



Sombras



<http://flickr.com/photos/tquinn>

Bibliografía:

- Superbiblia 7ª ed, cap 9 y pp. 599-605
- Real-Time Rendering, 3ª ed. 9.1



Preliminares: *framebuffer objects*

- OpenGL permite crear framebuffers que no están asociados a ninguna ventana del S.O.
- Así, no están limitados al tamaño de una ventana real.
- Hay varias técnicas que necesitan dibujar fuera de la pantalla (por ejemplo, para almacenar un fondo que no cambia mucho, reflejos dinámicos, etc.)
- Pueden contener varios buffers de color.
- Se le puede asociar una textura, que luego se puede usar para dibujar la escena



Framebuffer Objects (FBO)

- Los FBO son contenedores de estado, y se le asocian buffers, que son en los que realmente se dibujará
- Uso de FBO:
 - **void glGenFramebuffers(GLsizei n, GLuint *ids)**
 - Generar identificadores (nombres)
 - **void glBindFramebuffer(GLenum target, GLuint id)**
 - **target:** GL_DRAW_FRAMEBUFFER (para dibujar en él), GL_READ_FRAMEBUFFER (para leer de él), GL_FRAMEBUFFER (ambos)
 - **id:** identificador del framebuffer a vincular, o 0 para vincular el framebuffer por defecto
 - **void glDeleteFramebuffers(sizei n, const uint *ids)**



Framebuffer Objects (FBO)

Renderbuffer objects

- Los renderbuffer son los buffers donde realmente se dibuja. Para usarlos hay que vincularlos a un FBO (que puede tener varios RBO)
- Varios tipos:
 - color, profundidad, stencil o una combinación de los dos últimos
- Uso:
 - `void glGenRenderbuffers(GLsizei n, GLuint *rbos)`
 - `void glBindRenderbuffer(GLenum target, GLuint id)`
 - target: GL_RENDERBUFFER
 - `void glDeleteRenderbuffers(GLsizei n, GLuint *rbos)`



Framebuffer Objects (FBO)

Renderbuffer objects

- Después de definir el RBO activo, hay que asignarle memoria:
 - **void glRenderbufferStorage(GLenum target, GLenum internalformat, GLsizei width, GLsizei height)**
 - **target:** GL_RENDERBUFFER
 - **internalformat:** GL_RED, GL_RG, GL_RGB, GL_RGBA, GL_DEPTH_COMPONENT, GL_DEPTH_STENCIL...¹
 - **width, height:** tamaño en píxeles del buffer. Tamaño máximo consultable con glGetIntegerv y GL_MAX_RENDERBUFFER_SIZE (32768 desde GeForce GTX 1060)

¹ Ver el apartado 9.4 de la especificación de OpenGL 4.6

A partir de GL 4.5

```
void glNamedRenderbufferStorage(GLuint renderbuffer, GLenum internalformat,  
GLsizei width, GLsizei height)
```



Framebuffer Objects (FBO)

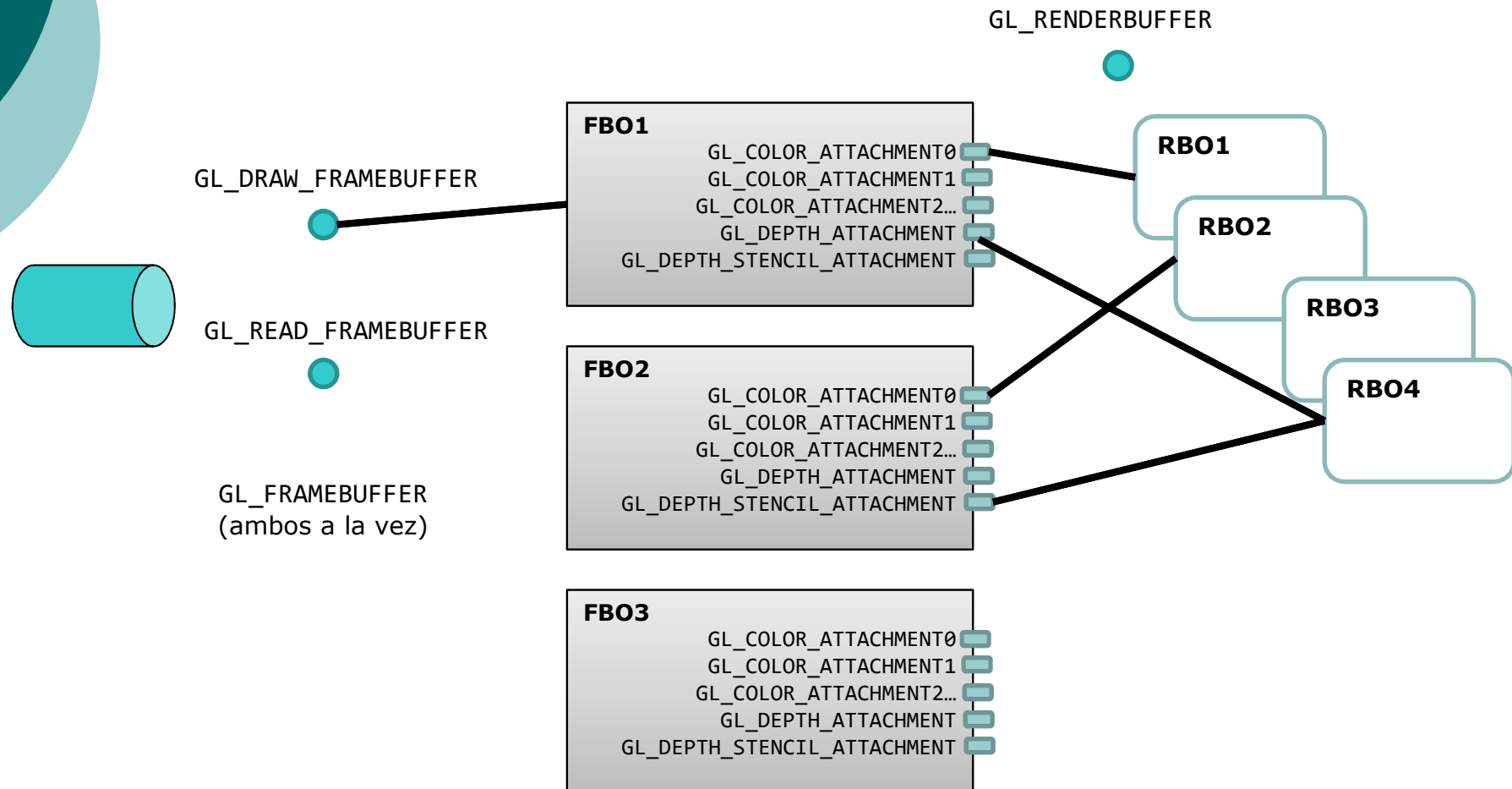
Renderbuffer objects

- El siguiente paso es asociar los RBO con el FBO
 - Un FBO tiene varios puntos de vinculación para conectar RBO: uno de profundidad, otro de stencil y varios de color (hasta GL_MAX_COLOR_ATTACHMENTS) [normalmente 8]
 - El FBO tiene que estar vinculado (p.e. a GL_DRAW_FRAMEBUFFER)
 - **void glFramebufferRenderbuffer(GLenum target, GLenum attachment, GLenum renderbuffertarget, GLuint renderbuffer)**
 - **target:** GL_{DRAW,READ}_FRAMEBUFFER
 - **attachment:** GL_COLOR_ATTACHMENTi, GL_DEPTH_ATTACHMENT, GL_STENCIL_ATTACHMENT, GL_DEPTH_STENCIL_ATTACHMENT
 - **renderbuffertarget:** GL_RENDERBUFFER
 - **renderbuffer:** identificador del buffer a vincular, o 0 para desvincular el buffer actual

A partir de GL 4.5

```
void glNamedFramebufferRenderbuffer(GLuint framebuffer, GLenum attachment,
    GLenum renderbuffertarget, GLuint renderbuffer);
```

Framebuffer Objects (FBO)





Framebuffer Objects (FBO)

- Se puede asociar a un FBO:
 - RBO de distinto formato de color
 - RBO de distinto tamaño (el efectivo será el más pequeño y se usa, por ejemplo, para compartir un buffer de profundidad entre varios FBO)
 - Un RBO que almacene simultáneamente el buffer de profundidad y el stencil (crearlo con `GL_DEPTH_STENCIL` y vincularlo a `GL_DEPTH_STENCIL_ATTACHMENT`)
 - Texturas



Framebuffer Objects (FBO)

- Para usar texturas en vez de RBO:
 - Crear las texturas y reservarles espacio normalmente
 - Crear el FBO como antes
 - Vincular las texturas al FBO con:
 - `void glFramebufferTexture2D(GLenum target, GLenum attachment, GLenum textarget, GLuint texture, GLint level)`
 - **target**: `GL_{DRAW,READ}_FRAMEBUFFER`
 - **attachment**: `GL_COLOR_ATTACHMENTi`, `GL_DEPTH_ATTACHMENT`, `GL_STENCIL_ATTACHMENT`, `GL_DEPTH_STENCIL_ATTACHMENT`
 - **textarget**: `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP_POSITIVE_X`, `GL_TEXTURE_CUBE_MAP_POSITIVE_Y...`
 - **texture**: identificador del objeto textura
 - **level**: nivel de mipmap de la textura

A partir de GL 4.5

```
void glNamedFramebufferTexture(GLuint framebuffer, GLenum attachment, GLuint texture, GLint level)
```



