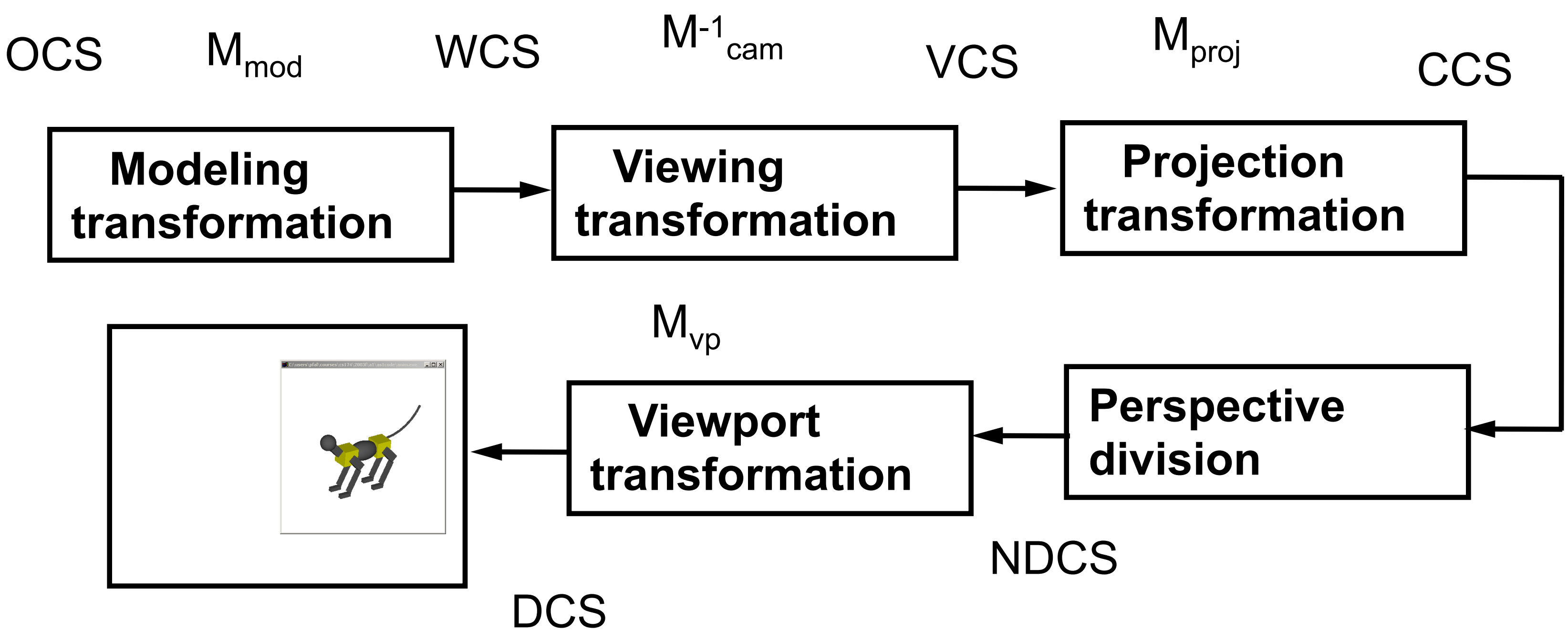
All transformations



Line Rendering Algorithm

Compute $\mathbf{M}_{mod}$

Compute $\mathbf{M}^{-1}_{cam}$

Compute $\mathbf{M}_{modelview} = \mathbf{M}^{-1}_{cam}\ \mathbf{M}_{mod}$

Compute $\mathbf{M}_{O}$

Compute $\mathbf{M}_{P}$ // disregard M_P here and below for orthographic-only case

Compute $\mathbf{M}_{proj} = \mathbf{M}_{O}\mathbf{M}_{P}$

Compute $\mathbf{M}_{VP}$

Compute $\mathbf{M} = \mathbf{M}_{VP}\ \mathbf{M}_{proj}\ \mathbf{M}_{modelview}$

for each line segment *i* between vertices P_i and Q_i do

  $P = \mathbf{M}P_i$; $Q = \mathbf{M}Q_i$

  drawline(P_x/h_P, P_y/h_P, Q_x/h_Q, Q_y/h_Q) // h_P,h_Q are the 4th coordinates of P,Q

end for

Vertex Shader

```
in vec4 vPosition;
in vec3 vNormal;
uniform mat4 projectionMatrix;
uniform mat4 modelViewMatrix ;

out vec4 fColor;


void
main()
{
    gl_Position = projectionMatrix * modelViewMatrix * vPosition;
    fColor = vec4(1.0f, 0.0f, 0.0f, 1.0f) ;
}

// Notice that perspective division happens later.
// gl_Position is in CLIPPING Coordinates
```

3D Clipping

Keep what is visible

We can clip

1. in the WCS

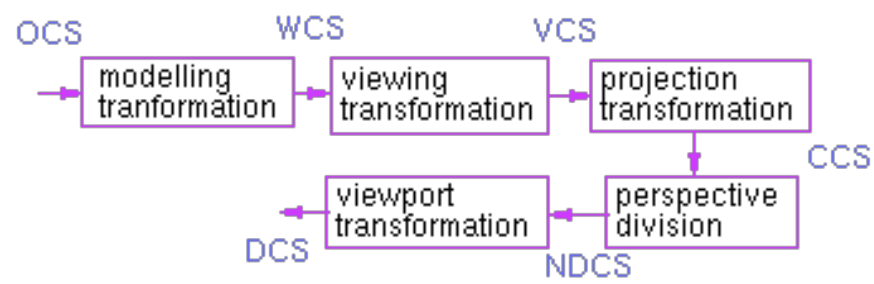
   What are the six plane equations?

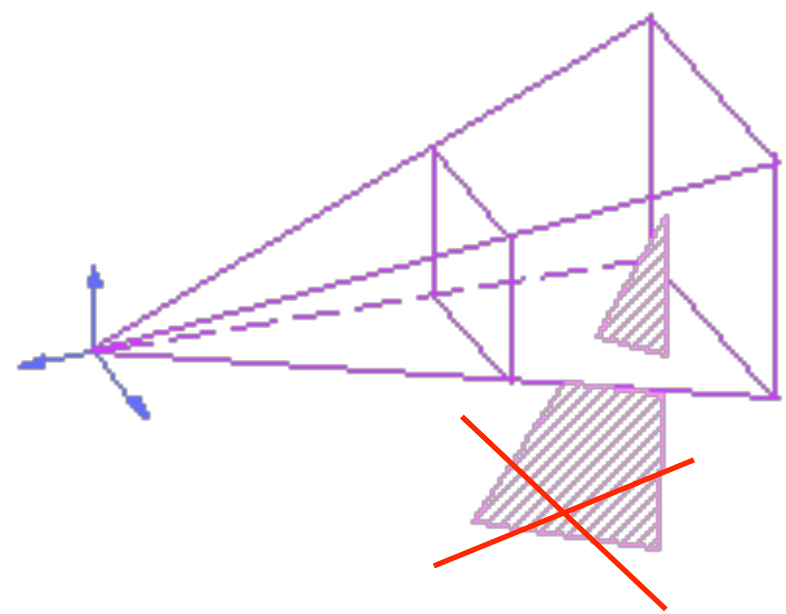
2. in the CCS

   Usually done here(!)
   Clipping in homogeneous coordinates 4D volume bounded by 3D planes
   Still simple and efficient

3. In the NDCS

   Singularity at $P_z = 0$





In any case we must clip against planes

Orthographic view volume

Planes in viewing coordinates

Normals pointing inside (arbitrary choice)

left: $x - left = 0$

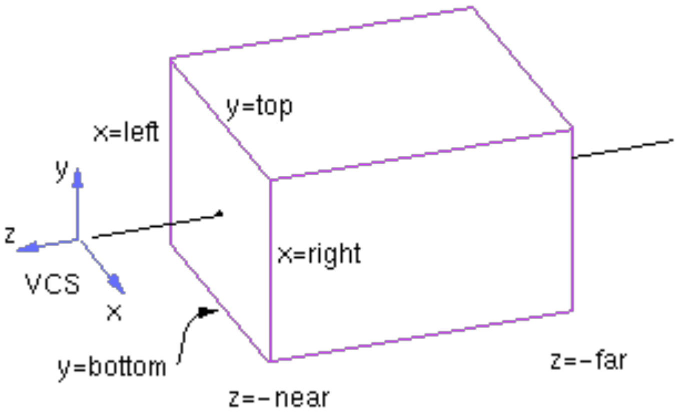
right: $-x + right = 0$

bottom: $y - bottom = 0$

top: $-y + top = 0$

front: $-z - near = 0$

back: $z + far = 0$



Perspective View volume

Planes in viewing coordinates

Normals pointing inside

left: $x + left*z/near = 0$

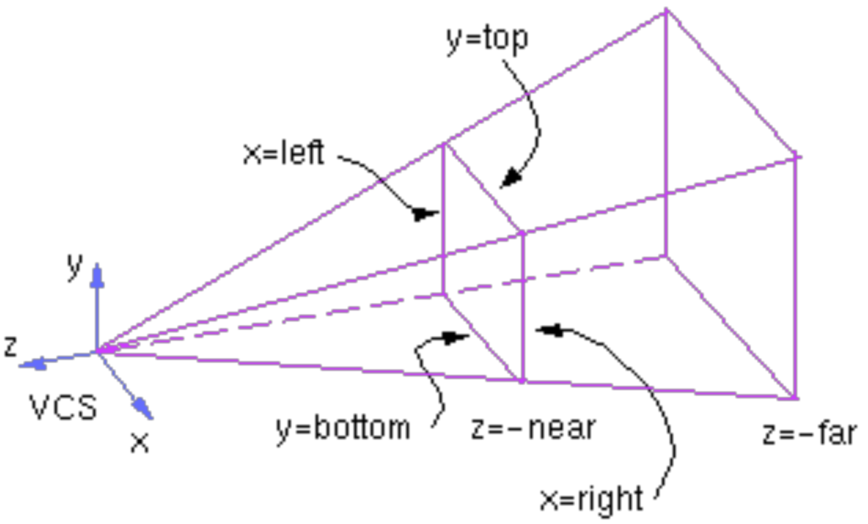
right: $-x - right*z/near = 0$

top: $-y - top*z/near = 0$

bottom: $y + bottom*z/near = 0$

front: $-z - near = 0$

back: $z + far = 0$



Clipping in NDCS (Aside)

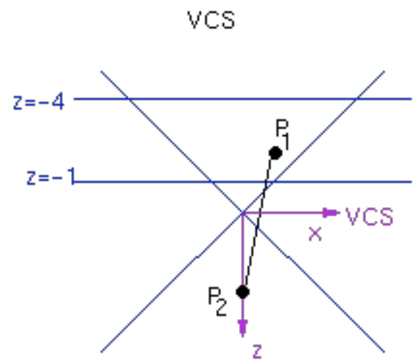
Normalized view volume

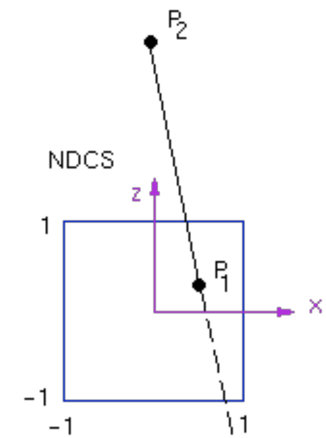
- Constant planes
- Lines in VCS lines NDCS

Problem

- Z coordinate loses its sign

	P_1	P_2
VCS	(1, 0, −2)	(0,0,2)
CCS	(1, 0, 2/3, 2)	(0,0,−6,−2)
NDCS	(1/2, 0, 1/3)	(0, 0, 3)





Clipping in CCS (Aside)

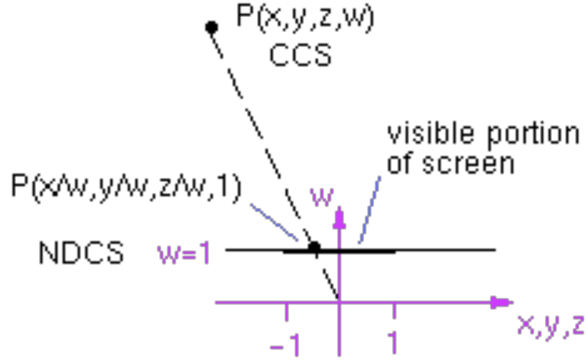
We'll define the clipping region in CCS by first looking at the clipping region in NDCS:

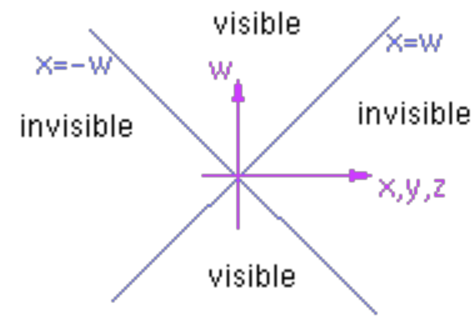
$-1 <= x/w <= 1$

This means that in CCS, we have:

$-w <= x <= w$

Similarly for y,z

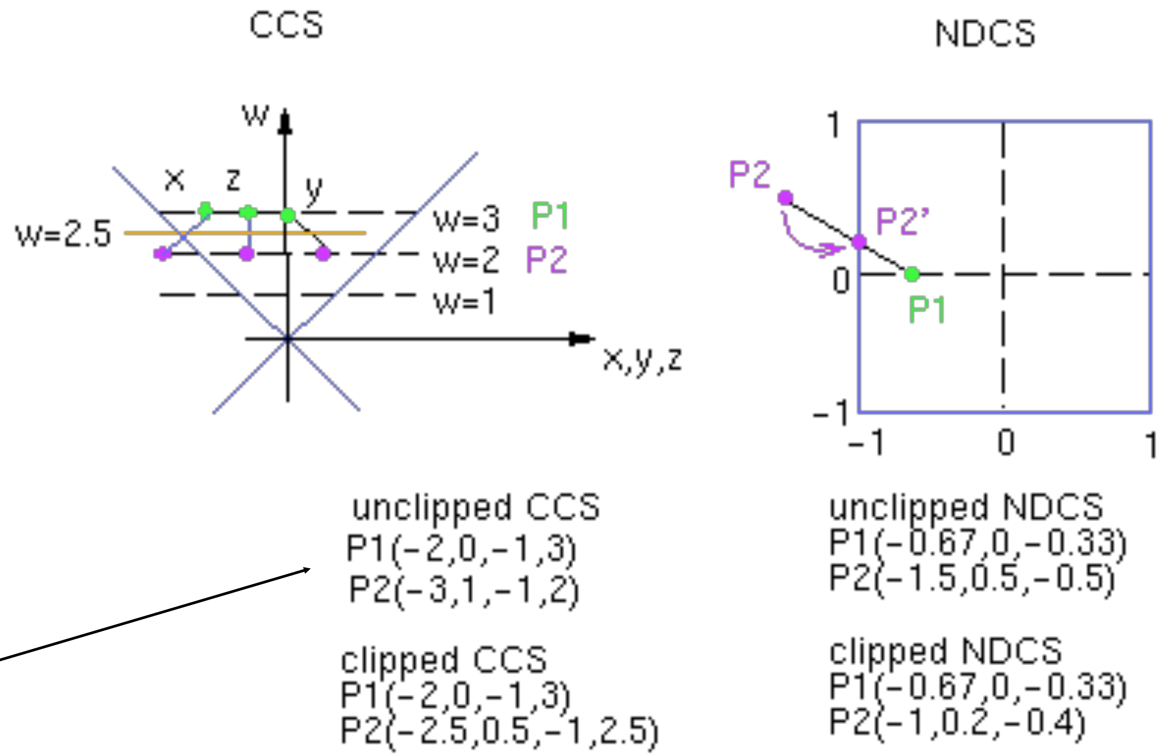




Example (Aside)

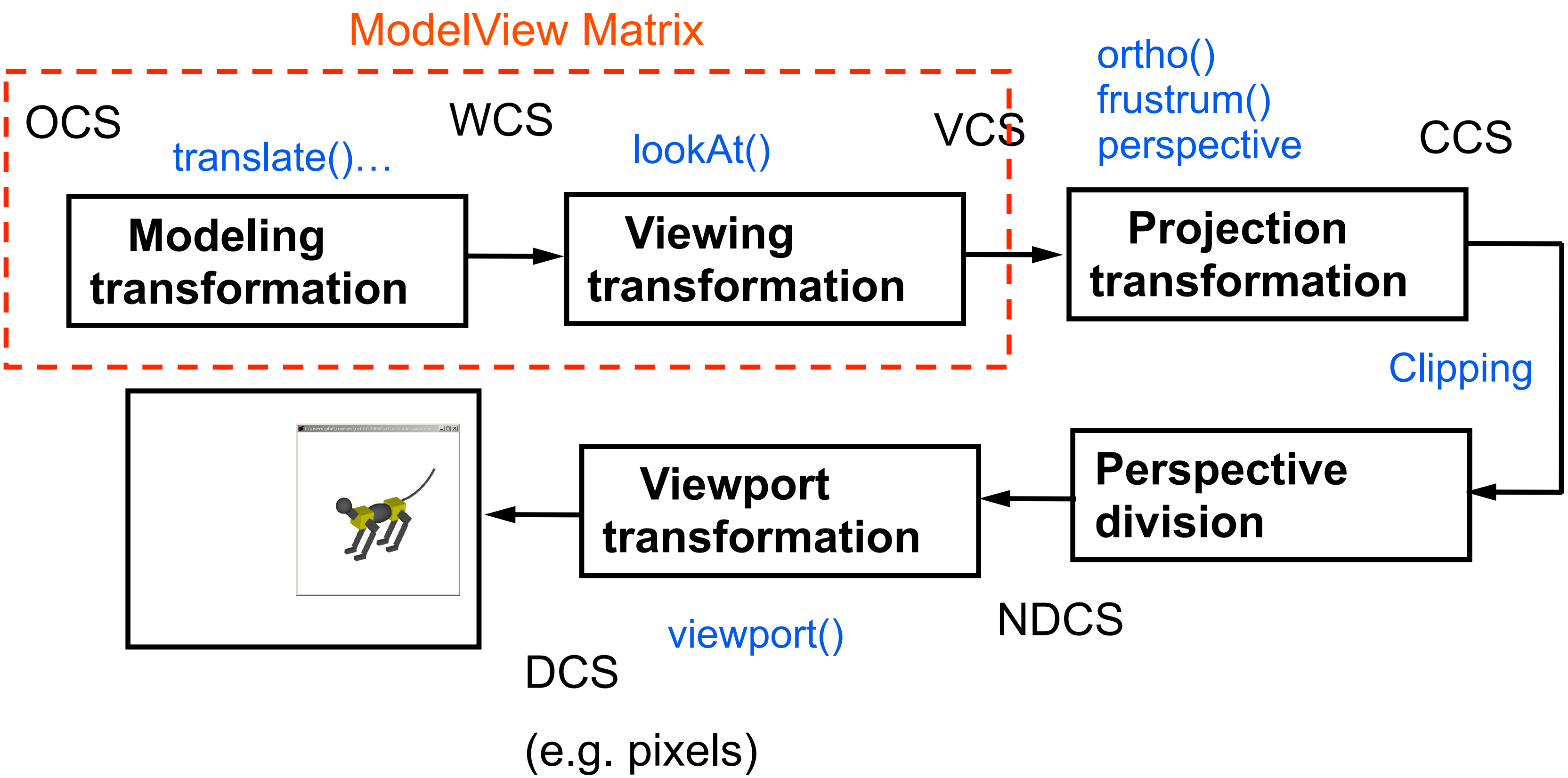
The perspective transformation creates

W = -z



Typo: they should have different z

So far our Pipeline



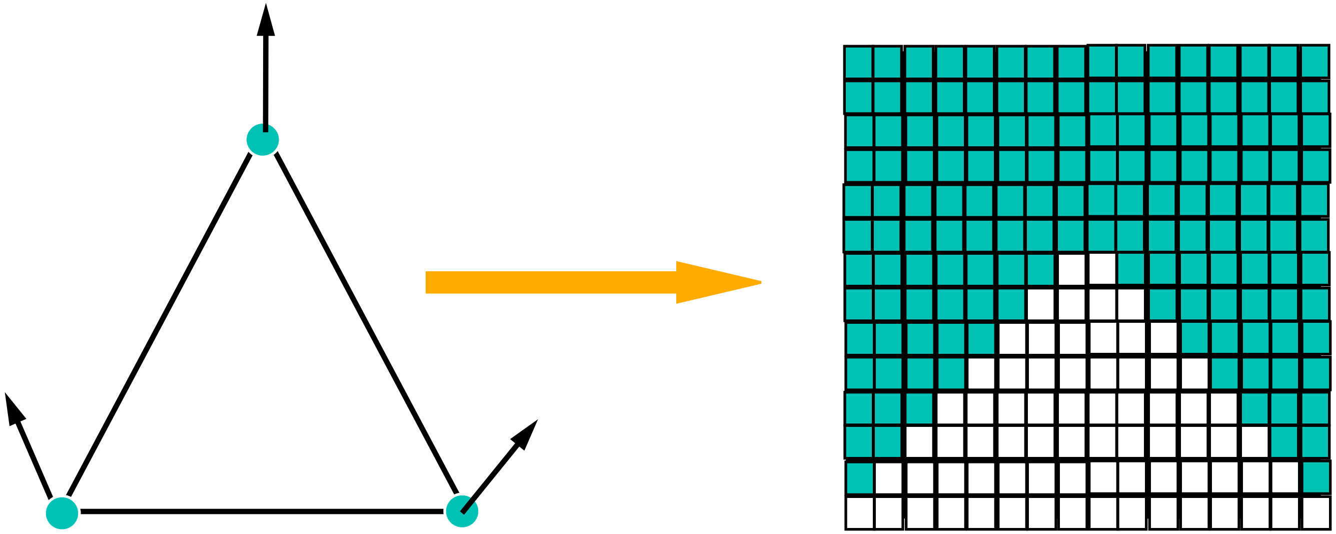
Rasterization

- The rasterizer outputs the location of fragments, i.e. pixel size screen elements. We can consider a fragment as pixel in most cases but they are not exactly equivalent.
- The programmable fragment shader computes the colors of the fragments and how they affect the corresponding pixel.
- In most systems, the graphics context is “double buffered”. We render first into the back buffer, when the image is complete, it is copied to the front buffer.

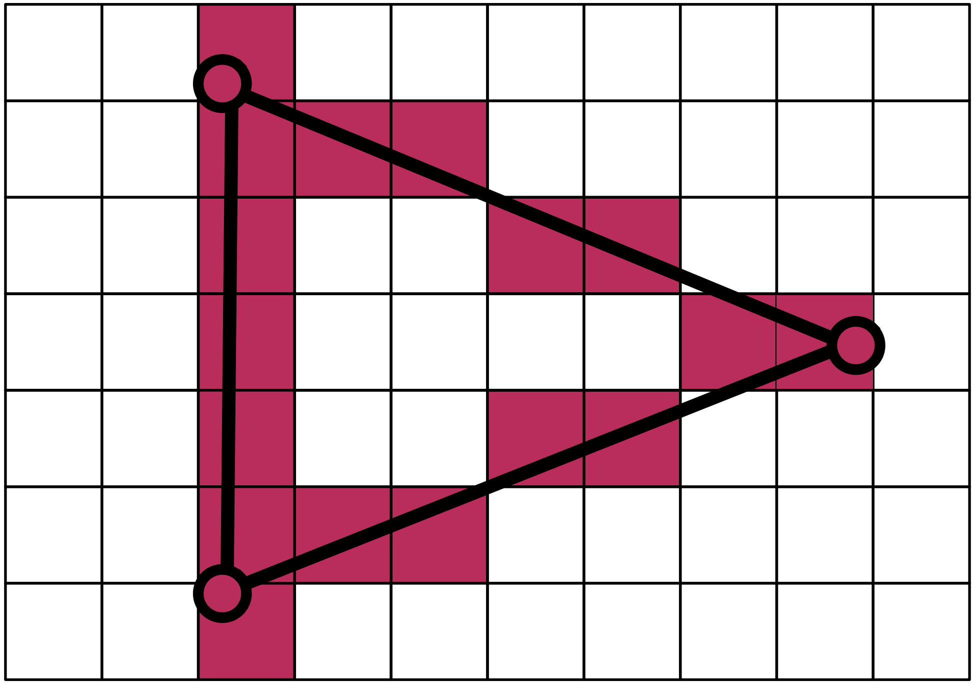
Rasterization

Primitives must be rasterized

- Mathematical form --> Set of finite size pixels



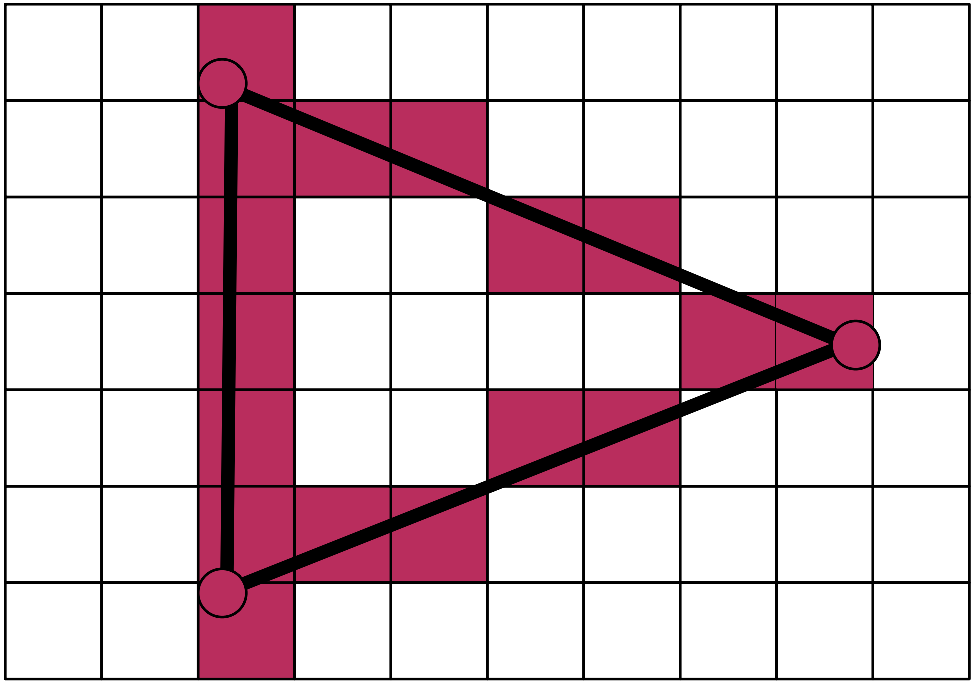
Line rasterization



Line rasterization

Desired properties

- Straight
- Pass through end points
- Smooth
- Independent of end point order
- Uniform brightness
- Brightness independent of slope
- Efficient



