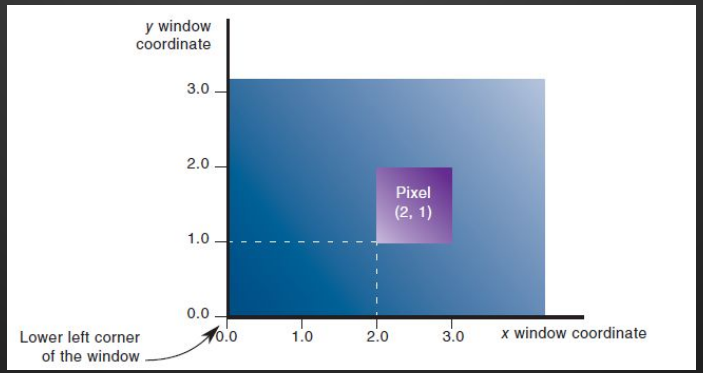
OpenGL

Técnicas avanzadas

Buffers

- El objetivo principal de OpenGL es dibujar las imágenes generadas en la pantalla (o fuera de la pantalla).
- Un framebuffer es una tabla rectangular de píxeles
- Después de la fase de rasterización, cuando se realiza el fragment shader, los datos todavía no son pixeles son ***fragmentos***. Cada fragmento tiene *coordenadas* que corresponden a un pixel. contiene también un *color* y una información de *profundidad.*

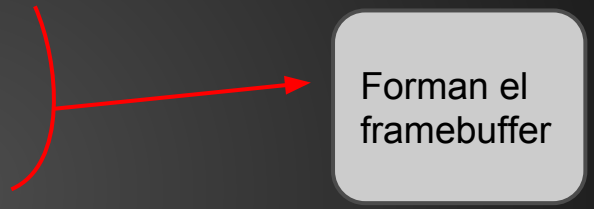


Buffers

- Ejemplo : Una pantalla de 1920 pixeles de ancho y 1080 pixeles de alto y de 24 bits por color (16,777,216 colores). (24 bits = 3 bytes). Un buffer de colores contiene al menos 3 bytes por pixel, de los 2,073,600 píxeles de la pantalla (1920 * 1080).
- El buffer de color es solamente uno de los buffers que contiene la información sobre un pixel.
- La información sobre el color de un pixel es incluso contenido en varios buffers de colores : *render buffers*)
- Un framebuffer contiene todo estos buffers y podemos utilizar varios framebuffers.

Buffers

- En un sistema de OpenGL disponemos de estos buffers
  - Color buffers, uno o más activos
  - Depth buffer (profundidad)
  - Stencil buffer



- Al iniciar una aplicación OpenGL se utiliza el framebuffer por defecto de un sistema de ventana que contiene un double-buffered color buffer.

Buffers

Color buffer

- Es el buffer en el cual generalmente terminamos dibujando
- Contiene los datos RGB y también puede contener el componente Alpha por cada pixel del framebuffer
- Puede haber varios color buffers (color buffer “on-screen” / color buffers “off-screen”)
- Generalmente almacena un color por píxel, también podemos dividir un pixel en ***subpixel*** para realizar sistemas de antialiasing (multi-sampling).
- Double buffered : se divide el color buffer del sistema de ventana en dos buffers : un front-buffer y un back-buffer
- Estereoscopia : encima dividimos el color buffer en dos buffers derecha y buffer izquierda. (4 partes : FL-BL/FR-BR)

Buffers

Depth buffer

- Almacena el valor de profundidad de cada pixel
- La profundidad es calculada en función a la distancia de pixel hacia la cámara.
- Utiliza el valor de la profundidad para determinar la visibilidad de los objetos tridimensionales en la escena.
    - Los fragmentos que contienen una profundidad más grande que el valor almacenado en el depth buffer son descartados del color buffer
    - Si se encuentra un fragmento con una distancia inferior se pone al día el depth buffer con este valor
- Depth buffer es también llamado Z-buffer ya que ‘x’ y ‘y’ corresponden al desplazamiento horizontal y vertical en la pantalla y ‘z’ a la profundidad perpendicular a la pantalla.

Buffers

Stencil buffer

- Utilizado para restringir el dibujo el algunas partes de la pantalla
- Ejemplo : Se renderiza en el stencil buffer la forma del retrovisor de un carro. Después, se renderiza toda la escena trasera. El stencil buffer asegura que solamente visualizamos las partes reflejadas por el retrovisor.

Frame Buffer Object - FBO

- Renderizar fuera de la pantalla (“off-screen rendering”)
- Útil para la realización de GPGPU con shaders….
- El framebuffer otorgado por el sistema de ventana (Glut, Qt, GLFW….) es el único que visualizamos en la pantalla.
  - Los FBO que creamos solo sirven para renderizar la escena fuera de la pantalla.
- Cuando se crea una ventana, el sistema de ventana crea los buffers : color, depth, stencil.
  - Cuando creamos un FBO, necesitamos también crear los otros buffers porque nunca se puede asociar un FBO con los buffers manejados por la ventana.
    - *void glGenFramebuffers(GLsizei n, GLuint *ids);*

Frame Buffer Object - FBO

Render Buffer es un objeto que vamos a linkear con un FBO :

- *void glGenRenderbuffers(GLsizei n, GLuint *ids);*
- *void glBindRenderbuffer(GL_RENDERBUFFER, name)*
- *glRenderbufferStorage(GL_RENDERBUFFER, internalFormat, w,h)*
  - Sirve para asignar la memoria del buffer según los tipos de datos que usaremos en este buffer : *internalFormat*

GL_R32F	GL_RG16F	GL_RG32F
GL_RGB16F	GL_RGB32F	GL_RGBA16F
GL_RGBA32F	GL_R11F_G11F_B10F	GL_RGB9_E5
GL_R8I	GL_R8UI	GL_R16I
GL_R16UI	GL_R32I	GL_R32UI
GL_RG8I	GL_RG8UI	GL_RG16I
GL_RG16UI	GL_RG32I	GL_RG32UI
GL_RGB8I	GL_RGB8UI	GL_RGB16I
GL_RGB16UI	GL_RGB32I	GL_RGB32UI
GL_RGBA8I	GL_RGBA8UI	GL_RGBA16I
GL_RGBA16UI	GL_RGBA32I	GL_R8_SNORM
GL_R16_SNORM	GL_RG8_SNORM	GL_RG16_SNORM
GL_RGB8_SNORM	GL_RGB16_SNORM	GL_RGBA8_SNORM
GL_RGBA16_SNORM		

To use a renderbuffer as a depth buffer, it must be depth-renderable, which is specified by setting internalformat to either GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT16, GL_DEPTH_COMPONENT32, GL_DEPTH_COMPONENT32, or GL_DEPTH_COMPONENT32F.

For use exclusively as a stencil buffer, internalformat should be specified as either GL_STENCIL_INDEX, GL_STENCIL_INDEX1, GL_STENCIL_INDEX4, GL_STENCIL_INDEX8, or GL_STENCIL_INDEX16.

Frame Buffer Object - FBO

Linkear el renderbuffer con el FBO :

```
glFramebufferRenderbuffer(target, attachment, renderBufferTarget , renderbuffer);
```

ejemplo :

```
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthRB);
```

Render to texture

- Crear un FBO
- Crear una textura en la cual vamos a renderizar la escena.
- Crear un depth buffer para el FBO
- Configurar el FBO
- Renderizar en la textura
- Hacer algo con la textura

Render to texture

- Crear un FBO

```
GLuint FramebufferName = 0;
glGenFramebuffers(1, &FramebufferName);
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
```

- Crear una textura en la cual vamos a renderizar la escena.

```
GLuint renderedTexture;
glGenTextures(1, &renderedTexture);

// "Bind" the newly created texture : all future texture functions will modify this texture
glBindTexture(GL_TEXTURE_2D, renderedTexture);

// Give an empty image to OpenGL ( the last "0" )
glTexImage2D(GL_TEXTURE_2D, 0,GL_RGB, 1024, 768, 0,GL_RGB, GL_UNSIGNED_BYTE, 0);


glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
```

Render to texture

- Crear un depth buffer para el FBO

```
// The depth buffer
GLuint depthrenderbuffer;
glGenRenderbuffers(1, &depthrenderbuffer);
glBindRenderbuffer(GL_RENDERBUFFER, depthrenderbuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, 1024, 768);
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthrenderbuffer);
```

