Shadows

Prof. George Wolberg
Dept. of Computer Science
City College of New York

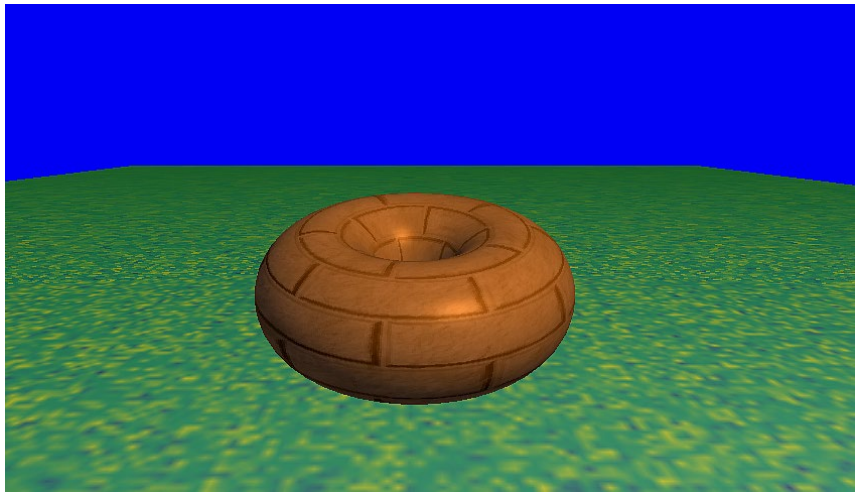
Objectives

- Introduce shadow algorithms
  - Projective shadows
  - Shadow volumes
  - Shadow mapping
  - PCF filtering for soft shadows

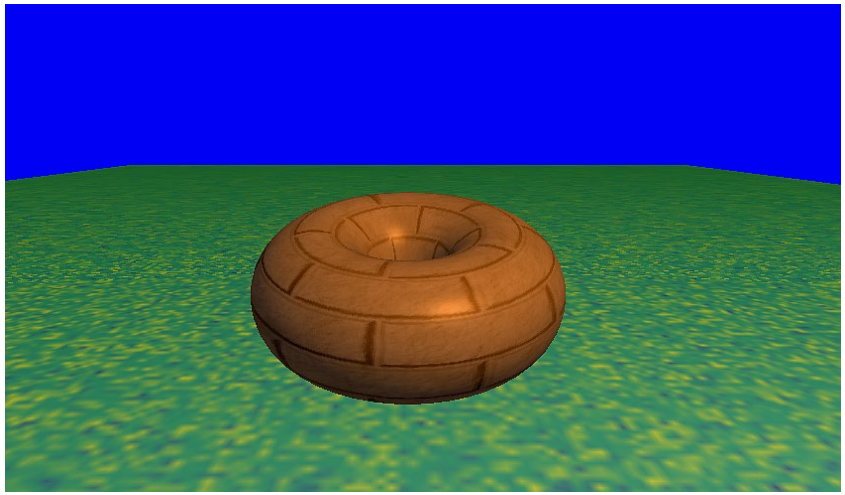
Importance of Shadows

- ADS model added lighting to our 3D scenes
- However, it didn't actually add light
- It simulated the effects of light on objects
- The limitations become apparent when lighting more than one object in the same scene

- Missing shadows in scene
- Ambiguous location of models
- Is torus resting on plane?
- Is torus floating above it?
- Impossible to know without shadows



Importance of Shadows



???

Resting on ground plane

Floating on ground plane

Flashlight in the Eye Graphics

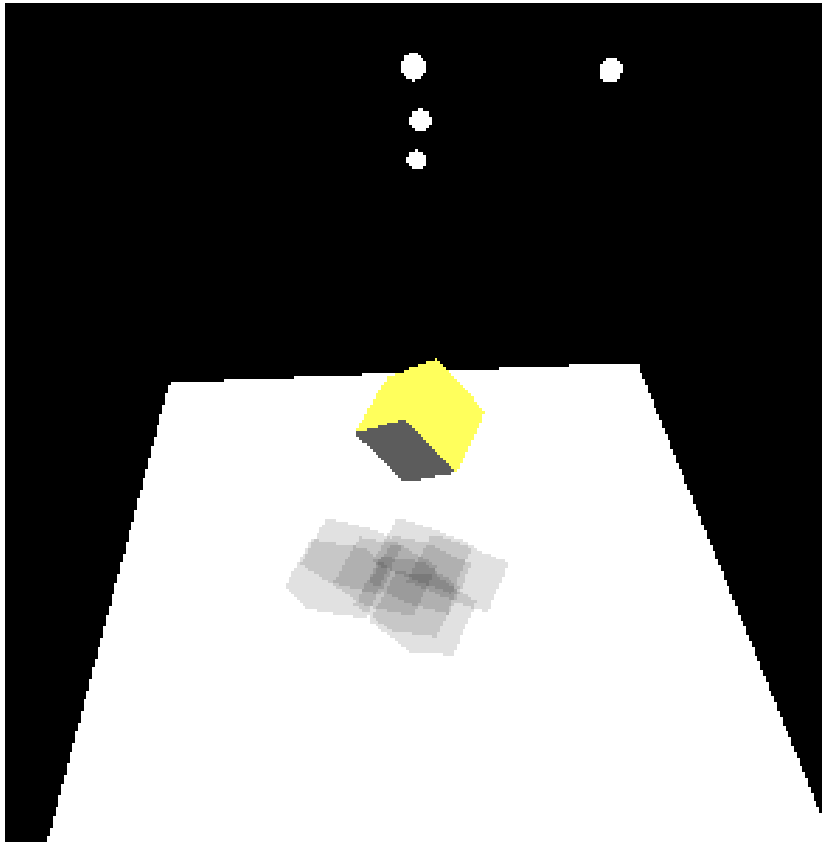
- When do we not see shadows in a real scene?
  - When the only light source is a point source at the eye or center of projection
  - Shadows are behind objects and not visible
- Shadows are a global rendering issue
  - Is a surface visible from a light source
  - May be obscured by other objects

Projective Shadows

- Oldest shadow method
  - Used in flight simulators to provide visual clues
- Projection of a polygon is a polygon called a **shadow polygon**
- Given a point light source and a polygon, the vertices of the shadow polygon are the projections of the original polygon's vertices from a point light source onto a surface

Planar Projected Shadows

- Easy to implement
- Good results when casting shadows onto one plane
- Projects model onto a plane in just one matrix multiplication
- Very fast
- Can be extended to cast onto multiple planes, but each plane will require another rendering of the model.



Planar Projected Shadows

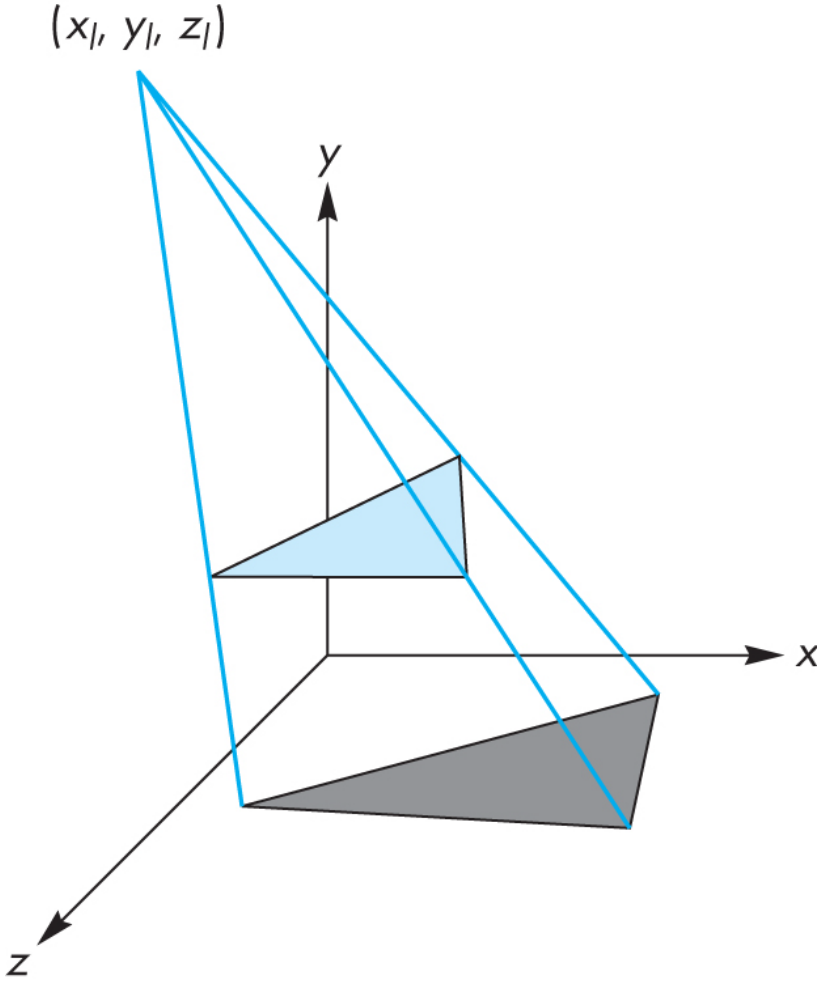
- Plane Eq:

$Ax + By + Cz + D = 0$

$$\begin{bmatrix} By + Cz & -Bx & -Cx & -Dx \\ -Ay & Ax + Cz & -Cy & -Dy \\ -Az & -Bz & Ax + By & -Dz \\ 0 & 0 & 0 & Ax + By + Cz \end{bmatrix}$$

- [x, y, z] is the difference between the center of the model and the light source.
- Ideally you want to use the distance from each vertex, However it is not as fast since the matrix will have to be computed on a per vertex basis instead of a per model basis.
- The vertices of the model are multiplied by the Planar Projection matrix.