Straightforward Implementation

Line between two points

$$(x_1, y_1), (x_2, y_2)$$

$$y(x) = y_1 + \frac{y_1 - y_2}{x_1 - x_2}(x - x_1)$$

Straightforward Implementation

Line between two points

$$(x_1, y_1), (x_2, y_2)$$

$$y(x) = y_1 + \frac{y_1 - y_2}{x_1 - x_2}(x - x_1)$$

Exercise for those not comfortable with this math

- Start with $y = ax + b$, use $y(x_1) = y_1$ and $y(x_2)=y_2$
- Solve for a,b

Straightforward Implementation

Line between two points (slope < 45)

```
DrawLine(int x1,int y1,int x2,int y2)
    {
        float y;
        int x;
        for (x=x1; x<=x2; x=x+1) {
                y = y1 +  (x-x1)*(y2-y1)/(x2-x1)
                SetPixel(x, Round(y) );
        }
    }
```

Better Implementation

How can we improve this algorithm?

```
DrawLine(int x1,int y1,int x2,int y2)
  {
      float y;
      int x;
      for (x=x1; x<=x2; x = x + 1) {
            y = y1 +  (x-x1)*(y2-y1)/(x2-x1)
            SetPixel(x, Round(y) );
      }
  }
```

Better Implementation

```
DrawLine(int x1,int y1,int x2,int y2)
  {
      float y,m;
      int x;
      dx = x2-x1 ;
      dy = y2-y1 ;
      m = dy/ (float) dx ;
      for (x=x1; x<=x2; x = x + 1) {
              y = y1 + m*(x-x1) ;
              SetPixel(x, Round(y) );
      }
  }
```

Even Better Implementation: Incremental

```
DrawLine(int x1,int y1,int x2,int y2)
    {
        float y,m;
        int x;
        dx = x2-x1 ;
        dy = y2-y1 ;
        m = dy/ (float) dx ;
        y = y1 + 0.5 ;
        for (x=x1; x<=x2; x = x + 1) {
                SetPixel(x, Floor(y) );
                y = y + m ;
        }
    }
// y(x) = mx + d --> y(x+1) = y(x) + m
```

Midpoint algorithm (Bresenham) (ASIDE)

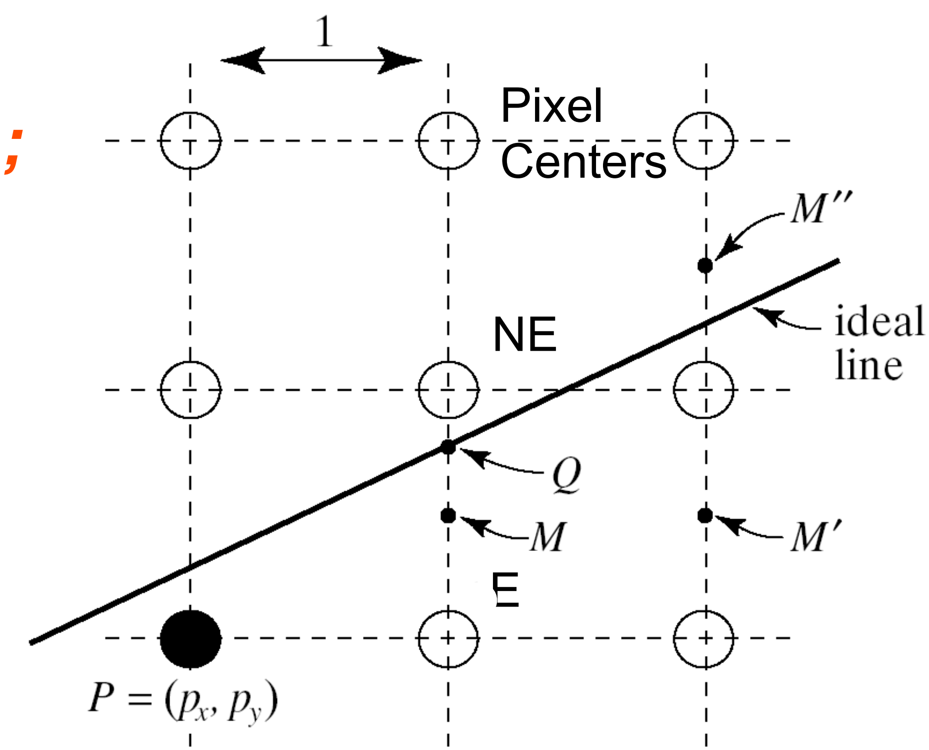
Line in the first quadrant (0<slope < 45 deg)

Implicit function:

F(x,y) = xdy - ydx + c,

dx,dy > 0 and dy/dx <= 1.0 ;

- Current choice P = (x,y).
- How do we chose next of P, P'= (x+1,y') ?



Midpoint algorithm (Bresenham) (ASIDE)

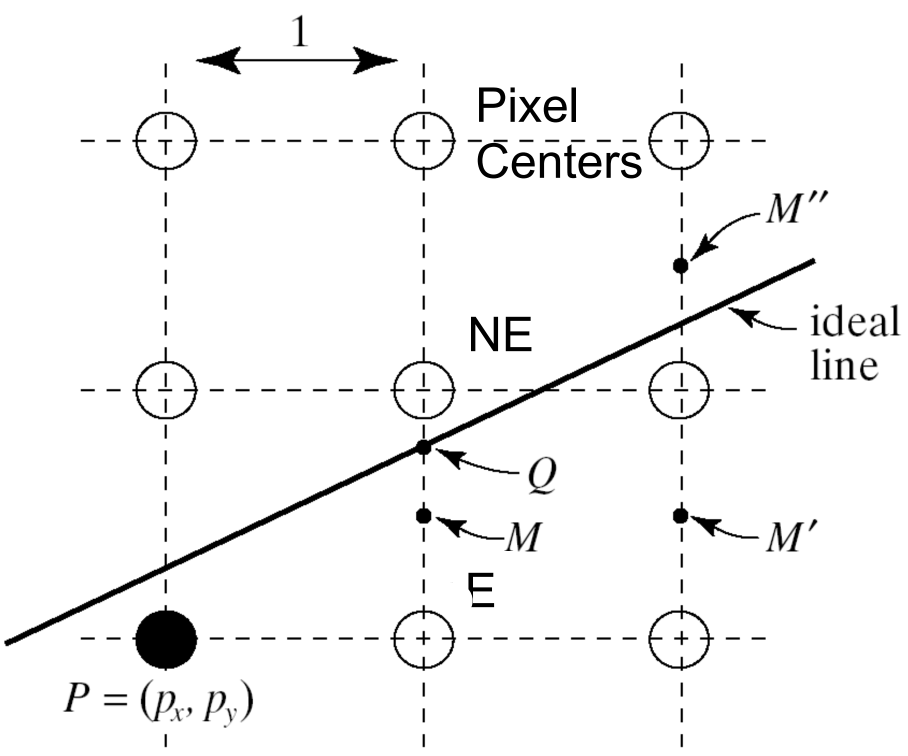
Line in the first quadrant (0<slope < 45 deg)

Implicit function:

F(x,y) = xdy - ydx + c,

dx,dy > 0 and dy/dx <= 1.0 ;

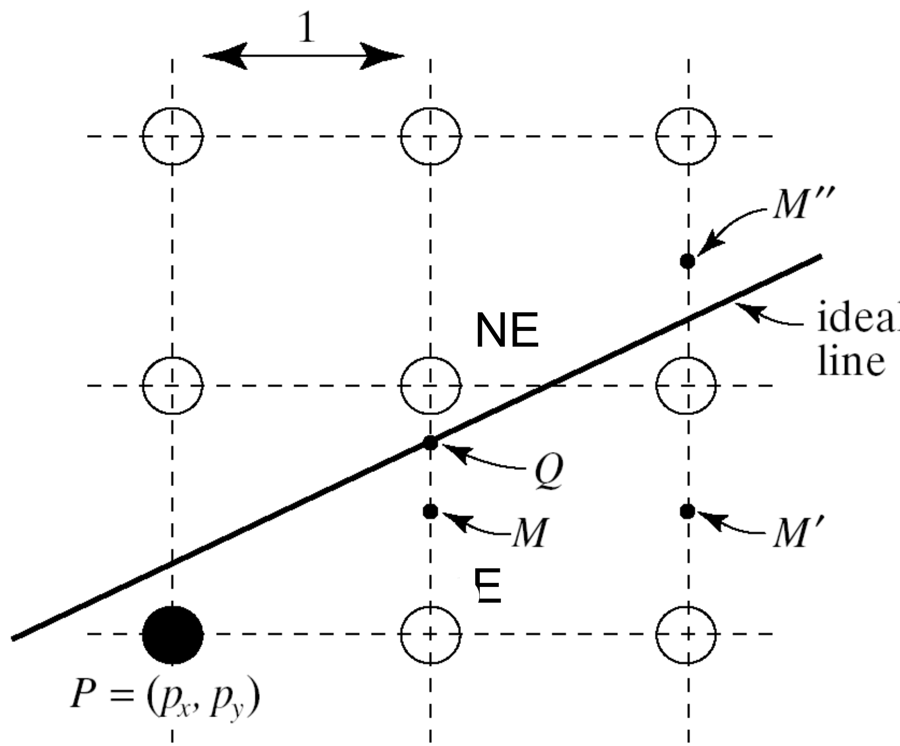
- Current choice P = (x,y).
- How do we chose next of P,
  P'= (x+1,y') ?
  If(F(M) = F(x+1,y+0.5) < 0)
  M above line so E
  else
  M below line so NE



Midpoint algorithm (Bresenham)

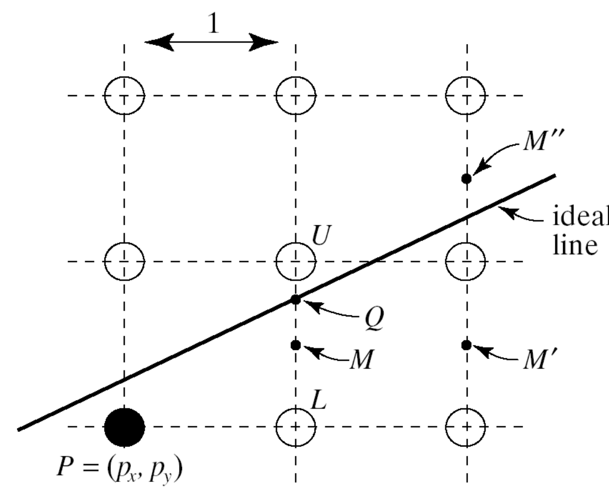
```
DrawLine(int x1, float y1, int x2, float y2, int color)
    {
        int x,y,dx,dy;

        y = Round(y1) ;
        for (x=x1; x<=x2; x++) {
      SetPixel(x, y );
      if (F(x+1,y+0.5)>0) {
            y = y + 1 ;
      }
        }
    }
```



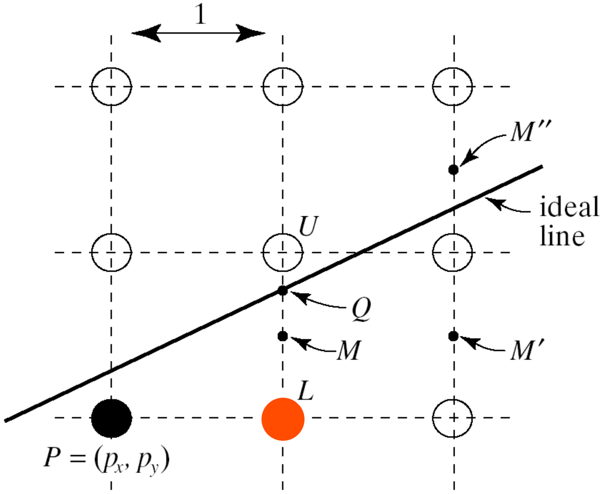
Can we compute F in a smart way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and E=(x+1,y) or NE=(x+1,y+1) accordingly.
  (Reminder: F(x,y) = xdy - ydx + c)



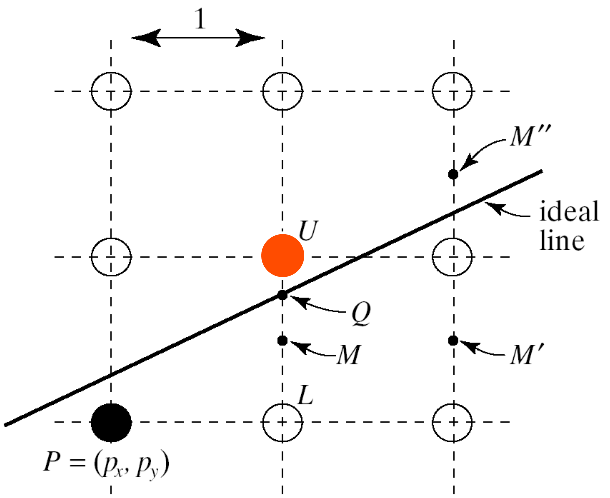
Can we compute F in a smart way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and E=(x+1,y) or NE=(x+1,y+1) accordingly.
  (Reminder: F(x,y) = xdy - ydx + c)
- If we chose E for x+1 the next criteria will be at M':
  $F(x+2,y+0.5) = [(x+1)dy + dy] - (y+0.5)*dx + c \rightarrow$
  $F(x+2,y+0.5) = F(x+1,y+0.5) + dy \rightarrow$
  $F_E = F + dy = F + dF_E$



Can we compute F in a smart way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and E=(x+1,y) or NE=(x+1,y+1) accordingly.
  (Reminder: F(x,y) = xdy - ydx + c)
- If we chose E for x+1 the next criteria will be at M':
  F(x+2,y+0.5) = (x+1)dy +dy - (y+0.5)*dx +c→
  F(x+2,y+0.5) = F(x+1,y+0.5) + dy →
  $F_E = F + dy$
- If we chose NE then the next criteria will be at M'':
  F(x+2,y+1+0.5) =
  F(x+1,y+0.5) + dy – dx →
  $F_{NE} = F + dy - dx$



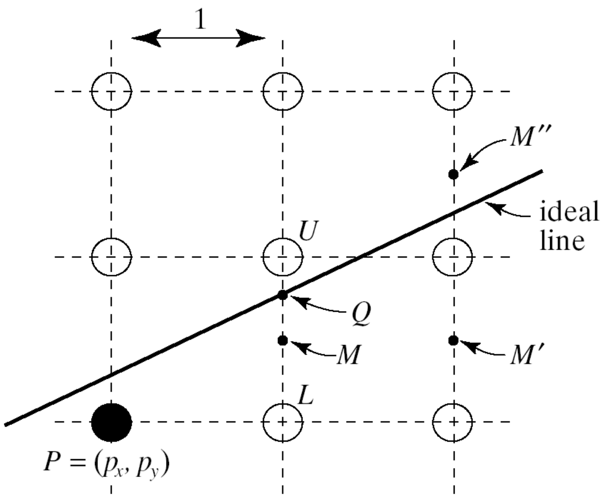
Can we compute F in a smart way?

