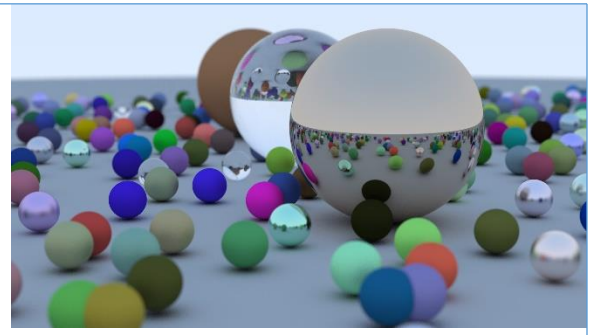
Comp4422



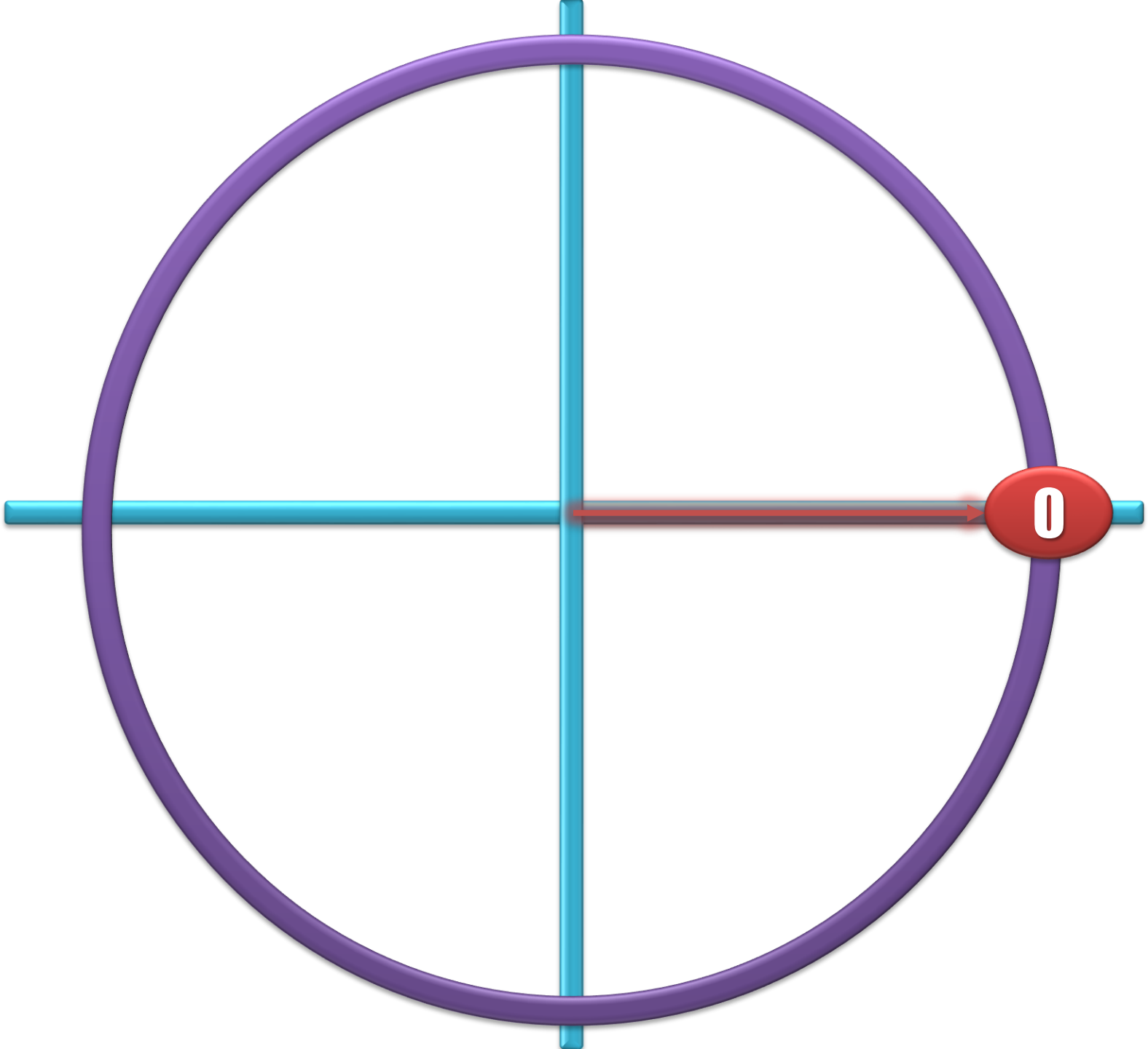
Computer Graphics

Lecture 03:

Viewing

3D Viewing

- Getting the geometry on the screen
- Pin-hole camera model
- Viewing transformations
- Clipping


0

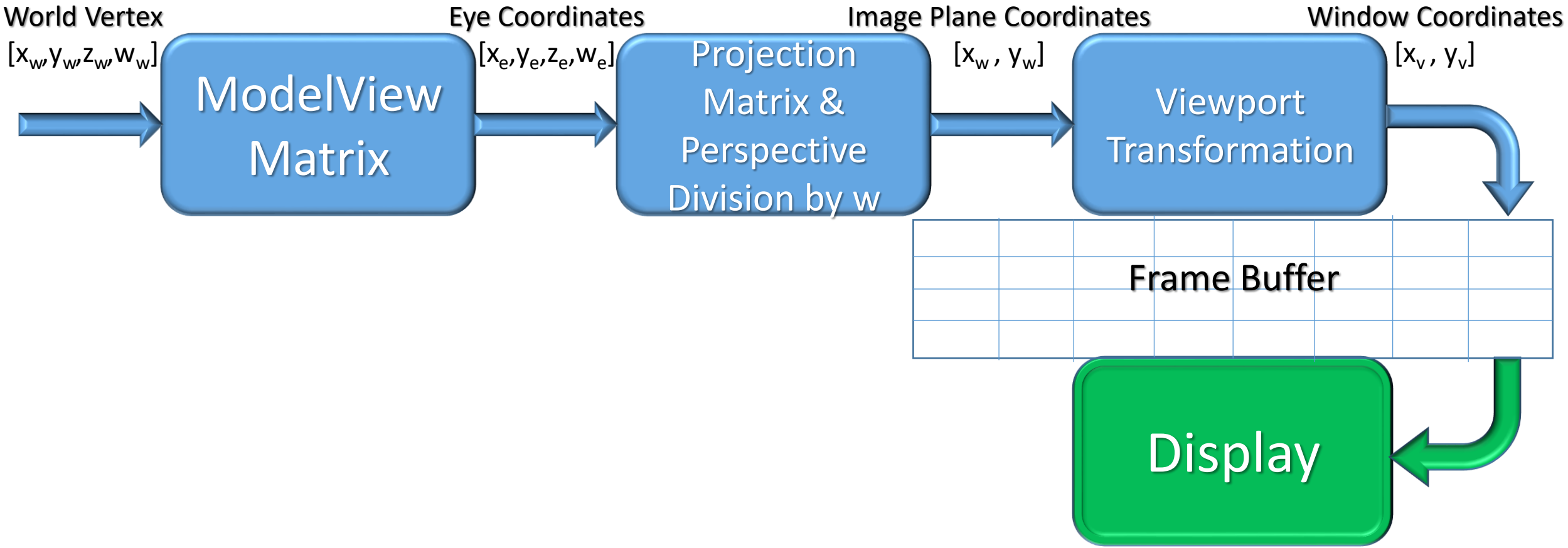
Displaying the Geometry

- Given an application in the world coordinate system, how do we display it on a digital screen?

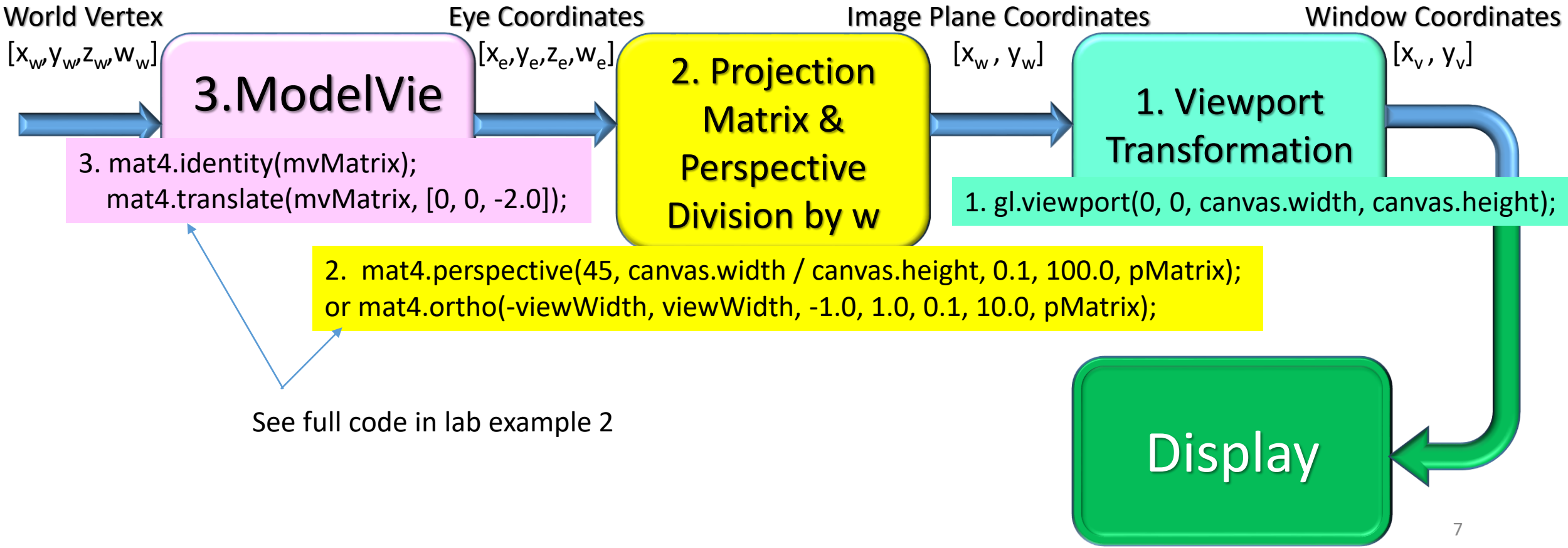
Displaying the Geometry

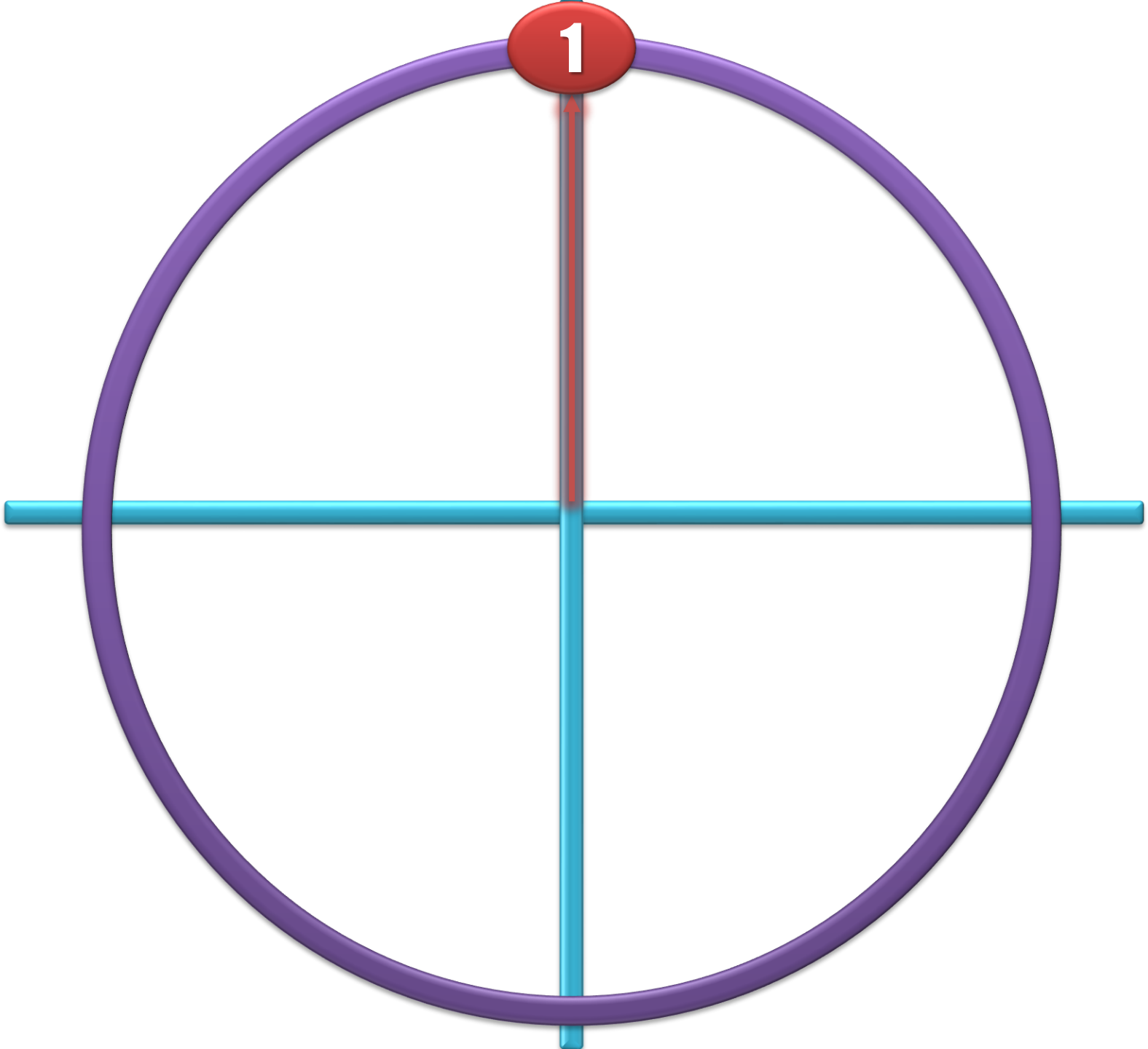
1. Transform to camera coordinates
2. Transform into canonical view volume
3. Clip objects and surfaces to frustum
4. Projection onto the viewing plane
5. Viewport transformation
6. Rasterize

