UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Centre de la Imatge i la Tecnologia Multimèdia

CG Basics VII
Shadow Mapping

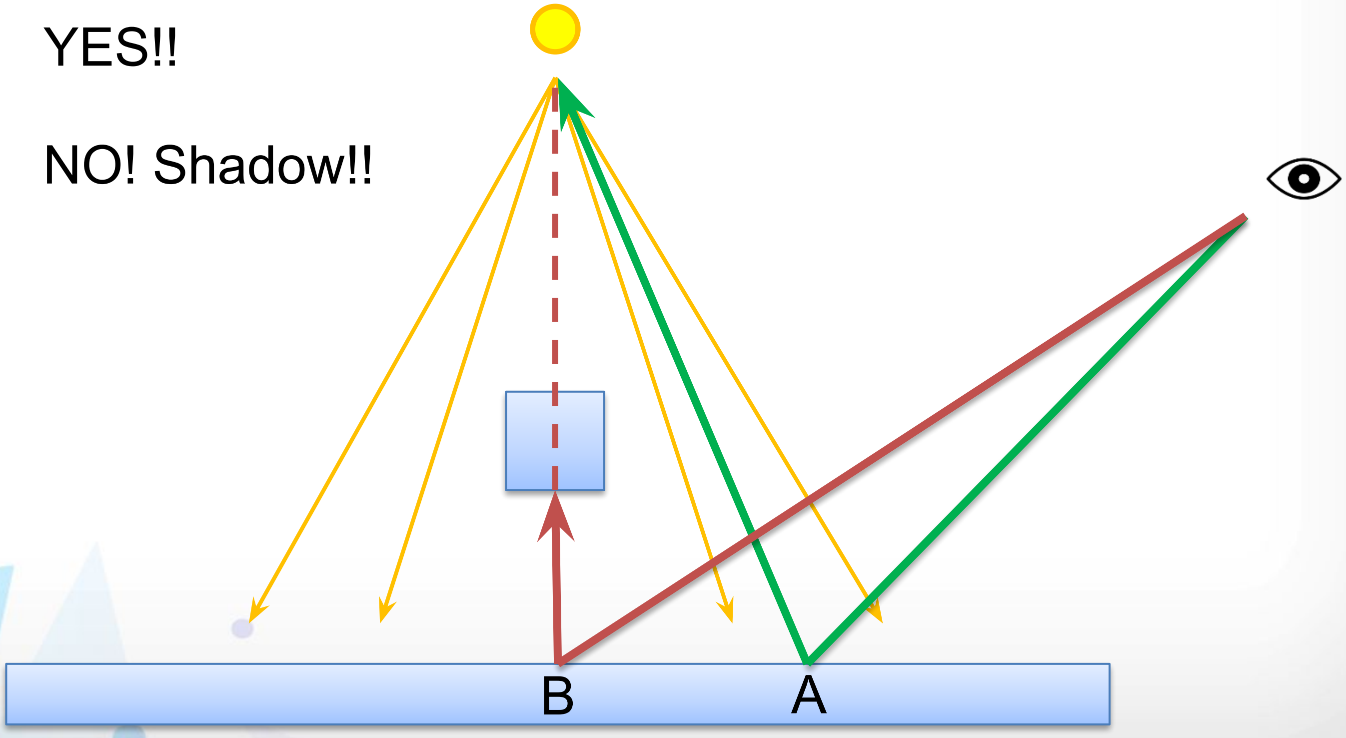
Bachelor's Degree in Video Game Design and Development

