Visible Surface Determination

Given a 3D scene (set of polygons)

\( p \) Viewpoint

\( F \) View frustum

Compute the portion of \( \Sigma \) visible from \( p \) within \( F \).

**Perspective view:** \( p \) is at finite distance.

**Orthographic view:** \( p \) is at infinity.

Only viewing direction is relevant.

---

Occlusion

- Viewing direction is along the \((-z)\)-axis.
- \( P_1 \) and \( P_2 \): Two points in 3D.
- \( P_1 \) **occludes** \( P_2 \) if
  - \( P_1 \) and \( P_2 \) lie on the same projector, and
  - \( P_1 \) is closer to the center of projection.

- If \( P_1 \) and \( P_2 \) are **not** on the same projector, they do not occlude each other.

---

Canonical View Volumes

View volume affects the projectors, which define occlusion.

**Perspective view volume:**

\[ P_1 = (x_1, y_1, z_1), P_2 = (x_2, y_2, z_2) \]

\( P_1, P_2 \) on the same projector if

- \( x_1 / z_1 = x_2 / z_2 \),
- \( y_1 / z_1 = y_2 / z_2 \).

**Orthographic:** \( P_1, P_2 \) on the same projector if

- \( x_1 = x_2 \),
- \( y_1 = y_2 \),
- No divisions!
**Canonical View Volumes**

Transform perspective view volume to orthographic view volume.

- **Slide 5**

**General Approaches**

- **Slide 6**

\[ \sum \text{ Set of objects.} \]
\[ \Pi \text{ Set of pixels.} \]

**Object-space approach:**

for (each object \( \sigma \in \Sigma \) {  
  Find the visible part \( \sigma' \) of \( \sigma \);  
  Draw the pixels corresponding to \( \sigma' \) in color of \( \sigma \);  
}

**Scan-Line Algorithm**

Generalize the polygon scan-conversion algorithm to handle many polygons.

- **Slide 7**

- **Slide 8**

  - Scan from bottom to top.
  - For each scan line, visit the pixels from left to right.
  - For each pixel \( \pi \) determine the polygon \( \sigma \) closest to the viewpoint.
  - Draw \( \pi \) with the color of \( \sigma \).
**Edge Coherence**

- Scan line corresponds to a horizontal plane \( h \) (parallel to the \( xz \)-plane) in 3D.
- Intersection of \( h \) and a polygon \( P \) is a line segment \( ab \).
- Endpoints of \( ab \) are intersection points of the scan line and the edges of \( P \).
- Let \( q_1, q_2, \ldots \) be the intersection points of projections of polygon edges and the scan line.
- Same polygon is visible between \( q \) and \( q_{+1} \).

**Scan-Line Algorithm**

**Data structures:**
- **Edge table:** Same as in polygon scan conversion.
- **Polygon table:** Each entry stores
  - Coefficients of the plane equation.
  - Shading and color information.
- **Active edge lists**

<table>
<thead>
<tr>
<th>Scan line</th>
<th>Entries</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \alpha )</td>
<td>( AB, AC )</td>
</tr>
<tr>
<td>( \beta )</td>
<td>( AB, AC, FD, FE )</td>
</tr>
<tr>
<td>( \gamma, \gamma + 1 )</td>
<td>( AB, DE, CB, FE )</td>
</tr>
<tr>
<td>( \delta )</td>
<td>( AB, CB, DE, FE )</td>
</tr>
</tbody>
</table>

**Z Buffer Algorithm**

- Simple to implement in hardware.
- Requires two buffers.

- **Frame buffer** \( F[x, y] \): color of pixel \((x, y)\).
- **Z buffer** \( Z[x, y] \): \( z \)-coordinate of the object visible at pixel \((x, y)\).

Initially, each element in the Z buffer stores the \( z \)-value of the far clipping plane of the view volume.

\( \Sigma \): Set of polygons.

\[
\text{for } \sigma \in \Sigma \{ \\
\quad \text{for each pixel } p = (x, y) \text{ in projection of } \sigma \{ \\
\quad \quad \hat{z}_p = z\text{-value of } \sigma \text{ at } p \\
\quad \quad \text{if } \hat{z}_p > Z[x, y] \{ \\
\quad \quad \quad Z[x, y] = \hat{z}_p; \\
\quad \quad \quad \text{writePixel } (x, y, \sigma \text{'s color at } p); \\
\quad \quad \} \\
\quad \} \\
\}
\]

**Z Buffer Algorithm**

**Visibility**
Z Buffer Algorithm

Use polygon scan conversion algorithm to process the pixels of $\sigma$.

