2D Clipping Algorithm

Outline

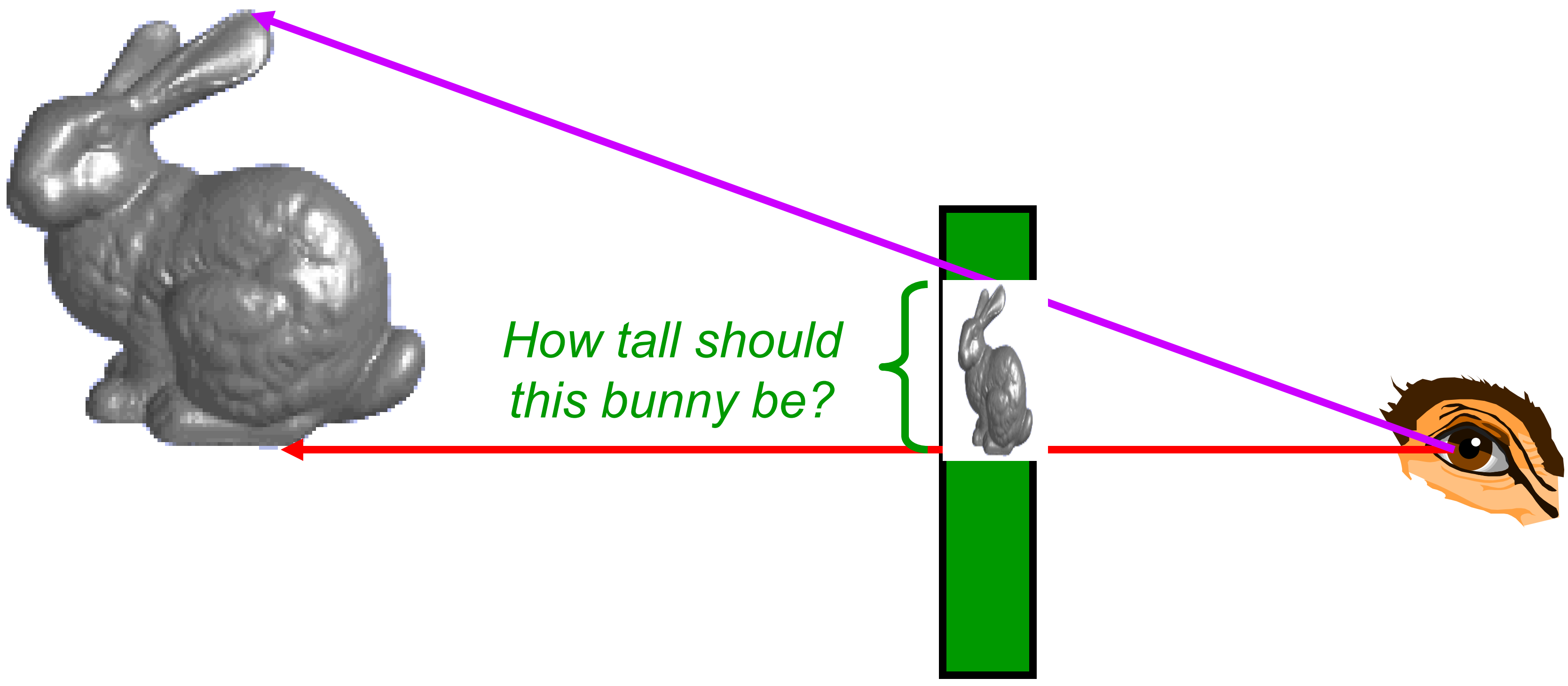
- Review
- Clipping Basics
- Cohen-Sutherland Line Clipping
- Clipping Polygons
- Sutherland-Hodgman Clipping
- Perspective Clipping

Recap: Homogeneous Coords

- Intuitively:
  - The *w* coordinate of a homogeneous point is typically 1
  - Decreasing *w* makes the point “bigger”, meaning further from the origin
  - Homogeneous points with *w* = 0 are thus “points at infinity”, meaning infinitely far away in some direction. *(What direction?)*
  - To help illustrate this, imagine subtracting two homogeneous points: the result is (as expected) a vector