Framebuffer Objects (FBO)

- Antes de usar un FBO hay que comprobar que está completo. Dos tipos de completitud:
 - Completitud de adjuntos
 - Todos los RBO o texturas asociados deben estar completos (deben tener memoria asociada, su tamaño no debe ser cero, debe tener un formato compatible...)
 - Completitud de todo el framebuffer
 - Todos los buffers adjuntos deben estar completos, debe tener memoria asociada, la combinación de formatos internos debe ser compatible...
 - **GLenum glCheckFramebufferStatus(GLenum target)**
 - **target:** GL_{DRAW,READ}_FRAMEBUFFER
 - devuelve GL_FRAMEBUFFER_COMPLETE si el FBO está listo para usarse, o un código de error en caso contrario:
 - GL_FRAMEBUFFER_UNDEFINED,
GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT,
GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT... (ver lista completa y significados en el apartado 9.4.2 de la especificación de OpenGL 4.6)

A partir de GL 4.5

GLenum glCheckNamedFramebufferStatus(GLuint framebuffer, GLenum target)



Framebuffer Objects (FBO)

Dibujando a un FBO

- El shader de fragmento es responsable de generar el color del fragmento
 - Puede escribir a varios buffers de color
- Por defecto, si el shader genera únicamente un color, este se almacena en el RBO asociado a `GL_COLOR_ATTACHMENT0`
- En caso de tener más de un RBO de color, hay que especificar siempre qué salida del shader va a qué RBO



Framebuffer Objects (FBO)

Eligiendo el buffer para escribir

- OpenGL permite seleccionar en qué buffer (o buffers de color) se van a dibujar las siguientes primitivas
 - **`void glDrawBuffer(GLenum mode);`**
 - Para el FB por defecto:
 - `GL_NONE`, `GL_FRONT_LEFT`, `GL_FRONT_RIGHT`, `GL_BACK_LEFT`, `GL_BACK_RIGHT`, `GL_FRONT`, `GL_BACK`, `GL_LEFT`, `GL_RIGHT`, `GL_FRONT_AND_BACK`
 - Si se omite `LEFT` o `RIGHT`, hace referencia a ambos
 - Si se omite `FRONT` o `BACK`, hace referencia a ambos
 - P.e. `GL_LEFT`, activa el frontal y trasero izquierdo
 - Para un FBO de usuario
 - `GL_NONE`, `GL_COLOR_ATTACHMENTi`



Framebuffer Objects (FBO)

Eligiendo el buffer para escribir

- También se puede establecer varios buffers a la vez.
 - **void glDrawBuffers(GLsizei n, const GLenum *mode)**
 - **n** es el número de elementos en **mode**
 - Para el FB por defecto:
 - **mode** es un vector de constantes (sólo se aceptan GL_NONE, GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT)
 - Para un FBO de usuario:
 - **mode** sólo puede contener GL_NONE y GL_COLOR_ATTACHMENTi

Framebuffer Objects (FBO)

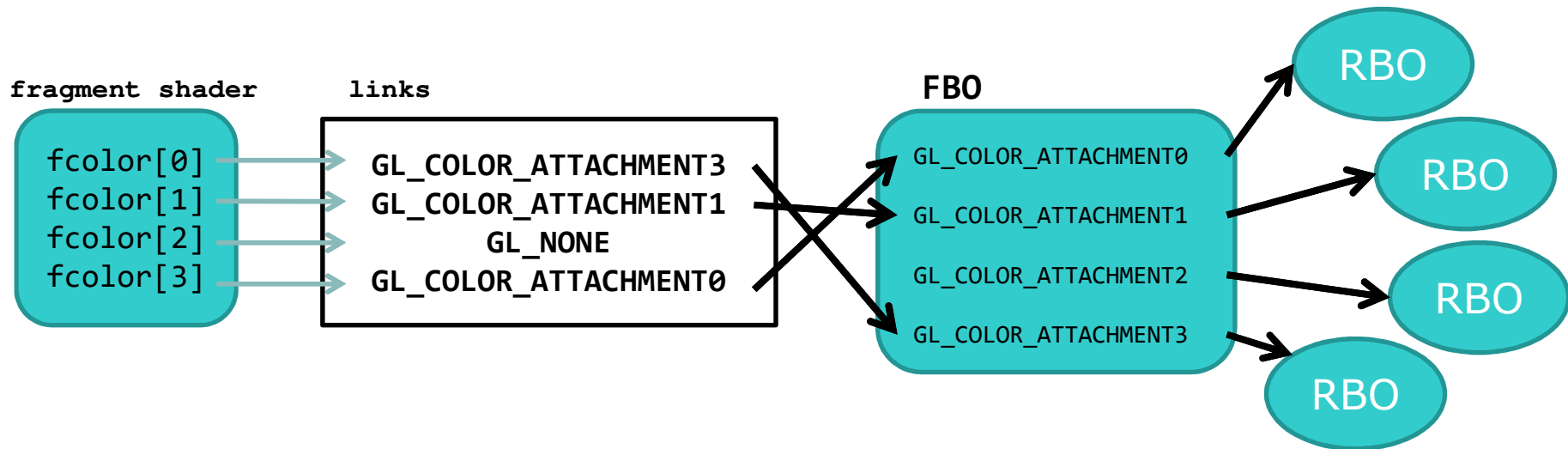
Eligiendo el buffer para escribir

- Relación entre salidas y RBO:

- `void glDrawBuffers(GLsizei n, GLenum *bufs)`

- Ejemplo:

```
GLenum links[] = {GL_COLOR_ATTACHMENT3, GL_COLOR_ATTACHMENT1,  
                  GL_NONE, GL_COLOR_ATTACHMENT0};  
glDrawBuffers(4, links);
```





Framebuffer Objects (FBO)

Eligiendo el buffer para escribir

- Si el shader de fragmento sólo genera un color de salida, se escribirá en todos los RBO de color activos.
- Hay un máximo de buffers a los que se puede escribir, consultable con `GL_MAX_DRAW_BUFFERS` (normalmente 8)



Framebuffer Objects (FBO)

- Se pueden activar/desactivar el scissor test y el blending por RBO:
 - `void glEnablei(enum target, uint index)`
 - `void glDisablei(enum target, uint index)`
 - Donde:
 - target: GL_SCISSOR_TEST, GL_BLEND
 - index: índice del buffer (dentro de los especificados en la llamada a `glDrawBuffers`)
- Y para el blending, además:
 - `glBlendEquationi, glBlendFunci, glBlendFuncSeparatei`



Framebuffer Objects (FBO)

Leyendo de un FBO

- Hay operaciones que permiten copiar de un buffer a otro dentro de la GPU
- Para establecer el buffer de lectura, se usa:
 - **void glReadBuffer(GLenum mode)**
 - Para el FBO por defecto, **mode**: GL_NONE, GL_FRONT_LEFT, GL_FRONT_RIGHT, GL_BACK_LEFT, GL_BACK_RIGHT, GL_FRONT, GL_BACK, GL_LEFT, GL_RIGHT, GL_FRONT_AND_BACK
 - Para un FBO de usuario, **mode**: GL_NONE, GL_COLOR_ATTACHMENTi
- Para leer de un buffer se usan las funciones:
 - **glReadPixels, glBlitFramebuffer, glCopyTexImage* y glCopyTexSubImage*, glCopyTextureSubImage***



Framebuffer Objects (FBO)

- Copiando datos entre FBO:

- **void glBlitFramebuffer(GLint srcX0, GLint srcY0, GLint srcX1, GLint srcY1, GLint dstX0, GLint dstY0, GLint dstX1, GLint dstY1, GLbitfield mask, GLenum filter)**
 - **src{X0,Y0,X1,Y1}**: coordenadas del rectángulo a copiar. Si $x_0 > x_1$, flip horizontal, si $y_0 > y_1$, flip vertical
 - **dst{X0,Y0,X1,Y1}**: coordenadas del rectángulo destino
 - **mask**: combinación de GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT, GL_COLOR_BUFFER_BIT
 - **filter**: GL_NEAREST ó GL_LINEAR (GL_NEAREST obligatorio para copiar profundidades o stencil)
- **void glBlitNamedFramebuffer(GLuint readFB, GLuint drawFB, GLint srcX0, GLint srcY0, GLint srcX1, GLint srcY1, GLint dstX0, GLint dstY0, GLint dstX1, GLint dstY1, GLbitfield mask, GLenum filter)**
 - **readFB, drawFB**: fbo de origen y destino, respectivamente

A partir de GL 4.5, sin necesidad de bind

Framebuffer Objects (FBO)

ej9-1

```
GLuint fbo;
GLuint rbos[2];

glGenFramebuffers(1, &fbo);
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);

glGenRenderbuffers(2, rbos);
glBindRenderbuffer(GL_RENDERBUFFER, rbos[0]);
glRenderbufferStorage(GL_RENDERBUFFER, GL_RGBA, fbowidth, fboheight);

glBindRenderbuffer(GL_RENDERBUFFER, rbos[1]);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT, fbowidth, fboheight);

glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                          GL_RENDERBUFFER, rbos[0]);
glFramebufferRenderbuffer(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                          GL_RENDERBUFFER, rbos[1]);
if (glCheckFramebufferStatus(GL_DRAW_FRAMEBUFFER) == GL_FRAMEBUFFER_COMPLETE)
    std::cout << "FBO completo" << std::endl;
else
    std::cout << "FBO incompleto" << std::endl;
```