Shadow Mapping

- **Basic idea**

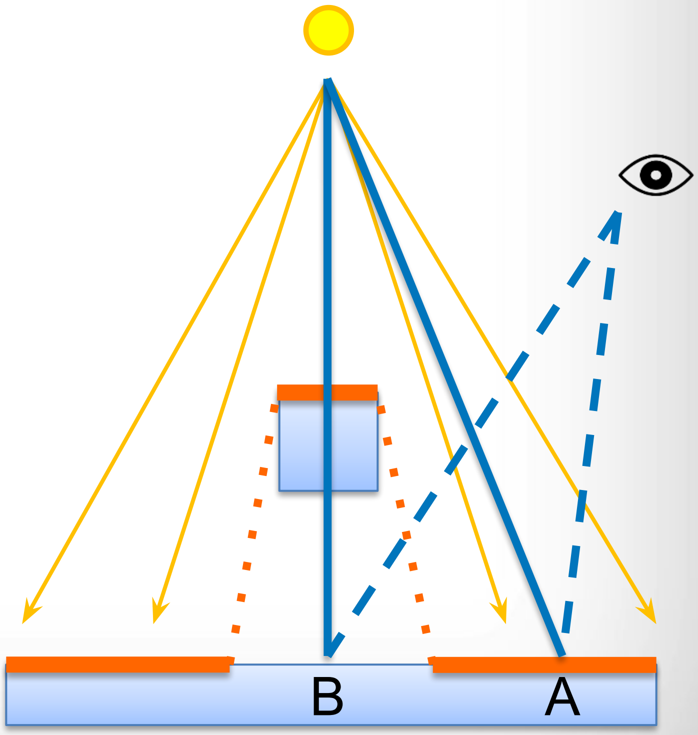
A lit? YES!!

B lit? NO! Shadow!!



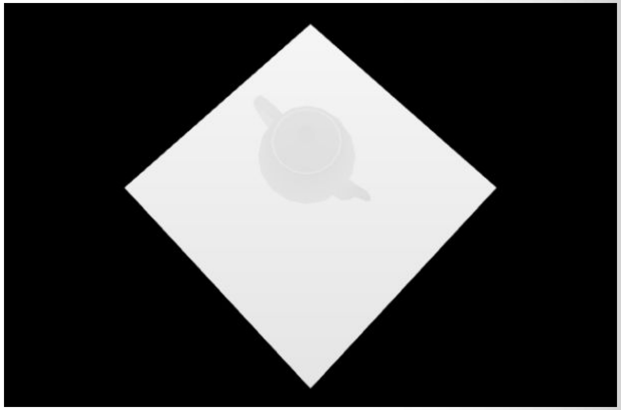
Shadow Mapping

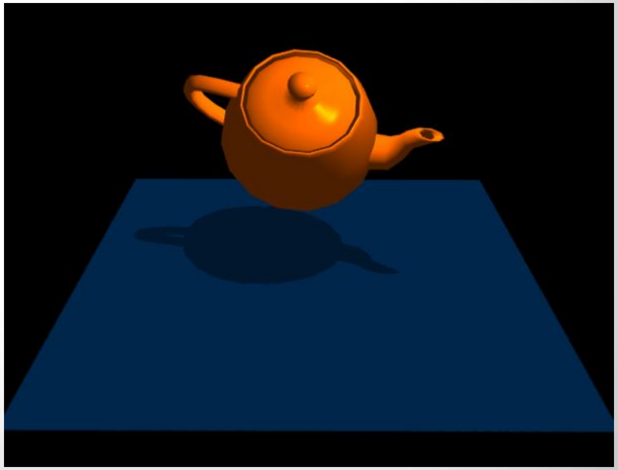
- **The basic shadow map algorithm consists on**
  - **Computing the distance of the objects to the light from the light's point of view (depth from light source)**
  - **Computing the distance to the light of the objects rendered from the camera**
  - **If this last on is bigger than the first one, then the point is in the shadow (B)**



Basic Algorithm

- Two rendering passes
  - Render the scene from the light's point of view
    - Save the distances (depth) into a texture
  - Render the scene from the camera
    - Compute the distance of each pixel to the light source
    - Compare it with the distance stored in the texture
    - Dim the color of the pixels whose distance to the light is bigger than the one stored in the texture





Basic Algorithm

- **Before rendering the scene from the light's point of view...**
  - **We must create a framebuffer object**
  - **Create and bind a color texture to it (this texture will store the depth from the light point of view).**
  - **Create and bind a render buffer that acts as the default z-buffer (not used after rendering)**

```
var rttFramebuffer;
var rttTexture;

function initTextureFramebuffer()
{
    rttFramebuffer = gl.createFramebuffer();
    gl.bindFramebuffer(gl.FRAMEBUFFER, rttFramebuffer);
    rttFramebuffer.width = 2048;
    rttFramebuffer.height = 2048;

    rttTexture = gl.createTexture();
    gl.bindTexture(gl.TEXTURE_2D, rttTexture);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR);
    gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
    gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA,
                  rttFramebuffer.width, rttFramebuffer.height,
                  0, gl.RGBA, gl.UNSIGNED_BYTE, null);

    var renderbuffer = gl.createRenderbuffer();
    gl.bindRenderbuffer(gl.RENDERBUFFER, renderbuffer);
    gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16,
                           rttFramebuffer.width, rttFramebuffer.height);

    gl.framebufferTexture2D(gl.FRAMEBUFFER, gl.COLOR_ATTACHMENT0,
                            gl.TEXTURE_2D, rttTexture, 0);
    gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT,
                               gl.RENDERBUFFER, renderbuffer);

    gl.bindTexture(gl.TEXTURE_2D, null);
    gl.bindRenderbuffer(gl.RENDERBUFFER, null);
    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
}
```

