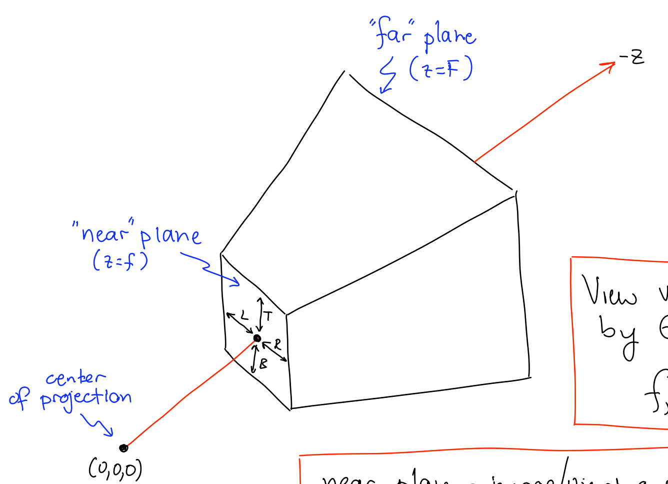
Polygon Clipping

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

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

The View Volume



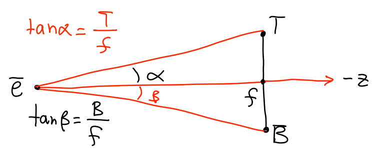
View volume determined by 6 parameters:

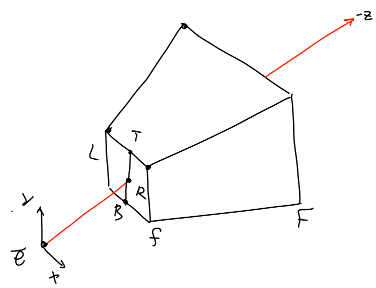
f, F, L, R, T, B

near plane = image/viewing plane
far plane = points behind it are <u>not</u> 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)





- angular $FOV = \alpha + \beta$
- for fixed T, B:
  - small $f \Rightarrow$ wide FOV
  - large $f \Rightarrow$ small FOV

Transformation Chain for 3D Viewing (complete)

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

$$\begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} = \begin{bmatrix} R_{ow} & e_{ow} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_o \\ y_o \\ z_o \\ 1 \end{bmatrix}$$

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

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

$$\begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} R_{wc} & e_{wc} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix}$$

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

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

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

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

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

$$\begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix} \simeq \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix}$$

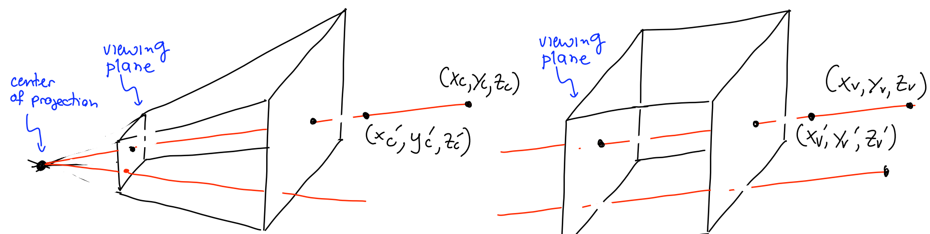
a 4x3 matrix that maps canonical view 3D coords to 2D image coords

Orthographic projection matrix

The Canonical View Volume Transform

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} \cong \overbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & \frac{1}{f} & 0 \end{bmatrix}}^{M_{cv}} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

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

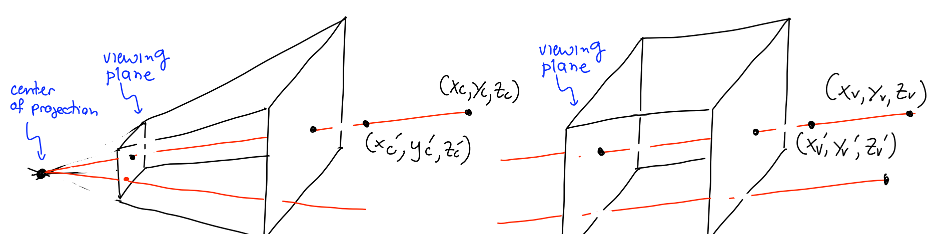


points projecting to same pixel $\Rightarrow$ points projecting to same pixel

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

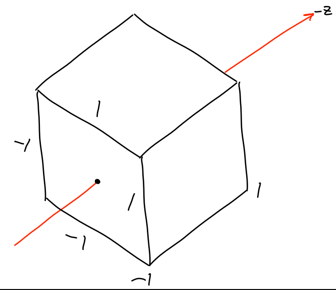


points projecting to same pixel $\Rightarrow$ points projecting to same pixel

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 $x = \pm 1$, $y = \pm 1$, $z = \pm 1$

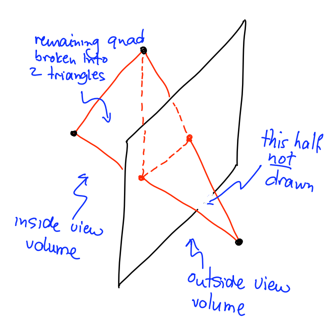
- Triangles whose edges lie outside the canonical view volume are not drawn
- Triangles whose edges cross the plane(s) $x = \pm 1$, $y = \pm 1$, $z = \pm 1$ must be clippled



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 clippled



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 clippled

see Shirley Ch. 12.1 – 12.2

Topic 8:

Visibility

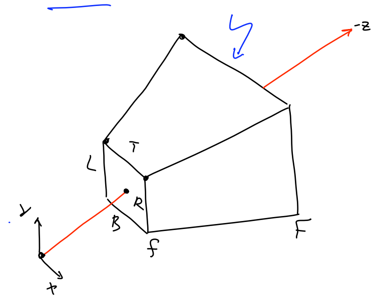
- Elementary visibility computations:
  - Clipping
  - "Shaping" the canonical view volume
  - Backface culling
- Algorithms for visibility determination
  - Z-Buffering
  - Painter's algorithm
  - BSP Trees