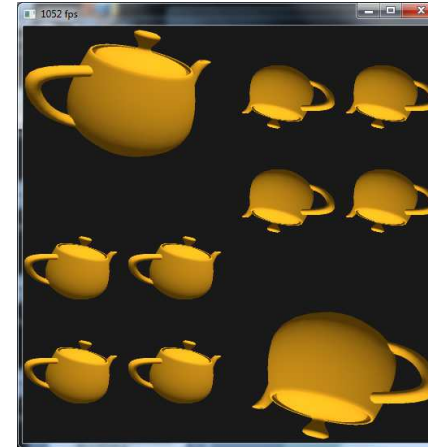
Framebuffer Objects (FBO)

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo);  
glDrawBuffer(GL_COLOR_ATTACHMENT0);  
glViewport(0, 0, fbowidth, fboheight);
```

```
/* Dibujar tetera */
```

```
glBindFramebuffer(GL_READ_FRAMEBUFFER, fbo);  
glReadBuffer(GL_COLOR_ATTACHMENT0);  
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);  
glDrawBuffer(GL_BACK_LEFT);  
glViewport(0, 0, width, height);
```

```
glBlitFramebuffer(0, 0, fbowidth, fboheight, 0, height/2, width/2, height,  
                  GL_COLOR_BUFFER_BIT, GL_LINEAR);  
glBlitFramebuffer(fbowidth, fboheight, 0, 0, width/2, 0, width, height/2,  
                  GL_COLOR_BUFFER_BIT, GL_LINEAR);  
for (int i = 0; i < 2; i++)  
    for (int j = 0; j < 2; j++) {  
        glBlitFramebuffer(0, 0, fbowidth, fboheight, i*width/4, j*height/4, (i+1)*width/4,  
                           (j+1)*height/4, GL_COLOR_BUFFER_BIT, GL_LINEAR);  
        glBlitFramebuffer(fbowidth, fboheight, 0, 0, i*width/4+width/2,  
                           j*height/4+height/2, (i+1)*width/4+width/2, (j+1)*height/4+height/2,  
                           GL_COLOR_BUFFER_BIT, GL_LINEAR);  
    }
```



ej9-1 y
ej9-2

Introducción

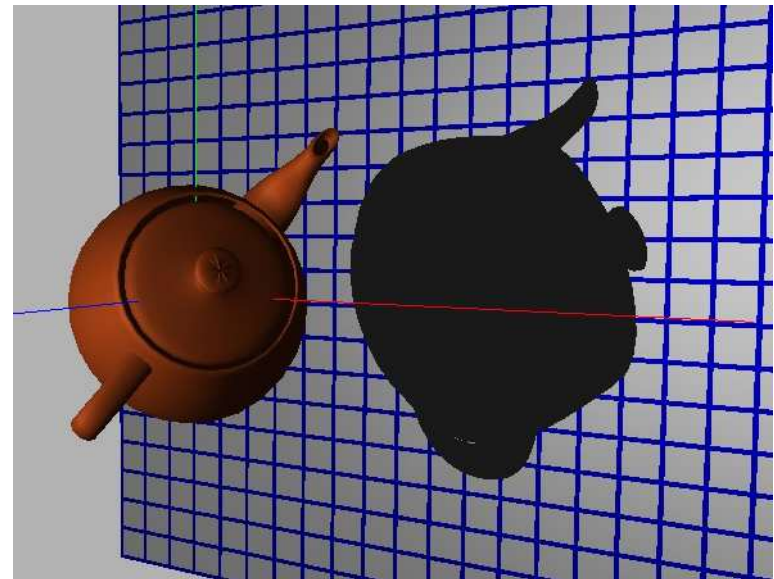
- La sombra se produce sobre una superficie cuando un objeto se interpone entre ésta y una fuente de luz
- Normalmente, en la realidad, los contornos de las sombras no están perfectamente definidos, sino que aparecen como gradientes