Size power of 2 in WebGL!!

Basic Algorithm

- **Render the scene from the light's point of view**
  - **Enable the framebuffer object in order to render the scene to the created texture**
  - **Set an appropriate viewport to match the texture size**
  - **Disable the framebuffer at the end of the function**

```
function renderingLoop() {
    requestAnimFrame(renderingLoop);
    drawSceneFromLight();
    drawSceneFromCamera();
}
```

You already had this one

```
function drawSceneFromLight()
{
    gl.bindFramebuffer(gl.FRAMEBUFFER, rttFramebuffer);

    gl.viewport(0, 0, rttFramebuffer.width, rttFramebuffer.height);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    // Rendering code

    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
}
```

Basic Algorithm

- You need two different shaders
  - One to render from the light's point of view (outputs depth)
  - Another to render from the point of view of the camera (outputs shaded color)

```
<script id="light-vs" type="x-shader/x-vertex">
    // Shader code ...
</script>

<script id="light-fs" type="x-shader/x-fragment">
    // Shader code ...
</script>
```

```
<script id="camera-vs" type="x-shader/x-vertex">
    // Shader code ...
</script>

<script id="camera-fs" type="x-shader/x-fragment">
    // Shader code ...
</script>
```

Basic Algorithm

- **Configure shaders in each rendering function**
  - **Set the shader to use at the beginning of the function**
  - **Configure vertex attributes at the beginning / disable them at the end.**

```
function drawSceneFromLight()
{
    gl.bindFramebuffer(gl.FRAMEBUFFER, rttFramebuffer);

    gl.viewport(0, 0, rttFramebuffer.width, rttFramebuffer.height);
    gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

    gl.useProgram(programLight);

    gl.enableVertexAttribArray(programLight.vertexPositionAttribute);

    // More rendering code ...

    gl.disableVertexAttribArray(programLight.vertexPositionAttribute);

    gl.bindFramebuffer(gl.FRAMEBUFFER, null);
}
```

Basic Algorithm

- **Render the scene from the light's point of view**
  - **Send the transformation matrices from the point of view of the light**

```
attribute vec3 aVertexPosition;

// Matrices
uniform mat4 modelViewMatrixLight;
uniform mat4 projectionMatrixLight;

void main(void)
{
    gl_Position = projectionMatrixLight * modelViewMatrixLight * vec4(aVertexPosition, 1.0);
}
```

Vertex shader

Basic Algorithm

- **Render the scene from the light's point of view**
  - **We just store the depth of each fragment (*gl_FragCoord.z*)**
  - **Depth needs more than 8-bit precision. The encodeFloat() function stores a floating point number into a 32-bit RGBA color (8-bits per channel).**

```
precision mediump float;

vec4 encodeFloat (float depth)
{
    const vec4 bitShift = vec4(
      256 * 256 * 256,
      256 * 256,
      256,
      1.0
    );
    const vec4 bitMask = vec4(
      0,
      1.0 / 256.0,
      1.0 / 256.0,
      1.0 / 256.0
    );
    vec4 comp = fract(depth * bitShift);
    comp -= comp.xxyz * bitMask;
    return comp;
  }

void main(void)
{
    gl_FragColor = encodeFloat(gl_FragCoord.z);
}
```

Fragment shader

Basic Algorithm

- Render the scene from the camera
  - Besides the usual modelview and projection matrices, we also pass the modelview and the projection matrices for the "light camera" (used in the previous pass to render from the light)
  - In the vertex shader, we compute the position of each vertex with respect to the light by using both matrices

```
attribute vec3 aVertexPosition;

// ...

// Light matrices
uniform mat4 modelViewMatrixLight;
uniform mat4 projectionMatrixLight;

varying vec4 positionProjectedLightspace;

void main(void)
{
    // ...

    positionProjectedLightspace = projectionMatrixLight * modelViewMatrixLight * vec4(aVertexPosition, 1.0);
}
```