- Configurar el FBO

```
// Set "renderedTexture" as our colour attachement #0
glFramebufferTexture(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, renderedTexture, 0);

// Set the list of draw buffers.
GLenum DrawBuffers[1] = {GL_COLOR_ATTACHMENT0};
glDrawBuffers(1, DrawBuffers); // "1" is the size of DrawBuffers
```

Render to texture

- Renderizar en la textura

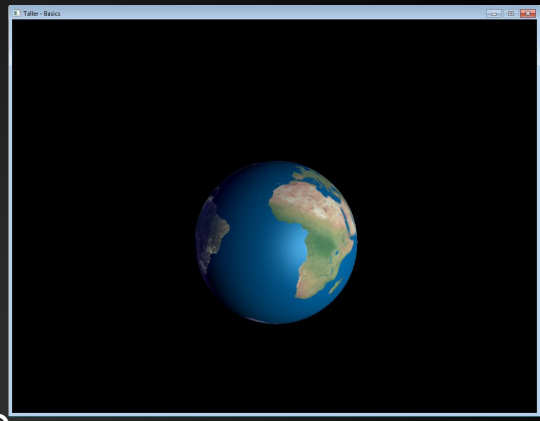
```
glBindFramebuffer(GL_FRAMEBUFFER, FramebufferName);
glViewport(0, 0, 1024, 768);
// Clear the screen
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
//Dibujar escena
```

. . . . .

- Hacer algo con la textura

```
glBindFramebuffer(GL_FRAMEBUFFER, 0);
glViewport(0, 0, 1024, 768);
// Clear the screen
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
//Dibujar utilizando la textura renderedTexture
```

Render to texture



- Una vez que tenemos la textura que representa nuestra escena. Podemos Mapear esta textura sobre un cuadrilátero y visualizarlo sobre toda la superficie de la pantalla :

```
static const GLfloat g_quad_vertex_buffer_data[] = {
        -1.0f, -1.0f, 0.0f,
        1.0f, -1.0f, 0.0f,
        -1.0f, 1.0f, 0.0f,
        -1.0f, 1.0f, 0.0f,
        1.0f, -1.0f, 0.0f,
        1.0f, 1.0f, 0.0f,
};
```

Vertex Shader

```
layout(location = 0) in vec3 vertexPosition_modelspace;

out vec2 UV;

void main(){
        gl_Position =  vec4(vertexPosition_modelspace, 1);
        UV = (vertexPosition_modelspace.xy+vec2(1,1))/2.0;
}
```

Render to texture

- Que pasa si lo aplico al mismo cuadrilátero visto con perspectiva

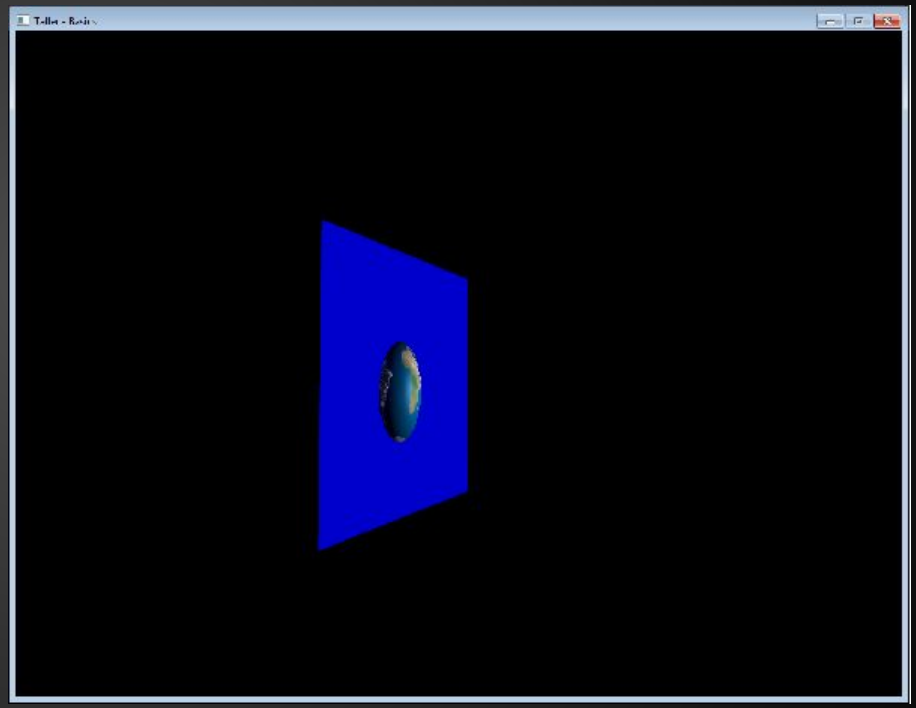
```
static const GLfloat g_quad_vertex_buffer_data[] = {
        -1.0f, -1.0f, 0.0f,
        1.0f, -1.0f, 0.0f,
        -1.0f, 1.0f, 0.0f,
        -1.0f, 1.0f, 0.0f,
        1.0f, -1.0f, 0.0f,
        1.0f, 1.0f, 0.0f,
};
```

Vertex Shader

```
layout(location = 0) in vec3 vertexPosition_modelspace;

out vec2 UV;
uniform mat4 MVP;
void main(){
        gl_Position =  MVP * vec4(vertexPosition_modelspace, 1);
        UV = (vertexPosition_modelspace.xy+vec2(1,1))/2.0;
}
```

Render to texture

- Ideal para simular pantallas virtuales cuyo contenido es calculado en tiempo real.



Render to Texture

- Podemos visualizar el contenido del z-buffer ?
- Si, podemos generar una textura y linkearla al FBO pidiéndole que renderize el z-buffer dentro

```
GLuint depthTexture;
glGenTextures(1, &depthTexture);
glBindTexture(GL_TEXTURE_2D, depthTexture);
glTexImage2D(GL_TEXTURE_2D, 0,GL_DEPTH_COMPONENT24, 1024, 768, 0,GL_DEPTH_COMPONENT, GL_FLOAT,
0);
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_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthTexture, 0);
```