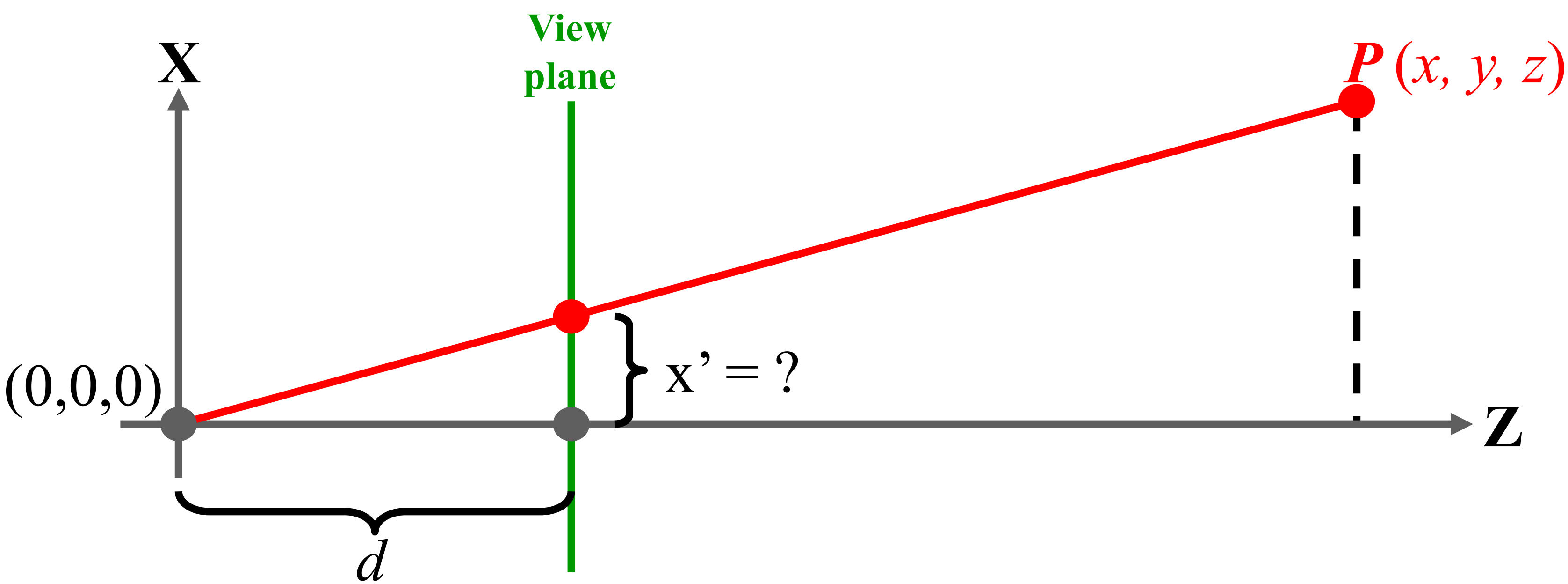
Recap: Perspective Projection

- When we do 3-D graphics, we think of the screen as a 2-D window onto the 3-D world:



Recap: Perspective Projection

- The geometry of the situation:



- Desired result:

$$x' = \frac{d \cdot x}{z} = \frac{x}{z/d}, \quad y' = \frac{d \cdot y}{z} = \frac{y}{z/d}, \quad z = d$$

Recap: Perspective Projection Matrix

- Example:

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

- Or, in 3-D coordinates: $\left(\frac{x}{z/d}, \quad \frac{y}{z/d}, \quad d \right)$

Recap: OpenGL's Persp. Proj. Matrix

- OpenGL's **`gluPerspective()`** command generates a slightly more complicated matrix:

$$\begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \left(\frac{Z_{far}+Z_{near}}{Z_{near}-Z_{far}}\right) & \left(\frac{2\times Z_{far}\times Z_{near}}{Z_{near}-Z_{far}}\right) \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\text{where} \quad f = \cot\left(\frac{fov_y}{2}\right)$$