Vertex Transformation Pipeline

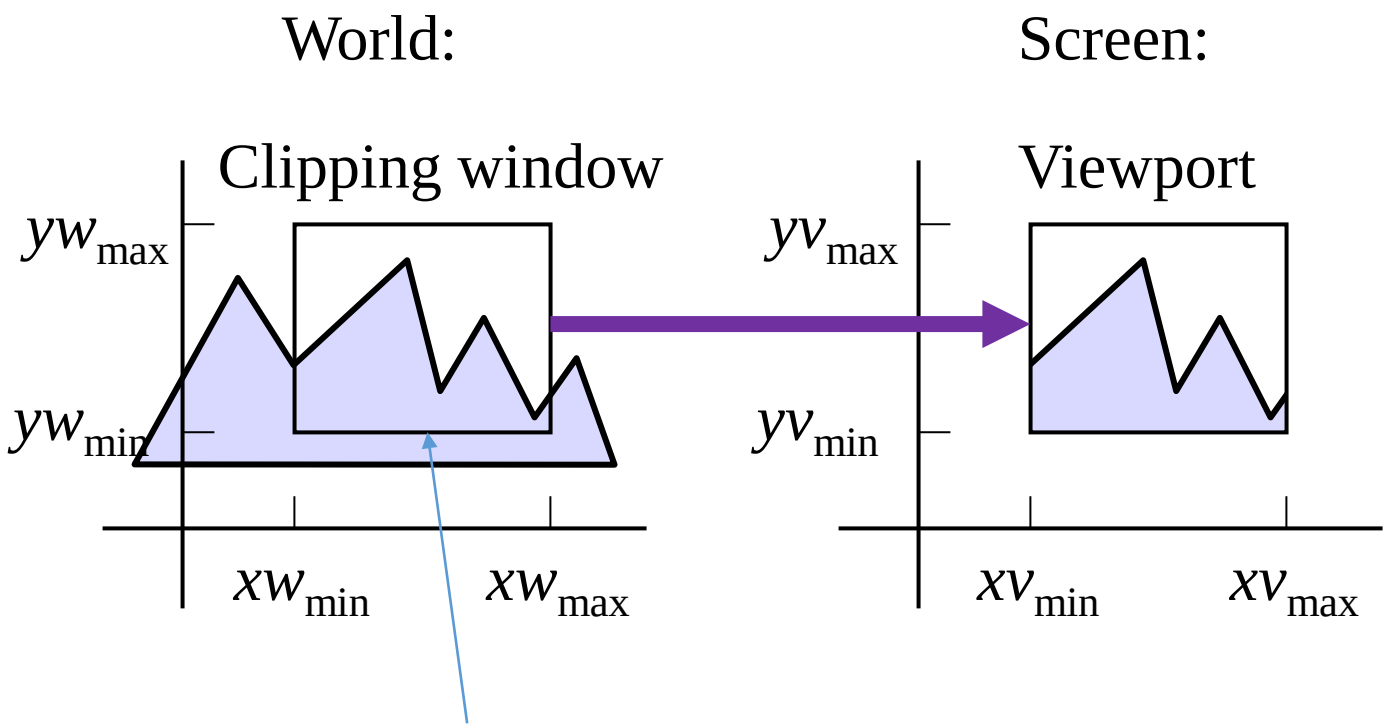


Vertex Transformation Pipeline in WebGL




1

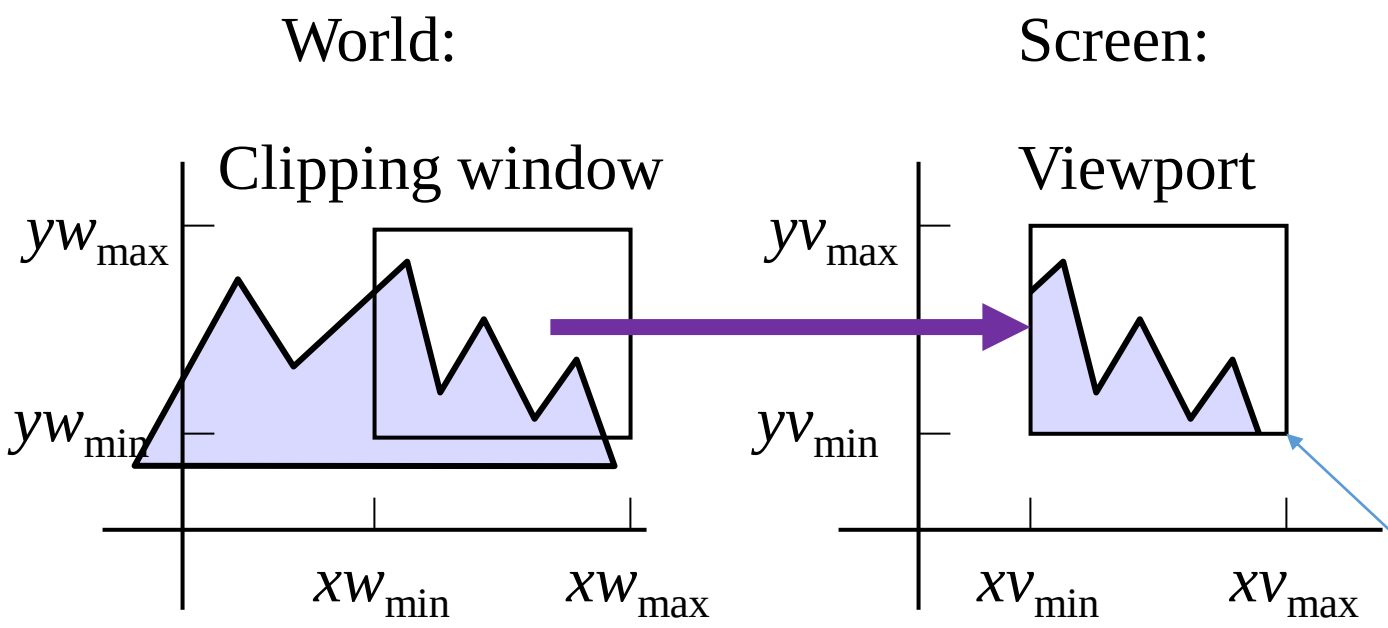
Viewport



Note: “world window” is in the “world-eye” coordinates

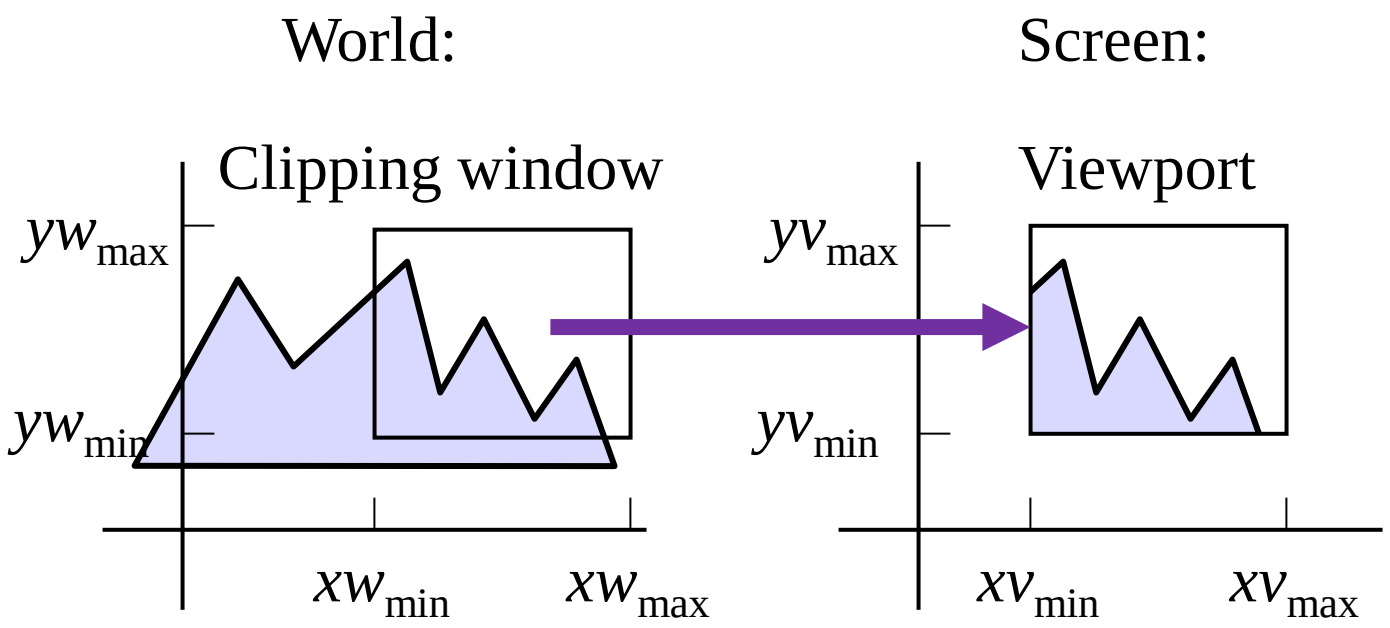
Note: “world window” is NOT the Windows 7 window

World Window to Viewport

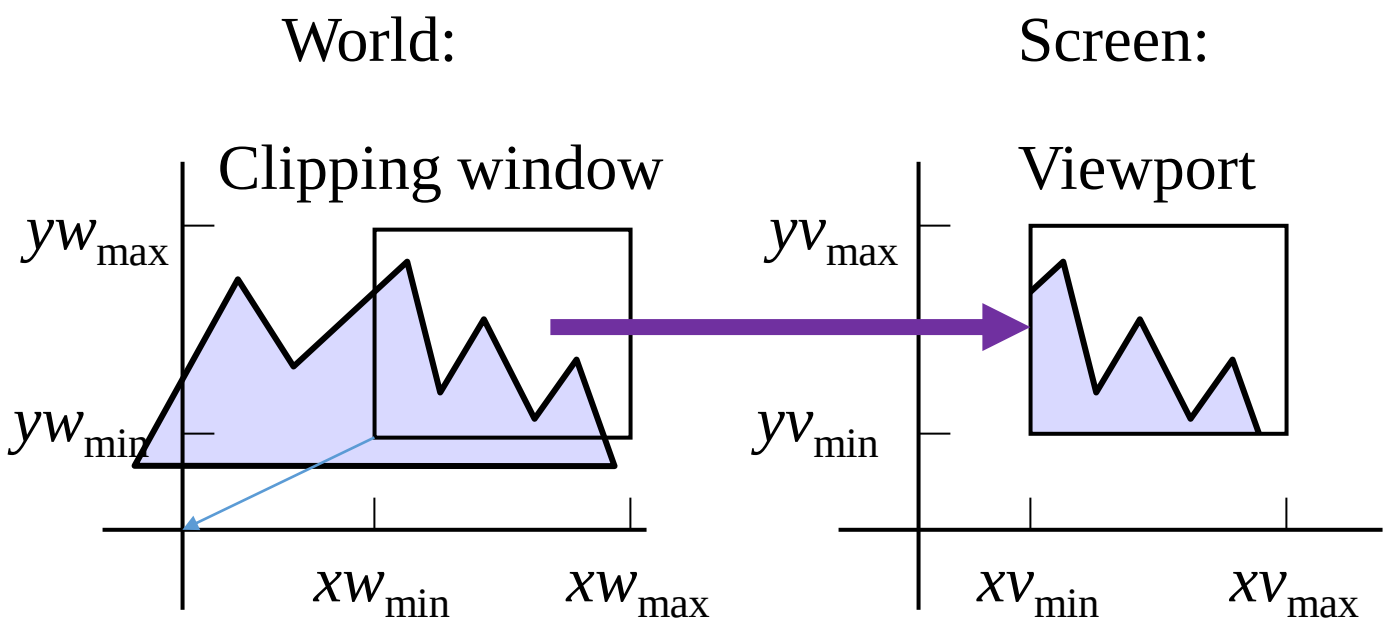


If the world window is moved,
we want only the portion in the window to be viewed

Window-Viewport, How?

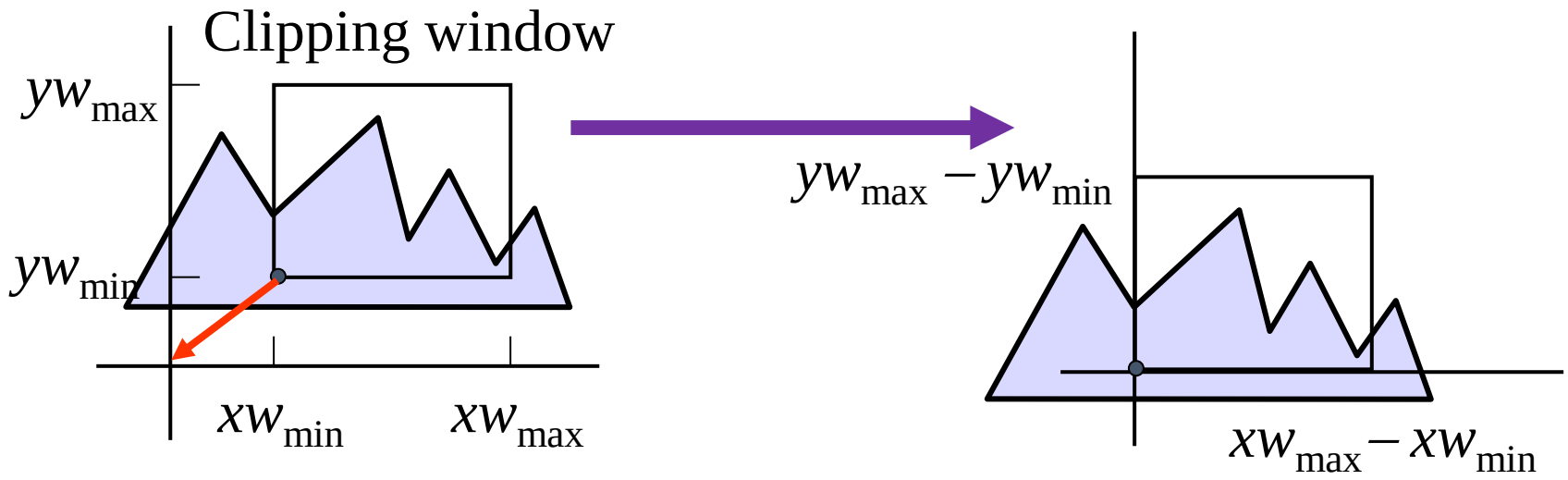


Window-Viewport Map 1



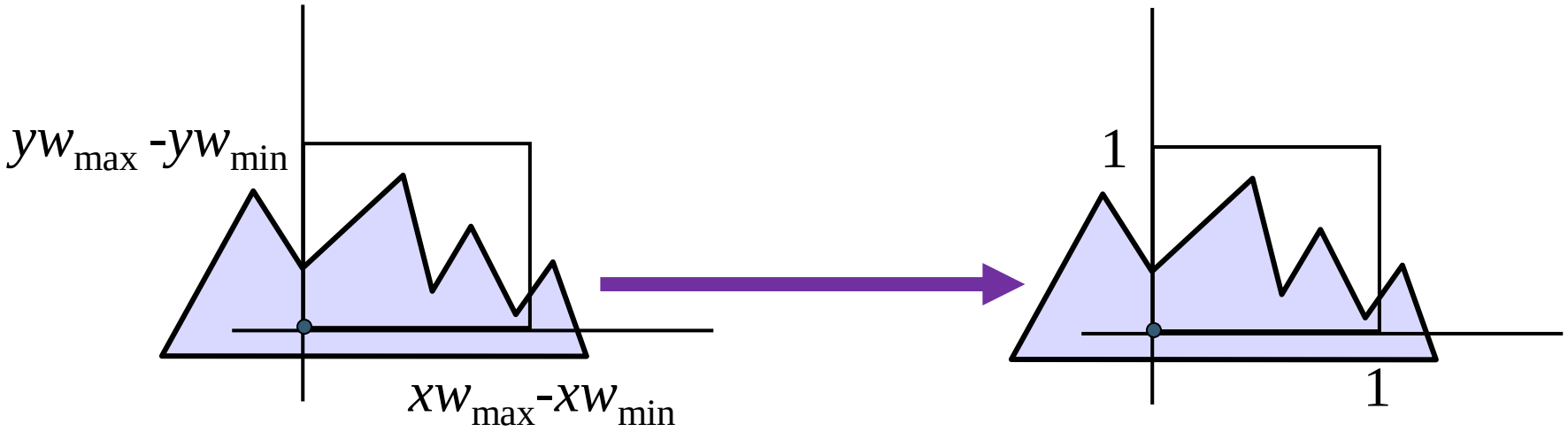
1. Translate by $(-xw_{min}, -yw_{min})$

Window-Viewport Map 1



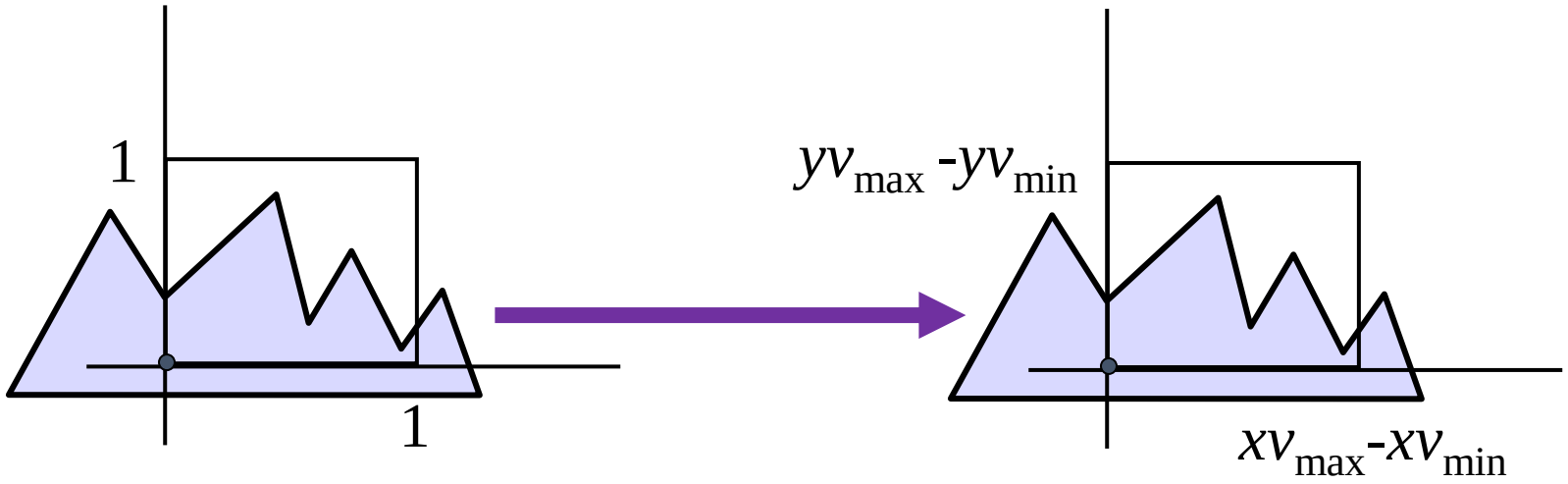
1. Translate by ($-xw_{min}$, $-yw_{min}$)

Window-Viewport Map 2



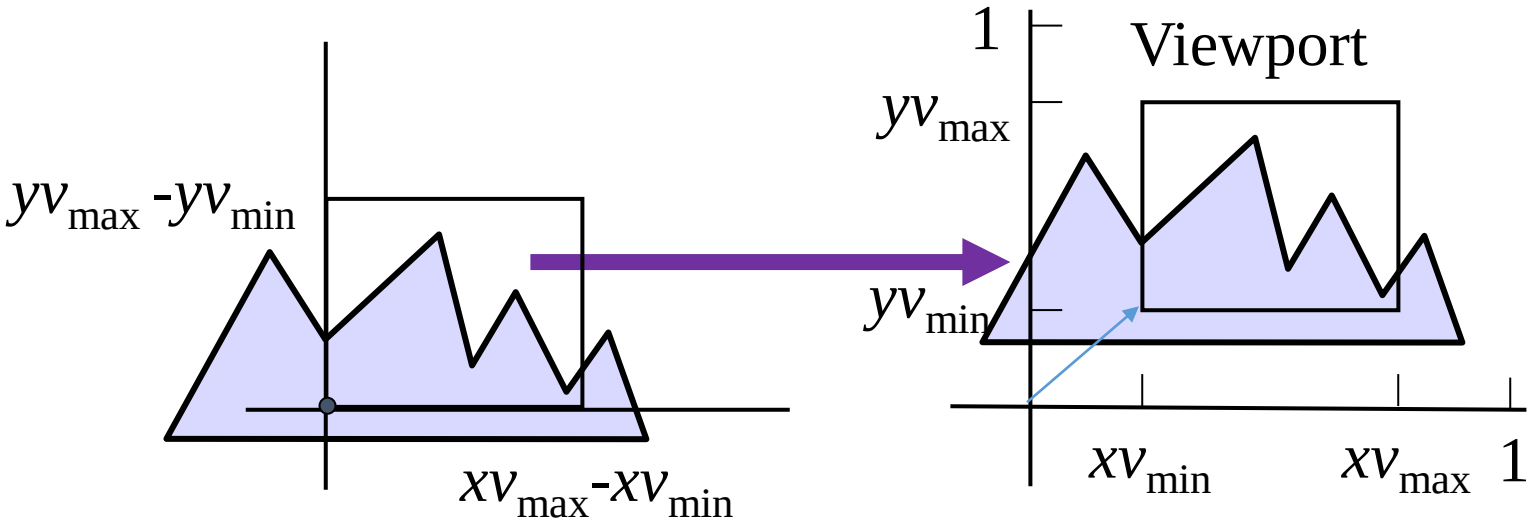
2. Scale by $(1/(xw_{max} - xw_{min}), 1/(yw_{max} - yw_{min}))$