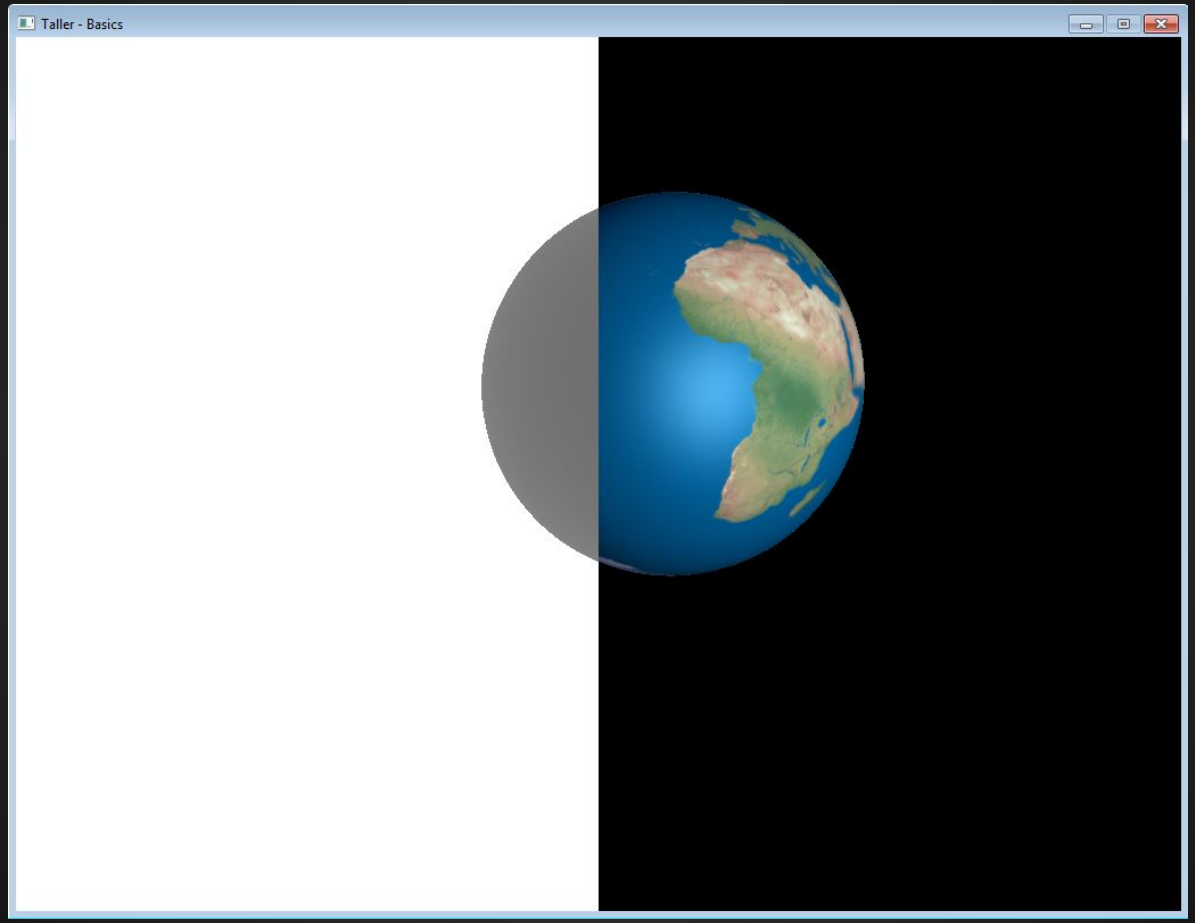
Render to texture

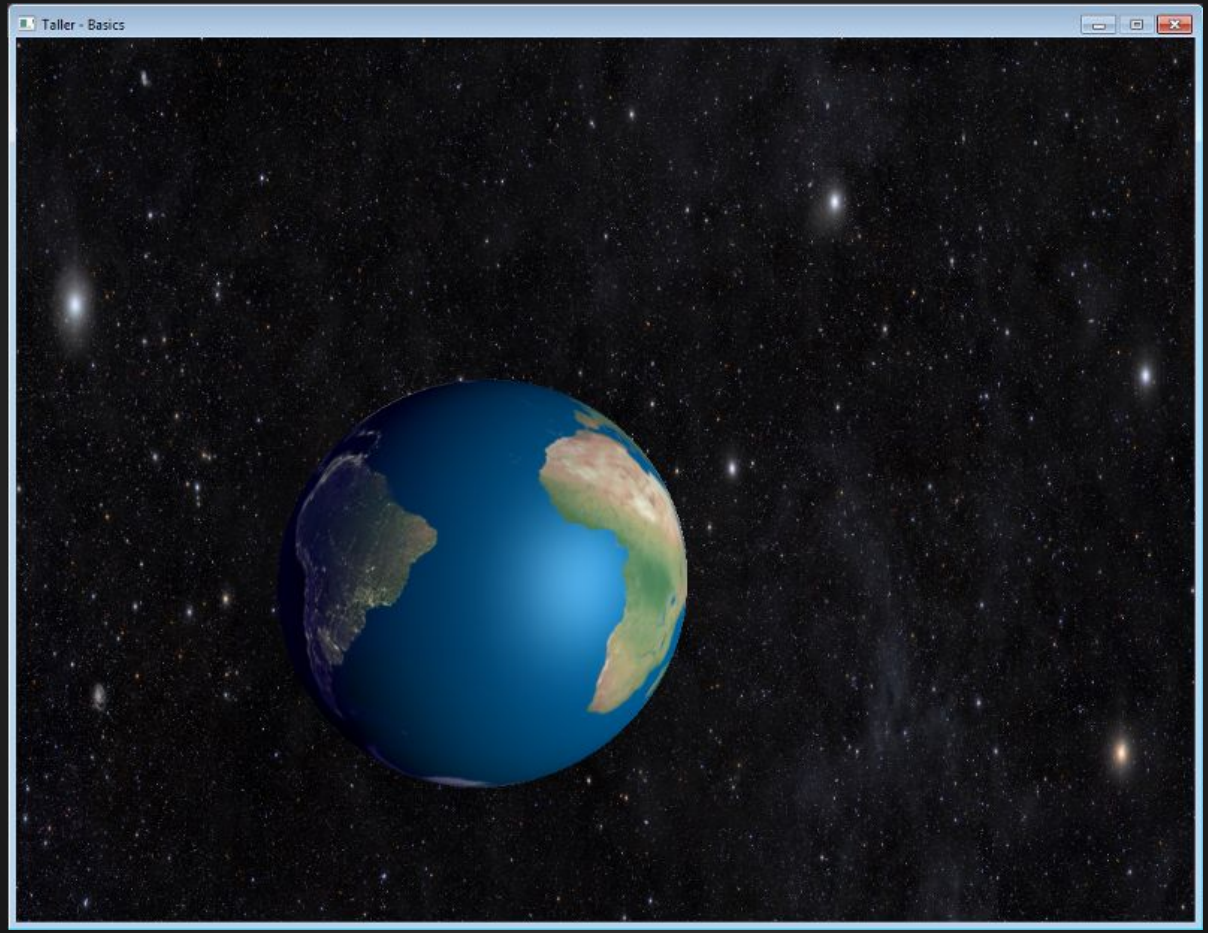
- Queremos visualizar mitad z-buffer mitad color buffer

```
static const GLfloat g_quad_vertex_buffer_data[] = {
        -1.0f, -1.0f, 0.0f,
        1.0f, -1.0f, 0.0f,
        -1.0f, 1.0f, 0.0f,
        -1.0f, 1.0f, 0.0f,
        1.0f, -1.0f, 0.0f,
        1.0f, 1.0f, 0.0f,
};
```

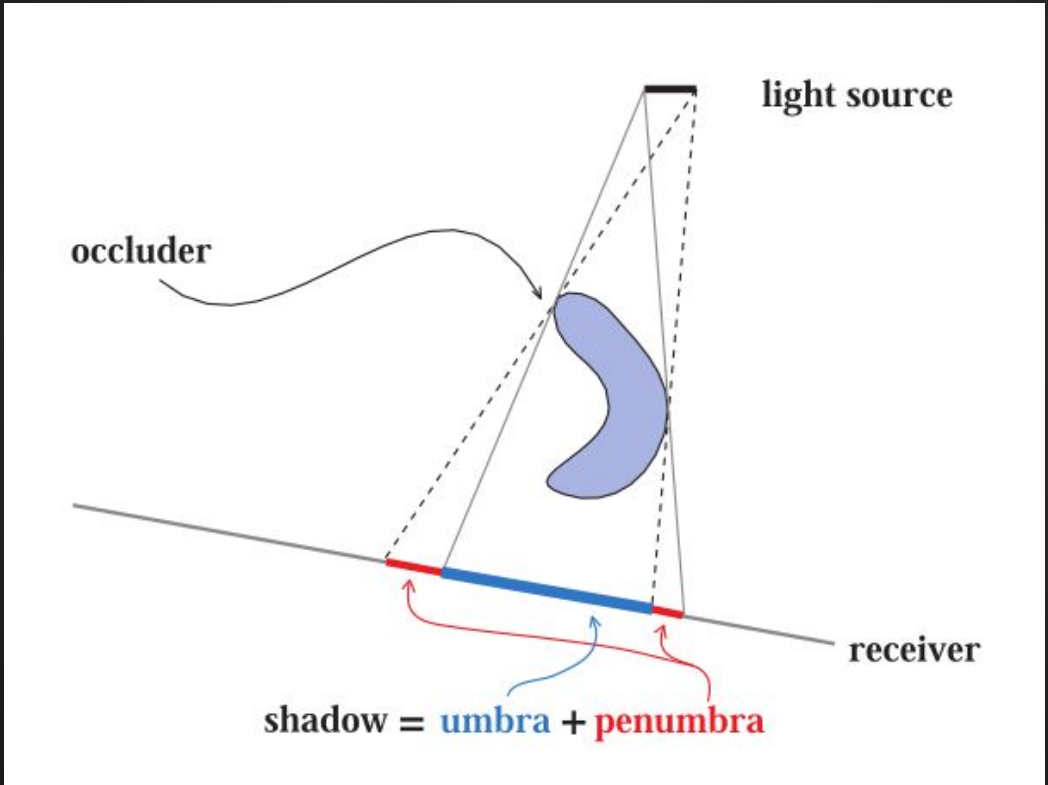
Fragment Shader

```
in vec2 UV;

out vec3 color;

uniform sampler2D renderedTexture;
uniform sampler2D depthTexture;
void main(){

        float z = texture(depthTexture, UV).r;      // fetch the z-value from
our depth texture
        float n = 0.1;                              // the near plane
        float f = 100.0;                              // the far plane
        float c = (2.0 * n) / (f + n - z * (f - n));  // convert to linear values
        color = vec3(c);
        if(UV.x>=0.5)
                color=texture( renderedTexture, UV ).rgb;

}
```