- *Can you figure out what this matrix does?*

Projection Matrices

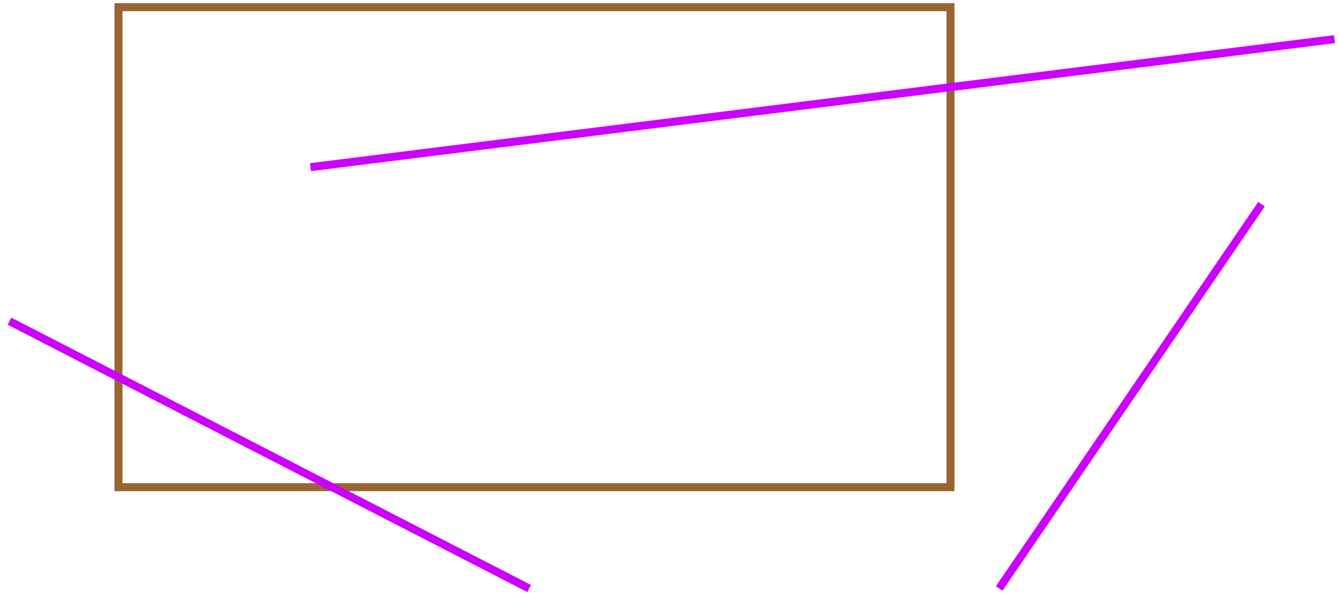
- Now that we can express perspective foreshortening as a matrix, we can composite it onto our other matrices with the usual matrix multiplication
- End result: can create a single matrix encapsulating modeling, viewing, and projection transforms
  - Though you will recall that in practice OpenGL separates the **`modelview`** from **`projection`** matrix *(why?)*

Outline

- Review
- Clipping Basics
- Cohen-Sutherland Line Clipping
- Clipping Polygons
- Sutherland-Hodgman Clipping
- Perspective Clipping