Window-Viewport Map 3



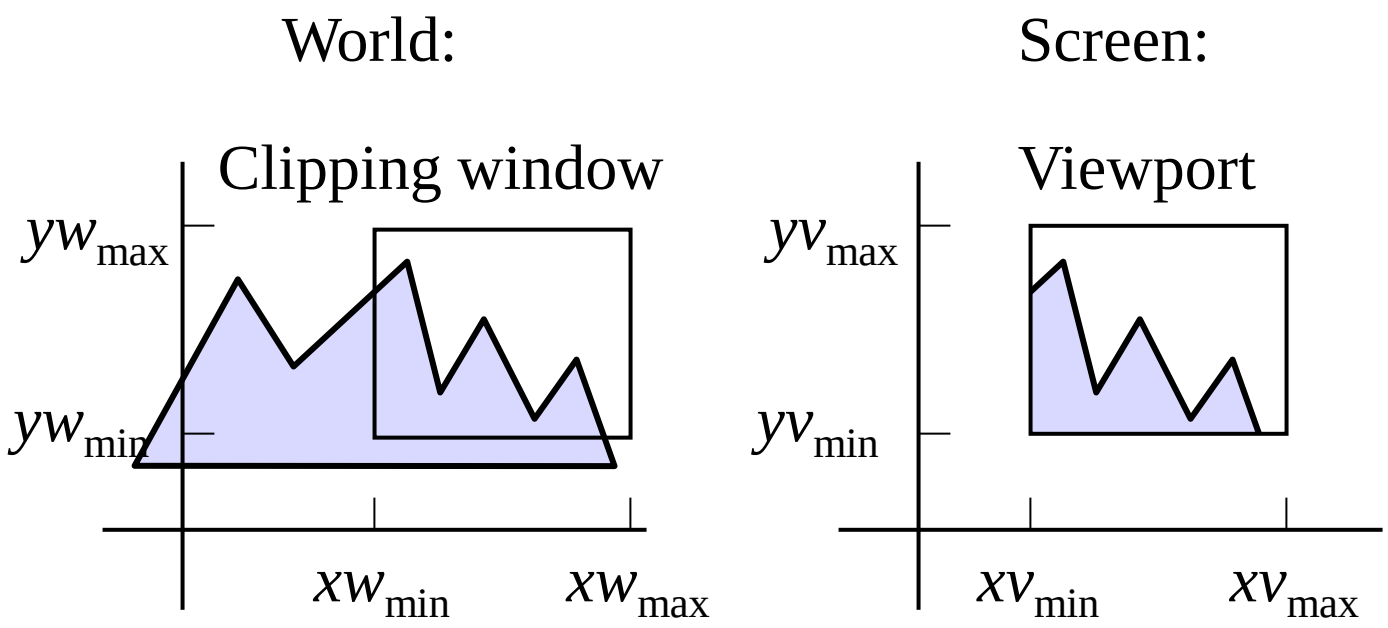
3. Scale by $((xv_{max} - xv_{min}), (yv_{max} - yv_{min}))$

Window-Viewport Map 4



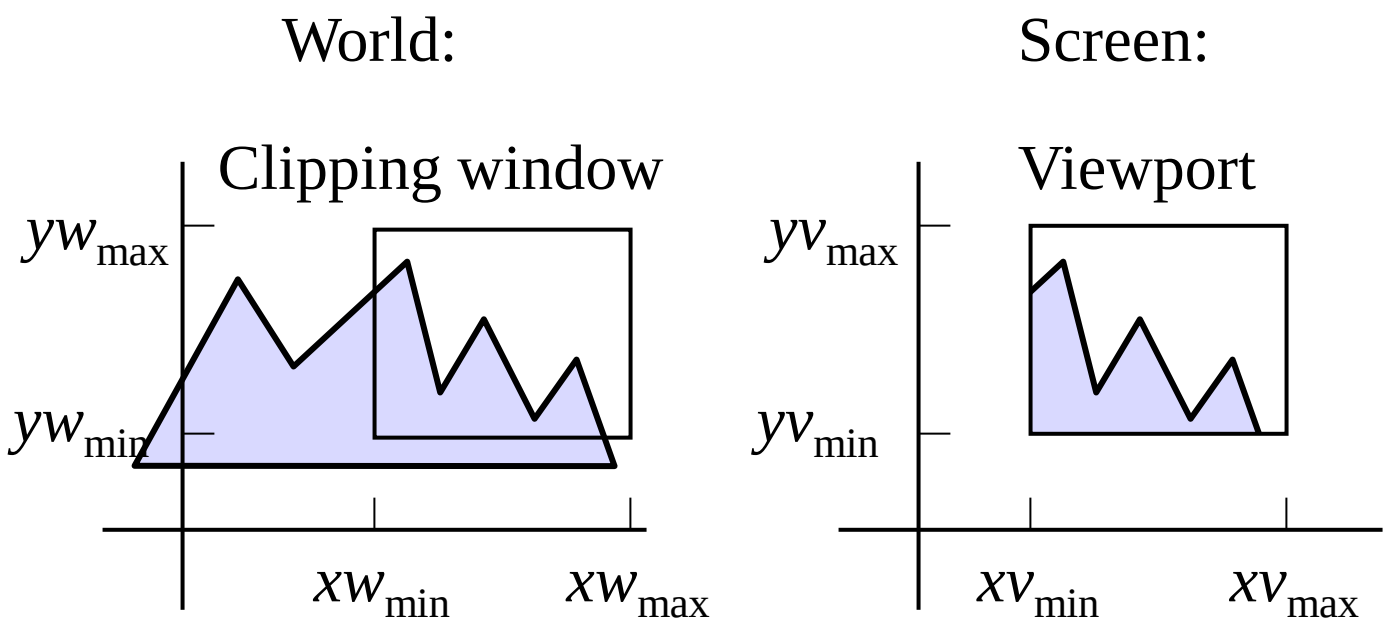
4. Translate by (+xv_{min}, + yv_{min})

What do you get?



$$\mathbf{T}(xv_{min}, yv_{min})\,\mathbf{S}\left(\frac{xv_{max}-xv_{min}}{xw_{max}-xw_{min}}, \frac{yv_{max}-yv_{min}}{yw_{max}-yw_{min}}\right)\mathbf{T}(-xw_{min}, -yw_{min})$$

Why?



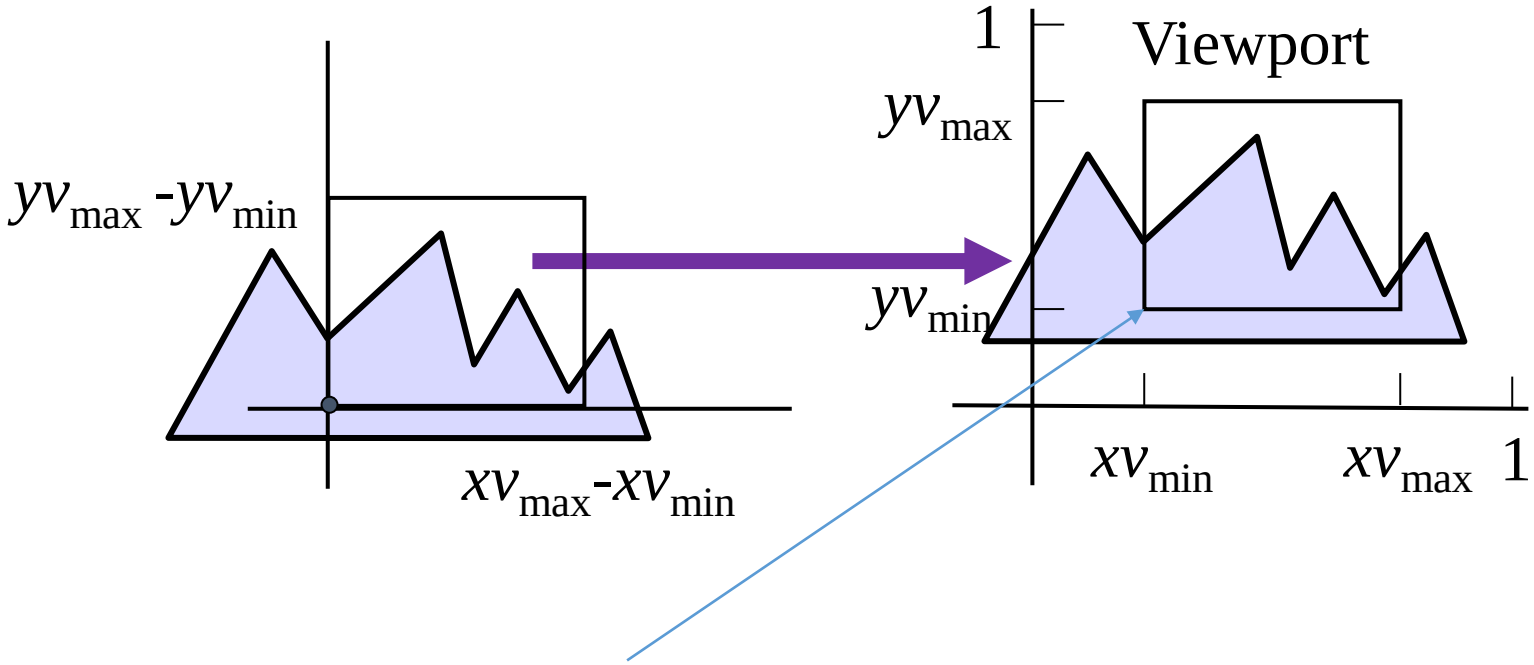
$$\mathbf{T}(xv_{min}, yv_{min})\,\mathbf{S}\left(\frac{xv_{max}-xv_{min}}{xw_{max}-xw_{min}}, \frac{yv_{max}-yv_{min}}{yw_{max}-yw_{min}}\right)\mathbf{T}(-xw_{min}, -yw_{min})$$

Because we pre-multiply all points (x,y) by M

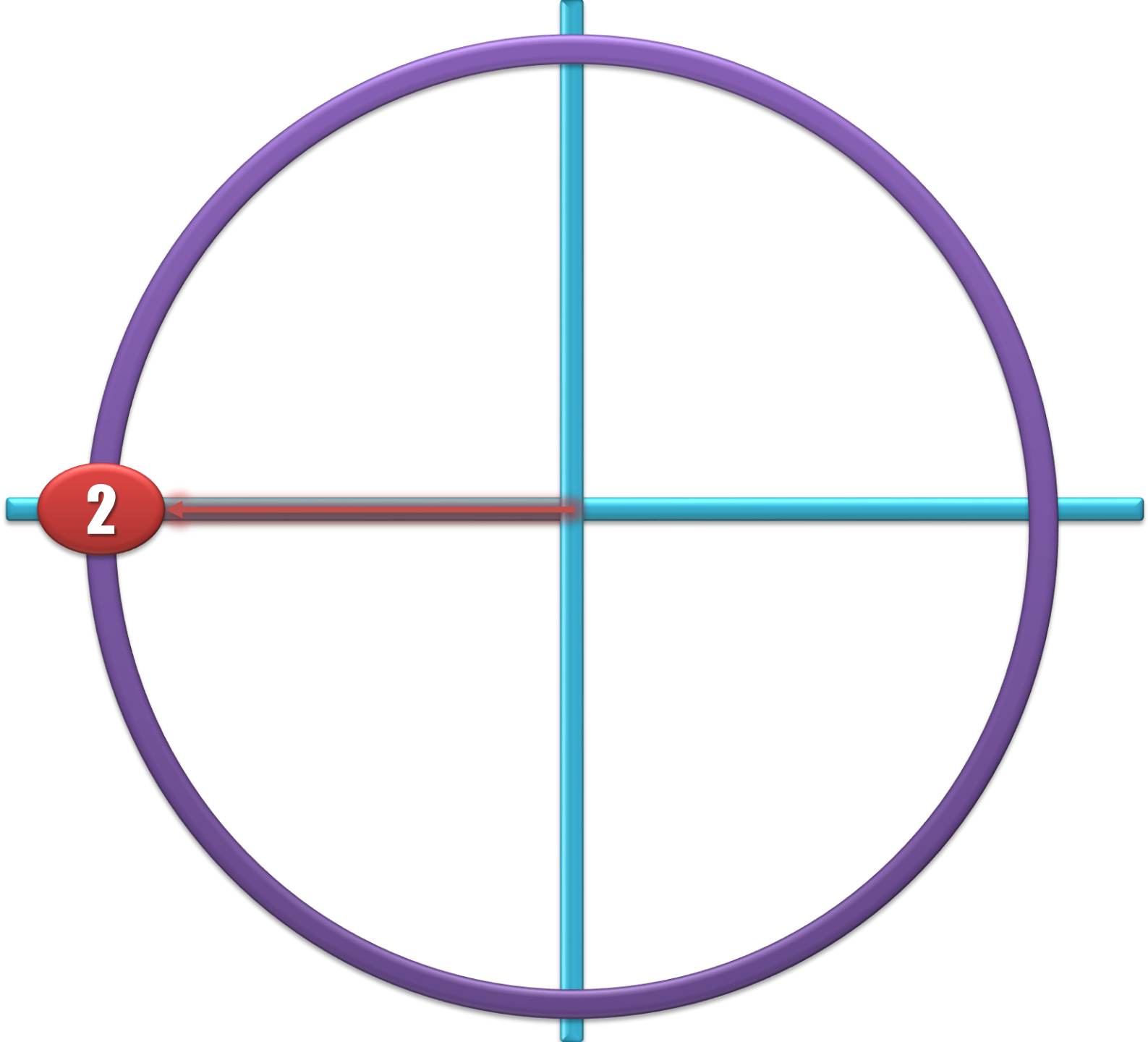
$$M\,p_w = p_v$$

$$\left[\mathbf{T}(xv_{\min}, yv_{\min})\,\mathbf{S}\left(\frac{xv_{\max}-xv_{\min}}{xw_{\max}-xw_{\min}}, \frac{yv_{\max}-yv_{\min}}{yw_{\max}-yw_{\min}}\right)\mathbf{T}(-xw_{\min}, -yw_{\min})\right]\begin{bmatrix} x \\ y \end{bmatrix}$$

What do you get?



Screen Coordinates!


2

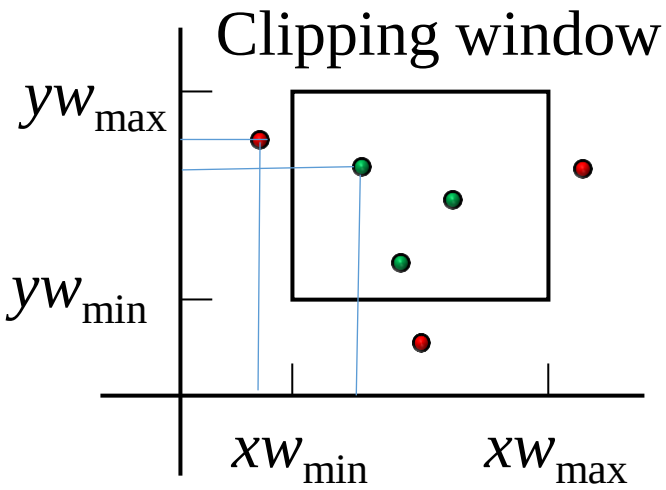
Clipping Points in 2D Windows Coordinates

- Points in 2D
- Save P(x,y)
  - if x and y are in the box

$$xw_{\min} \le x \le xw_{\max}$$

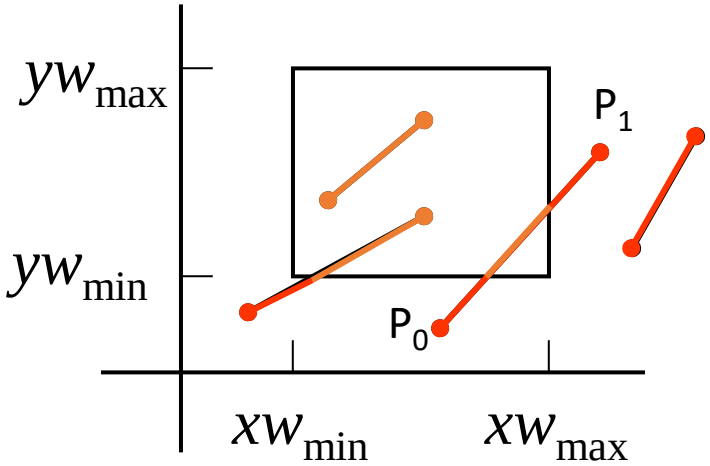
$$yw_{\min} \le y \le yw_{\max}$$

  - else, discard the point P(x,y)



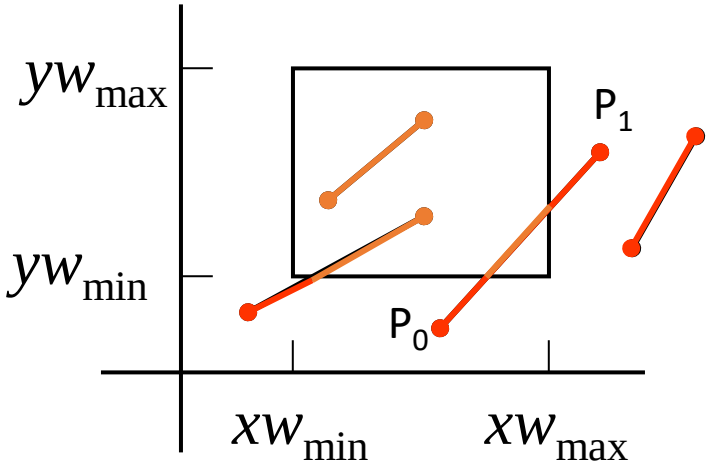
Clipping Lines in 2D Windows Coordinates

- Lines in 2D
- Points $P_0(x,y)$ and $P_1(x,y)$
  - Are connected
  - We need to computing intersections!