Taller - Basics


Taller - Basics

Shadows




15
00
2

Shadows

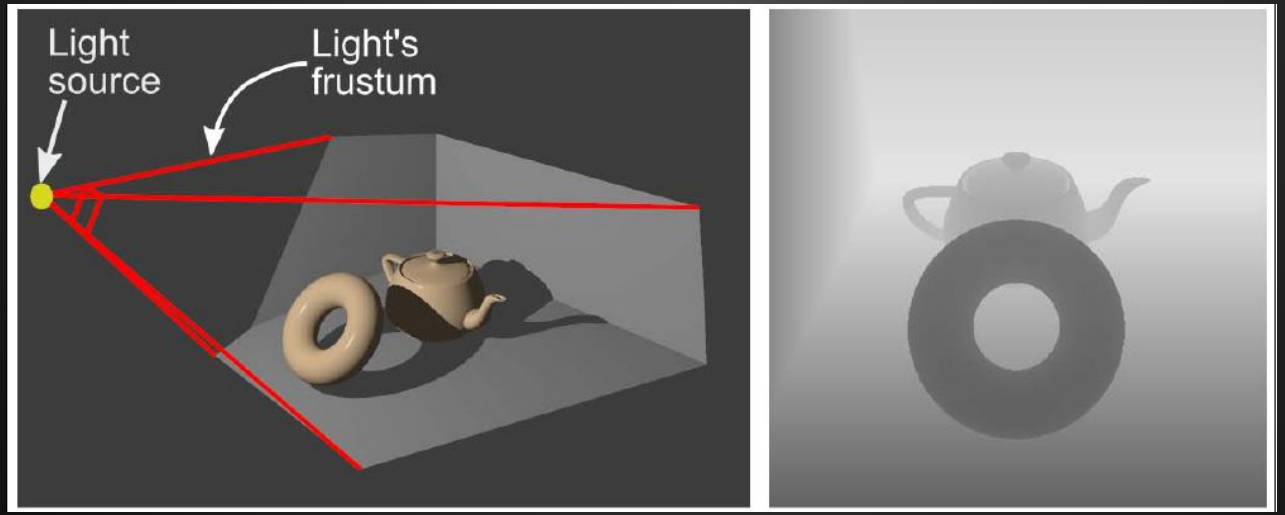
- Shadow mapping
- Técnica que consiste en estudiar la escena desde el punto de vista de la luz que producirá las sombras.
- Utilizaremos el z-buffer para determinar si un punto del espacio está iluminado o no.
- Renderizamos el z-buffer en una textura desde el punto de vista de la luz : *shadow map*
- Los puntos visibles por la luz serán renderizados en el depth buffer y los que no cumplen el test de profundidad serán descartados.

Shadows

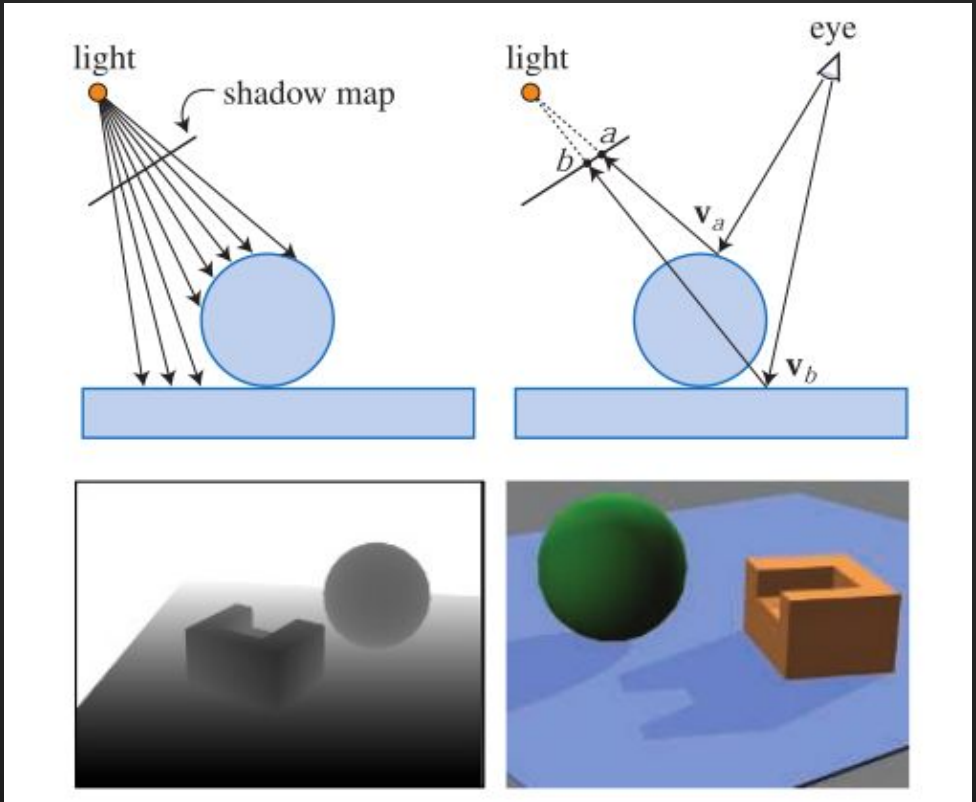
- Dos etapas
  - Renderizar la escena desde le punto de vista de la luz, solo queremos los valores de profundidad desde la luz. Así que necesitamos crear un FBO linkeando una textura de profundidad donde renderizamos la escena.
  - Renderizar la escena desde el punto de vista del usuario. Los fragmentos que quedan más lejos que los valores de la textura de profundidad son en la sombra.

Shadows

- Renderizar la escena desde le punto de vista de la luz, solo queremos los valores de profundidad desde la luz. Así que necesitamos crear un FBO linkeando una textura de profundidad donde renderizamos la escena.



Shadows



Shadows

- Renderizar la escena desde le punto de vista de la luz, solo queremos los valores de profundidad desde la luz. Así que necesitamos crear un FBO linkeando una textura de profundidad donde renderizamos la escena.

Alistar la matriz MVP para mandar al shader

```
glm::vec3 lightInvDir = glm::vec3(0.5f, 2, 2);
// Compute the MVP matrix from the light's point of view
glm::mat4 depthProjectionMatrix = glm::ortho<float>(-10, 10, -10, 10, -10, 20);
glm::mat4 depthViewMatrix = glm::lookAt(lightInvDir, glm::vec3(0, 0, 0), glm::vec3(0, 1, 0));

glm::mat4 depthModelMatrix = glm::mat4(1.0);
glm::mat4 depthMVP = depthProjectionMatrix * depthViewMatrix * depthModelMatrix;

// Send our transformation to the currently bound shader,
// in the "MVP" uniform
glUniformMatrix4fv(depthMatrixID, 1, GL_FALSE, &depthMVP[0][0]);
```

Shadows

- Renderizar la escena desde le punto de vista de la luz, solo queremos los valores de profundidad desde la luz. Así que necesitamos crear un FBO linkeando una textura de profundidad donde renderizamos la escena.

vertex Shader

```
// Input vertex data, different for all executions of this shader.
layout(location = 0) in vec3 vertexPosition_modelspace;

// Values that stay constant for the whole mesh.
uniform mat4 depthMVP;

void main(){
	gl_Position =  depthMVP * vec4(vertexPosition_modelspace,1);
}
```

Shadows mapping

- Renderizar la escena desde el punto de vista del usuario. Los fragmentos que queden más lejos que los valores de la textura de profundidad están en la sombra.

