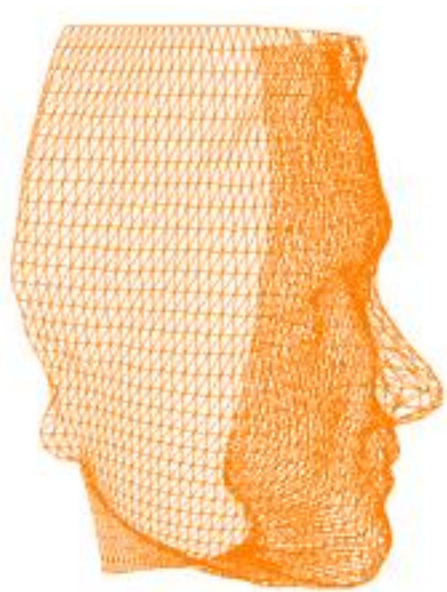
COSC422 Advanced Computer Graphics



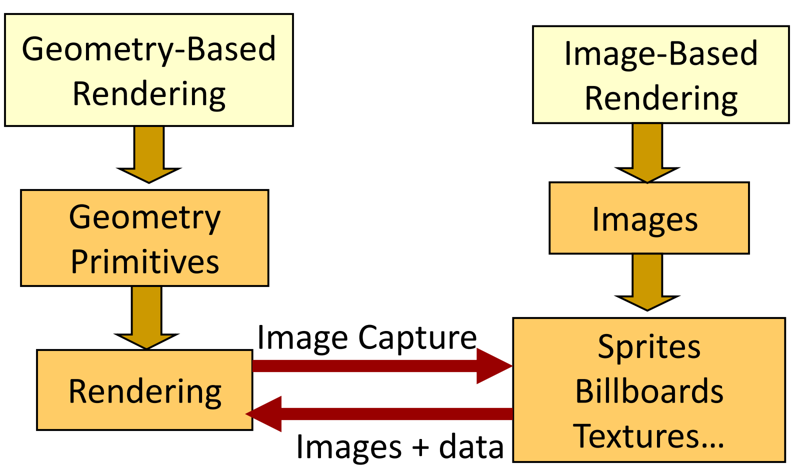
5 Image Based Rendering (IBR)

Semester 2
2021

R. Mukundan (mukundan@canterbury.ac.nz)
Department of Computer Science and Software Engineering
University of Canterbury, New Zealand.

Introduction - Rendering

- Conventional polygon-based computer graphics:
  - Geometry computations are primarily based on vertex data. Rendering complexity is proportional to number of primitives
- Image-based rendering (IBR):
  - Images used to create **sampled representations of geometry** to reduce rendering complexity. Image processing used to extract additional features such as edges.



Lecture Outline

- Impostors
- Framebuffer Objects - render scene to image
- Render to Texture (RTT)
- Depth texture
- Projective texturing
- Shadow mapping

IBR Example: Rendering Trees

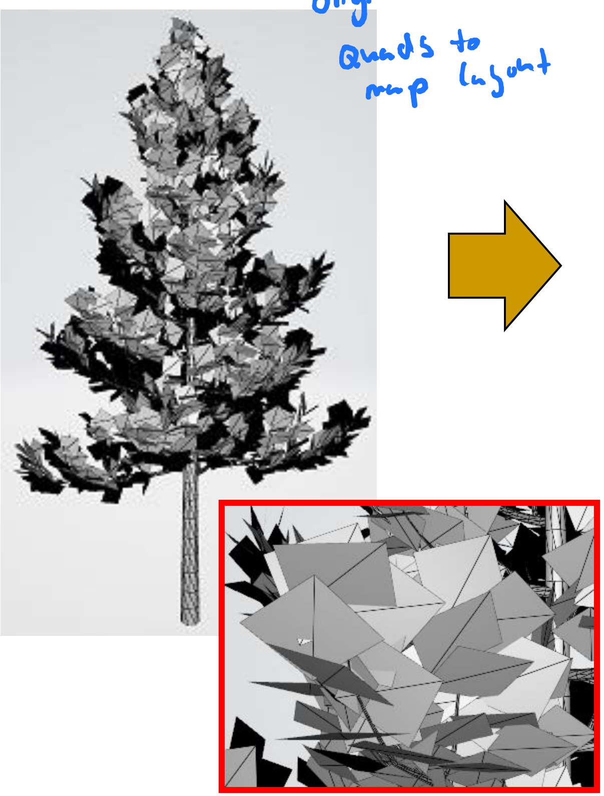
- Highly detailed models may contain millions of triangles

≈1.5 Million vertices
≈ 1.6 Million triangles
≈ 25 textures

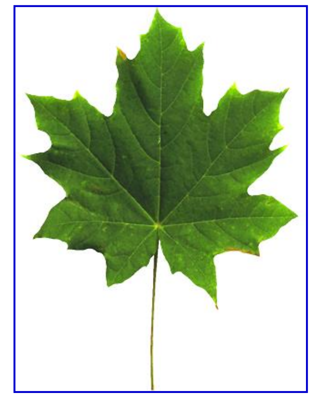
Rendering Trees: Quads

- Each leaf approximated by a textured quad. This is an example of sampled representations of geometry.
- Provides a sufficiently detailed 3D representation of trees

original Quads to map layout



. new shape + texture



RGB

Alpha

135,300 Vertices
45,000 Quads
2 Textures (leaf, stem)

Rendering Trees:

- Small branches approximated by textured quads
- Coarse-grained model

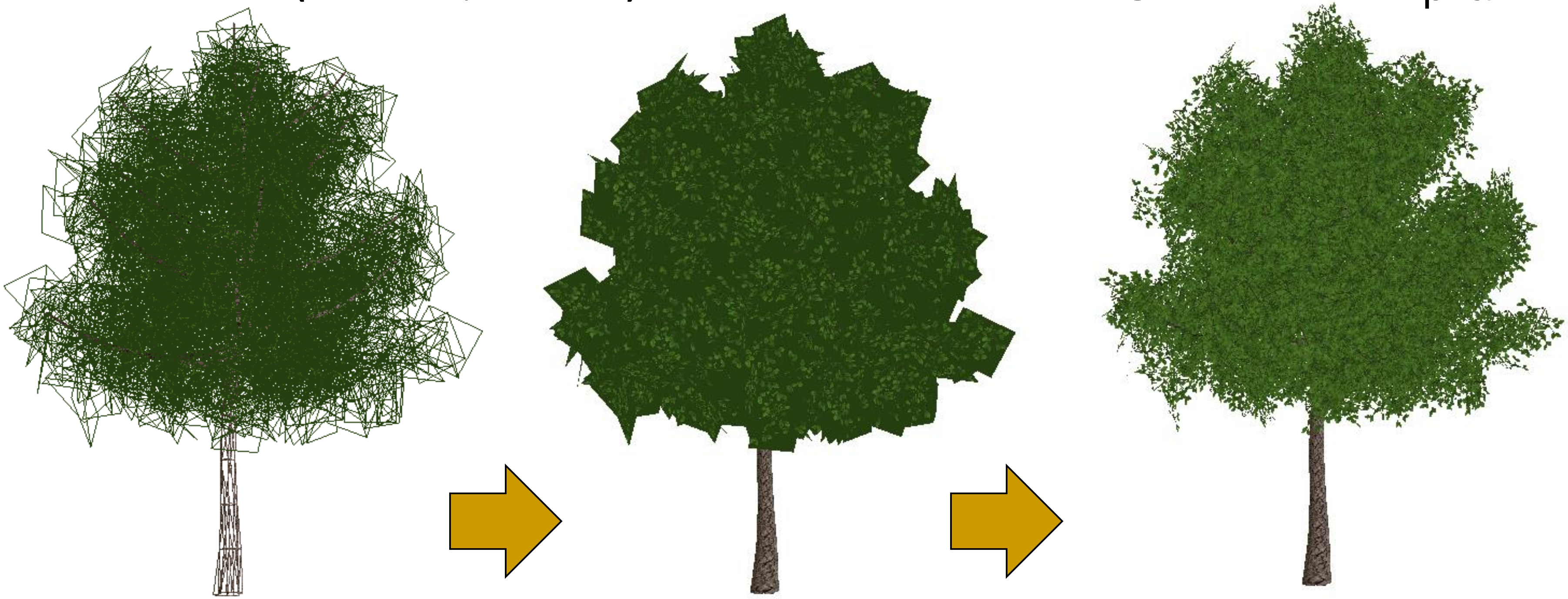
6807 Vertices

4156 Quads

2 Textures (branch, trunk)

RGB

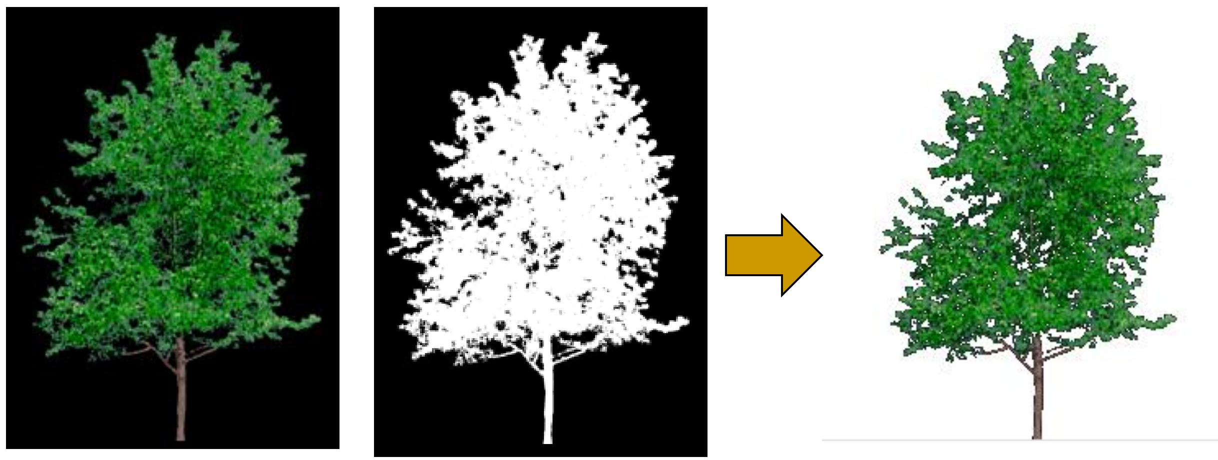
Alpha



Rendering Trees: Billboards - most commonly used

- Whole model represented by a single texture
- Limitations:
  - Provides the same view from all directions - Billbord
  - Useful only for distant objects

4 Vertices
1 Quad
1 Texture



RGB Alpha Billboard

Alpha Texturing (OpenGL2):

```
glAlphaFunc(GL_GREATER, 0.5);
glEnable(GL_ALPHA_TEST);
```

Alpha Texturing (Fragment Shader):

```
vec4 col = texture(treeTex, tcoord);
if(col.a < 0.5) discard;
```

Rendering Trees: Texture Atlas

- impostors look like 3D models
-But they are just quads (billboards)

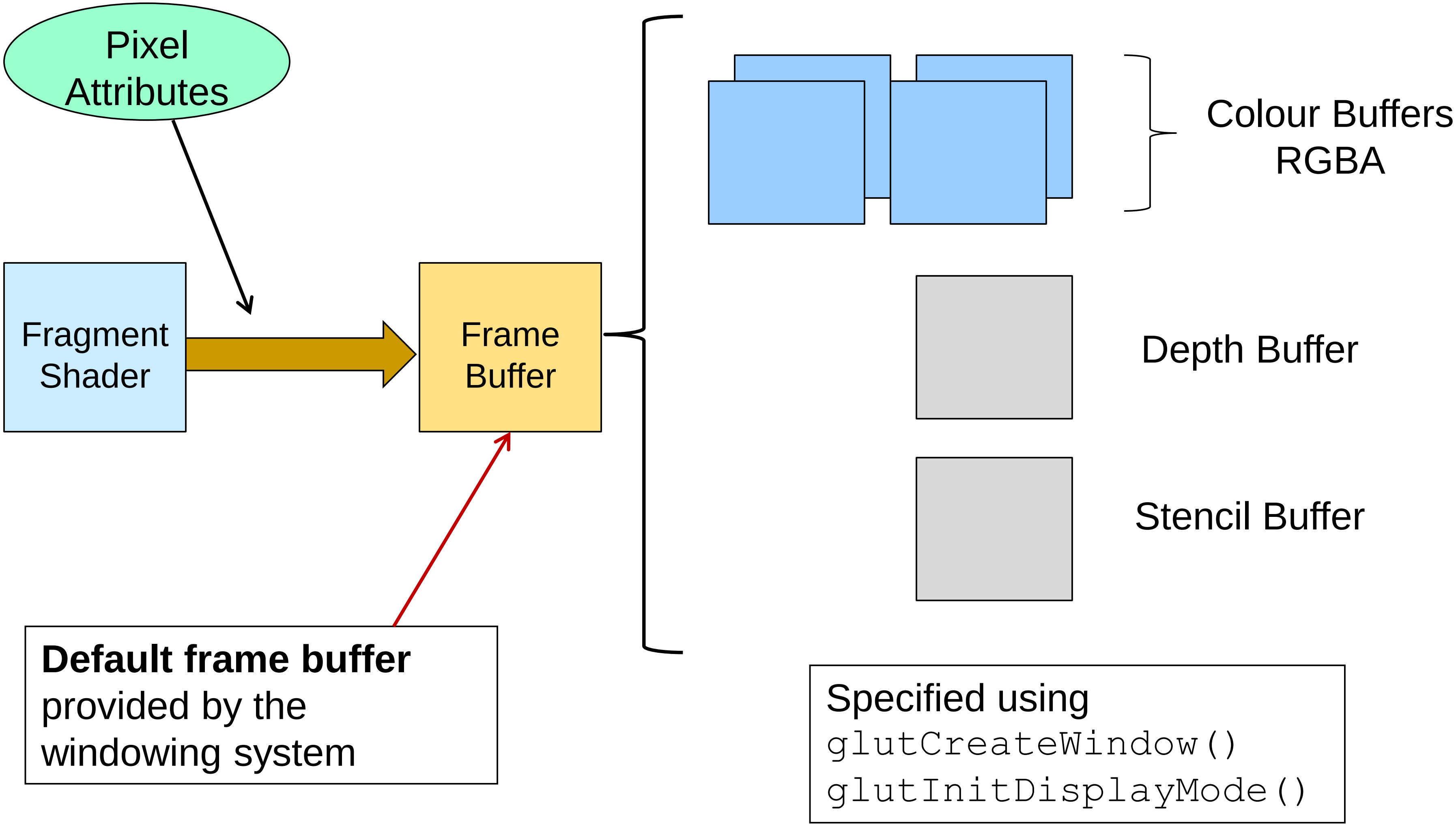
- A texture atlas containing different views of a tree are created by rendering a 3D model under a rotational transformation about the y-axis.
- The textures are then used for rendering an impostor of the model. - updating Billboard

Impostors

- Impostors are dynamic billboards. The texture is updated dynamically so as to reduce the visual error incurred by using a flat object.
- Impostors are useful for rendering distant objects rapidly.

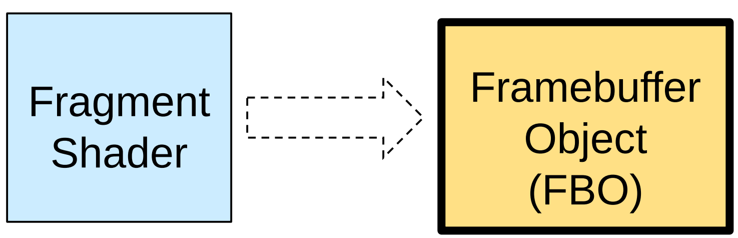
  rapid processing
- Impostors provide different geometrical perspectives of an object from different view angles - Different views
- Impostors can be created using the following two methods
  - Pre-generated textures (as in previous example)
  - Render to texture (Using Framebuffer Objects)

Default Frame Buffer



OpenGL Frame Buffer Object (FBO)

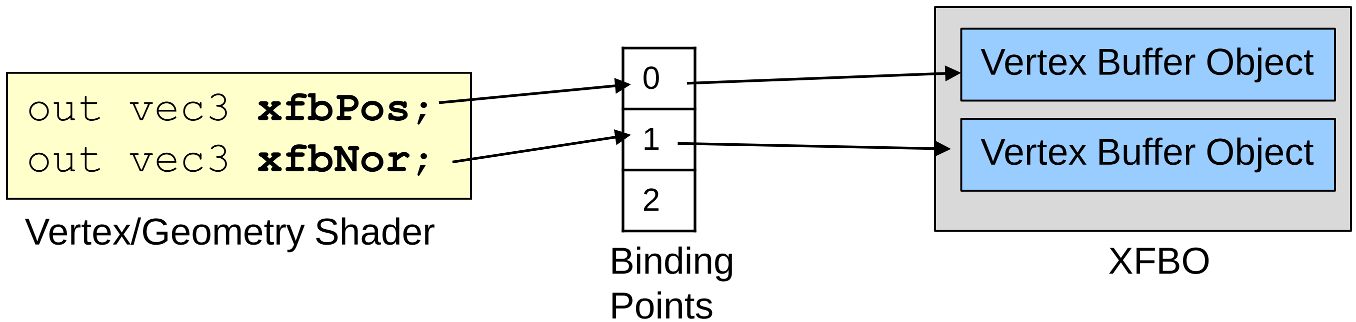
- A user-defined frame buffer that can be used to capture the outputs of the fragment shader
- The captured outputs are images that are often reused as textures in an application