Clipping Lines in 2D Windows Coordinates

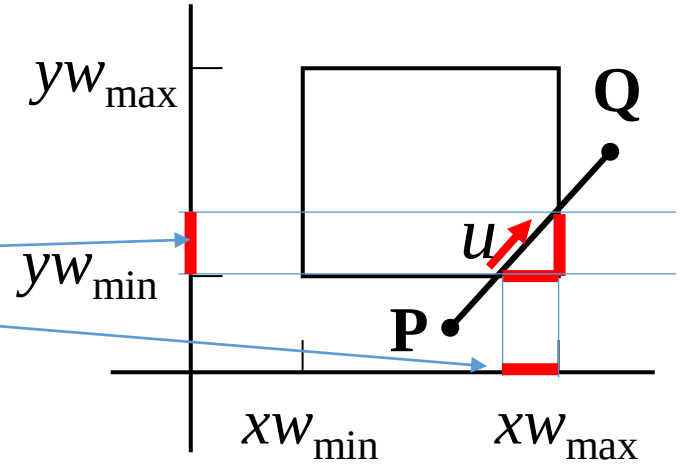
- But, how to do it fast?



Clipping Lines in 2D

- Basic Algorithm:

Determine the interval (u_0, u_1) in the window rectangle, by calculating the crossings with edges of window.

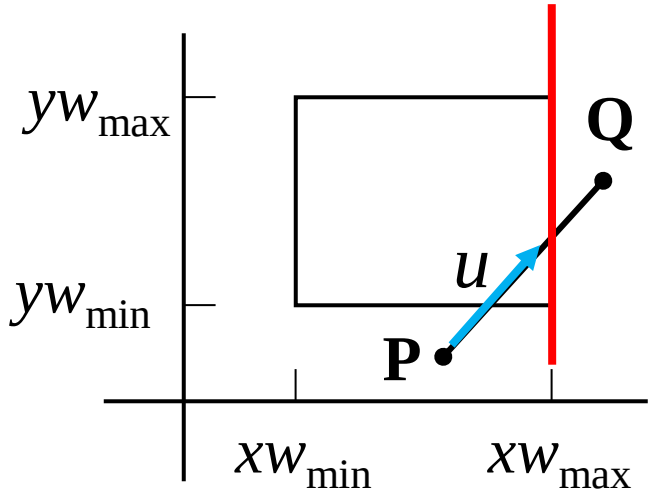


A line segment is given by:
$\mathbf{X}(u) = \mathbf{P} + u\,(\mathbf{Q}–\mathbf{P})$,
with $0 <= u <= 1$

Clipping Lines in 2D Right

- Basic Algorithm:

$u_0 := 0$; $u_1 = 1$;
(* *Right side:* *)
if $Px == Qx$ then
  if $Px > xw_{max}$ **then** return empty
else
  $u \leftarrow (xw_{max} - Px) / (Qx - Px)$
  if $Px < Qx$ **then**
    if $u < u_1$ then $u_1 := u$;
  else **if** $u > u_0$ then $u_0 := u$;
if $u_0 > u_1$ **then** return empty

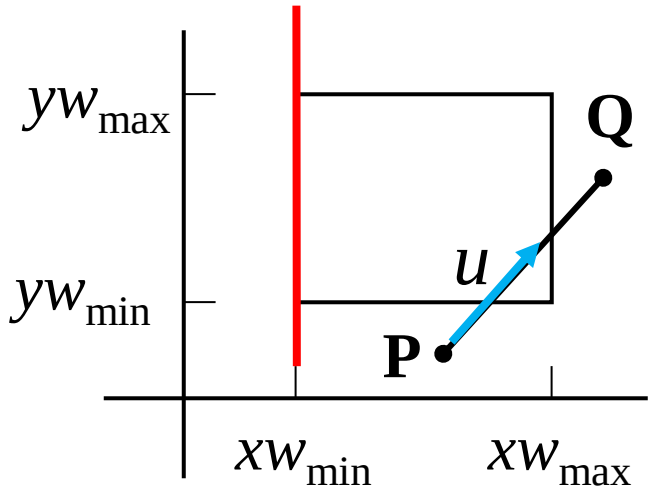


A line segment is given by:
$\mathbf{X}(u) = \mathbf{P} + u\,(\mathbf{Q}–\mathbf{P})$,
with $0 <= u <= 1$

Clipping Lines in 2D Left

- Basic Algorithm:

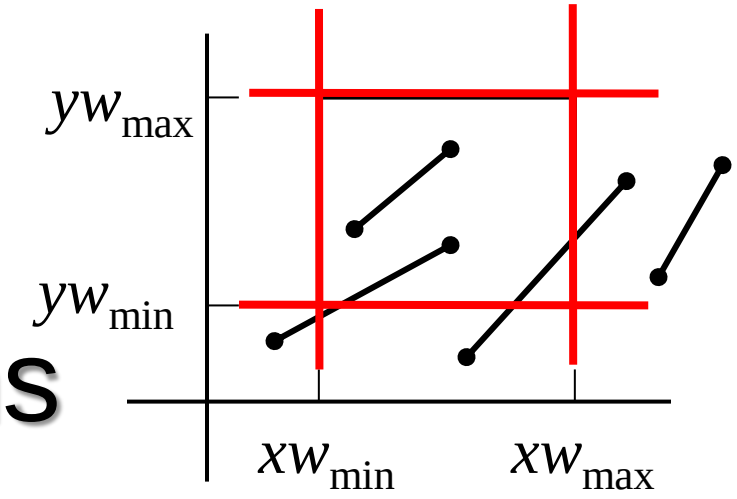
```
(* Left side: *)
if Px == Qx then
    if Px < xw_min then return empty
else
   u ← (xw_min – Px) / (Qx - Px)
   if Px < Qx then
            if u > u_0  then u_0 := u;
      else if u < u_1 then u_1 := u;
if u_0 > u_1 then return empty

(* Lower and upper side similar*)
```



A line segment is given by:
X(u) = **P** + u (**Q**–**P**),
with 0 <= u <= 1

But, this is expensive; why?

- Because
  - must compute intersections for all lines against all four window sides

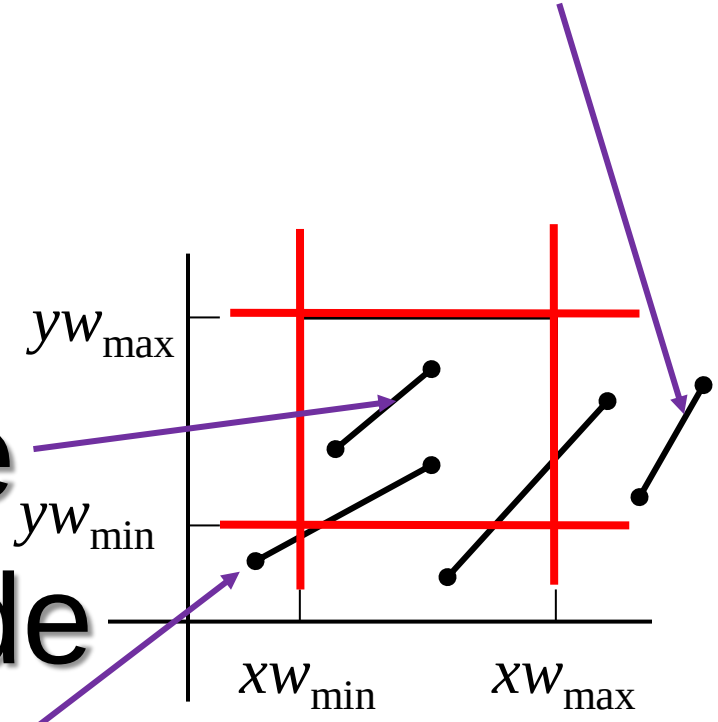


Can you find a better algorithm?

Yes!

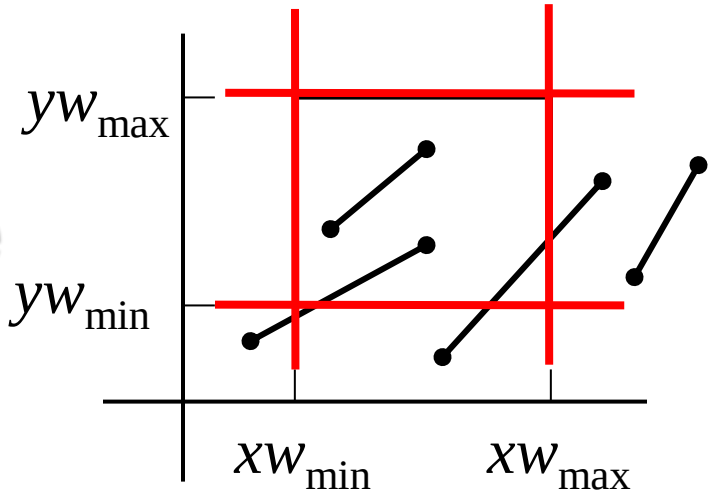
Observations:

- Some lines are fully inside
- Some lines are fully outside
- Some cross corners
- Some cross outside



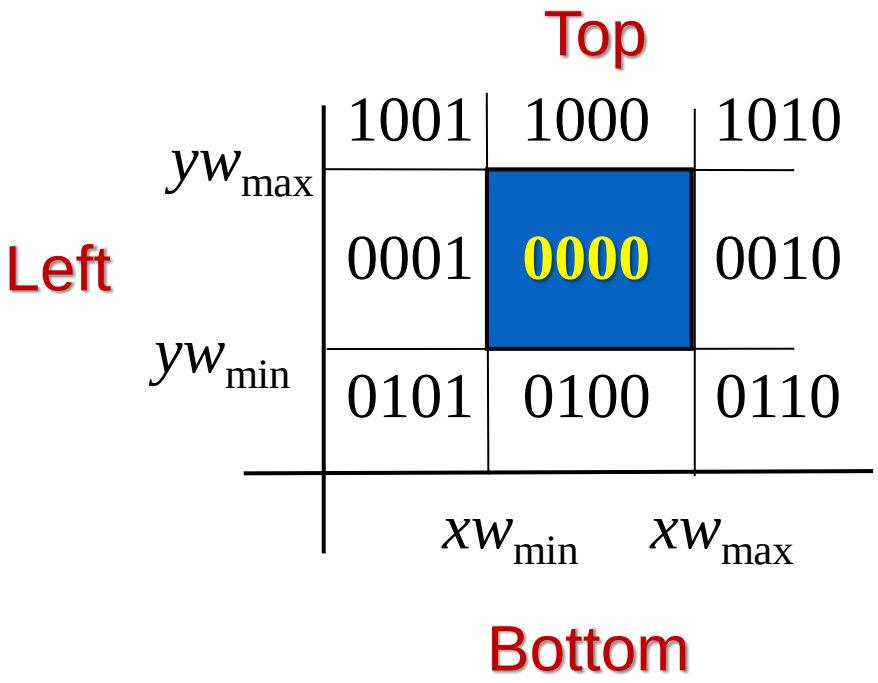
Answer: Cohen-Sutherland

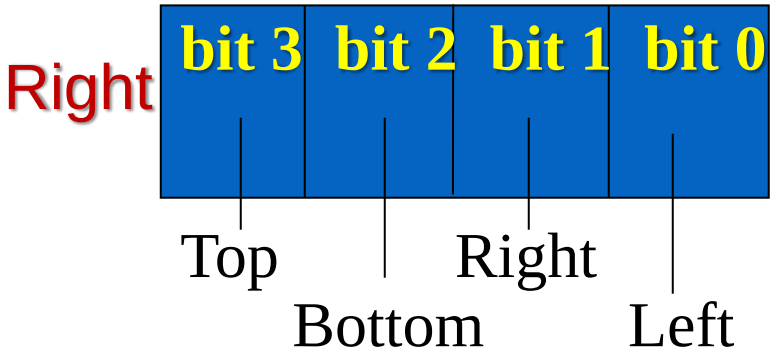
- Label the points with a binary code
- Use the binary code to decide which lines are completely outside
- Then, use them again to find intersections



Answer: Cohen-Sutherland

- Assign each point a *region code C*
- Let the bit = 1 for outside, 0 for inside.

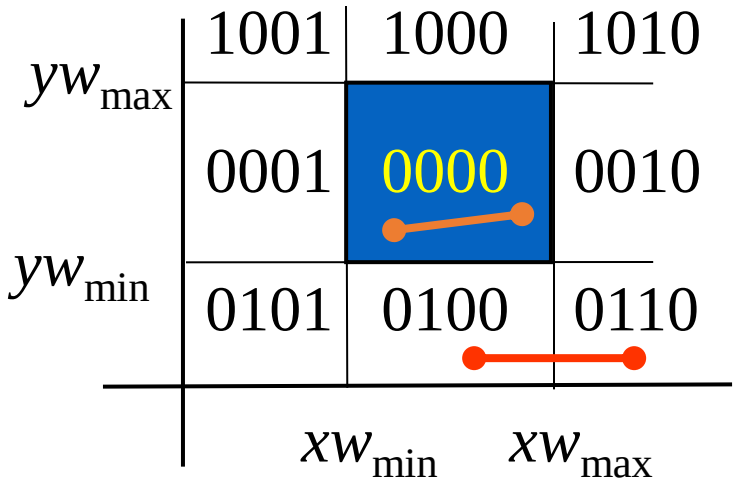


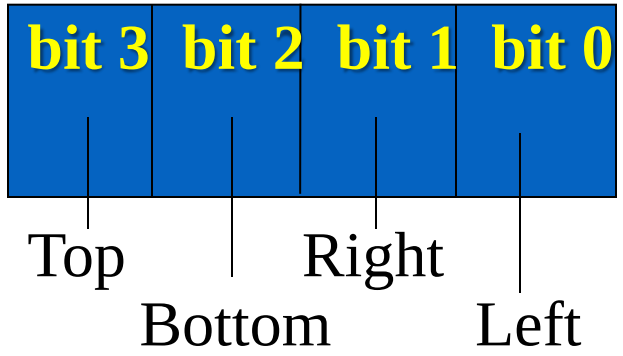


How to set the code of P

- Let codeP = 0 0 0 0
- For each window edge i = 0,1,2,3
  - Compare Px with wx_{Left} and wx_{Right}
  - If Px is outside then
    - codeP= codeP xor 0 0 0 1 (codeLeft)
    - codeP= codeP xor 0 1 0 0 (codeRight)
  - Same for Py with wy_{Bottom} and wy_{top}

Example





Fast test on status end points line with codes C_0 and C_1 :

C_0 *bitwise-or* C_1 == *0000*: then completely IN;

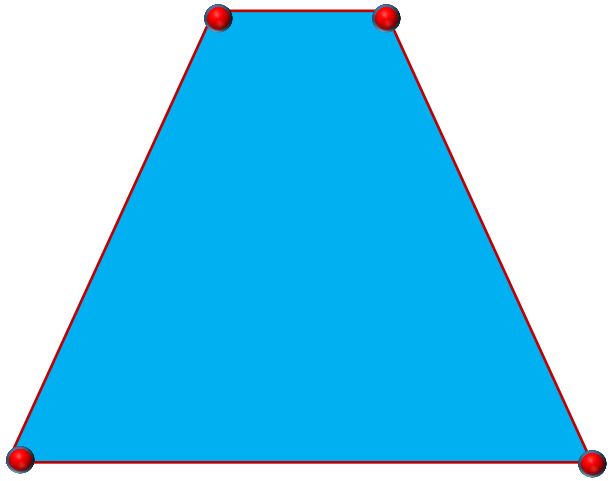
C_0 *bitwise-and* C_1 <> *0000*: then completely OUT;

Else: fast intersection-calculation using codes.

Polygon Clipping

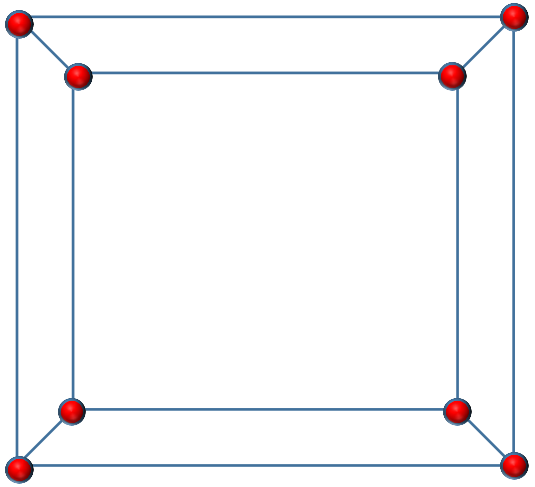
- Not as simple as line clipping; why?
- Must be systematic
- Must be general: any shapes

First, what is a polygon?



