CENG 477
Introduction to Computer Graphics
Textures and Framebuffers
Texture Mapping

- **Goal:** Increase visual realism by using **images** to simulate reflectance characteristics of objects.
- A cheap and effective way to **spatially vary** surface reflectance.

The ideas we learned during ray tracing apply here as well!
Texture Mapping

- **Step 1:** Associate an \((u, v)\) coordinate system with the texture image where \((u, v) \in [0,1] \times [0,1]\)
Texture Mapping

• **Step 2:** Parameterize the surface to be texture mapped using two coordinates:

$$v_0 = (0, 0) \quad v_1 = (1, 0) \quad v_2 = (0, 1) \quad v_3 = (1, 1)$$
Texture Mapping

• **Step 3:** Compute a \((u, v)\) value for every surface point
  For a triangle, this can be computed using barycentric interpolation (rasterizer does it for us):

\[
\begin{align*}
    u(\beta, \gamma) &= u_a + \beta(u_b - u_a) + \gamma(u_c - u_a) \\
    v(\beta, \gamma) &= v_a + \beta(v_b - v_a) + \gamma(v_c - v_a)
\end{align*}
\]

• **Step 4:** Find the texture image coordinate \((i, j)\) at the given \((u, v)\) coordinate:

\[
\begin{align*}
    i &= u.n_x \\
    j &= v.n_y
\end{align*}
\]

Note that \(i, j\) can be fractional!

\(n_x = \) texture image width

\(n_y = \) texture image height
Texture Mapping

- **Step 5:** Choose the texel color using a suitable interpolation strategy
  - **Nearest Neighbor:** fetch texel at nearest coordinate
    \[ \text{Color}(x, y, z) = \text{fetch}(\text{round}(i, j)) \]
  - **Bilinear Interpolation:** Average four closest neighbors:
    \[
    p = \text{floor}(i) \\
    q = \text{floor}(j) \\
    dx = i - p \\
    dy = j - q \\
    \text{Color}(x, y, z) = \text{fetch}(p, q).(1 - dx).(1 - dy) + \\
    \text{fetch}(p+1, q).(dx).(1 - dy) + \\
    \text{fetch}(p, q+1).(1 - dx).(dy) + \\
    \text{fetch}(p+1, q+1).(dx).(dy)
    \]
NN vs Bilinear Interpolation
NN vs Bilinear Interpolation
Result

(0, 0) (1, 0) (0, 1) (1, 1)
Texture Mapping using OpenGL

Step 1: Generate a name for your texture and sampler:

```c
GLuint mySampler, myTexture;
glGenSamplers(1, &mySampler);
glGenTextures(1, &myTexture);
```

these are just handles to refer to your texture and sampler
Texture Mapping using OpenGL

• Step 2: Set your sampling parameters:

```c
glSamplerParameteri(mySampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glSamplerParameteri(mySampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glSamplerParameteri(mySampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glSamplerParameteri(mySampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

- What to do if your texture needs to be minified
- What to do if your texture needs to be magnified
- What to do if you make out-of-bounds access
Texture Mapping using OpenGL

• **Step 3:** Bind your sampler to the desired texture unit:

```c
// Bind mySampler to unit 0 so that texture fetches from unit 0
// will be done according to the above sampling properties

glBindSampler(0, mySampler);
```

• **Step 4:** Activate the desired unit and bind your texture to the proper target of that unit as well

```c
glActiveTexture(GL_TEXTURE0);
gerBindTexture(GL_TEXTURE_2D, myTexture);
```
Texture Mapping using OpenGL

- **Step 5:** Read the texture image from an image file (.jpg, .png, etc.) into a one dimensional array and tell OpenGL about the address of this array:

```cpp
// When reading a texture image, do not assume that it is aligned
// to any boundary larger than a single byte