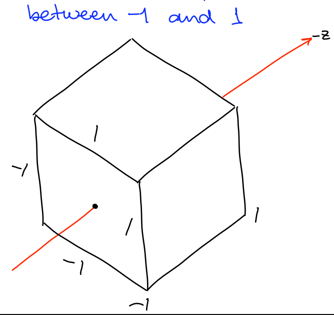
“Shaping” the Canonical View Volume

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

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} \cong \begin{bmatrix} ? & 0 & ? & 0 \\ 0 & ? & ? & 0 \\ 0 & 0 & ? & ? \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

if point (x_c, y_c, z_c) lies inside this view volume $\Longleftrightarrow$ its (x_v, y_v, z_v) 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?

$$\begin{bmatrix} \\ \\ z_v \\ 1 \end{bmatrix} \cong \begin{bmatrix} \\ \\ a z_c + b \\ z_c/f \end{bmatrix} = \begin{bmatrix} & & & \\ & & & \\ \overbrace{0 \quad 0}^{z_v \text{ is independent of } x_c, y_c} & & a & b \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

we have two constraint eqs:

$-1 = af - b$

$1 = \frac{f}{F}(aF - b)$

$\Rightarrow$

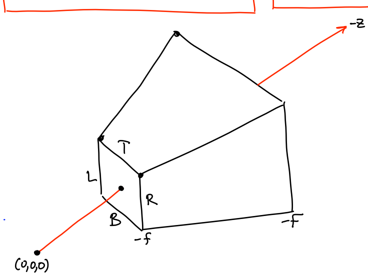
$b = af + 1$

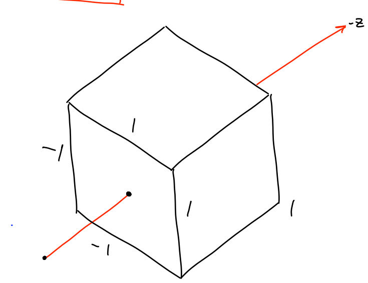
$F = afF - fb$ $\Rightarrow$

$F = afF - af^2 - f \Rightarrow$

$f + F = af(F - f)$

$$\boxed{z_v = \frac{f}{z_c}(a z_c + b)} \qquad \boxed{a = \frac{(f+F)}{f(F-f)} \quad b = \frac{2F}{F-f}}$$





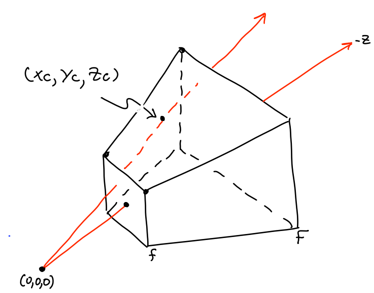
The Pseudo-Depth of a 3D Point

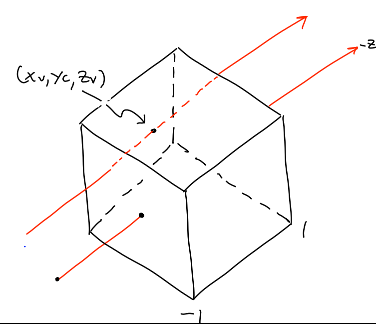
Q: What should the canonical view transform be to map the view volume onto a cube?

called the "pseudo-depth"

$$\begin{bmatrix} \\ \\ z_v \\ 1 \end{bmatrix} \cong \begin{bmatrix} & & & \\ & & & \\ 0 & 0 & \frac{f+F}{f(F-f)} & \frac{2F}{F-f} \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

"true depth" of the point





The Pseudo-Depth of a 3D Point

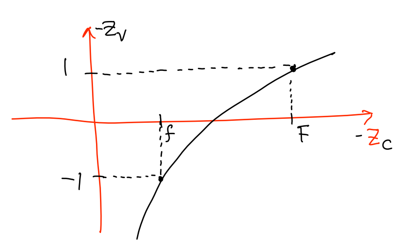
Q: What should the canonical view transform be to map the view volume onto a cube?

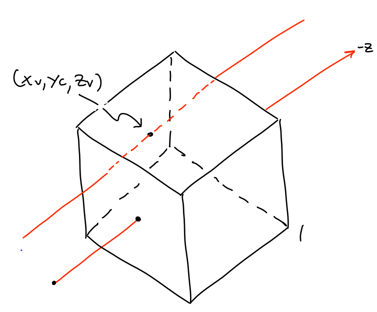
called the "pseudo-depth"

$$\begin{bmatrix} \\ \\ z_v \\ 1 \end{bmatrix} \cong \begin{bmatrix} & & & \\ & & & \\ 0 & 0 & \frac{f+F}{f(F-f)} & \frac{2F}{F-f} \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

"true depth" of the point

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

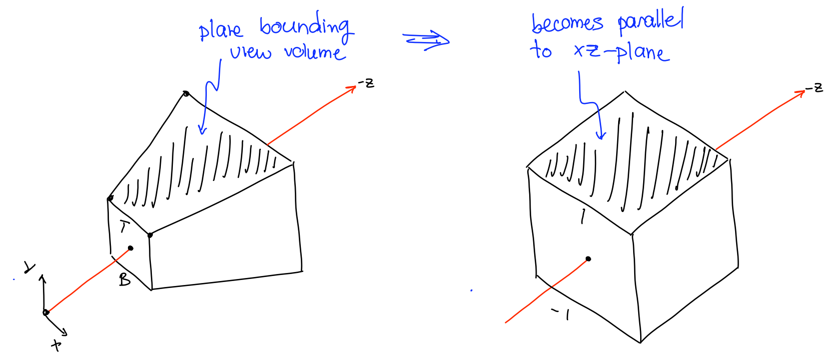




“Shaping” the Transformed y-Coordinates

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

$$\begin{bmatrix} y_v \\ z_v \\ 1 \end{bmatrix} \cong \begin{bmatrix} a y_c + b z_c \\ \\ z_c/f \end{bmatrix} = \begin{bmatrix} 0 & a & b & 0 \\ 0 & 0 & \frac{f+F}{f(F-f)} & \frac{2F}{F-f} \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$



