Visibility

Determining which objects / triangles / pixels can be seen

Methods

- view volume culling
- view volume clipping
- backface culling
- z-buffer occlusion test
- painter’s algorithm & BSP trees
- occlusion culling
- raycasting (and raytracing)
**View Volume Culling (for triangles)**

Idea: cull a triangle if all vertices are outside the view volume

No! This will falsely cull some triangles!

cull = true for each vertex \(V\)
for each view-volume plane \(P\)

cull if all vertices of a triangle/polygon/object are "outside" not at least one of the view-volume planes

This happens in hardware for triangles, "fixed function" part of pipeline.

**View Volume Culling (for objects)**

Software!

Idea: fast cull of entire objects

Bounding sphere:
cull if \(\text{dist}(C, \text{plane}) > r\) for at least one \(W\) plane

Bounding box:
cull if all vertices are "outside" for at least one \(W\) plane
View Volume Clipping

2D clipping

max # of edges for a clipped triangle?

general procedure to produce a clipped polygon: (Sutherland-Hodgeman)

procedure for triangles with bounding-box scan conversion:

View Volume Clipping

Sutherland Hodgeman clipping

original: 1, 2, 3, 4, 5, 1
clip L: 1, 2, A, B, 4, 5, 1
clip B: 1, 2, A, B, C, D, 5, 1
clip R: 1, 2, A, B, C, E, F, 1
clip T: (unchanged)

for each side of clipping window also add: clip near, clip far.
for each edge of polygon output points based upon the following table

<table>
<thead>
<tr>
<th>case</th>
<th>first point</th>
<th>second point</th>
<th>point(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>inside</td>
<td>inside</td>
<td>second point</td>
</tr>
<tr>
<td>2</td>
<td>inside</td>
<td>outside</td>
<td>intersection point</td>
</tr>
<tr>
<td>3</td>
<td>outside</td>
<td>outside</td>
<td>none</td>
</tr>
<tr>
<td>4</td>
<td>outside</td>
<td>inside</td>
<td>intersection point and second point</td>
</tr>
</tbody>
</table>

How to compute 3D intersection points? => in a few slides
Clipping in VCS

Plane equations

Orthographic View Volume

- left: \( x - \text{left} = 0 \)
- right: \( -x + \text{right} = 0 \)
- bottom: \( y - \text{bottom} = 0 \)
- top: \( -y + \text{top} = 0 \)
- front: \( -z - \text{near} = 0 \)
- back: \( z + \text{far} = 0 \)

Perspective View Volume

- left: \( x + \text{left} z/\text{near} = 0 \)
- right: \( -x - \text{right} z/\text{near} = 0 \)
- top: \( -y - \text{top} z/\text{near} = 0 \)
- bottom: \( y + \text{bottom} z/\text{near} = 0 \)
- front: \( -z - \text{near} = 0 \)
- back: \( z + \text{far} = 0 \)

Note: clipping in VCS works fine, but the actual plane equations depend on the view volume parameters. Why not just clip to \( \pm 1 \) for \( x, y, z \) in NDCS?

---

Clipping in NDCS (?)

(no!)

Top view: line goes from \( P_2 \) behind the eye, to \( P_1 \), in front.

VCS, VCS

NDCS

P2 projects here!

NDCS

\[ \begin{bmatrix} 1 \\ 1 \\ -5/3 \\ -8/3 \\ -1 \end{bmatrix} \]

<table>
<thead>
<tr>
<th>VCS</th>
<th>CCS</th>
<th>NDCS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 0, -2)</td>
<td>(1, 0, 2/3, 2)</td>
<td>(1/2, 0, 1/3)</td>
</tr>
<tr>
<td>(0, 0, 2)</td>
<td>(0, 0, -6, -2)</td>
<td>(0, 0, 3)</td>
</tr>
</tbody>
</table>
Clipping in CCS

Works! Usually where clipping is done.

**NDCS:**
\[-1 \leq \frac{x_{ccs}}{h_{ccs}} \leq 1\]

**CCS:**
\[-h_{ccs} \leq x_{ccs} \leq +h_{ccs}\]

Use this instead.

**Canonical plane equations:**

left: \[ x + h = f(P_{ccs}) \]
right: \[ -x + h = f(P_{ccs}) \]
bottom: \[ y + h = f(P_{ccs}) \]
top: \[ -y + h = f(P_{ccs}) \]

\[ h < 0 \text{ means point is behind the eye.} \]

**Near:** \[ z + h = f(P_{ccs}) \]
**Far:** \[ -z + h = f(P_{ccs}) \]

Clipping in CCS

*Example (skip)*
Line-Plane intersection

Computation of a normal to a plane:
\[ \vec{N} = (\vec{P}_2 - \vec{P}_0) \times (\vec{P}_1 - \vec{P}_0) \]

Plane equation:
\[ A \vec{x} + B \vec{y} + C \vec{z} + D = 0 \]
\[ <A,B,C> \cdot <x,y,z> + D = 0 \]
\[ \vec{N} \cdot \vec{P} + D = 0 \]

To solve for \( D \): substitute any point:
\[ e.g. \quad \vec{N} \cdot \vec{P}_1 + D = 0 \quad \Rightarrow \quad D = -\vec{N} \cdot \vec{P}_1 \]

To find intersection point:
\[ P = \vec{P}_0 + t(\vec{P}_1 - \vec{P}_0) \]

Backface Culling in VCS

Idea: cull if \( N_z < 0 \)
Works, but it will miss culling some faces. e.g.