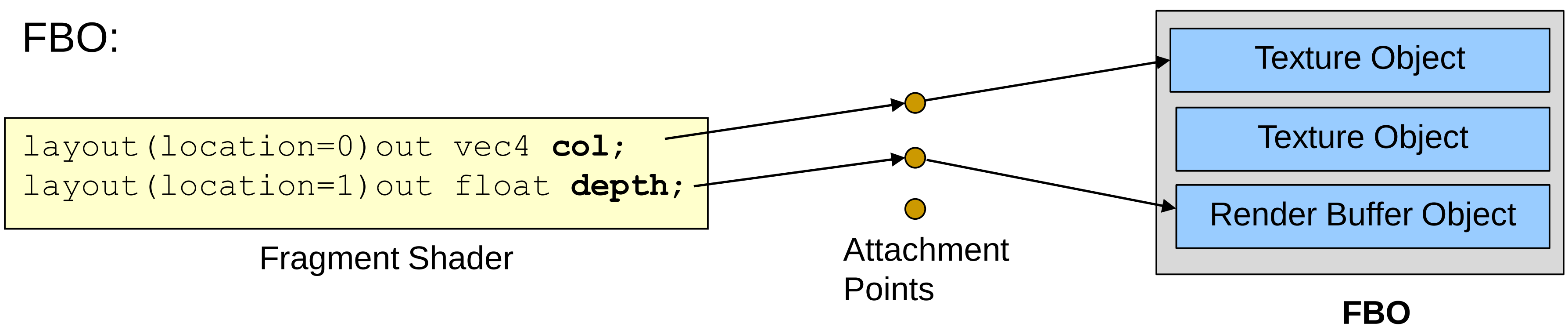
XFBO vs FBO: A Comparison

- Both XFBO and FBO provide mechanisms for capturing information from shader stages

XFBO:

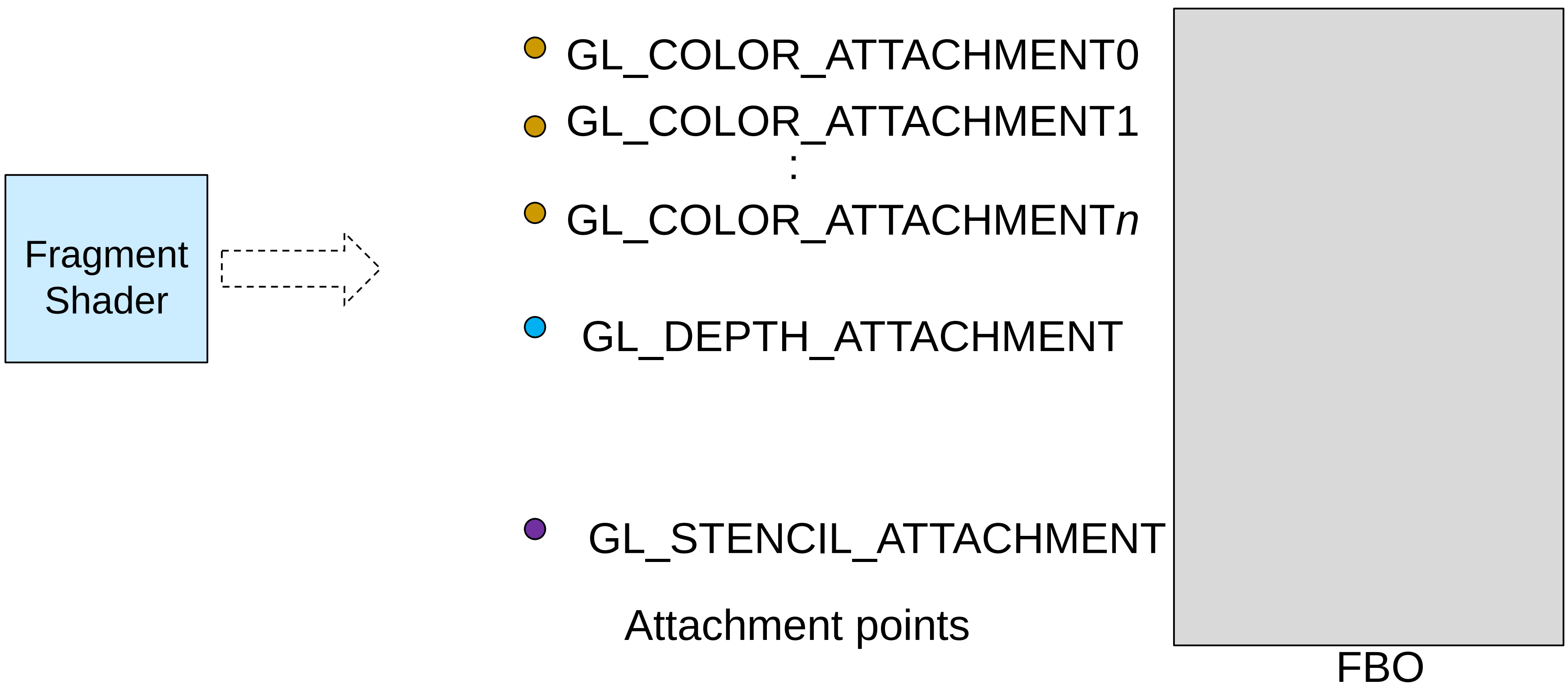


FBO:



Framebuffer Attachment Points

- A framebuffer object may be viewed as a collection of attachments.
- Attachment points specify the type and format of images that can be bound to them.



OpenGL Framebuffer Objects

- glDrawBuffers() define an array of buffers into which the outputs from the fragment shader will be written.
- A fragment shader can simultaneously output several values.

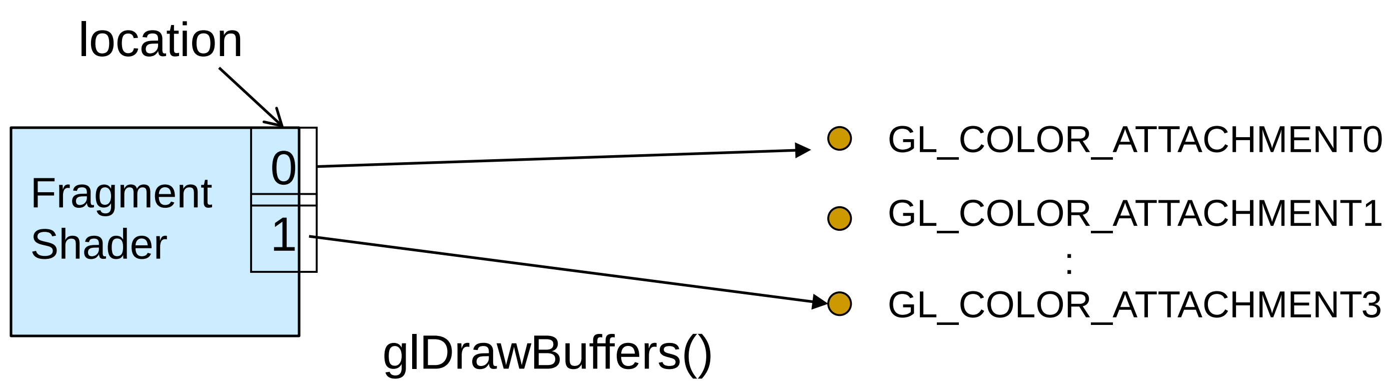
• Attaching shader & Application

Application:

```
GLenum bufs[] = {GL_COLOR_ATTACHMENT1, GL_COLOR_ATTACHMENT3};
glDrawBuffers(2, bufs);
```

Fragment shader:

```
layout (location = 0) out vec4 color;
layout (location = 1) out vec4 something;
```

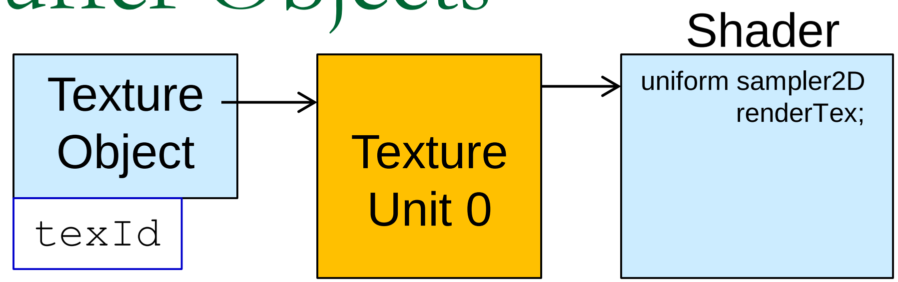


OpenGL Framebuffer Objects

A Framebuffer Object is just a container. Two types of image objects can be attached to a FBO:

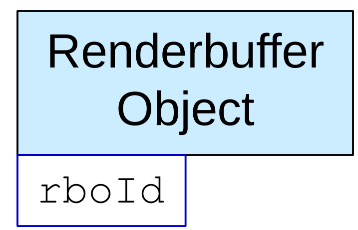
- Texture Objects (One or more images)
  - Texture objects are associated with texture memory and accessed by shaders.
  - Supports texture sampling and filtering functions
  - Commonly used for "render to texture" (RTT) operations
- Renderbuffer Objects (A single image)
  - A render buffer can hold only a single 2D image data
  - Optimized for use as render targets
  - Cannot be bound to shaders.