Plane equation of $\sigma$: $Ax + By + Cz + D = 0$.

$$z = \frac{-D - Ax - By}{C}$$

Between scan lines, $y$ increments by 1, so

$$z(x, y + 1) = z(x, y) - \frac{B}{C}$$

Disadvantages

- Uses only the front-most polygon to set pixel colors.
- Cannot handle transparent objects.
- Aliasing problems: Different polygons may share the same pixel.
- Roundoff errors
  - Perspective view to orthogonal view transformation reduces $z$ precision.
  - Polygons with different depths may have the same $z$-value.
  - Static objects may swap occlusion as the camera moves.

Z Buffer in OpenGL

```c
glClear(GL_DEPTH_BUFFER_BIT);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GLenum fn);
```

- $z$-value: Distance between the object and the viewpoint.
- If $z$-value of the new fragment satisfies $fn$, its $z$-value is written in the depth buffer.
- Default is GL_LESS.

```c
glDepthMask(GLboolean mask);
```

- GL_TRUE: $z$-buffer in read/write mode.
- GL_FALSE: $z$-buffer in read only mode.

Blending

- Without blending a pixel is overwritten in the frame buffer (objects are opaque).
- Blending allows to combine the existing color of a pixel with that of incoming fragment.
- Blending allows to display transparent/translucent objects.
- A (Alpha) in RGBA mode specifies blending
`glColor4f(R, G, B, A)`
- Smaller values of $A$ denote higher transparency.

Example: `glColor4f(1.0, 0.0, 0.0, 0.2)`
$A = 0$: transparent; $A = 1$: opaque
- Red glass with 80% transparency.
- Can have Multiple transparent objects.
**Blending**

**Source:** Incoming fragment.

**Value:** \((R_s, G_s, B_s, A_s)\)

**Blending factor:** \((S, S_p, S_h, S_a)\)

**Destination:** Stored pixel

**Value:** \((R_d, G_d, B_d, A_d)\)

**Blending factor:** \((D, D_p, D_h, D_a)\)

**New value of pixel:**

\[(R, G, B, A) = (R_s + R_d S, G_s + G_d S, B_s + B_d S, A_s + A_d S)\]

- `glEnable(GL_BLEND)`
- `glBlendFunc(sfactor, dfactor)`

---

**Blending: Examples**

**Example 1:** Combining two images with equal blending factor.

- Draw the first image with
  - `sfactor = GL_ONE`
  - `dfactor = GL_ZERO`
- Draw the second image with
  - `sfactor = GL_SRC_ALPHA`
  - `dfactor = GL_ONE_MINUS_SRC_ALPHA`
  - \(A_s = 0.5\)

**Example 2:** Image through a photographic filter that blocks 20% red light, 60% green light, and 28% blue light.

- Set destination color
  - \((R_d, G_d, B_d, A_d) = (0.8, 0.4, 0.72, 1.0)\).
- Draw the image with
  - `sfactor = GL_DST_COLOR`
  - `dfactor = GL_ZERO`.

---

**Handling Transparent Objects**

\(\Sigma\): Scene (set of polygons) consisting of

- Opaque objects
- Transparent objects

- Draw all the opaque objects
  - Depth buffer in the normal write mode
- `glDepthMask(GL_FALSE)` (Read only mode)
- `glEnable(GL_BLEND)` (Turn on blending)

- Draw translucent objects with blending.
  - Translucent objects behind an opaque object do not have any effect.
  - Translucent objects in front of all opaque objects do not change the z-value.
  - Colors are blended.
Blending and Antialiasing

- Pixel $\pi$ is not a point.
- $z$-value is not the same over the entire pixel.
- Compute $z$-values at the center of $\pi$.