<http://flickr.com/photos/emdot/>

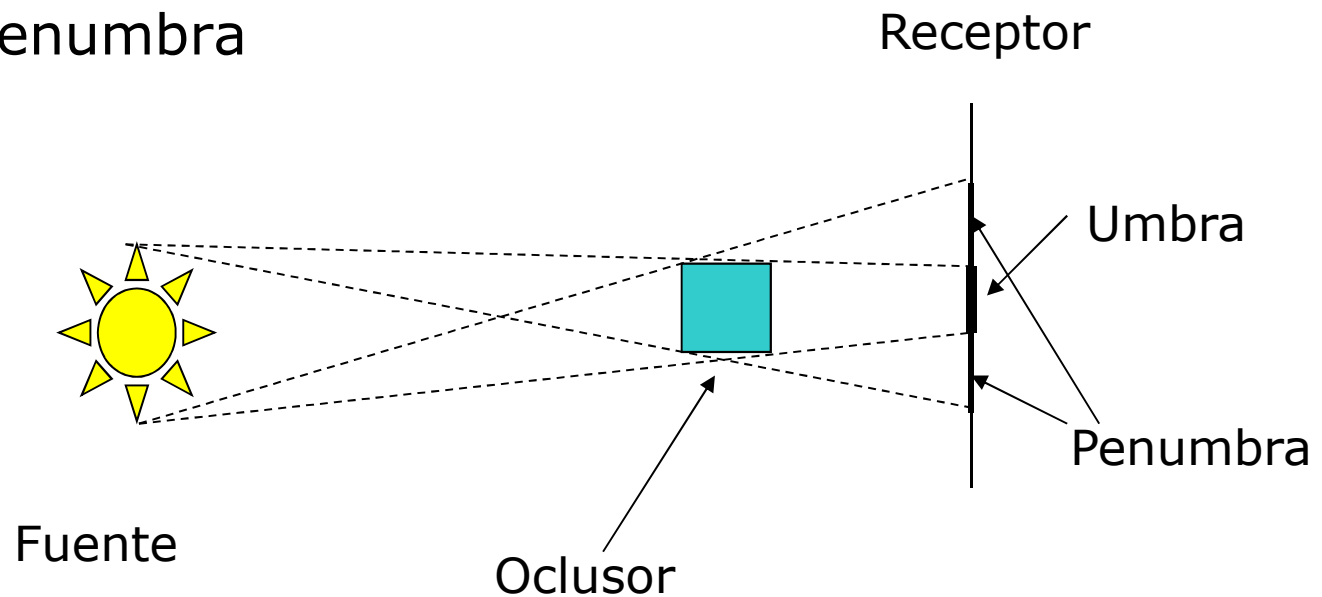
Introducción

- Para calcular el color de un objeto, OpenGL no tiene en cuenta si hay otros objetos de la escena entre éste y las fuentes de luz
- Hay que incorporar explícitamente las sombras en el proceso de visualización
- Dos tipos de sombras:
 - Duras (generadas por fuentes puntuales)
 - Suaves (generadas por fuentes definidas por un área)



Introducción

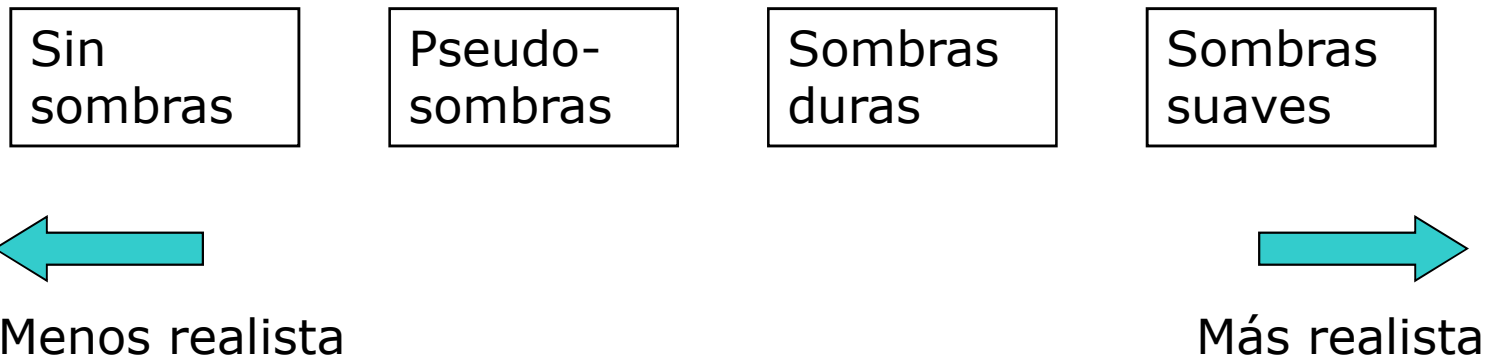
- Las sombras suaves están compuestas de dos partes:
 - Umbra
 - Penumbra





Introducción

- La complejidad de las técnicas de dibujo de sombras depende de:
 - La complejidad del ocluser
 - La complejidad de la parte de la escena que recibe la sombra
- Nivel de realismo:



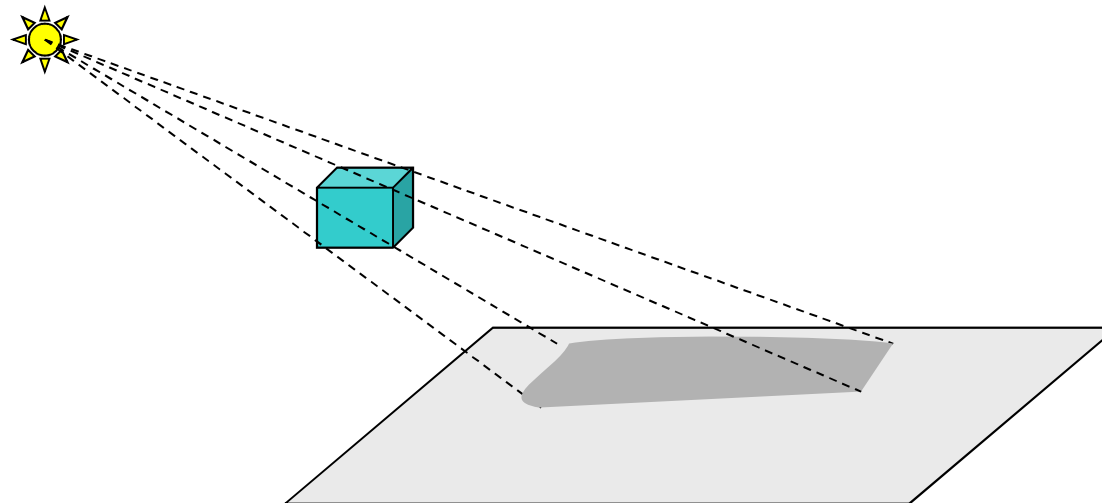
Introducción

- No se puede simular una sombra suave simplemente suavizando una dura:
 - La zona de sombra más cercana al ocluser está más definida
 - Además, la umbra en una sombra blanda depende del tamaño de la fuente y la distancia entre ocluser y receptor.



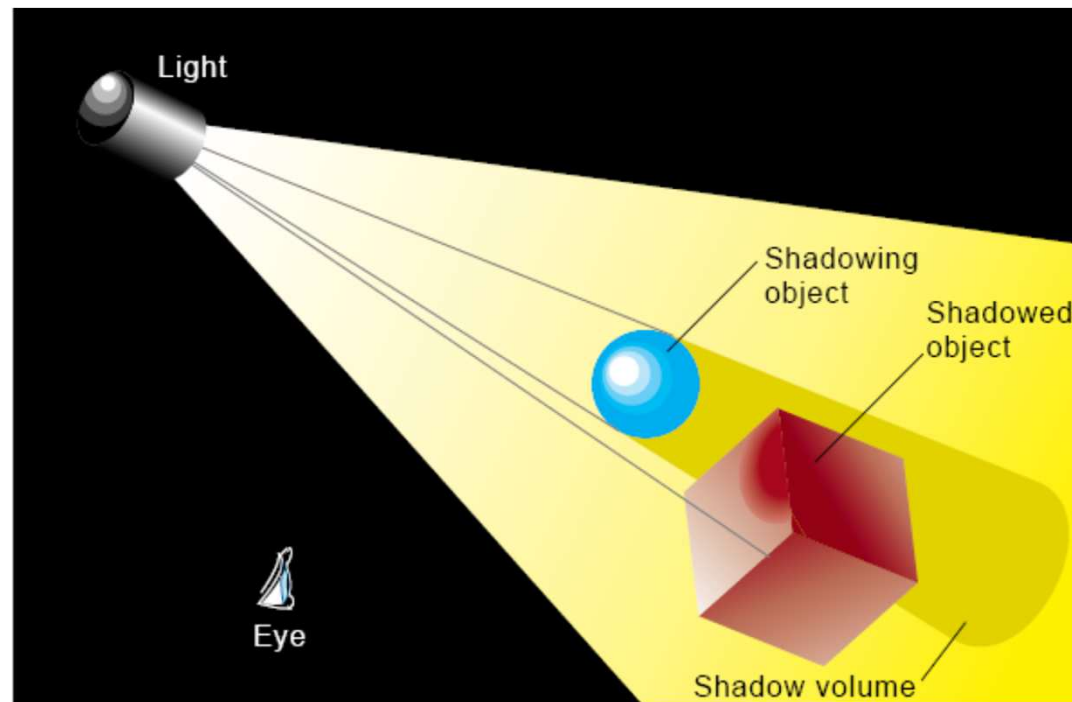
Introducción

- Existen muchos métodos para calcular sombras en tiempo real:
 - <https://developer.nvidia.com/gpugems/GPUGems>
 - Sombras planas: proyectar la geometría del ocluser sobre el receptor de la sombra con respecto a la luz



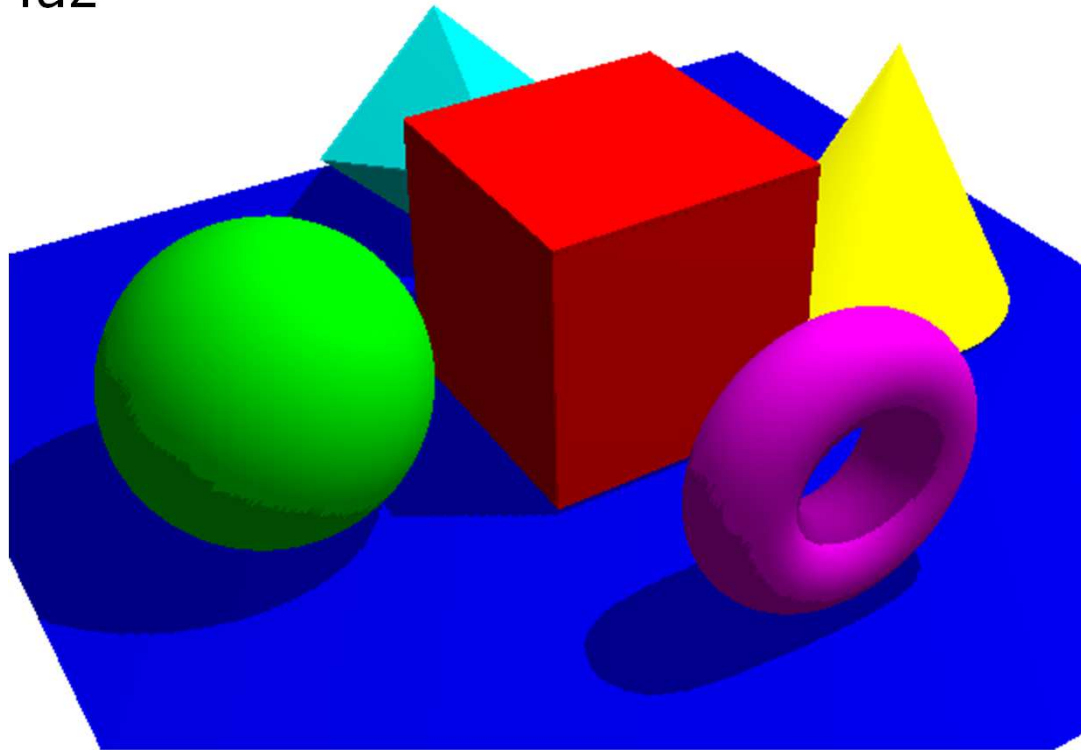
Introducción

- Volúmenes de sombra. Usando el *stencil buffer*, el oclisor define un volumen de sombra.





Introducción

