Polygon Clipping

Goal: Remove points and parts of objects outside the view volume.

Clipping is especially important when objects/scenes contain large numbers of polygons (most of which are not in the field of view).

The View Volume

View volume determined by 6 parameters: \( f, F, L, R, T, B \)

- Near plane = image/visioning plane
- Far plane = points behind it are not drawn
- B, T, L, R = bounding sides of view volume
The Field of View

The parameters $L, R, T, B$ control which 3D points are within the field of view (FOV).

- $\tan x = \frac{T}{f}$
- $\tan y = \frac{B}{f}$
- angular FOV = $\alpha + \beta$
- for fixed $T, B$:
  - small $f$ = wide FOV
  - large $f$ = small FOV

Transformation Chain for 3D Viewing (complete)

Object-to-world transformation ($M_{ow}$)

$$\begin{bmatrix} X_o \\ Y_o \\ Z_o \end{bmatrix} = \begin{bmatrix} R_{ow} & E_{ow} \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

a 4x4 matrix that maps object-centered coords to world-centered coords

World-to-camera transformation ($M_{wc}$)

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = \begin{bmatrix} R_{wc} & E_{wc} \end{bmatrix} \begin{bmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{bmatrix}$$

a 4x4 matrix that maps world-centered coords to camera-centered coords

Camera-to-canonical view transformation ($M_{cv}$)

$$\begin{bmatrix} X_v \\ Y_v \\ Z_v \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

a 4x4 matrix that maps camera-centered coords to canonical view coords

Canonical view-to-image transformation ($M_{vi}$) (a.k.a. projection)

$$\begin{bmatrix} X_i \\ Y_i \\ Z_i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X_v \\ Y_v \\ Z_v \\ 1 \end{bmatrix}$$

a 3x3 matrix that maps canonical view 3D coords to 2D image coords

Orthographic projection matrix
The Canonical View Volume Transform

\[ M_{CV} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
x_c \\
y_c \\
z_c \\
1
\end{bmatrix} \]

rays converging to center of projection \[ \Rightarrow \] rays parallel to z-axis

points projecting to same pixel \[ \Rightarrow \] points projecting to same pixel

The Canonical View Volume Transform

Why do we care about transforming the viewing “cone” of lines of sight into a cube where lines of sight are parallel?

Ans: Clipping & visibility computation become much simpler
Clipping in the Canonical View Volume

Clipping rule very simple when done in a canonical view volume shaped like a CUBE whose faces are on planes $x = \pm 1$, $y = \pm 1$, $z = \pm 1$

...Triangles whose edges lie outside the canonical view volume are not drawn
...Triangles whose edges cross the plane(s) $x = \pm 1$, $y = \pm 1$, $z = \pm 1$ must be clipped
Clipping in the Canonical View Volume

Clipping rule very simple when done in a canonical view volume shaped like a CUBE whose faces are on planes \( x = \pm 1, y = \pm 1, z = \pm 1 \).

Triangles whose edges lie outside the canonical view volume are not drawn. Triangles whose edges cross the plane(s) \( x = \pm 1, y = \pm 1, z = \pm 1 \) must be clipped.

---

Topic 8:

Visibility

- Elementary visibility computations:
  - Clipping
  - “Shaping” the canonical view volume
  - Backface culling
- Algorithms for visibility determination:
  - Z-Buffering
  - Painter’s algorithm
  - BSP Trees
“Shaping” the Canonical View Volume

Q: What should the canonical view transform be to map the view volume onto a cube?

\[
\begin{bmatrix}
X_v \\
Y_v \\
Z_v \\
1
\end{bmatrix} = \begin{bmatrix}
? & 0 & ? & 0 \\
0 & ? & ? & 0 \\
0 & 0 & ? & 0 \\
0 & 0 & 1 & 0
\end{bmatrix} \begin{bmatrix}
X_c \\
Y_c \\
Z_c \\
1
\end{bmatrix}
\]

If point \((x_v, y_v, z_v)\) lies inside this view volume \(\iff\) its \((x_c, y_c, z_c)\) coordinates after the transform will be between \(-1\) and \(1\).