Esta parte es lo único que cambia de un pass through shader. Lo que hacemos es recuperar la coordenada del vértice actual desde el punto de vista de la luz. El “Bias” es una matriz que manda las coordenadas [-1,1] hasta [0,1] que corresponden a la textura del z-buffer

```
layout(location = 0) in vec3 vertexPosition_modelspace;
layout(location = 1) in vec2 vertexUV;
layout(location = 2) in vec3 vertexNormal_modelspace;

out vec2 UV;
out vec4 ShadowCoord;
// Values that stay constant for the whole mesh.
uniform mat4 MVP;
uniform mat4 DepthBiasMVP;
void main(){
        // Output position of the vertex, in clip space : MVP * position
        gl_Position =  MVP * vec4(vertexPosition_modelspace,1);
        ShadowCoord = DepthBiasMVP *vec4(vertexPosition_modelspace,1);

        // UV of the vertex. No special space for this one.
        UV = vertexUV;
}
```

Shadows mapping

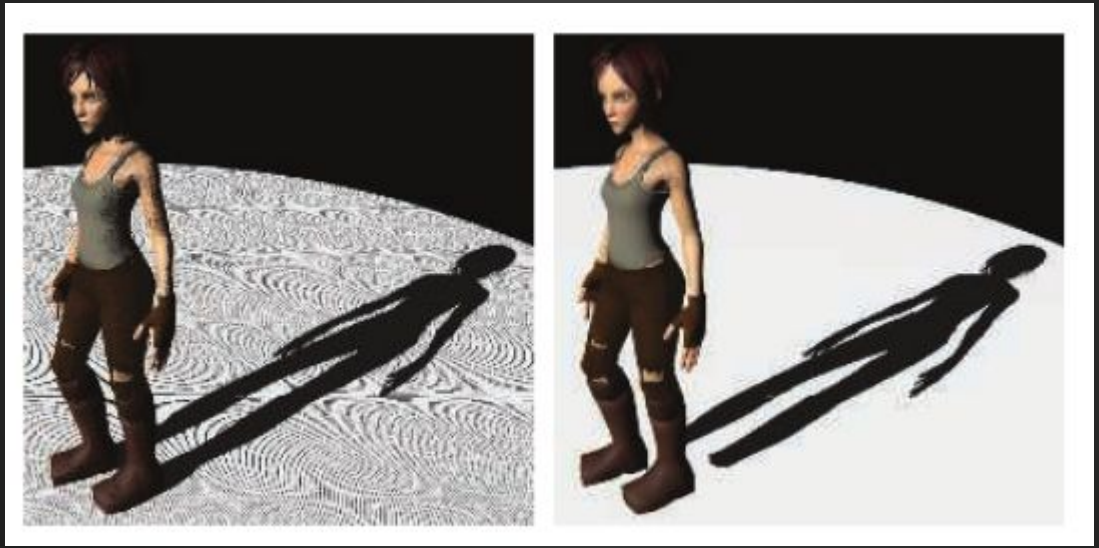
- Renderizar la escena desde el punto de vista del usuario. Los fragmentos que quedan más lejos que los valores de la textura de profundidad son en la sombra.

texture(shadowMap, ShadowCoord.xy).z es la distancia entre el entre la luz y el primer obstaculo

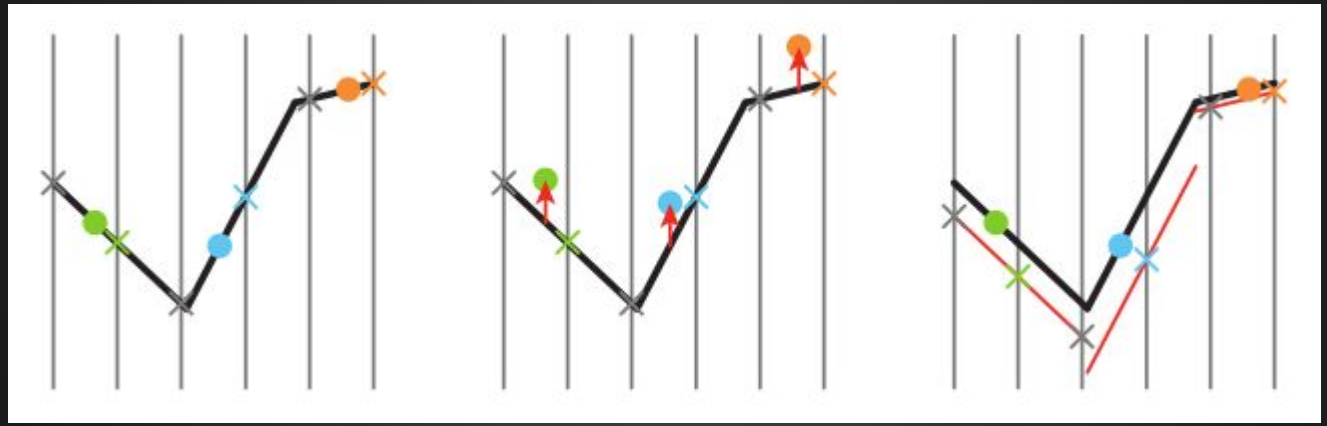
ShadowCoord.z es la distancia entre la luz y el fragmento actual.

```
in vec2 UV;
in vec4 ShadowCoord;

layout(location = 0) out vec3 color;
uniform sampler2D myTextureSampler;
uniform sampler2DShadow shadowMap;
void main(){

    vec3 LightColor = vec3(1,1,1);
    vec3 MaterialDiffuseColor = texture2D( myTextureSampler, UV ).rgb;
     float visibility = 1.0;

     if ( texture( shadowMap, ShadowCoord.xy ).z  <  ShadowCoord.z)
              visibility = 0.5;

       color = visibility * MaterialDiffuseColor * LightColor;

}
```