- We are at pixel (x,y) we evaluate F at M = (x+1,y+0.5) and E=(x+1,y) or NE=(x+1,y+1) accordingly.
  (Reminder: F(x,y) = xdy - ydx + c)
- If we chose E for x+1 the next criteria will be at M':

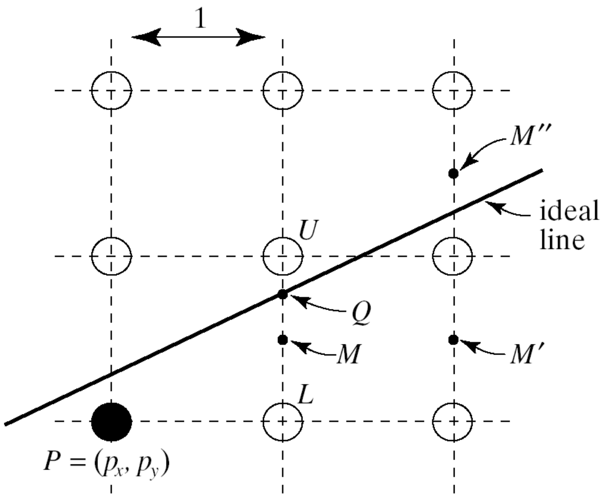
$F_E = F + dy$

- If we chose NE then the next criteria will be at M'':

$F_{NE} = F + dy - dx$



Criterion update



Update

$F_E = F + dy = F + dF_E$

$F_{NE} = F + dy - dx = F + dF_{NE}$

Starting value?

Line equation: $F(x,y) = xdy - ydx + c$

Assume line starts at pixel (x_0,y_0)

$$F_{start} = F(x_0+1, y_0+0.5) = (x_0+1)dy - (y_0+0.5)dx + c =$$

$$= (x_0dy - y_0dx + c) + dy - 0.5dx = F(x_0,y_0) + dy - 0.5dx.$$

(x_0,y_0) belongs on the line so: $F(x_0,y_0) = 0$

Therefore:

$$F_{start} = dy - 0.5dx$$

Criterion update (Integer version)

Update

$F_{start} = dy - 0.5dx$

$F_E = F + dy = F + dF_E$

$F_{NE} = F + dy - dx = F + dF_{NE}$

Everything is integer except F_{start}.

Multiply by 2 → $F_{start} = 2dy - dx$

$dF_E = 2dy$

$dF_{NE} = 2(dy-dx)$

Midpoint algorithm

```
DrawLine(int x1, float y1, int x2, float y2, int color)
    {
        int x,y,dx,dy,dE, dNE;
        dx = x2-x1 ;
        dy = y2-y1 ;
        d = 2*dy-dx ; // initialize d
        dE = 2*dy ;
        dNE = 2*(dy-dx) ;
        y = Round(y1) ;
        for (x=x1; x<=x2; x++) {
                SetPixel(x, y, color );
                if (d>0) {      // chose NE
                        d = d + dNE ;
                        y = y + 1 ;
                } else {        // chose E
                        d = d + dE ;
                }
        }
    }
```

Incremental algorithms for polynomials (ASIDE)

General form or a polynomial of degree n:

$$F(x) = a_n x^n + \underbrace{a_{n-1}x^{n-1} \cdots + a_1 x + a_0}_{Q^{n-1}(x)}, a_n \neq 0$$

or

$$F(x) = a_n x^n + Q^{n-1}(x), \quad a_n \neq 0$$

Incremental algorithms for polynomials

$$
\begin{aligned}
F(x) &= a_n x^n + Q^{n-1}(x), a_n \neq 0 \\
F(x+d) &= a_n (x+d)^n + Q^{n-1}(x+d) = a_n (x+d)^n + P^{n-1}(x) \\
&= a_n \sum_{k=0}^{n} \left(\begin{array}{c} n \\ k \end{array} \right) x^{n-k} d^k + P^{n-1}(x) \\
&= a_n \sum_{k} \left(\frac{n}{k!(n-k)!} \right) x^{n-k} d^k + P^{n-1}(x) \\
&= a_n x^n + \sum_{k=1}^{n} \left(\frac{n}{k!(n-k)!} \right) x^{n-k} d^k + P^{n-1}(x) \\
&= a_n x^n + R^{n-1}(x) + P^{n-1}(x) \\
&= a_n x^n + G^{n-1}(x)
\end{aligned}
$$

N-order differences (ASIDE)

Polynomial forms

$$F(x) = a_n x^n + Q^{n-1}(x), a_n \neq 0$$

$$F(x+d) = a_n x^n + G^{n-1}(x)$$

First order differences

$$\Delta F = F(x+d) - F(x+d) = a_n x^n + Q^{n-1}(x) - a_n x^n - G^{n-1}(x) = D_1^{n-1}(x)$$

Second order differences

$$\Delta^2 F(x) = \Delta F(x+d) - \Delta F(x) = D_2^{n-2}(x)$$

$$\vdots$$

n-order Differences

$$\Delta^n F(x) = \Delta^{n-1} F(x+d) - \Delta^{n-1} F(x) = D_n^0 = c$$

N-order difference update

Computing the polynomial incrementally from the differences

$$F(x) = a_n x^n + Q^{n-1}(x), a_n \neq 0$$
$$F(x+d) = a_n x^n + G^{n-1}(x)$$

$$F(x+d) = F(x) + \Delta^1 F(x)$$
$$\Delta F(x+d) = \Delta F(x) + \Delta^2 F(x)$$
$$\Delta^2 F(x+d) = F(x) + \Delta F(x)$$
$$\vdots$$
$$\Delta^{n-1} F(x+d) = \Delta^{n-1} F(x) + \Delta^n F(x)$$
$$\Delta^n F(x+d) = c$$

Example: y = x^2

$$
\begin{aligned}
y(x+d) &= x^2 + 2xd + d^2 = y(x) + 2xd + d^2 \\
\rightarrow \quad & y(x+d) = y(x) + \Delta y(x) \\
& where \quad \Delta y(x) = 2xd + d^2 \\
\Delta y(x+d) &= 2(x+d)d + d^2 = \Delta y(x) + 2d^2 \\
\rightarrow \quad & \Delta y(x+d) = \Delta y(x) + \Delta^2 y(x) \\
& where \quad \Delta^2 y(x) = 2d^2
\end{aligned}
$$

The incremental algorithm to compute y = x^2 (END ASIDE)

```
computePar(int d)
  {
      float y = 0 ;
      int x = 0 ;
      DY = d^2 ; // at x = 0
      DDY = 2*d^2 ;
      for( x = 0 ; x < X_MAX ; x++ ) {
              printf(“d, %f\”, x,y) ;
              y = y + DY ;
              DY = DY + DDY ;
  }
```

Polygons

Collection of points connected with lines

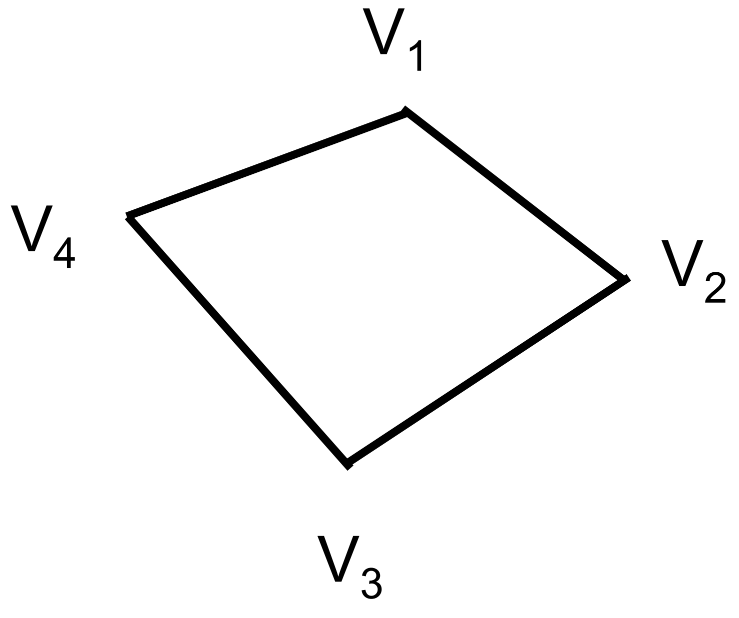
- Vertices: v1,v2,v3,v4
- Edges:

$e_1 = v_1v_2$

$e_2 = v_2v_3$

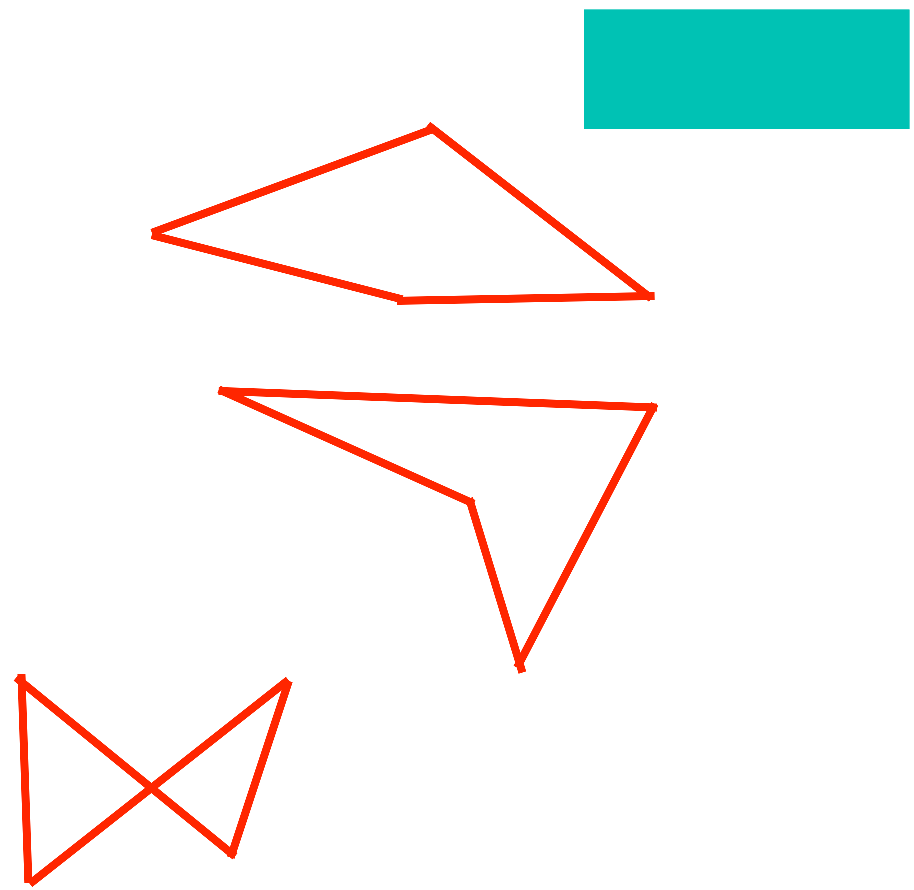
$e_3 = v_3v_4$

$e_4 = v_4v_1$



Polygons

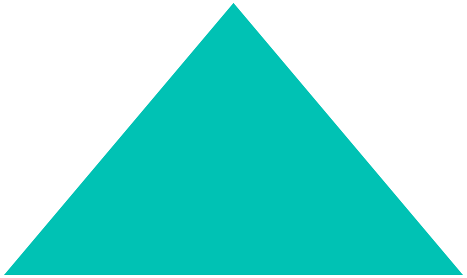
- Open / closed
- Planar / non-planar
- Filled / wireframe
- Convex / concave
- Simple / non-simple