Shadow Polygon



Computing Shadow Vertex

1. Source at (x_l, y_l, z_l)
2. Vertex at (x, y, z)
3. Consider simple case of shadow projected onto ground at $(x_p, 0, z_p)$
4. Translate source to origin with $T(-x_l, -y_l, -z_l)$
5. Perspective projection

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix}$$

6. Translate back

Shadow Process

1. Put two identical triangles and their colors on GPU (black for shadow triangle)
2. Compute two model-view matrices as uniforms
3. Send model-view matrix for original triangle
4. Render original triangle
5. Send second model-view matrix for shadow triangle
6. Render shadow triangle

- Note shadow triangle undergoes two transformations
- Note hidden surface removal takes care of depth issues

Generalized Shadows

- Approach was OK for shadows on a single flat surface
- Note with geometry shader we can have the shader create the second triangle
- Cannot handle shadows on general objects
- There exists a variety of other methods based on same basic idea
- We'll pursue methods based on projective textures

Image Based Lighting

- We can project a texture onto the surface in which case we are treating the texture as a “slide projector”
- This technique is the basis of projective textures and image based lighting
- Supported in OpenGL and GLSL through four-dimensional texture coordinates

4D Textures Coordinates

- Texture coordinates (s, t, r, q) are affected by a perspective division so the actual coordinates used are (s/q, t/q, r/q) or (s/q, t/q) for a two dimensional texture
- GLSL has a variant of the function texture *textureProj* which will use the two- or three-dimensional texture coordinate obtained by a perspective division of a 4D texture coordinate a texture value from a sampler

```
color = textureProj(my_sampler, tex_coord)
```

Shadow Maps

- If we render a scene from a light source, the depth buffer will contain the distances from the light source to each fragment.
- We can store these depths in a texture called a **depth map** or **shadow map**
- Note that although we don't care about the image in the shadow map, if we render with some light, anything lit is not in shadow.
- Form a shadow map for each source

Final Rendering

- During the final rendering we compare the distance from the fragment to the light source with the distance in the shadow map
- If the depth in the shadow map is less than the distance from the fragment to the source, the fragment is in shadow (from this light source)
- Otherwise we use rendered color

Application's Side

- Start with vertex in object coordinates
- Want to convert representation to texture coordinates
- Form LookAt matrix from light source to origin in object coordinates (MVL)
- From projection matrix for light source (PL)
- From a matrix to convert from [-1, 1] clip coordinates to [0, 1] texture coordinates
- Concatenate to form object to texture coordinate matrix (OTC)

Vertex Shader

```
uniform mat4 modelview;
uniform mat4 projection;
uniform normalmatrix;          // for diffuse lighting
uniform mat4 otc;              // object to texture coordinate
uniform vec4 diffuseproduct;   // diffuse light*diffuse reflectivity

in vec4 vPosition;
in vec4 normal;

out vec4 color;
out vec4 shadowCoord;

void main()
{
// compute diffuse color as usual
// using normal, normal matrix, diffuse product
   color = ...

   gl_Position = projection*modelview*vPosition;
   shadowCoord = OTC*vPosition;
}
```

textureProj function

- Application provides the shadow map as a texture object
- The GLSL function **`textureProj`** compares the third value of the texture coordinate with the third value of the texture image
- For nearest filtering of the texture object, **`textureProj`** returns 0.0 if the shadow map value is less than the third coordinate and 1.0 otherwise
- For other filtering options, **`textureProj`** returns values between 0.0 and 1.0

Fragment Shader

```
uniform sampler2DShadow ShadowMap;

in vec4 shadowCoord;
in vec4 Color;

main()
{
  // assume nearest sampling in ShadowMap
  float shadeFactor = textureProj(ShadowMap, ShadowCoord);
  gl_FragColor = vec4(shadeFactor*Color.rgb, Color.a)
}
```

Shadow Mapping

- **Pass 1:** Render the scene from the light's position. The depth buffer then contains, for each pixel, the distance between the light and the nearest object to it.
- Copy the depth buffer to a separate "shadow buffer".
- **Pass 2:** Render the scene normally. For each pixel:
  - look up the corresponding position in the shadow buffer
  - If the distance to the point being rendered is greater than the value retrieved from the shadow buffer
    - the object being drawn at this pixel is further from the light than the object nearest the light
    - therefore, this pixel is in shadow
    - make the pixel darker
  - else draw it normally

Shadow Mapping Using Textures

- Pass 1 as before
- Copy the depth buffer into a texture
- Pass 2 as before, except that the shadow buffer is now a shadow texture.

OpenGL has support for shadow textures in the form of a “sampler2DShadow” type

Pass 1

- Configure the buffer and shadow texture
- Disable color output
- Build a “look at” matrix from the light to objects in the scene
- For each object drawn, create a “shadowMVP” matrix (projection, look-at, and model matrices)
- Call glDrawArrays()
- No textures or lighting are necessary because we aren’t drawing anything in Pass 1.
- We are just creating a shadow map filled with Z-buffer values
- As a result, the fragment shader doesn’t do anything

Pass 1:

Render scene from point of view of light source

Pass one shaders:

```
#version 430                    // vertex shader
layout (location=0) in vec3 vertPos;
uniform mat4 shadowMVP;

void main(void)
{    gl_Position = shadowMVP * vec4(vertPos,1.0);
}
```

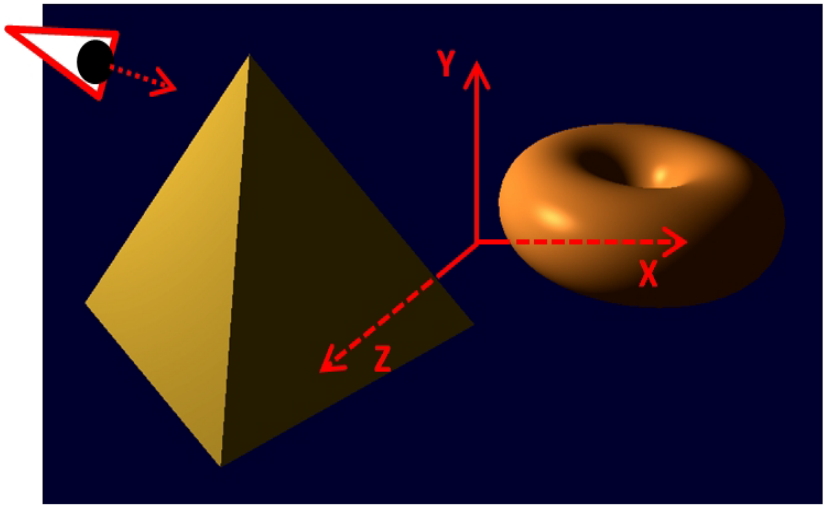
```
#version 430                    // fragment shader
void main(void)  { }
```

- No textures or lighting necessary in fragment shader
- Only Z-buffer information from vertex shader needed

Pass 1:

Can use fragment shader for optional testing

```
#version 430
out vec4 fragColor;
void main(void) {
        fragColor = vec4(1.0, 0.0, 0.0, 0.0);
}
```



Intermediate Step:
Copying the Z-buffer to a texture

One approach:

- Generate an empty shadow texture, and
- Use glCopyTexImage2D()

Another approach:

- Use a “custom framebuffer” in Pass 1
- Attach the shadow texture with glFrameBufferTexture()

Preparing for Pass 2

Convert from [-1,1] light space to [0,1] texture space:

$$B = \begin{bmatrix} 0.5 & 0 & 0 & 0.5 \\ 0 & 0.5 & 0 & 0.5 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\text{shadowMVP2} = [B]\,[\text{shadowMVP}_{(\text{pass1})}]$$

Bias matrix B translates by ½ and scales by ½ to change the light coordinates from a range of (-1..1) to a range of (0..1).