OpenGL Framebuffer Objects



Creating Texture Objects:

```
GLuint texId;
glGenTextures(1, &texId);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, texId);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, wid, hgt, 0, GL_RGBA,
                                        GL_UNSIGNED_BYTE, NULL);
GLuint texLoc = glGetUniformLocation(program, "renderTex");
glUniform1i(texLoc, 0);
```



Creating Renderbuffer Objects:

```
GLuint rboId;
glGenRenderbuffers(1, &rboId);
glBindRenderbuffer(GL_RENDERBUFFER, rboId);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT,
                                                 wid, hgt);
```

OpenGL Framebuffer Objects

Creating Framebuffer Objects:

```
GLuint fboId;
glGenFramebuffers(1, &fboId);
glBindFramebuffer(GL_FRAMEBUFFER, fboId);
```

Attaching a texture object to a FBO:

```
glFramebufferTexture2D (GL_FRAMEBUFFER,
           GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, texId, 0);
```

Attaching a render buffer object to a FBO:

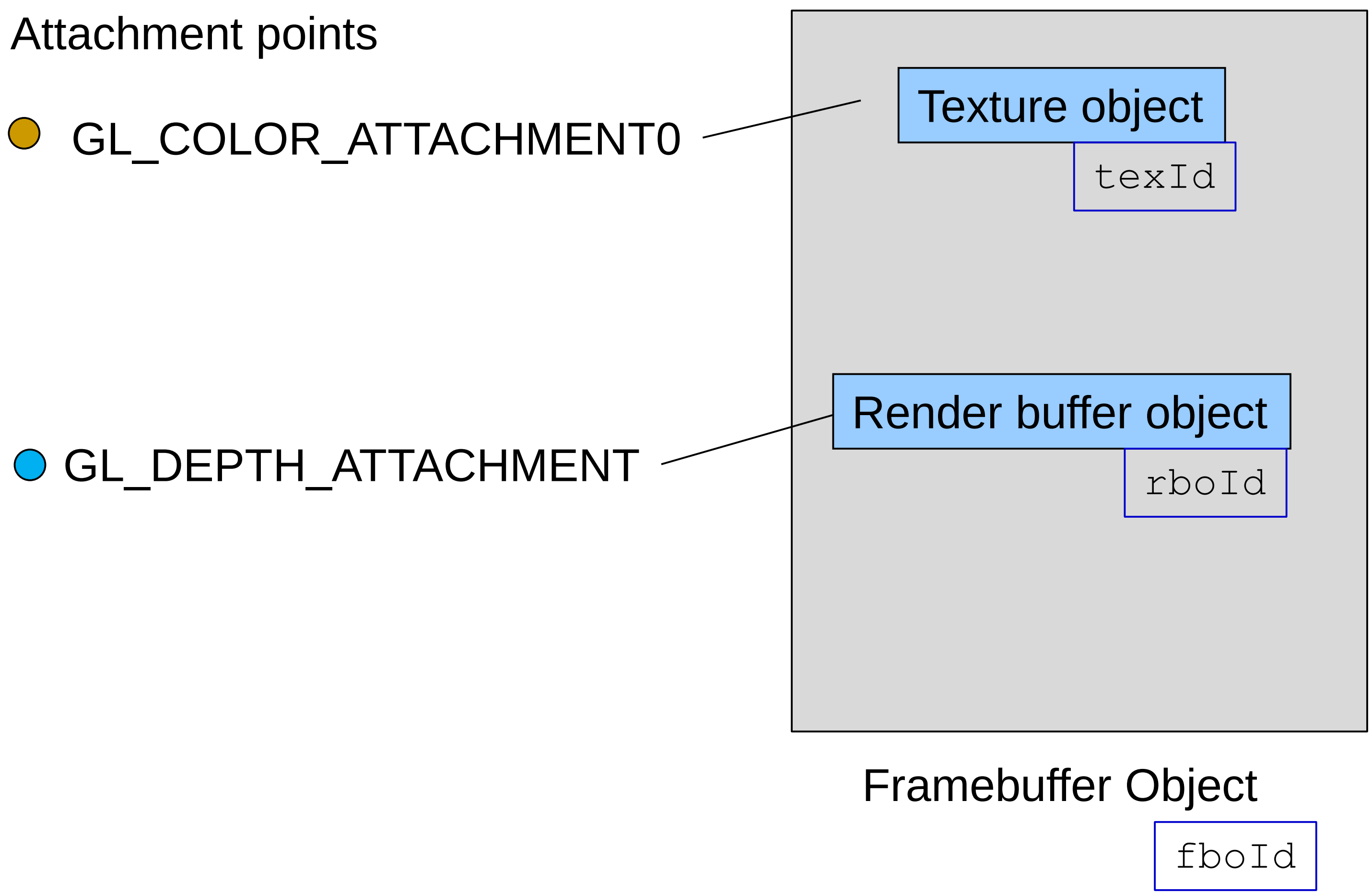
```
glFramebufferRenderbuffer(GL_FRAMEBUFFER,
              GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, rboId);
```

Checking framebuffer object completeness:

```
GLenum status =  glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status != GL_FRAMEBUFFER_COMPLETE)
                            cout << "FBO Error!" << endl;
```

OpenGL Framebuffer Object

- creating Ids

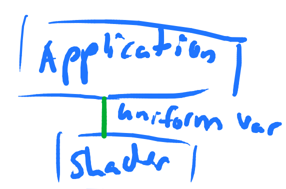


<u>Off-screen rendering</u>: A scene is rendered to a texture object or a render buffer. We cannot use the default depth buffer here for depth testing. So, we need a separate image buffer attached to GL_DEPTH_ATTACHMENT.

Rendering to Texture

- Create a FBO with a texture object attached to a colour attachment point, and a render buffer to the depth attachment point.
- In the first pass, use the FBO and render the 3D model to the texture object
- In the second pass, use the default framebuffer and render a billboard that uses the stored texture.
- A uniform variable is needed to distinguish between the two passes in the shader.

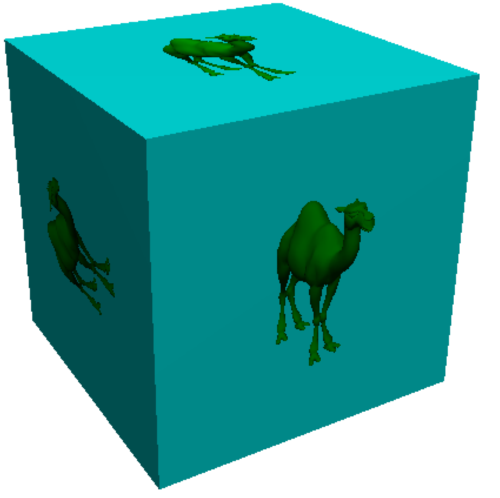
- Allows us to communicate w/ shader & application



Rendering to Texture

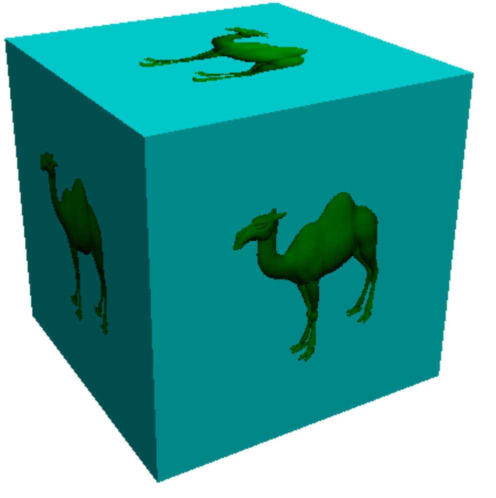
display()

```
glBindFramebuffer(GL_FRAMEBUFFER, fboID);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glUniform1i(passLoc, 0);
  mesh->render();   //Camel

glBindFramebuffer(GL_FRAMEBUFFER, 0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glUniform1i(passLoc, 1);
   cube->render();
```



Fragment shader:

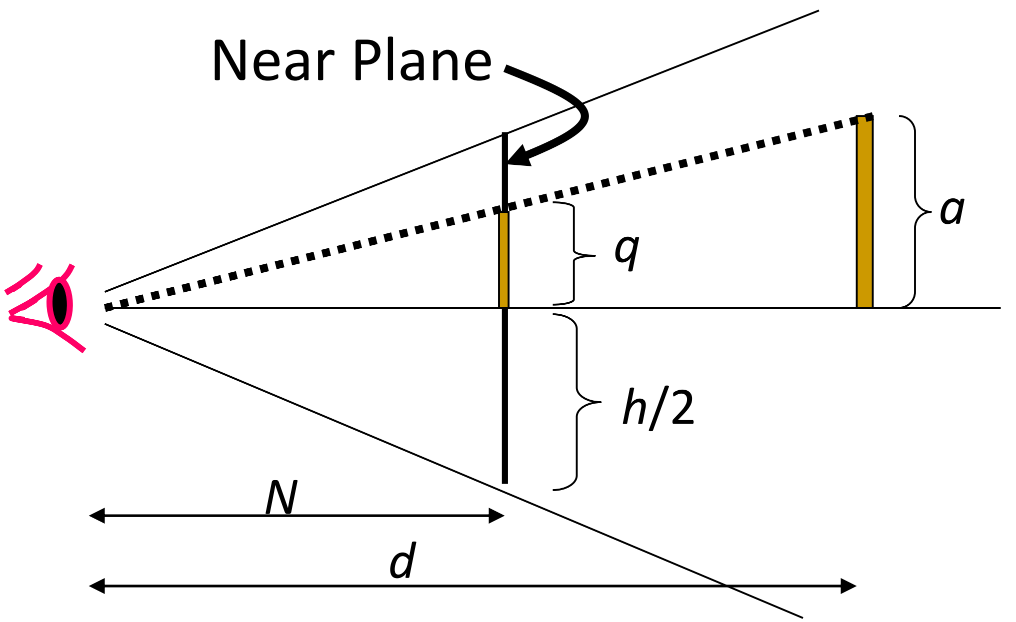
```
layout (location = 0) out vec4 oColor;
uniform sampler2D renderTex;
uniform int pass;
in vec4 vColour;
in vec2 TexCoord;
void main() {
 if(pass == 0) oColor = vColour;
 else
  oColor = vColour * texture(renderTex, TexCoord);
}
```



Impostor Resolution

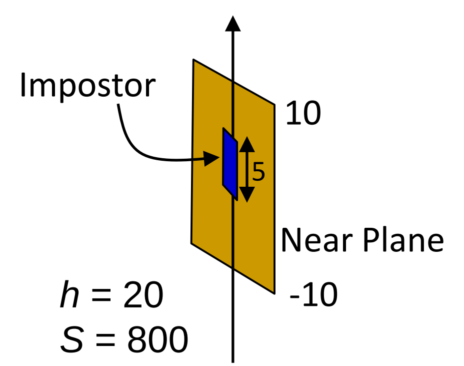
Consider an impostor with height ‘a’ at a distance ‘d’ from the camera. Let *h* units be the height of the display window with a screen resolution *S*.

The impostor has a projected height ‘q’ on the screen containing *qS/h* pixels. This is the minimum number of pixels required for the impostor texture.



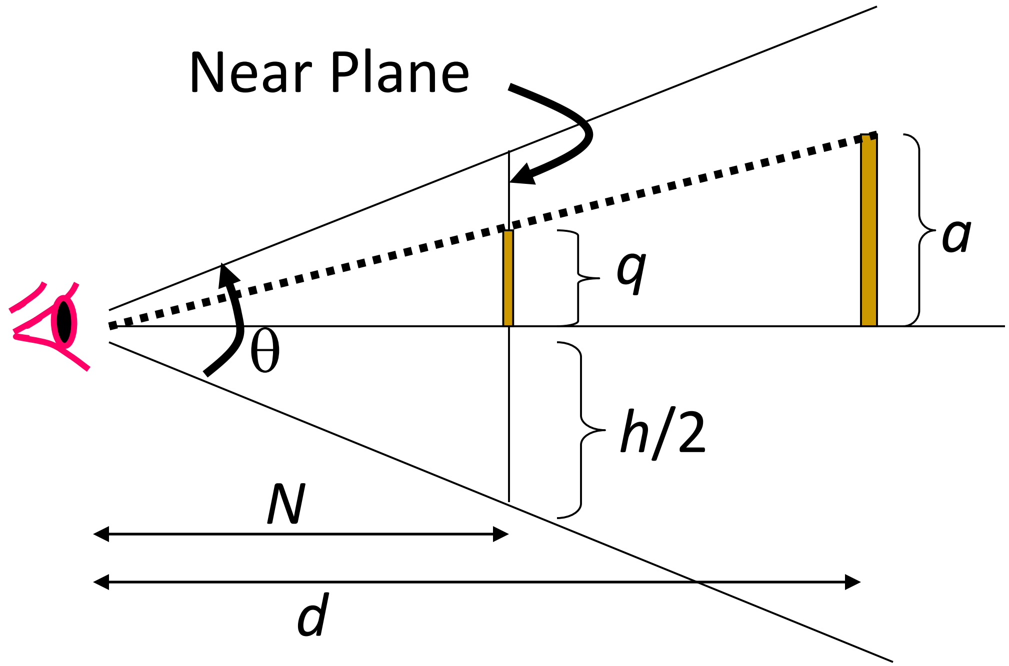
Example:

```
glFrustum(-10, 10, -10, 10, 5, 100)
glutInitWindowSize(800, 800);
```



$h = 20$
$S = 800$
$q = 5$
$qS/h = 200$ pixels

Impostor Resolution



$a/d = q/N$.

$h/(2N) = \tan(\theta/2)$

$\therefore$ Required impostor resolution = qS/h = $aNS/(dh)$ = $$\frac{S\,a}{2d\,.\tan\left(\frac{\theta}{2}\right)}$$

Silhouette Edges using FBO

- Render an object to a texture in the first pass (as in previous slide)
- In the second pass, render a single quad with the texture mapped to it. The projected size of the quad must be approximately the size of the texture (wid x hgt).
  - Each fragment of the quad corresponds to a pixel of the rendered texture.
  - Inside the fragment shader, we access this pixel as

    col = texture(renderTex, TexCoord);
  - The neighbouring pixels can be accessed using offsets to the texture coordinates:

    TexCoord.s $\pm$ (1/wid)

    TexCoord.t $\pm$ (1/hgt)

Silhouette Edges using FBO

- Having obtained the eight neighbouring pixels for each pixel in the fragment shader, we perform two operations:
- RGB→Intensity conversion:

  $I = (0.299)R + (0.587)G + (0.114)B$

- Sobel filter for edge detection:

S_x:

1		-1
2		-2
1		-1

Vertical Edge Detector

S_y:

1	2	1
-1	-2	-1

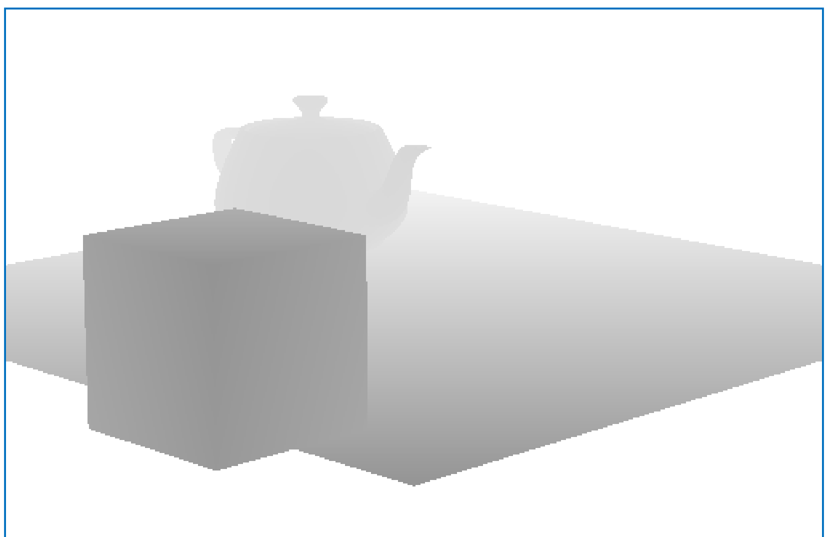
Horizontal Edge Detector

$$g = \sqrt{s_x^2 + s_y^2}$$

If g is above a certain threshold, the current fragment is rendered with the edge colour.