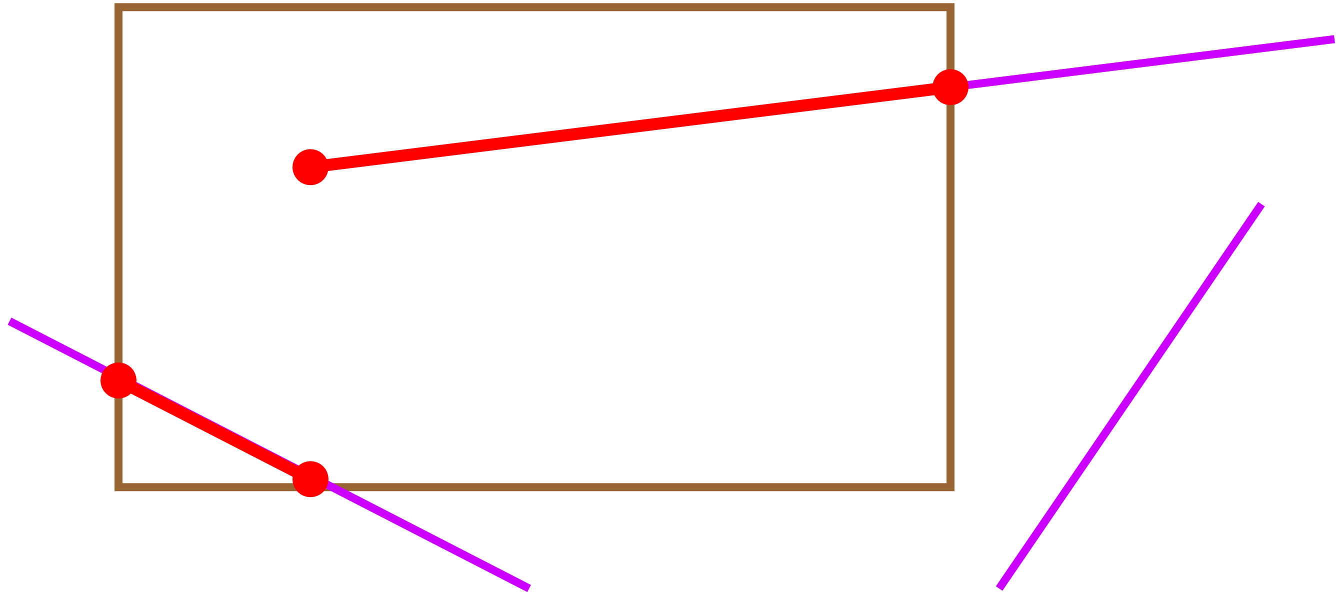
Next Topic: Clipping

- We've been assuming that all primitives (lines, triangles, polygons) lie entirely within the *viewport*
- In general, this assumption will not hold



Clipping

- Analytically calculating the portions of primitives within the viewport



Why Clip?

- Bad idea to rasterize outside of framebuffer bounds
- Also, don’t waste time scan converting pixels outside window

Clipping

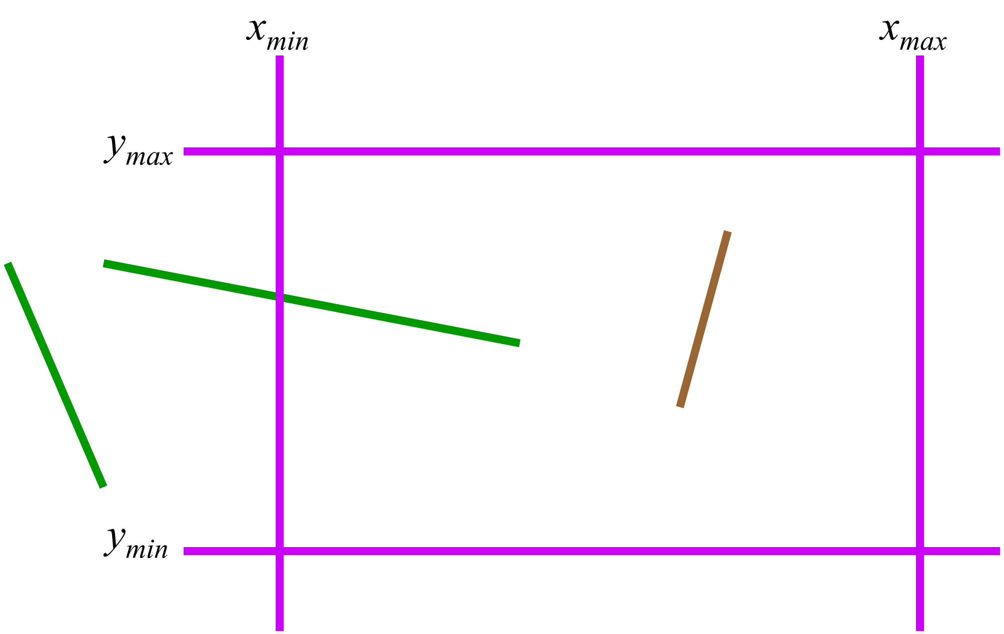
- The naïve approach to clipping lines:

```
for each line segment
    for each edge of viewport
        find intersection points
        pick "nearest" point
    if anything is left, draw it
```