Pass 2:
Render actual scene with shadows

- Build “B” transform matrix to convert from light to texture space (typically done in init())
- Enable the shadow texture for look-up
- Enable color output
- Enable the GLSL pass 2 rendering program, containing both vertex and fragment shaders
- Build MVP matrix for the object being drawn based on the camera position (as normal)
- Build shadowMVP2 matrix (incorporating the “B” matrix) – the shaders will need it to look up pixel coordinates in the shadow texture
- Send the matrix transforms to shader uniform variables
- Enable buffers containing vertices, normal vectors, and texture coordinates (if used), as usual
- Call glDrawArrays()

Pass 2:
Other shader tasks

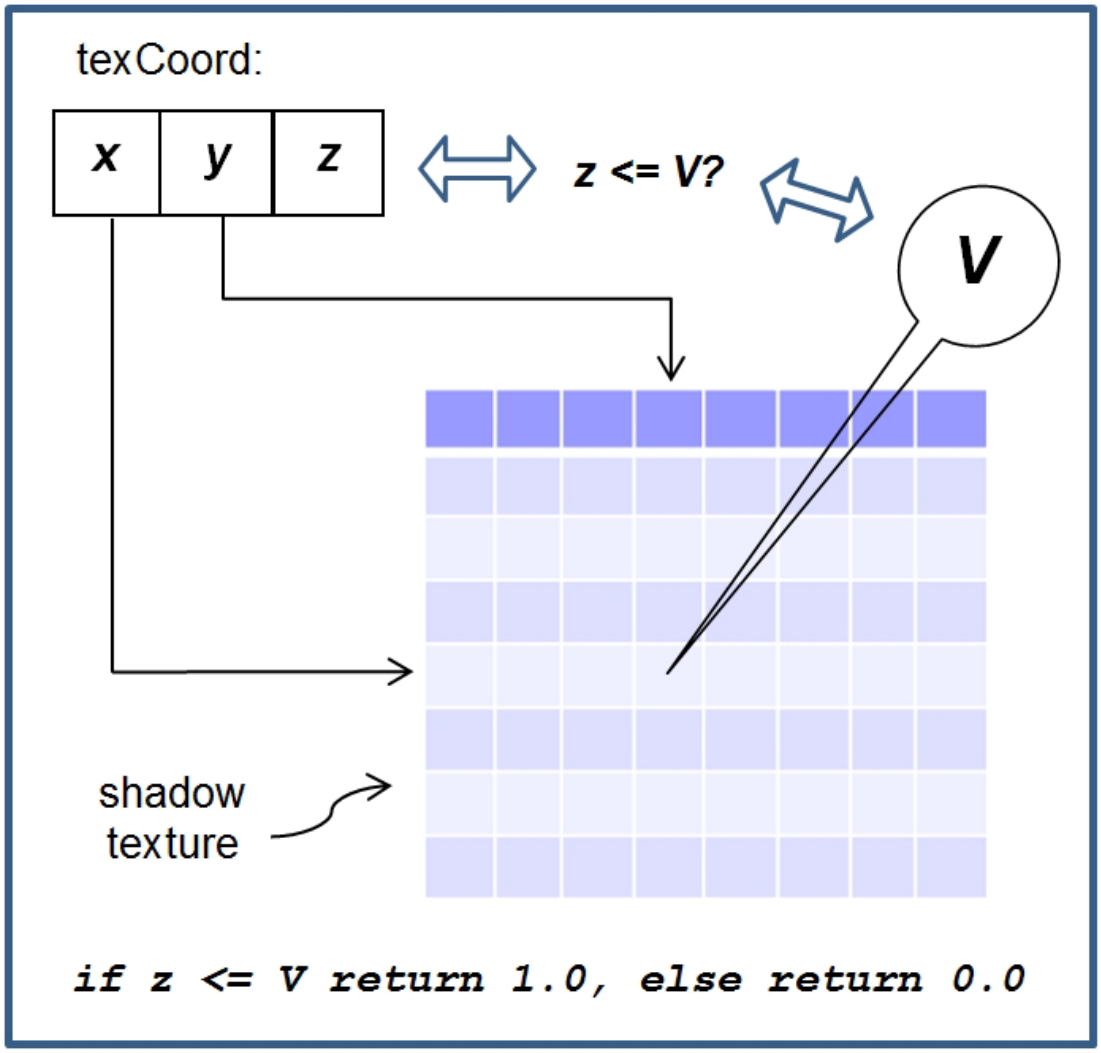
- The vertex shader converts vertex positions from model space to projected coordinates from the light's point of view, and sends the resulting coordinates to the fragment shader in a vertex attribute so that they will be interpolated. This makes it possible to retrieve the correct values from the shadow texture.

- The fragment shader calls textureProj(), which returns a 0 or 1 indicating whether or not the pixel is in shadow (described on the next slide). If it is in shadow, the shader darkens the pixel by not including its diffuse and specular contributions.

Shadow-mapping is such a common task that GLSL provides a special type of sampler variable called a sampler2DShadow . . .

Pass 2:

Automatic depth comparison in sampler2DShadow



Vertex Shader

```
. . .
out vec4 shadow_coord;
. . .
uniform mat4 shadowMVP2;
void main(void)
{   . . .
    shadow_coord = shadowMVP2 * vec4(vertPos,1.0);
}
```

Fragment Shader

```
. . .
in vec4 shadow_coord;
layout (binding=0) uniform sampler2DShadow shTex;
. . .
void main(void)
{   . . .
    float notInShadow = textureProj(shTex, shadow_coord);
    fragColor = globalAmbient * material.ambient +  light.ambient * material.ambient;
    If (notInShadow == 1.0)
    {     fragColor += light.diffuse * material.diffuse * max(dot(L,N),0.0)
            + light.specular * material.specular
            * pow(max(dot(H,N),0.0),material.shininess*3.0);
    }
}
```

C++/OpenGL application

```
. . .
// shadow-related variables
nt screenSizeX, screenSizeY;
GLUint shadowTex, shadowBuffer;
glm::mat4 lightVmatrix;
glm::mat4 lightPmatrix;
glm::mat4 shadowMVP1;
glm::mat4 shadowMVP2;
glm::mat4 b;

void init(GLFWwindow* window)  {
    . . .
    setupShadowBuffers();
    // create "B" matrix
    b = glm::mat4(
            0.5f, 0.0f, 0.0f, 0.0f,
            0.0f, 0.5f, 0.0f, 0.0f,
            0.0f, 0.0f, 0.5f, 0.0f,
            0.5f, 0.5f, 0.5f, 1.0f );
}
```

continued . . .

```
void setupShadowBuffers(GLFWwindow* window)  {
    glfwGetFramebufferSize(window, &width, &height);
    screenSizeX = myCanvas.getWidth();
     screenSizeY = myCanvas.getHeight();
```

```
    // create the custom frame buffer
    glGenFramebuffers(1, &shadowBuffer);
    //    create the shadow texture and configure it to hold depth information.
    //    these steps are similar to those in Program 5.2
    glGenTextures(1, &shadowTex);
    glBindTexture(GL_TEXTURE_2D, shadowTex);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
                                    screenSizeX, screenSizeY, 0,
                                    GL_DEPTH_COMPONENT, GL_FLOAT, 0);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_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_LEQUAL);
}
```

continued . . .

```
void display(GLFWwindow* window, double currentTime)  {
    . . .
    // vector from light to origin
    lightVmatrix = glm::lookAt(currentLightPos, origin, up);
    lightPmatrix = glm::perspective(toRadians(60.0f), aspect, 0.1f, 1000.0f);
    //  make the custom frame buffer current, and associate it with the shadow texture
    glBindFramebuffer(GL_FRAMEBUFFER, shadowBuffer);
    gl.glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                                                    shadowTex, 0);
    //  disable drawing colors, but enable the depth computation
    glDrawBuffer(GL_NONE);
    glEnable(GL_DEPTH_TEST);
    passOne();
    //  restore the default display buffer, and re-enable drawing
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glActiveTexture(GL_TEXTURE0);
    glBindTexture(GL_TEXTURE_2D, shadowTex);
    glDrawBuffer(GL_FRONT);  // re-enables drawing colors
    passTwo();
}
```

