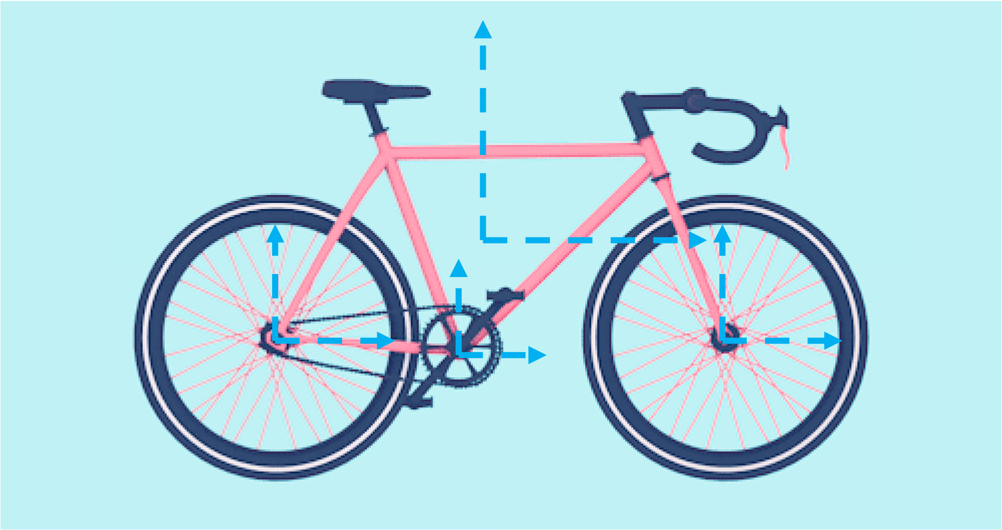
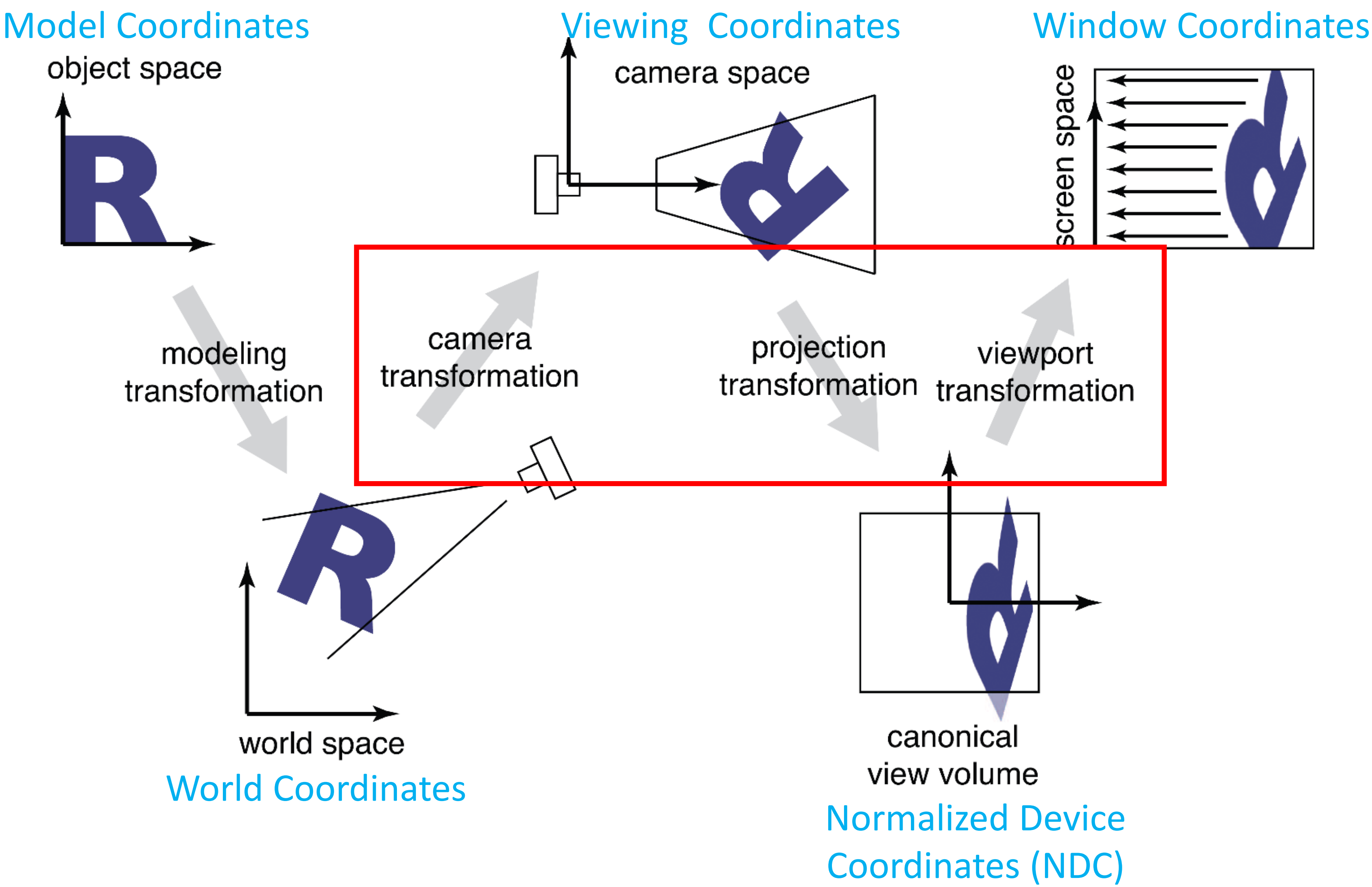
CS100433
2D and 3D Viewing

Junqiao Zhao 赵君峤
Department of Computer Science and Technology
College of Electronics and Information Engineering
Tongji University

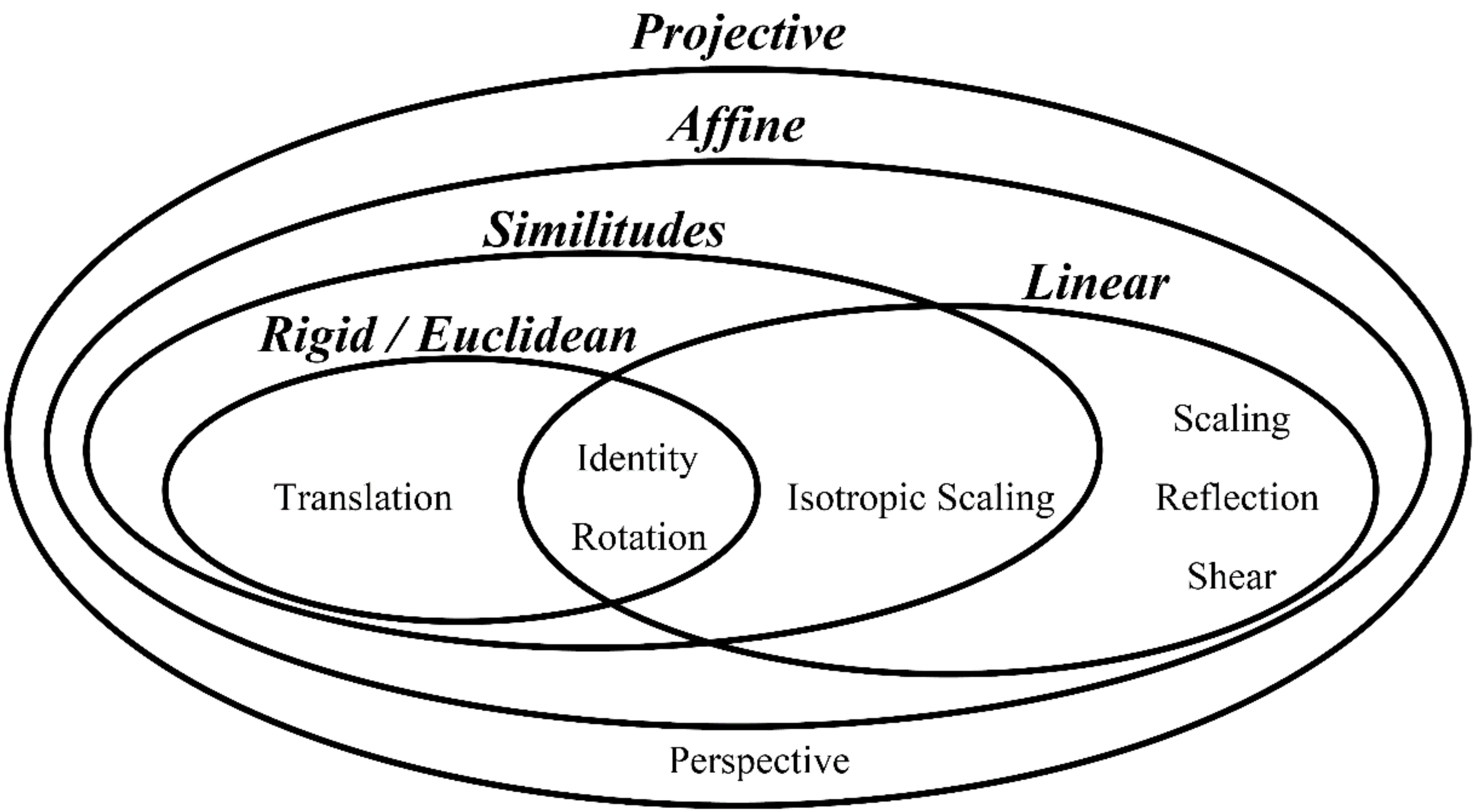
How to **animate** a bicycle?



Viewing Pipeline



Recall Transformations

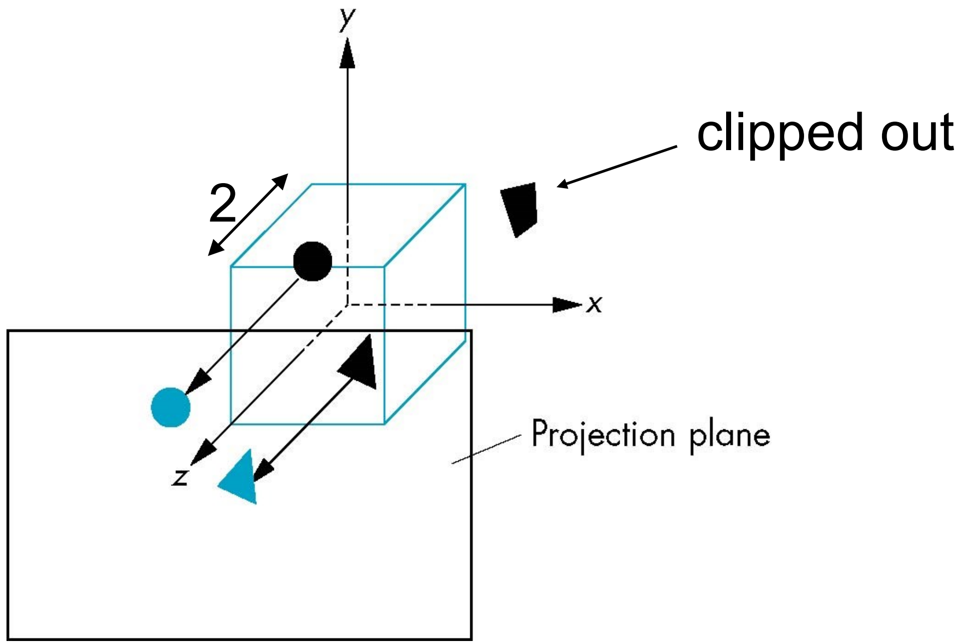


Viewing implementation

- Transform into camera coordinates
- Perform projection into view volume
- Clip geometry outside the clipping volume
- Project into screen coordinates
- Remove hidden surfaces (next lecture)

The Default Viewing

- Convention - the “camera” is located at origin and points in the negative z direction
- The default view volume is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity
- NDC



glm::LookAt(eye, center, up)

- creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector, usually (0, 1, 0)
- Let
  - $f = normalized(eye - center)$
  - $u = UP \times f$
  - $up = normalized(f \times u)$
- $F = \begin{bmatrix} u & up & f & eye \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- $T = F^{-1}$
- Note the camera is looking at the negative z direction in camera space

Moving the Camera Frame

- If we want to visualize object with both positive and negative z values we can either
  - Move the camera in the positive z direction
    - Translate the camera frame
  - Move the objects in the negative z direction
    - Translate the world frame
- So called ModelView matrix
- We can move the camera/model to any desired position by a sequence of rotations and translations

Projection Transformation

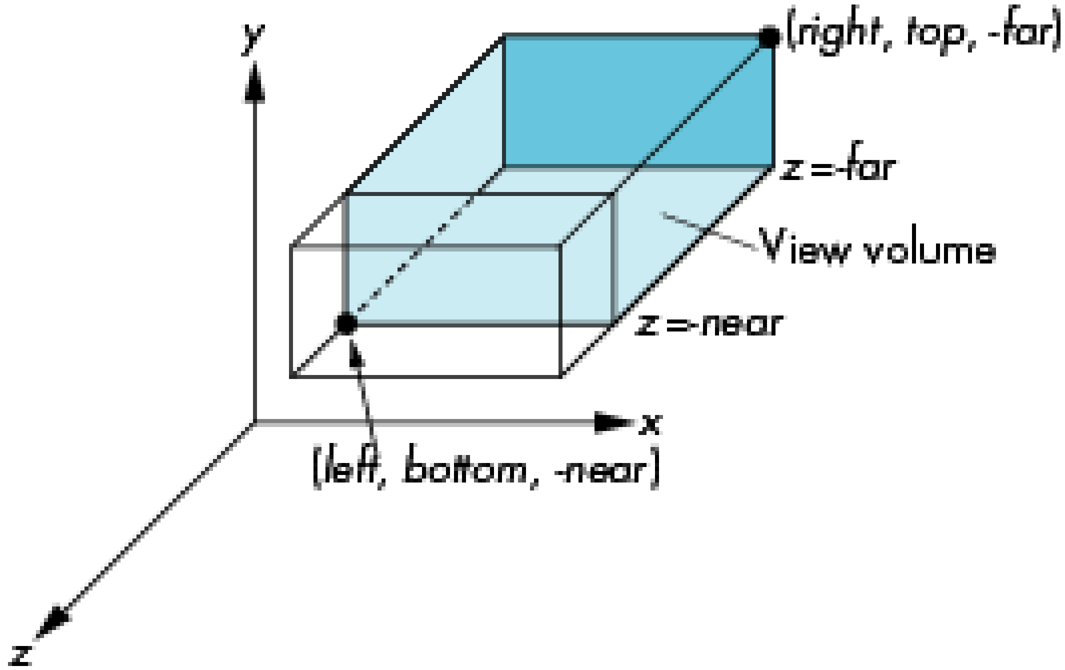
- After the viewing transformation everything are oriented as we would like them to appear in the final image
- All that remains is to project out the depth z: convert the 3D coordinates to 2D
  - Orthographic
  - Perspective

Mathematics of Projection

- Always work in eye space
- Orthographic projection
  - a simple projection: just toss out *z*
  - In practice, we can directly set *z = 0*
  - $$\begin{bmatrix} x \\ y \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$
- Perspective case: scale diminishes with *z*