“Shaping” the Transformed z-Coordinates

Q: What should the canonical view transform be to map the view volume onto a cube?

\[
\begin{bmatrix}
Z_v \\
1
\end{bmatrix} = \begin{bmatrix}
(aZ_c + b) \\
\frac{Z_c}{Z_c}
\end{bmatrix} \begin{bmatrix}
X_c \\
Y_c \\
Z_c \\
1
\end{bmatrix}
\]

\[
Z_v = \frac{f}{Z_c} (aZ_c + b)
\]

\[
a = \frac{(f + F)}{f(F - F)} \quad b = \frac{2f}{F - F}
\]

We have two constraint eqns:

\[
\begin{align*}
b &= \frac{af + b}{f + F} \\
1 &= \frac{af - b}{f + F} \\
F &= af + b \\
F &= af - b
\end{align*}
\]
The Pseudo-Depth of a 3D Point

Q: What should the canonical view transform be to map the view volume onto a cube?

-called the pseudo-depth-

\[
\begin{bmatrix}
Z_v \\
1
\end{bmatrix}
= \begin{bmatrix}
0 & 0 & \frac{2}{f} & \frac{2}{f} \\
0 & 1 / f & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x_c \\
y_c \\
z_c \\
1
\end{bmatrix}
\]

"true depth" of the point

Key property of pseudo-depth: it is monotonic with \( z_c \)

\[ x = x_c \]

\[ y = y_c \]

\[ z = z_c \]

\[ -z = z_c \]

\[ -x = x_c \]

\[ -y = y_c \]

\[ -z = z_c \]
**“Shaping” the Transformed y-Coordinates**

**Q:** What should the canonical view transform be to map the view volume onto a cube?

\[
\begin{bmatrix}
  y_c \\
  z_c \\
  1
\end{bmatrix}
= \begin{bmatrix}
  a & b & c \\
  d & e & f \\
  g & h & i
\end{bmatrix}
\begin{bmatrix}
  x_c \\
  y_c \\
  z_c
\end{bmatrix}
\]

Consider the point \((0, t, -5)\):

\[
\begin{align*}
(aT - bf)x + f & = 1 \\
(aB - bf)y + f & = -1
\end{align*}
\]

\[
\begin{align*}
aT - bf & = -1 \\
& \Rightarrow \quad aT - bf = -1
\end{align*}
\]

\[
\begin{align*}
aT - Ab & = -2 \\
& \Rightarrow \quad \begin{array}{c}
\frac{a}{b} = \frac{2}{T} \\
\frac{b}{T} = \frac{2}{b(T)}
\end{array}
\end{align*}
\]

We have 2 constraints:

\[
\begin{align*}
(aT + bT)c + f & = 1 \\
(aB + bT)c + f & = -1
\end{align*}
\]
“Shaping” the Transformed y-Coordinates

Q: What should the canonical view transform be to map the view volume onto a cube?

\[
\begin{bmatrix}
    y_v \\
    z_v \\
    1
\end{bmatrix} =
\begin{bmatrix}
    0 & 2 & 0 & 0 \\
    0 & 0 & 2 & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x_c \\
    y_c \\
    z_c
\end{bmatrix}
\]

plane bounding view volume becomes parallel to xz-plane

“Shaping” the Transformed x-Coordinates

Q: What should the canonical view transform be to map the view volume onto a cube?

\[
\begin{bmatrix}
    x_v \\
    y_v \\
    z_v \\
    1
\end{bmatrix} =
\begin{bmatrix}
    2 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 1 & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
    x_c \\
    y_c \\
    z_c
\end{bmatrix}
\]

1st row computed exactly the same way as for the scaling along y

plane bounding view volume becomes parallel to yz-plane
The Full Canonical View Transformation

Q: What should the canonical view transform be to map the view volume onto a cube?

\[
\begin{bmatrix}
X_v \\
Y_v \\
Z_v \\
1
\end{bmatrix} =
\begin{bmatrix}
\frac{2x - L - R}{2f} \\
\frac{2y - B - T}{2f} \\
\frac{2z - F - B}{2f} \\
0
\end{bmatrix}
\times
\begin{bmatrix}
X_c \\
Y_c \\
Z_c \\
1
\end{bmatrix}
\times
\begin{bmatrix}
\frac{-L + R}{2f} \\
\frac{0}{2f} \\
0 \\
0
\end{bmatrix}
\]

If point \((x_c, y_c, z_c)\) lies inside this view volume \(\iff\) its \((x_v, y_v, z_v)\) coordinates after the transform will be between \(-1\) and \(1\).

---

Topic 8:

Visibility

- Elementary visibility computations:
  - Clipping
  - "Shaping" the canonical view volume
  - Backface culling
- Algorithms for visibility determination
  - Z-Buffering
  - Painter's algorithm
  - BSP Trees
Less is More...