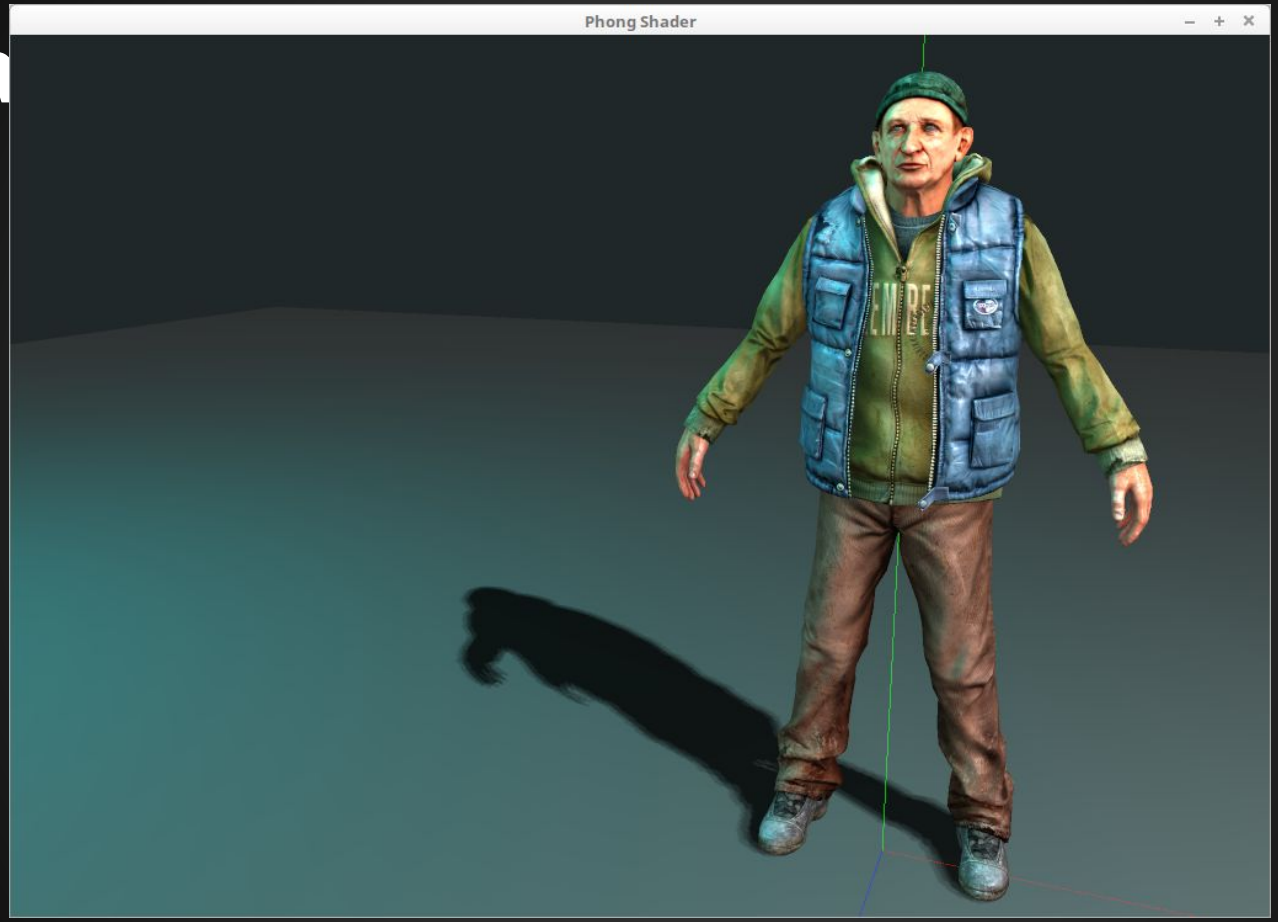
Shadow Acne

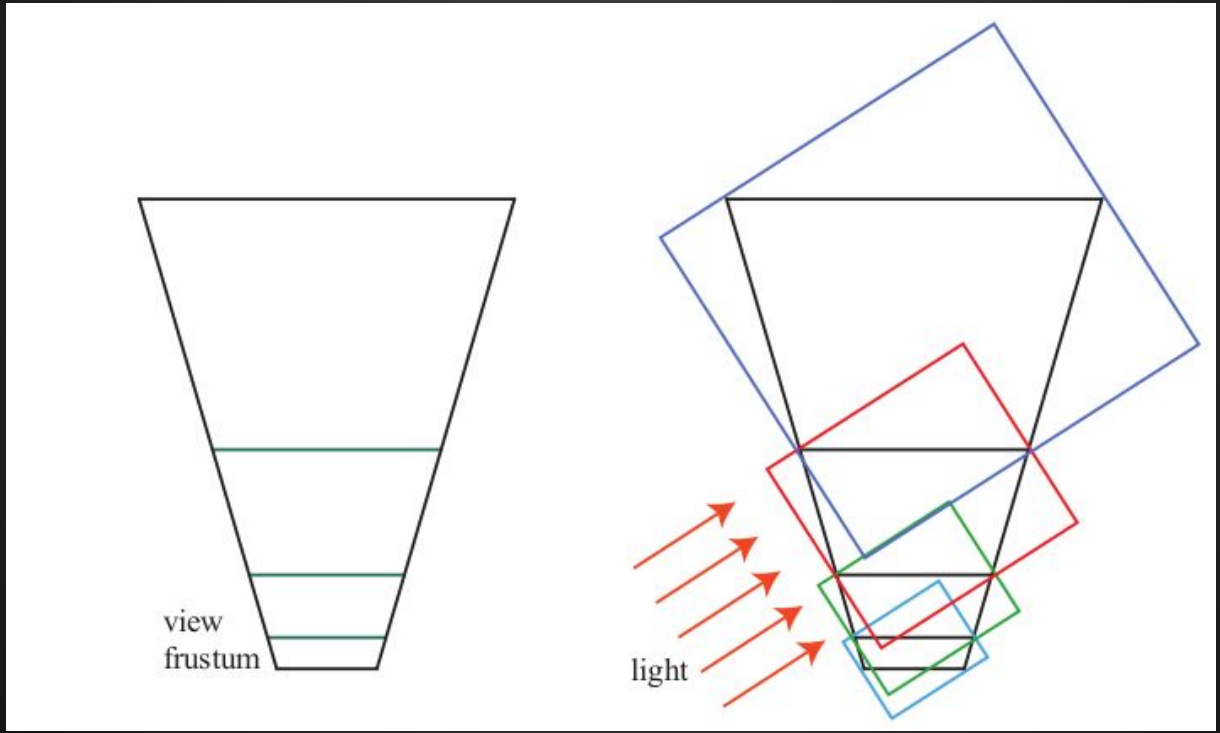


Shadow Acne -> Bias

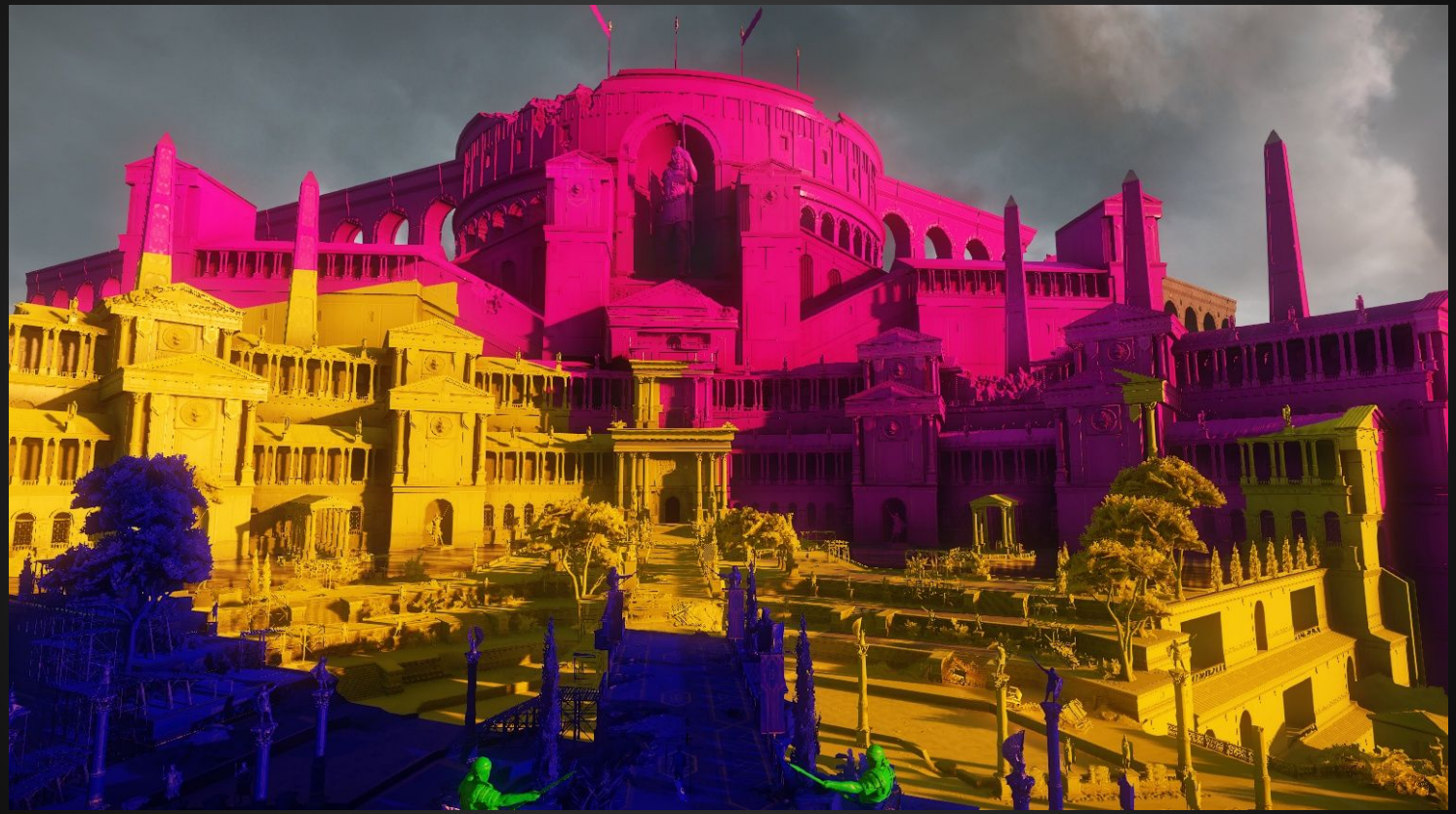


Sha

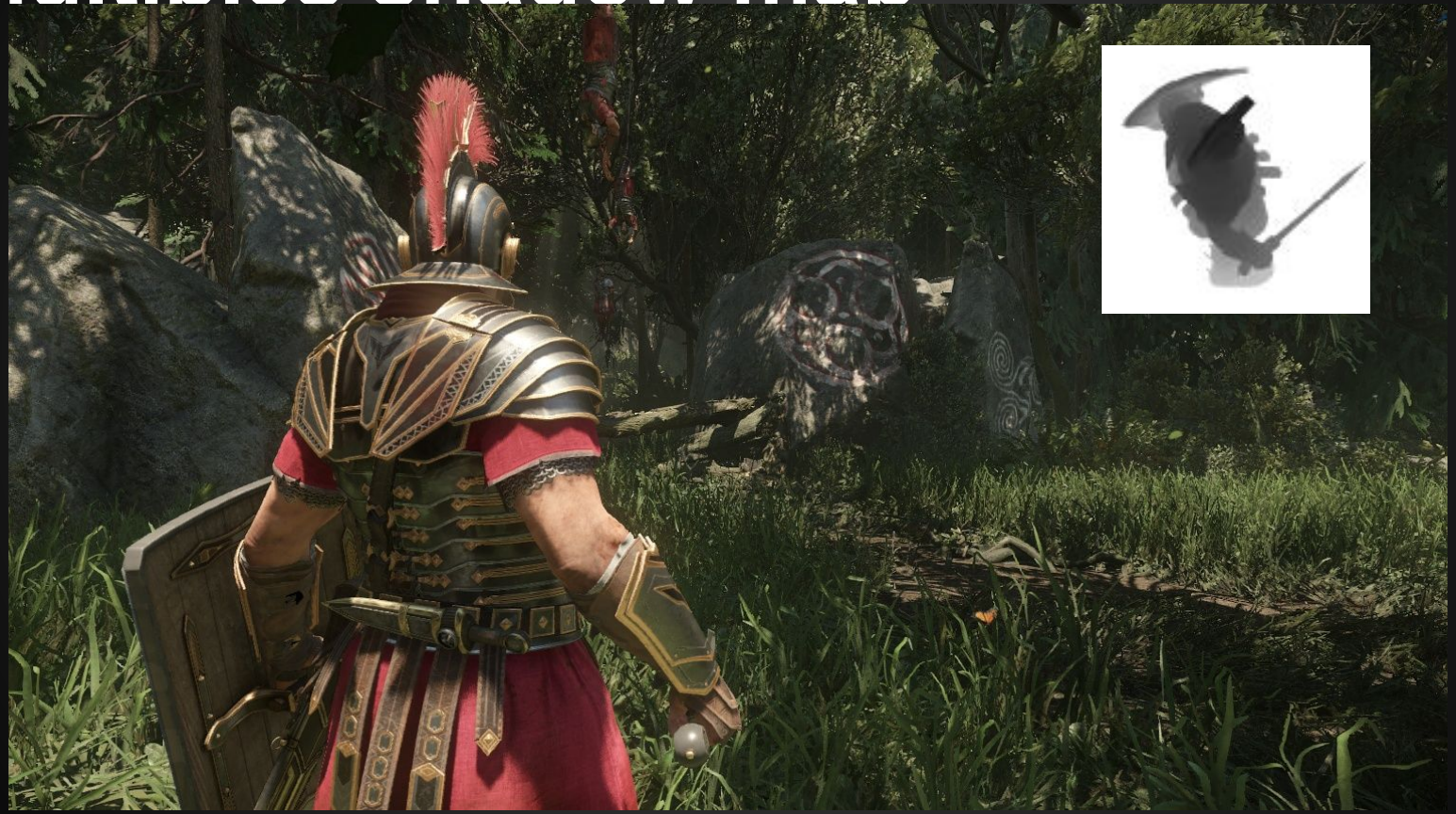



view
frustum
light

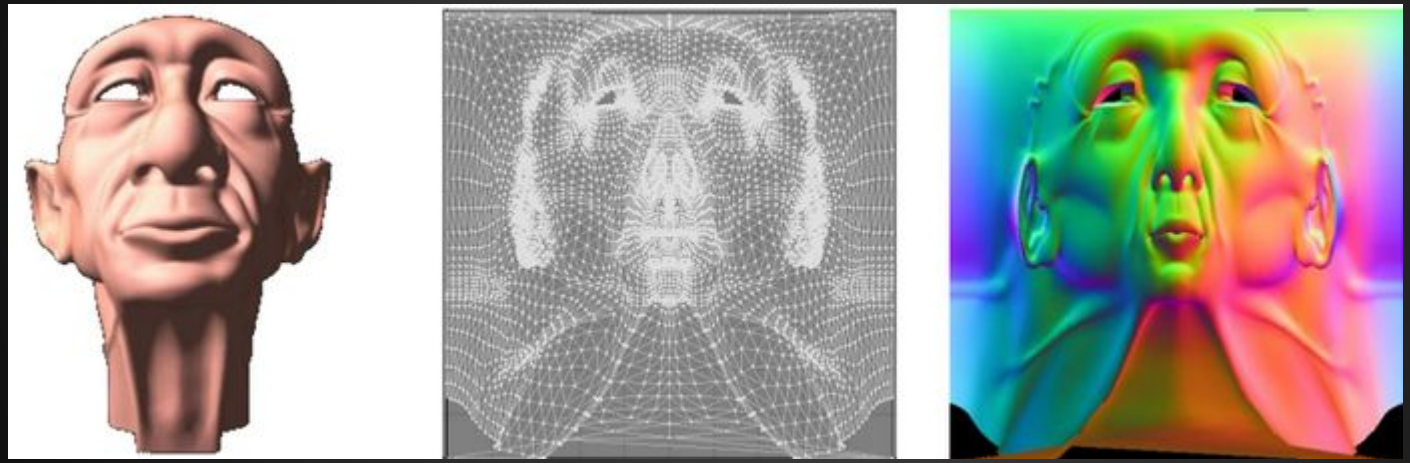
Multiples shadow map



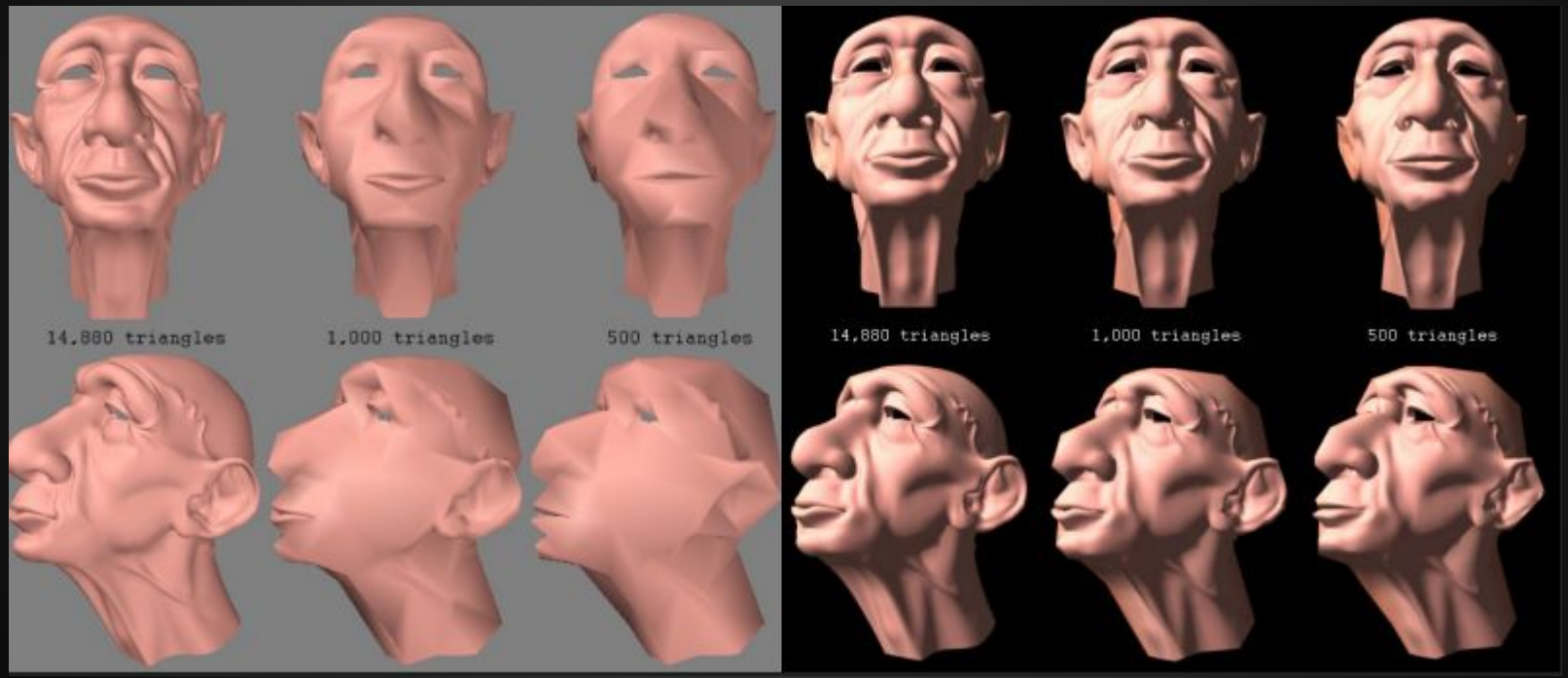
Multiples shadow map