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

// Upload the image to the texture

gTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
            width, height, 0, GL_RGB,
            GL_UNSIGNED_BYTE, image1D);
```

Pointer to the first byte of your image
Texture Mapping using OpenGL

- At this point we have the following picture:
Texture Mapping using OpenGL

- **Step 6:** Provide \( uv \) coordinates for each vertex
  - In immediate mode you can use: `glTexCoord2f`
  - If using vertex arrays, we must provide the texture coordinates in an array (as we did for vertex positions, colors, etc.)
  - As before, this array can be on the system memory or uploaded to GPU memory (remember VBOs)

```
glTexCoordPointer(size, type, stride, pointer)
```

- Number of coordinates per texture vertex
- Type of each coordinate
- Byte offset between consecutive texture vertices
- Pointer to texture vertex coordinate data
Texture Mapping using OpenGL

• **Step 7:** In the vertex shader, pass along these texture coordinates to the rasterizer:

```c
void main(void)
{
    gl_FrontColor = gl_Color; // vertex color defined by the programmer
    gl_TexCoord[0] = gl_MultiTexCoord0; // pass along to the rasterizer
    gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

This value comes from the vertex array whose data is provided by `glTexCoordPointer`  
This value comes from the vertex array whose data is provided by `glVertexPointer`
Texture Mapping using OpenGL

• **Step 8:** In the fragment shader, fetch from the texture image using a suitable sampling method:

```c
uniform sampler2D mySampler;

void main(void)
{
    // get the color from the texture
    gl_FragColor = texture2D(mySampler, gl_TexCoord[0].st);
}
```

This variable represents the texture unit index. If its value is zero it will fetch from texture unit 0. Its value is given such as `glUniform1i(mySamplerLoc, 0)`
Texture Mapping using OpenGL

- At this point we have the following picture:
Texture Mapping using OpenGL

- If we call `glUniform1f(mySamplerLoc, 1):

  ![Diagram showing texture mapping using OpenGL](image_url)
Texture Mapping using OpenGL

• What to do once we have the texture color? We have several options
• For instance to blend the texture color with the color of the fragment:

```c
void main(void)
{
    // get the color from the texture
    gl_FragColor = alpha * gl_Color +
                  (1 - alpha) * texture2D(mySampler, gl_TexCoord[0].st);
}
```

Interpolated color value

User-defined interpolation parameter. Can be a `uniform`. 
Sampling

• Sampling is the process of fetching the value from a texture image given its texture coordinate

• Nearest-neighbor and bilinear interpolation are two examples

• Need to tell OpenGL about the type of sampling we want

• Previously we set sampling parameters using:

```c
glSamplerParameteri(mySampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glSamplerParameteri(mySampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glSamplerParameteri(mySampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glSamplerParameteri(mySampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

• There is another important concept called mipmapping
Mipmapping

- Mipmapping deals with cases when the resolution of the primitive is different from the resolution of the texture (which often is the case)
- Consider three cases where
  - The polygon that is texture mapped is the same size (in screen space) as the texture image
  - The polygon that is texture mapped is larger than the texture image
  - The polygon that is texture mapped is smaller than the texture image
Mipmapping

- Polygon same size as texture (map as usual):
Mipmapping

- Polygon larger (texture needs to be magnified):
Mipmapping

- Polygon smaller (texture needs to be **minified**):
Mipmapping

- **Minification**: A change of 1 pixel in image space causes a change of >1 pixel in texture space.

- To avoid **artifacts**, one should use the **average** of all texels that should fall on the same image pixel.
Mipmapping

- **Take the extreme case:** 1 pixel change in image space corresponds to as many pixels as the width of the texture in texture space:

- For accurate mapping, this requires computing the average value of the entire row – otherwise *aliasing* artifacts will occur
Aliasing
Improved Result
Fixing Aliasing

- Aliasing artifacts are even more disturbing if animation is present in the scene.
- Aliasing artifacts occur as we are sampling a high frequency texture at very low frequencies.
- Our sample does not faithfully represent the real signal:
  - It adopts a different persona – thus called aliasing.
- Sampling at a higher rate is not an option as samples are determined by our fragments.
- **Solution:** Reduce the frequency of the original signal by low-pass filtering (blurring).
- **Problem:** Expensive to continuously filter in runtime.
Mipmapping

- **Solution:** Pre-filter images to create smaller resolution versions during initialization (or offline):

- Then sample from the appropriate resolution in runtime

- Memory requirement – how much memory does a mipmap chain require?

\[
A + A/4 + A/16 + A/64 + \ldots = 4A/3
\]
OpenGL Support

- Mipmap levels can be created offline and then given to OpenGL. This allows custom filtering for each level:

```c
for (int level = 0; level < numLevels; ++level) {
    glTexImage2D(GL_TEXTURE_2D, level, GL_RGB,
                 width, height, 0,
                 GL_RGB, GL_UNSIGNED_BYTE, image[level]);
}
```
OpenGL Support

• Alternatively, we can ask OpenGL to automatically generate mipmap levels for us:

```c
GLenum err;

err = glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
                   width, height, 0,
                   GL_RGB, GL_UNSIGNED_BYTE, image1D);

err = glGenerateMipmap(GL_TEXTURE_2D);
```

• To use mipmapping, we must set the sampler parameters correctly:

```c
//glSamplerParameteri(mySampler, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glSamplerParameteri(mySampler, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glSamplerParameteri(mySampler, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glSamplerParameteri(mySampler, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glSamplerParameteri(mySampler, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```
Framebuffer Objects

- Until now, we always rendered to the screen
- But many visual effects require rendering an image to an off-screen buffer and processing it before displaying it
Framebuffer Objects

- Framebuffer objects are designed to allow such effects
- **Step 1:** To use an FBO you must first generate a name for it and bind it as the current framebuffer

```c
GLuint gFBOId;
glGenFramebuffers(1, &gFBOId);
glBindFramebuffer(GL_FRAMEBUFFER, gFBOId);
```
Framebuffer Objects

**Step 2:** Next we must allocate memory for its color and (optionally) depth buffers. These memories are allocated as textures

- For color buffer:

  ```c
  glGenTextures(1, &gColorTextureId);
  glBindTexture(GL_TEXTURE_2D, gColorTextureId);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_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);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, gFBOWidth, gFBOHeight, 0,
                GL_RGB, GL_UNSIGNED_BYTE, 0);
  ```
Framebuffer Objects

- **Step 2:** Next we must allocate memory for its color and (optionally) depth buffers. These memories are allocated as textures

- For depth buffer:

```c
glGenTextures(1, &gDepthTextureId);
glBindTexture(GL_TEXTURE_2D, gDepthTextureId);
glTexParameter(i(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST));
glTexParameter(i(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST));
glTexParameter(i(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE));
glTexParameter(i(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE));
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, gFBOWidth, gFBOHeight, 0,
             GL_DEPTH_COMPONENT, GL_FLOAT, 0);
```
Framebuffer Objects

• **Step 3:** We must attach these textures to the FBO:

```c
glfwFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                         GL_TEXTURE_2D, gColorTextureId, 0);
glfwFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                         GL_TEXTURE_2D, gDepthTextureId, 0);
```

• **Step 4:** Make sure that FBO is complete:

```c
GLenum status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
assert(status == GL_FRAMEBUFFER_COMPLETE);
```
Framebuffer Objects

• When we render while this FBO is bound, the attached textures’ contents will be updated

• Important: before rendering make sure that you set your viewport to match the resolution of this framebuffer using `glViewport(0, 0, gFBOWidth, gFBOHeight)`

• This is needed as the size of the window (for which the viewport was originally set) can be different from the size of our FBO
Framebuffer Objects

• Once you make the FBO rendering pass, you can *detach* your textures and switch back to the *default framebuffer*:

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

  glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, 0, 0);

  glBindFramebuffer(GL_FRAMEBUFFER, 0);
  ```

• Now you can use these textures as *source textures* for various special effects

• One such usage is for generating *shadows* as we will learn next week
CENG 477
Introduction to Computer Graphics
Shadows in Forward Rendering
Shadows

- Shadows are an important element of visual realism

From Sintorn et al. – Siggraph Asia 2012
Shadows

• Shadows give important cues about light positions

From wikipedia.com
Shadows

- Shadows also give cues about object positions
Shadows in OpenGL

- OpenGL does not have built-in support for shadows
Shadows in OpenGL

• Compare this to a ray traced image:
Shadows in OpenGL

- OpenGL **does not** natively support generating shadows (neither does D3D)
  - That is, it does not have a function like glMakeShadow()!
- But, several shadowing algorithms can easily be implemented using features of OpenGL (or D3D)
  - Stencil buffer
  - Depth buffer
  - Render to texture
  - ...
Generating Shadows

• Two algorithms that are commonly used are:
  – Shadow volumes (Crow, 1977)
  – Shadow mapping (Williams, 1978)

• Both algorithm has advantages and disadvantages and many variants

• Still an active research area:

• We’ll study the basic versions of these algorithms
Shadow Volumes

• The idea is to create a 3D shape that represents the shadow that is casted by an object.
Shadow Volumes

• A shadow volume can be created for any arbitrary object.
• We need to determine the contour edges (silhouette-edges) of the object as seen from the light source
• A contour edge has one adjacent polygon facing the light source and the other away from the light source
Contour Edges

- A **contour edge** has one adjacent polygon facing the light source and the other away from the light source.
- We can use dot product to decide whether a face is toward or away from the light source.
Contour Edges

// transform the light to the coordinate system of the object
LightPosition = Inverse(ObjectWorldMatrix) * LightPosition;
for (every polygon) {
    IncidentLightDir = AveragePolyPosition - LightPosition;
    // if the polygon faces away from the light source....
    if (DotProduct(IncidentLightDir, PolygonNormal) >= 0.0) {
        for (every edge of the polygon) {
            if (the edge is already in the contour edge list) {
                // then it can't be a contour edge since it is
                // referenced by two triangles that are facing
                // away from the light
                remove the existing edge from the contour list;
            } else {
                add the edge to the contour list;
            }
        }
    }
}
Extruding Contour Edges

- Once the contours are found, we need to extrude them to create a large shadow volume
Extruding Contour Edges

- Extrusion amount should be large enough to cover all objects which can receive shadow from this light source