Vertex shader

Basic Algorithm

- **Render the scene from the camera**
  - **The lightDepth texture is the depth of the scene rendered from the light**
  - **The position of the surface as projected by the light matrices is passed from the vertex shader**
  - **The decodeFloat() function will be used to recover the depth from the lightDepth texture in the following steps**

Fragment shader

```
precision mediump float;

// ...

// Light depth map
uniform sampler2D lightDepth;

varying vec4 positionProjectedLightspace;

float decodeFloat (vec4 color)
{
    const vec4 bitShift = vec4(
      1.0 / (256.0 * 256.0 * 256.0),
      1.0 / (256.0 * 256.0),
      1.0 / 256.0,
      1
    );
    return dot(color, bitShift);
}
```

Basic Algorithm

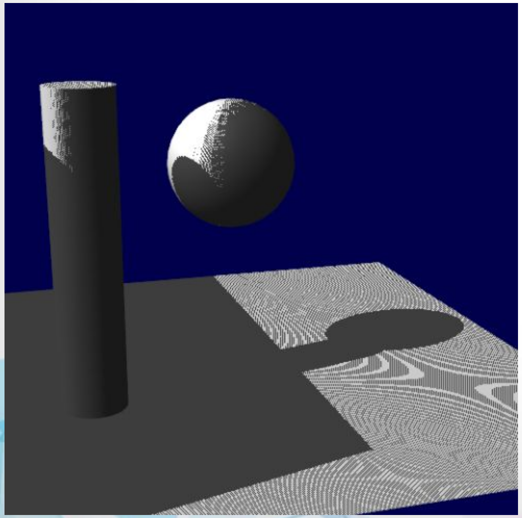
- **Render the scene from the camera**
  - **The distance of the pixel to the light is computed and compared to the one in the texture of the first rendering pass**
  - **A threshold is normally used when comparing in order to avoid artifacts produced by accuracy issues**

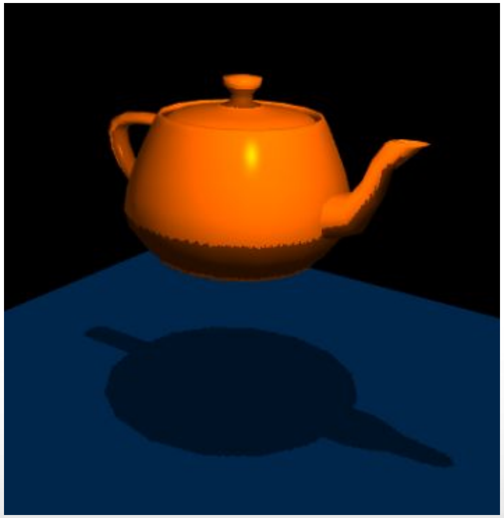
Fragment shader

```
void main(void)
{
    // Phong code ...

    vec3 positionTextureLightspace =
                0.5 * positionProjectedLightspace.xyz /
                      positionProjectedLightspace.w + vec3(0.5);

    float depthFromLightTexture = decodeFloat(
        texture2D(lightDepth,
        positionTextureLightspace.xy));

    float lightFactor = 0.0;
    if (depthFromLightTexture + 0.001 > positionTextureLightspace.z) {
        lightFactor = 1.0;
    }

    vec3 shadedColor = ambient + (diffuse + specular) * lightFactor;

    gl_FragColor = vec4(shadedColor, 1.0);
}
```

Basic Algorithm

- **Due to accuracy issues, this technique may produce artifacts when applying shadows**
  - **Use the *encodeFloat / decodeFloat* functions to encode depth with high precision and avoid z-fighting**
  - **Use a texture of high resolution to remove jagged edges**





Bibliography

The slides of the course related to Computer Graphics are mostly based on the material of the subject *Virtual and Augmented Reality* of the *MIRI Master* of the UPC

As well, the part related to shaders is based on the book *OpenGL Shading Language by R.J. Rost (Addison-Wesley)*

Questions?

www.citm.upc.edu