problema de *shadow acne*, pero puede producir otro efecto indeseado que consiste en que las sombras se separan un poco de los objetos (lo que se conoce como *peterpanning*, por la escena de la sombra de Peter Pan)



# Índice

- El problema más importante que tiene la versión básica del algoritmo ShadowMap es el *aliasing*. El pixelado de la sombra resulta muy evidente al proyectarla. Se puede disminuir este efecto aumentando el tamaño de la textura, pero para que no sea apreciable es necesario generar la textura con un tamaño enorme.
- Para reducir este problema sin aumentar en exceso el tamaño de la textura se han propuesto diferentes técnicas de *antialiasing*.

- Una de estas técnicas se conoce como *percentage-closer filtering* (PCF). Básicamente consiste en estudiar los texels vecinos para obtener un porcentage de sombra, en lugar de un valor 0 o 1. El resultado es que el borde de la sombra se difumina, disminuyendo la sensación de aliasing.

- Para desarrollar esta técnica hay que modificar la configuración de la textura de profundidad.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                               GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC,GL_LESS);
```

- El tercer comando asigna el valor GL_COMPARE_REF_TO_TEXTURE a la propiedad GL_TEXTURE_COMPARE_MODE. De esta forma, al acceder a una coordenada de textura, en lugar de devolver el valor del color, devuelve el resultado de la comparación entre la componente Z de la coordenada y el valor de profundidad almacenado.
- El último comando asigna la función de comparación. Si el valor almacenado es menor que la componente Z de la coordenada, la función devuelve 1. En caso contrario devuelve 0.

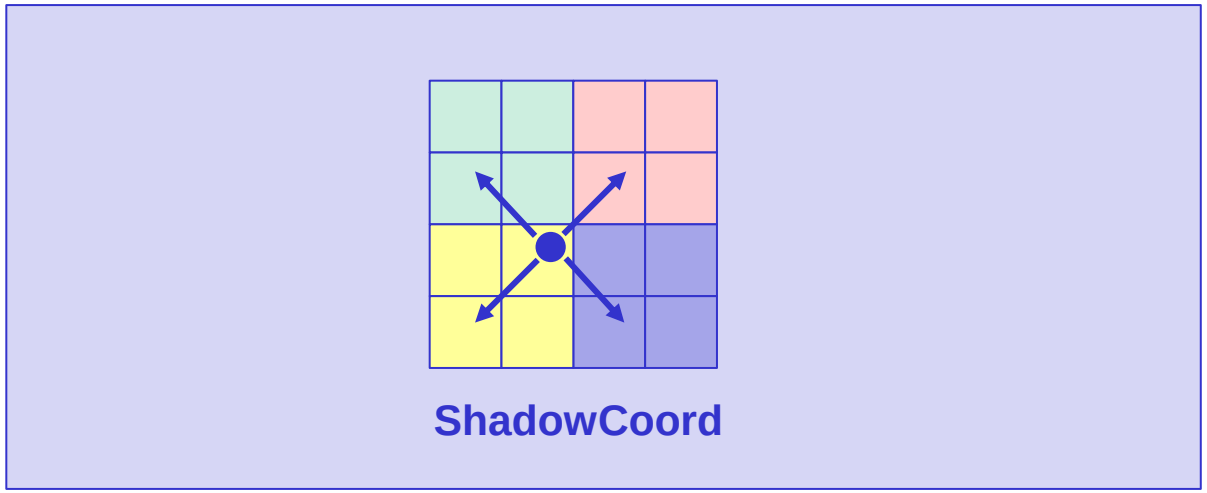
- Con respecto al FragmentShader, para utilizar el modo de comparación hay que tratar la textura como un tipo especial denominado *sampler2DShadow*. Con este tipo de textura, el comando texture..() requiere que las coordenadas utilicen 3 componentes (para incluir la componente Z a comparar) y el resultado no es *vec4* sino *float* (el resultado de la comparación).
- Si se utilizan filtros lineales, entonces se hace una interpolación lineal entre los cuatro resultados, obteniendo un valor entre 0.0 y 1.0.

```
  …
  uniform sampler2DShadow ShadowMap;
  …
  subroutine (RenderPassType)
  void shadeWithShadow()
  {
    vec3 ambient = Light.La * Material.Ka;
    vec3 diffAndSpec = DiffusseAndSpecular();
    float shadow = textureProj(ShadowMap, ShadowCoord);
    FragColor = vec4(diffAndSpec * shadow + ambient, 1.0);
}
```

- Para aplicar la técnica PCF, se calcula el valor de *shadow* en varios texels cercanos y se calcula la media. De esta forma se suaviza el contorno de la sombra.
- Para obtener el valor sobre un texel cercano se utiliza la función *textureProjOffset()* añadiendo el desplazamiento en texels.