Polygon?

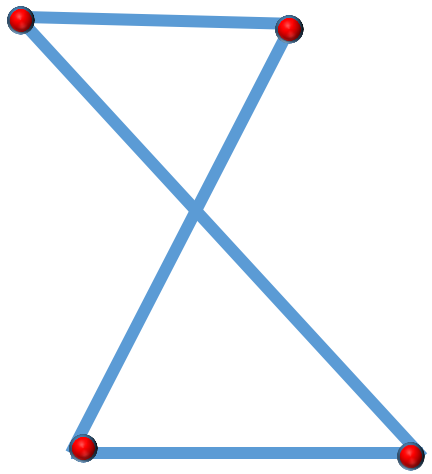
Yes!



Polygon?

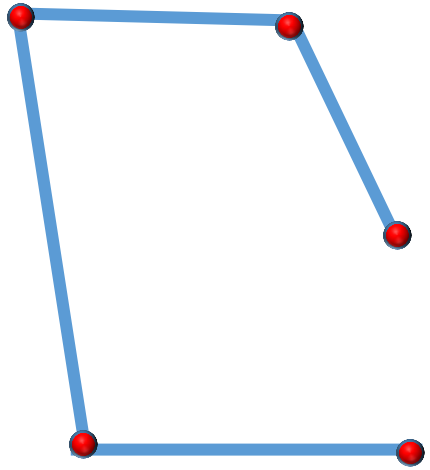
No!

A mesh



Polygon?

No!
Not a simple polygon

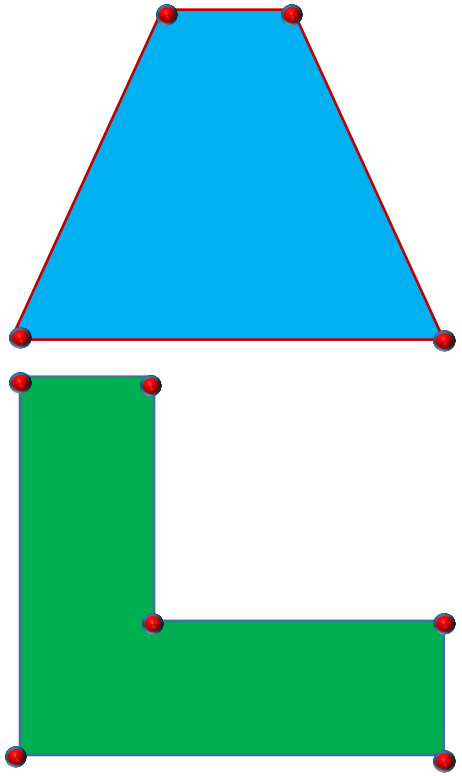


Polygon?

No!
A poly-line

A Simple Polygon

- Must enclose a region in the plane
  - Convex
  - Concave

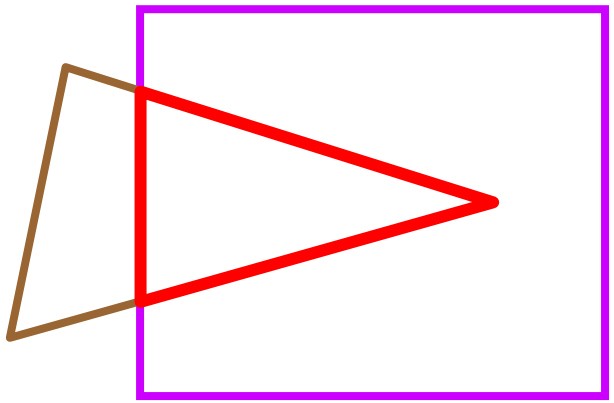


Polygon Clipping

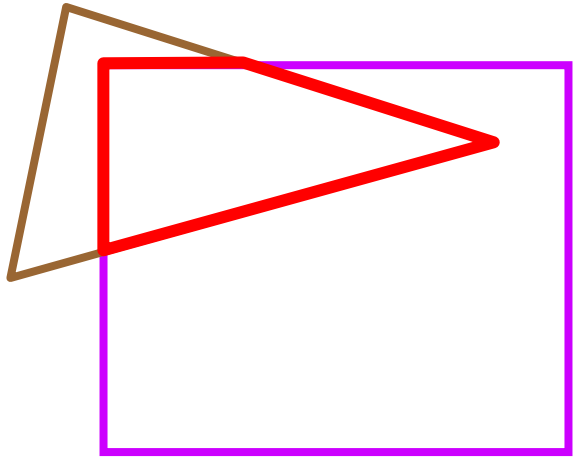
- Assumptions:
  - Clipping polygon must be convex
  - The target polygon must be simple

Why is clipping hard?

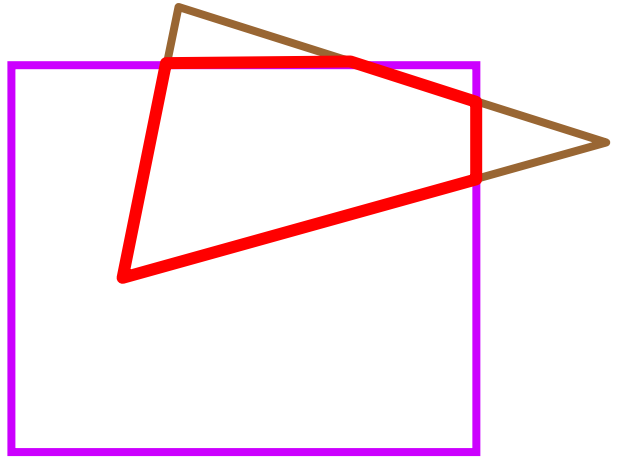
- Let's consider a simple triangle
- Clip against a rectangular window



Result: a triangle

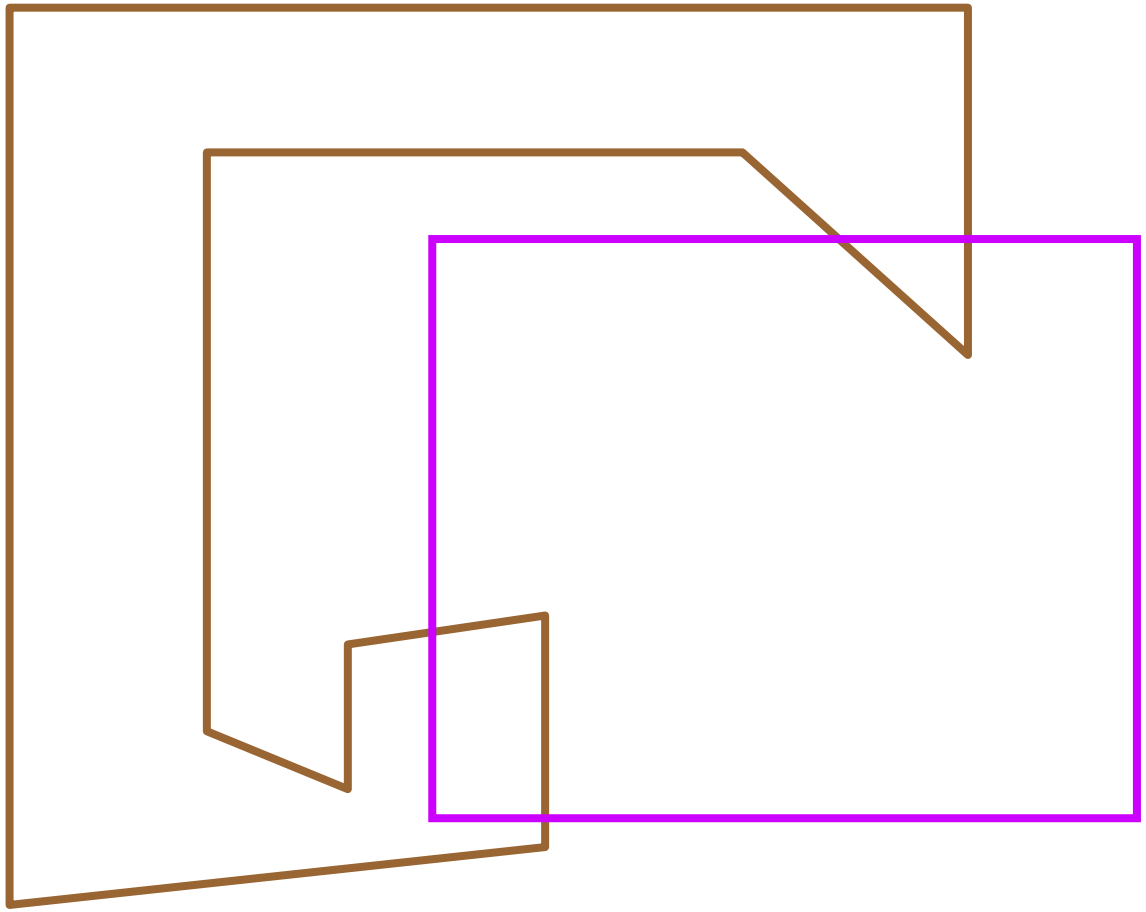


Result: a quad

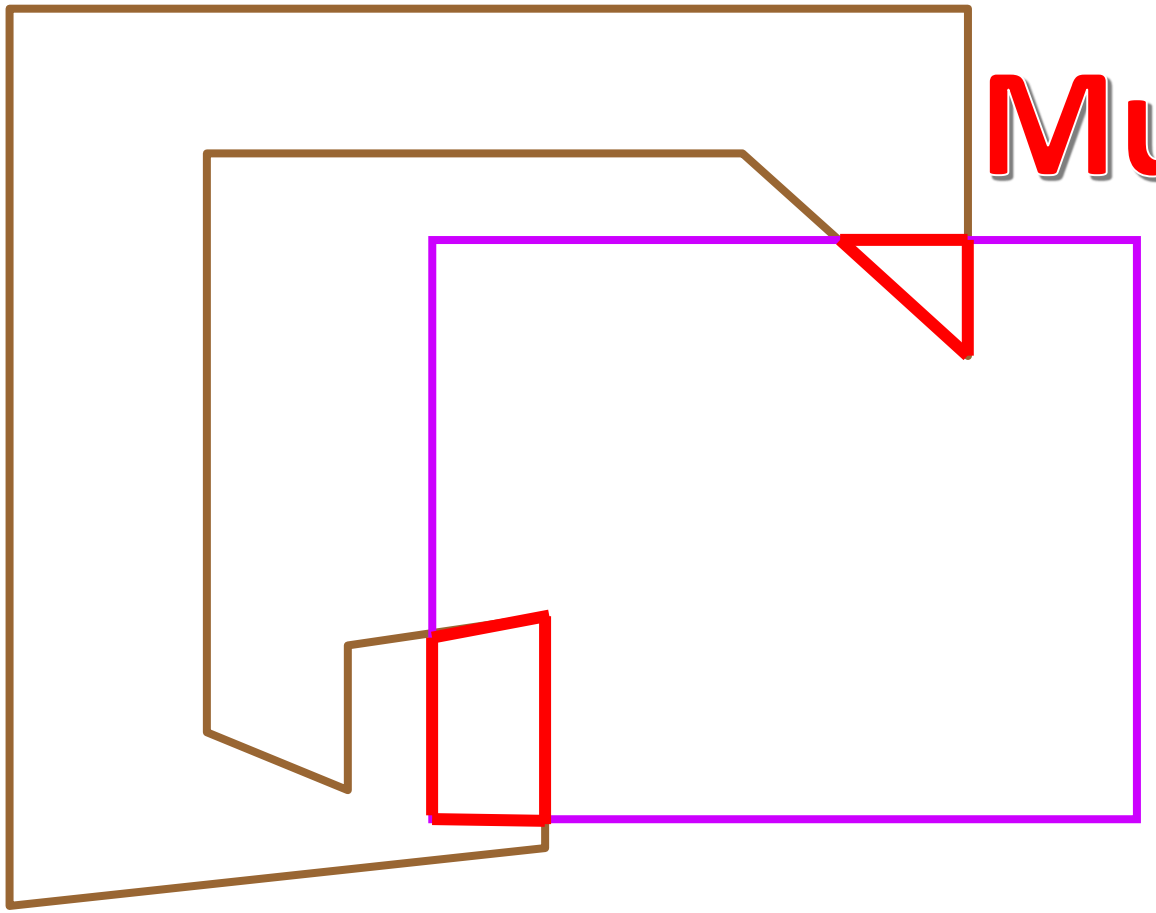


Result: a pentagon

What about this?



One Polygon goes to:

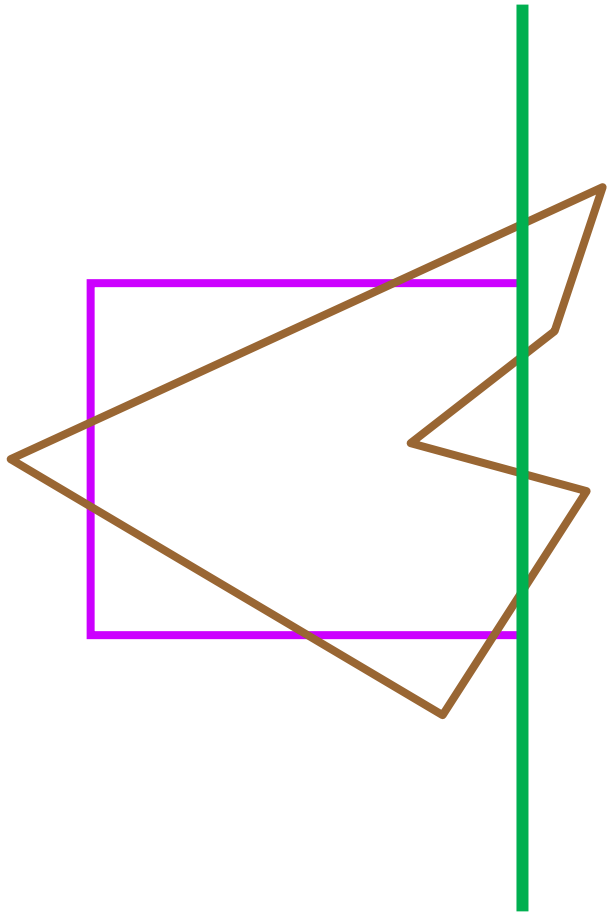


Multi-Polygons

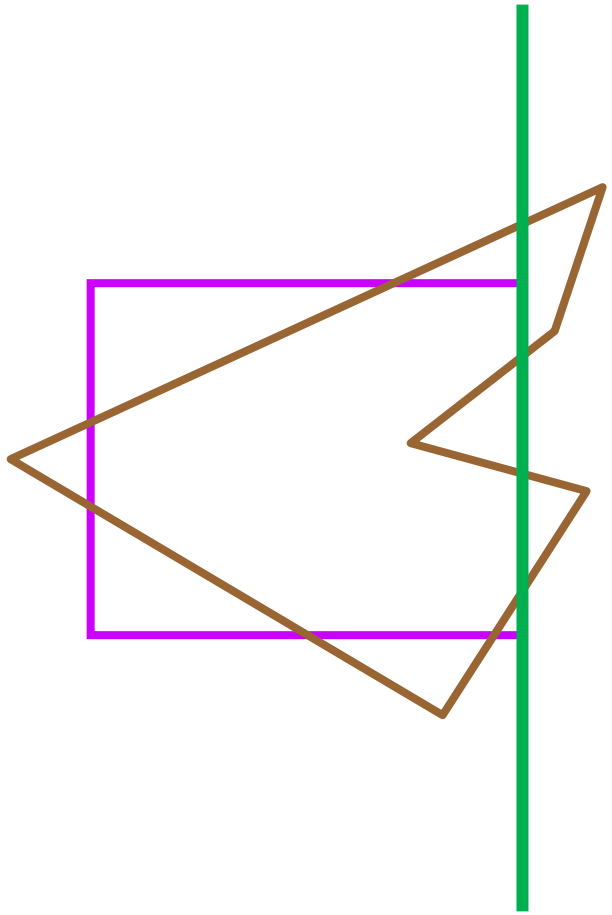
Sutherland-Hodgman

- Basic idea:
  - Treat each window edge as an infinite border
  - Clip against each edge of the viewport
  - Circle around the viewport window
  - Build a new vertex list for the clipped polygon