“Shaping” the Transformed y-Coordinates

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

$$\begin{bmatrix} y_v \\ z_v \\ 1 \end{bmatrix} \cong \begin{bmatrix} a y_c + b z_c \\ \\ z_c/f \end{bmatrix} = \begin{bmatrix} 0 & a & b & 0 \\ 0 & 0 & \frac{f+F}{f(F-f)} & \frac{2F}{F-f} \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

we have 2 constraints:

$$(aT + b z_c) f / z_c = 1$$
$$(aB + b z_c) f / z_c = -1$$

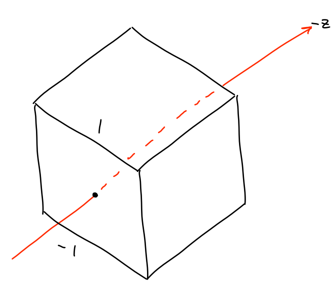
Consider the point $(0, T, -f)$:

$$\left.\begin{aligned} (aT - bf)(-f)/f &= 1 \\ (aB - bf)(-f)/f &= -1 \end{aligned}\right\} \Rightarrow$$

$$\left.\begin{aligned} aT - bf &= -1 \\ aB - bf &= 1 \end{aligned}\right\} \xRightarrow{\text{subtract}}$$

$$aT - aB = -2 \Rightarrow$$

$$\boxed{a = \frac{2}{B-T}} \quad b = \frac{aT+1}{f} \Rightarrow \boxed{b = \frac{B+T}{f(B-T)}}$$

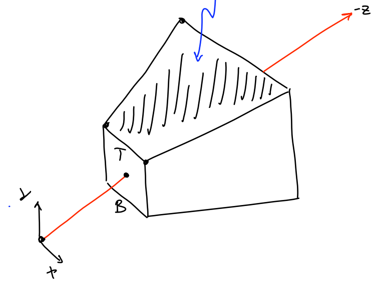


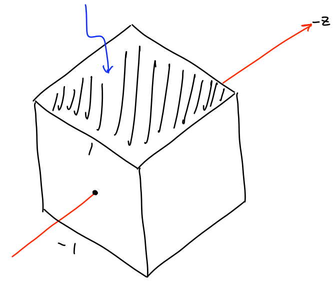
"Shaping" the Transformed y-Coordinates

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

$$\begin{bmatrix} \\ y_v \\ z_v \\ 1 \end{bmatrix} \cong \begin{bmatrix} & & & \\ 0 & \frac{2}{B-T} & \frac{B+T}{f(B-T)} & 0 \\ 0 & 0 & \frac{f+F}{f(F-f)} & \frac{2F}{F-f} \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

plane bounding view volume $\Rightarrow$ becomes parallel to xz-plane





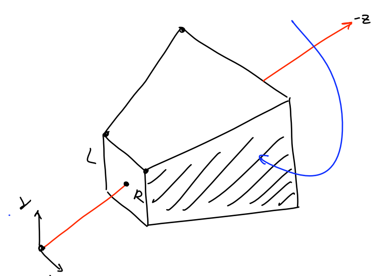
"Shaping" the Transformed x-Coordinates

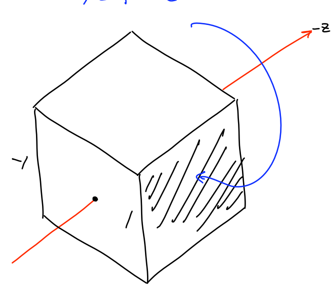
Q: What should the canonical view transform be to map the view volume onto a cube?

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} \cong \begin{bmatrix} \frac{2}{R-L} & 0 & \frac{R+L}{f(R-L)} & 0 \\ 0 & \frac{2}{B-T} & \frac{B+T}{f(B-T)} & 0 \\ 0 & 0 & \frac{f+F}{f(F-f)} & \frac{2F}{F-f} \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

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

plane bounding view volume $\Rightarrow$ becomes parallel to yz-plane



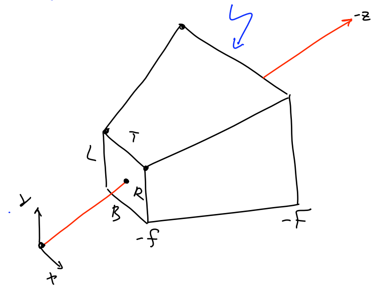


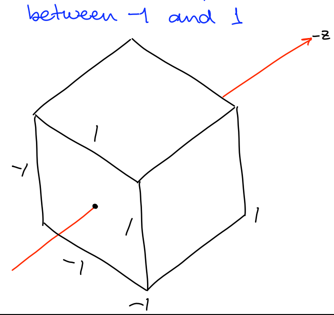
The Full Canonical View Transformation

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

$$\begin{bmatrix} x_v \\ y_v \\ z_v \\ 1 \end{bmatrix} \cong \begin{bmatrix} \frac{2}{L-R} & 0 & \frac{L+R}{f(L-R)} & 0 \\ 0 & \frac{2}{B-T} & \frac{B+T}{f(B-T)} & 0 \\ 0 & 0 & \frac{f+F}{f(F-f)} & \frac{2F}{F-f} \\ 0 & 0 & 1/f & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} \underset{\text{mult by } f}{\cong} \begin{bmatrix} \frac{2f}{L-R} & 0 & \frac{R+L}{L-R} & 0 \\ 0 & \frac{2f}{B-T} & \frac{B+T}{B-T} & 0 \\ 0 & 0 & \frac{f+F}{F-f} & \frac{2Ff}{F-f} \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}$$

if point (x_c, y_c, z_c) lies inside this view volume $\Longleftrightarrow$ 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:

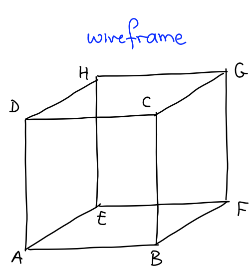
> Don't spend cycles drawing what you don't have to
>
> (= whatever will not contribute to final image)

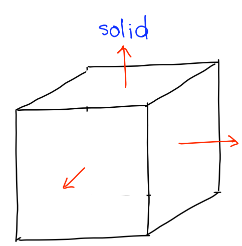
Two other ways this is applied:

- Backface culling
- Visibility determination

Backface Culling

Goal: Remove surface patches that point away from the camera (i.e. the backfacing patches)





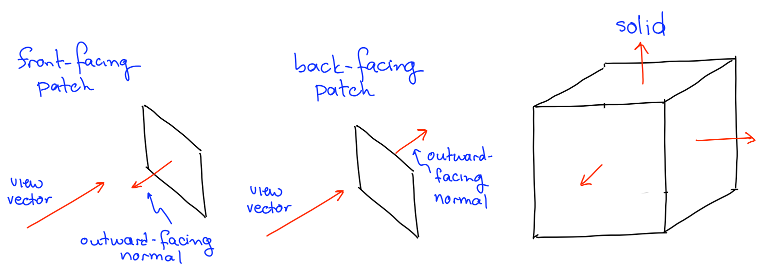
Backfacing faces on the wireframe:

$\boxed{\text{ADHE, EHGF, AEFB}}$

↑ will not be drawn

Front-facing vs. Back-facing Polygons

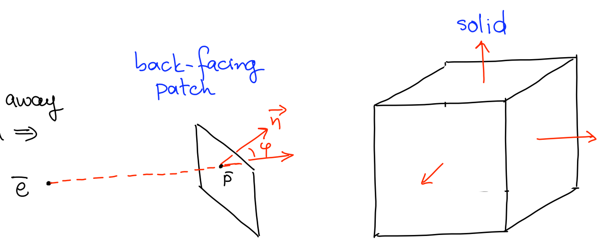
Goal: Remove surface patches that point away from the camera (i.e. the backfacing patches)



Backface Culling Criterion

Goal: Remove surface patches that point away from the camera (i.e. the backfacing patches)

if $\varphi < 90^\circ \Rightarrow$
patch faces away from camera $\Rightarrow$
CULL



Culling criterion:

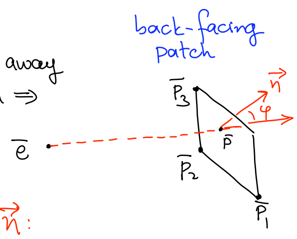
$(\bar{p}-\bar{e})\cdot\vec{n} > 0 \Rightarrow$ CULL

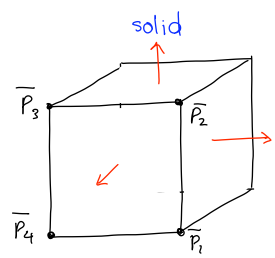
$(\bar{p}-\bar{e})\cdot\vec{n} \leq 0 \Rightarrow$ DO NOT CULL (may be visible)

Computing Outward-Facing Normals

Goal: Remove surface patches that point away from the camera (i.e. the backfacing patches)

if $\varphi < 90^\circ \Rightarrow$ patch faces away from camera $\Rightarrow$ CULL





Computing $\vec{n}$:

- if $\bar{P}_1, \bar{P}_2, \bar{P}_3$ are patch vertices in CCW order as seen from outside:

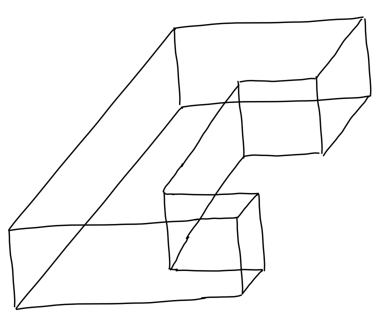
$$\vec{n} = (\bar{P}_2 - \bar{P}_1) \times (\bar{P}_3 - \bar{P}_1)$$

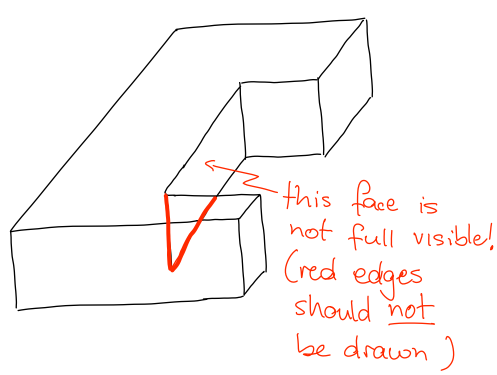
(uses right-hand rule)

- 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?



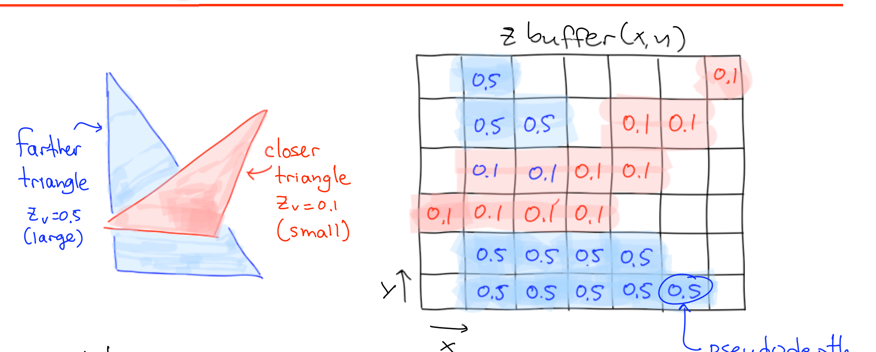


Topic 8:

Visibility

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

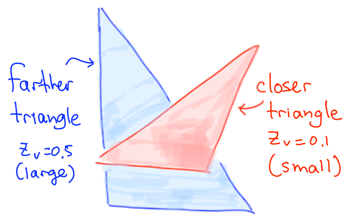
Z-Buffering



Main ideas:

- visibility determined pixel by pixel during polygon scan-conversion
- to draw an MxN-pixel image, we maintain an MxN buffer that holds closest z-value at each pixel of polygons drawn so far

Z-Buffering & Scan Conversion



z buffer(x,y)

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

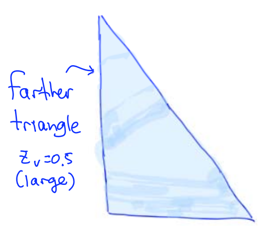
y ↑ x →

initial pseudodepth

Step 0:

- Start with blank image
- Initialize z-buffer to z_{max} (always = 1)

Z-Buffering & Scan Conversion



z buffer(x,y)

1	0.5	1	1	1	1	1
1	0.5	0.5	1	1	1	1
1	0.5	0.5	1	1	1	1
1	0.5	0.5	0.5	1	1	1
1	0.5	0.5	0.5	0.5	1	1
1	0.5	0.5	0.5	0.5	0.5	1