Clipping is a rudimentary form of the following principle:

Don't spend cycles drawing what you don't have to

(= whatever will not contribute to final image)

Two other ways this is applied:
- Backface culling
- Visibility determination

Backface Culling

Goal: Remove surface patches that point away from the camera (i.e., the back-facing patches)

Back-facing faces on the wireframe:
- ADHE, EHGF, AEFB

* will not be drawn
Front-facing vs. Back-facing Polygons

Goal: Remove surface patches that point away from the camera (i.e., the back-facing patches)

---

Backface Culling Criterion

Goal: Remove surface patches that point away from the camera (i.e., the back-facing patches)

If \( \varphi < 90^\circ \) =>
patch faces away from camera =>
CULL

Culling criterion:
\[
(\overrightarrow{p-e} \cdot \overrightarrow{n}) > 0 \Rightarrow \text{CULL} \\
(\overrightarrow{p-e} \cdot \overrightarrow{n}) \leq 0 \Rightarrow \text{DO NOT CULL (may be visible)}
\]
Computing Outward-Facing Normals

Goal: Remove surface patches that point away from the camera (i.e., the back-facing patches)

If $\theta < 90^\circ$ ⇒
patch faces away from camera ⇒
CULL

Computing $\hat{n}$:
- If $\vec{P}_1, \vec{P}_2, \vec{P}_3$ are patch vertices in CCW order as seen from outside:
  $$\vec{m} = (\vec{P}_2 - \vec{P}_1) \times (\vec{P}_3 - \vec{P}_1)$$
  (uses right-hand rule)
  - do computation in world coordinates (no point in transforming vertices/edges that will never be drawn)

Backface Culling is not Enough ...

Which faces will not be drawn if backface culling is applied to this polyhedron?

This face is not fully visible!
(red edges should not be drawn)
Topic 8:

Visibility

- Elementary visibility computations:
  - Clipping
  - “Shaping” the canonical view volume
  - Backface culling
  - Algorithms for visibility determination
    - Z-Buffering
    - Painter’s algorithm
    - BSP Trees

Z-Buffering

Main ideas:
- Visibility determined pixel by pixel during polygon scan-conversion
- To draw an $M \times N$-pixel image, we maintain an $M \times N$ buffer that holds closest $z$-value at each pixel of polygons drawn so far
Z-Buffering & Scan Conversion

Step 0:
- Start with blank image
- Initialize z-buffer to \( z_{\text{max}} \) (always = 1)

Z-Buffering & Scan Conversion

Step 1:
- Scan convert a polygon, copying the polygon's color to each pixel & updating the z-buffer
Z-Buffering & Scan Conversion

Step 2:
- Scan-convert next polygon
- To draw color \( c \) at pixel \((x,y)\) with depth \( d \)
  - if \( d < z\text{buffer}(x,y) \)
    - putpixel \((x,y,c)\)
  - end
    - \( z\text{buffer}(x,y) = d \)

Rendering Order of Polygons

Q: What would be the result if we drew the closest triangle first?

Ans.: The result would be the same.
**Z-Buffering: Pros & Cons**

Advantages:
- Simple, accurate
- Independent of order, polygons are drawn

Disadvantages:
- Memory for z-buffer (not a problem these days!)
- Wasted computation when over-writing distant points

**Triangle Scan Conversion with Z-Buffering**

Step a: Build edge list
- For each scanline, store x-intersection & pseudodepth of each edge

Step b: Fill z-buffer & image pixels
- For each scanline, interpolate pseudodepth along scanline & compare to z-buffer
Triangle Scan Conversion with Z-Buffering

Step a: Build edge list
for each scanline, store x-intersection & pseudodepth of each edge

Step b: Fill z-buffer & image pixels
for each scanline, interpolate pseudodepth along scanline & compare to z-buffer

Edge List Construction

Step a: Build edge list
for each scanline, store x-intersection & pseudodepth of each edge

for each edge \((x_u, y_u, d_u), (x_e, y_e, d_e)\) with \(y_u > y_e\):
- \(x = x_e, d = d_e, \Delta x = x_u - x_e, \Delta d = d_u - d_e, y_u - y_e\)
- for \((y = y_e; y < y_u; y += d)\):
  - add \((x, d)\) to list of scanline \(y\)
- sort the list
  - \(x = x + \Delta x, d = d + \Delta d\)
Scanline Filling with Z-Buffering

\[ y = \text{min}(y_e, y_i, y_c) \]
\[ y' = \text{max}(y_e, y_i, y_c) \]

For \( y = y - j, y + j; j > 0 \)
get \((x_e, d_e)\) and \((x_i, d_i)\) from edge list of \( y \), with \( x_e < x_i \)
\[ \Delta d = \frac{d_e - d_i}{x_i - x_e} \]
for \( k = x_e, d = d_e; k < x_i, k++ \)
if \( d < \text{zbuffer}(x, y) \)
putpixel \((x, y, \text{color})\), \text{zbuffer}(x, y) = d
\[ d = d + \Delta d \]