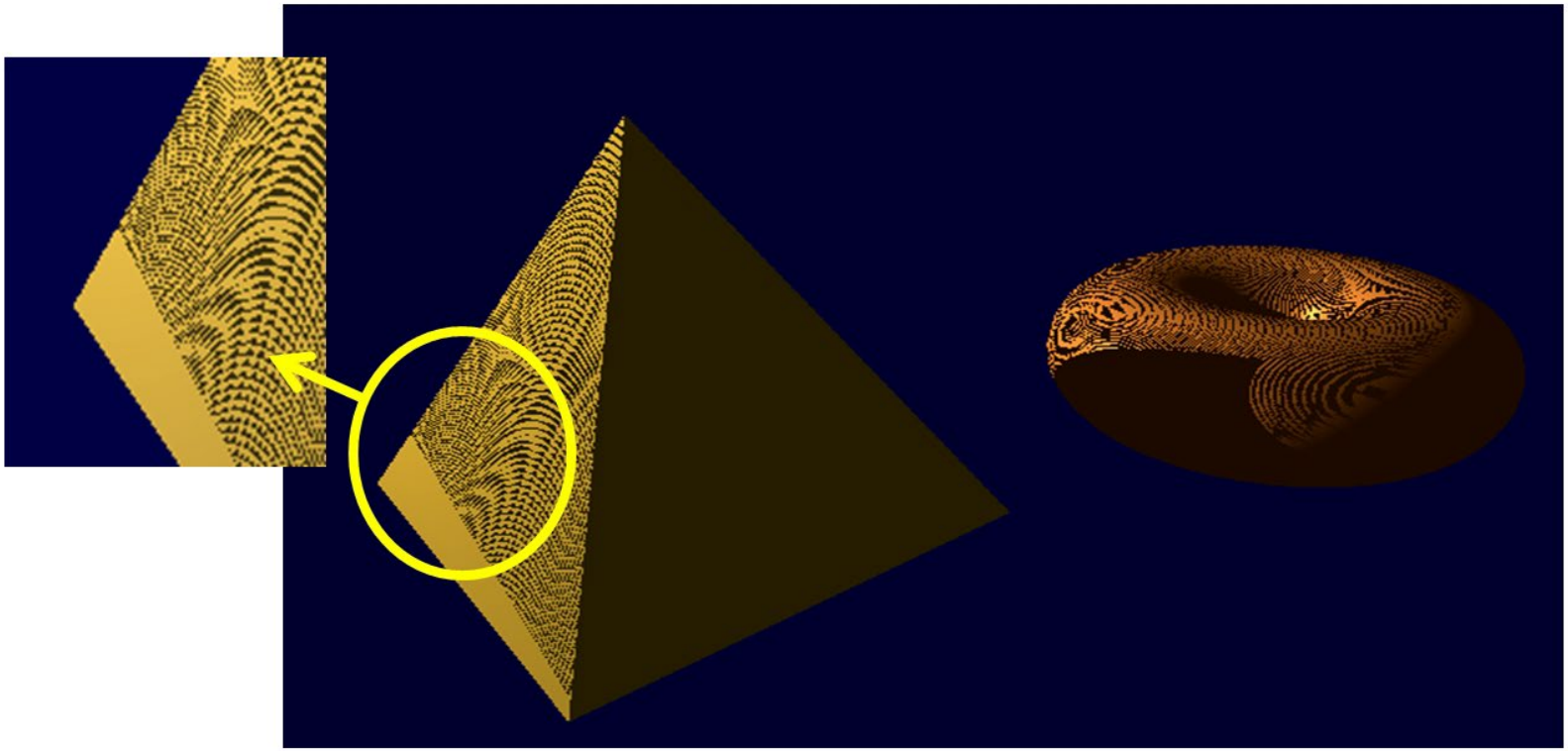
continued . . .

```
void passOne(void)  {
    // rendering_program1 contains only the pass one vertex shader
    glUseProgram(renderingProgram1);

    // build the torus object's Model matrix
    mMat = glm::translate(glm::mat4(1.0f), torusLoc);

    // we are drawing from the light's point of view, so we use the light's P and V matrices
    shadowMVP1 = lightPmatrix * lightVmatrix * mMat;
    sLoc = glGetUniformLocation(renderingProgram1, "shadowMVP");
    glUniformMatrix4fv(sLoc, 1, GL_FALSE, glm::value_ptr(shadowMVP1));
    . . .
    glDrawElements(…)

    //  repeat for the pyramid (but don't clear the GL_DEPTH_BUFFER_BIT)
}
```

continued . . .

```
void passTwo(void)  {
    . . .
    glUseProgram(renderingProgram2);
    . . .
    sLoc = glGetUniformLocation(renderingProgram2, "shadowMVP2");
    . . .
    shadowMVP2 = b * lightPmatrix * lightVmatrix * mMat;
    . . .
    glUniformMatrix4fv(sLoc, 1, GL_FALSE, glm::value_ptr(shadowMVP2));
    . . .
    glDrawElements( . . . )

    // repeat for each object drawn in the scene
    . . .
}
```