Depth Texture

- Depth values range from 0 (Near plane) to 1 (Far plane).
- Rendering the depth values to a texture is useful in some applications
  - Silhouette edge detection
  - Shadow mapping (discussed later)



Depth map

Depth Texture (FBO)

GL_COLOR_ATTACHMENT0

GL_COLOR_ATTACHMENT*n*

Texture Object

GL_DEPTH_ATTACHMENT

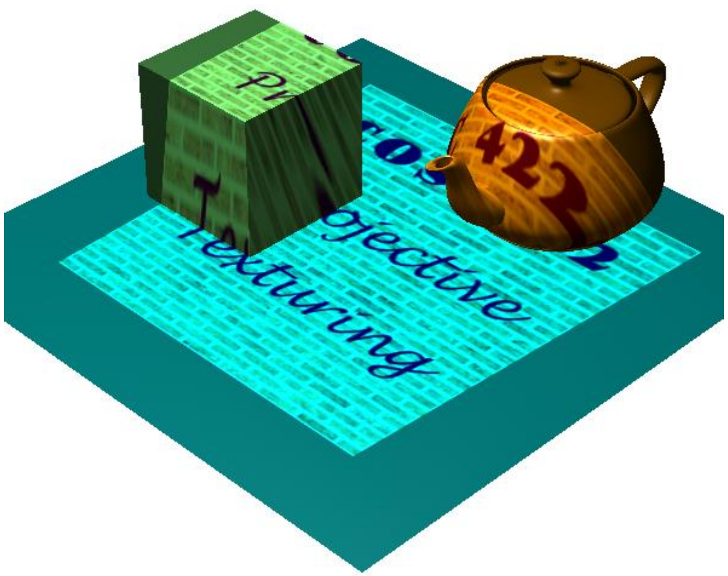
```
glGenFramebuffers(1, &fboID);
glBindFramebuffer(GL_FRAMEBUFFER, fboID);

GLuint depthTex;
glGenTextures(1, &depthTex);
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, depthTex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, 512, 512,  0,
                                GL_DEPTH_COMPONENT, GL_FLOAT,  NULL);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, depthTex,  0);
GLuint texLoc = glGetUniformLocation(program, "depthTex");
glUniform1i(texLoc, 0);

glDrawBuffer(NULL);    //Not using any colour buffer
```

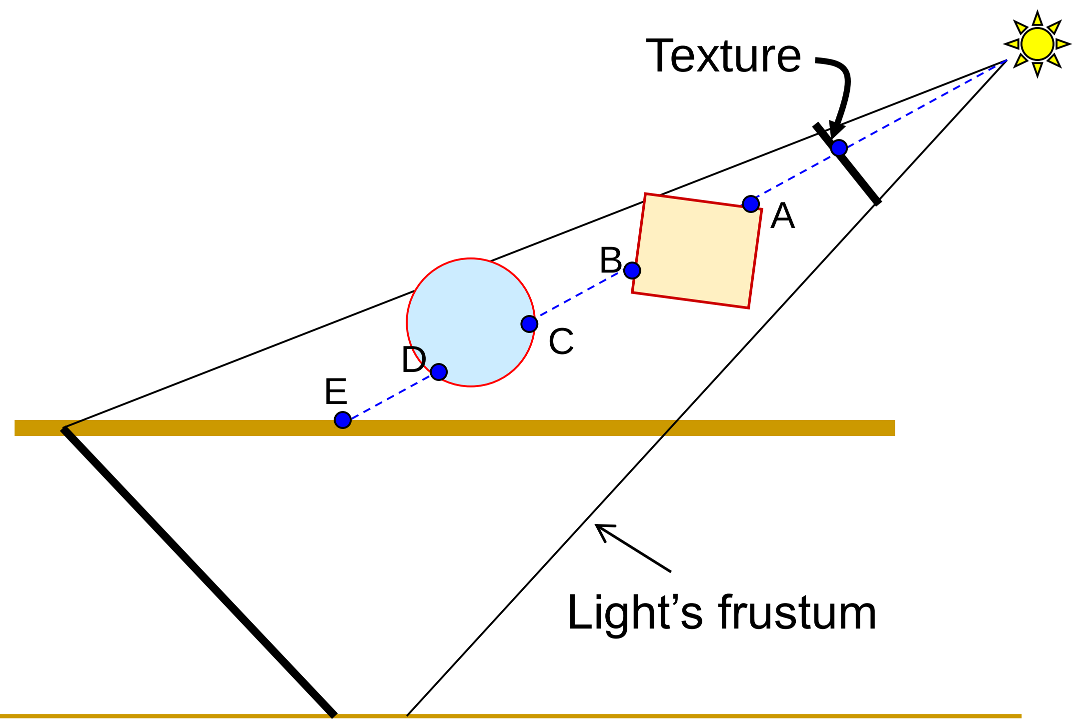
Depth Texture (Fragment Shader)

```
#version 330

uniform  sampler2D  depthTex;
uniform int pass;

in vec2 TexCoord;

void main()
{
  float depth;
  if(pass == 0) gl_FragColor = vec4(0);          //Render the scene
  else
  {
    depth = texture(depthTex, TexCoord).r;       //Render a  quad
    gl_FragColor = vec4(depth);
  }
}
```

Projective Texturing

- Maps a texture to parts of a scene to generate the effect of projecting an image from a source.
- Texture coordinates are computed using a frustum attached to the light source.





Projective Texturing

- We define the view and projection matrices for the light source!

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \underbrace{\begin{bmatrix} Light \\ Frustum \end{bmatrix}\begin{bmatrix} Light \\ View \end{bmatrix}\begin{bmatrix} Model \\ Matrix \end{bmatrix}}_{\text{Light's model-view-projection matrix.}} P$$

- Every vertex P is transformed into clip coordinates (x, y, z, w).
- If a point P is inside light's frustum, then

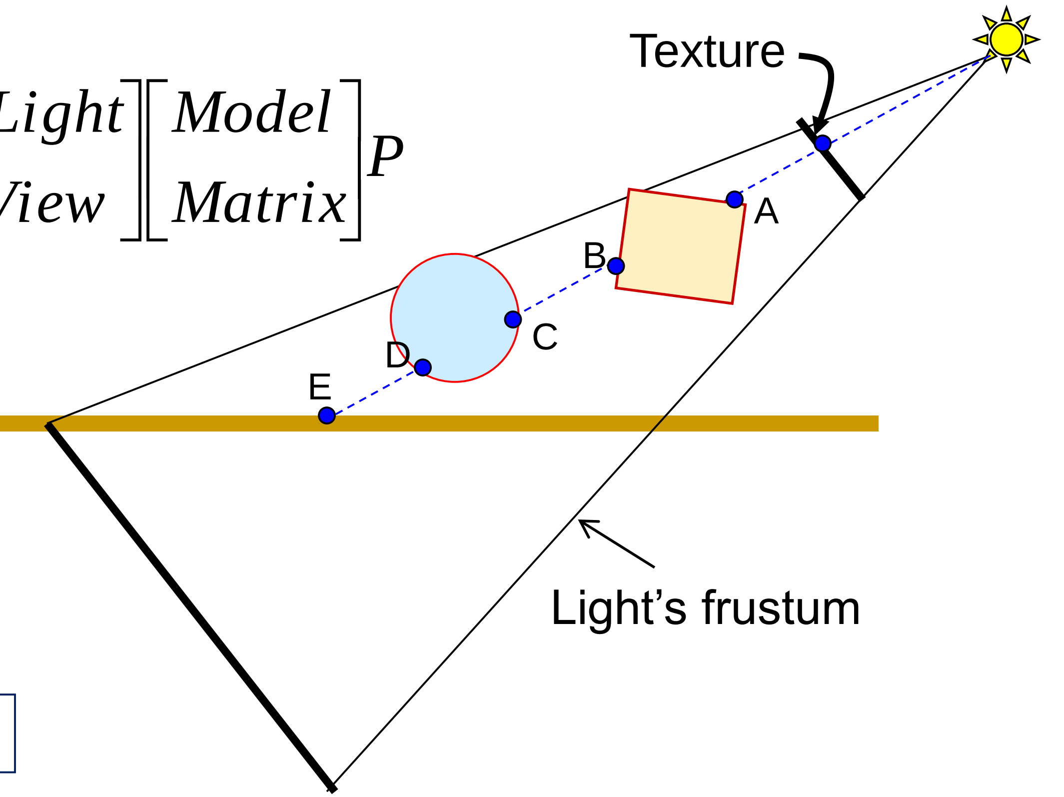
$$-1 \le (x/w) \le 1, \quad -1 \le (y/w) \le 1, \quad -1 \le (z/w) \le 1$$

- We can convert these values to texture coordinates in the range [0, 1] by scaling by (½) and adding (½).

Projective Texturing

$$\begin{bmatrix} s \\ t \\ p \\ q \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Light \\ Frustum \end{bmatrix} \begin{bmatrix} Light \\ View \end{bmatrix} \begin{bmatrix} Model \\ Matrix \end{bmatrix} P$$

Texture Coords: (s/q , t/q).