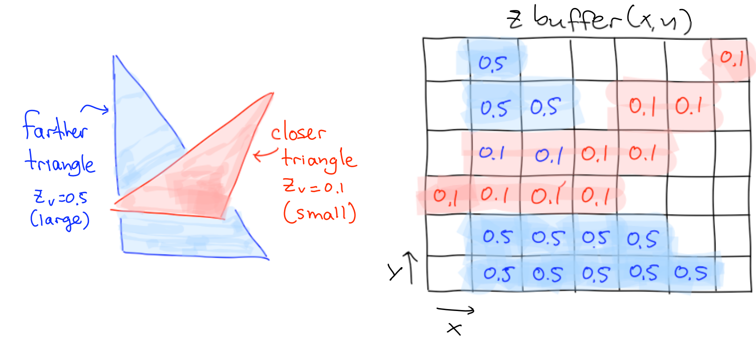
y ↑ x →

pseudodepth (z_v) of triangle at this pixel

Step 1:

- Scan convert a polygon, copying the polygon's color to each pixel & updating the z-buffer

Z-Buffering & Scan Conversion

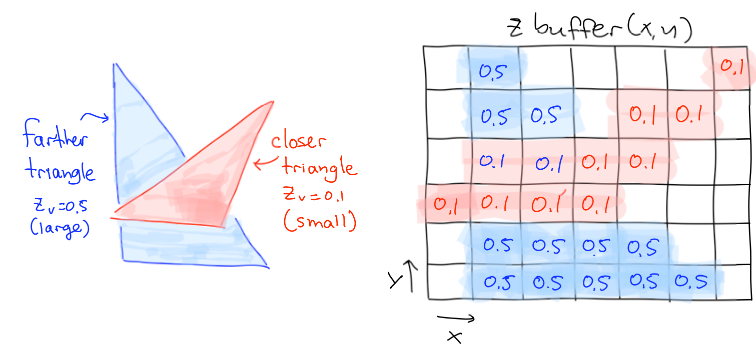


Step 2:

- Scan-convert next polygon
- To draw color c at pixel (x,y) with depth d:

```
if d < zbuffer(x,y)
    putpixel(x,y,c)
    zbuffer(x,y)=d
end
```

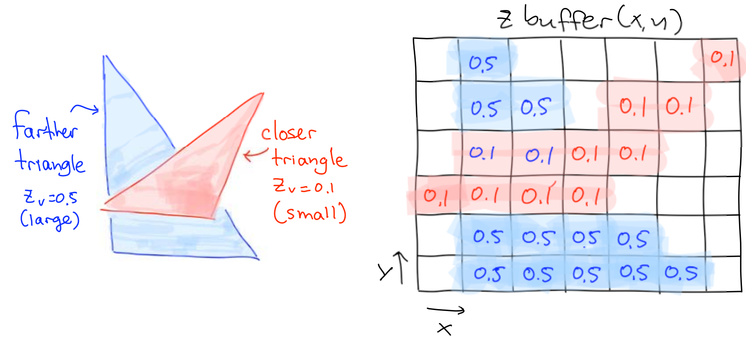
Rendering Order of Polygons



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

Ans: The result would be the <u>same</u>

Z-Buffering: Pros & Cons



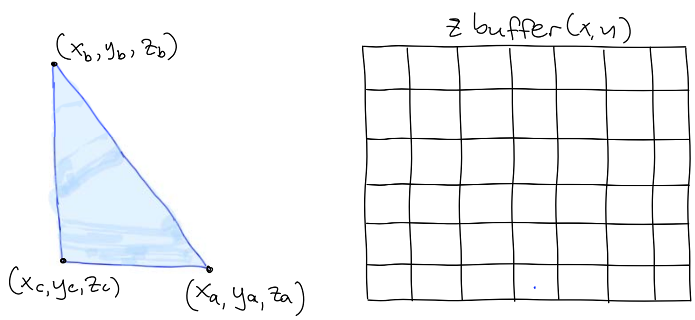
Advantages:

- simple, accurate
- independent of order polygons are drawn

Disadvantages:

- memory for zbuffer (not a problem these days!)
- wasted computation when over-writing distant points

Triangle Scan Conversion with Z-Buffering



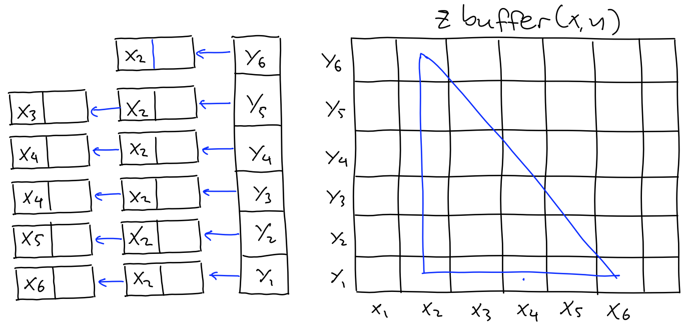
Step a: Build edge list

for each scanline, store x-intersection & pseudodepth of each edge

Step b: Fill zbuffer & image pixels

for each scanline, interpolate pseudodepth along scanline & compare to z-buffer

Triangle Scan Conversion with Z-Buffering



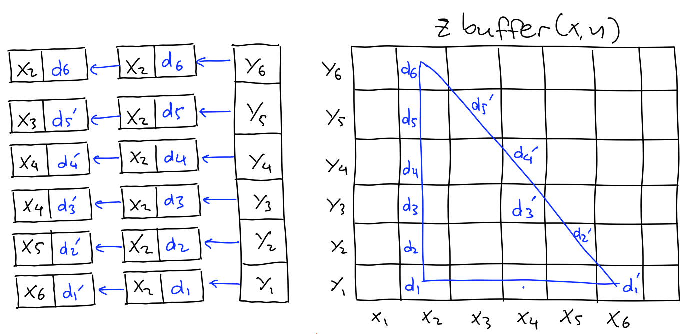
Step a: Build edge list

for each scanline, store x-intersection & pseudodepth of each edge

Step b: Fill zbuffer & 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 $y_u>y_e$:

$x=x_e,\ d=d_e,\ \Delta x=\frac{x_u-x_e}{y_u-y_e},\ \Delta d=\frac{d_u-d_e}{y_u-y_e}$