- *What do we mean by "nearest"?*
- *How can we optimize this?*

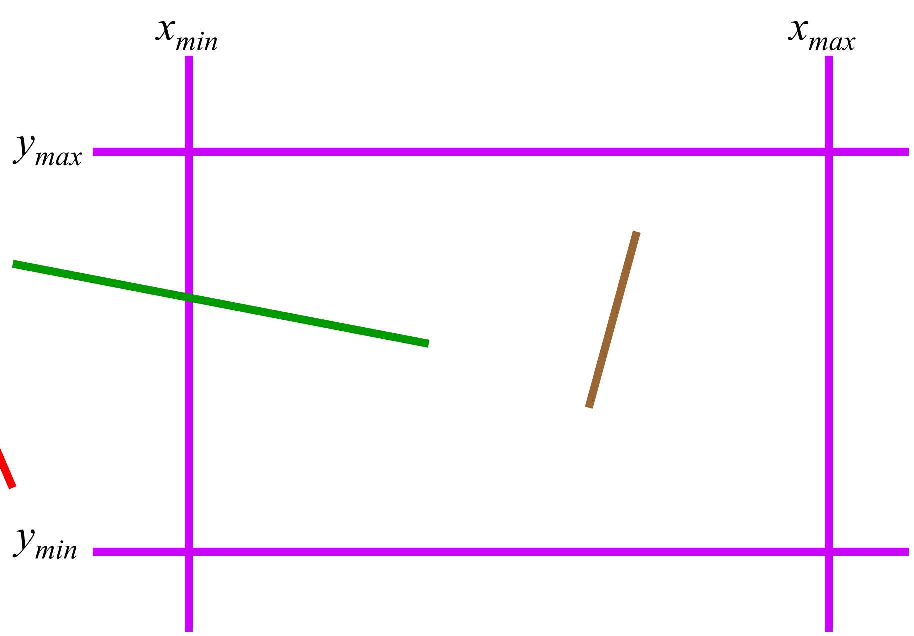
Trivial Accepts

- Big optimization: trivial accept/rejects
- *How can we quickly determine whether a line segment is entirely inside the viewport?*
- A: test both endpoints.



Trivial Rejects

- *How can we know a line is outside viewport?*
- A: if both endpoints on wrong side of same edge, can trivially reject line