Orthogonal Projection

`glm::Ortho(left,right,bottom,top,near,far)`



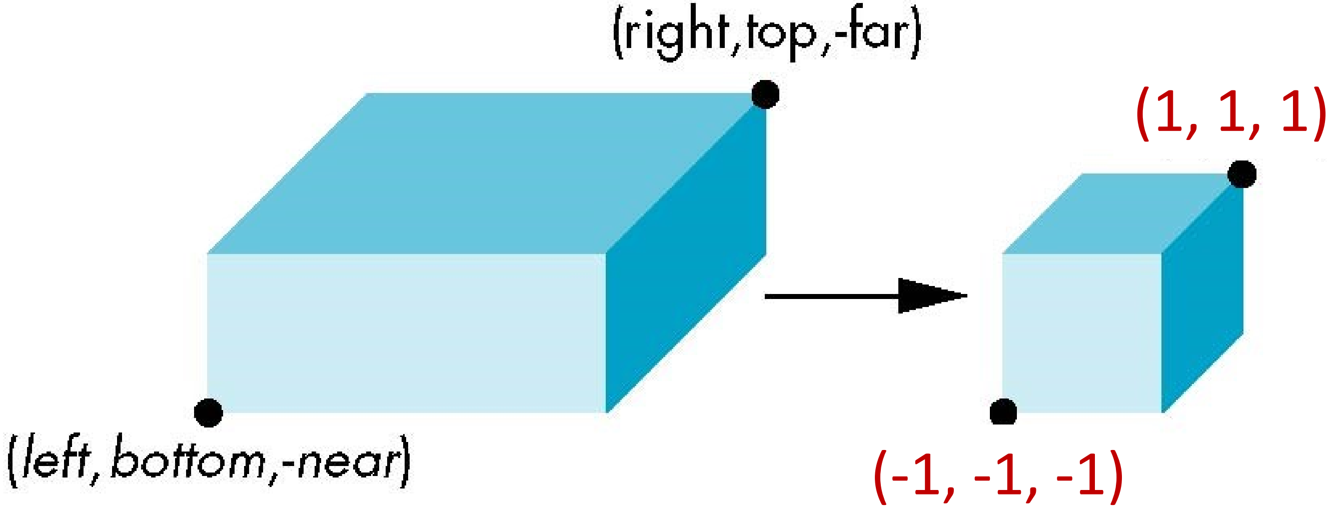
`near` and **`far`** measured distance from eye

Normalization

- Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume
- This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping

Orthogonal Normalization

```
glm::Ortho(left,right,bottom,top,near,far)
```



Orthogonal Matrix

- Two steps
  - Move center to origin

    T(-(left+right)/2, -(bottom+top)/2, (near+far)/2))
  - Scale to have sides of length 2

    S(2/(right-left), 2/(top-bottom), 2/(-far –(- near))

$$\mathbf{P} = \mathbf{ST} = \begin{bmatrix} \frac{2}{right - left} & 0 & 0 & -\frac{right + left}{right - left} \\ 0 & \frac{2}{top - bottom} & 0 & -\frac{top + bottom}{top - bottom} \\ 0 & 0 & \frac{2}{near - far} & -\frac{far + near}{far - near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Final Projection

- Set $z = 0$
- Equivalent to the homogeneous coordinate transformation

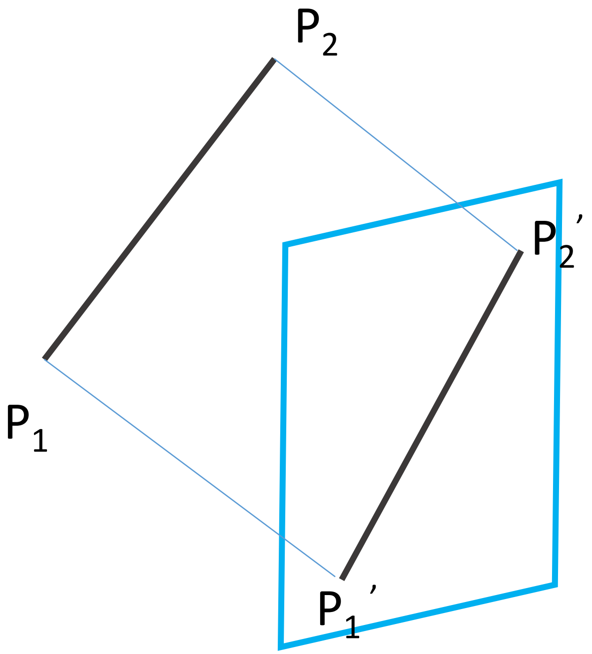
$$\mathbf{M}_{\text{orth}} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

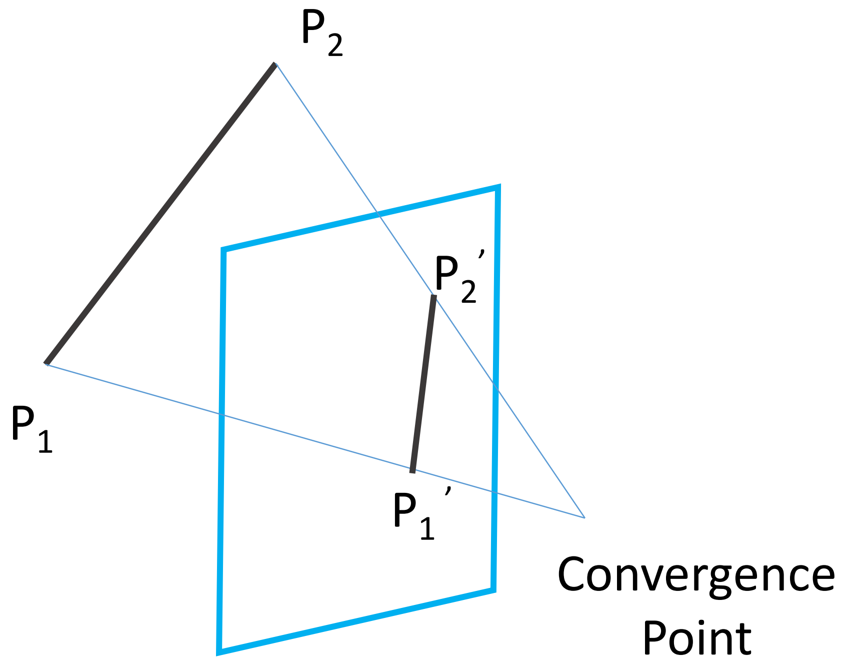
- Hence, general orthogonal projection in 4D is

$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{ST}$

- Questions?

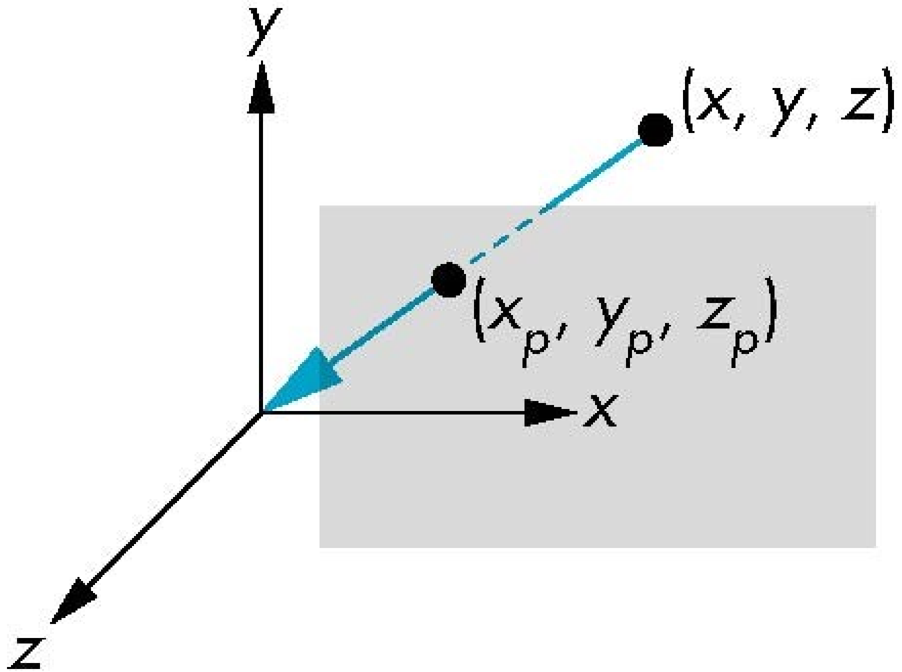
Perspective Projection





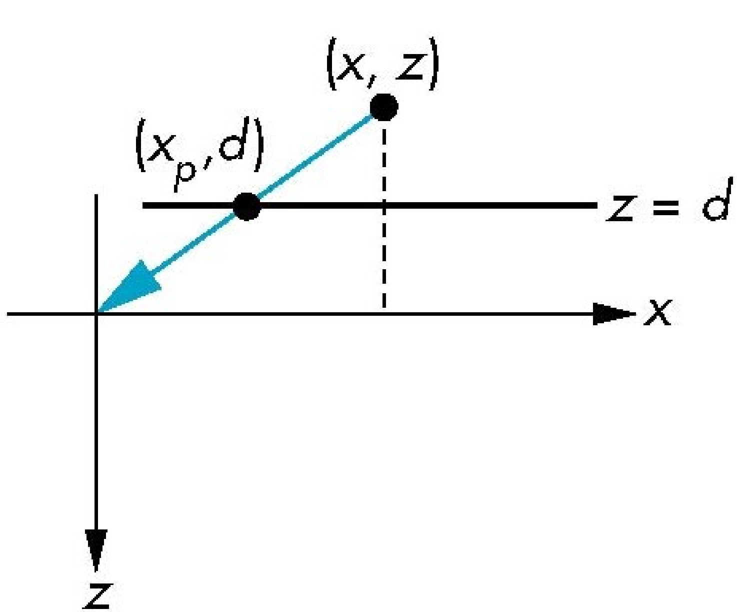
Simple Perspective

- Center of projection at the origin
- Projection plane $z = d$, $d < 0$

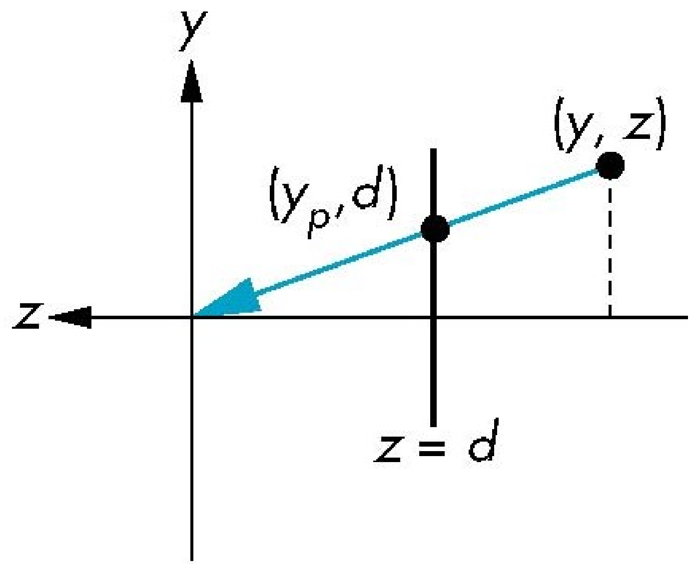


Perspective Equations

Consider top and side views



$\frac{x}{z/d}$



$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

Homogeneous Coordinate Form

- $$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot d/z \\ y \cdot d/z \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & {}^{1}\!/_{d} & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

homogenize

Perspective Division

- $w \neq 1$, so we must divide by w to return from homogeneous coordinates
- This *perspective division* yields

$$x_p = \frac{x}{z/d} \qquad y_p = \frac{y}{z/d} \qquad z_p = d$$

the desired perspective equations

Alternate Perspective Projection

- Center of projection at *z = d*
- Projection plane *z = 0*
- $$\begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix} = \begin{bmatrix} x \cdot d/(z+d) \\ y \cdot d/(z+d) \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 0 \\ (z+d)/d \end{bmatrix} =$$

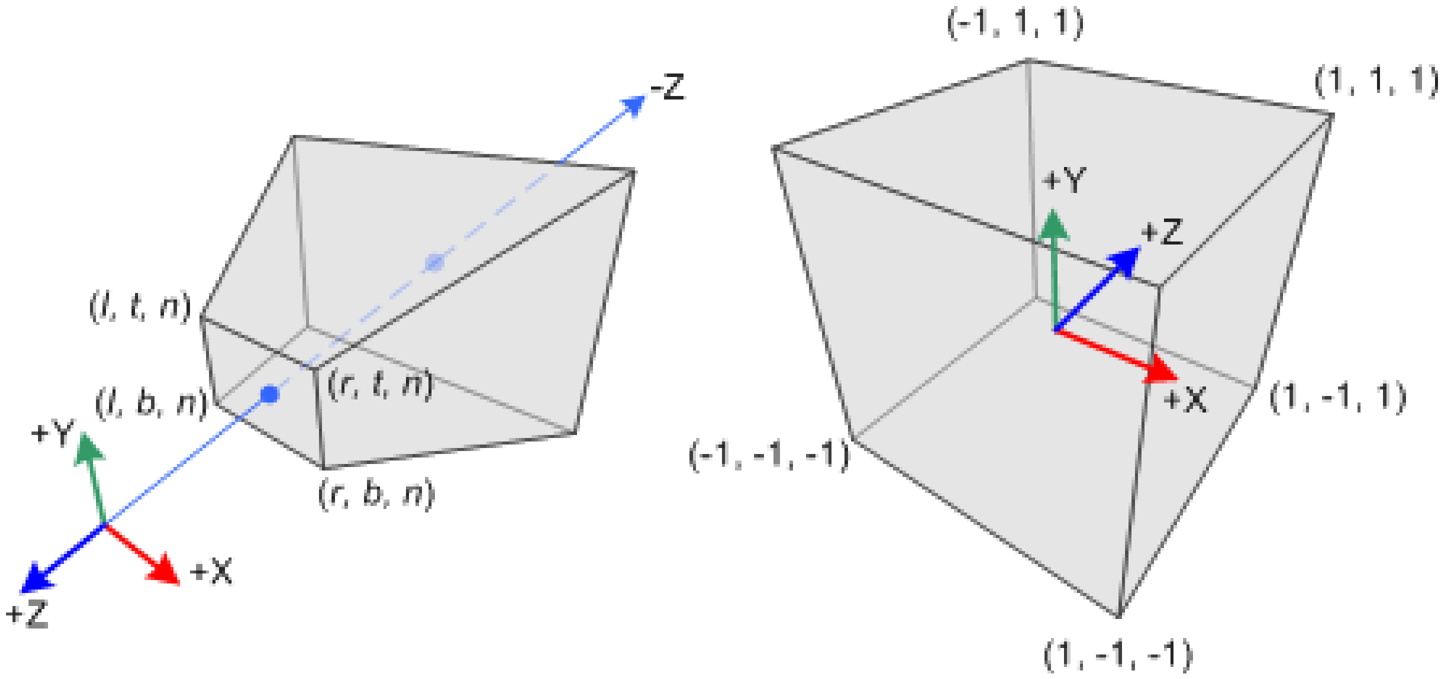
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & {}^{1}/_{d} & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

To the Limit, as $d \to \infty$

- The perspective projection matrix is simply an orthographic projection
- $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & {}^{1}/_{d} & 1 \end{bmatrix} \to \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

- Questions?

