Shadow Algorithms
Why Shadows?

- Makes 3D Graphics more believable
- Provides additional cues for the shapes and relative positions of objects in 3D
What is shadow?

- Shadow: comparative darkness given by shelter from direct light; patch of shade projected by a body intercepting light
**Terminology**

- **umbra** – fully shadowed region
- **penumbra** – partially shadowed region
“Hard” and “Soft” Shadows

- Depends on the type of light sources
  - Point or Directional ("Hard Shadows", umbra)
  - Area ("Soft Shadows", umbra, penumbra), more difficult problem
“Hard” and “Soft” Shadows

- Hard shadow
  - *point* light source
- Soft shadow
  - *area* light source
Simple Approach: Ray tracing

- Cast ray to light (shadow rays)
- Surface point in shadow if the shadow rays hits an occluder object.
- Ray tracing is slow, can we do better? (perhaps at the cost of quality)
Shadow Algorithms

- We will first focus on hard shadows
  - Planar Shadows
  - Shadow Maps
  - Shadow Volume
The simplest algorithm – shadowing occurs when objects cast shadows on planar surfaces (projection shadows).

\[ \mathbf{n} \cdot \mathbf{x} + d = 0 \]
Planar Shadows

- Special case: the shadow receiver is an axis plane
  - Just project all the polygon vertices to that plane and form shadow polygons

Given:
- Light position \( l \)
- Plane position \( y = 0 \)
- Vertex position \( v \)

Calculate: \( p \)
Planar Shadows

- We can use similar triangles to solve $p$

$$\frac{p_x-l_x}{v_x-l_x} = \frac{l_y}{l_y-v_y}$$

$$p_x = \frac{l_y v_x - l_x v_y}{l_y - v_y}$$

- Same principle applied to different axis planes
Planar Shadows

- How about arbitrary plane as the shadow receiver?

Plane equation: \( n \cdot x + d = 0 \) or

\[ ax + by + cz + d = 0 \]

where \( n = (a,b,c) \) - plane normal

Again, given light position \( l, v \)

Find \( p \)
Planar Shadows

- Finding \( \mathbf{p} \)

We know: \( \mathbf{p} = \mathbf{l} + (\mathbf{v} - \mathbf{l}) \times \alpha \)

Also we know: \( \mathbf{n} \cdot \mathbf{p} + d = 0 \)

Because \( \mathbf{p} \) is on the plane

We can solve \( \alpha \) easily:

\[
\mathbf{p} = \mathbf{l} - \frac{d + \mathbf{n} \cdot \mathbf{l}}{\mathbf{n} \cdot (\mathbf{v} - \mathbf{l})} (\mathbf{v} - \mathbf{l})
\]
Issues about planar shadows

- Shadow polygon generation (z fighting)
  - Add an offset to the shadow polygons (glPolygonOffset)
  - Draw receivers first, turn z-test off, then draw the shadow polygons. After this, draw the rest of the scene.

- Shadow polygons fall outside the receiver
  - Using stencil buffer – draw receiver and update the stencil buffer

- Shadows have to be rendered at each frame
  - Render into texture

- Restrictive to planar objects
Issues about planar shadows

- Anti-shadows and false shadows

![Correct shadow](image1)

- Anti-shadow

- False shadow
Shadow Volumes

- A more general approach for receivers that have arbitrary shapes
What is shadow volume?

- A volume of space formed by an occluder
- Bounded by the edges of the occluder
- Any object inside the shadow volume is in shadow
2D Cutaway of a Shadow Volume

- Light source
- Shadowing object
- Partially shadowed object
- Surface outside shadow volume (illuminated)
- Shadow volume (infinite extent)
- Surface inside shadow volume (shadowed)

Eye position:

(note that shadows are independent of the eye position)
In Shadow or not?