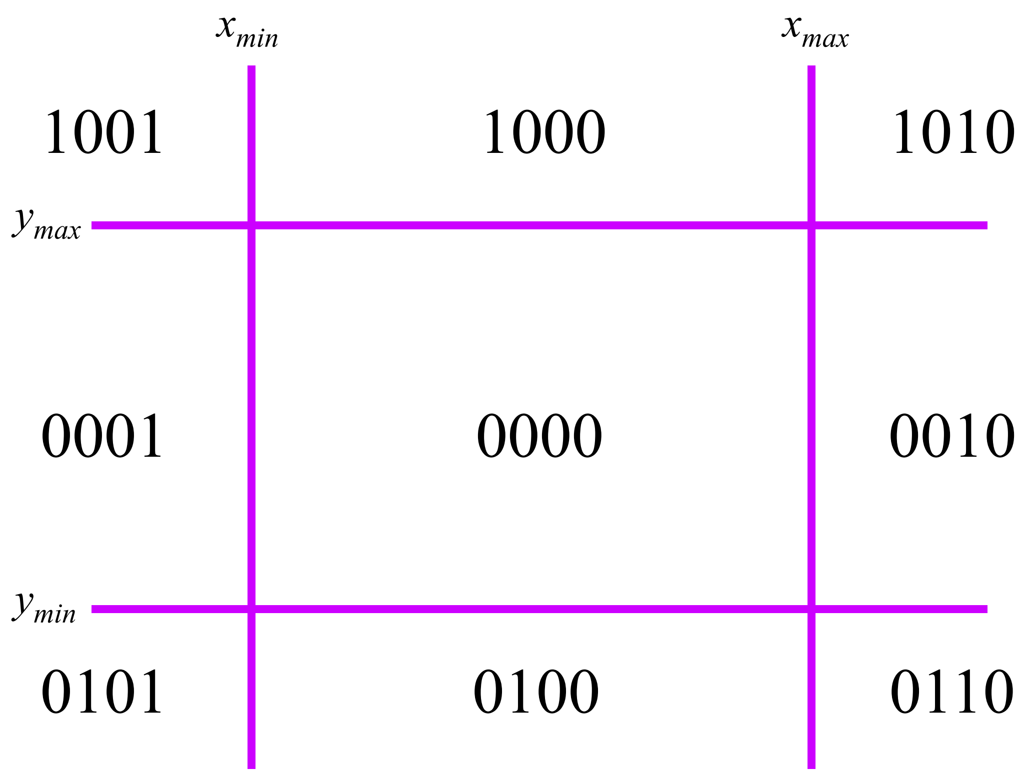


Outline

- Review
- Clipping Basics
- Cohen-Sutherland Line Clipping
- Clipping Polygons
- Sutherland-Hodgman Clipping
- Perspective Clipping

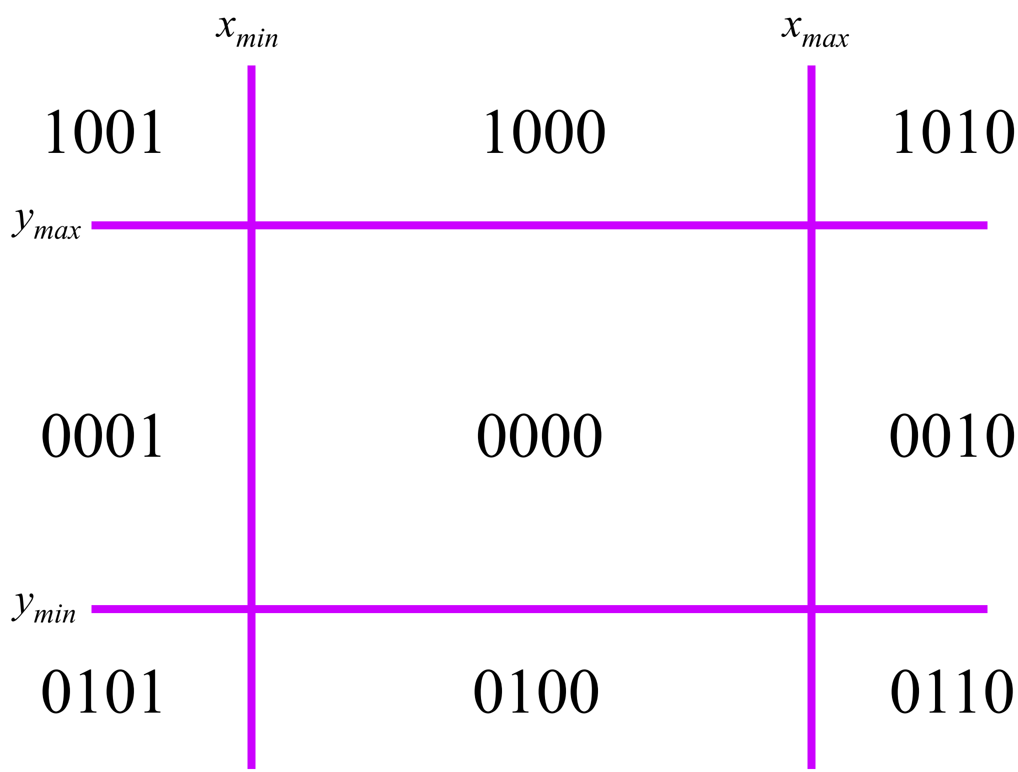
Cohen-Sutherland Line Clipping

- Divide viewplane into regions defined by viewport edges
- Assign each region a 4-bit *outcode*:



Cohen-Sutherland Line Clipping

- *To what do we assign outcodes?*
- *How do we set the bits in the outcode?*
- *How do you suppose we use them?*