```
subroutine (RenderPassType)
void shadeWithShadow()
{
  vec3 ambient = Light.Intensity * Material.Ka;
  vec3 diffAndSpec = phongModelDiffAndSpec();

  float sum = 0;
  sum += textureProjOffset(ShadowMap, ShadowCoord, ivec2(-1,-1));
  sum += textureProjOffset(ShadowMap, ShadowCoord, ivec2(-1,1));
  sum += textureProjOffset(ShadowMap, ShadowCoord, ivec2(1,1));
  sum += textureProjOffset(ShadowMap, ShadowCoord, ivec2(1,-1));

  float shadow = sum * 0.25;
  FragColor = vec4(ambient + diffAndSpec * shadow,1.0);
}
```

- Con esta técnica, el valor final de *shadow* se basa en una interpolación sobre 16 texels. Se puede añadir más llamadas a *textureProjOffset()* para aumentar el tamaño de la zona suavizada.




- Nota: La diferencia entre *texture()* y *textureProj()* es que la última normaliza las coordenadas dividiéndolas por la coordenada W.
- El valor de *ShadowCoord* debe encontrarse ya entre 0 y 1, así que la transformación (xyz * 0.5 + 0.5) debe incluirse en *lightMVP*.

- La técnica PCF tiene la desventaja de que obliga a estudiar *N* texels para cada pixel. Cuando un pixel está en una zona interior de luz o de sombra todo este cálculo es inutil porque todos los texels darán el mismo valor (0 o 1).
- Una opción alternativa consiste en generar los texels vecinos de forma aleatoria sobre un determinado radio. Si todos los texels tienen el mismo valor, entonces el pixel está totalmente iluminado o totalmente en sombra. Si no todos los texels coinciden, entonces se procesan más texels aleatorios para obtener un buen efecto de difuminado.
- Para no calcular constantemente valores aleatorios, se genera un conjunto de patrones aleatorios ciruclares y se almacenan en forma de textura 3D para compartirlos en todos los pixeles.

- Creación de la textura de offset en OpenGL

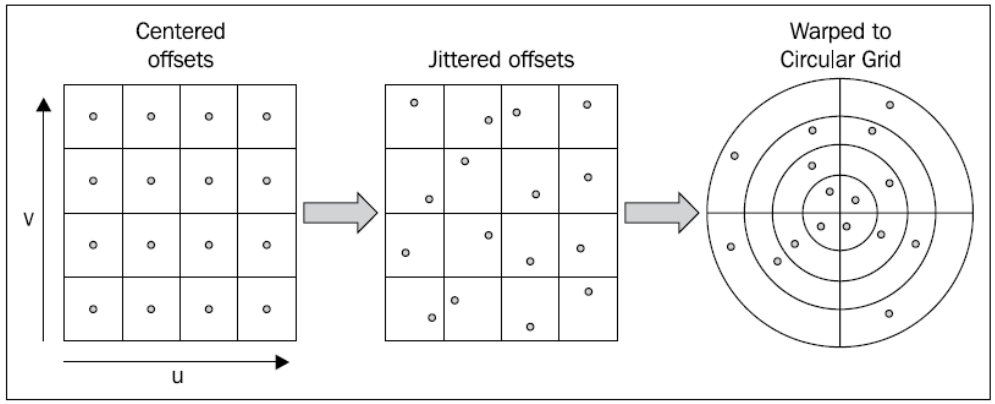
```
void buildOffsetTex(int texSize, int samplesU, int samplesV)
{
  int size = texSize;
  int samples = samplesU * samplesV;
  int bufSize = size * size * samples * 2;
  float *data = new float[bufSize];

  for( int i = 0; i< size; i++ ) {
    for(int j = 0; j < size; j++ ) {
      for( int k = 0; k < samples; k += 2 ) {
        int x1,y1,x2,y2;
        x1 = k % (samplesU);
        y1 = (samples - 1 - k) / samplesU;
        x2 = (k+1) % samplesU;
        y2 = (samples - 1 - k - 1) / samplesU;

        vec4 v;
        // Center on grid and jitter
        v.x = (x1 + 0.5f) + jitter();
        v.y = (y1 + 0.5f) + jitter();
        v.z = (x2 + 0.5f) + jitter();
        v.w = (y2 + 0.5f) + jitter();
```

- Creación de la textura de offset en OpenGL