- Use shadow volume to perform such a test
- How do we know an object is inside the shadow volume?
  1. Allocate a counter
  2. Cast a ray into the scene
  3. Increment the counter when the ray enter a front-facing polygon of the shadow volume (enter the shadow volume)
  4. Decrement the counter when the ray crosses a back-facing polygon of the shadow volume (leave the shadow volume)
  5. When we hit the object, check the counter.
     - If counter >0; in shadow
     - Otherwise - not in shadow
Counter for Shadow Volume

Light source

Shadowing object

zero
+1

Eye position

+1
+2
+2

+3

In shadow
Real time shadow volume

- How can we render the idea of shadow volume in real time?
  - Use OpenGL Stencil buffer as the counter
- Stencil buffer?
  - Similar to color or depth buffer, except it’s meaning is controlled by application (and not visible)
  - Part of OpenGL fragment operation – after alpha test before depth test
  - Control whether a fragment is discarded or not
    - **Stencil function (Stencil test)** - used to decide whether to discard a fragment
    - **Stencil operation** – decide how the stencil buffer is updated as the result of the test
Stencil Function

- Comparison test between reference and stencil value
  - GL.Never always fails
  - GL.Always always passes
  - GL.Less passes if reference value is less than stencil buffer
  - GL.LEQUAL passes if reference value is less than or equal to stencil buffer
  - GL.Equal passes if reference value is equal to stencil buffer
  - GL.GEQUAL passes if reference value is greater than or equal to stencil buffer
  - GL.Greater passes if reference value is greater than stencil buffer
  - GL.NOTEQUAL passes if reference value is not equal to stencil buffer

- If the stencil test fails, the fragment is discarded and the stencil operations associated with the stencil test failing is applied to the stencil value.

- If the stencil test passes, the depth test is applied
  - If the depth test passes, the fragment continue through the graphics pipeline, and the stencil operation for stencil and depth test passing is applied
  - If the depth test fails, the stencil operation for stencil passing but depth failing is applied
Stencil Operation

- **Stencil Operation:** Results of Operation on Stencil Values
  - GL_KEEP stencil value unchanged
  - GL_ZERO stencil value set to zero
  - GL_REPLACE stencil value replaced by stencil reference value
  - GL_INCR stencil value incremented
  - GL_DECR stencil value decremented
  - GL_INVERT stencil value bitwise inverted

- **Remember you can set different operations for**
  - Stencil fails
  - Stencil passes, depth fails
  - Stencil passes, depth passes
OpenGL for shadow volumes

- Z-pass approach
- Z-fail approach
- Ideas used by both of the algorithms are similar
  - Z-pass: see whether the number of visible front-facing shadow volume polygons and the number of visible back-facing polygons are equal. If yes – objects are not in shadow
  - Z-fail: see whether the number of invisible back-facing shadow volume polygons and the number of invisible front-facing polygons are equal. If yes – objects are not in shadow
Z-pass approach

- Render visible scene with only ambient and emission and update depth buffer
- Turn off depth and color write, turn on stencil (but still keep the depth test on)
- Init. stencil buffer
- Draw shadow volume twice using face culling
  - 1st pass: render *front* faces and *increment* stencil buffer when depth test passes
  - 2nd pass: render *back* faces and *decrement* when depth test passes
- stencil pixels != 0 in shadow, = 0 are lit
- Render the scene again with diffuse and specular when stencil pixels = 0
Problems of Z-pass algorithm

1. When the eye is in the shadow volume
   - Counter= 0 does not imply out of shadow anymore
   - In this case, the stencil buffer should be init. with the number of shadow volumes in which the eye is in (instead 0)

2. When the near plane intersects with the shadow volume faces (and thus will clip the faces)
Z-pass algorithm problem

- illustration

Mistakenly determined as not in shadow
Z-fail approach

- Render visible scene to depth buffer
- Turn off depth and color, turn on stencil
- Init. stencil buffer given viewpoint
- Draw shadow volume twice using face culling
  - 1st pass: render \textit{back} faces and \textit{increment} when depth test fails
  - 2nd pass: render \textit{front} faces and \textit{decrement} when depth test fails
- stencil pixels $\neq 0$ in shadow, $= 0$ are lit
Problem of z-fail algorithm

- Shadow volume can penetrate the far plane

Solution: depth clamping - close up the shadow volume
Shadow Map

- Basic idea: objects that are not visible to the light are in shadow
- How to determine whether an object are visible to the eye?
  - Use z-buffer algorithm, but now the “eye” is light, i.e., the scene is rendered from light’s point of view
  - This particular z-buffer for the eye is called *shadow map*
Shadow Map Algorithm

- illustration
Shadow Map Algorithm

1. Render the scene using the light as the camera and perform z-buffering.
2. Generate a light z buffer (called shadow map).
3. Render the scene using the regular camera, perform z-buffering, and run the following steps: (next slide)
Shadow Map Algorithm (cont’d)