Better: "cull if the eye is below plane"
Math:
\[ \vec{N} \cdot \vec{P} + D = f(x,y,z) \]
where \( D = -\vec{N} \cdot \vec{P}_1 \)
\[ \vec{P}_{eye} = (0,0,0) \]
\[ \Rightarrow \quad -\vec{N} \cdot \vec{P}_1 < 0 \quad \text{? cull!} \]
Backface Culling in NDCS

Cull if eye is below plane.

Cull if the normal points away, i.e., \( N_z > 0 \).

Computing Surface Normals

**Method 1**

Take cross product
\[
\mathbf{N} = (\mathbf{P}_4 - \mathbf{P}_5) \times (\mathbf{P}_1 - \mathbf{P}_5)
\]

**Method 2**

Use projected areas onto the \( xy, yz, xz \) planes.

\[
N_x = \sum_{i=1}^{n} (y_i - y_j)(z_i + z_j) \\
N_y = \sum_{i=1}^{n} (z_i - z_j)(x_i + x_j) \\
N_z = \sum_{i=1}^{n} (x_i - x_j)(y_i + y_j)
\]

Ok to skip.
Transforming Normals

Using $h=0$

$$
\begin{bmatrix}
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
N_x \\
N_y \\
N_z \\
0
\end{bmatrix}
$$

Problem with non-uniform scaling:

E.g. $N < 1, 1>$ Scale(2,1)

Wrong answer!

Transforming Normals

develop a normal transformation matrix:

Consider a plane: $Ax + By + Cz + D = 0$

Write this as

$$
\begin{bmatrix}
A & B & C & D
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} = 0
$$

$N' = \hat{M} N$

$$
\begin{bmatrix}
A & B & C & D
\end{bmatrix}
\begin{bmatrix}
N^T
\end{bmatrix} = 0
$$

$\hat{M} M = I$

$\hat{M} = (M^{-1})$
Occlusion

view occluded by objects in front of a given pixel or polygon?

- image space algorithms:
  - operate on pixels or scan-lines
  - visibility resolved to the precision of the display
  - e.g.: Z-buffer
- object space algorithms:
  - explicitly compute visible portions of polygons
  - painter’s algorithm: depth-sorting, BSP trees

Z-buffer

store $\langle r, g, b, z \rangle$ for each pixel

```plaintext
for all i, j {
  Depth[i, j] = MAX_DEPTH
  Image[i, j] = BACKGROUND_COLOUR
}
for all polygons P {
  project vertices into screen-space, i.e., DCS
  for all pixels in P {
    if (Z_pixel < Depth[i, j]) { // closer?
      Image[i, j] = C_pixel // overwrite pixel
      Depth[i, j] = Z_pixel // overwrite z
    }
  }
}
```
Z-buffer

- hardware support
- extra memory
- jaggies, i.e., steps along intersections
- poor performance for high depth complexity scenes;
  - use occlusion culling to mitigate this

Occlusion Culling

- occlusion queries
  - virtual render of bounding box
- precomputed visibility tables
  - store a list of visible cells
- horizon maps
  - for terrain models
Painter's Algorithm

- draw polygons from back-to-front, i.e., sorted by z
  - e.g. draw A, C, B
- problems:
  1. Which z value to use?
  2. What about cyclic overlap?
  3. What about intersecting geometry?
- fix this by cutting polygons as needed, e.g. BSP trees
  (see upcoming slides)

Binary Space Partition (BSP)
trees recursively divides 3D spaces into half-spaces

- object-space method
- cuts intersecting polygons
- produces a back-to-front ordering

Steps:
- build the BSP tree (do this only once)
- for each render, traverse the BSP in a view-dependent fashion
BSP trees (example)

Build tree: (insertion in alphabetical order)

Using a BSP tree

Building a BSP tree

```c
BSPtree *BSPmaketree(polygon list) {
    choose a polygon as the tree root
    for all other polygons {
        if polygon is in front, add to front list
        if polygon is behind, add to behind list
        else split polygon and add one part to each list
    }
    BSPtree = BSPcombinetree(BSPmaketree(front list),
                            root, BSPmaketree(behind list) )
}
```
Using a BSP tree

*producing a back-to-front ordering*

```c
DrawTree(BSPtree) {
    if (eye is in front of root) {
        DrawTree(BSPtree->behind)
        DrawPoly(BSPtree->root)
        DrawPoly(BSPtree->front)
    } else {
        DrawTree(BSPtree->front)
        DrawPoly(BSPtree->root)
        DrawTree(BSPtree->behind)
    }
}
```

Visibility in Practice:
WebGL, OpenGL

Commonly supported by hardware & OpenGL / DirectX
- view volume culling (for triangles)
- view volume clipping
- backface culling
- z-buffer occlusion test

Software, i.e., on your own
- view volume culling (for objects)
- painter’s algorithm & BSP trees
- occlusion culling
Raycasting and Raytracing

*alternative to projective rendering*

- for each pixel $p$
  - construct ray $r$ from eye through $p$
  - intersect $r$ with all polygons or objects
  - color $p$ according to closest surface