```
        …
        // Scale between 0 and 1
        v.x /= samplesU;
        v.y /= samplesV;
        v.z /= samplesU;
        v.w /= samplesV;
        // Warp to disk
        int cell = ((k/2) * size * size + j * size + i) * 4;

        data[cell+0] = sqrtf(v.y) * cosf(TWOPI*v.x);
        data[cell+1] = sqrtf(v.y) * sinf(TWOPI*v.x);
        data[cell+2] = sqrtf(v.w) * cosf(TWOPI*v.z);
        data[cell+3] = sqrtf(v.w) * sinf(TWOPI*v.z);
      }
    }
  }

  glActiveTexture(GL_TEXTURE1);
  GLuint texID;
  glGenTextures(1, &texID);
  glBindTexture(GL_TEXTURE_3D, texID);
  …
```

- Creación de la textura de offset en OpenGL

```
  …
  glTexImage3D(GL_TEXTURE_3D, 0, GL_RGBA32F, size, size,
               samples/2, 0, GL_RGBA, GL_FLOAT, data);
  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_3D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  delete [] data;
}

// Return random float between -0.5 and 0.5
float jitter()
{
  return ((float)rand() / RAND_MAX) - 0.5f;
}
```



- Con respecto al FragmentShader, hay que considerar la textura que describe los texels vecinos (*OffsetTex*), el tamaño de la textura (*OffsetTexSize*) y la relación entre el radio de desplazamiento máximo y el tamaño del ShadowMap (*Radius*).

```
…
uniform sampler3D OffsetTex;
uniform vec3 OffsetTexSize; // (width, height, depth)
uniform float Radius;
…
subroutine (RenderPassType)
void shadeWithShadow()
{
  vec3 ambient = Light.La * Material.Ka;
  vec3 diffAndSpec = DiffusseAndSpecular();

  ivec3 offsetCoord;
  offsetCoord.xy = ivec2( mod(gl_FragCoord.xy,OffsetTexSize.xy ));
  float sum = 0.0;
  int samplesDiv2 = int(OffsetTexSize.z);
  …
```

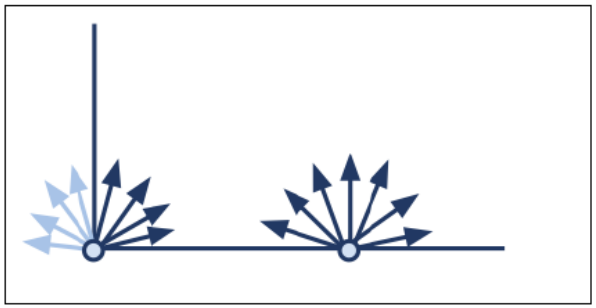
```
    vec4 sc = ShadowCoord;
    for( int i = 0 ; i< 4; i++ ) {
      offsetCoord.z = i;
      vec4 offsets = texelFetch(OffsetTex,offsetCoord,0) *
                                          Radius * ShadowCoord.w;
      sc.xy = ShadowCoord.xy + offsets.xy;
      sum += textureProj(ShadowMap, sc);
      sc.xy = ShadowCoord.xy + offsets.zw;
      sum += textureProj(ShadowMap, sc);
    }
    float shadow = sum / 8.0;

    if( shadow != 1.0 && shadow != 0.0 ) {
      for( int i = 4; i< samplesDiv2; i++ ) {
        offsetCoord.z = i;
        vec4 offsets = texelFetch(OffsetTex, offsetCoord,0) *
                                           Radius * ShadowCoord.w;
        sc.xy = ShadowCoord.xy + offsets.xy;
        sum += textureProj(ShadowMap, sc);
        sc.xy = ShadowCoord.xy + offsets.zw;
        sum += textureProj(ShadowMap, sc);
      }
      shadow = sum / float(samplesDiv2 * 2.0);
    }
    FragColor = vec4(diffAndSpec * shadow + ambient, 1.0);
}
```

# Índice

- Hasta ahora en nuestro modelo de iluminación hemos considerado que la luz ambiental procede de todas direcciones, choca contra cualquier punto y es emitida hacia todas las direcciones. De esta manera, el efecto de la luz ambiental se modela simplemente como

$$Intensidad = Light_A * Material_A$$

- En realidad, la luz ambiental no llega de forma homogenea a todos los puntos. Por ejemplo, en una esquina o en el interior de un hueco la luz ambiental no puede llegar desde todas las direcciones y, por tanto, la intensidad en esos puntos debe ser menor. Este fenómeno se conoce como ***Oclusión Ambiental***.



- Intentar modelar este efecto calculando la proporción de direcciones que alcanzan cada punto es muy costoso. En lugar de esto se utilizan texturas asociadas a los objetos que almacenan el nivel de oclusión de cada punto. De esta forma, se asocia a cada vértice una coordenada de textura vinculada a su oclusión ambiental.
- Algunas aplicaciones permiten generar automáticamente estas texturas a partir del diseño 3D de los objetos.