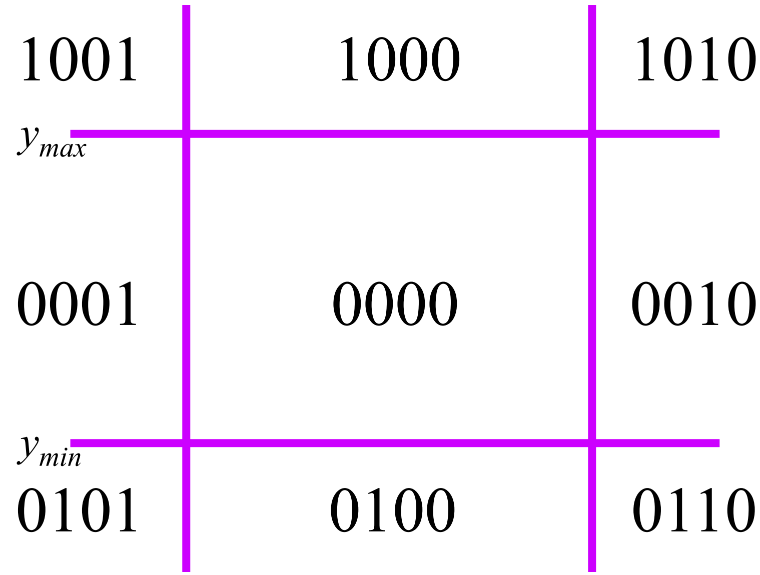


Cohen-Sutherland Line Clipping

- Set bits with simple tests

  $x > x_{max}$ $y < y_{min}$ etc.

- Assign an outcode to each vertex of line
  - If both outcodes = 0, trivial accept
  - bitwise AND vertex outcodes together
  - If result $\neq$ 0, trivial reject
    - As those lines lie on one side of the boundary lines

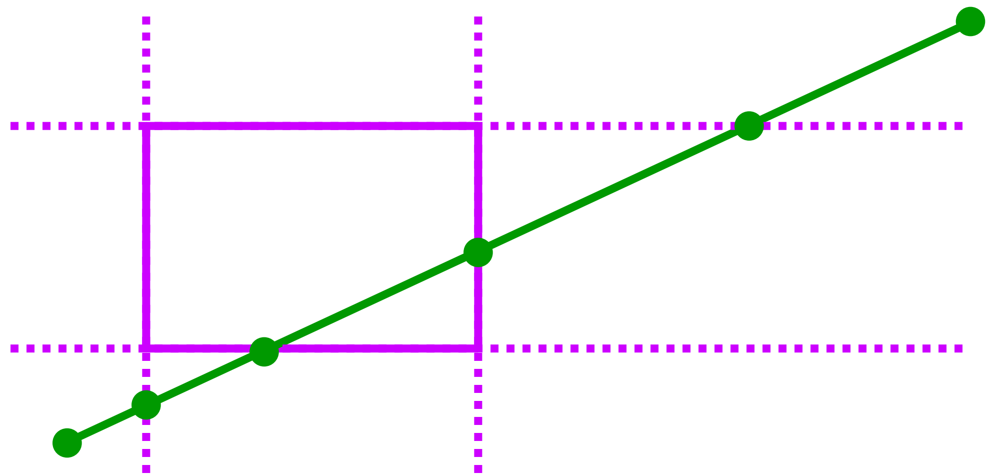


Cohen-Sutherland Line Clipping

- If line cannot be trivially accepted or rejected, subdivide so that one or both segments can be discarded
- Pick an edge that the line crosses (*how?*)
- Intersect line with edge (*how?*)
- Discard portion on wrong side of edge and assign outcode to new vertex
- Apply trivial accept/reject tests; repeat if necessary

Cohen-Sutherland Line Clipping

- Outcode tests and line-edge intersects are quite fast (*how fast?*)
- But some lines require multiple iterations:
  - Clip top
  - Clip left
  - Clip bottom
  - Clip right



- Fundamentally more efficient algorithms:
  - Cyrus-Beck uses parametric lines
  - Liang-Barsky optimizes this for upright volumes