- Shadow mapping: usa una textura de Z para calcular la visibilidad de un punto con respecto a la luz





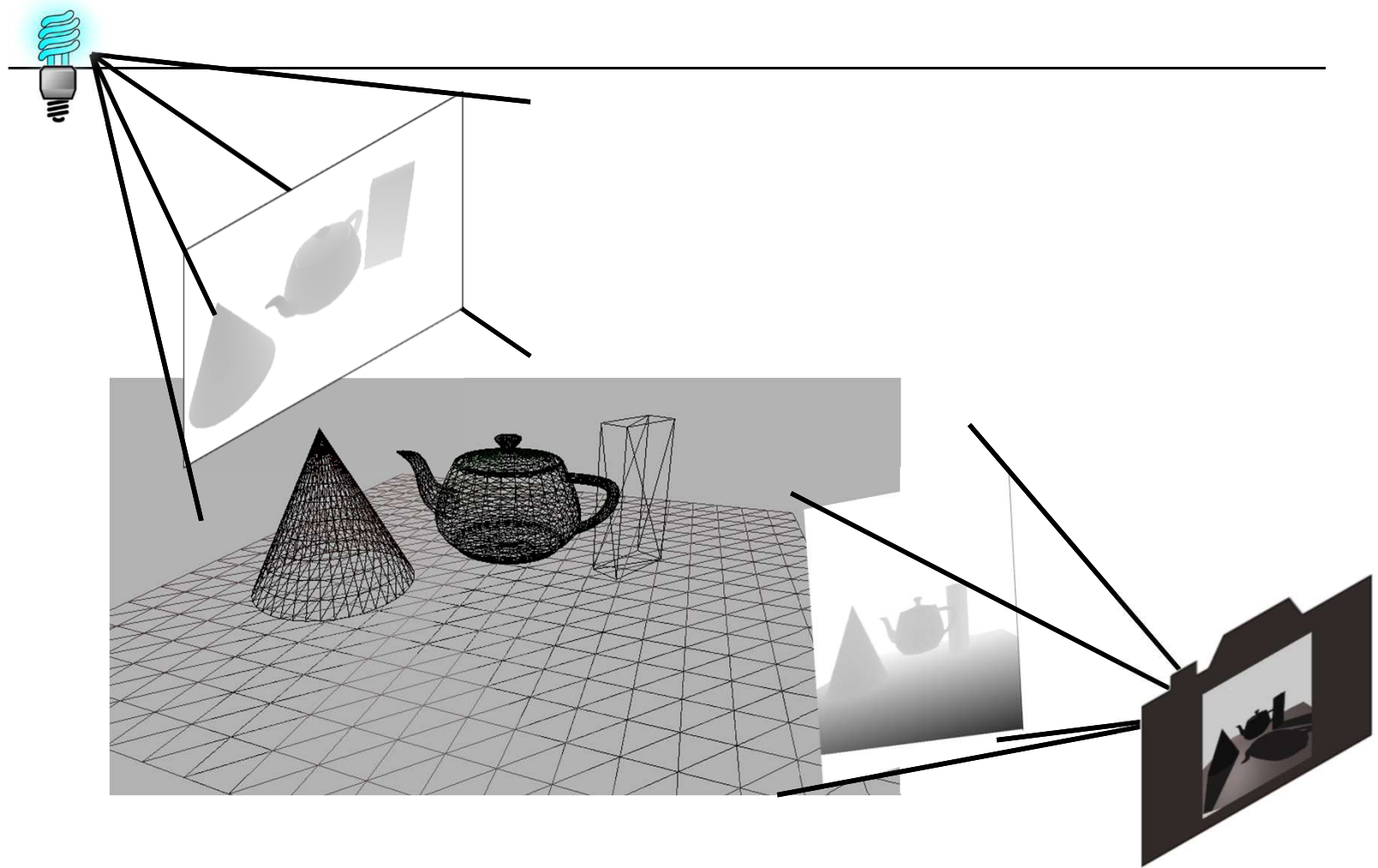
Shadow Mapping

- Es un método genérico de cálculo de sombras muy utilizado en la actualidad
- Es un algoritmo en el espacio de la imagen
 - Es independiente de la geometría de la escena 
 - Pueden aparecer problemas de aliasing 
- Se usa tanto en aplicaciones interactivas como off-line
- Para dibujar las sombras, hay que calcular si cada punto de la escena puede “ver” una fuente de luz o no...
- este algoritmo lo hace al revés, precalculando qué puntos de la escena son visibles desde el punto de vista de la luz

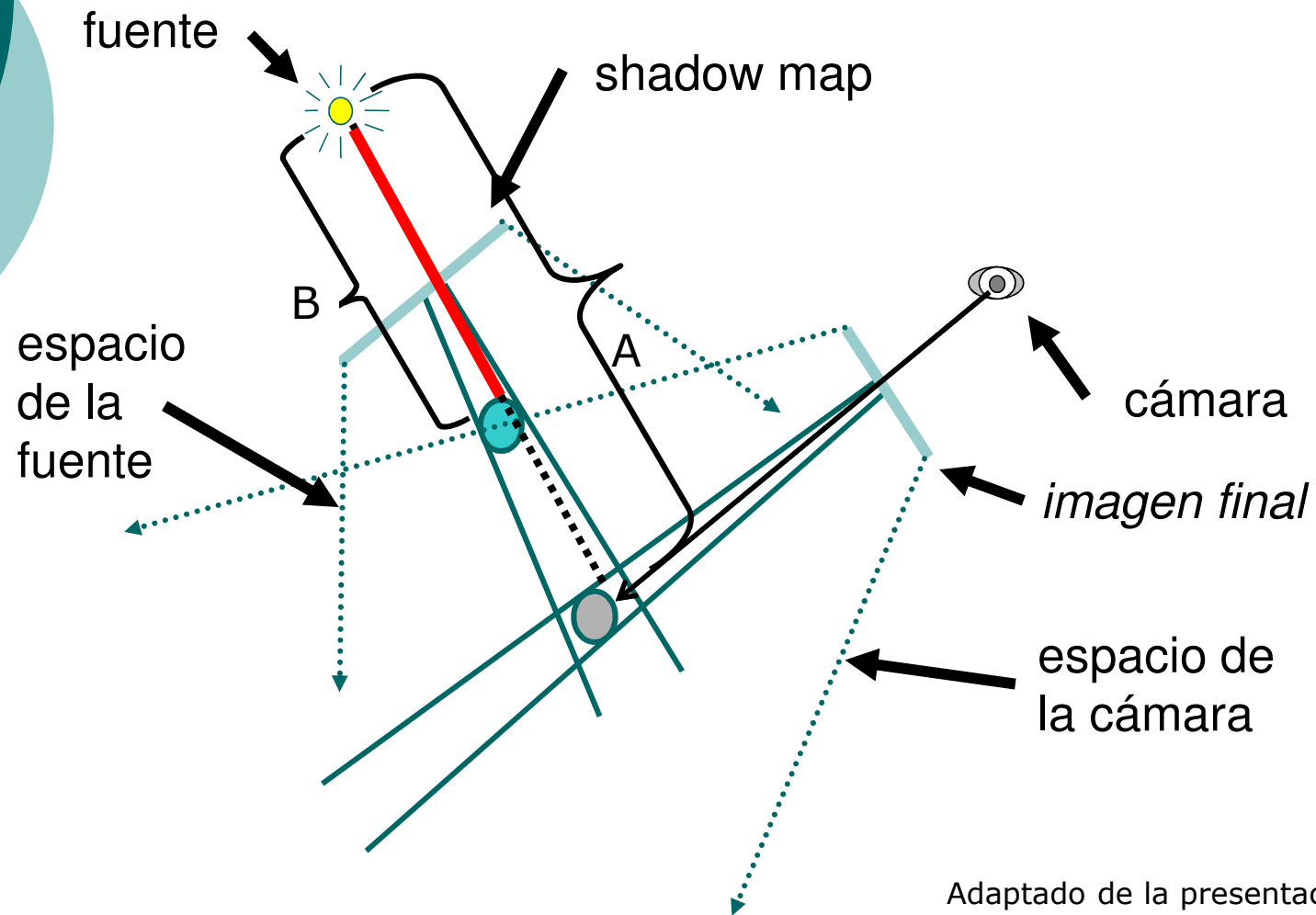


Shadow Mapping

- Dos pasos:
 1. Visualizar la escena desde la fuente
 - Sólo nos interesa el contenido del Z-Buffer, por lo que no se calcula la iluminación
 - El resultado es un mapa de profundidad (el *shadow map*)
 2. Visualizar la escena desde la cámara
 - Comparar la distancia a la fuente de cada fragmento dibujado con el valor del mapa de profundidad para decidir si está en sombra o no



Shadow Mapping



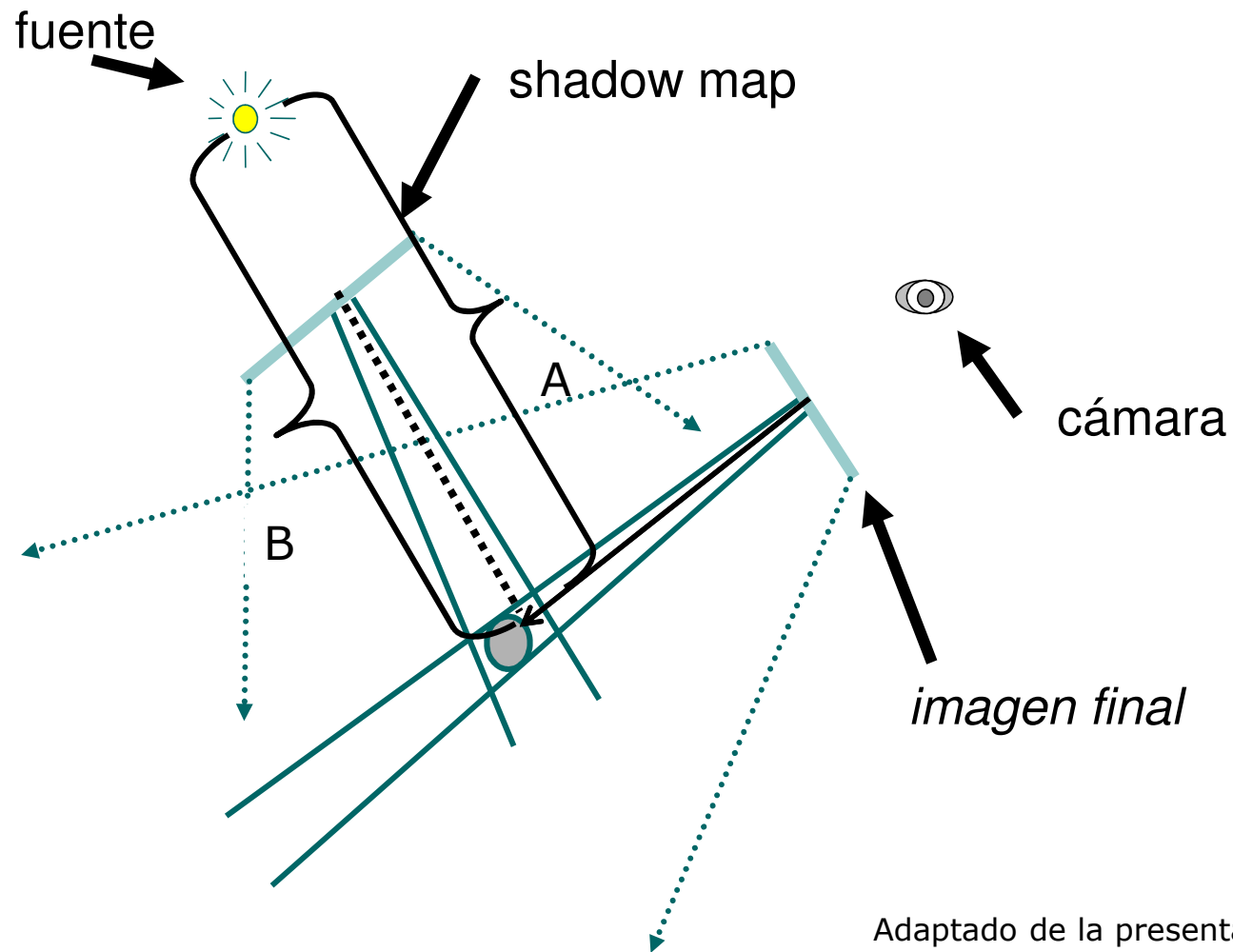
Adaptado de la presentación de
NVIDIA sobre Shadow Mapping



Shadow Mapping

- Posibles resultados de la comparación:
 - A: distancia desde la fuente a la posición del fragmento que se está procesando desde la cámara
 - B: distancia desde la fuente al objeto más cercano en la dirección a la posición del fragmento que se está procesando (valor almacenado en el mapa de profundidad)