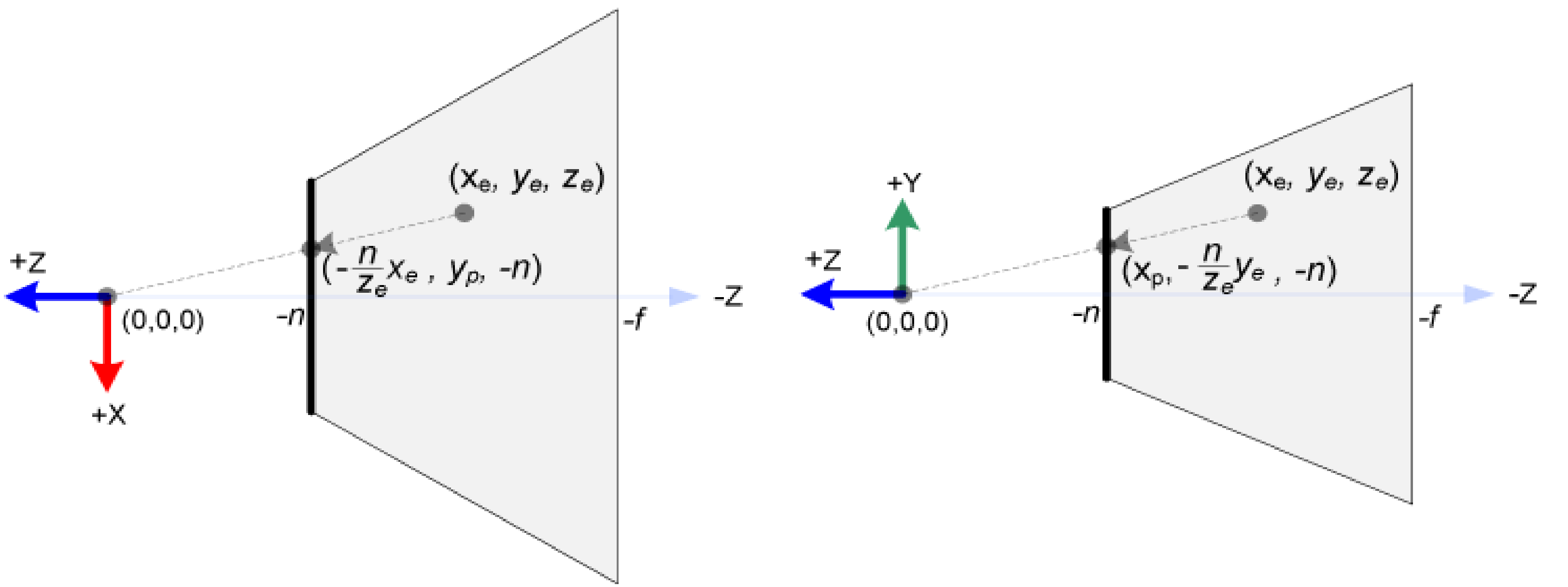
Normalize Perspective Transformation



http://www.songho.ca/opengl/gl_projectionmatrix.html

Normalize Perspective Transformation

- $x_p = -\frac{n}{z_e} x_e$
- $y_p = -\frac{n}{z_e} y_e$



http://www.songho.ca/opengl/gl_projectionmatrix.html

Normalize Perspective Transformation

- Normalize $x_p\ and\ y_p$ instead
- Using the same "trick" as we did for orthogonal projection
- $x_n = \frac{2}{r-l} \cdot x_p - \frac{r+l}{r-l}$
- $y_n = \frac{2}{t-b} \cdot \mathrm{y}_p - \frac{t+b}{t-b}$
- Substitute $x_e\ and\ y_e$ into the equations
- $x_n = (\frac{2n}{r-l} \cdot x_e + \frac{r+l}{r-l} \cdot z_e)/-z_e$
- $y_n = (\frac{2n}{t-b} \cdot \mathrm{y}_e + \frac{t+b}{t-b} \cdot z_e)/-z_e$

http://www.songho.ca/opengl/gl_projectionmatrix.html

Normalize Perspective Transformation

- $$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ ? & ? & ? & ? \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_\mathrm{e} \\ z_\mathrm{e} \\ 1 \end{pmatrix}$$

- $$\begin{pmatrix} x_n \\ y_n \\ z_n \\ 1 \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \\ 1 \end{pmatrix}$$

http://www.songho.ca/opengl/gl_projectionmatrix.html

Normalize Perspective Transformation

- $\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & A & B \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_{\mathrm{e}} \\ z_{\mathrm{e}} \\ 1 \end{pmatrix}$
- $\mathrm{z_c} = A \cdot z_e + B$
- $\mathrm{z}_n = \frac{z_c}{-z_e}$
- How to solve A and B?

http://www.songho.ca/opengl/gl_projectionmatrix.html

Normalize Perspective Transformation

- $\mathrm{z_c} = A \cdot z_e + B$
- $\mathrm{z}_n = \frac{z_c}{-z_e}$
- $(\mathrm{z}_e, z_n)\text{: } (-\mathrm{n}, -1), (-\mathrm{f}, 1)$
- A=$-\frac{f+n}{f-n}$
- B=$-\frac{2fn}{f-n}$

http://www.songho.ca/opengl/gl_projectionmatrix.html

Normalize Perspective Transformation

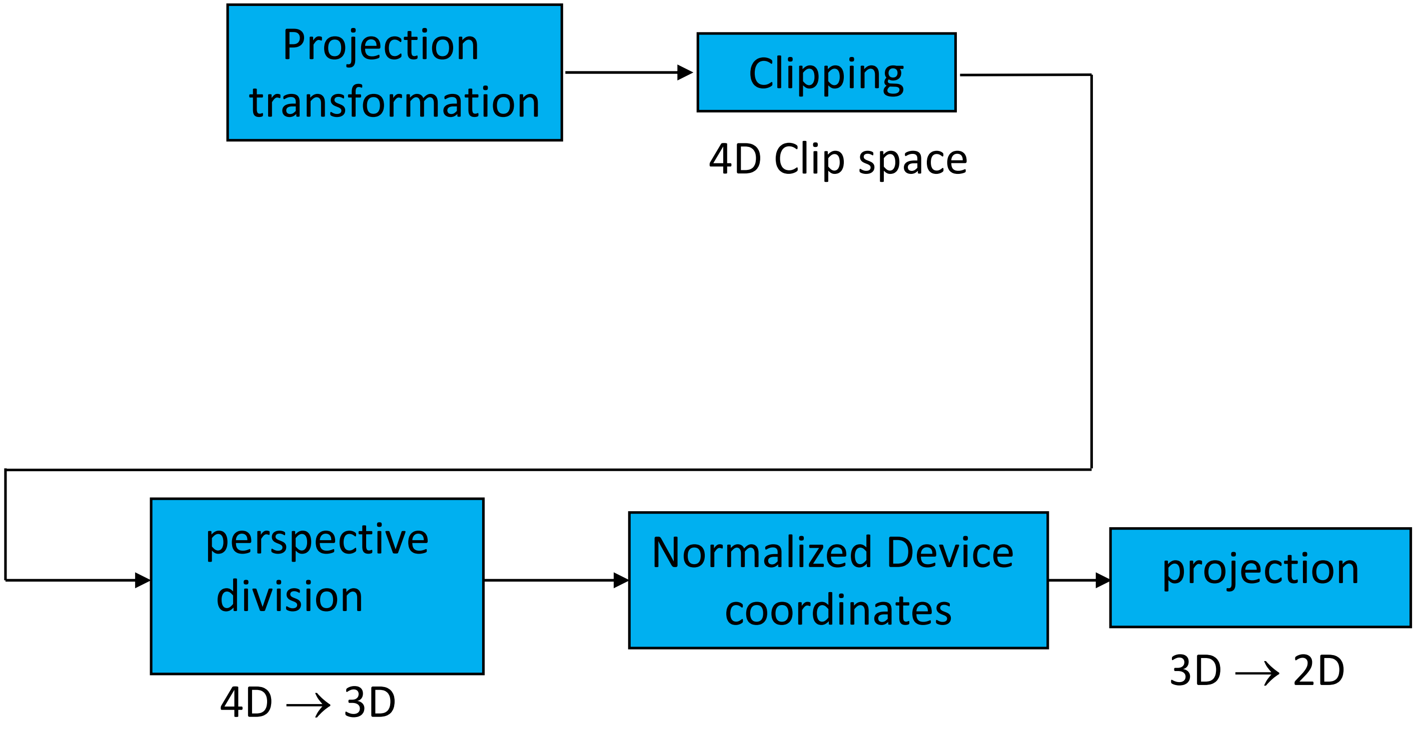
- $$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_e \\ z_e \\ 1 \end{pmatrix}$$

$\mathbf{P} = \mathbf{M}_{\text{orth}}\mathbf{M}_{\text{pers}}$

http://www.songho.ca/opengl/gl_projectionmatrix.html

- Questions？

Clipping



Projection transformation
Clipping
4D Clip space
perspective division
4D → 3D
Normalized Device coordinates
projection
3D → 2D

Clipping-When?

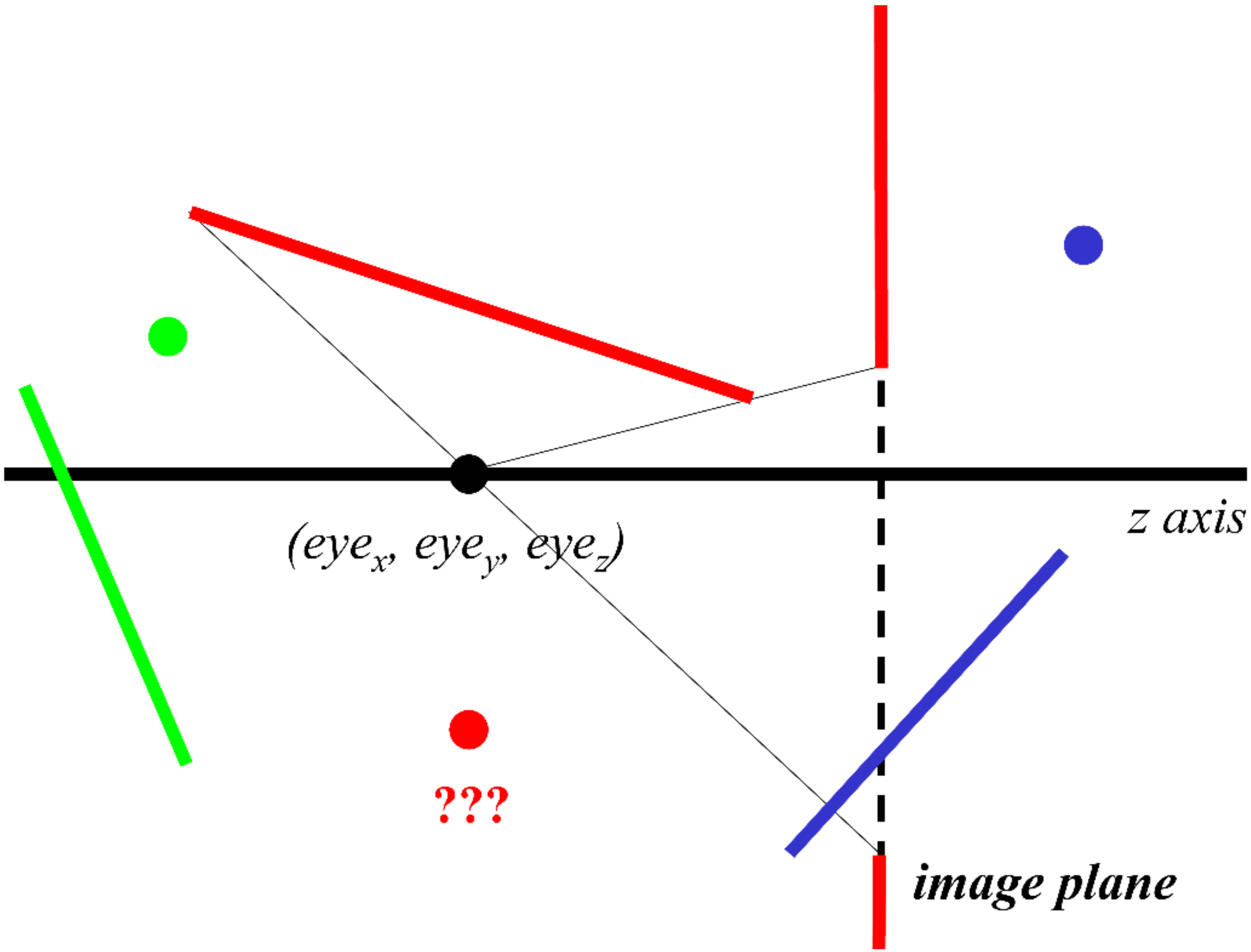
- Before projection transform
  - Use the equation of 4 lines (2D), 6 planes (3D)
  - Natural
- In homogenous clip space
  - 4D space
  - In canonical space, independent of camera and viewport
  - The Simplest to implement, Why?
- In NDC, after perspective division
  - Problematic!