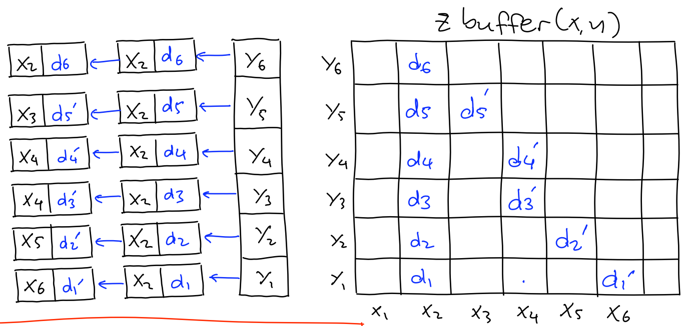
for $(y=y_e;\ y<y_u;\ y{+}{+})$

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- = min(ya, yb, yc)   y+ = max(ya, yb, yc)
for (y=y-; y≤y+; y++)
   get (xl, dl) and (xu, du) from
       edge list of y, with xl < xu
   Δd = (du-dl)/(xu-xl)
   for (x=xl, d=dl; x≤xu; x++)
       if  d < zbuffer(x,y)
           putpixel(x,y,color), zbuffer(x,y)=d
       d = d+Δd
```

Step 6: Fill zbuffer & image pixels

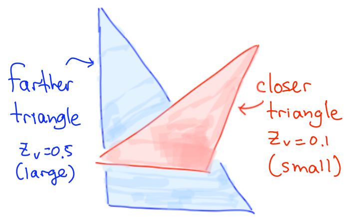
For each scanline, interpolate pseudodepth along scanline & compare to z-buffer

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:

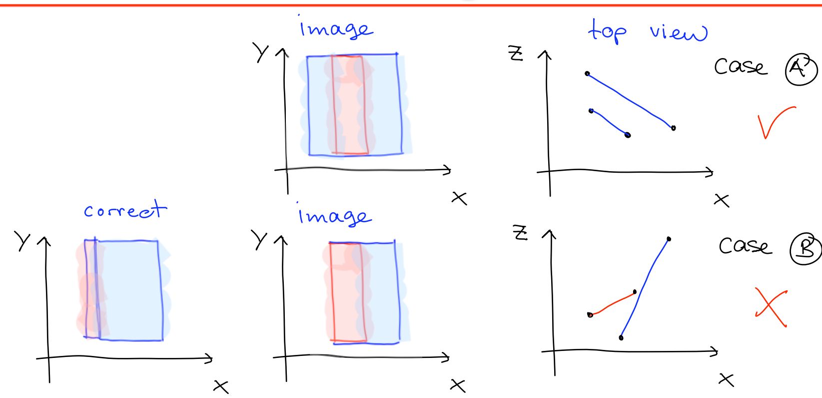
- Instead of deciding the depth order pixel by pixel, draw the polygons back to front
- Must sort polygons in decreasing z order

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

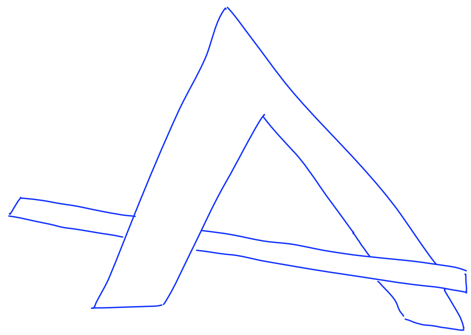


Ans: Sort according to depth of farthest vertex

Q: Does this always work? No!

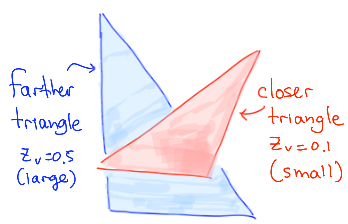
The Heedless Painter's Algorithm: Limitations

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

The Heedless Painter's Algorithm: Limitations



Main issues/problems:

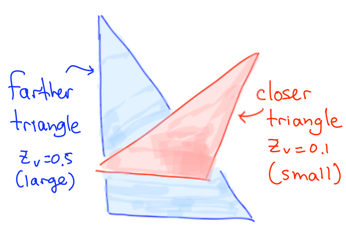
- Depth order depends on eye position (very expensive to recompute for 1000s of polygons at every frame)
- Correct visibilities may not be achievable w/out polygon splitting

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



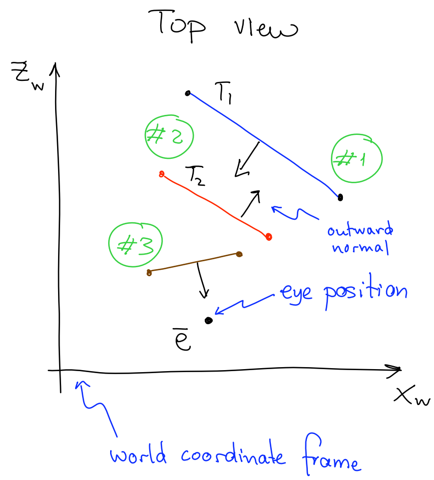
Main idea:

- 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:

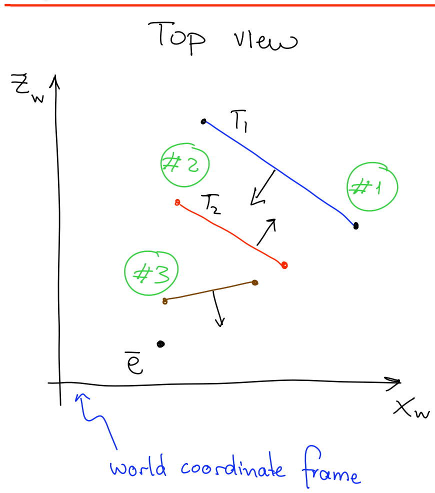