 - $A > B$, entonces debe haber algo entre la luz y el punto (en sombra)
 - $A \sim B$, entonces el fragmento está iluminado

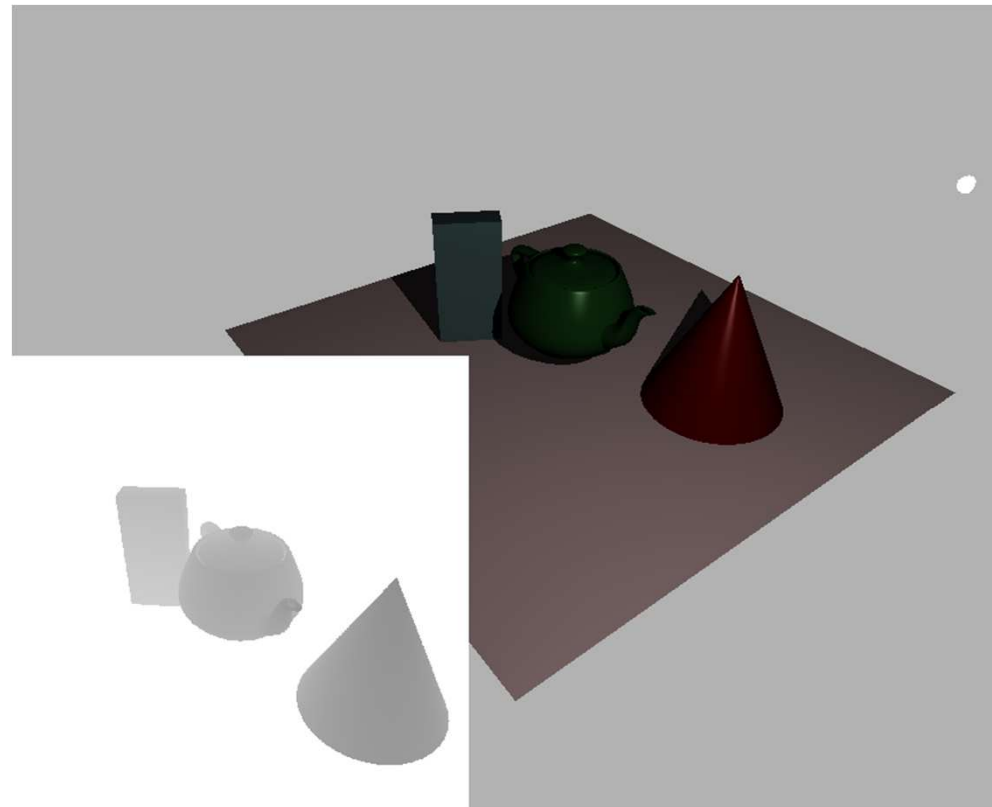
Shadow Mapping



Adaptado de la presentación de
NVIDIA sobre Shadow Mapping
<http://developer.nvidia.com>

Shadow Mapping

Demo: p9



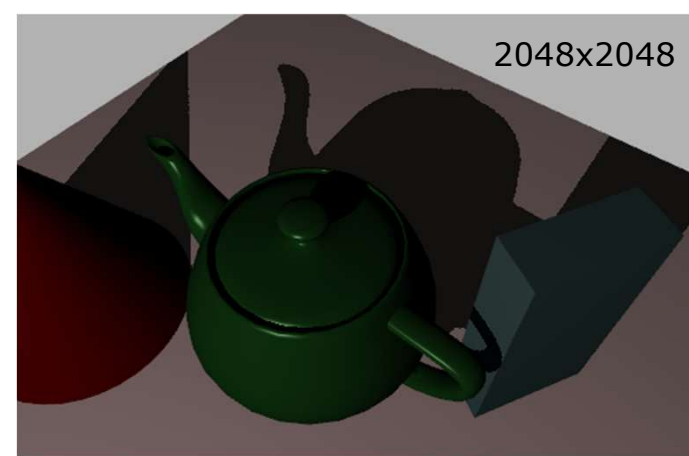
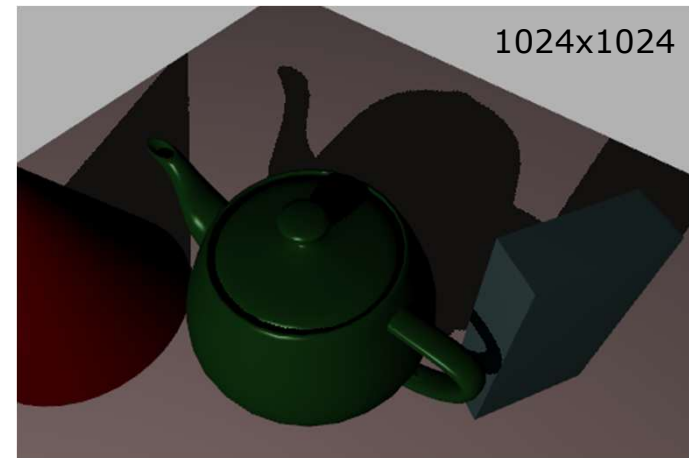
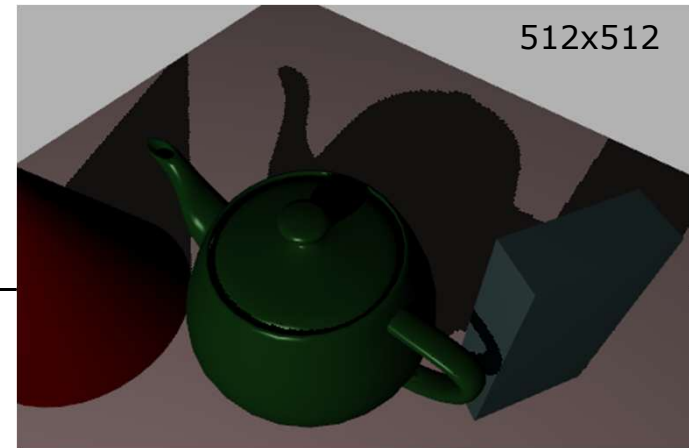
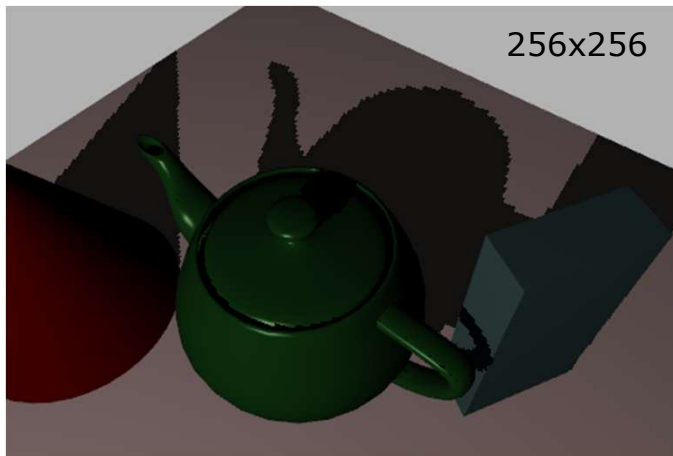
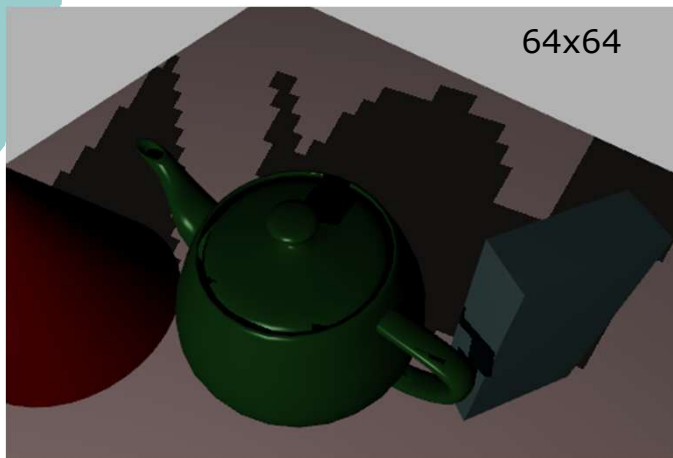


Shadow Mapping

- Primera pasada: cálculo del *shadow map*
- Preparación:
 - Crear una textura donde almacenar el Z-Buffer (se pueden declarar texturas con tipo `GL_DEPTH_COMPONENT[{16|24|32}]`)
 - El tamaño de la textura determina el detalle de las sombras
 - Crear un FBO que contenga sólo la textura anterior, vinculada como `GL_DEPTH_ATTACHMENT`

Shadow Mapping

- Tamaños del shadow map
- GL_NEAREST





Shadow Mapping

- Primera pasada: cálculo del *shadow map*
- Ejecución:
 - Vincular el FBO
 - Ajustar el viewport al tamaño del FBO
 - Borrar el z-buffer
 - Desactivar el dibujo al buffer de color con `glDrawBuffer(GL_NONE)`
 - Dibujar los ocluidores desde el punto de vista de la luz
 - Creamos la *matriz de sombra*:
 - Model: identidad
 - View: lookAt desde la posición de la fuente al centro de la escena
 - Projection: definir un volumen de cámara que abarque toda la escena
 - En el shader de vértice, escribir la posición en el espacio de clip
 - En el de fragmento, no hacer nada



Shadow Mapping