Normalize Perspective Transformation

- $$\begin{pmatrix} x_c \\ y_c \\ z_c \\ w_c \end{pmatrix} = \begin{pmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix} \begin{pmatrix} x_e \\ y_\mathrm{e} \\ z_\mathrm{e} \\ 1 \end{pmatrix}$$
- $$\begin{pmatrix} x_n \\ y_n \\ z_n \\ w_n \end{pmatrix} = \begin{pmatrix} x_c/w_c \\ y_c/w_c \\ z_c/w_c \\ 1 \end{pmatrix}, w_c = -z_e$$

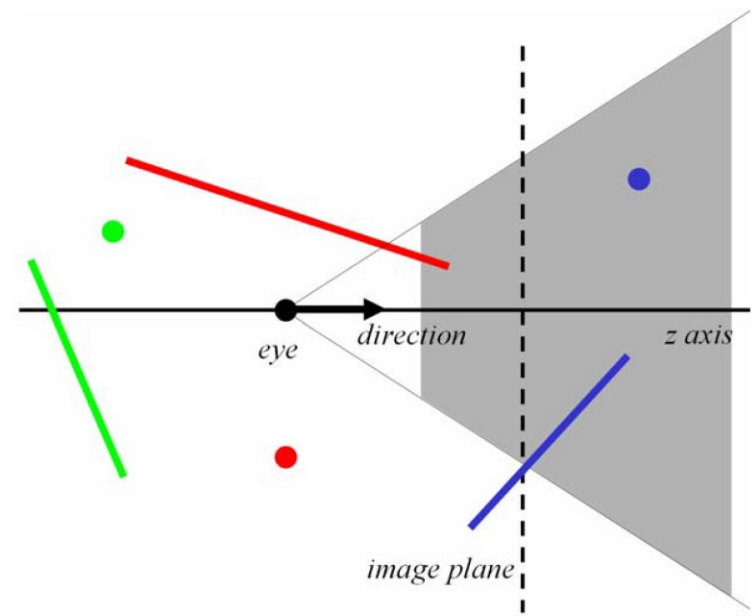
http://www.songho.ca/opengl/gl_projectionmatrix.html

What if the z is $\leq eye_z$



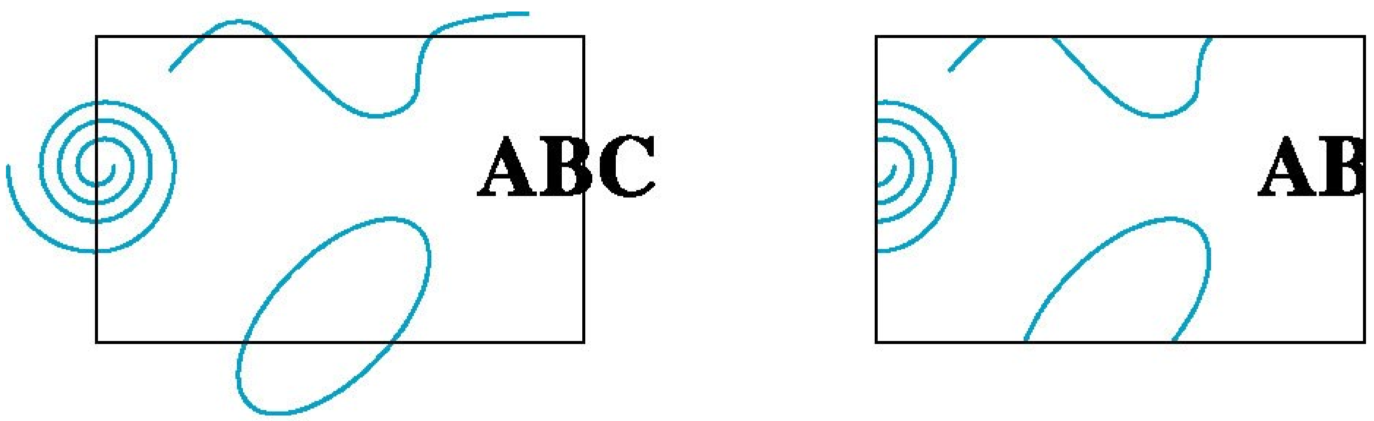
Clipping-Why?

- Avoid degeneracies
  - Do not draw things behind the eye
  - Avoid division by 0
- Efficiency
  - Do not waste time on objects outside the boundary
- Other applications
  - CSG Boolean operations
  - Hidden-surface removal
  - Shadows

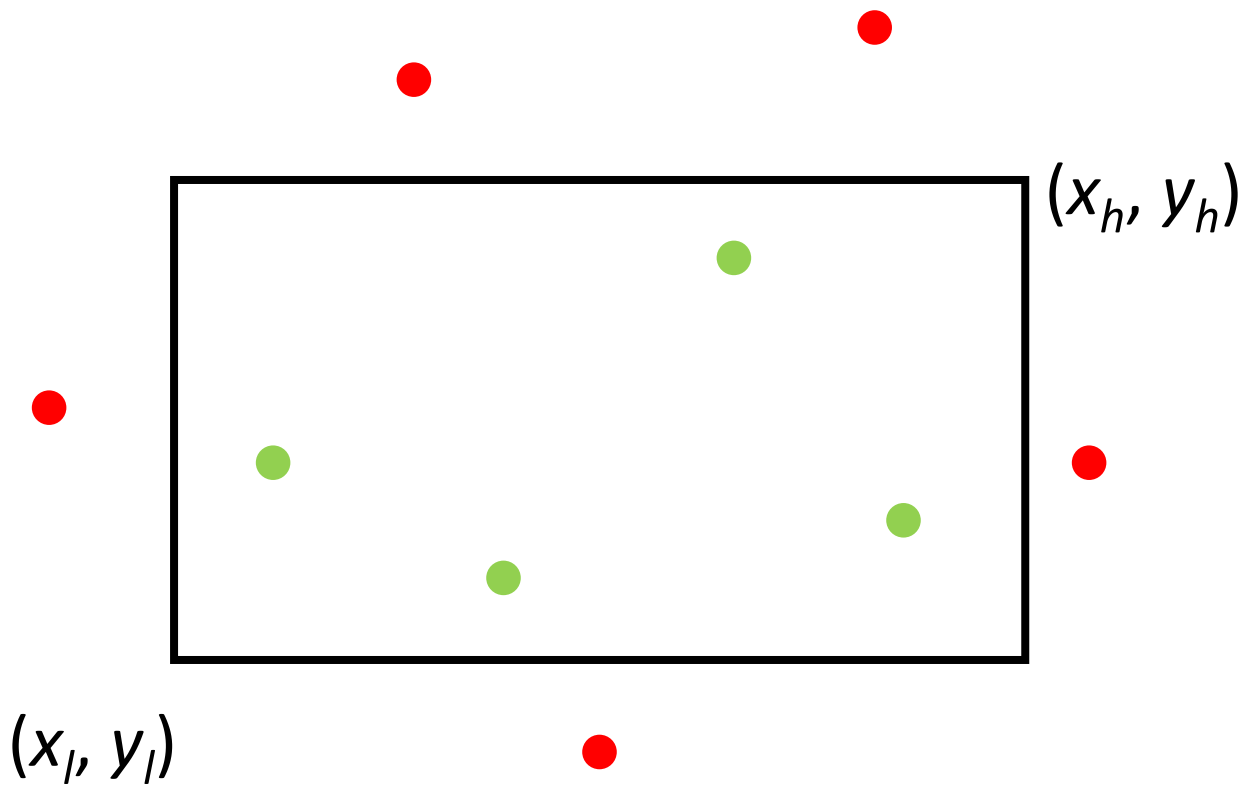


Clipping

- 2D against clipping window
- 3D against clipping volume
- Easy for line segments polygons
- Hard for curves and text
  - Convert to lines and polygons first

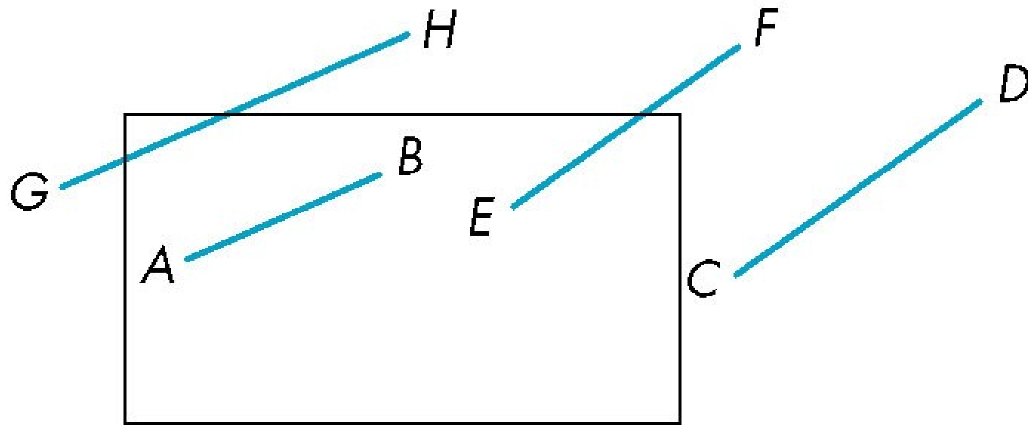


Clipping Points in 2D



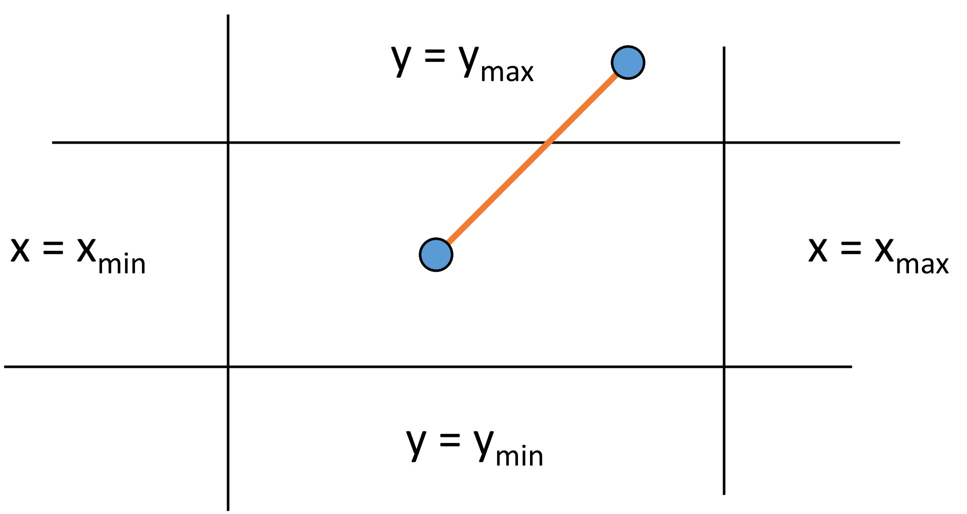
Clipping 2D Line Segments

- Brute force approach: compute intersections with all sides of clipping window
  - Inefficient: one division per intersection



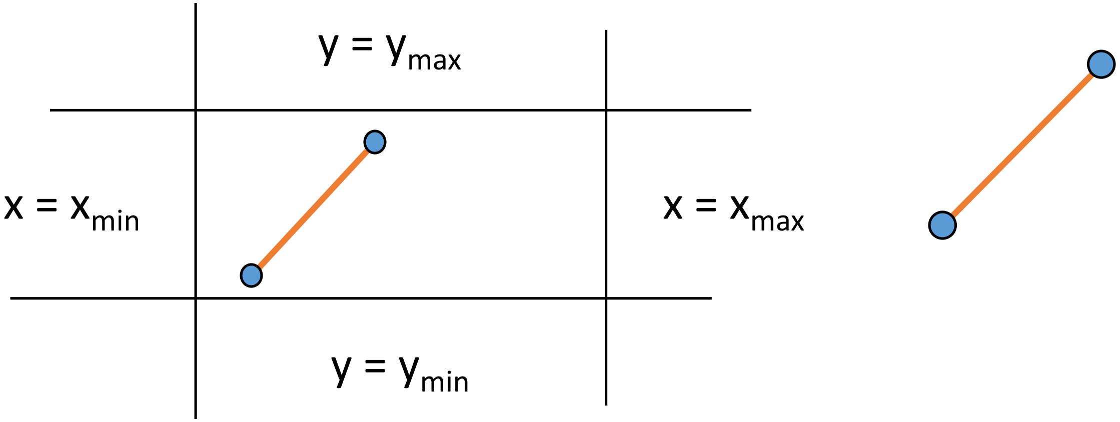
Cohen-Sutherland Algorithm

- Idea: eliminate as many cases as possible without computing intersections
- Start with four lines that determine the sides of the clipping window



The Cases

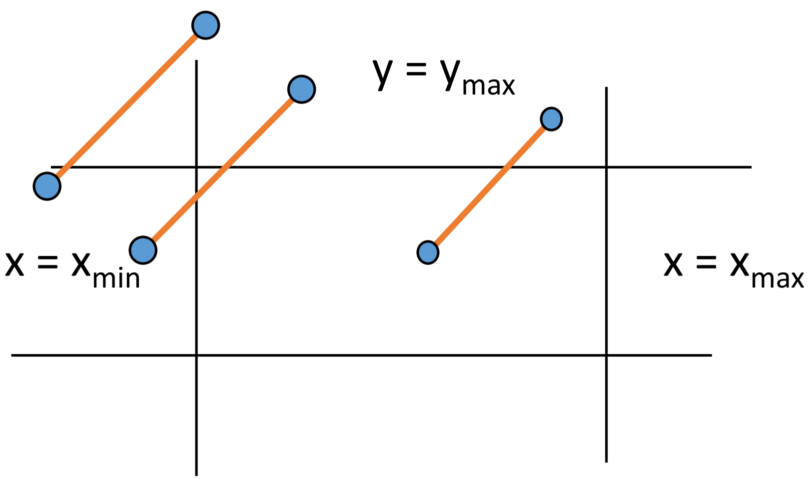
- Case 1: both endpoints of line segment inside all four lines
  - Draw (accept) line segment as is



- Case 2: both endpoints outside all lines and on same side of a line
  - Discard (reject) the line segment

The Cases

- Case 3: One endpoint inside, one outside
  - Must do at least one intersection
- Case 4: Both outside, but on different side of a line
  - May have part inside
  - Must do at least one intersection