Triangles

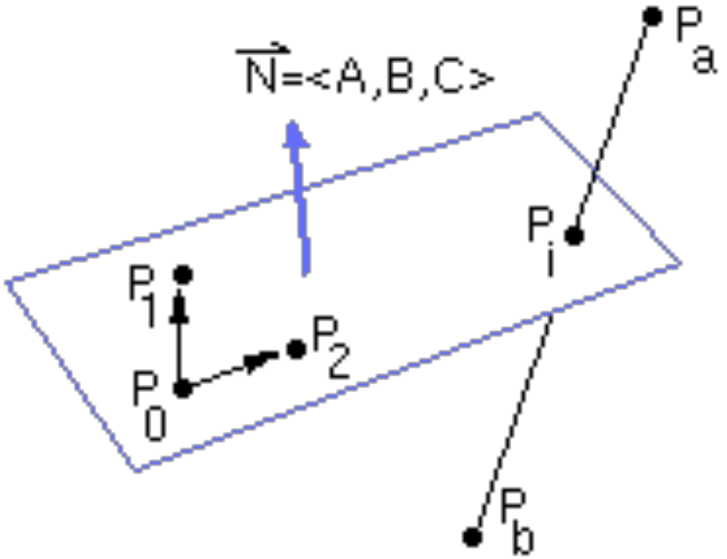
The most common primitive

- Convex
- Planar
- Simple



Reminder

Plane equations



Implicit

$$F(x, y, z) = Ax + By + Cz + D = \mathbf{N} \bullet P + D$$

Points on Plane $F(x, y, z) = 0$

Parametric

$$Plane(s,t) = P_0 + s(P_1 - P_0) + t(P_2 - P_0)$$

P_0, P_1, P_2 not colinear

or

$$Plane(s,t) = (1 - s - t)P_0 + sP_1 + tP_2$$

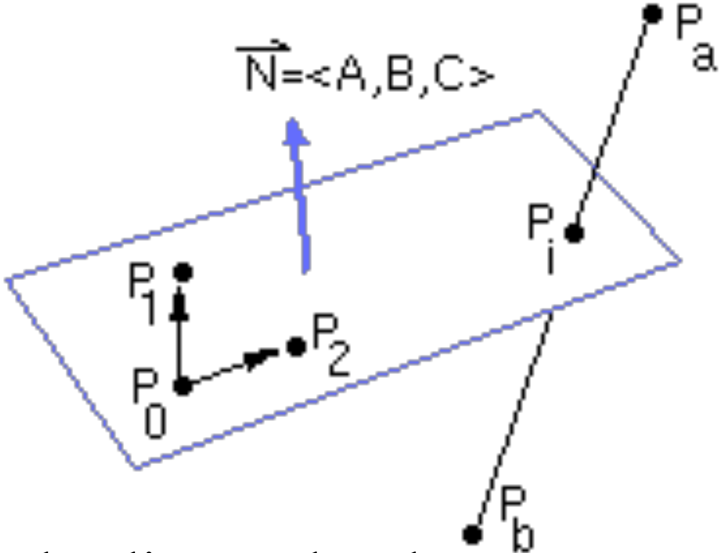
$Plane(s,t) = P_0 + sV_1 + tV_2$ where V_1, V_2 basis vectors

Explicit

$$z = -(A/C)x - (B/C)y - D/C, \quad C \neq 0$$

Point normal form

Plane equation



$$F(x,y,z) = Ax + By + Cz + D = \mathbf{N} \bullet P + D$$

Points on Plane $F(x,y,z) = 0$

Observation : Let's take an arbitrary vector u that lies on the plane which can be defined by two points e.g. P1, P2 on the plane.

$$\mathbf{u} = P2 - P1$$

$$\left.\begin{array}{l} \mathbf{N} \bullet P1 + D = 0 \\ \mathbf{N} \bullet P2 + D = 0 \end{array}\right\} \Rightarrow \mathbf{N} \bullet (P2 - P1) = 0 \Rightarrow \mathbf{N} \bullet \mathbf{u} = 0 \Rightarrow \boxed{\mathbf{N} \perp \mathbf{u}}$$

Computing point normal form from 3 Points

$F(x, y, z) = Ax + By + Cz + D = \mathbf{N} \bullet P + D$

Points on Plane $F(x, y, z) = 0$

First way:

$\mathbf{N} \bullet P0 + D = 0$

$\mathbf{N} \bullet P1 + D = 0$

$\mathbf{N} \bullet P2 + D = 0$

$|\mathbf{N}| = 1$ (arbitrary choice)

Second way:

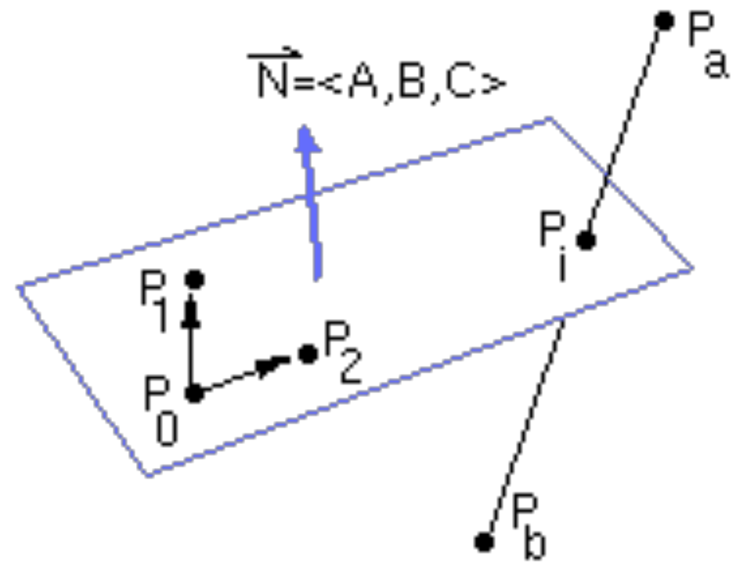
$\mathbf{N}$ is normal to F

Let's find a normal vector:

$\mathbf{N} = (P1 - P0) \times (P2 - P0)$

Compute D:

$D = -\mathbf{N} \bullet P0$



Intersection of line and plane

Implicit equation for the plane:

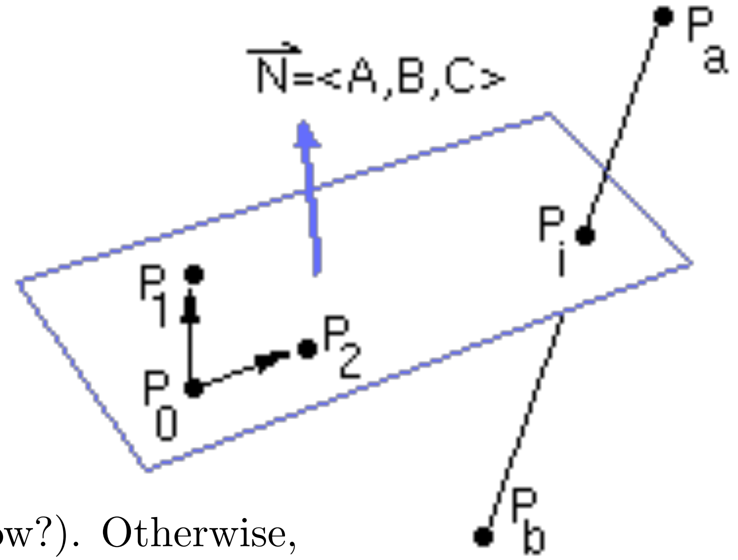
$$F(P) = \mathbf{N} \cdot P + D$$

Parametric equation for the line from P_a to P_b:

$$L(t) = P_a + t(P_b - P_a)$$

Plug $L(t)$ in $F(P)$ and solve for $t = t_i$:

$$\mathbf{N} \cdot [P_a + t_i(P_b - P_a)] = -D$$



If $N \cdot (P_a - P_b) = 0$ then zero or infinite solutions (how?). Otherwise,

$$t_i = \frac{-D - \mathbf{N} \cdot P_a}{\mathbf{N} \cdot P_b - \mathbf{N} \cdot P_a} = \frac{-F(P_a)}{F(P_b) - F(P_a)}$$

Finally, evaluate $L(t_i)$ for the intersection point P_i:

$$P_i = P_a + \frac{-F(P_a)}{F(P_b) - F(P_a)}(P_b - P_a) = \frac{P_a F(P_b) - P_b F(P_a)}{F(P_b) - F(P_a)}$$

Polygons in [Open/Web]GL

New versions ONLY TRIANGLES

Vertices have attributes (position, normal, color, etc)

- Arrays of floats: GLfloat positions[] = { ...} ;
- Indexed Arrays (Element Arrays)

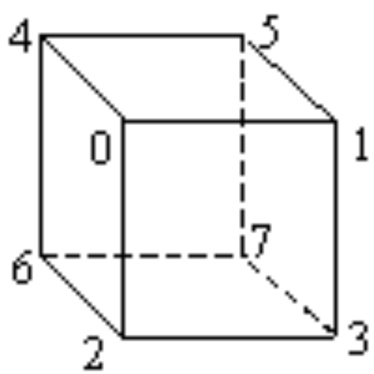
Special functionality to store and interpret the arrays

- Vertex Buffer Objects
- Vertex Array Objects

- We will see the details later

Indexed Face Sets

OpenGL element arrays



faces

#	vertex list
0	0,2,3,1
1	1,3,7,5
2	5,7,6,4
3	4,6,2,0
4	4,0,1,5
5	2,6,7,3

vertex list

#	x,y,z
0	0,1,1
1	1,1,1
2	0,0,1
3	1,0,1
4	0,1,0
5	1,1,0
6	0,0,0
7	1,0,0

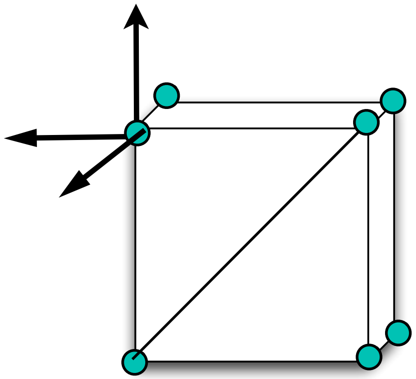
- The **ordering** of the vertices in the faces should be consistent (clock wise or counter-clockwise)
- In OpenGL it defines the orientation (front or back) of the surface (not the normal! -- confusing, I know)

Vertex attributes

Generic attributes (user defined)

Commonly defined attributes

- Position
- Normal vector
- Color
- Texture coordinates



- Position has slightly special status, in the sense that a vertex shader must output a position

Computing the normal of a polygon

One way:

$$N = (V_{n-1}-V_0)x(V_1-V_0)$$

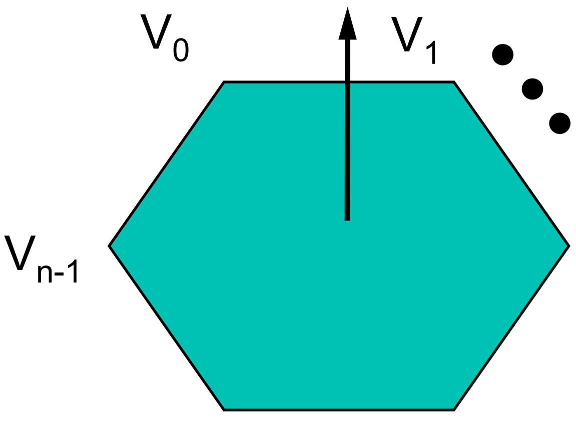
Newell's method

$$N_x = \sum_{i=0}^{n-1} (y_i - y_{next(i)})(z_i + z_{next(i)})$$

$$N_y = \sum_{i=0}^{n-1} (z_i - z_{next(i)})(x_i + x_{next(i)})$$

$$N_z = \sum_{i=0}^{n-1} (x_i - x_{next(i)})(y_i + y_{next(i)})$$

$$where \quad next(j) = (j+1) \;\; mod \;\; n$$

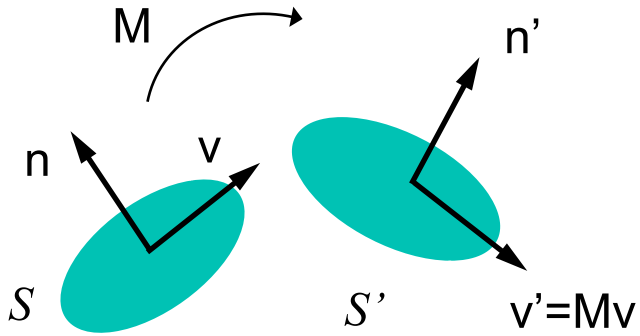


Normalize to get unit normal

Transformation of normal vectors

Given an affine transformation M

- Is the new normal the M-transformed version of the original normal, i.e. n' = Mn?

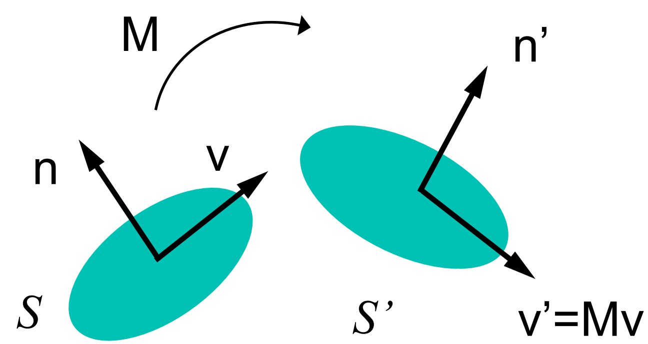


Transformation of normal vectors

Given an affine transformation M

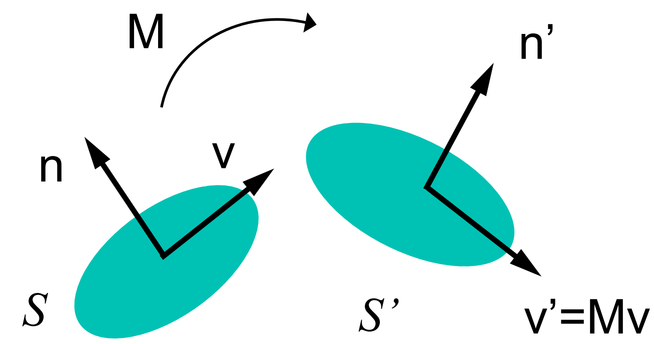
- If dot(n,v) = 0 does it mean that dot(Mn, Mv) = 0?
- In other words is the new normal the M-transformed version of the original normal?