- Depth order depends on eye position (very expensive to recompute for 1000s of polygons at every frame)
- Correct visibilities may not be achievable w/out polygon splitting

Eye Position & Correct Drawing Order



Q: What should the drawing order be when camera is at $\bar{e}$?

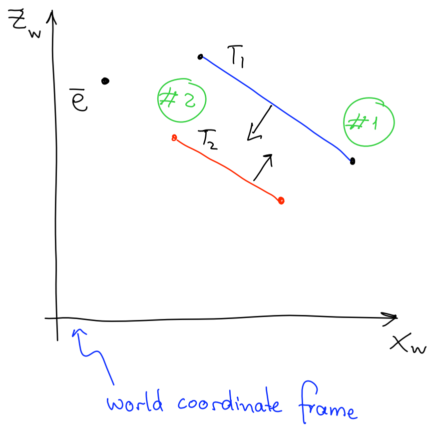
Eye Position & Correct Drawing Order



Q: What should the drawing order be when camera is at $\bar{e}$?

Eye Position & Correct Drawing Order

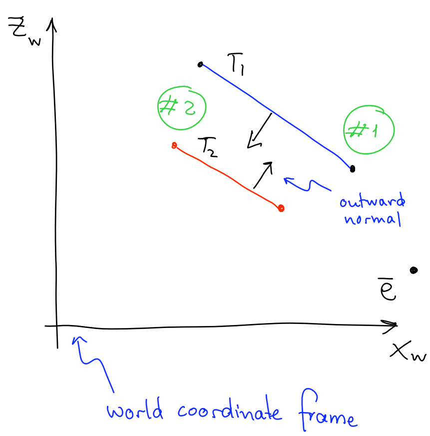
Top view



Q: What should the drawing order be when camera is at $\bar{e}$?

Eye Position & Correct Drawing Order

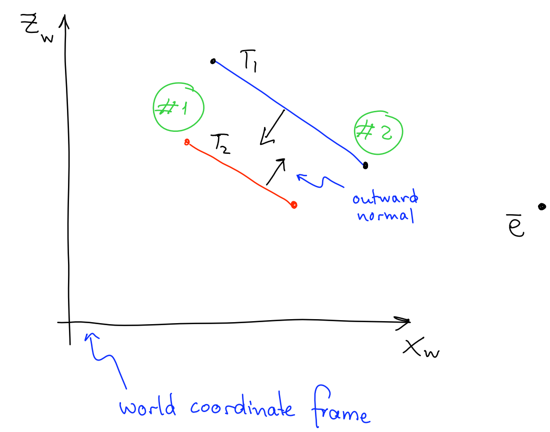
Top view



Q: What should the drawing order be when camera is at $\bar{e}$?

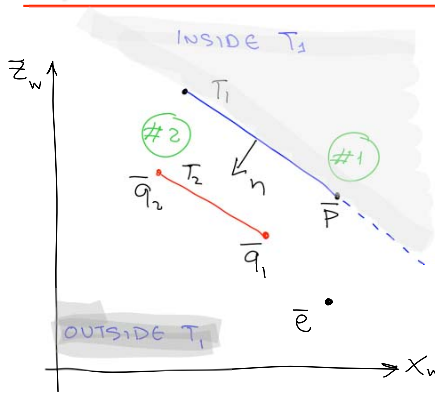
Eye Position & Correct Drawing Order

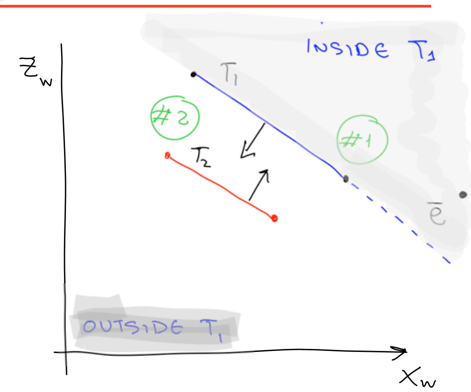
Top view



Q: What should the drawing order be when camera is at $\bar{e}$?

Eye Position & Drawing Order: Basic Idea





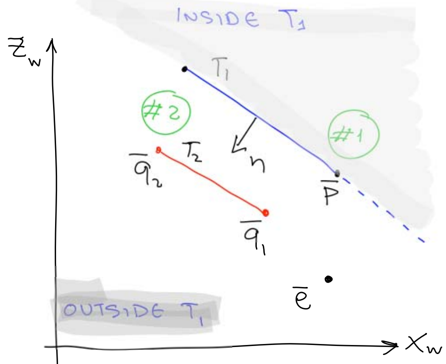
if $\bar{e}, T_2$ on same side of T_1

$\Rightarrow$ draw T_1, then draw T_2

if $\bar{e}, T_2$ on opposite sides

$\Rightarrow$ draw T_2, then draw T_1

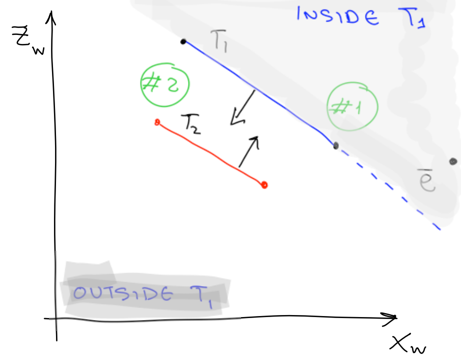
Eye Position & Drawing Order: Basic Idea



implicit plane eq: $f(\bar{q}) = (\bar{q} - \bar{p}) \cdot \vec{n}$

$$\boxed{\text{if } f(\bar{q}_1) \cdot f(\bar{e}) > 0}$$

$\Rightarrow$ draw T_1,
then draw T_2

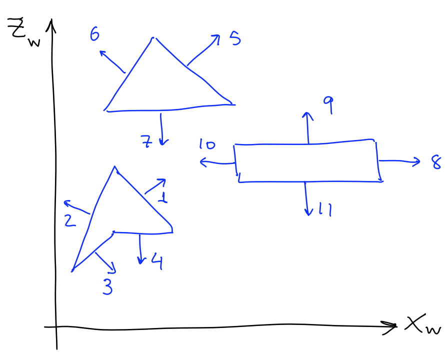


$$\boxed{\text{if } f(\bar{q}_1) \cdot f(\bar{e}) < 0}$$

$\Rightarrow$ 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