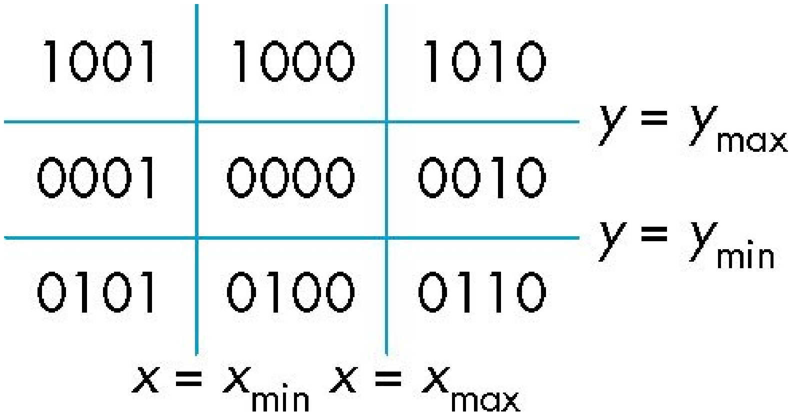


Defining Outcodes

- For each endpoint, define an outcode

$$b_0 b_1 b_2 b_3$$

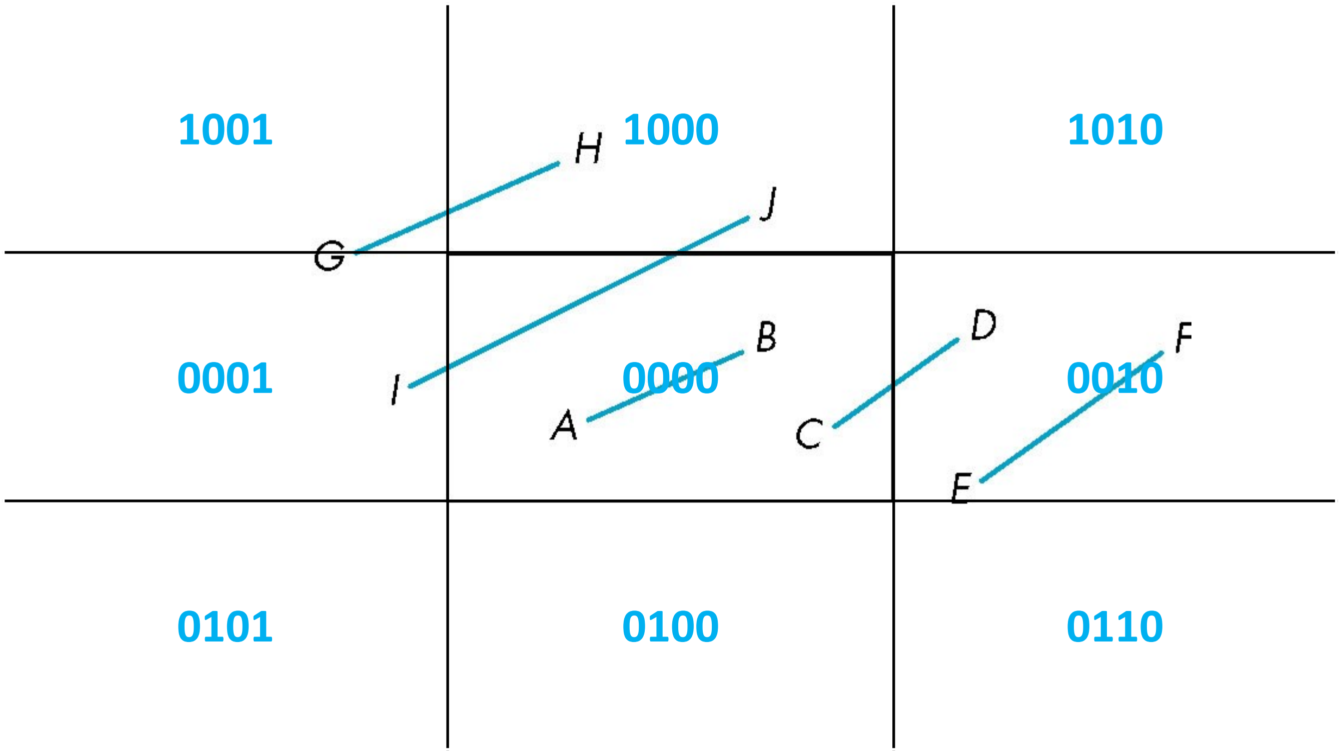
$b_0 = 1$ if $y > y_{max}$, 0 otherwise
$b_1 = 1$ if $y < y_{min}$, 0 otherwise
$b_2 = 1$ if $x > x_{max}$, 0 otherwise
$b_3 = 1$ if $x < x_{min}$, 0 otherwise



- Outcodes divide space into 9 regions
- Computation of outcode requires at most 4 subtractions

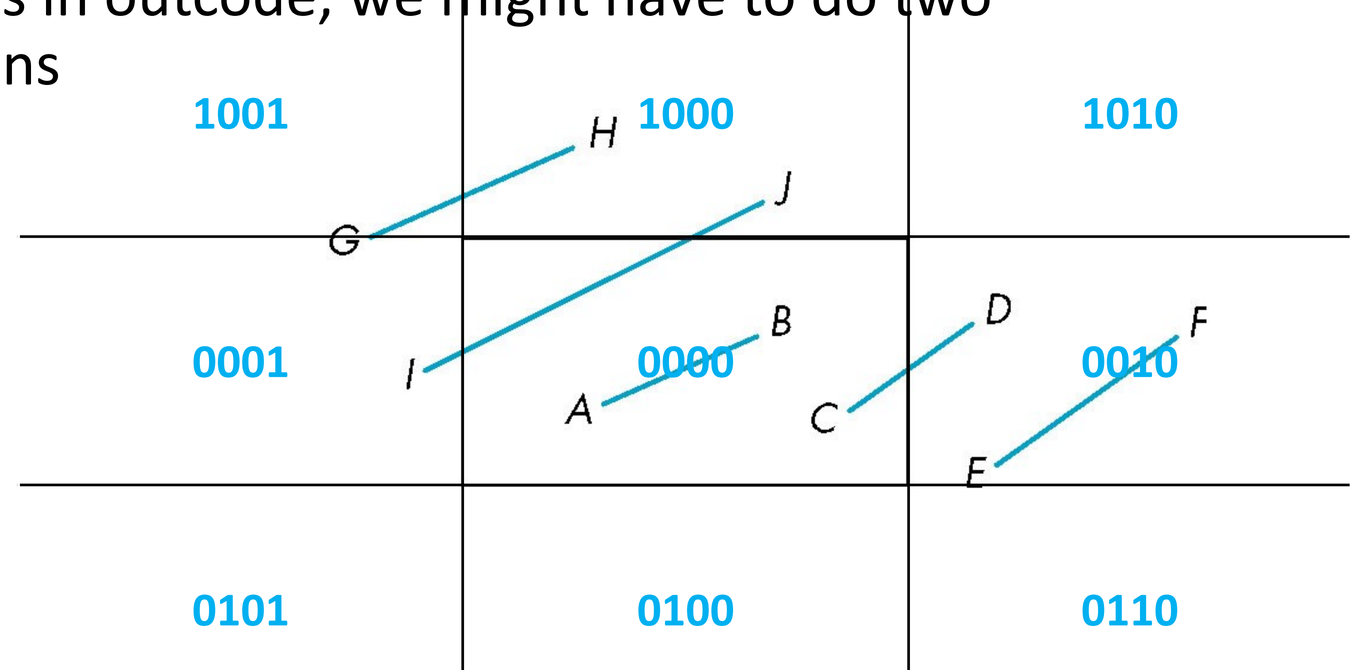
Using Outcodes

- Consider the 5 cases below
- AB: outcode(A) = outcode(B) = 0
  - Accept line segment



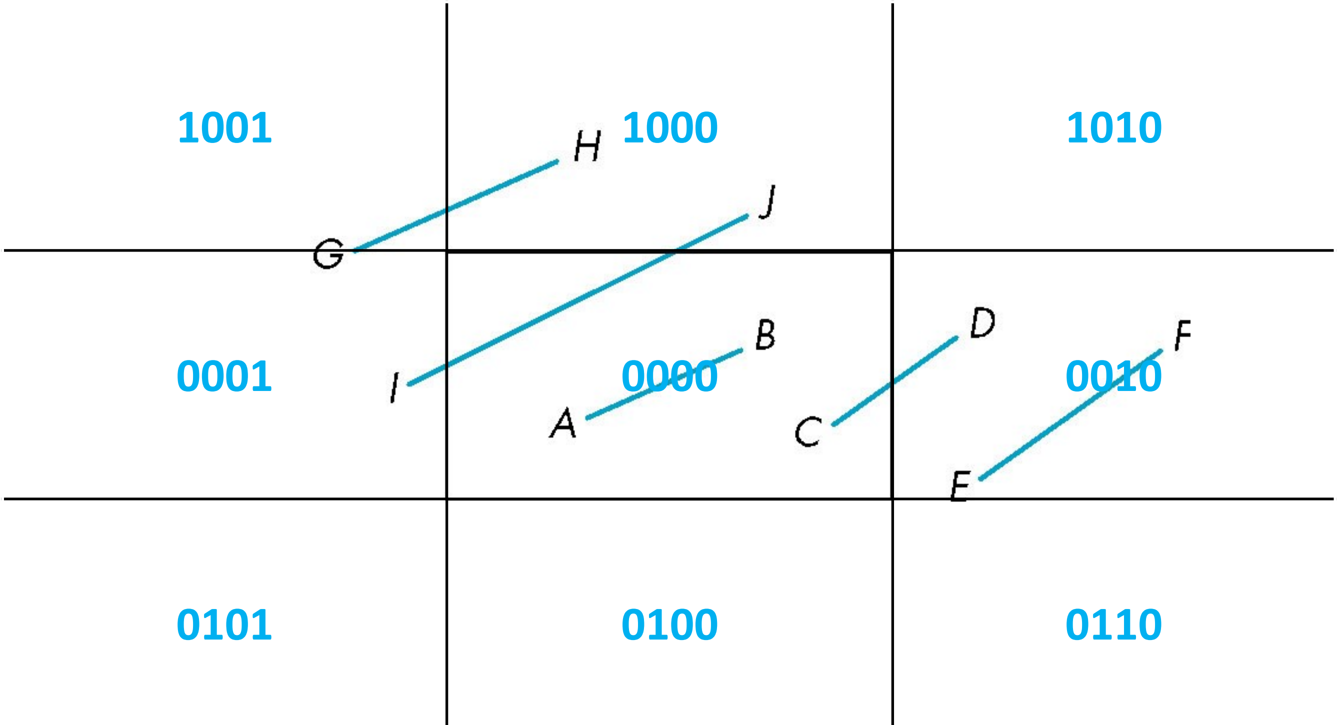
Using Outcodes

- CD: outcode (C) = 0, outcode(D) $\neq$ 0
  - Compute intersection
  - Location of 1 in outcode(D) determines which edge to intersect with
  - Note if there were a segment from A to a point in a region with 2 ones in outcode, we might have to do two intersections



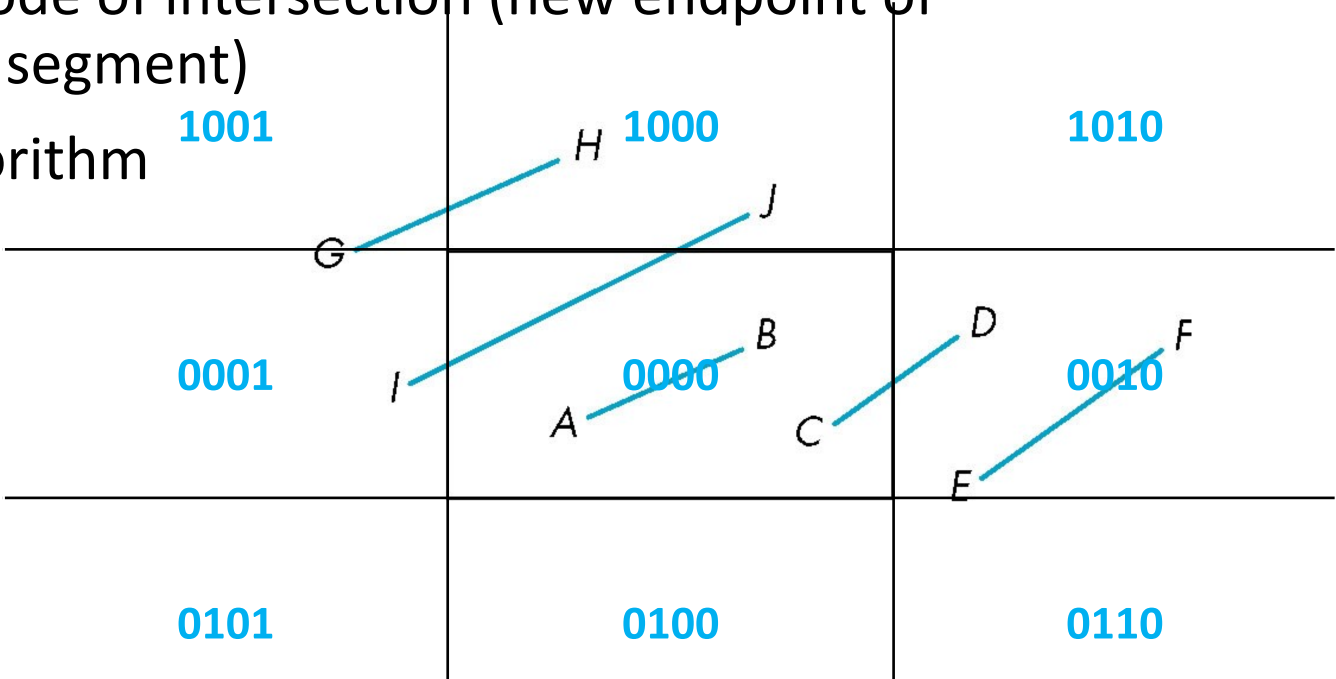
Using Outcodes

- EF: outcode(E) logically ANDed with outcode(F) (bitwise) $\neq 0$
  - Both outcodes have a 1 bit in the same place
  - Line segment is outside of corresponding side of clipping window
  - **reject**



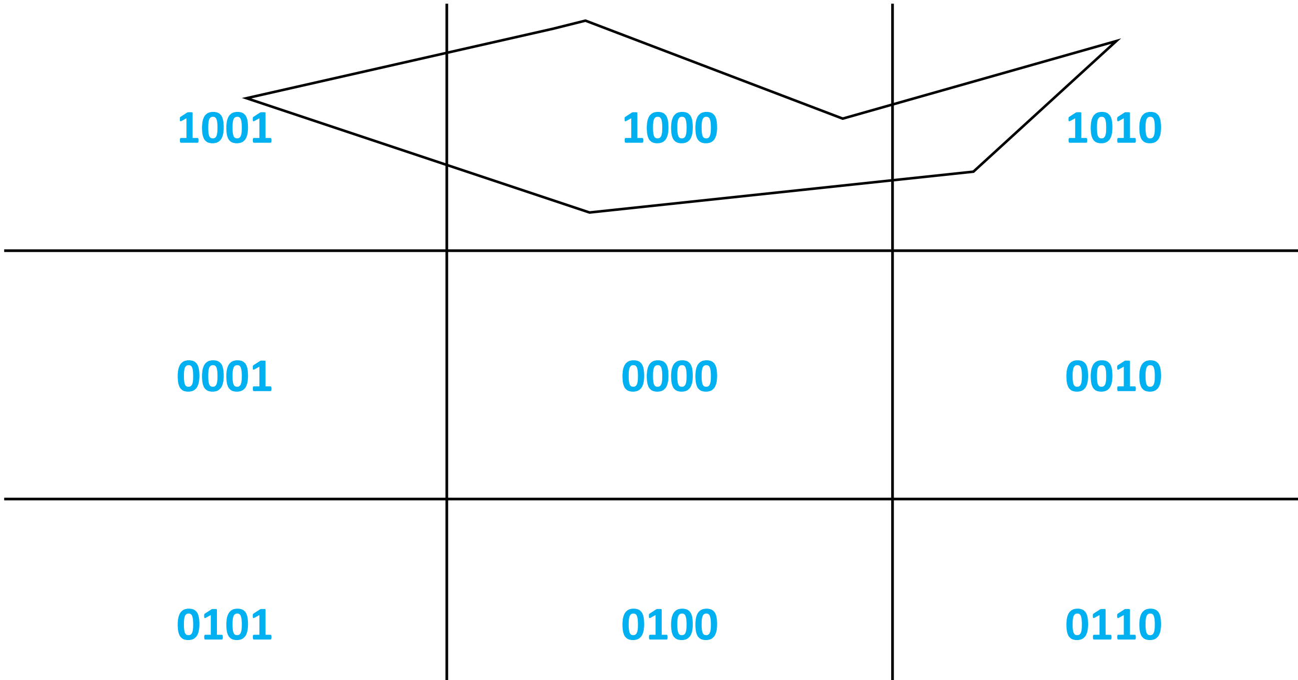
Using Outcodes

- GH and IJ: same outcodes, neither zero but logical AND yields zero
- Shorten line segment by intersecting with one of sides of window
- Compute outcode of intersection (new endpoint of shortened line segment)
- Reexecute algorithm



Using Outcodes

- It works for arbitrary primitives
- And for arbitrary dimensions
- 1001 & 1000 & 1010 & 1010 & 1000 & 1000 = 1000
  Reject!