- NOT in general
  - *Non uniform scale*
  - *Shear*



Transformation of normal vectors

$$\mathbf{n} = (n_x, n_y, n_z, 0)^T \quad normal \quad to \quad S$$
$$\mathbf{v} = (v_x, v_y, v_z, 0)^T \quad tangent \quad to \quad S$$
$$S' = MS, \quad what \quad is \quad \mathbf{n}'?$$

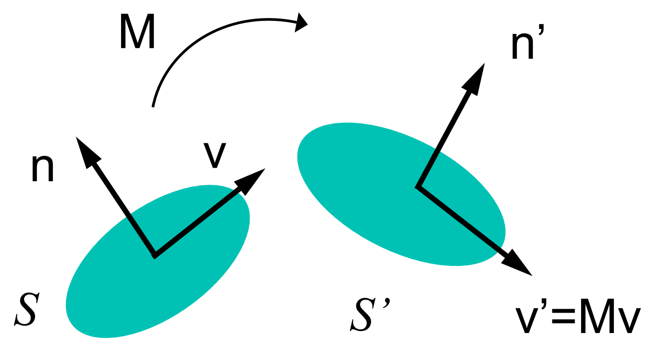


$$\mathbf{n} \cdot \mathbf{v} = \mathbf{n}^T \mathbf{v} = 0$$
$$\mathbf{n}^T \mathbf{v} = 0 \rightarrow \mathbf{n}^T I \mathbf{v} = 0 \rightarrow \mathbf{n}^T (M^{-1} M) \mathbf{v} = 0$$
$$\rightarrow (\mathbf{n}^T M^{-1})(M\mathbf{v}) = 0 \rightarrow (M^{-T}\mathbf{n})^T (M\mathbf{v}) = 0$$
$$\rightarrow (M^{-T}\mathbf{n}) \cdot (M\mathbf{v}) = 0$$

So: $n' = M^{-T} n$

The inverse transpose M^{-T} of the Modelview Matrix must be given to the shaders for transforming normals

Transformation of normal vectors



Note:

- If M is pure rotation then $M^{-T} = M$
- Vectors do not translate so we can and should consider only the top left 3x3 part of the matrix in this process

Normalization

Unit normals may not stay unit after transformation.

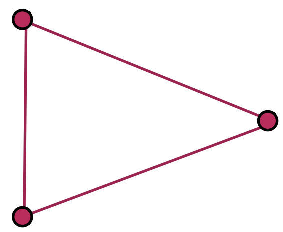
- Transformation includes scale or shear

Polygon Rasterization (for OpenGL, only triangles)

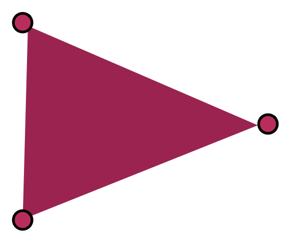
We can render triangles in three different ways

- As points: that is, only the vertices

- As lines: that is only the edges



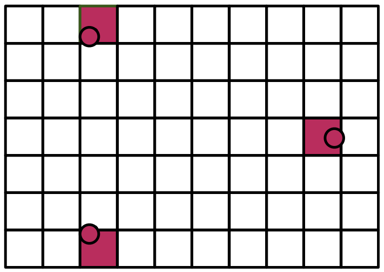
- As filled in: all interior points



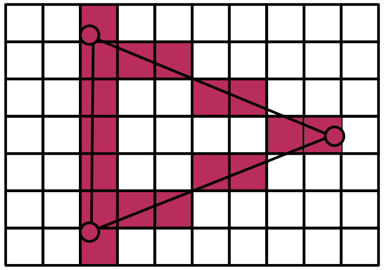
Polygon Rasterization (for OpenGL, only triangles)

We can render triangles in three different ways

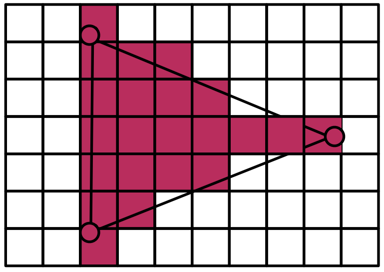
- As points: that is, only the vertices
  Rasterize produces the screen coordinates of the vertices



- As lines: that is only the edges
  Rasterize produces 3 lines using line scan conversion



- As filled in: all interior points
  ...coming up



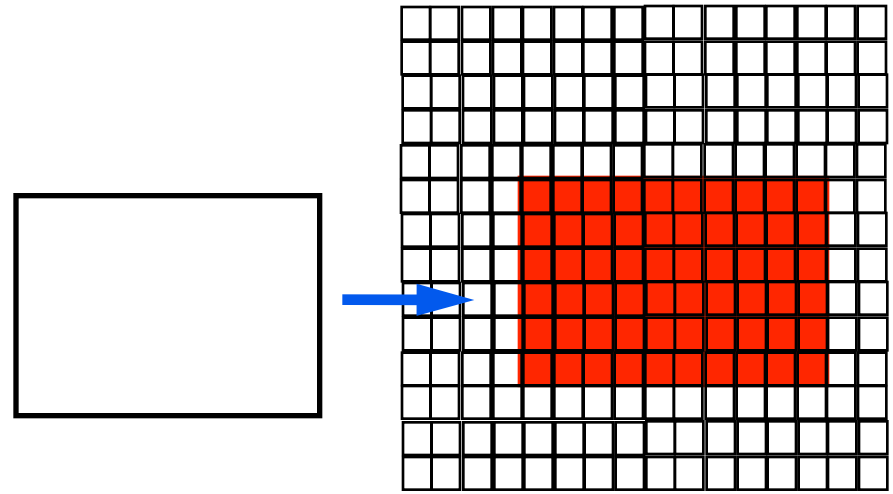
Polygon Rasterization (for OpenGL, only triangles)

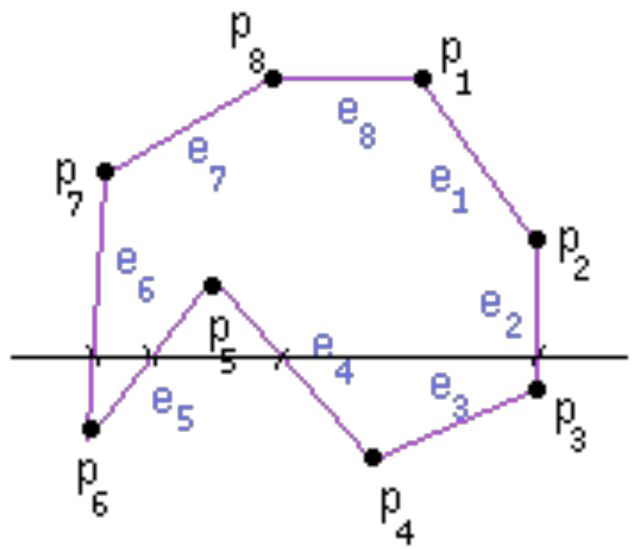
Scan conversion

shade pixels lying within a closed polygon efficiently.

Algorithm

- For each row of pixels define a *scanline* through their centers
- intersect each scanline with all edges
- sort intersections in x
- calculate parity of intersections to determine in/out
- fill the 'in' pixels

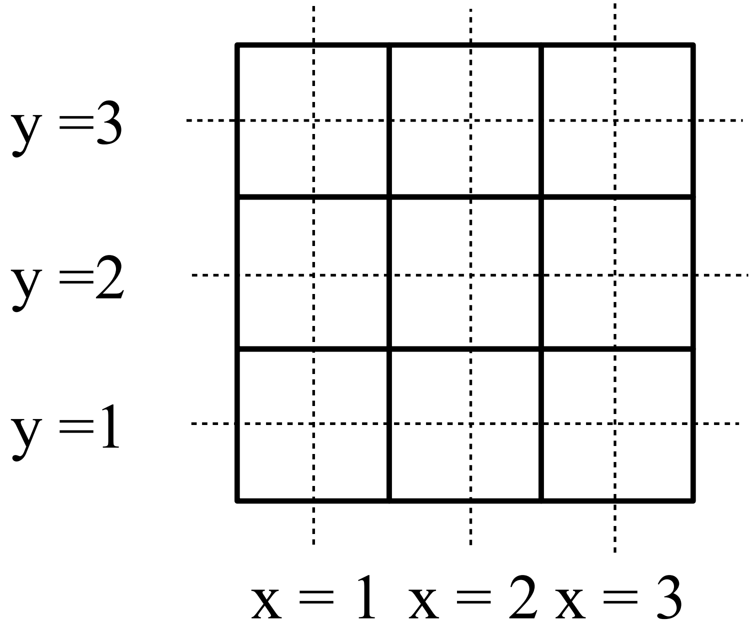




Note

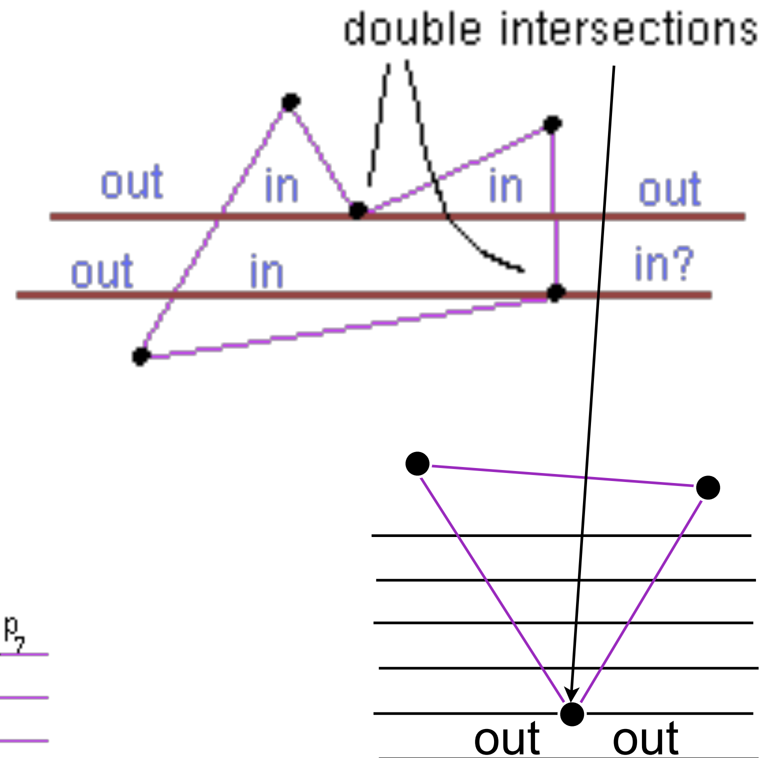
During rasterization

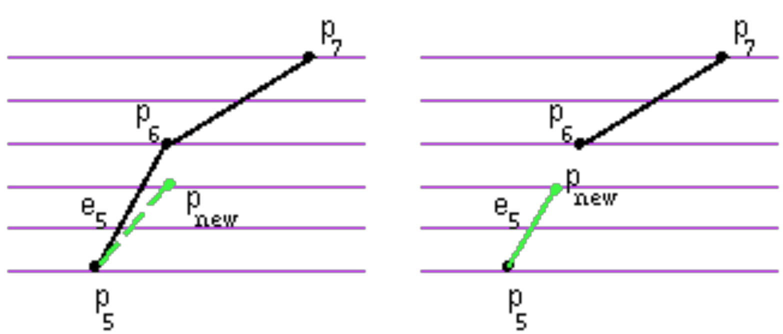
- Pixels centers are considered at integer values (n,k)
- Therefore scanlines are of the of form:
  y = k, k in (1,2,…,)
- Also, x = n, n in (1,2,...)



Special cases (ASIDE)

- Horizontal edges can be excluded
- Vertices lying on scanlines
  - *Change in sign of y_i-y_{i+1}: count twice*
  - *No change: shorten edge by one scanline*





Efficiency?