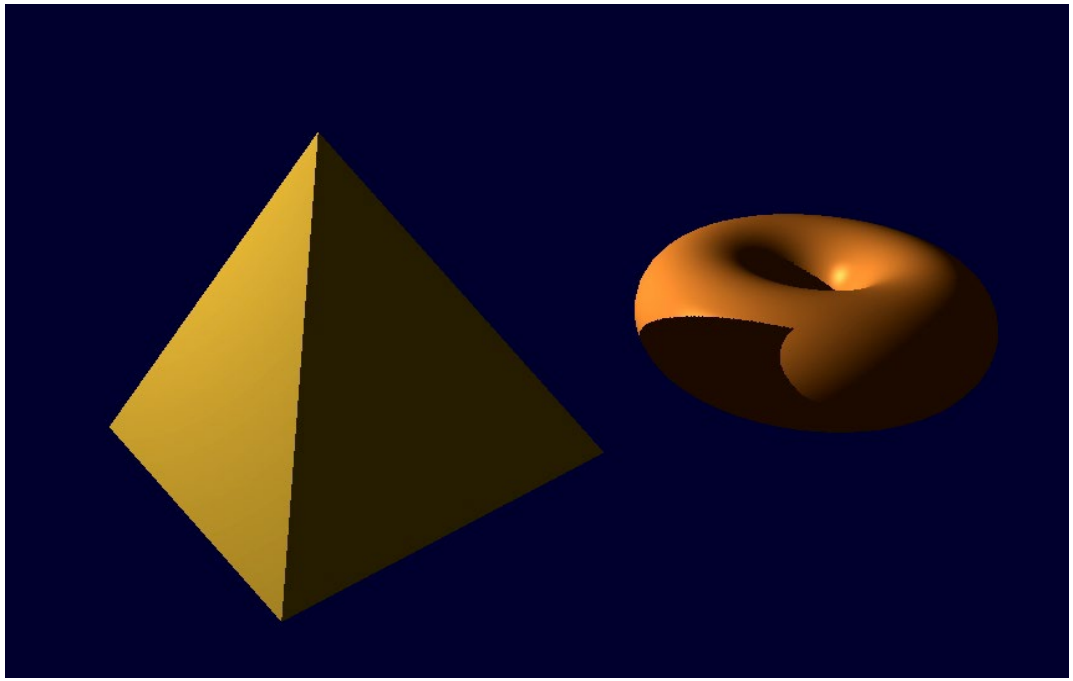
Result



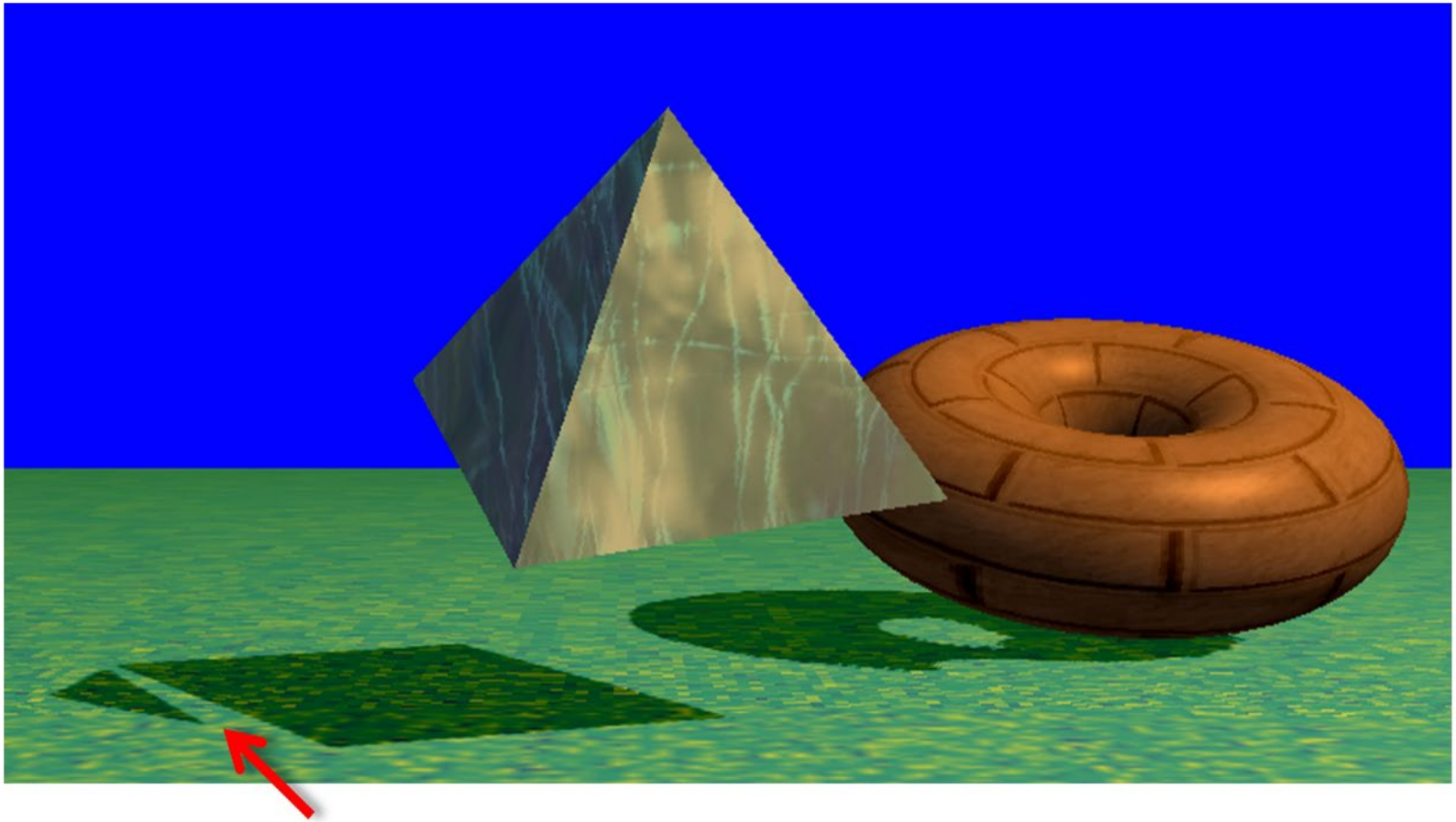
These artifacts are called “shadow acne” or “erroneous self-shadowing”

Combatting Shadow Acne

```
void display(GLFWwindow* window double currentTime)  {
    . . .
    glEnable(GL_POLYGON_OFFSET_FILL);
    glPolygonOffset(2.0f, 4.0f);
    passOne();
    glDisable(GL_POLYGON_OFFSET_FILL);
    . . .
    passTwo();
}
```

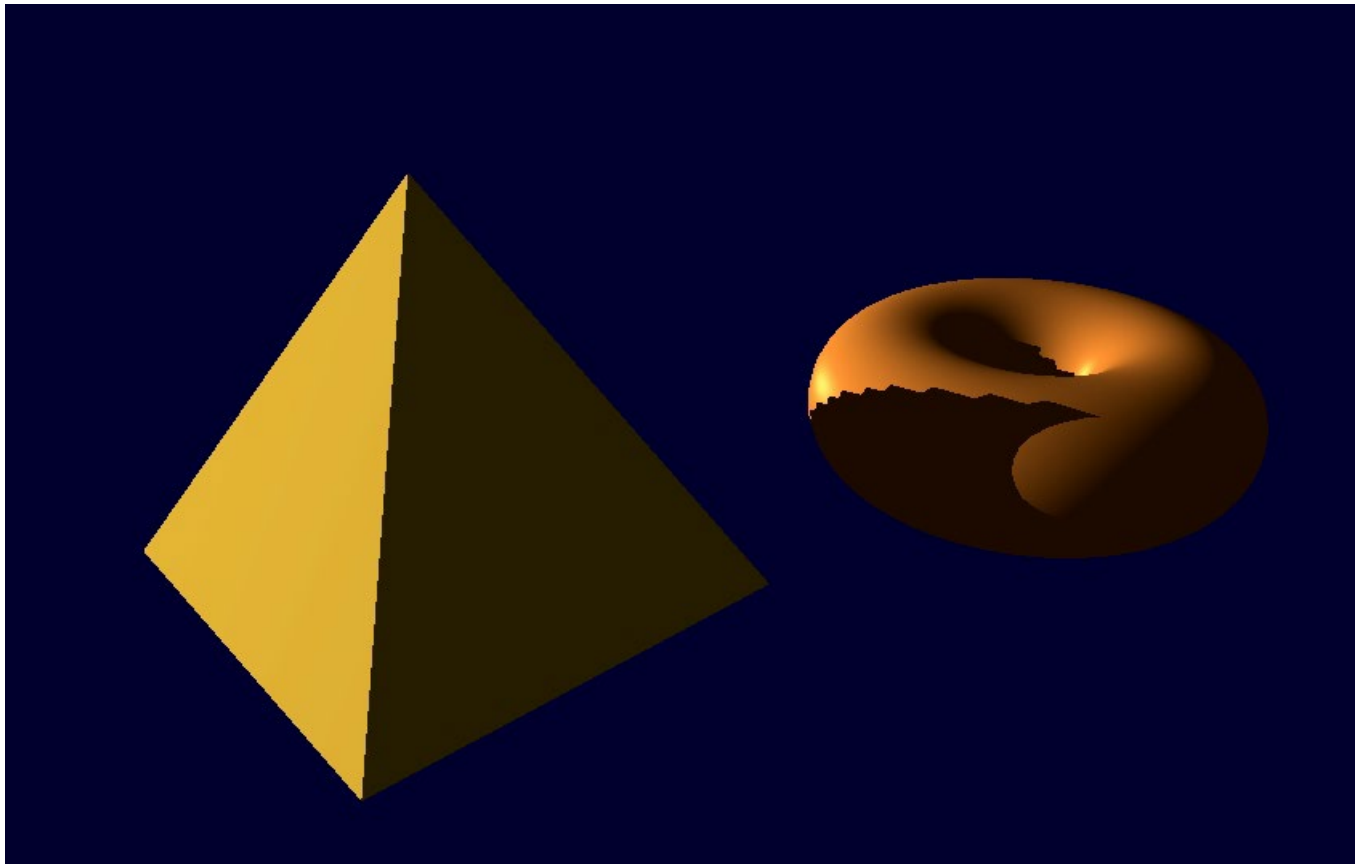


Other Shadow Mapping Artifacts



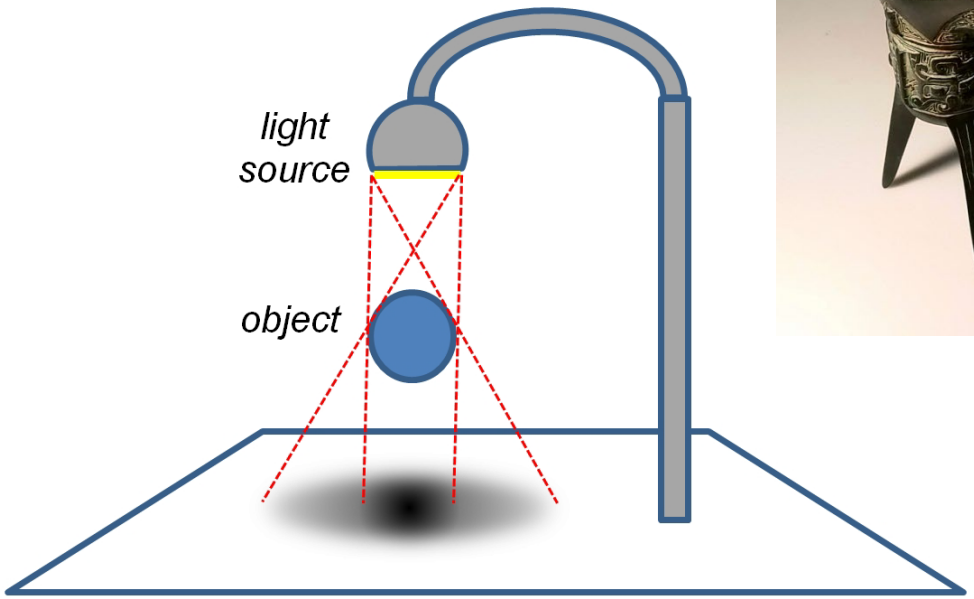
"Peter Panning"