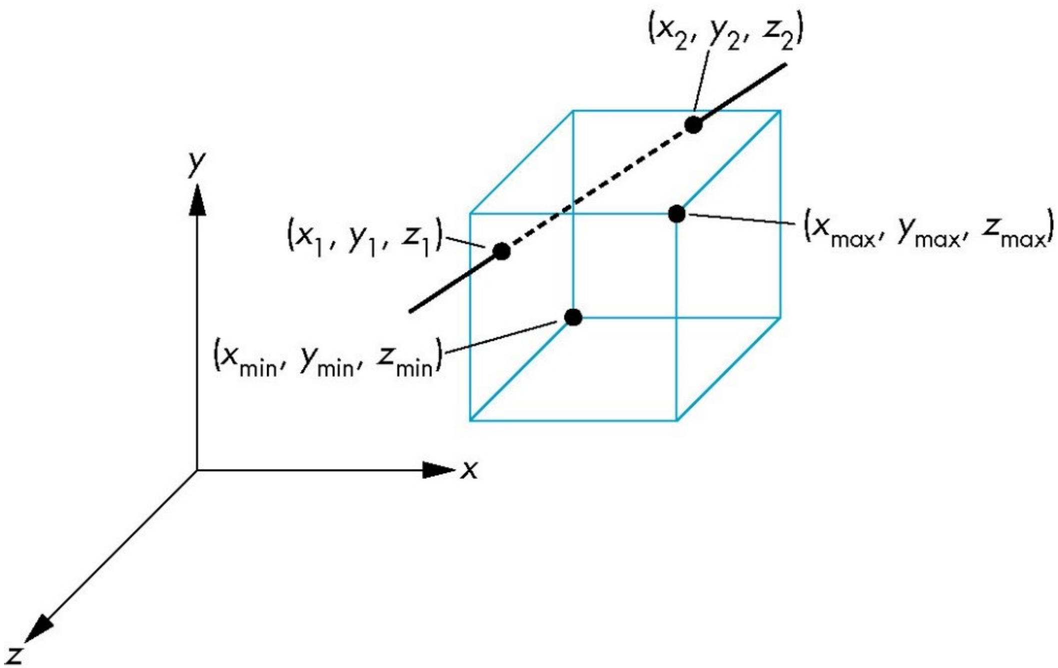


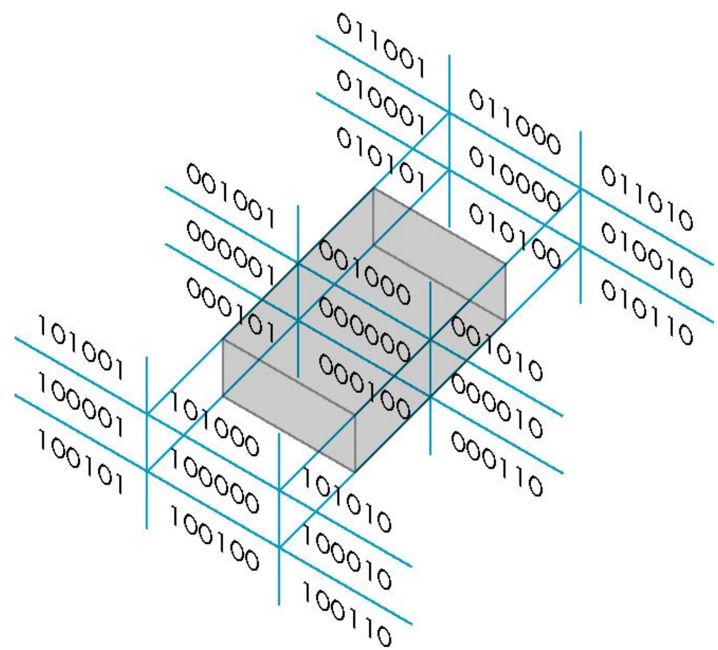
Efficiency

- In many applications, the clipping window is small relative to the size of the entire data base
  - Most line segments are outside one or more side of the window and can be eliminated based on their outcodes
- Inefficiency when code has to be reexecuted for line segments that must be shortened in more than one step

Cohen Sutherland in 3D

- Use 6-bit outcodes
- When needed, clip line segment against planes

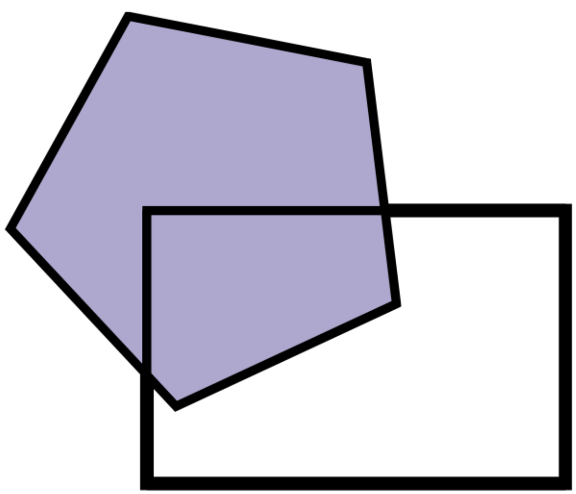


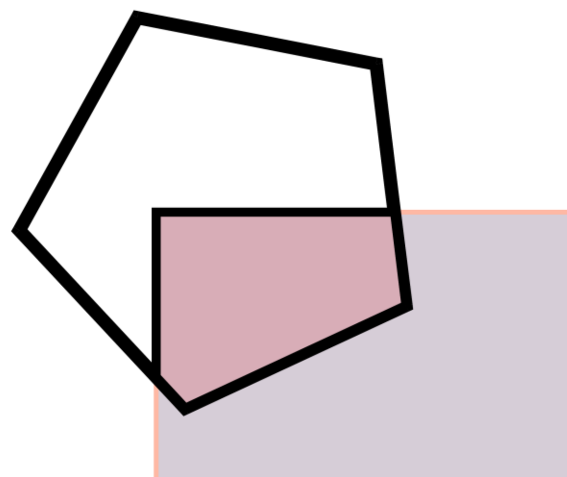


- Questions?

Clipping Polygon

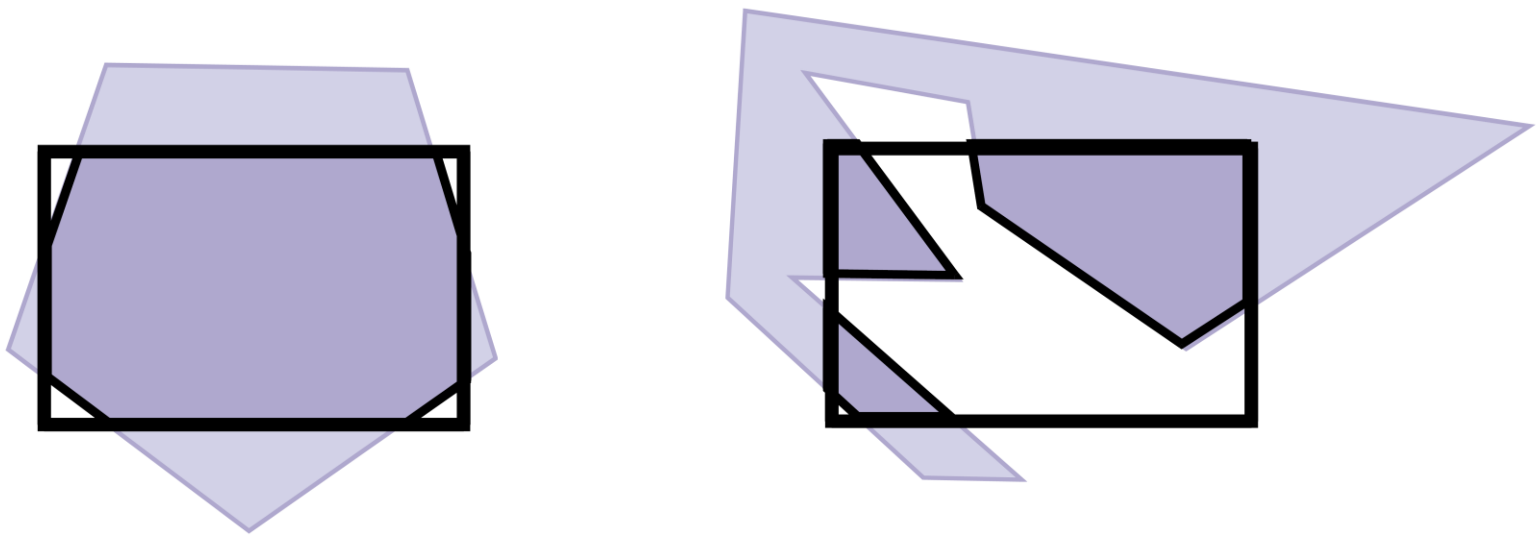
- Clipping polygon is symmetric





Clipping Polygon in 2D

- Clipping polygon is complex



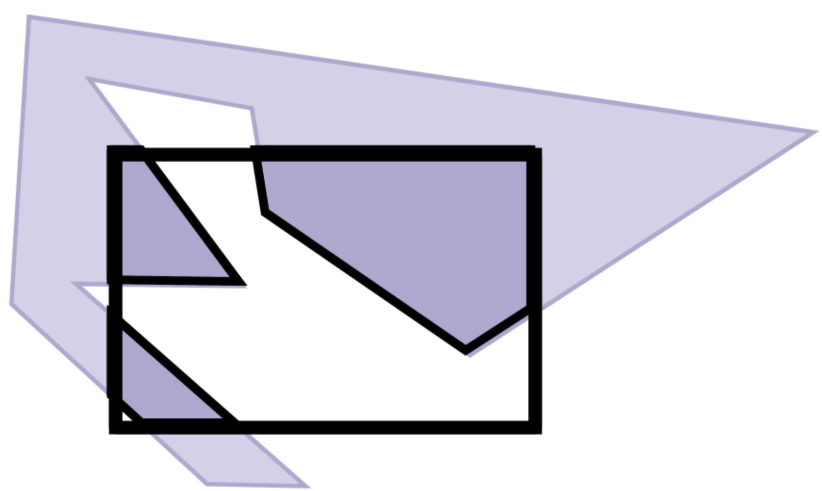
Polygon Clipping

- Not as simple as line segment clipping
  - Clipping a line segment yields at most one line segment
  - Clipping a polygon can yield multiple polygons

- However, clipping a convex polygon can yield at most one other polygon

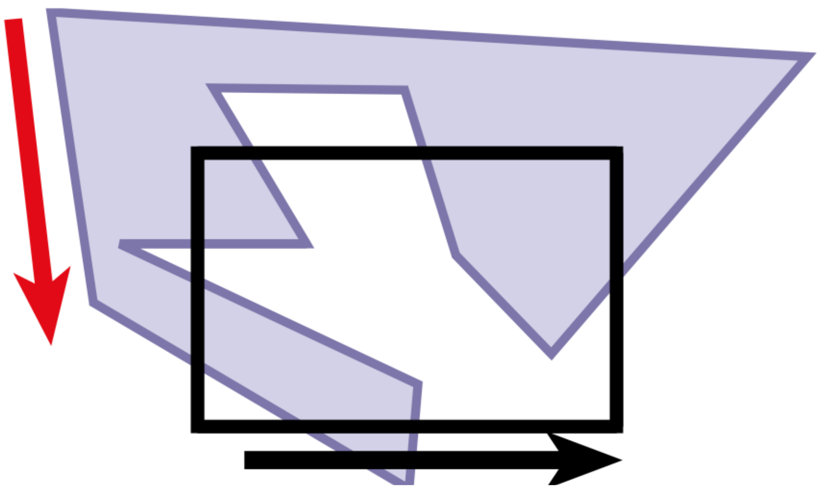
The naive method

- N*M intersections
- Must link all the segments
- Not efficient and even not easy



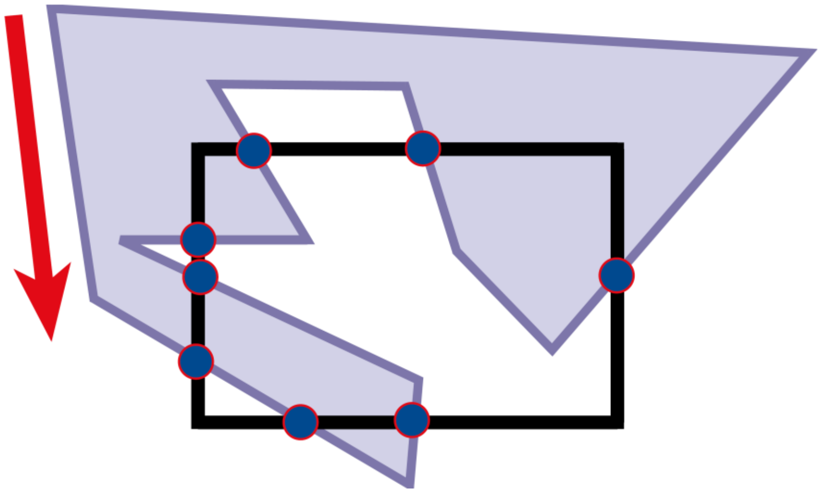
Weiler-Atherton Clipping

- Strategy: “Walk” polygon/window boundary
- Polygons are oriented (CCW)



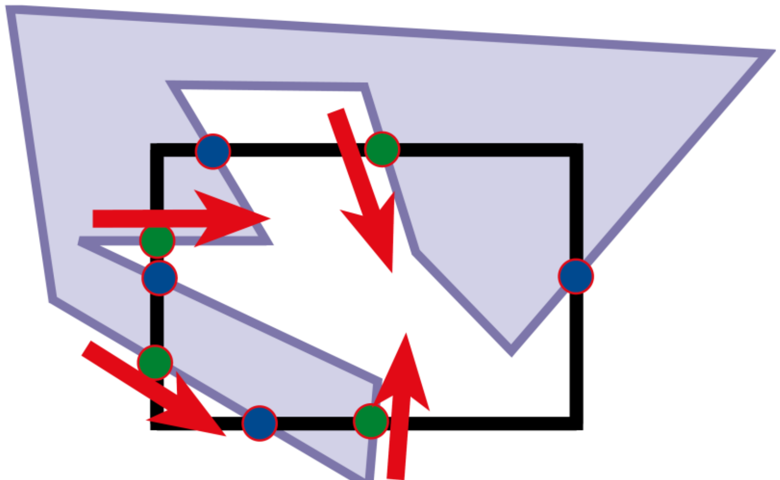
Weiler-Atherton Clipping

- Compute intersection points



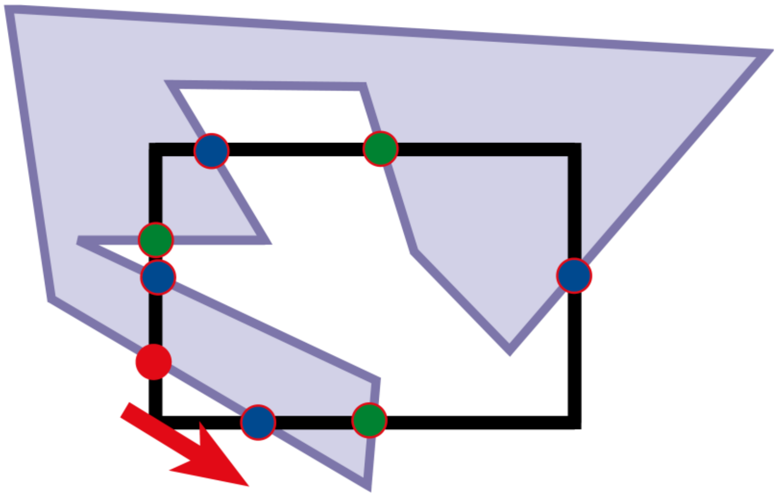
Weiler-Atherton Clipping

- Compute intersection points
- Mark points where polygons enters clipping window (green here)