The light's model-view-projection matrix is pre-multiplied by the scale transformation matrix to get the projective texturing matrix.

p/q gives the normalized depth of the point in the light's frustum.

Projective Texturing

Vertex Shader

```
uniform mat4 projTexMatrix;
out vec4 vColor;
out vec4 texCoord;
void main()
{
   ...
   gl_Position = mvpMatrix * position;
   vColour = lgtAmb + lgtDiff + lgtSpec;
   texCoord = projTexMatrix * position;
}
```

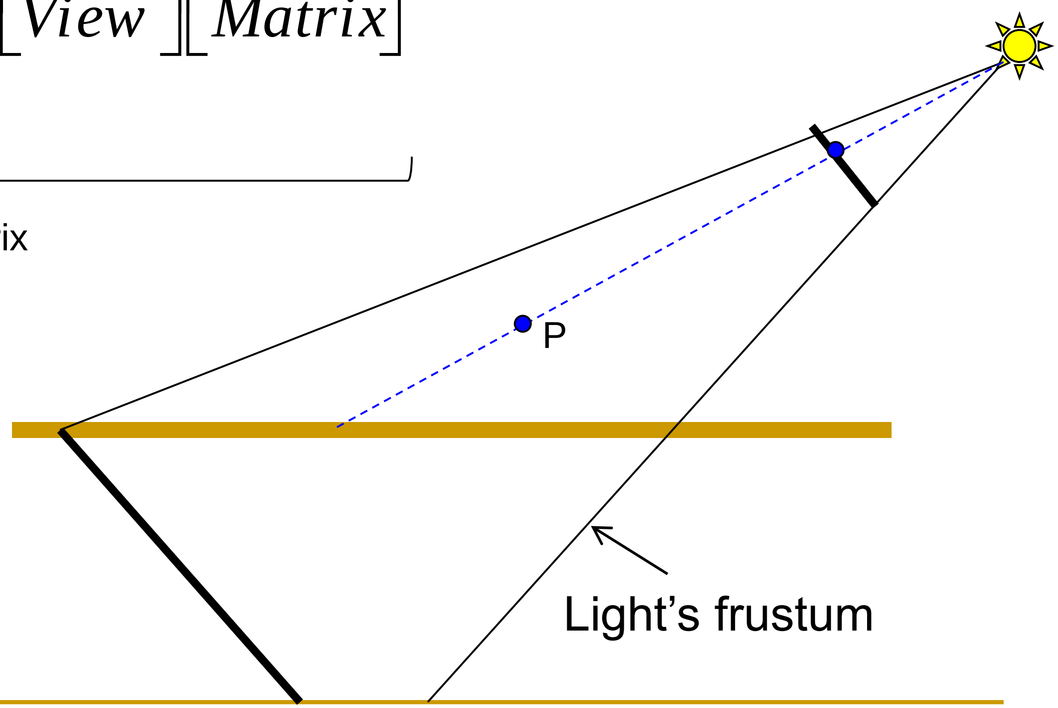
Fragment Shader

```
uniform sampler2D texBrick;
in vec4 vColour;
in vec4 texCoord;      ————→ (s, t, p, q)

void main() {
   float x = texCoord.s/texCoord.q;
   float y = texCoord.t/texCoord.q;
   if(x < 0 || x > 1 || y < 0 || y > 1)
     gl_FragColor = 0.4 * vColour;
   else
     gl_FragColor  = vColour * textureProj(texBrick, texCoord);
}
```

Light's Frustum

The matrix on slide 28 will now be called the shadow matrix. It transforms any point *P* to coordinates (s, t, p, q) such that if the point is within the frustum, s/q, t/q $\in$ [0, 1].

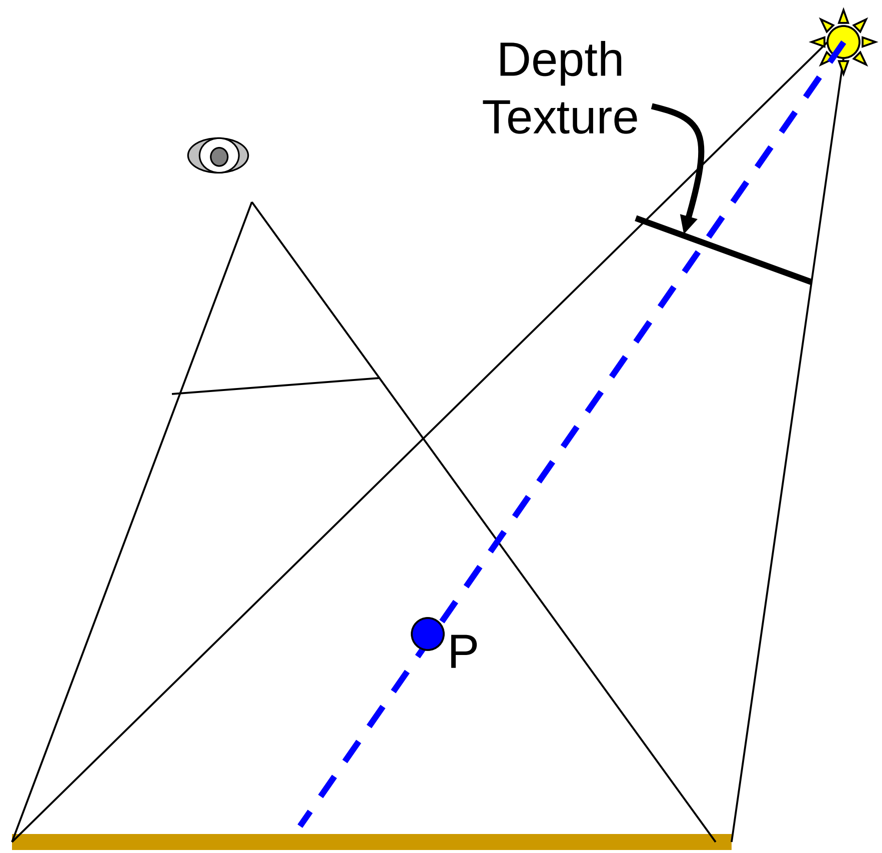
$$\begin{bmatrix} s \\ t \\ p \\ q \end{bmatrix} = \underbrace{\begin{bmatrix} \frac{1}{2} & 0 & 0 & \frac{1}{2} \\ 0 & \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & 0 & \frac{1}{2} & \frac{1}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} Light \\ Frustum \end{bmatrix} \begin{bmatrix} Light \\ View \end{bmatrix} \begin{bmatrix} Model \\ Matrix \end{bmatrix}}_{\text{Shadow Matrix}} P$$



Shadow Mapping

Shadow mapping is a technique that uses projective texturing for generating complex shadows.

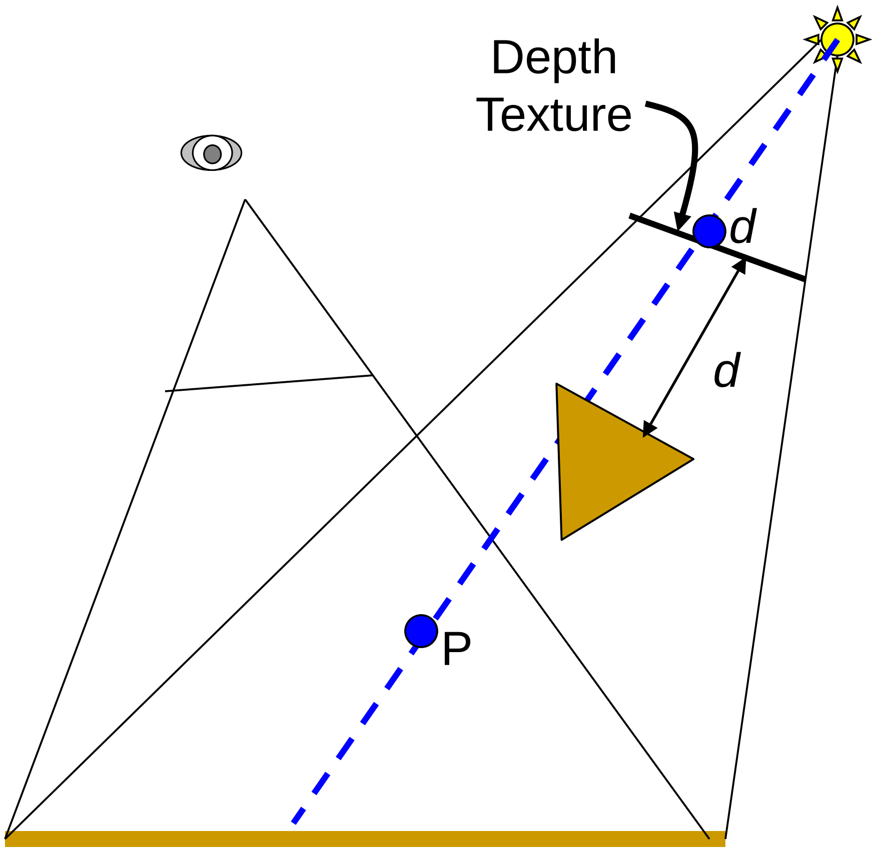
- Render the scene from light's point of view, and generate a depth texture (Slides 23, 24).
- Render the scene from the eye position, with each vertex also transformed to (s, t, p, q) in the vertex shader.
- The fragment shader has the interpolated values of (s, t, p, q) for each fragment, and also access to the depth texture.