Other Shadow Mapping Artifacts



jagged edges due to inadequate resolution

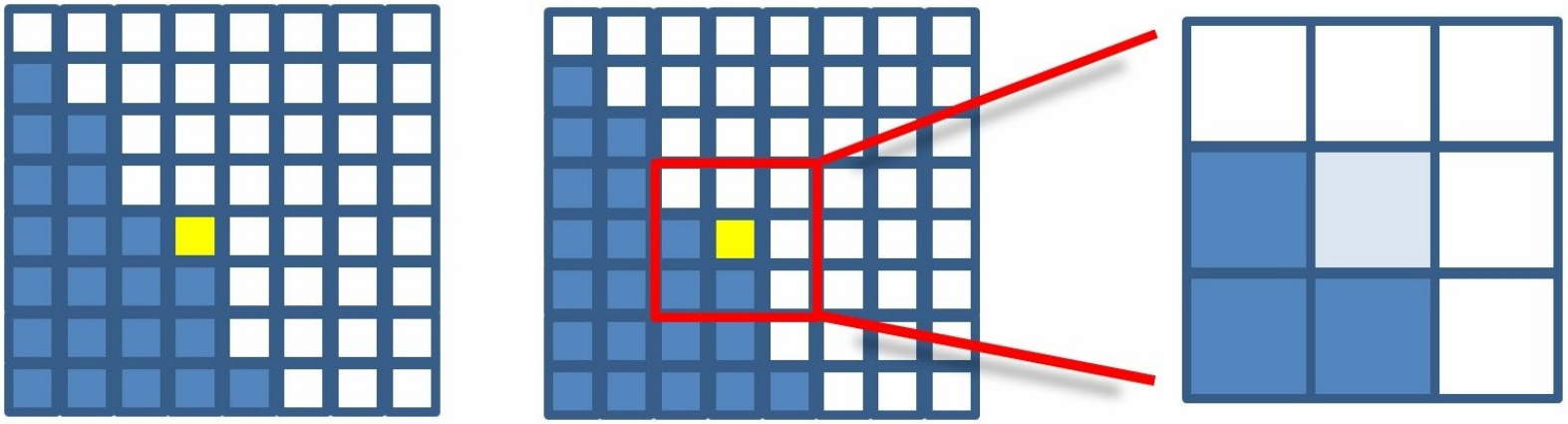
Soft Shadows



Shadows that occur in nature are usually “soft shadows”

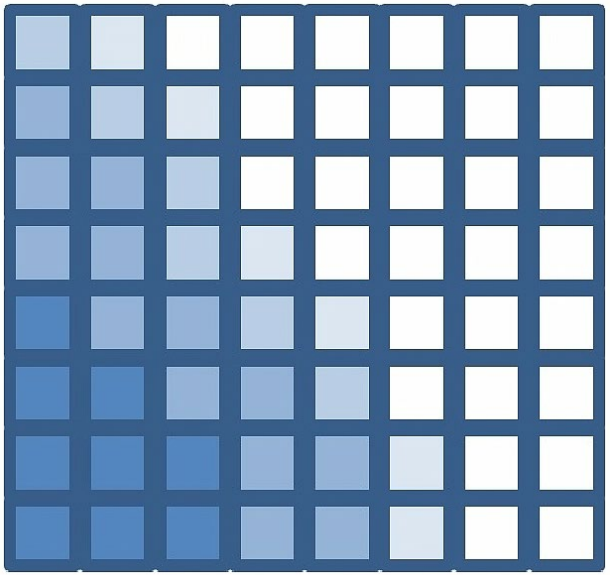
Percentage Closer Filtering (PCF)

Generate soft shadows:

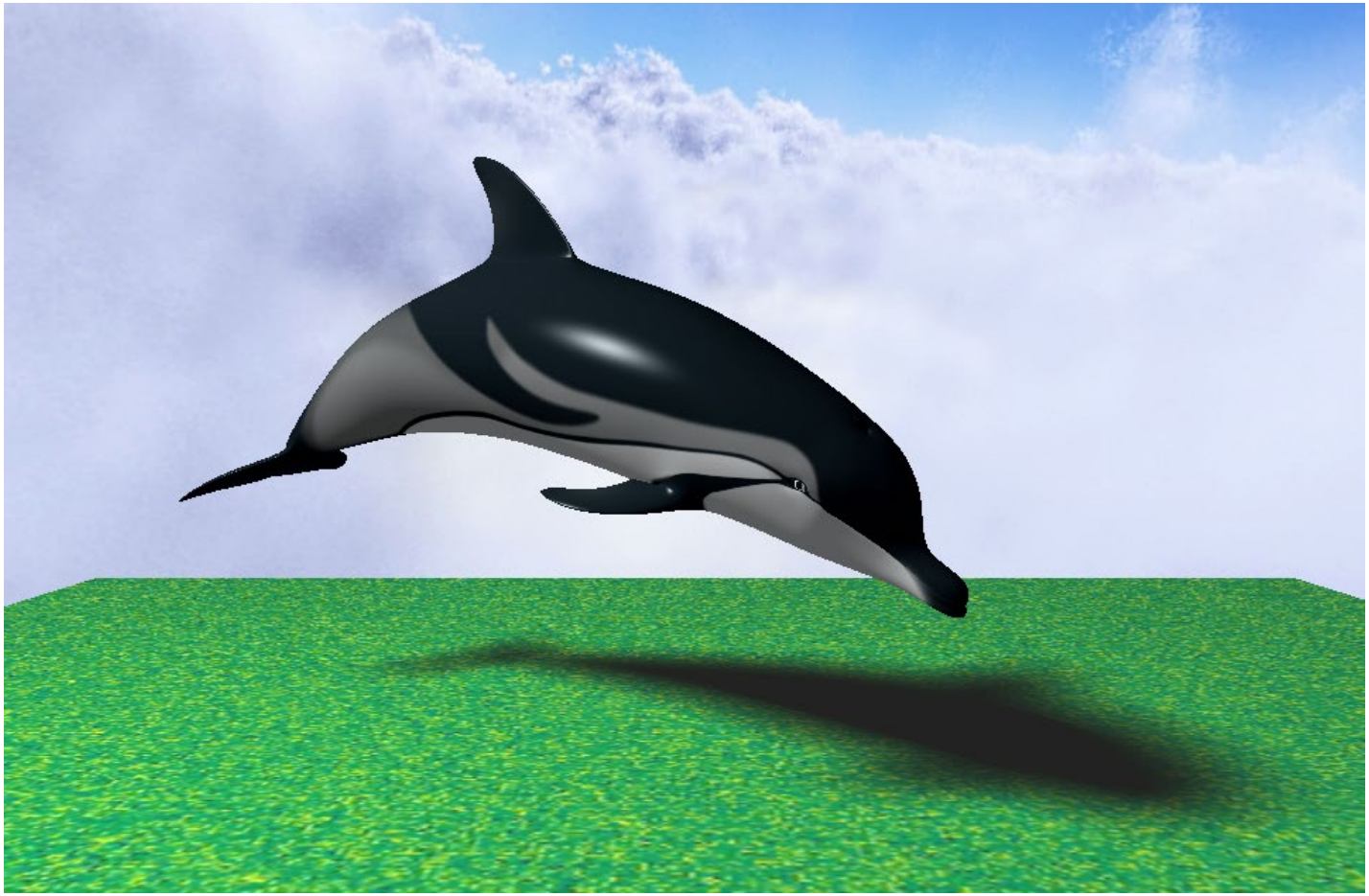


Concept:

Lighten or darken pixels based on how many neighboring pixels are in shadow



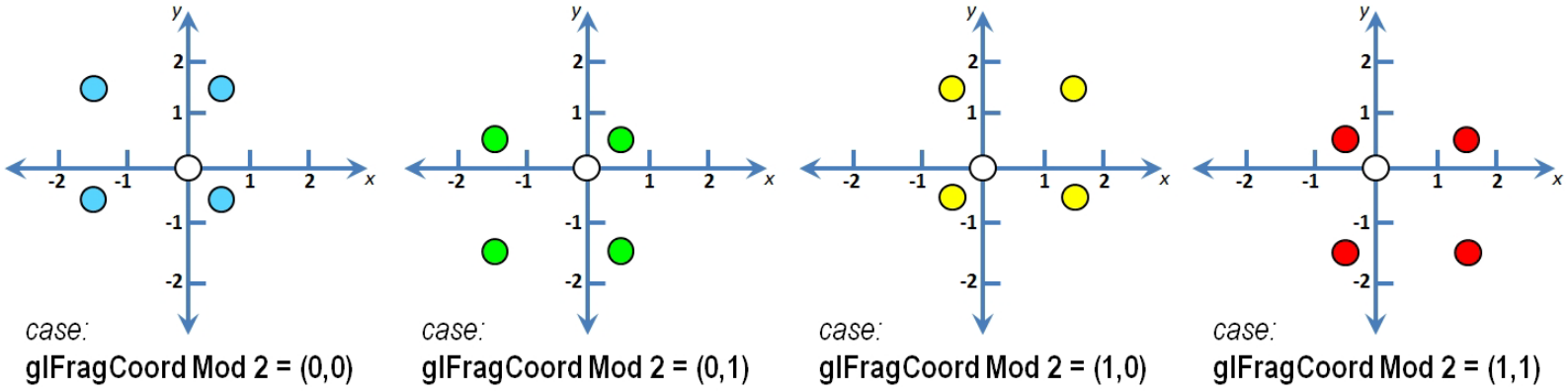
Percentage Closer Filtering Result



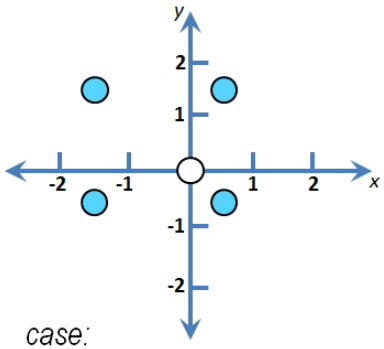
Soft shadow generated by sampling 64 neighboring pixels for each pixel being rendered

