Visibility

- view volume culling
- view volume clipping
- backface culling
- z-buffer occlusion test
- painter’s algorithm & BSP trees
- occlusion culling
- raytracing

View Volume Culling

polygons outside the view volume?

View Volume Clipping

polygons partly outside of view volume?
View Volume Clipping

Sutherland Hodgeman clipping

for each side of clipping window
for each edge of polygon
output points based upon the following table


Clipping in VCS

VCS

Clipping in NDCS

NDCS

Clipping in CCS

NDCS:

CCS:
**Line-Plane intersection**

Plane equation:

\[ A x + B y + C z + D = 0 \]

1. \[ A x + B y + C z + D = 0 \]
2. \[ P(t) = P_a + t(P_b - P_a) \]
3. \[ N \cdot (P_a + t(P_b - P_a)) + D = 0 \]
4. \[ t = -D - N \cdot P_a \]
5. \[ N \cdot (P_b - N \cdot P_a) \]
6. \[ N \cdot P_a - N \cdot P_a \]
7. \[ N \cdot P_b - N \cdot P_a \]
8. \[ N \cdot P_a - N \cdot P_a \]
9. \[ N \cdot P_b - N \cdot P_a \]

**Backface Culling in VCS**

Idea: flip if \( N_z < 0 \)

Problem: polygon \( A \) would not be culled.

Fix: test to see if eye is below.

If below, then flip.

Plane: \[ A x + B y + C z + D = 0 \]

For eg: \( P(0,0,0) \)
\[ \frac{0}{A} = -N \cdot P \]
If \( 0 < 0 \) then flip, if \( -N \cdot P < 0 \) then.

**Backface Culling in NDCS**

Can cull in either VCS plane or NDCS plane.

**Computing Surface Normals**

1. **Method 1**
   
   \[ N = (P_2 - P_1) \times (P_3 - P_1) \]

   (more often use CCW)

2. **Method 2**

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

   (area on xz plane)
   
   \[ N_x = \sum_{i=1}^{n} (y_i - z_i)(z_i + x_i) \]
   
   (area on yz plane)
**Occlusion**

- **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**

- **Hardware support**
- **Limitations:**
  - Extra memory
  - Jaggies in, stereo aliasing, intersecting objects
  - Poor performance for high depth complexity

---

**Painter's Algorithm**

- **Step 1:** Sort by depth
  - Which Z value to use?
    - Cyclic overlap
    - Intersecting geometry?
  - Draw the polygons from back to front
- **Step 2:** Solve for intersections
  - Use BSP tree

---

**Z-buffer**

store \((r,g,b,z)\) 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] {
            Image[i,j] = C_pixel
            Depth[i,j] = Z_pixel
        }
    }
}
```
Binary Space Partition (BSP) trees

- object-space method
- produces a back-to-front ordering
- build the BSP tree once
- traverse the BSP in a view-dependent fashion

BSP trees (example)

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

```c
using a back-to-front ordering

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)
    }
}
```
**Occlusion Culling**

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

**Visibility in Practice: WebGL, OpenGL**

- view volume culling
- view volume clipping
- backface culling
- z-buffer occlusion test
- 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

**Transforming Normals**

Using $h=0$

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

Problem

$$\begin{bmatrix} N_x \\ N_y \\ N_z \end{bmatrix} = \begin{bmatrix} h \end{bmatrix}$$

$h=1$ for points
Transforming Normals

develop a normal transformation matrix:

\[
\mathbf{N}_n = (\mathbf{M}_n^{-1})^T \quad \text{matrix b transform normals.}
\]

\[
\begin{bmatrix}
A & B & C & D \\
4 & 2 & 0 & 0 \\
N^T & 0 & 0 & 0
\end{bmatrix}
\begin{bmatrix}
x \\ y \\ z \\ 1
\end{bmatrix}
= p
\]

\[
\begin{bmatrix}
x \\ y \\ z \\ 1
\end{bmatrix} \rightarrow \begin{bmatrix}
x' \\ y' \\ z' \\ 1
\end{bmatrix}
\]

\[
\mathbf{N}_n = (\mathbf{M}_n^{-1})^T \mathbf{M}_n \mathbf{P} = 0
\]

\[
\mathbf{N}_n^T \mathbf{M}_n^T \mathbf{M}_n \mathbf{P} = 0
\]

deduce \[\mathbf{M}_n^T \mathbf{M}_n = \mathbf{I}\]

\[
\mathbf{N}_n = (\mathbf{M}_n^{-1})^T
\]