- Compute all polygons $\sigma_1, \dotsc, \sigma_k$ visible at $\pi$.
  $C_i$: color of $\sigma_i$.
  Assume $\sigma_i$'s sorted by their depth value.
- For each polygon, compute the area $A_i$ of $\sigma_i$ visible within $\pi$.
- Blend the colors accordingly:
  $$C(\pi) = \sum_{i=1}^{k} C_i A_i$$

Slide 21

Blending and Antialiasing

$$\Pi = \pi$$
$$\text{for } (i = 1, i \leq k, i +) \{}$$
$$\Pi_{\text{front}} = \Pi \cap \sigma_i$$
$$C = C + \frac{\text{Area}(\Pi_{\text{front}})}{\text{Area}(\Pi)} \cdot C(\sigma_i)$$
$$\Pi = \Pi \setminus \Pi_{\text{front}}$$

Slide 22

Antialiasing and Blending

- Computing $\Pi_{\text{front}}$ and $\Pi$ are expensive!
- Use a *mask* ($M$) (super sampling) for each pixel.
  *(Typical masks size: $4 \times 8$).*
- Run the z-buffer algorithm on all the subpixels.
- Blend the colors of the subpixels to compute the color of a pixel.

Slide 23

Buffers

- Frame (color) buffer
- Depth buffer
- Accumulation buffer
- Stencil buffer

Slide 24
Accumulation Buffer

- Analogous to multiple exposures.
- Sequence of images is generated.
- Images are accumulated into the accumulation buffer.
- After accumulation the result is copied back to the frame buffer for viewing.
- Accumulation buffer has higher precision.

Antialiasing

\[
\text{for } (i = 1, i \leq \mu, i++) \{
\text{offset the image by } i\text{-th subpixel position;}
\text{draw the shifted image;}
\text{accumulate the color values into the A-buffer;}
\}
\]

Divide the color values of each pixel by \(\mu\);
Draw the normalized image;

Accumulation Buffer in OpenGL

\[
g\text{clearAccum}(R, G, B, A);
g\text{clear}(GL\_ACCUM\_BUFFER\_BIT);
g\text{accum}(op, val);
\]

\textbf{Buffer: Value of the frame buffer.}

\begin{align*}
\text{GL\_LOAD} & \quad \text{Acc} = \text{Buff} * \text{val} \\
\text{GL\_ACCUM} & \quad \text{Acc} = \text{Acc} + \text{Buff} * \text{val} \\
\text{GL\_RETURN} & \quad \text{Buff} = \text{Acc} * \text{val} \\
\text{GL\_ADD} & \quad \text{Acc} = \text{Acc} + \text{val} \\
\text{GL\_MULT} & \quad \text{Acc} = \text{Acc} * \text{val}
\end{align*}

Slide 25

Visibility

CPS124, 296: COMPUTER GRAPHICS

Slide 26

Accumulation Buffer

\textbf{Antialiasing}

\[
\text{for } (i = 1, i \leq ACSIZE, i++) \{
\text{g\_clear(GL\_COLOR\_BUFFER\_BIT)},
\text{GL\_DEPTH\_BUFFER\_BIT});
\text{accPerspective} (\cdots);
\text{displayobjects} ();
\text{g\_accum}(GL\_ACCUM, 1.0/ACSIZEx);
\}
\text{g\_accum}(GL\_RETURN, 1.0);
\]

\textbf{Motion Blur}

- Offset the image by the motion of objects.
- Different objects can move at different speed.
- Entire scene can be made dimmer by \text{g\_accum(GL\_MULT, decay)}

Slide 27

Depth of field

- Objects lying only on a particular plane are in focus in a picture.
- All objects are in focus in a scene drawn by OpenGL.
- Draw the scene repeatedly with slightly different values of \text{g\_frustum} and accumulate the results.
  - Viewpoint is slightly different
  - Objects on a particular plane remain focused

\textbf{Soft shadows}

- Draw the shadows from each light source separately
- Accumulate the results.

Slide 28
Stencil Buffer

☆ Restricts drawing to a certain portion of the image plane.
   E.g.: Drawing a scene through an odd-shaped window

☆ `glStencilFunc` (func, ref, mask)
  • `func`: Comparison function to decide whether a pixel should be drawn.
    
    | GL_NEVER    | GL_ALWAYS  |
    |----------------|------------|
    | GL_LESS      | GL_LEQUAL  |
    | GL_GREATER   | GL_GEQUAL  |
    | GL_EQUAL     | GL_NEQUAL  |
  • `ref`: The value stored in stencil buffer is compared with `ref` using `func`.
  • `mask`: Both `ref` and stencil-buffer are bitwise ANDed with `mask`.

☆ `glStencilOp` (fail, zfail, zpass)
  • Specifies how the data in the stencil buffer is updated when a fragment passes or fails the test.