Dithering

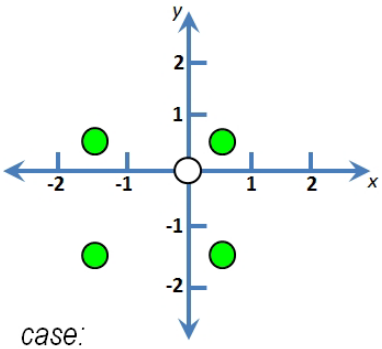
- Unfortunately, sampling this many neighbor pixels is generally not feasible for performance reasons
- A common compromise is called dithering
- Only a small number of neighbors are sampled
- The selection of neighboring pixels alternates depending on the location of the rendered pixel



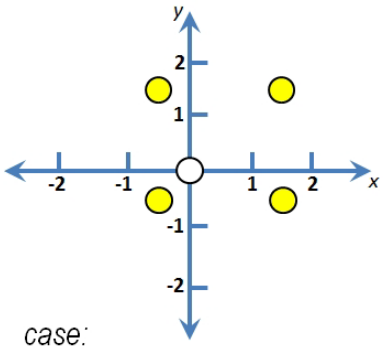
PCF with Dithering



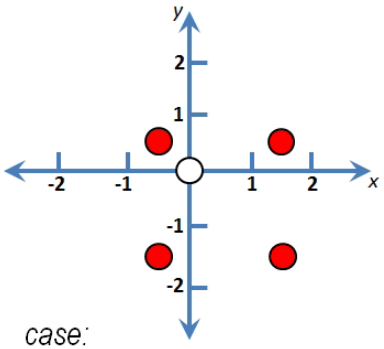
case:
glFragCoord Mod 2 = (0,0)



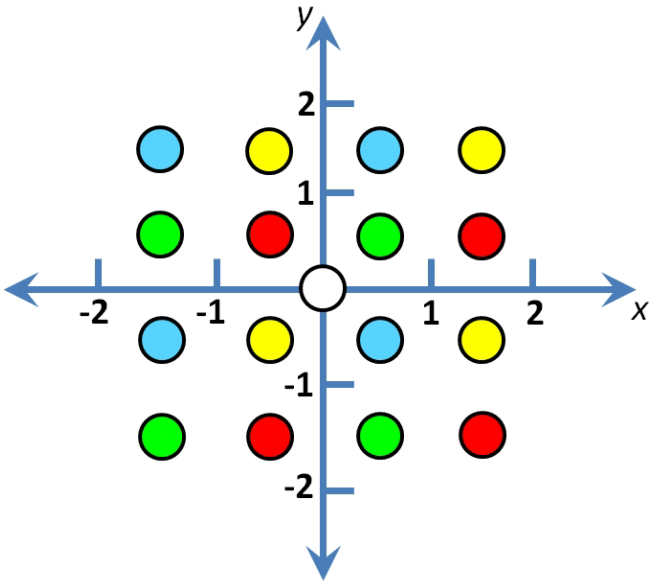
case:
glFragCoord Mod 2 = (0,1)



case:
glFragCoord Mod 2 = (1,0)

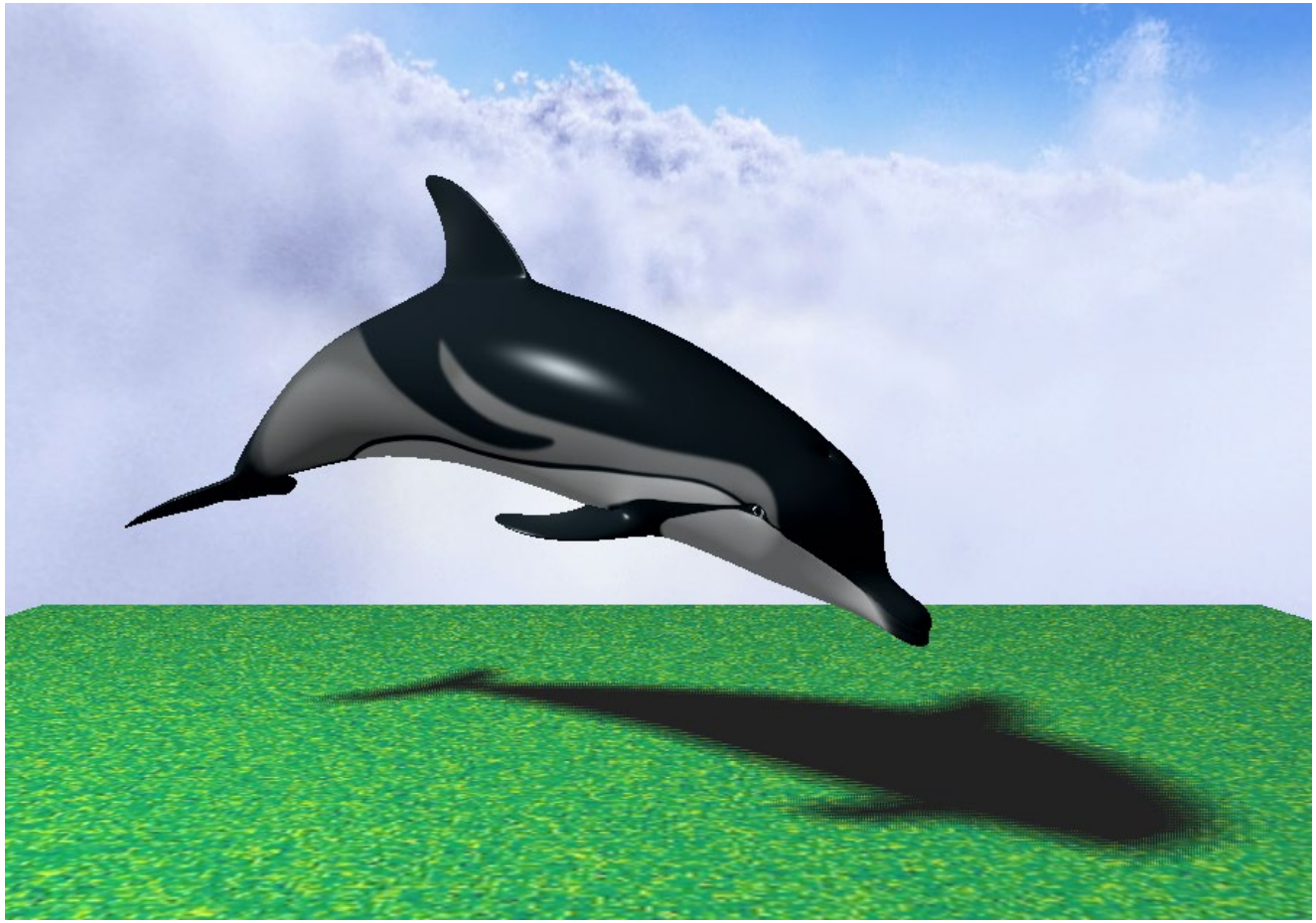


case:
glFragCoord Mod 2 = (1,1)