Step 1: Fill zbuffer & image pixels
For each scanline, interpolate pseudodepth along scanline & compare to zbuffer

---

**Topic 8:**

**Visibility**

- Elementary visibility computations:
  - Clipping
  - "Shaping" the canonical view volume
  - Backface culling
- Algorithms for visibility determination:
  - Z-Buffering
  - Painter's algorithm
  - BSP Trees
The Heedless Painter’s Algorithm

Main Idea:
- Instead of deciding the depth order pixel by pixel, draw the polygons back to front.
- Must sort polygons in decreasing z order.

Question: How do we sort polygons that do not have a single z value (i.e., not parallel to xy-plane)?

Ans: Sort according to depth of farthest vertex

Q: Does this always work? No!

The Heedless Painter’s Algorithm: Limitations

Ans: Sort according to depth of farthest vertex

Q: Does this always work? No!
The Heedless Painter’s Algorithm: Limitations

Another failure case

- This example shows that in some cases there is no sort order that allow correct visibility handling.
- What can we do in this case?
  Ans: Break polygons into smaller (convex) parts

Main issues/problems:
- Depth order depends on eye position (very expensive to recompute for 1000s of polygons at every frame)
- Correct visibilities may not be achievable w/o polygon splitting
Topic 8: Visibility

• Elementary visibility computations:
  Clipping
  “Shaping” the canonical view volume
  Backface culling
• Algorithms for visibility determination
  Z-Buffering
  Painter’s algorithm
  BSP Trees

Binary Space-Partitioning (BSP) Trees

Main idea:
- Maintain a data structure that allows fast computation of depth order for every eye position
- Have mechanism to split polygons if necessary

Main issues/problems:
- Depth order depends on eye position (very expensive to recompute for 1000s of polygons at every frame)
- Correct visibilities may not be achievable w/out polygon splitting
Eye Position & Correct Drawing Order

Q: What should the drawing order be when camera is at $\vec{e}$?
Eye Position & Correct Drawing Order

Top view

\[ \overline{e} \]

world coordinate frame

Q: What should the drawing order be when camera is at \( \overline{e} \)?
Eye Position & Correct Drawing Order

Q: What should the drawing order be when camera is at $E$?

Eye Position & Drawing Order: Basic Idea

If $E, T_2$ on same side of $T_1$

$\Rightarrow$ draw $T_1$, then draw $T_2$

If $E, T_2$ on opposite sides

$\Rightarrow$ draw $T_2$, then draw $T_1$
Eye Position & Drawing Order: Basic Idea

if $f(q_1) \cdot f(e) > 0$

$\Rightarrow$ draw $T_1$,
then draw $T_2$

if $f(q_1) \cdot f(e) < 0$

$\Rightarrow$ draw $T_2$,
then draw $T_1$

The BSP Tree

The BSP tree is an efficient data structure for quickly determining the inside/outside relation between polygons & the camera position.

Two phases:

- Preprocessing phase
  (Tree Construction)
done once
- Rendering phase
  (Tree Traversal)
done whenever eye position changes
BSP Tree Construction: Basic Idea

Idea:
- order faces in some way
- associate a node to each face
- face 1 becomes root node
- for face j, traverse tree to a leaf node i & add it as "in" or "out" child

BSP Tree Construction: Splitting Faces

Idea:
- order faces in some way
- associate a node to each face
- face 1 becomes root node
- for face j, traverse tree to a leaf node i & add it as "in" or "out" child
- must split face into two pieces before BSP insertion
Rendering with BSP Trees: Main Idea

If $\overline{E}$ is outside face $i$ (e.g. 1)
nothing inside $i$ can occlude $i$
$\implies$ can be drawn before $i$

after drawing the inside faces, draw $i$ & then the outside faces

Rendering with BSP Trees: Algorithm

1. if $E$ outside face $i$
   1. draw everything inside $i$
   1. draw $i$
   1. draw everything outside $i$
Rendering with BSP Trees: Algorithm

1. If \( E \) inside face \( i \)
   - draw everything outside \( i \)
   - draw \( i \)
   - draw everything inside \( i \)

BSP Trees & Backface Culling

1. If \( E \) inside \( i \) then \( i \) is back-facing
2. Everything here shown in 2D but construction is identical in 3D
3. Only complication is triangle/polygon splitting (see Ch. 8.1 in book)