```
// Shader de vértice para generar el shadow map
#version 330 core
$GLMatrices
in vec4 position;

void main(void) {
    gl_Position = modelviewprojMatrix * position;
}
```

```
// Shader de fragmento para generar el shadow map
#version 330 core

void main(void) {

}
```



Shadow Mapping

- Segunda pasada: dibujo de la escena
 - Vincular el FB por defecto y ajustar el viewport
 - Dibujar al buffer trasero (glDrawBuffer)
 - Para acceder al shadow map, hay que ajustar la matriz de sombra (que lleva los vértices definidos en el sistema del mundo al sistema de clip de la fuente, con coordenadas $[-1, 1]^3$), para que obtenga coordenadas usadas por el texturado $[0, 1]^3$:

$$\begin{pmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



Shadow Mapping

- Segunda pasada: dibujo de la escena
 - En el shader de vértice:
 - Propagar las variables para el cálculo de iluminación por fragmento (Phong)
 - Pasar la coordenada del vértice en el espacio escalado de clip de la luz (transformada con la matriz de sombra escalada)
 - En el shader de fragmento
 - Declarar un `sampler2DShadow` para acceder al shadow map
 - Invocar a `textureProj` con las coordenadas interpoladas desde el shader de vértice para obtener el valor de la sombra (entre 0 y 1), y multiplicarlo por las componentes difusa y especular en el cálculo de iluminación



Shadow Mapping

- Muestreadores de sombra
 - Hasta ahora, hemos estado usando los muestreadores (*samplers*) estándar para acceder a las texturas
 - Dichos muestreadores devuelven el color almacenado en la coordenada de textura pedida
 - Hay un tipo especial de muestreador para trabajar con texturas de profundidad
 - `sampler2DShadow`: es un tipo especial de textura que devuelve 1.0 si la coordenada p de la coordenada de textura satisface una comparación con el valor correspondiente del mapa, o 0.0 si no

Recuerda: las coordenadas de textura se llaman (s, t, p, q)



Shadow Mapping

- Muestreadores de sombra
 - En los shaders, se usan con la función:
 - `float texture(sampler2DShadow tex, vec3 P)`
 - donde:
 - `tex`: es el *sampler* (definido como un uniform, y donde la aplicación escribe el número de unidad de textura donde se encuentra)
 - `P (s, t, p)`: `(s, t)` es la coordenada de textura que se quiere acceder, `p` es el valor que se usará en la comparación
 - La función devuelve el resultado de la comparación (un valor entre 0 y 1)



Shadow Mapping

- El modo de comparación se activa con:
 - `void glTexParameterf(target, pname, param)`
 - target: GL_TEXTURE_2D...
 - pname: GL_TEXTURE_COMPARE_MODE
 - param: GL_COMPARE_REF_TO_TEXTURE o GL_NONE
 - Si `param == GL_COMPARE_REF_TO_TEXTURE`, en vez de devolver el valor del texel, devuelve el resultado de comparar la coordenada p de la coordenada de textura con el valor almacenado en la coordenada (s, t) de la textura

Shadow Mapping

- La función de comparación de la textura se selecciona con:

- `void glTexParameteri(target, pname, param)`
 - target: GL_TEXTURE_2D...
 - pname: GL_TEXTURE_COMPARE_FUNC
 - param: GL_LEQUAL, GL_GEQUAL, GL_LESS, GL_GREATER, GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS, GL_NEVER

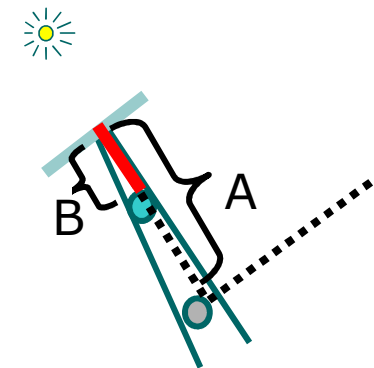
- Ejemplo:

- Para *shadow mapping*, usamos GL_LEQUAL
- Así, la llamada en el shader:

```
uniform sampler2DShadow tex;  
f = texture(tex, tc); // donde tc es un vec3
```

- devuelve:

- $$\begin{cases} 1.0, & \text{si } tc.p \leq tex[tc.s, tc.t] \\ 0, & \text{si no} \end{cases}$$





Shadow Mapping

- El ejemplo anterior no funciona si no se aplica la división perspectiva (las coordenadas para acceder al *shadow map* vienen de un proceso de proyección)
- *textureProj* es una función de textura proyectiva. Antes de acceder a la textura, divide las coordenadas de textura por su cuarta coordenada

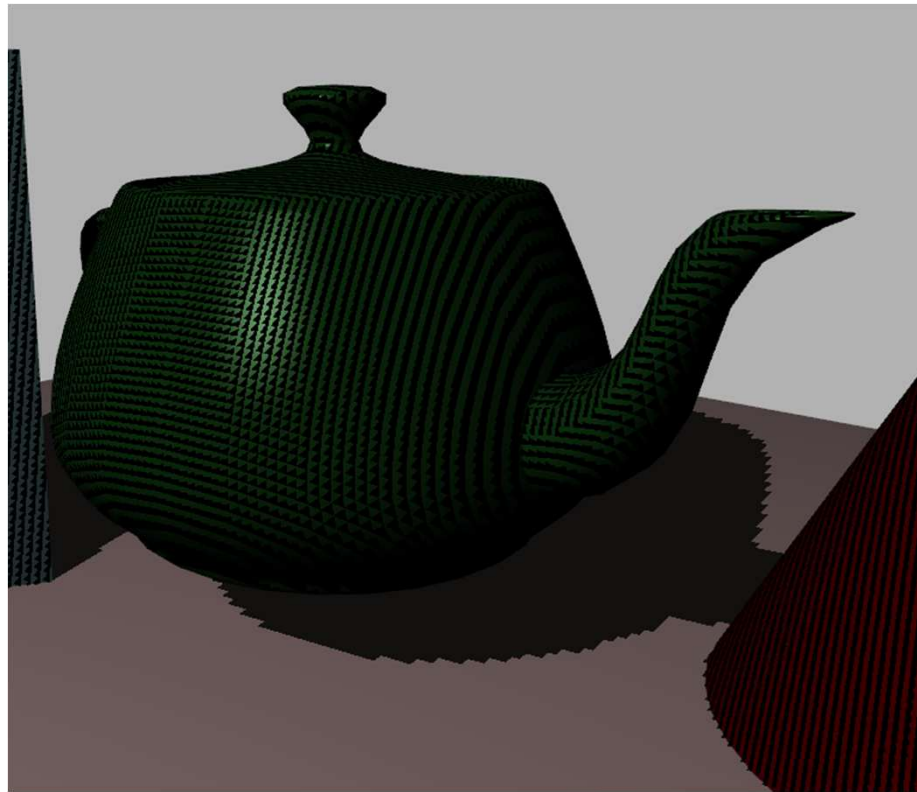
```
uniform sampler2DShadow shadowMap;  
in vec4 shadowCoord;  
...  
float f = textureProj(shadowMap, shadowCoord);
```

es equivalente a:

```
float f = texture(shadowMap, vec3(shadowCoord/shadowCoord.q));
```

Shadow Mapping

- Problemas del Shadow Mapping



Auto-sombras



Shadow Mapping

- Para resolver el problema de las autosombras se puede usar la función:

- `void glPolygonOffset(GLfloat factor, GLfloat units)`

- donde: la función modifica la coordenada Z del fragmento antes de aplicar el depth test y antes de escribirlo en el Z-buffer según la ecuación:

$$z' = z + factor \cdot DZ + r \cdot units$$

- donde DZ es un valor que depende de la inclinación del polígono con respecto a la ventana y r es el menor valor que garantiza un cambio en el z-buffer

- Para activar el offset de polígonos, hay que llamar a:

`glEnable(GL_POLYGON_OFFSET_FILL)`

- Al construir el shadow map, desplaza ligeramente los polígonos hacia atrás.

Shadow Mapping

- Cuidado al seleccionar los factores de offset. Si son muy grandes, pueden causar Peter Panning:





Shadow Mapping

- Problemas del Shadow Mapping
 - Coste: un z-buffer y una pasada a la escena por cada fuente de luz
 - Con el hardware actual, dibujar el shadow map no es problema, y la memoria disponible en la GPU también crece muy deprisa

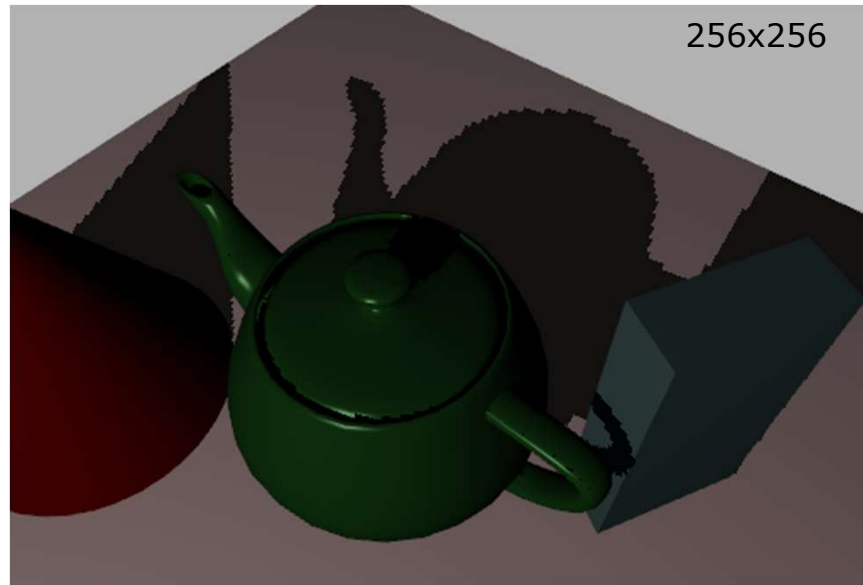


Shadow Mapping

- Problemas del Shadow Mapping
 - Se asume que la luz es focal
 - En caso de que la luz sea puntual y esté dentro de la escena, hay que calcular varios shadow maps para captar todas las posibles direcciones (por ejemplo, como un mapa cúbico):
Omnidirectional shadow maps.

Shadow Mapping

- Problemas del Shadow Mapping
 - Aliasing en los bordes de la sombra



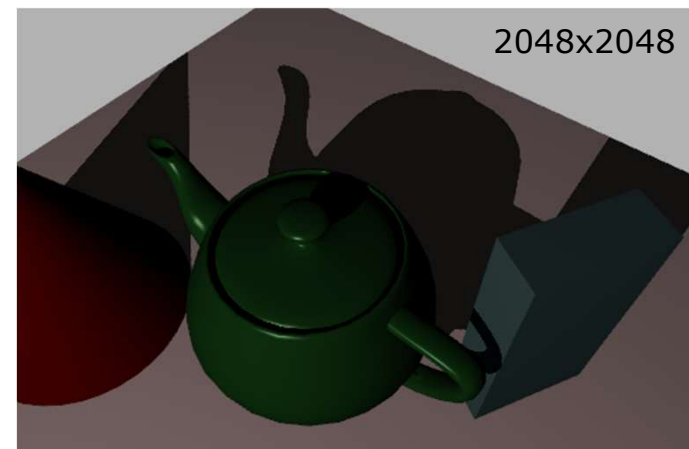
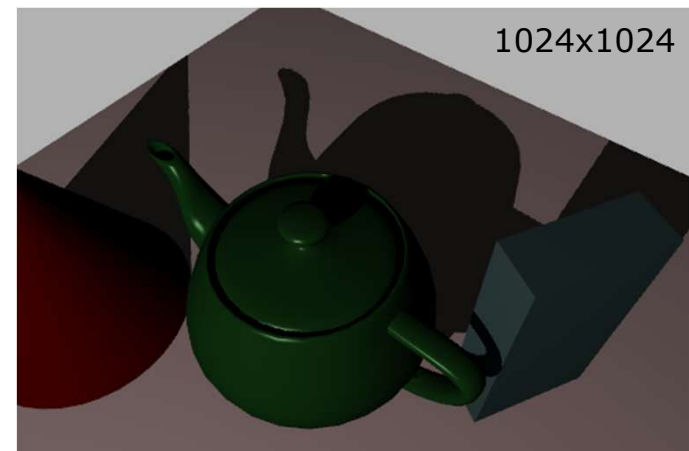
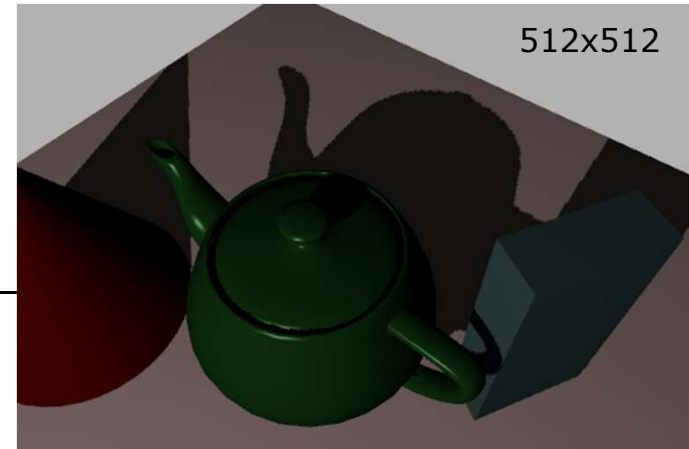
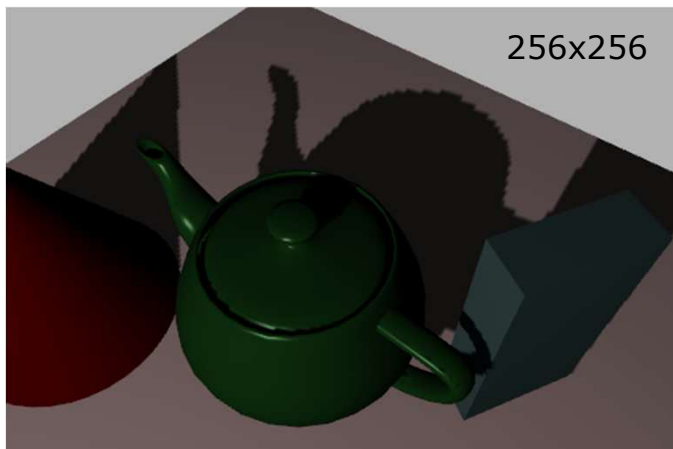
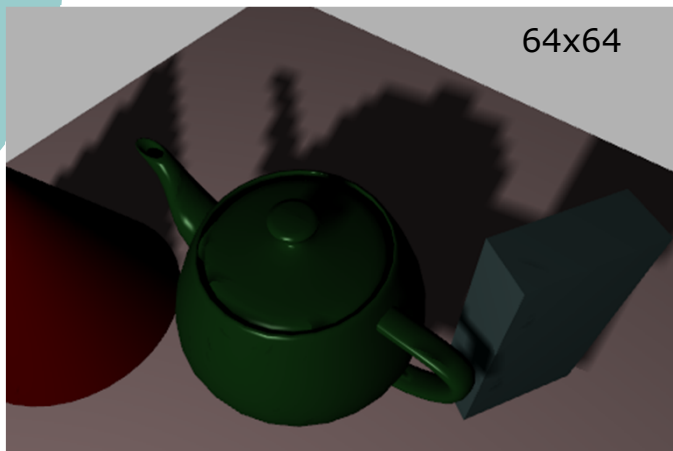


Shadow Mapping

- Problemas del Shadow Mapping
 - Solución: Percentage-Closer Filtering (PCF)
 - En vez de muestrear el shadow map una vez, tomar las cuatro muestras más cercanas (usar el filtrado de ampliación GL_LINEAR en vez de GL_NEAREST)
 - El hardware no interpola las cuatro profundidades y luego realiza la comparación, si no que devuelve una combinación de los resultados de las comparaciones (el resultado siempre estará entre 0.0 y 1.0)
 - Este algoritmo genera una sombra suave, aunque el tamaño de la penumbra depende de la resolución del shadow map

Shadow Mapping

- PCF (filtrado de textura GL_LINEAR)





Shadow Mapping

- Problemas del Shadow Mapping
 - Solución: Percentage-Closer Filtering (PCF)
 - En vez de dejarle al hardware que haga el muestreo del shadow map, lo podemos hacer nosotros, decidiendo:
 - Número de muestras, amplitud del área muestreada, patrón de muestreo, o los pesos de cada muestra
 - Así, se pueden obtener sombras suaves más realistas



Shadow Mapping

- Para optimizar el uso de la resolución del z-buffer, es conveniente:
 - Alejar todo lo posible el plano de recorte frontal
 - Acercar todo lo posible el plano de recorte trasero
- Aún así, aún queda el problema de intentar muestrear una zona alejada de la luz, o escenarios exteriores, con iluminación natural



Cascaded Shadow Mapping

- Este algoritmo usa varios shadow maps, para asegurarse de que la zona más cercana a la cámara está suficientemente muestreada
- Opciones:
 - Generar los shadow maps centrados en la cámara, ampliando poco a poco el campo de visión
 - Asignar cada grupo de objetos a un shadow map distinto
 - Tres shadow maps: uno para los objetos animados (o que se pueden romper) dentro del volumen de la vista de la cámara, otro que incluye la celda del mapa donde se encuentra la cámara, y otro que incluye todo el mapa
 - Subdividir el volumen de la vista a lo largo de la dirección de la vista, y generar un shadow map que incluya cada subdivisión (*parallel split shadow mapping*)
 - ...

Playing with Real-Time Shadows

Crytek, SIGGRAPH 2013

Shadows in Games: Crysis 1



<http://www.crytek.com/download/Playing%20with%20Real-Time%20Shadows.pdf>

Playing with Real-Time Shadows

Crytek, SIGGRAPH 2013

Shadows in Games: Crysis 2



<http://www.crytek.com/download/Playing%20with%20Real-Time%20Shadows.pdf>

Playing with Real-Time Shadows

Crytek, SIGGRAPH 2013

Shadows in Games: Crysis 3



<http://www.crytek.com/download/Playing%20with%20Real-Time%20Shadows.pdf>