Normal Mapping

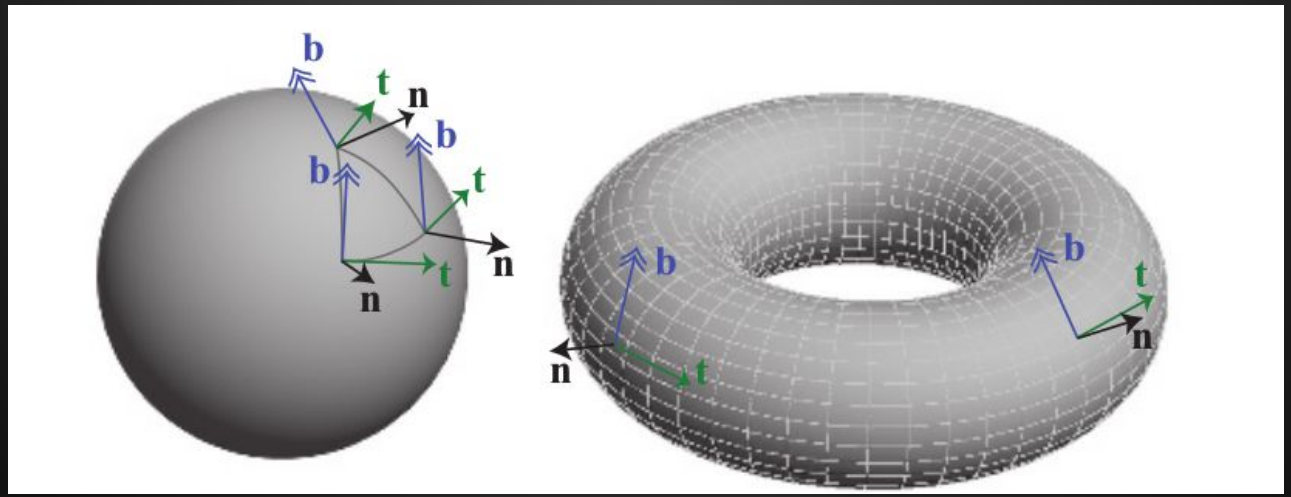


Normal Mapping



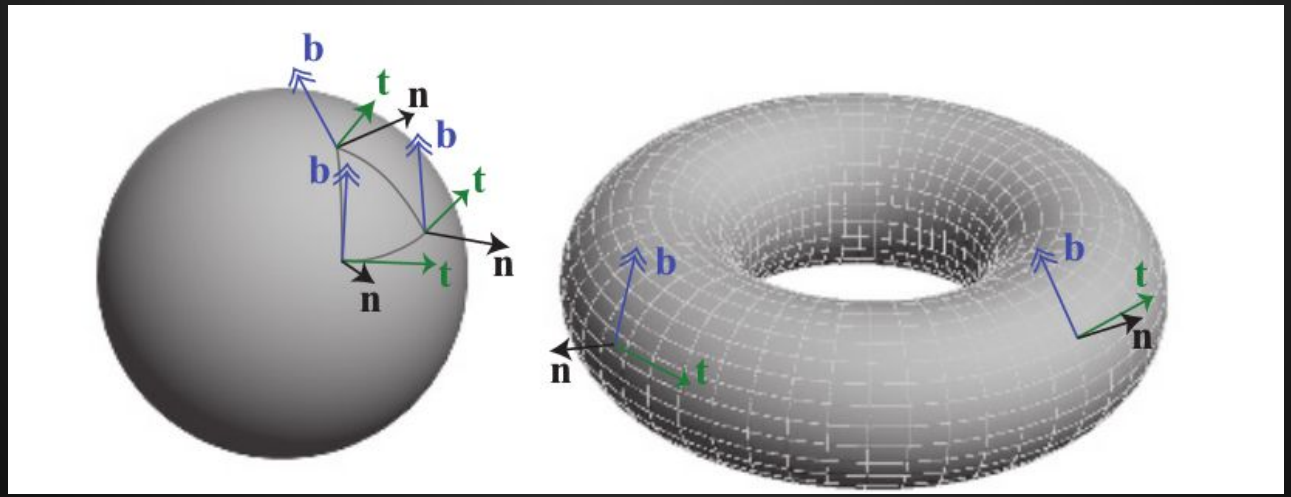
Normal Mapping

- Necesitamos la tangent y bitangente
- Ya que no podemos dar directamente la normal de la textura



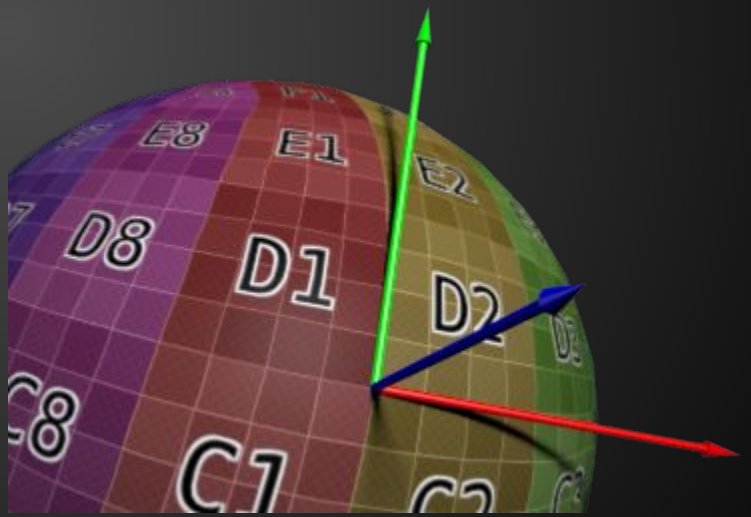
Normal Mapping

- Necesitamos la tangent y bitangente