```cpp
ExtrudeMagnitude = A_BIG_NUMBER;
for (every edge) {
    ShadowQuad[i].vertex[0] = edge[i].vertex[0];
    ShadowQuad[i].vertex[1] = edge[i].vertex[1];
    ShadowQuad[i].vertex[3] = edge[i].vertex[0] + ExtrudeMagnitude * (edge[i].vertex[0] - LightPosition);
}
```
Extruding Contour Edges

- The shadow caster object’s contour vertices serve as the *cap* of the shadow volume.
- The *bottom* can also be capped to obtain a closed volume.
Extruding Contour Edges

• This is how it looks like for a complex object
Rendering Shadows

- Now what? Any ideas about how we can proceed?
Rendering Shadows

- Assume a ray originating from the eye
- If it enters the shadow volume from a front face and exits from a back face, the point it reaches is not in shadow
- However, if it does not exit before hitting the object, the point should be in shadow
Rendering Shadows

• But we are not ray tracing! How can we know if the ray enters or exits?
• This is where **depth** and **stencil** buffers come in handy
• Stencil buffer:
  – An integer buffer that stores a value for every pixel (usually an 8-bit value)
  – We can clear it to any value that we want (glClearStencil(int) and glClear(GL_STENCIL_BUFFER_BIT))
  – Has operations such as **increment** and **decrement** based on the result of the depth test (glStencilOp(sfail, dfail, dpass))
• Think of stencil buffer as a **counter buffer**, which keeps a counter for every pixel
Shadow Volume Algorithm (Part I)

- **Clear** the depth (to 1.0) and stencil buffer (to 0)
- **Enable depth testing** and **disable stencil testing**
- Render the scene with **ambient light** (note that ambient light does not produce shadows). This updates the depth buffer value for every pixel that corresponds to an object
- **Disable writing** to the depth buffer

<table>
<thead>
<tr>
<th>Color Buffer</th>
<th>Depth Buffer</th>
<th>Stencil Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 100 100 100 100 0 0</td>
<td>1.0 1.0 1.0 0.5 0.5 0.5 1.0 1.0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 100 100 100 100 0 0</td>
<td>1.0 1.0 1.0 0.5 0.5 0.5 1.0 1.0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 100 100 100 100 0 0</td>
<td>1.0 1.0 1.0 0.5 0.5 0.5 1.0 1.0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 100 100 100 100 0 0</td>
<td>1.0 1.0 1.0 0.5 0.5 0.5 1.0 1.0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 100 100 100 100 0 0</td>
<td>1.0 1.0 1.0 0.5 0.5 0.5 1.0 1.0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 100 100 100 100 0 0</td>
<td>1.0 1.0 1.0 1.0 1.0 1.0 1.0 1.0</td>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
Shadow Volume Algorithm (Part II)

• Draw the **front faces** of the shadow volume. Increment the stencil value for every pixel that passes the depth test (`glStencilOp(GL_KEEP, GL_KEEP, GL_INCR)`)

![Stencil Buffer – Front Pass](image_url)
Shadow Volume Algorithm (Part II)

• Draw the **back faces** of the shadow volume. Decrement the stencil value for every pixel that passes the depth test (`glStencilOp(GL_KEEP, GL_KEEP, GL_DECR)`)
Shadow Volume Algorithm (Part III)

- Enable stencil testing such that pixels whose stencil value is zero will be rendered
- Enable writing to the depth buffer and clear it
- Enable the point light source
- Enable color buffer blending so the contribution of passing pixels will be added to the previous ambient values
Shadow Volume Algorithm

- No blending versus blending:
Used Applications

• Doom 3 is the most well-known example.

Pros and Cons

• Requires **preprocessing** of the scene geometry to create the shadow volume
• Needs to be updated if lights and/or objects move
• Can be time consuming for complex geometry
• Requires 4 rendering passes for each frame
  – Ambient pass, SV front-face pass, SV back-face pass, final light source pass.
• No need to update shadow volume if only the camera moves
Pros and Cons

- Inaccurate models can cause leaking artifacts

Pros and Cons

• For speed-up, two versions of objects can be created:
  – High polygon version is used for actual rendering of the object
  – Low polygon version is used to cast shadows of the object

From wikipedia.com
Questions

• How can the shadow volume algorithm be extended to deal with:
  – Camera inside the shadow volume?
  – Multiple light sources and multiple shadow casters?
  – Transparent shadow casters?

• Further reading:
Shadow Mapping

• The idea introduced by Lance Williams in his 1978 paper “Casting Curved Shadows on Curved Surfaces”
• Image space technique
• Advantages:
  – No knowledge or processing of scene geometry is required