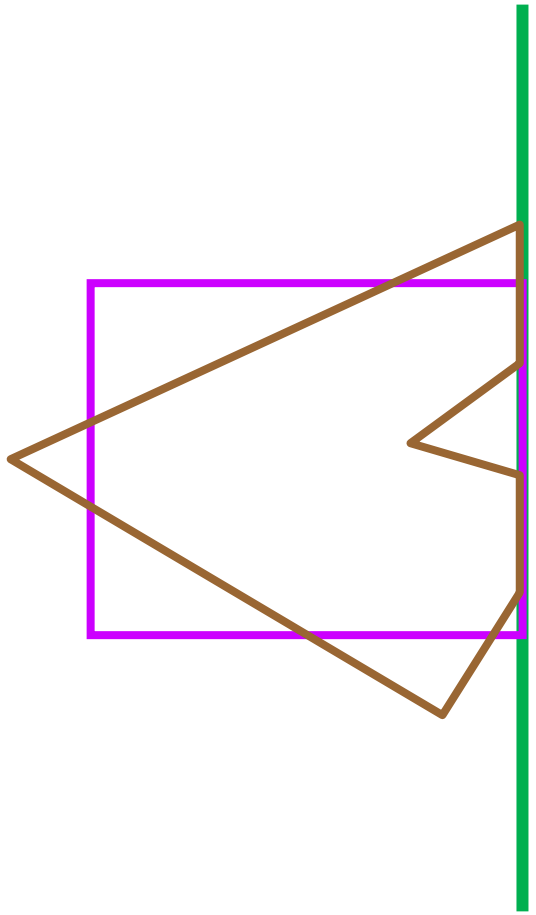
Sutherland-Hodgman



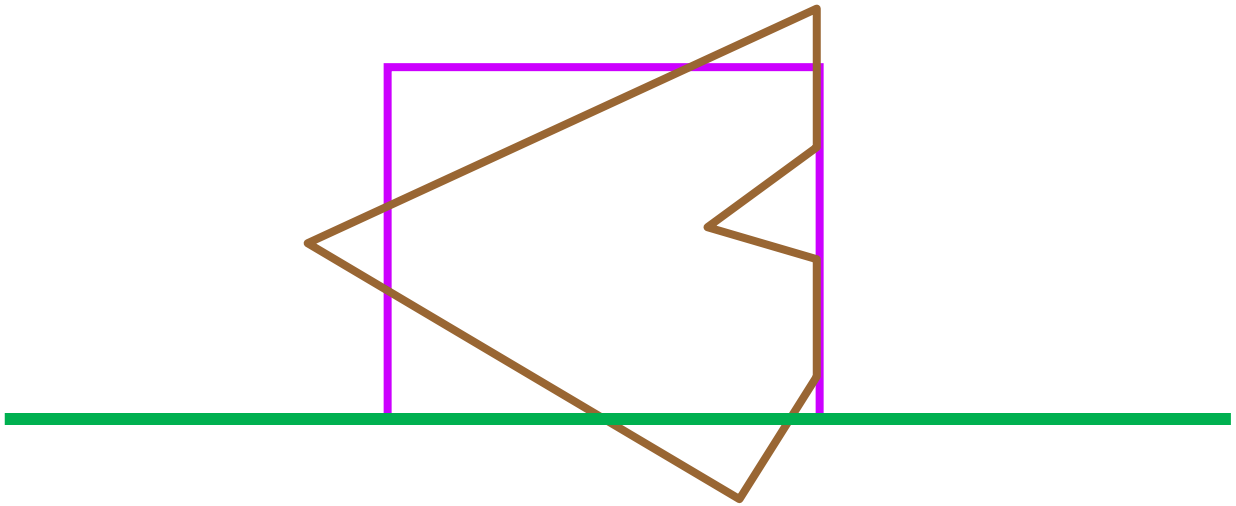
Sutherland-Hodgman



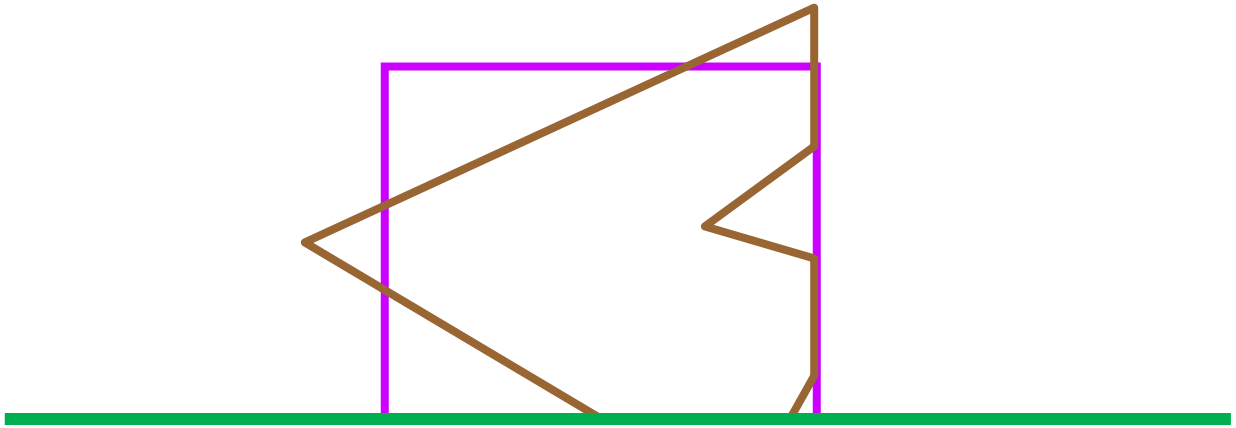
Sutherland-Hodgman



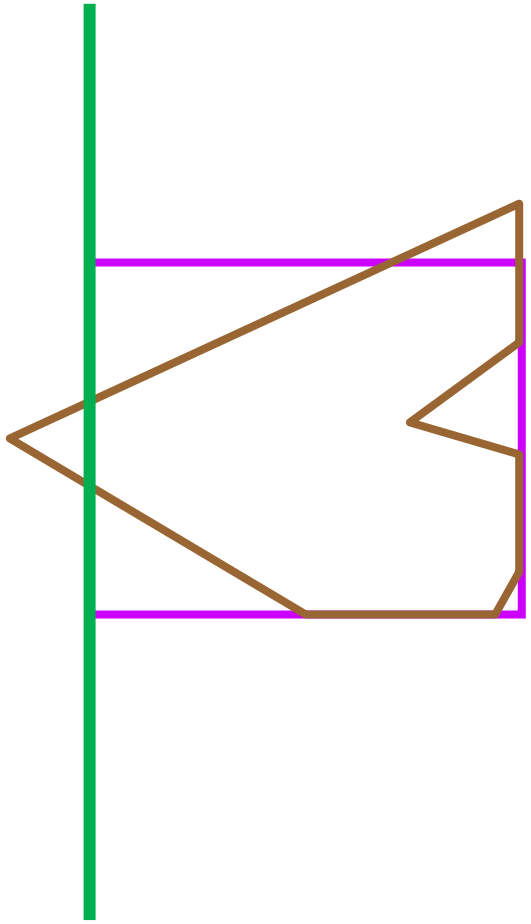
Sutherland-Hodgman



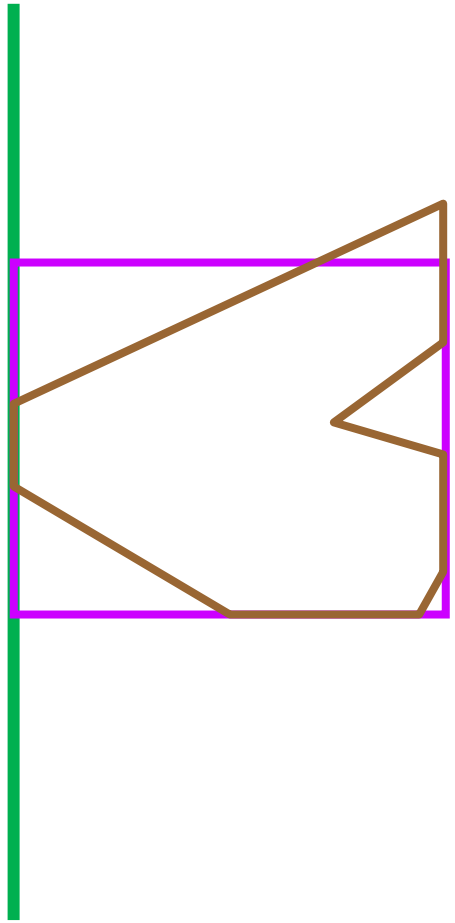
Sutherland-Hodgman



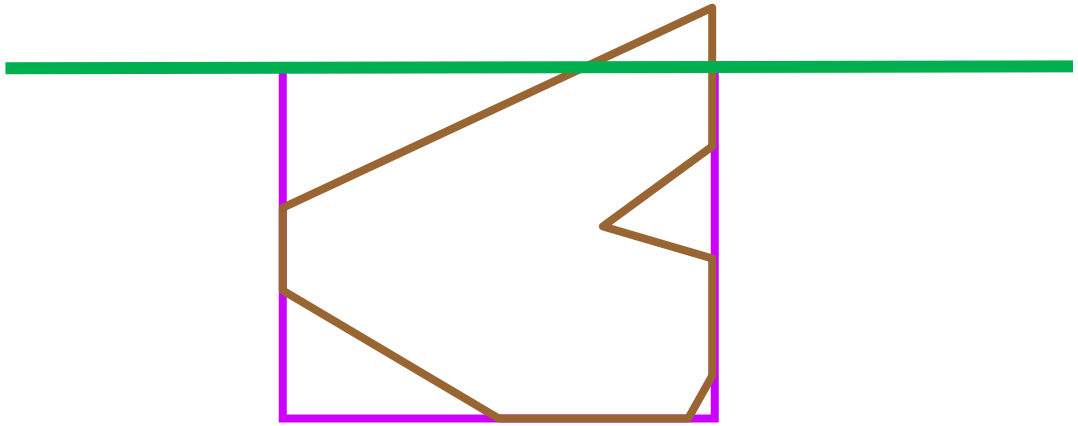
Sutherland-Hodgman



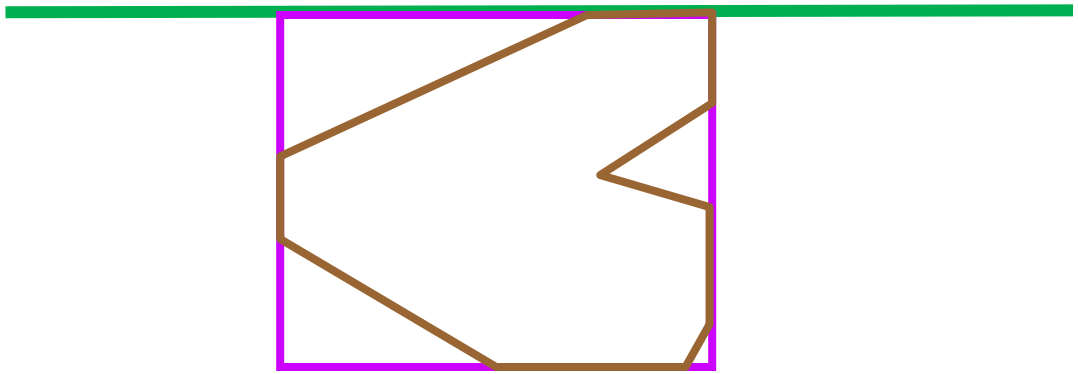
Sutherland-Hodgman



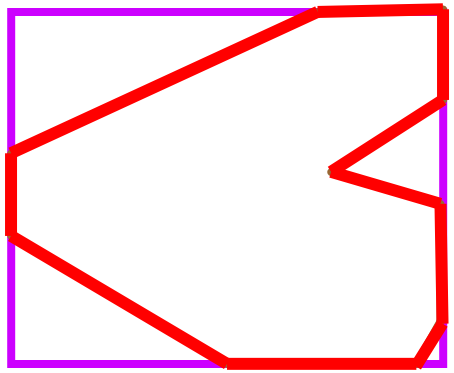
Sutherland-Hodgman



Sutherland-Hodgman



Final result



Sutherland-Hodgman Algo

Sutherland-Hodgman Algo

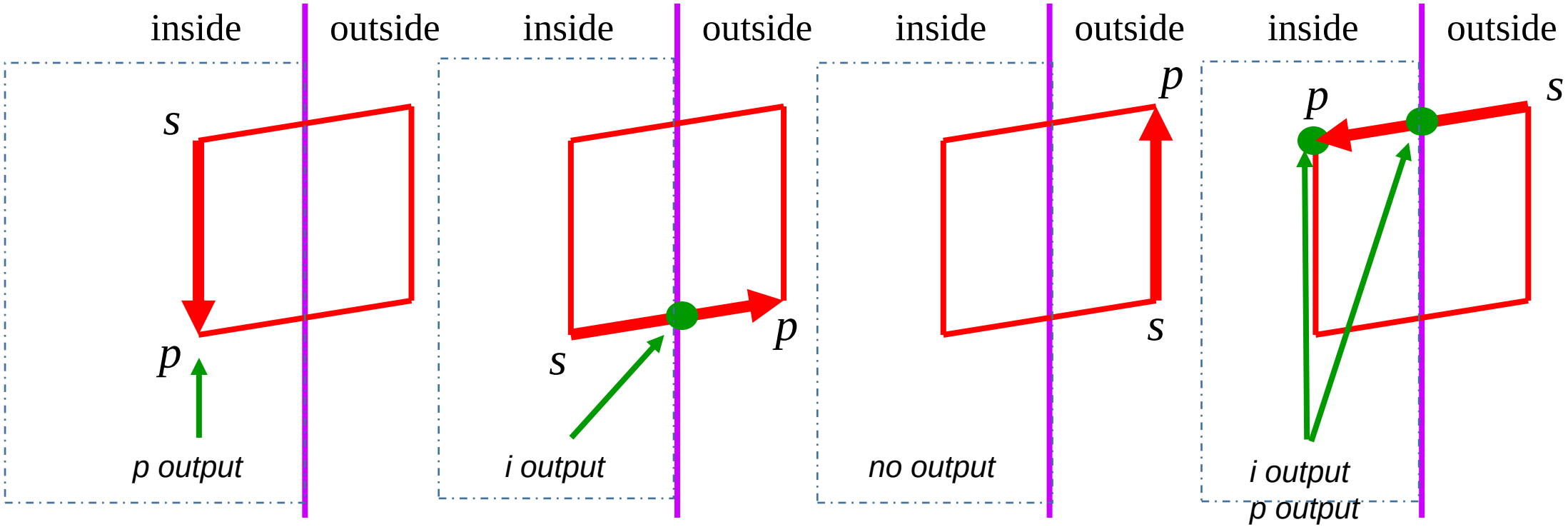
- Input: a list of ordered polygon vertices
- Output: list of clipped polygon vertices
- This algorithm generalizes to 3D

Sutherland-Hodgman Algo

- Let the current vertex be p
- Let the previous vertex be s
- Assume s has been added to the clipped polygon vertex list

Sutherland-Hodgman Algo

- Edge from s to p will be one of four cases:



Four cases:

- *s* inside plane and *p* inside plane
  - Add *p* to output
  - Note: *s* has already been added
- *s* inside plane and *p* outside plane
  - Find intersection point *i*
  - Add *i* to output
- *s* outside plane and *p* outside plane
  - Add nothing
- *s* outside plane and *p* inside plane
  - Find intersection point *i*
  - Add *i* to output, followed by *p*

Inside-Outside Test

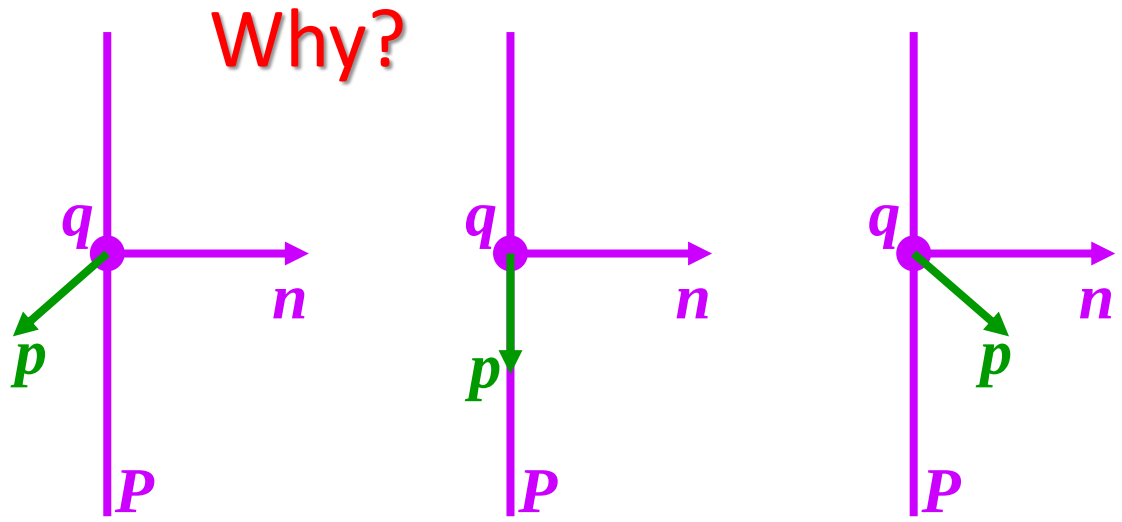
- A test to determine if a point p is on the "inside" half-plane P, defined by q and normal vector n