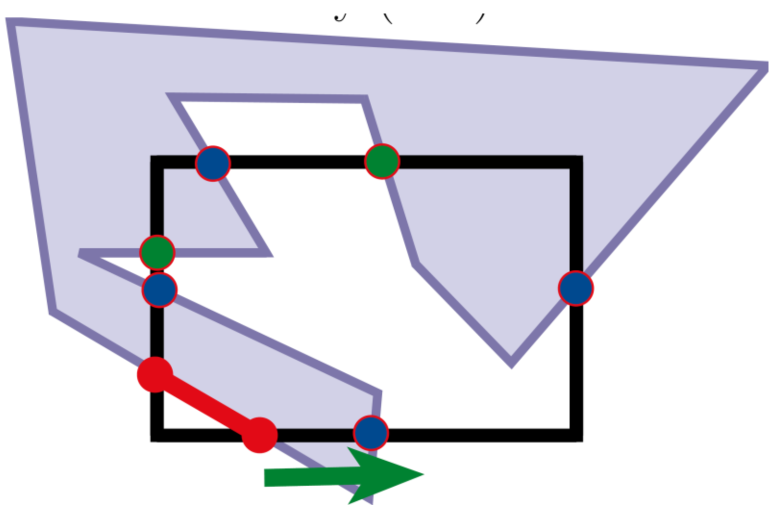
Weiler-Atherton Clipping

- While there is still an unprocessed entering intersection
- Walk" polygon/window boundary



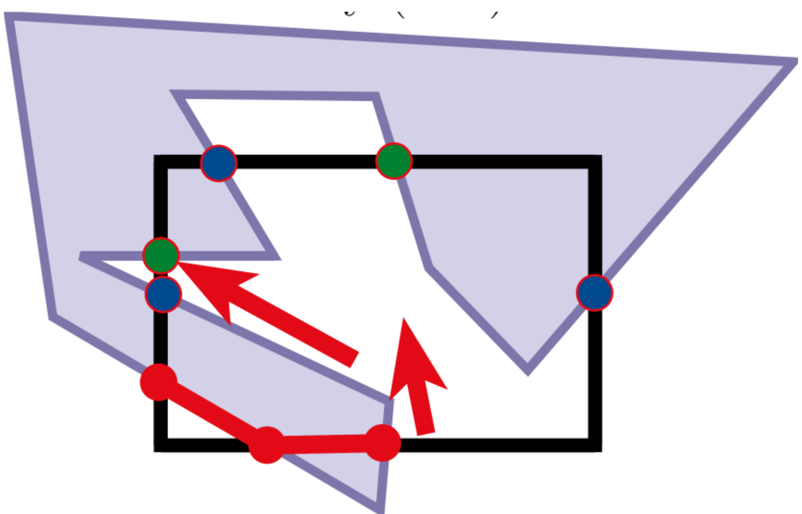
Walking rules

- Out-to-in point:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out point:
  - Record clipped point
  - Follow window boundary (ccw)



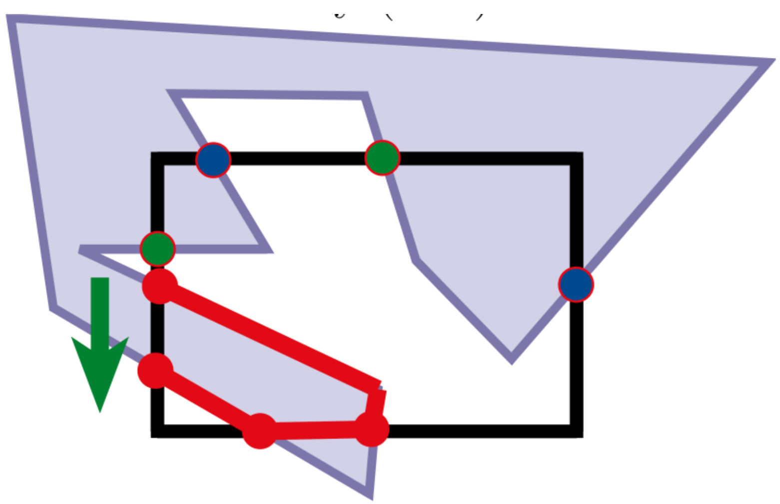
Walking rules

- Out-to-in point:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out point:
  - Record clipped point
  - Follow window boundary (ccw)



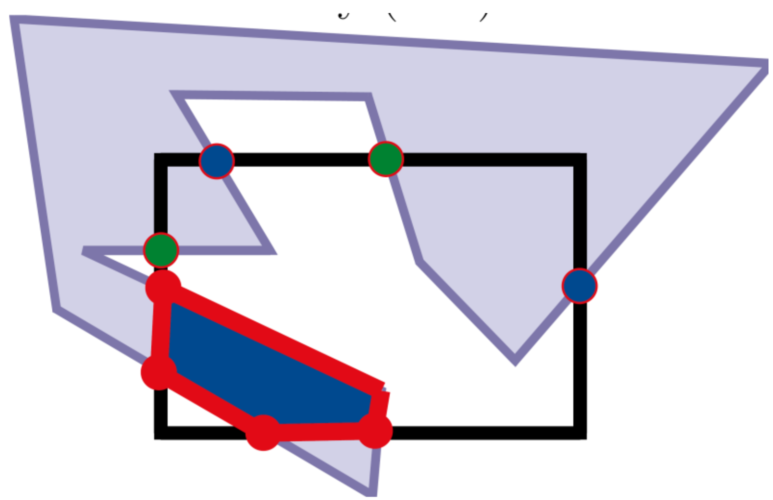
Walking rules

- Out-to-in point:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out point:
  - Record clipped point
  - Follow window boundary (ccw)



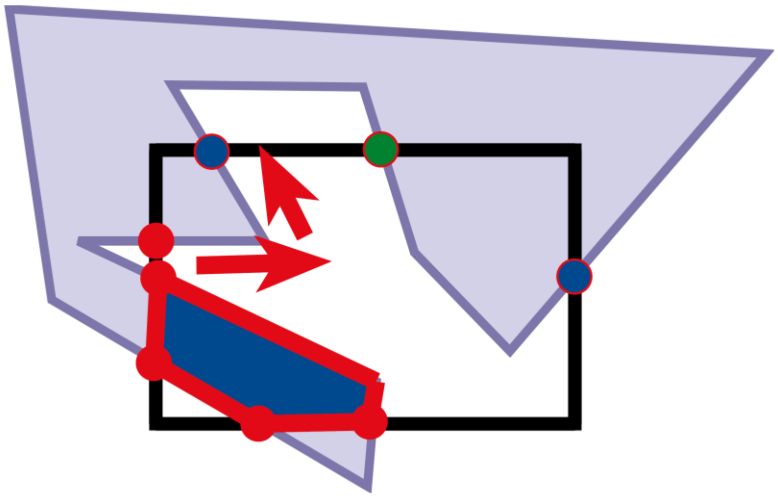
Walking rules

- Out-to-in point:
  - Record clipped point
  - Follow polygon boundary (ccw)
- In-to-out point:
  - Record clipped point
  - Follow window boundary (ccw)



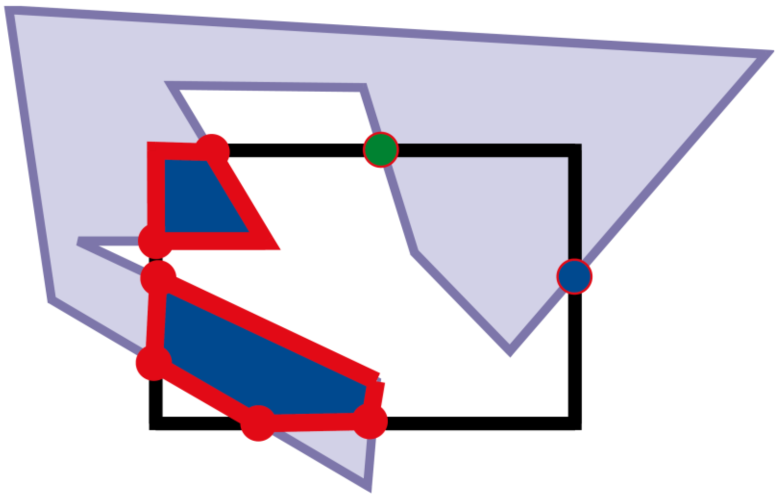
Walking rules

- While there is still an unprocessed entering intersection
- Walk" polygon/window boundary



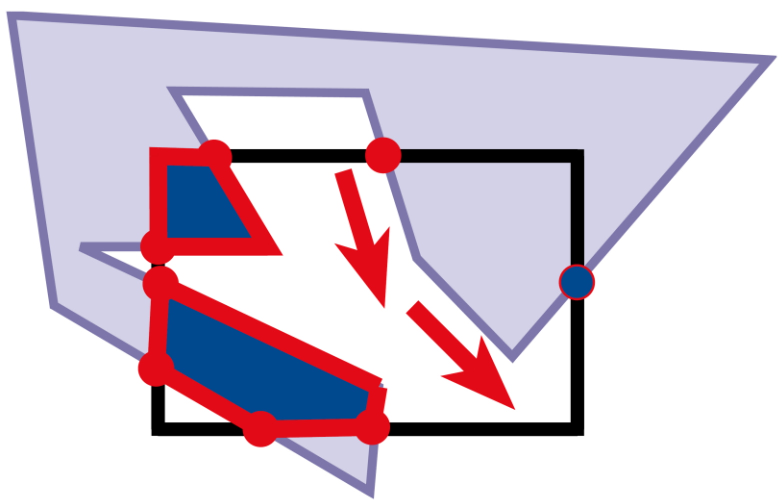
Walking rules

- While there is still an unprocessed entering intersection
- Walk” polygon/window boundary



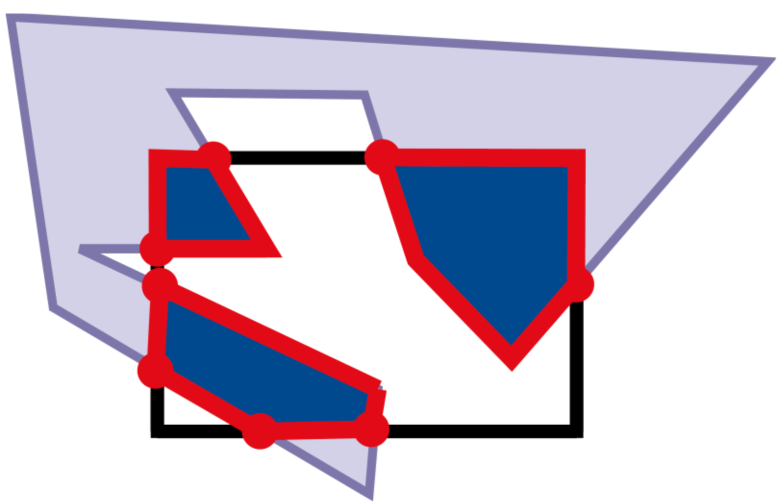
Walking rules

- While there is still an unprocessed entering intersection
- Walk" polygon/window boundary



Walking rules

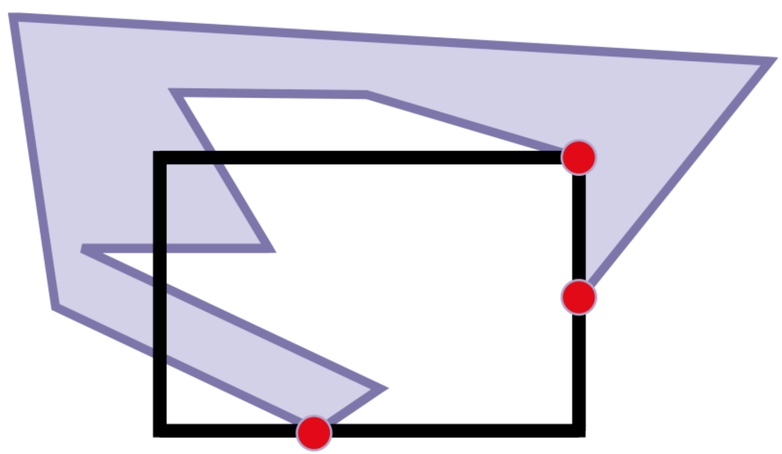
- While there is still an unprocessed entering intersection
- Walk” polygon/window boundary
- Importance of good adjacency data structure (here simply list of oriented edges)



Robustness, precision, degeneracies

- What if a vertex is on the boundary?
- What happens if it is “almost” on the boundary?