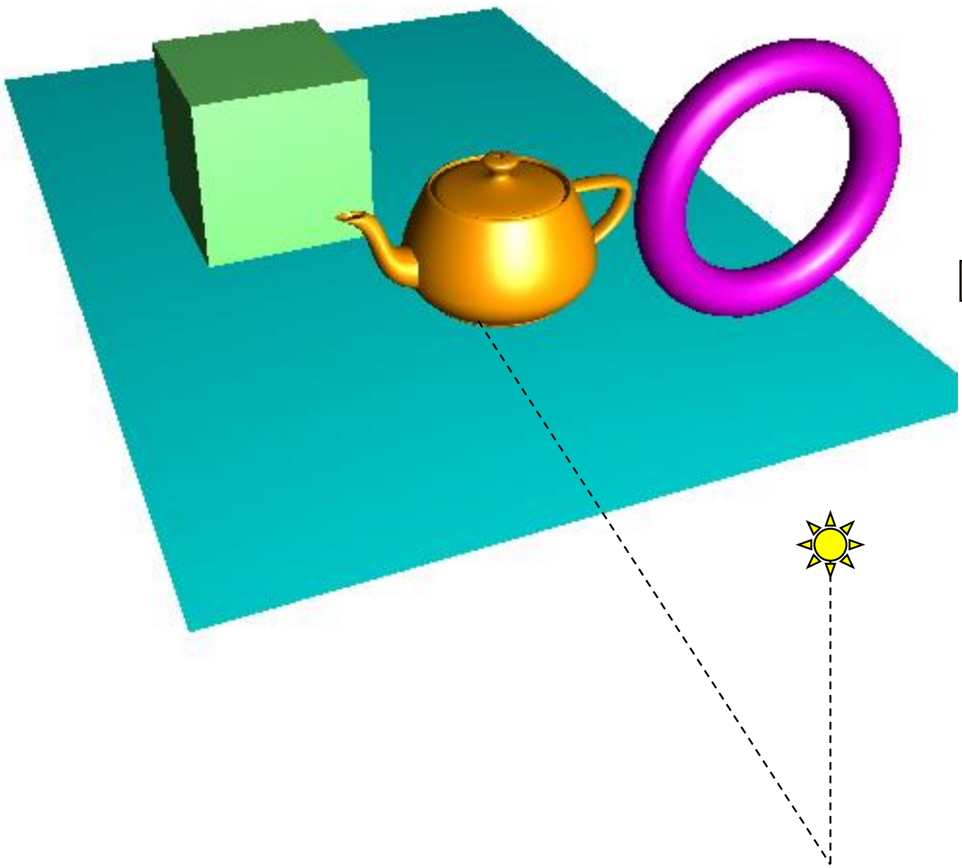
The depth texture from the light source is generally referred to as the shadow map.

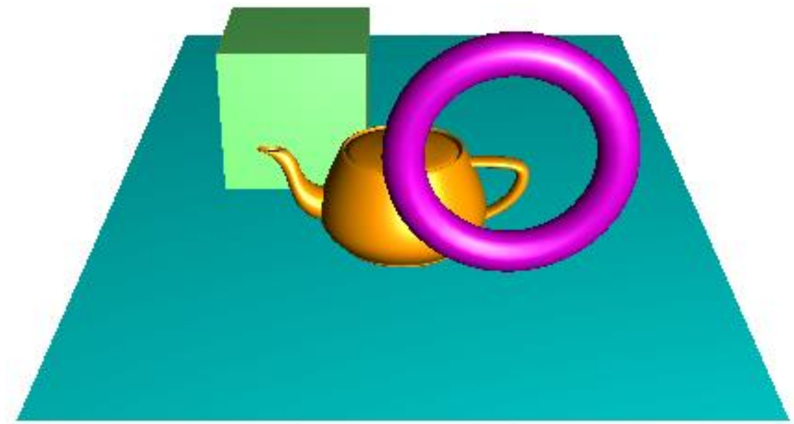
Shadow Mapping

- Using the values (s, t, p, q) of the current fragment, we use (s/q, t/q) as the texture coordinates to access the depth texture.
- The value *d* returned by the depth texture represents the depth of the closest point along that direction.
- The value p/q gives the true depth of the point *P* in light's frustum.
- If *d* < *p/q*, the fragment is in shadow.



Shadow Mapping (Step 1)





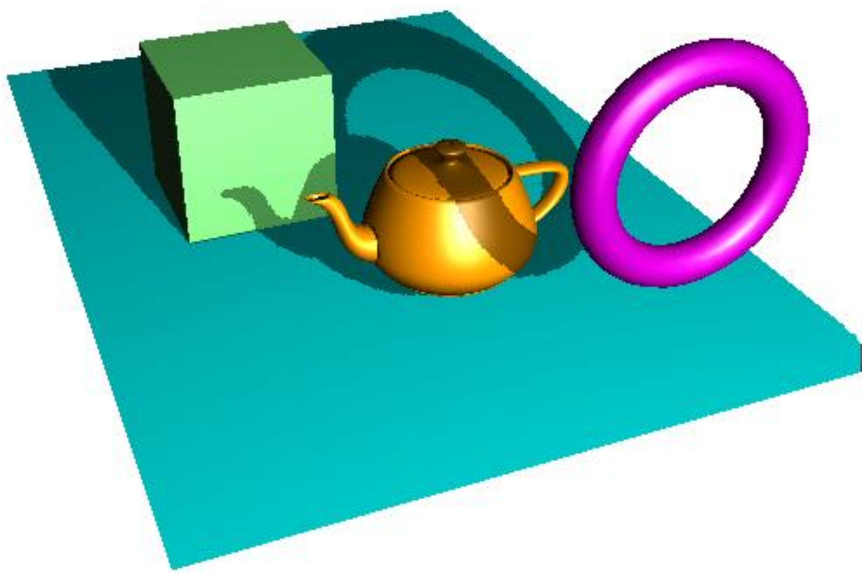
View from light

Depth texture
(Shadow map)

Render the scene from the light's point of view, and generate the depth map.

Shadow Mapping (Step 2)

- Render the scene from the eye position, and compare the value *d* of the depth texture at location (*s*/*q*, *t*/*q*) with the depth of the fragment *p*/q.
- Apply a fragment colour based on the result of the comparison.
- Common artifacts: depth-fighting, aliasing.



Shadow Mapping

Application:

Slide 24

```
void display()
{
glBindFramebuffer(GL_FRAMEBUFFER, fboID);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glUniform1i(passLoc, 0);

proj  = glm::perspective(40.0f, 1.0f, 10.0f, 50.0f);
view = glm::lookAt(light, glm::vec3(0.0, 0.0, 0.0),
                               glm::vec3(0.0, 1.0, 0.0));
  ...
scene();

glBindFramebuffer(GL_FRAMEBUFFER, 0);
glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
glUniform1i(passLoc, 1);
view = glm::lookAt(glm::vec3(12.0, 8.0, 5.0),
       glm::vec3(0.0, 0.0, 0.0), glm::vec3(0.0, 1.0, 0.0));
  ...
scene();
```

First pass

Second pass

Shadow Mapping

Shadow matrix

Vertex Shader:

```
...
uniform mat4 projTex;
...
out vec4 vColour;
out vec4 texCoord;
void main() {
  ...
  gl_Position = mvpMatrix * position;
  vColour = lgtAmb + lgtDiff + lgtSpec;
  texCoord = projTex * position;
}
```

Shadow Mapping

Fragment Shader: Shadow map (depth texture)

```
uniform sampler2D depthTex;
uniform int pass;

in vec4 vColour;
in vec4 texCoord;

void main()
{
   float depth_t, depth_f, scale;

   if(pass == 0) gl_FragColor = vec4(0);   //First rendering pass
   else                                     //Second rendering pass
   {
     depth_t = textureProj(depthTex, texCoord);  //depth from texture
     depth_f = texCoord.p/texCoord.q;             //fragment's depth
     scale = 1;
     if(depth_t < depth_f - 0.0001) scale = 0.5;   //in shadow!
     gl_FragColor = vec4(scale * vColour.rgb, 1);
   }
}
```