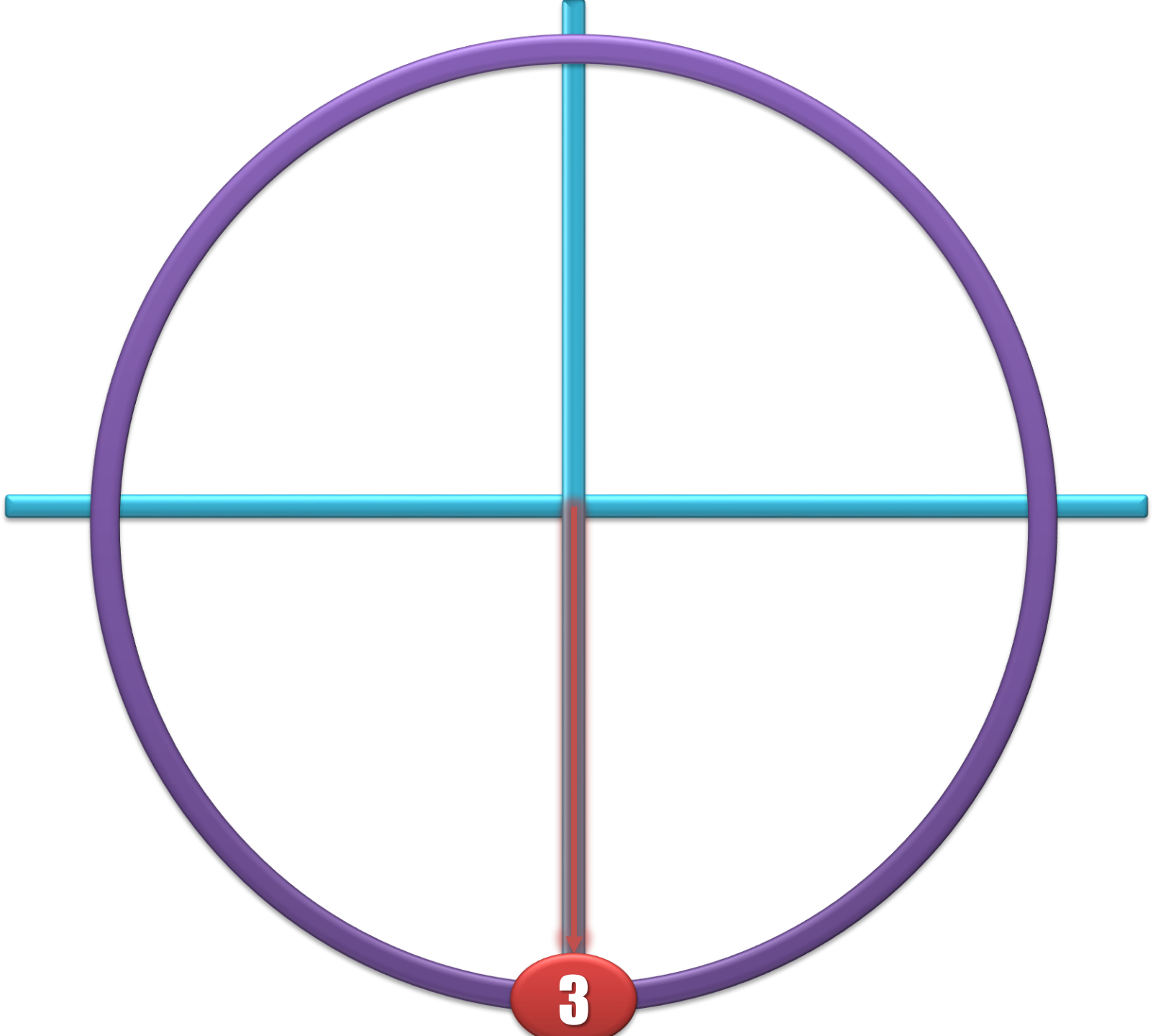
$(p - q) \cdot n < 0$: p inside $\boldsymbol{P}$

$(p - q) \cdot n = 0$: p on $\boldsymbol{P}$

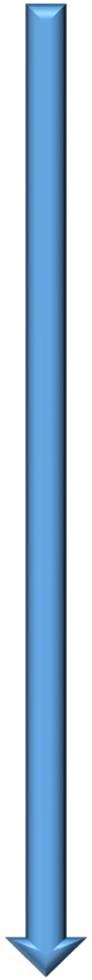
$(p - q) \cdot n > 0$: p outside $\boldsymbol{P}$

Why?




3

Geometry Pipeline Coord.



MC: Modeling Coordinates

Apply Model Transformations

WC: World Coordinates

Viewing Transformations

VC: Viewing Coordinates

Projection Transformation

PC: Projection Coordinates

To canonical std coordinates

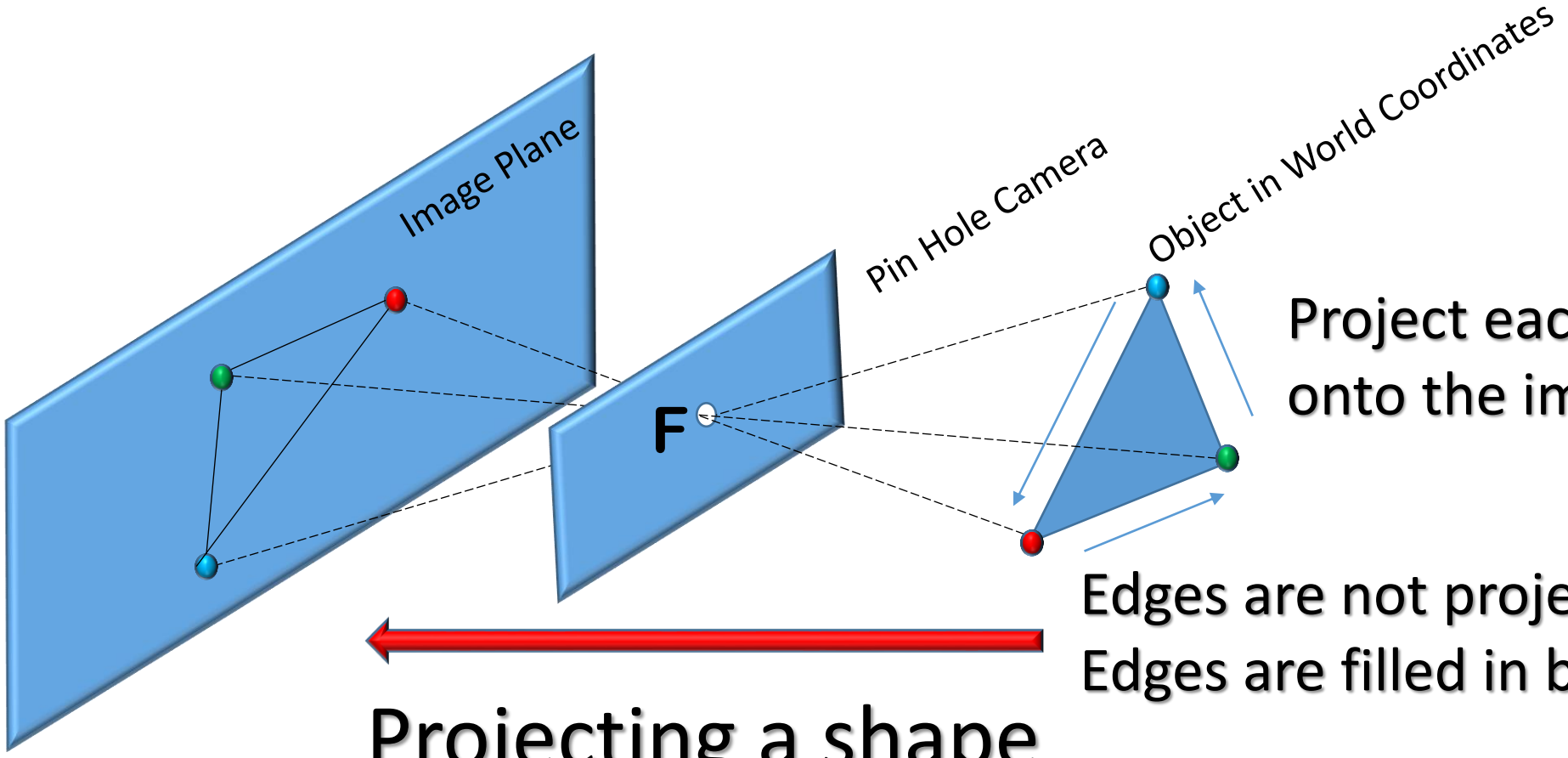
NC: Normalized Coordinates

Clip and rasterize

DC: Device/Display Coordinates

Projections

Image Formation

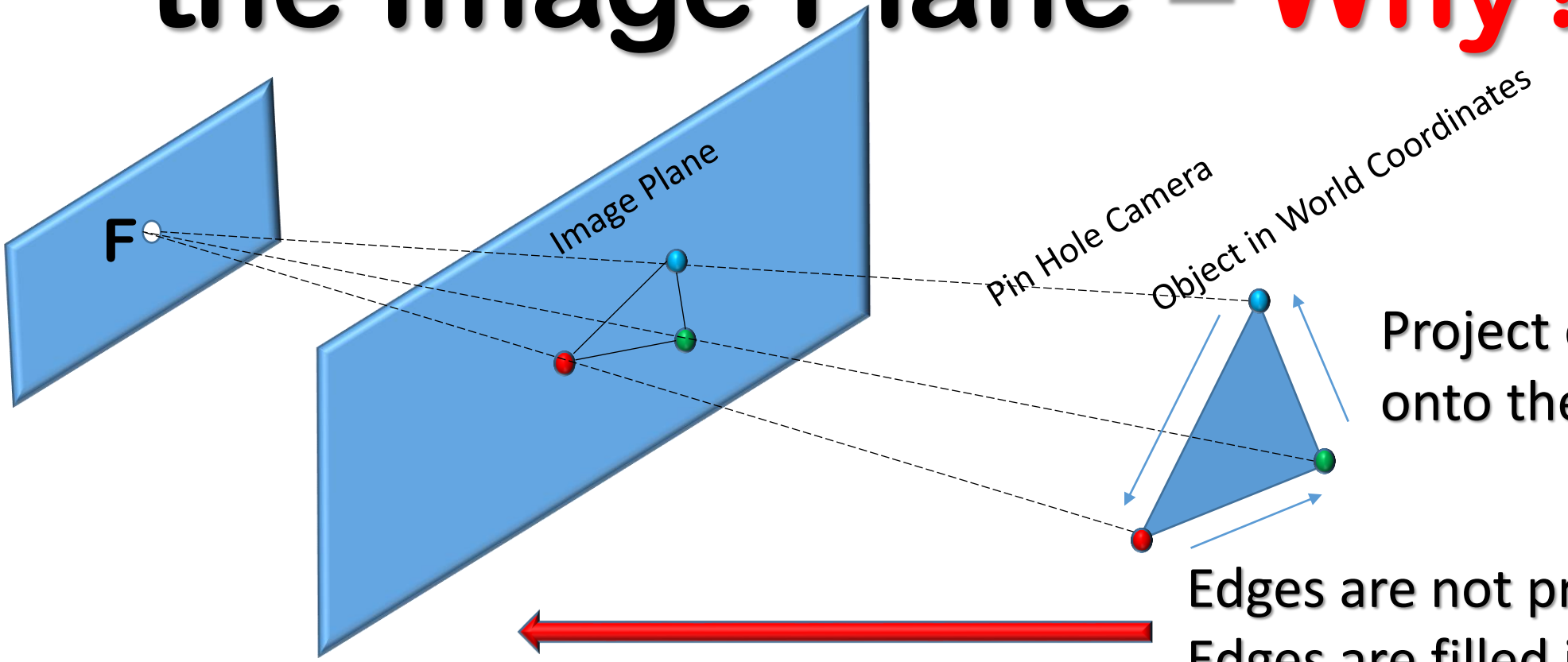


Project each point
onto the image plane

Edges are not projected
Edges are filled in by interpolation

Projecting a shape

Place F (focal point) behind the Image Plane – Why?

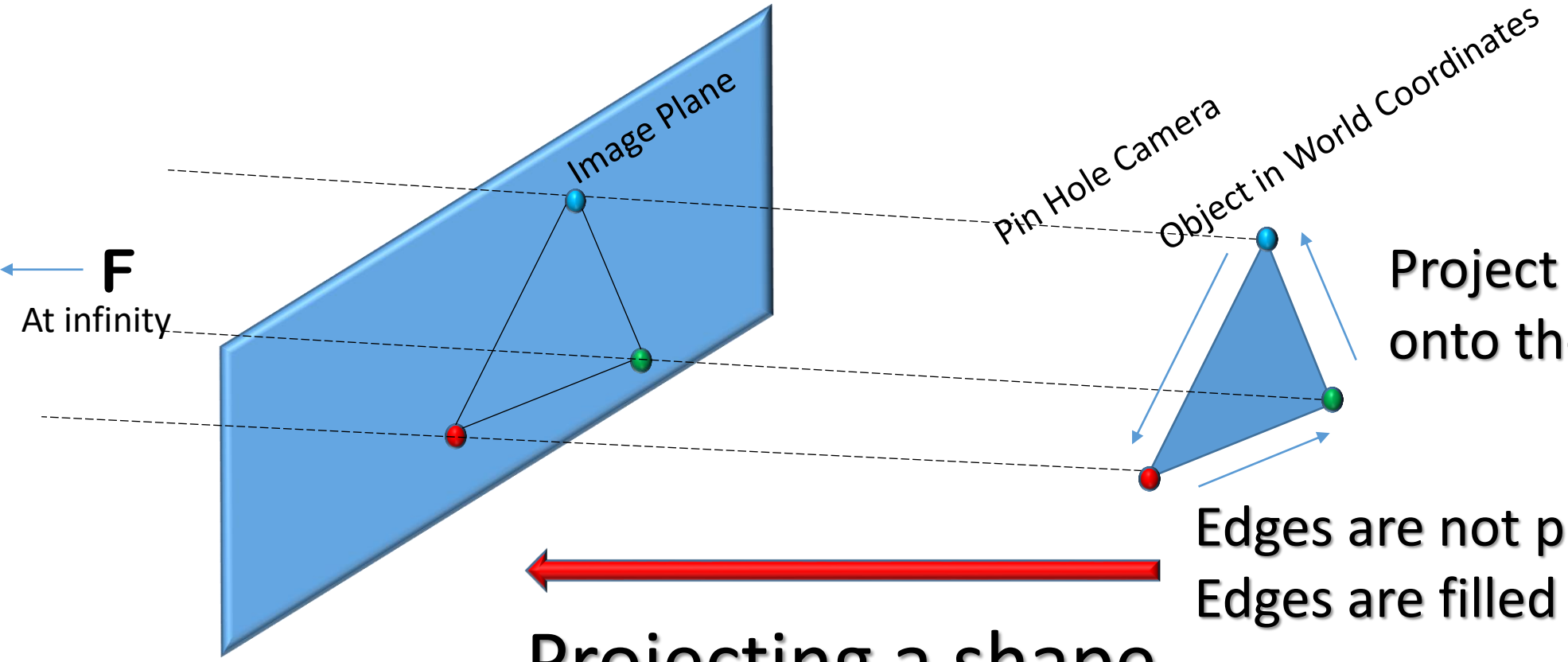


Project each point onto the image plane

Edges are not projected
Edges are filled in by interpolation

Projecting a shape

Orthographic Projection



Project each point
onto the image plane

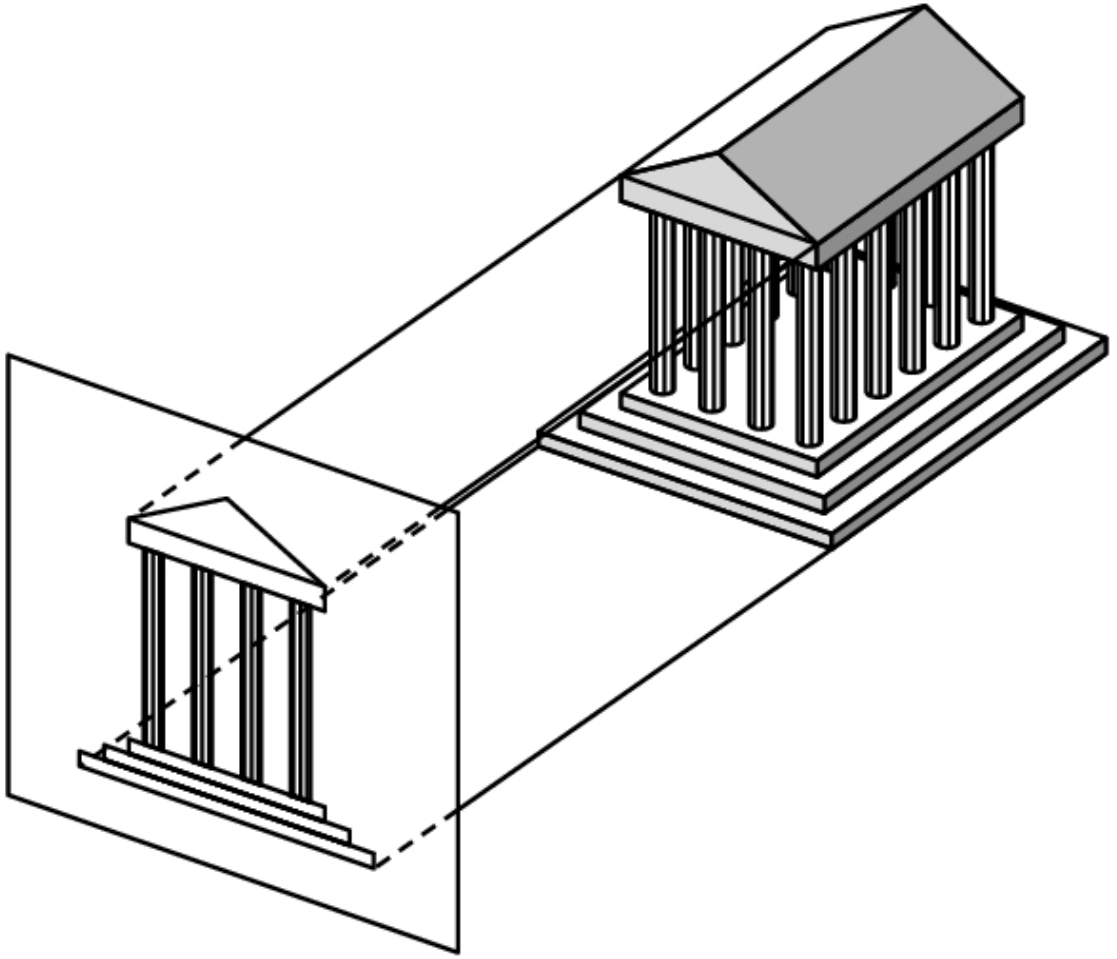
Edges are not projected
Edges are filled in by interpolation