- Ya que no podemos dar directamente la normal de la textura



Normal Mapping

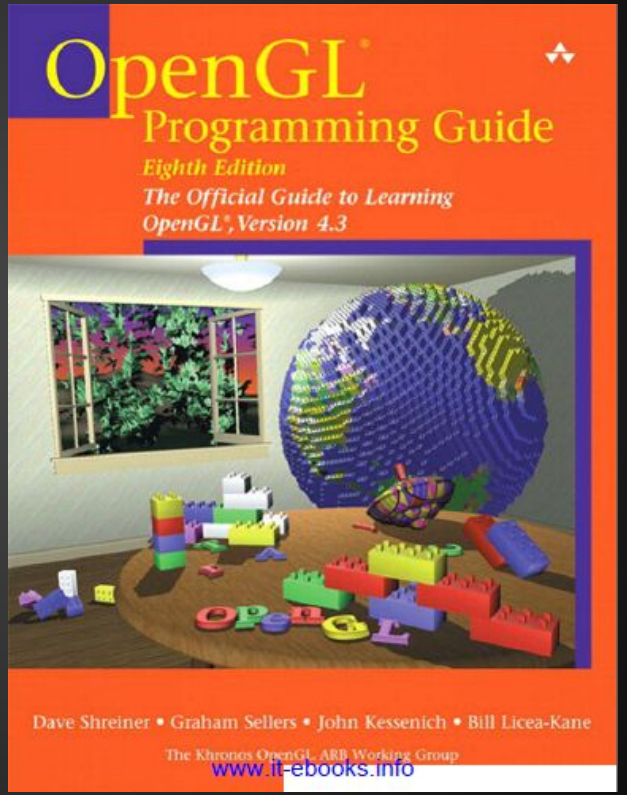
- Necesitamos la tangent y bitangente
- Ya que no podemos dar directamente la normal de la textura



Normal Mapping

- Calcular phong ya no desde el punto de vista sino desde el espacio tangente

Libros



http://www.opengl-tutorial.org/