3.1 For each visible fragment with \([x,y,z]\) in local space, perform a transformation to the light’s clip space (light as the eye) \([x_1,y_1,z_1]\)

3.2 Compare \(z_1\) with \(z = \text{shadow\_map}[x_1,y_1]\)
   - If \(z_1 \leq z\) (closer to light), then the pixel in question is not in shadow; otherwise the fragment is shadowed
1st Pass

View from light

Depth Buffer (shadow map)
2nd Pass

Visible surface depth
2nd Pass

Non-green in shadow

Final Image
Shadow Maps With OpenGL/GLSL

- Render scene using the light as a camera
- Render the depth buffer into a 2D texture.
  - We now have a depth texture.
- Render the scene using the real camera
  - Pass the transformation matrices as before to your shader (modelview, modelview projection, etc)
  - Pass one more matrix: ShadowMatrix
    - ShadowMatrix is to transform the vertices from local space to light’s clip space
    - Multiply your vertex positions by the ShadowMatrix to get the texture coordinates into the shadow map
Render depth into a texture

```c
void Init_FBO() {

    glGenFramebuffers(1, &shadowFBO);
    glBindFramebuffer(GL_FRAMEBUFFER, shadowFBO);

    GLfloat border[] = {1.0f, 0.0f, 0.0f, 0.0f};
    GLuint depthTex;

    glGenTextures(1, &depthTex);
    glActiveTexture(GL_TEXTURE0);  glBindTexture(GL_TEXTURE_2D, depthTex);

    glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT24,
                 512, 512, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_NEAREST);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_NEAREST);

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

    // check FBO status
    GLenum FBOstatus =
        glCheckFramebufferStatusEXT(GL_FRAMEBUFFER);
    if(FBOstatus != GL_FRAMEBUFFER_COMPLETE)
        printf("GL_FRAMEBUFFER_COMPLETE failed,
               CANNOT use FBO
");

    glBindFramebuffer(GL_FRAMEBUFFER, 0);  // go back to the default framebuffer
}
```
Shadow Matrix

- Shadow Matrix = B * P_L * V_L * M
- \( P_L \): Light’s projection matrix
- \( V_L \): Light’s view matrix
- \( M \): Object’s model matrix
- \( B \): change the range from \([-1,1]\) to \([0,1]\)

\[
B = \begin{bmatrix}
0.5 & 0 & 0 & 0.5 \\
0 & 0.5 & 0 & 0.5 \\
0 & 0 & 0.5 & 0.5 \\
0 & 0 & 0 & 1.0
\end{bmatrix}
\]
Shadow Map Look up

- Multiply the shadow matrix to the vertex’s local coordinate position -> [s,t,r,w]
- Pass the result to the fragment shader
- In the fragment shader
  - Perform perspective division [s,t,r,w]/w
  - Use (s/w, t/w) to look up the shadow map
  - Compare r/w with the value stores in the shadow map
  - If r/w is closer (smaller), fragment is not in shadow
  - Otherwise it is in shadow
Generate texture coordinates to look up shadow map

- Goal: compare the fragment’s distance to the light to the value stored in the corresponding position in the shadow map
- Major steps:
  - Map the fragment’s position to the light’s projection (clip) space
  - Generate texture coordinates (s,t,r) to look up the shadow map
  - Compare the fragment’s depth value in the light’s clip space to the value stored in the shadow map
  - The fragment is not in shadow if the depth value is less than or equal to the value stored in the shadow map

```glsl
uniform sampler2DShadow ShadowMap;
vec4 texC = ShadowCoord;
float shadow = shadow2DProj(ShadowMap, texC).x;
gl_FragColor = v_color*shadow;
```
Shadow map issues

- Shadow quality depends on
  - Shadow map resolution – aliasing problem
  - Z resolution – the shadow map is often stored in one channel of texture, which used to be only 8 bits, but now most of hardware supports 24 bits
  - Self-shadow aliasing – caused by different sample positions in the shadow map and the screen
Shadow map aliasing problem

- The shadow looks blocky – when one single shadow map pixel covers several screen pixels
- This is a problem similar to texture magnification
Percentage Closer Filtering

- Could average binary results of all depth map pixels covered
- Soft anti-aliased shadows