Projecting a shape

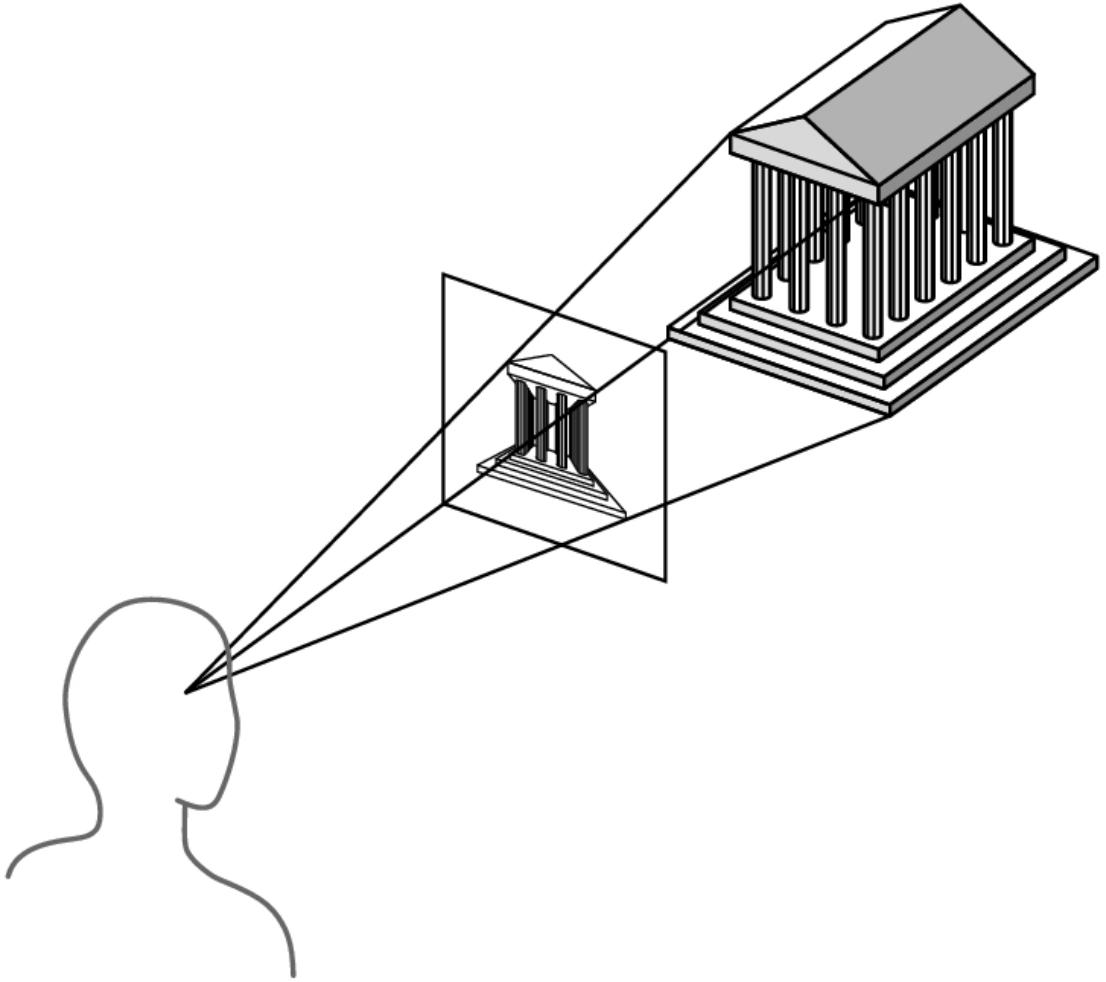
Orthographic Projection

- Focal point is at infinity
- Rays are parallel and orthogonal to the image plane
- When xy-plane is the projection plane: (x, y, z)→(x,y,0)

Orthographic Projection



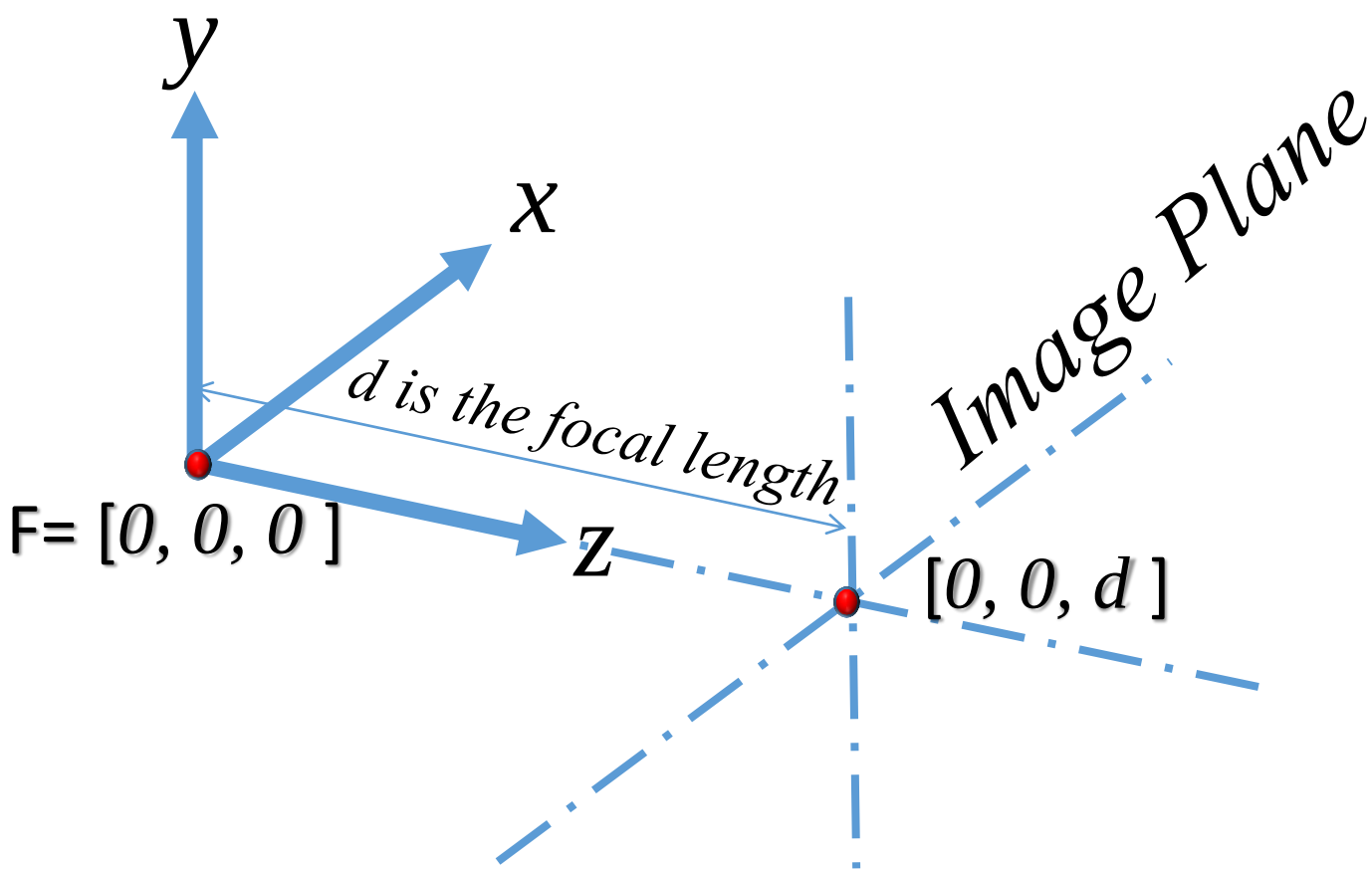
Perspective Projection

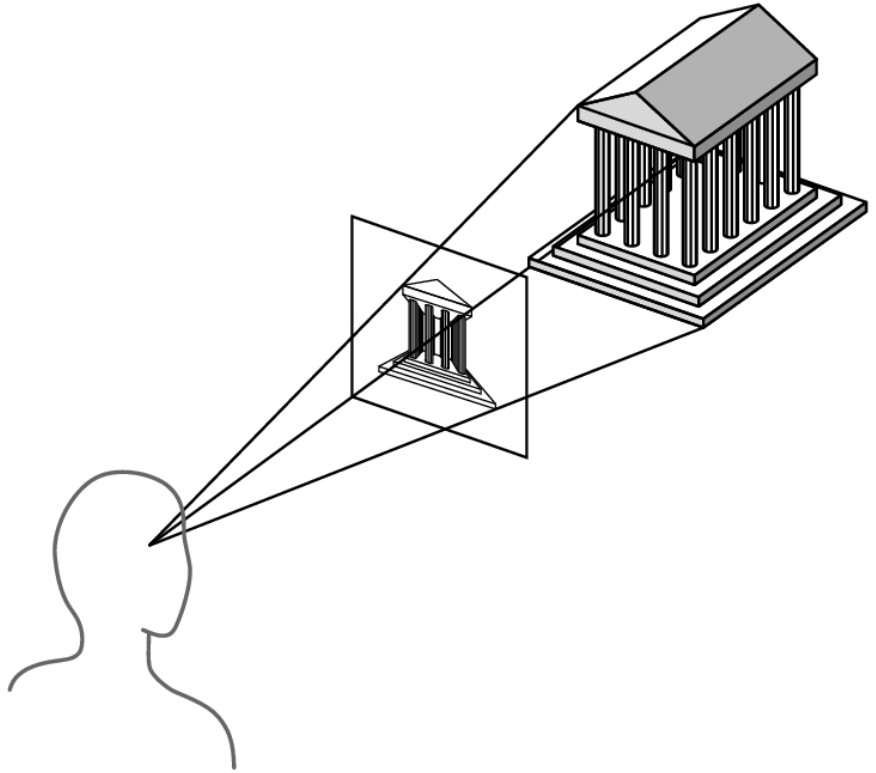


Perspective Projection

- A simple canonical case:
  - Camera looks along the z-axis
  - Focal point is at the origin
  - Image plane is parallel to the xy-plane
  - Image plane is at z = d

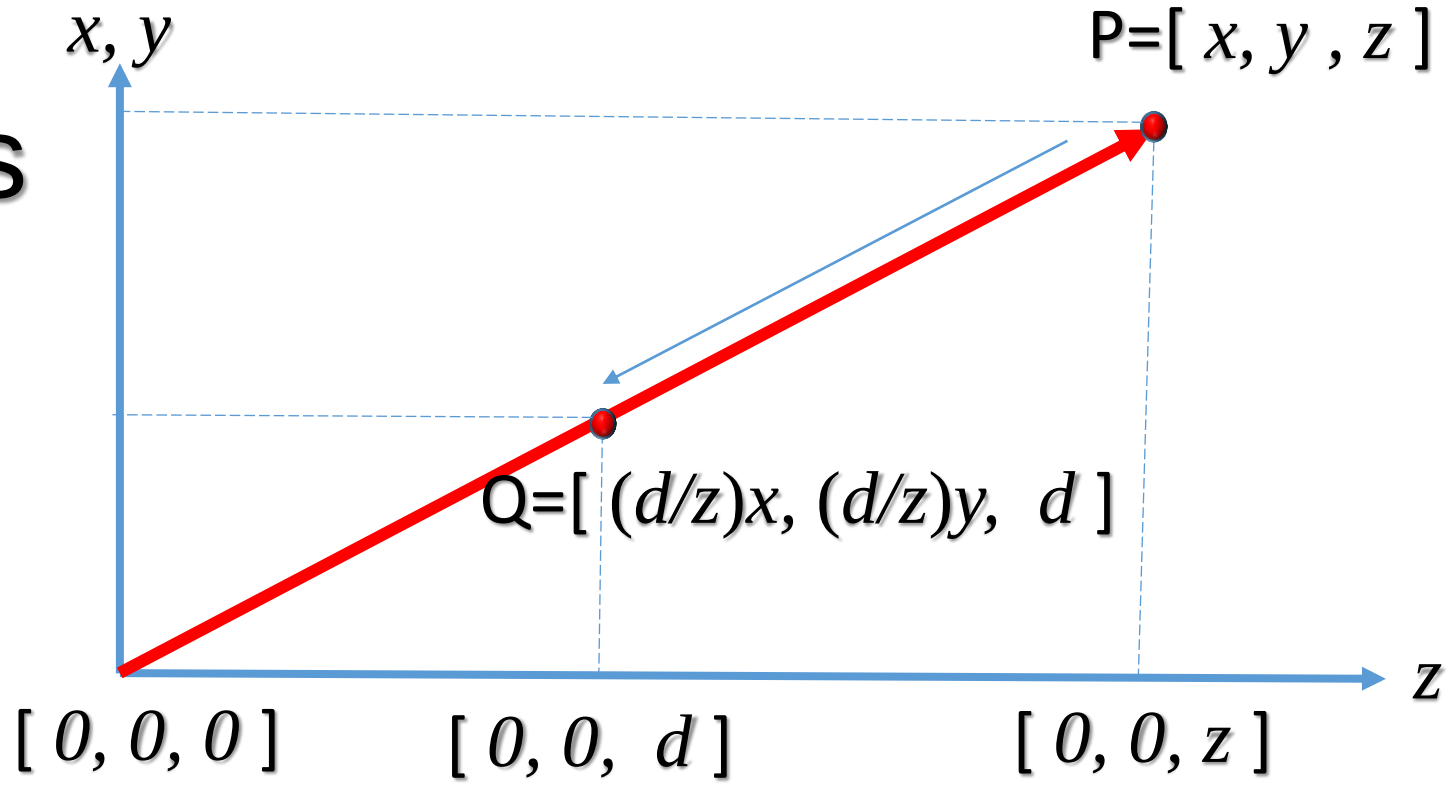
What does it look like?

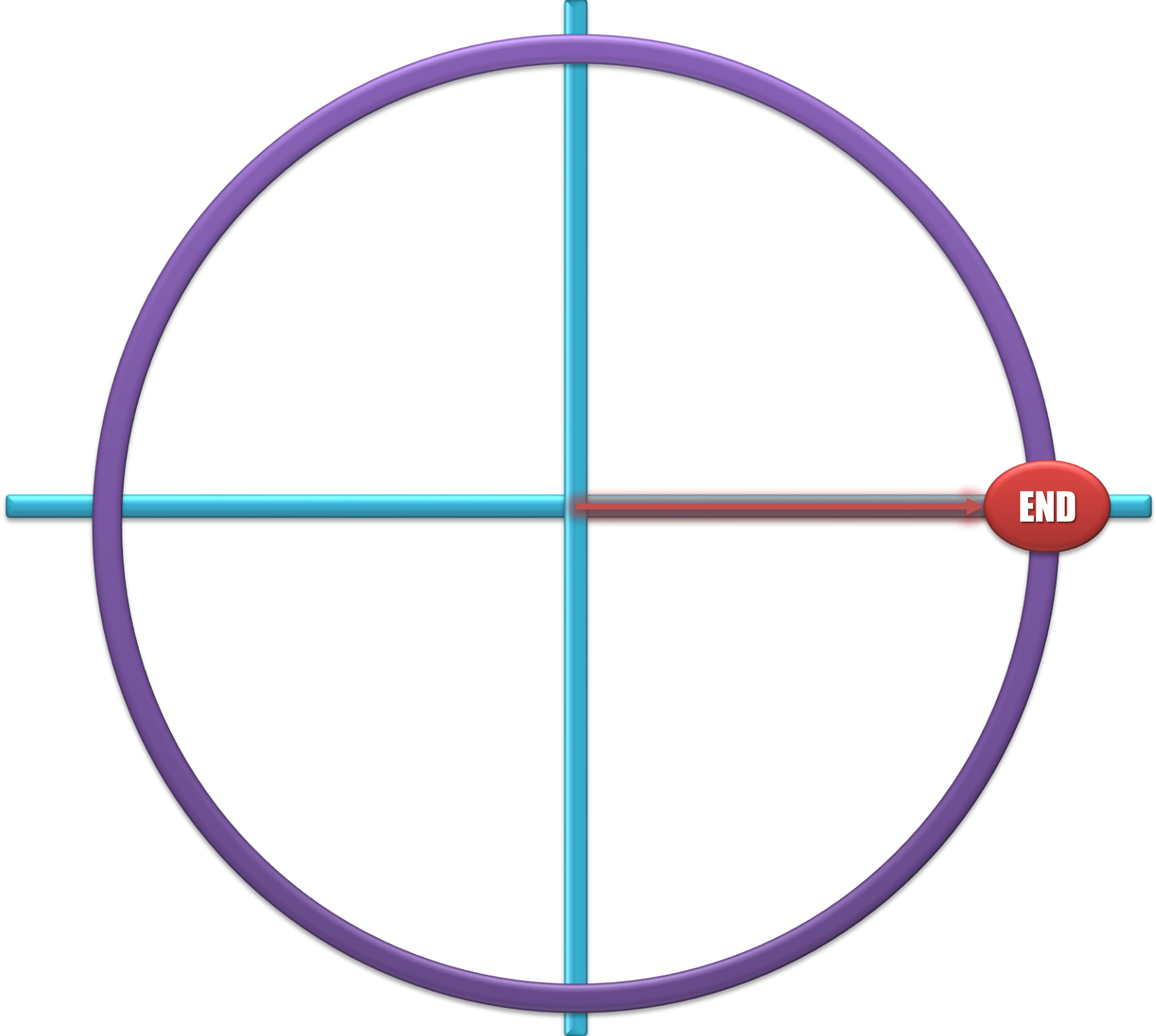




How to compute the Projection Coordinates

- Similar Triangles
  - P[x,y,z]
  - projects to:
  - Q[$(d/z)x$, $(d/z)y$, d]




END