  – No need to use stencil buffer
  – Fewer rendering passes than shadow volumes for a single light source
• Disadvantages:
  – Need an extra rendering pass for each additional light
  – Aliasing artifacts may occur (shadows may look jaggy)
Shadow Mapping

- **Part I:** Render the scene from the point of view of the light source (as if the light source was a camera)
  - Objects that are not visible are in shadow with respect to that light source
- **Part II:** Determine whether an object as seen from the camera is in shadow in the “light’s view”
Part I: Rendering from the Light Source

- Pretend that there is a camera at the light position
- Use **perspective** projection for spot (and point) lights
- Use **orthographic** projection for directional lights
Part I: Extracting the Depth Map

- Actually, we only need the depth buffer values from the light source view
- Save this depth buffer to an off-screen texture (FBOs)

Depth map from the light’s view

- $1 - z$ is shown for visualization purposes
- Normally, $z$ is larger for further away points
Part II: Rendering from the Camera

• Render the scene from the camera view as usual
• For every pixel, compare the depth value of that pixel w.r.t. the light source ($R$) to the stored value in the depth texture ($D$):

  - $R = D$: Object was directly visible from the light source
  - $R > D$: Object was behind another object in the light’s view

• But there is a problem:
  - Pixel $(i, j)$ in the camera view will not belong to the same object as pixel $(i, j)$ in the light view as they look at the scene from different positions (and with different orientations)
Projective Texturing

- For every pixel in the camera view, we need to find the corresponding pixel in the light’s view.

From http://www.riemers.net/images/Tutorials/DirectX/Csharp/Series3
Projective Texturing

• Assume \texttt{inPos} represents the world coordinates of an object. Its \textbf{camera coordinates} are computed by:

\[
\text{outPosCamera} = \text{cameraWorldViewProjection} \times \text{inPos};
\]

• Its \textbf{light view coordinates} are computed by:

\[
\text{outPosLight} = \text{lightWorldViewProjection} \times \text{inPos};
\]

• We can use \texttt{outPosLight} to look up the depth texture we generated in Part I

• But this value is in range \([-1,1]\) in all axis (CVV)
Projective Texturing - Bias

- Therefore, we need to multiply it with a bias matrix to bring all components to [0, 1] range:

\[
\begin{bmatrix}
0.5 & 0 & 0 & 0.5 \\
0 & 0.5 & 0 & 0.5 \\
0 & 0 & 0.5 & 0.5 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

\[\text{outPosLight} = \text{bias} \times \text{lightWorldViewProjection} \times \text{inPos};\]
Shadow Check

- Remember that $R$ is the depth value of the pixel from the light’s view and $D$ is the stored depth in the texture:
  - $R = D$: Object was directly visible from the light source
  - $R > D$: Object was behind another object in the light’s view

- $R$ and $D$ can be found by:
  - $R = \text{outputPosLight.z}$
  - $D = \text{texLookUp(\text{shadowMap}, \text{outputPosLight.xy})}$

- Now we can perform the shadow check:
  ```
  if (R > (D + 0.00001))
      Output.Color = Input.Color * 0.5;
  else
      Output.Color = Input.Color;
  ```
  Note that a small bias is added to avoid self-shadowing just like as in ray tracing.
Shadow Mapping

• More to read from:
  – Projective texturing:
    http://en.wikipedia.org/wiki/Projective_texture_mapping
  – OpenGL fixed function implementation:
    • http://www.paulsprojects.net/tutorials/smt/smt.html
  – OpenGL shader (GLSL) implementation:
    • www.gamedev.net/community/forums/topic.asp?topic_id=316147
    • http://sombermoon.com/shadowmappingdoc.html
Importance of Shadow Map Resolution

- If the depth texture we create in Part I does not have enough resolution, we can see blocking artifacts:

160x120 shadow map

1280x960 shadow map

From http://bytechunk.net
Projection Artifacts

• If an object falls outside the viewing frustum of the light, we can see artifacts with a naïve implementation:

• For how to fix these, read more at:
  – http://bytechunk.net/shadowmapping/index.php
Improving the Shadow Quality

• Shadow map can be filtered in various ways to create soft shadows:

From http://www.gamedev.net
Applications Using Shadow Maps

• Most games use shadow mapping

Rage
Applications Using Shadow Maps

• Riddick 2: Assault on Dark Athena

From http://uk.ps3.ign.com
Applications Using Shadow Maps

- Dragon Age
Applications Using Shadow Maps

• Assassin’s Creed