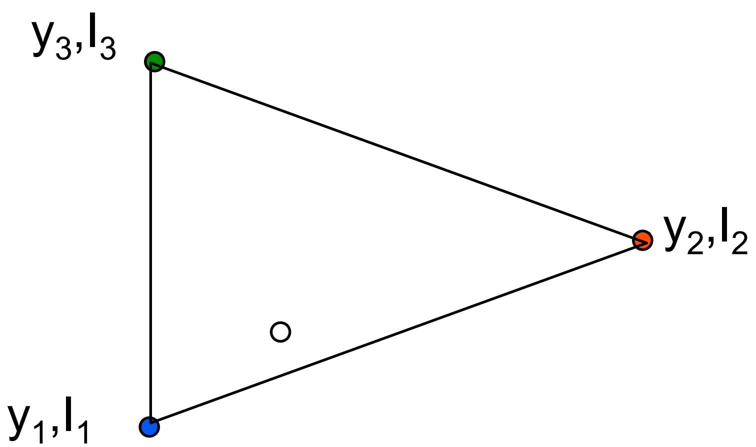
Many intersection tests can be eliminated by taking advantage of coherence between adjacent scanlines

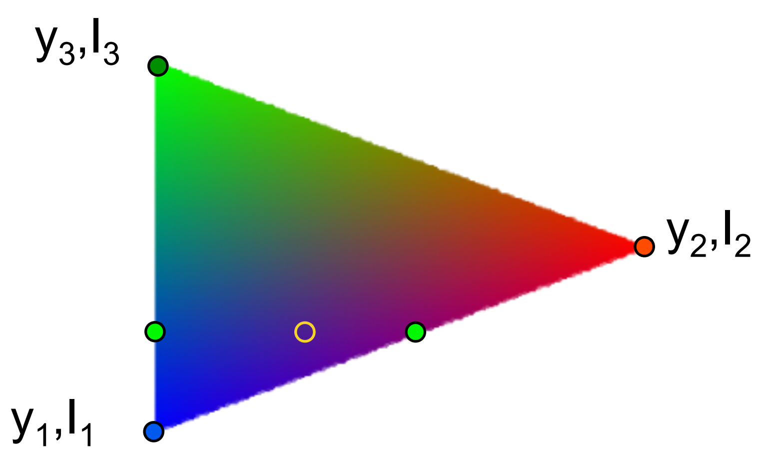
- Edges that intersect scanline y are likely to intersect y+1
- x changes predictably from scanline y to y+1

$y = mx+a \rightarrow x = 1/m(y-a) \rightarrow x(y+1) = x(y) + 1/m$

Attributes of Interior pixels?

- We only have attributes for vertices
- What about the other points of the triangle
- E.g. Colors:

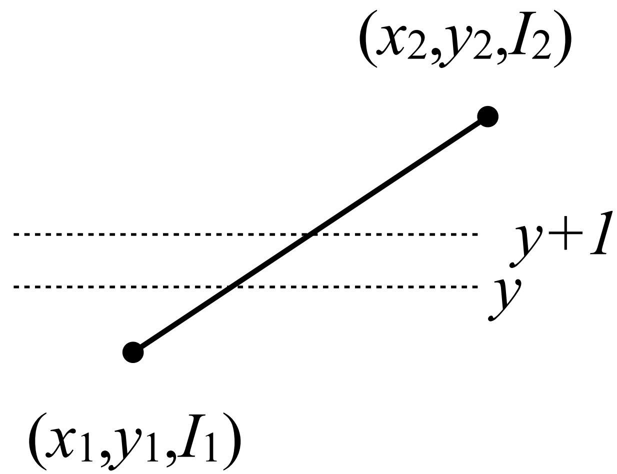




- Most common approach: interpolation

Interpolating information along a 2D line

Forms and relationships



- Parametric form

$$x = (1-t)x_1 + tx_2, \quad t \in [0,1]$$
$$y = (1-t)y_1 + ty_2,$$
$$I = (1-t)I_1 + tI_2.$$

- Using ratios (for $y_1 \neq y_2$)

$$\frac{I(t_a) - I(t_b)}{y(t_a) - y(t_b)} = \frac{I_1 - I_2}{y_1 - y_2}, \quad \forall t_a, t_b : t_a \neq t_b$$

- Choosing t_a and t_b we can get efficient incremental versions:

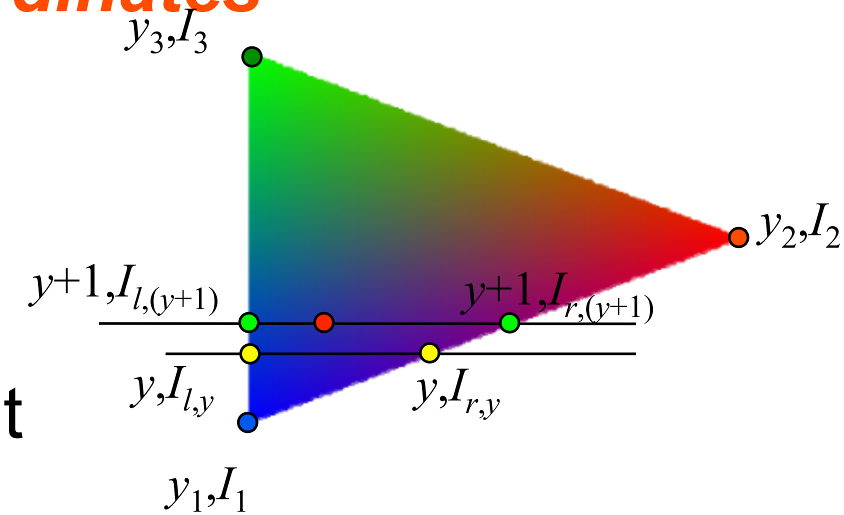
$$\frac{I_{y+1} - I_y}{(y+1) - y} = \frac{I_1 - I_2}{y_1 - y_2} \rightarrow I_{y+1} = I_y + \frac{I_1 - I_2}{y_1 - y_2}$$

constant along line

- Similarly for x (when moving along scaline)

Bilinear Interpolation of Information during scanconversion

Color, Normal, Texture coordinates



- Two levels of interpolation
- Along edges (green)
- Along scan-line(red)
- Remember pixel centres at integer values
- First scan-line y = 0, second y = 1,…
- Pixels along scaline y: (x_1,y), (x_1+1,y), (x_1+2, y), …..
- Incremental approach on both levels

Bilinear Interpolation of Information during scanconversion

Two levels of interpolation

Right edge $(1,2)$

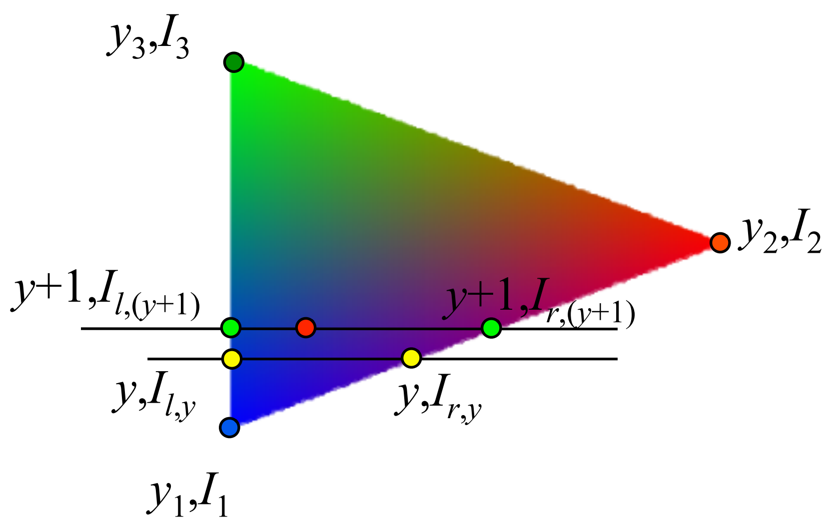
$$\frac{I_{r,y+1} - I_{r,y}}{(y+1) - y} = \frac{I_1 - I_2}{y_1 - y_2} \rightarrow I_{r,y+1} = I_{r,y} + \frac{I_1 - I_2}{y_1 - y_2}$$

Left edge $(1,3)$

$$\frac{I_{l,y+1} - I_{l,y}}{(y+1) - y} = \frac{I_1 - I_3}{y_1 - y_3} \rightarrow I_{l,y+1} = I_{l,y} + \frac{I_1 - I_3}{y_1 - y_3}$$

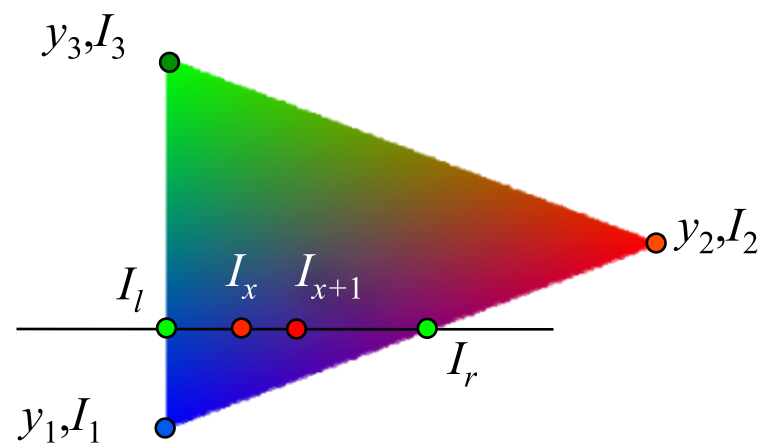
Along a scan line

$$\frac{I_{x+1} - I_x}{(x+1) - x} = \frac{I_r - I_l}{x_r - x_l} \rightarrow I_{x+1} = I_x + \frac{I_r - I_l}{x_r - x_l}$$



Bilinear Interpolation of Information during scanconversion

Color, Normal, Texture coordinates



Right edge $(1,2)$

$$\frac{I_{r,y+1} - I_{r,y}}{(y+1) - y} = \frac{I_1 - I_2}{y_1 - y_2} \rightarrow I_{r,y+1} = I_{r,y} + \frac{I_1 - I_2}{y_1 - y_2}$$

Left edge $(1,3)$

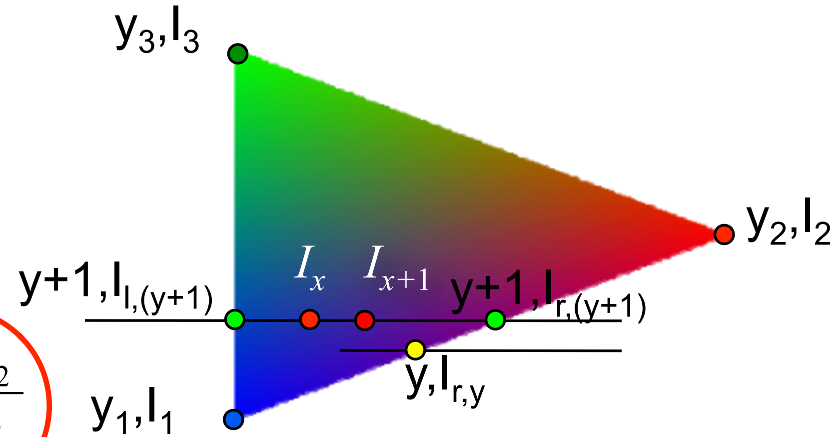
$$\frac{I_{l,y+1} - I_{l,y}}{(y+1) - y} = \frac{I_1 - I_2}{y_1 - y_2} \rightarrow I_{l,y+1} = I_{l,y} + \frac{I_1 - I_2}{y_1 - y_2}$$

Along a scan line

$$\frac{I_{x+1} - I_x}{(x+1) - x} = \frac{I_r - I_l}{x_r - x_l} \rightarrow I_{x+1} = I_x + \frac{I_r - I_l}{x_r - x_l}$$

Incremental interpolation during scanconversion

Color, Normal, Texture coordinates



Right edge (1,2):

$$\frac{I_{r,(y+1)} - I_{r,y}}{(y+1) - y} = \frac{I_1 - I_2}{y_1 - y_2} \Rightarrow I_{r,(y+1)} = I_{r,y} + \frac{I_1 - I_2}{y_1 - y_2}$$

Left Edge (1,3):

$$\frac{I_{l,(y+1)} - I_{l,y}}{(y+1) - y} = \frac{I_1 - I_3}{y_1 - y_3} \Rightarrow I_{l,(y+1)} = I_{l,y} + \frac{I_1 - I_3}{y_1 - y_3}$$

Along scanline:

$$\frac{I_{(x+1)} - I_x}{(x+1) - x} = \frac{I_r - I_l}{x_r - x_l} \Rightarrow I_{r,(y+1)} = I_{r,y} + \frac{I_r - I_l}{x_r - x_l}$$

Constant along the line

How does WebGL support this?