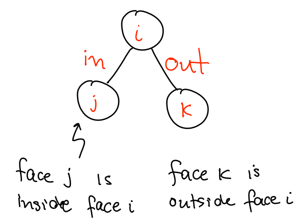
Two phases:

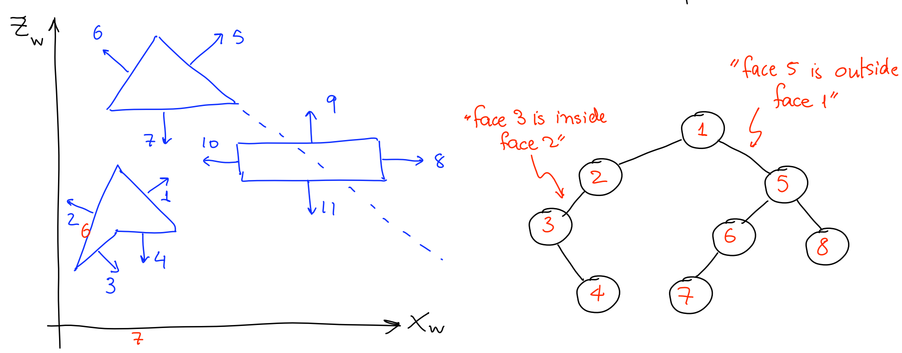
- Preprocessing phase (Tree Construction) ↵ done once per scene
- Rendering phase (Tree Traversal) ↵ done whenever eye position changes

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

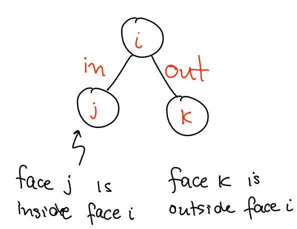


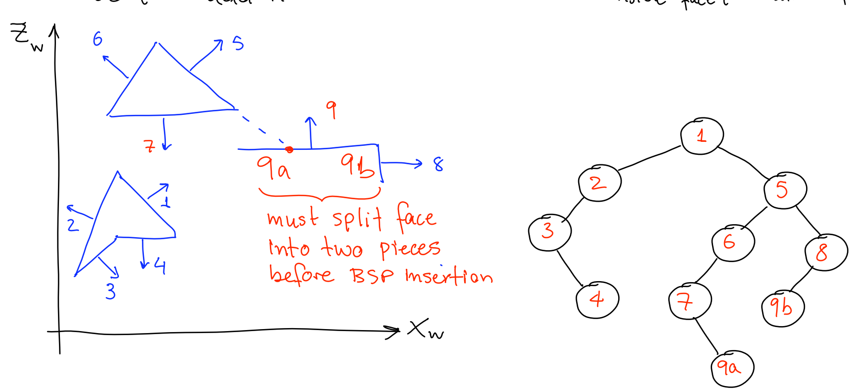


BSP Tree Construction: Splitting Faces

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





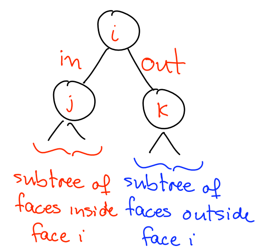
Rendering with BSP Trees: Main Idea

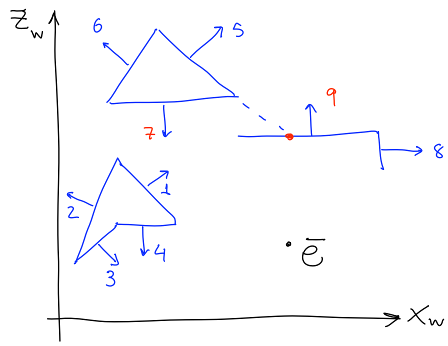
If $\bar{e}$ is outside face i (e.g. 1)

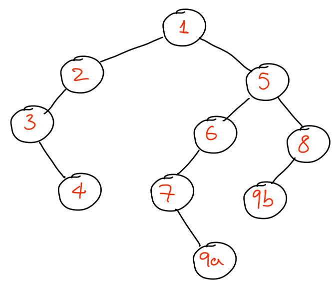
nothing inside i can occlude i

$\Rightarrow$ can be drawn before i

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

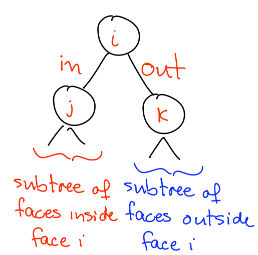


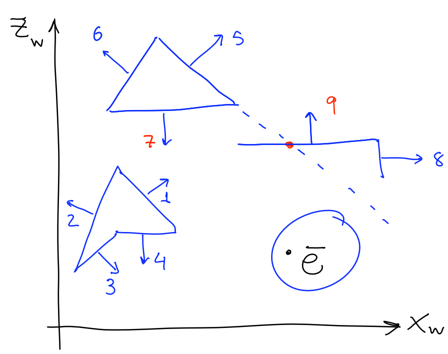


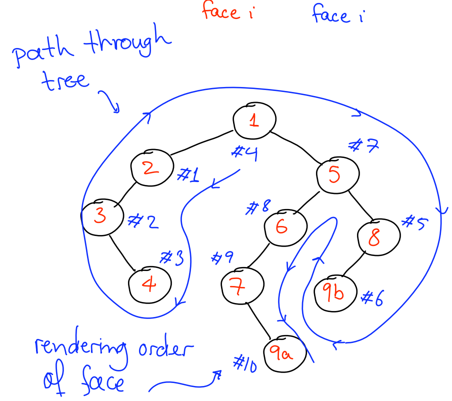


Rendering with BSP Trees: Algorithm

- If $\bar{e}$ outside face i
  - $\boxed{\text{draw everything inside } i}$ ← recursive calls to the same routine
  - draw i
  - $\boxed{\text{draw everything outside } i}$

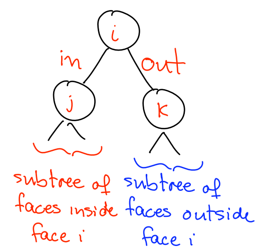


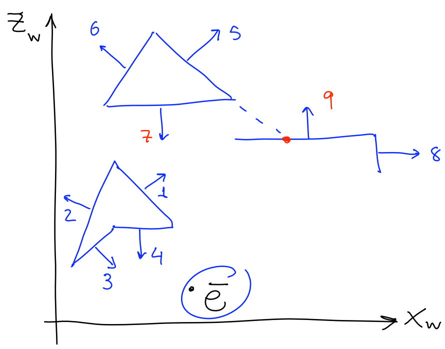


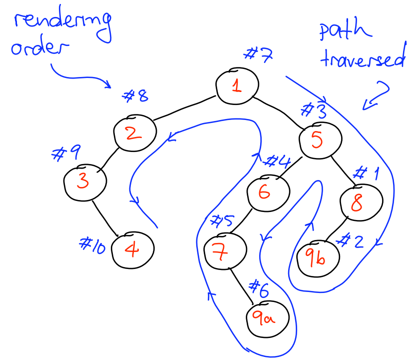


Rendering with BSP Trees: Algorithm

- If $\bar{e}$ inside face i
  - draw everything outside i
  - draw i
  - draw everything inside i







BSP Trees & Backface Culling

- If $\bar{e}$ "inside" i then i is backfacing
- Everything here shown in 2D but construction is identical in 3D
- Only complication is triangle/polygon splitting (see Ch. 8.1 in book)