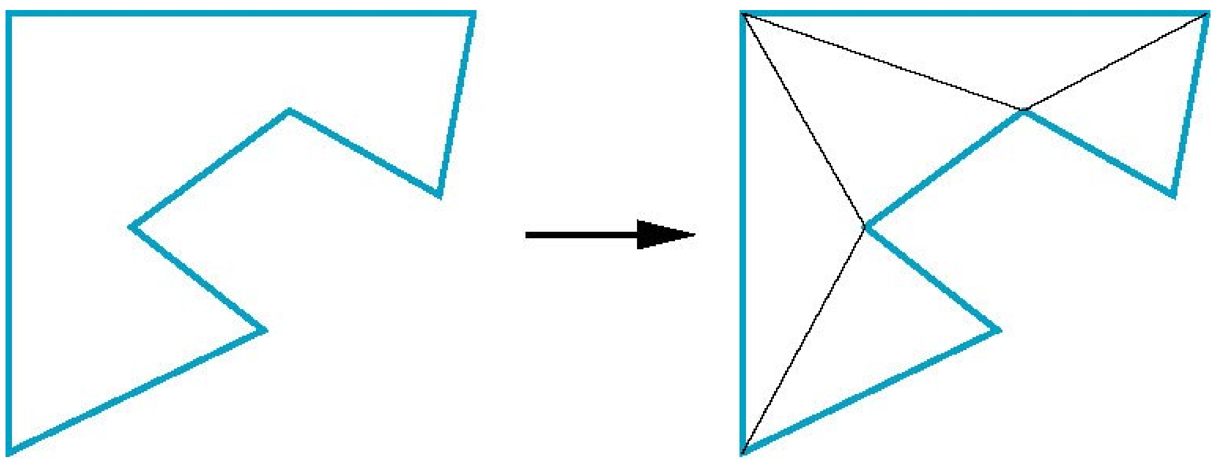
Problem with floating point precision



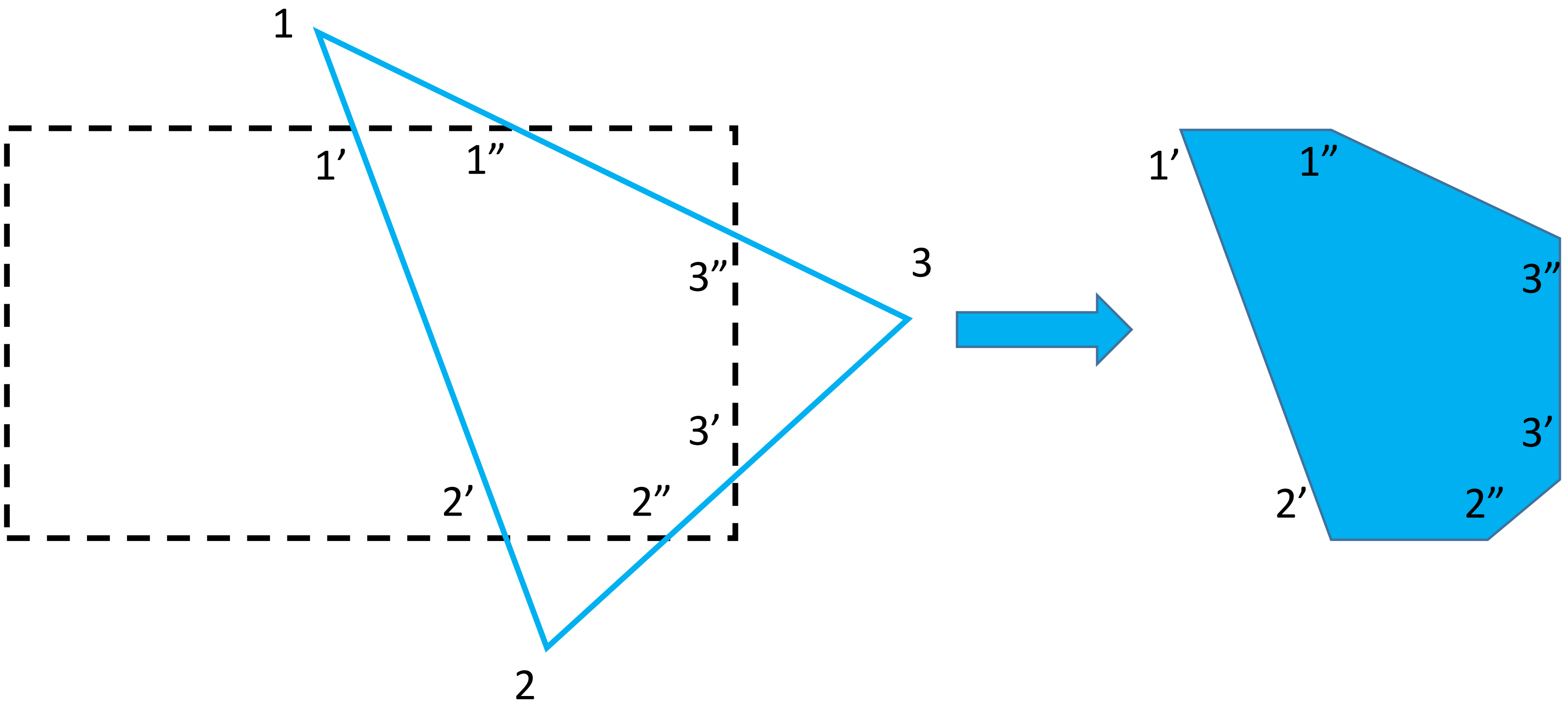
- Other ways?

Tessellation and Convexity

- Another strategy is to replace nonconvex (*concave*) polygons with a set of triangular polygons (a *tessellation*)
- Also makes fill easier

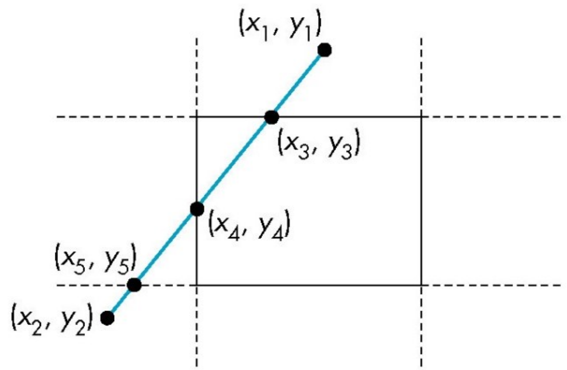


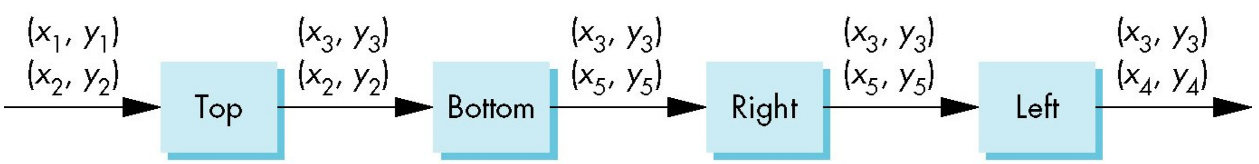
Clipping Convex Polygon



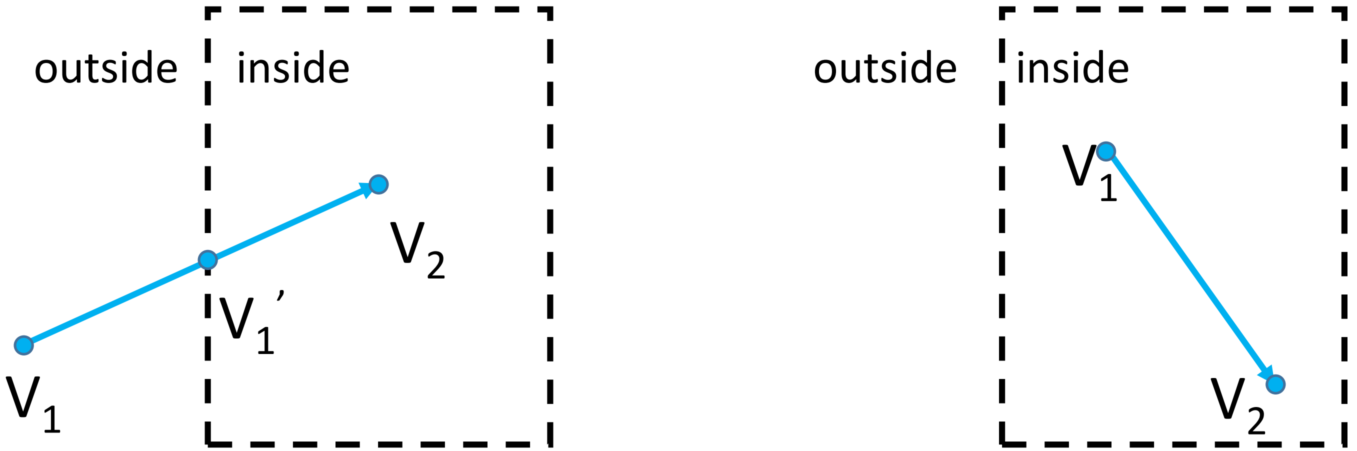
Sutherland-Hodgman Polygon Clipping

- Clipping against each side of window is independent of other sides
  - Can use four independent clippers in a pipeline





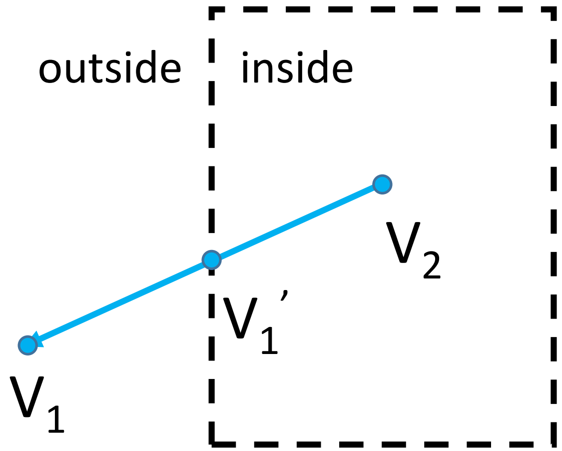
Sutherland-Hodgman Polygon Clipping



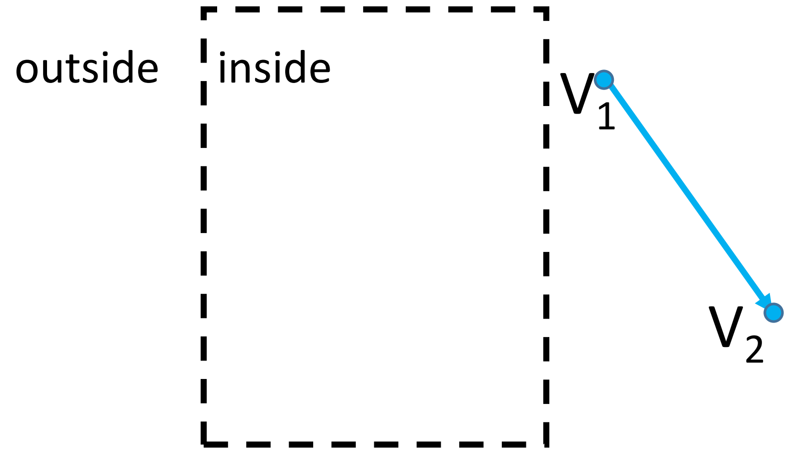
out -> in
Output: V_1', V_2

in -> in
Output: V_2

Sutherland-Hodgman Polygon Clipping

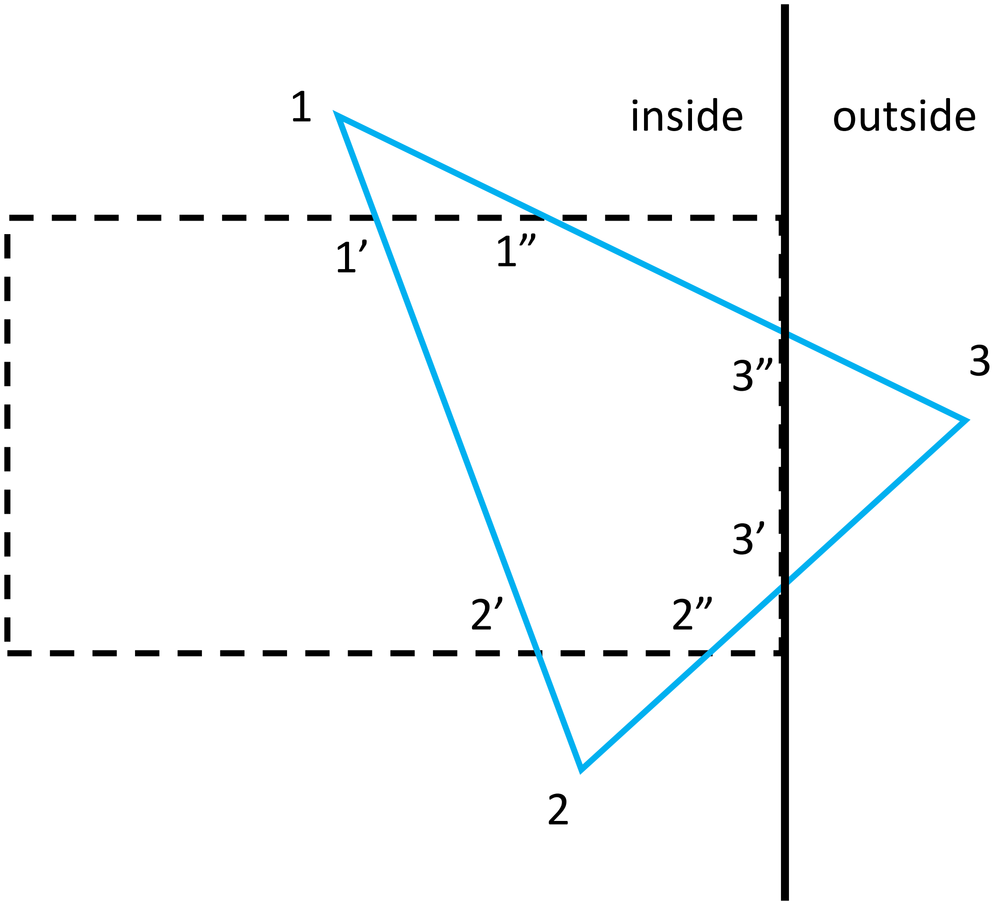


in -> out
Output: V_1'



out-> out
Output: None

Sutherland-Hodgman Polygon Clipping



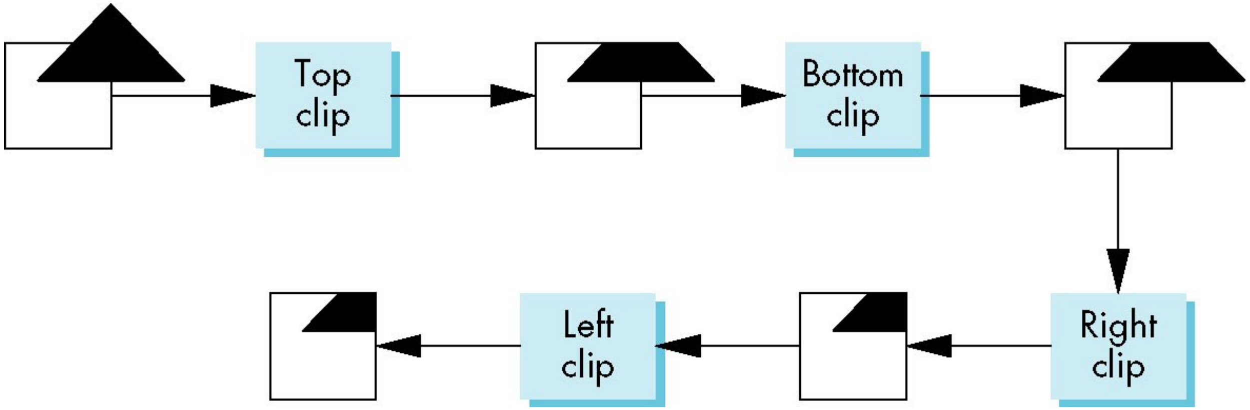
Right clipper:

(1, 2): (in - in) -> {2}

(2, 3): (in - out) -> {3’}

(3, 1): (out - in) -> {3”,1}

Pipeline Clipping of Polygons



- Three dimensions: add front and back clippers
- Strategy used in SGI Geometry Engine
- Small increase in latency

- Questions?

References

- Ed Angel, CS/EECE 433 Computer Graphics, University of New Mexico
- Steve Marschner, CS4620/5620 Computer Graphics, Cornell
- Tom Thorne, COMPUTER GRAPHICS, The University of Edinburgh
- Elif Tosun, Computer Graphics, The University of New York
- http://www.songho.ca/opengl/gl_projectionmatrix.html