Vertex shader:

- out vec4 vcolor ;

Fragment shader

- in vec4 vcolor ;

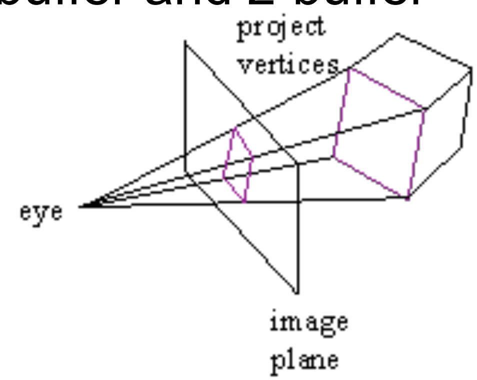
Rasterizer knows to interpolate

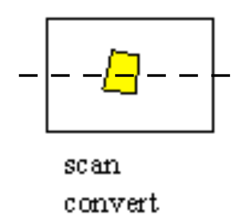
- vcolor

Z-buffer algorithm

Although part of the positions the z-value can be viewed as a special attribute of a vertex

- for each polygon in model
- project vertices of polygon onto viewing plane
- for each pixel inside the projected polygon
- calculate pixel colour
- calculate pixel z-value
- compare pixel z-value to value stored for pixel in z-buffer
- if pixel is closer, draw it in frame-buffer and z-buffer
- end
- end





Depth Test

- `gl.enable(gl.DEPTH_TEST) ;`
- `gl.disable(gl.DEPTH_TEST) ;`
- `void gl.depthFunc(func);`

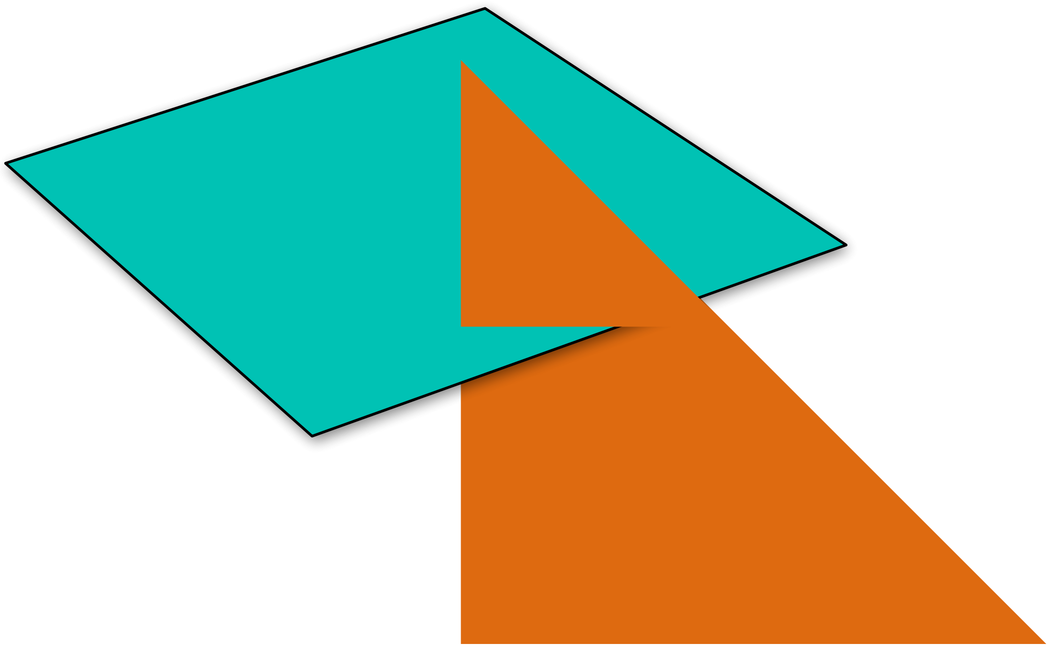
- `func` specifies the depth comparison function:
  `gl.NEVER, gl.LESS,`
  `gl.EQUAL,gl.LEQUAL,gl.GREATER,gl.NOTEQUAL,gl.GEQUAL,gl.ALWAYS`
- The default value is gl.LESS: test passes if the incoming depth value is less than the stored one.

- Size of the z-buffer: canvas.height * canvas.width floats

Z-fighting

Common problem with depth test based systems

- Intersections
- Overlaps
- Rendering highlights on top of geometry



Polygon Offset

- `gl.enable(gl.POLYGON_OFFSET_FILL) ;`
  `gl.enable(gl.POLYGON_OFFSET_LINE) ;`
  `gl.enable(gl.POLYGON_OFFSET_POINT) ;`

- `void gl.polygonOffset(GLfloat factor, GLfloat units);`

- Offsetting the z-values before depth comparison
- Useful for rendering hidden-line images, for applying decals to surfaces, and for rendering solids with highlighted edges

- see online manual

Depth Value Functions (aside)

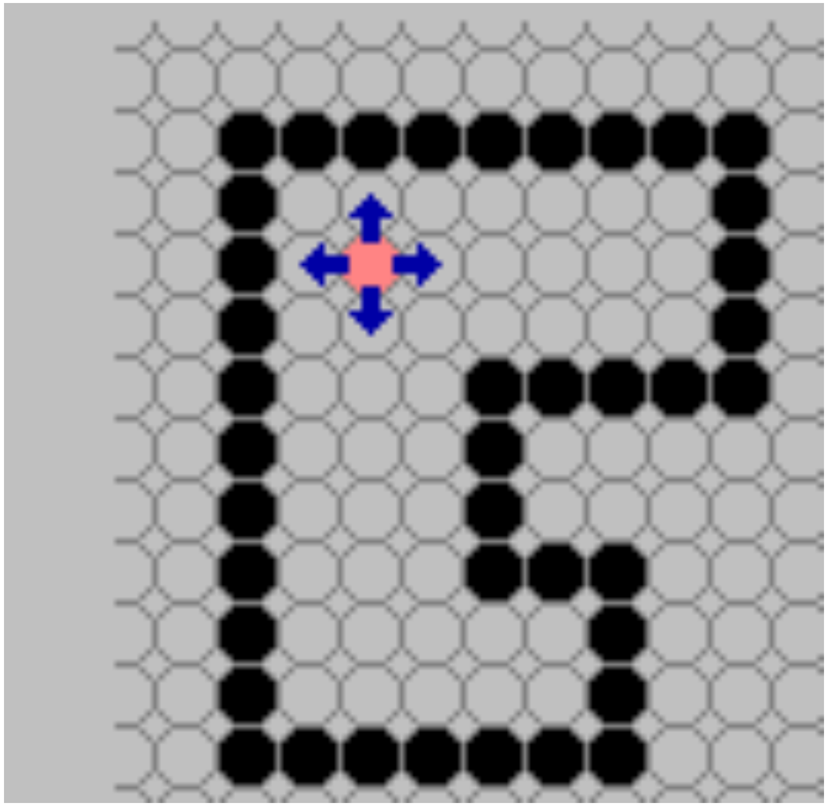
- `void gl.depthRangef(gl.FLOAT nearVal,  gl.FLOAT farVal);`

- After clipping and division by w, depth coordinates range from -1 to 1, corresponding to the near and far clipping planes. gl.depthRange specifies a linear mapping of the normalized depth coordinates in this range to window depth coordinates. Regardless of the actual depth buffer implementation, window coordinate depth values are treated as though they range from 0 through 1 (like color components). Thus, the values accepted by gl.depthRange are both clamped to this range before they are accepted.
- The setting of (0,1) maps the near plane to 0 and the far plane to 1. With this mapping, the depth buffer range is fully utilized.

Pixel Region filling algorithms

Scan convert boundary

Fill in regions



2D paint programs

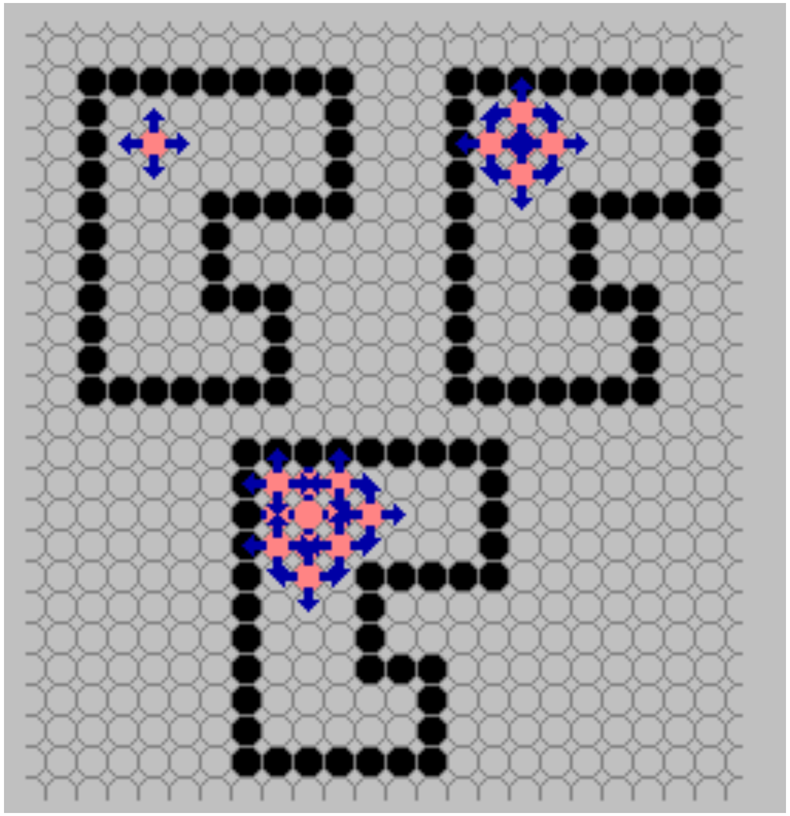
http://www.cs.unc.edu/~mcmillan/comp136/Lecture8/areaFills.html

BoundaryFill

```
boundaryFill(int x, int y, int fill, int boundary) {
        if ((x < 0) || (x >= raster.width)) return;
        if ((y < 0) || (y >= raster.height)) return;
        int current = raster.getPixel(x, y);
        if ((current != boundary) & (current != fill)) {
            raster.setPixel(fill, x, y);
    boundaryFill(x+1, y, fill,boundary);
            boundaryFill(x, y+1, fill, boundary);
            boundaryFill(x-1, y, fill, boundary);
            boundaryFill(x, y-1, fill, boundary);
        }
}
```

Flood Fill

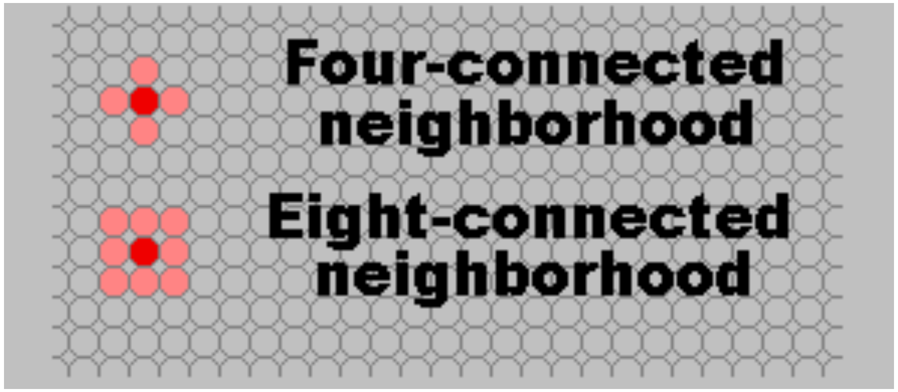
```
public void floodFill(int x, int y, int fill, int old)
{
      if ((x < 0) || (x >= raster.width)) return;
      if ((y < 0) || (y >= raster.height))  return;
       if (raster.getPixel(x, y) == old) {
          raster.setPixel(fill, x, y);
          floodFill(x+1, y, fill, old);
          floodFill(x, y+1, fill, old);
          floodFill(x-1, y, fill, old);
          floodFill(x, y-1, fill, old);
       }
}
```



Adjacency

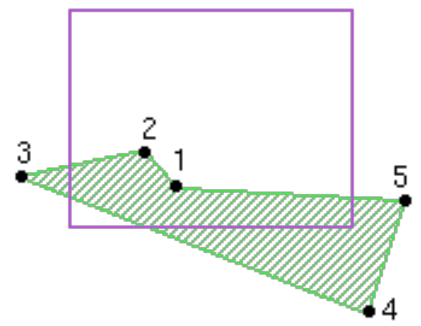
4-connected

8 connected



Polygon clipping (2D): Aside

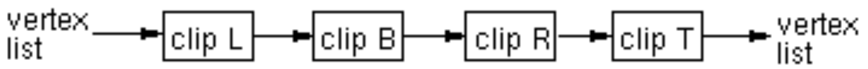
Sutherland-Hodgeman

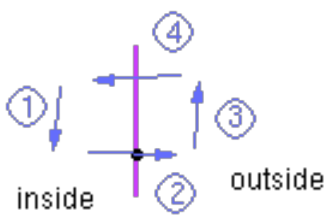


for each side of clipping window

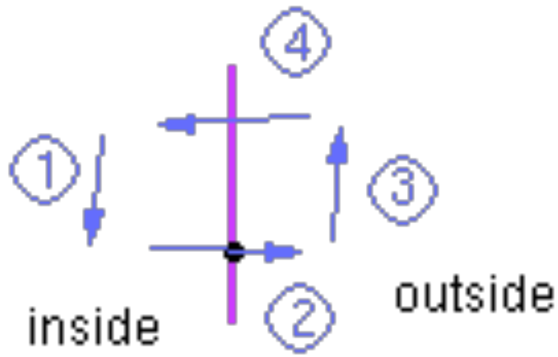
for each edge of polygon

output points based upon the following table

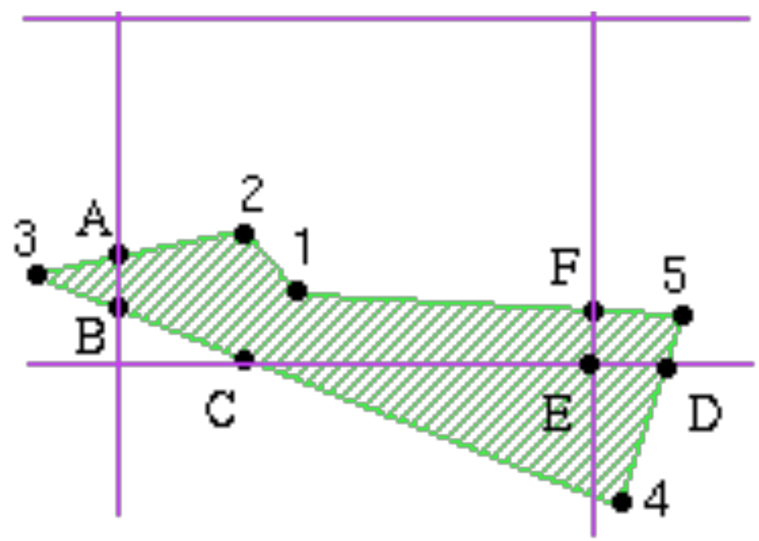




case #	first point	second point	output point(s)
1	inside	inside	second point
2	inside	outside	intersection point
3	outside	outside	none
4	outside	inside	intersection point and second point



case #	first point	second point	output point(s)
1	inside	inside	second point
2	inside	outside	intersection point
3	outside	outside	none
4	outside	inside	intersection point and second point



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: (same)