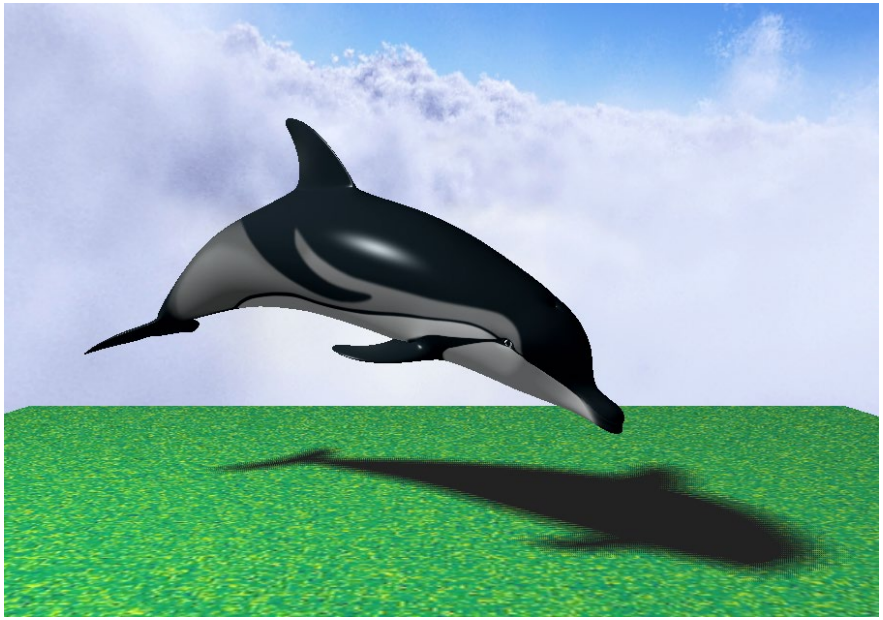
Dithering with only four neighbors

Percentage Closer Filtering Result

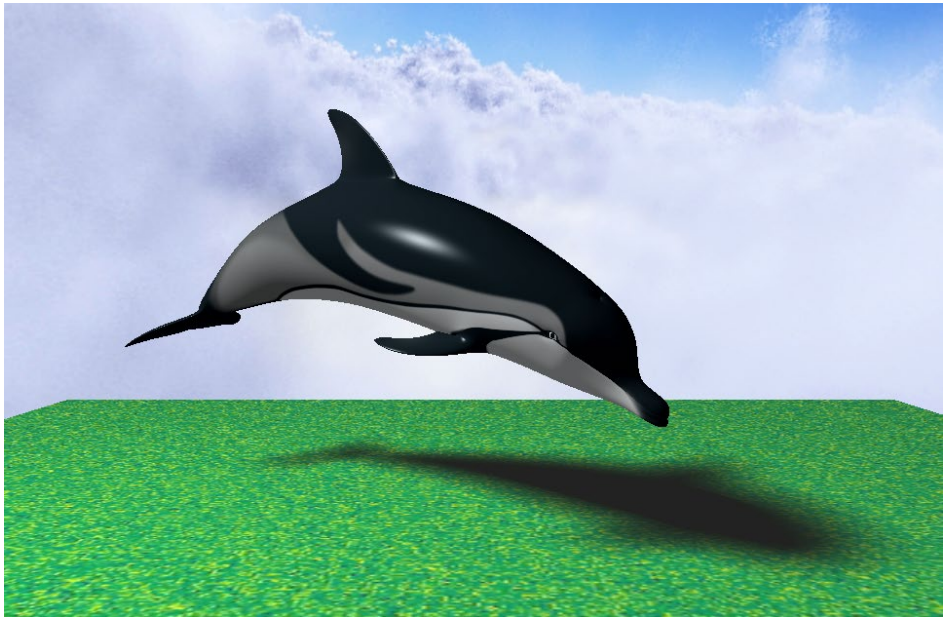


Soft shadow generated by sampling 4 neighboring pixels for each pixel being rendered (dithered)

Comparison

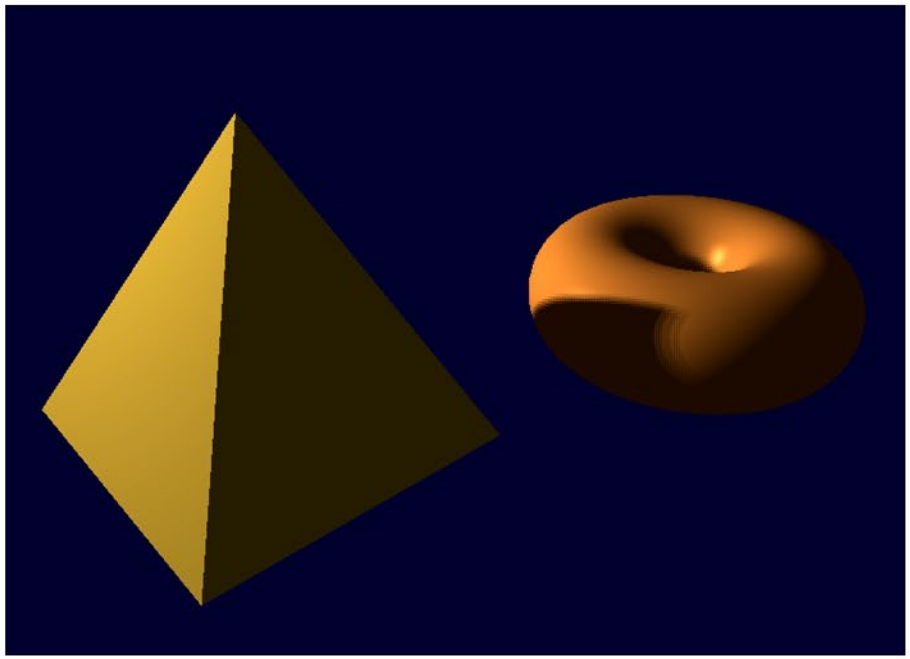


4 samples per pixel, dithered

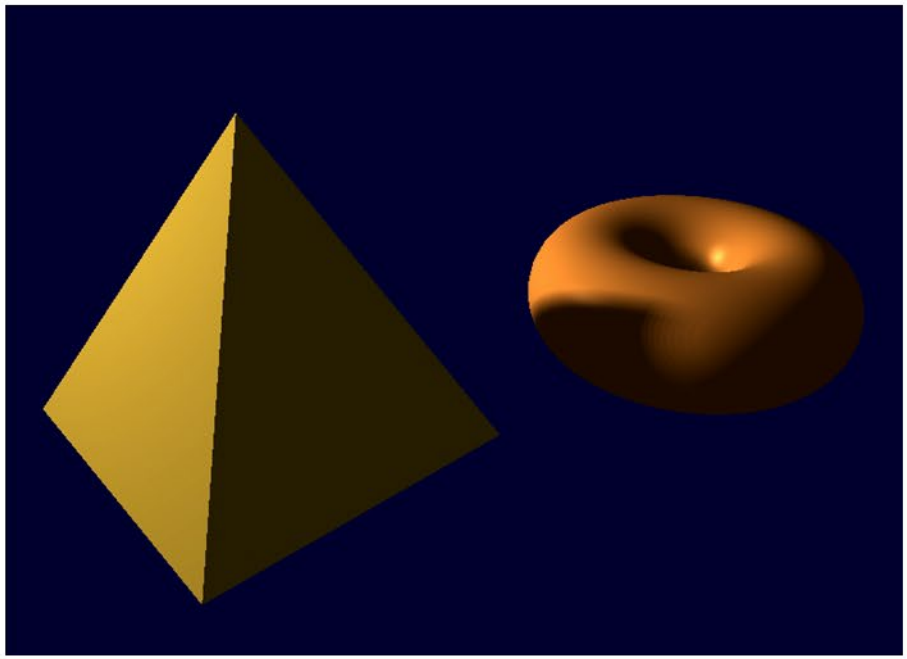


64 samples per pixel

Comparison



4 samples per pixel, dithered



64 samples per pixel

Fragment shader:

```
// Returns the shadow depth value for a texel at distance (x,y) from shadow_coord.
float lookup(float ox, float oy)
{    float t = textureProj(shadowTex,
         shadow_coord + vec4(ox * 0.001 * shadow_coord.w, oy * 0.001 * shadow_coord.w,
         -0.01, 0.0));      // the third parameter (-0.01) is an offset to counteract shadow acne
     return t;
}
void main(void)
{    . . .
     float shadowFactor = 0.0f;
     . . .
     // this section produces a 4-sample dithered soft shadow
     float swidth = 2.5;  // tunable amount of shadow spread
     // produces one of 4 sample patterns depending on glFragCoord mod 2
     vec2 offset = mod(floor(gl_FragCoord.xy), 2.0) * swidth;
     shadowFactor += lookup(-1.5*swidth + offset.x,  1.5*swidth - offset.y);
     shadowFactor += lookup(-1.5*swidth + offset.x, -0.5*swidth - offset.y);
     shadowFactor += lookup( 0.5*swidth + offset.x,  1.5*swidth - offset.y);
     shadowFactor += lookup( 0.5*swidth + offset.x, -0.5*swidth - offset.y);
     shadowFactor = shadowFactor / 4.0;        // average of the four sampled points
     . . .
     fragColor = vec4((shadowColor.xyz + shadowFactor*(lightedColor.xyz)),1.0);
}
```