Outline

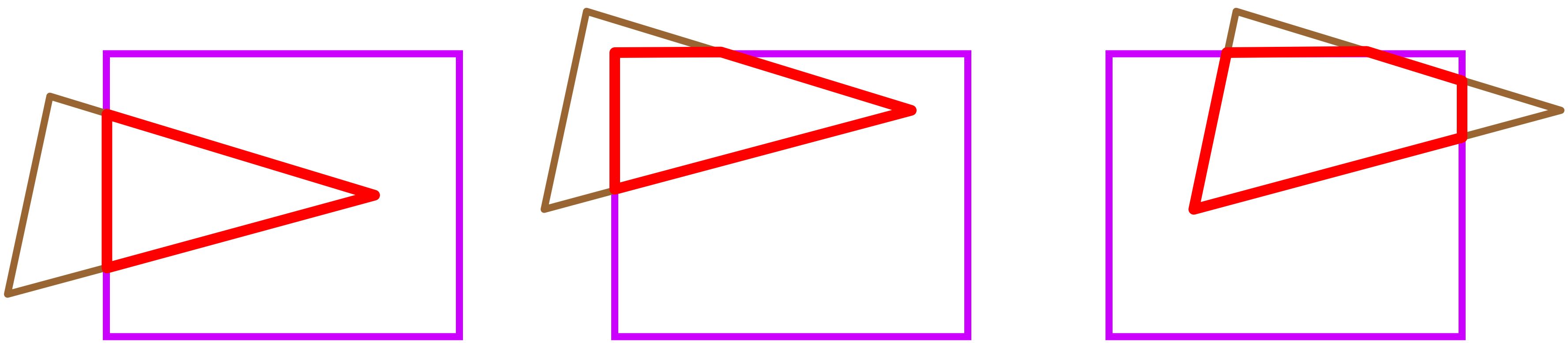
- Review
- Clipping Basics
- Cohen-Sutherland Line Clipping
- Clipping Polygons
- Sutherland-Hodgman Clipping
- Perspective Clipping

Clipping Polygons

- We know how to clip a single line segment
  - How about a polygon in 2D?
  - How about in 3D?
- Clipping polygons is more complex than clipping the individual lines
  - Input: polygon
  - Output: polygon, or nothing
- *When can we trivially accept/reject a polygon as opposed to the line segments that make up the polygon?*

Why Is Clipping Hard?

- *What happens to a triangle during clipping?*
- Possible outcomes:

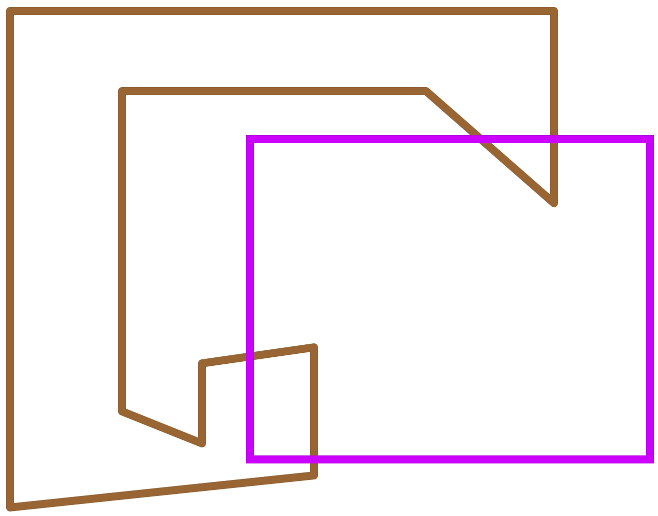


Triangle→triangle Triangle→quad Triangle→5-gon

- *How many sides can a clipped triangle have?*

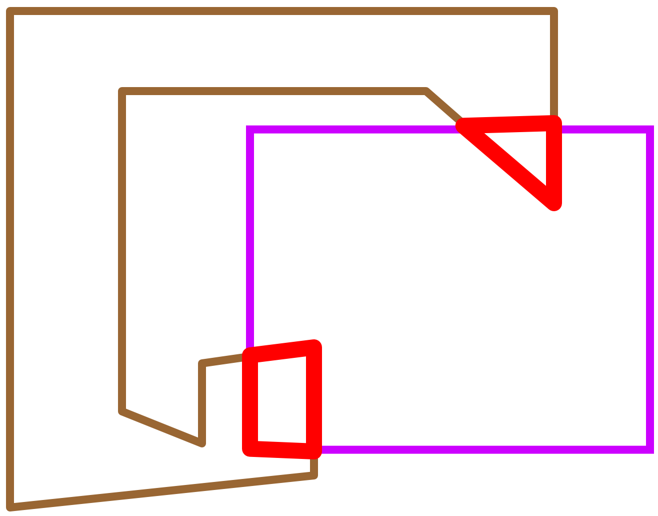
Why Is Clipping Hard?

- A really tough case:



Why Is Clipping Hard?

- A really tough case:



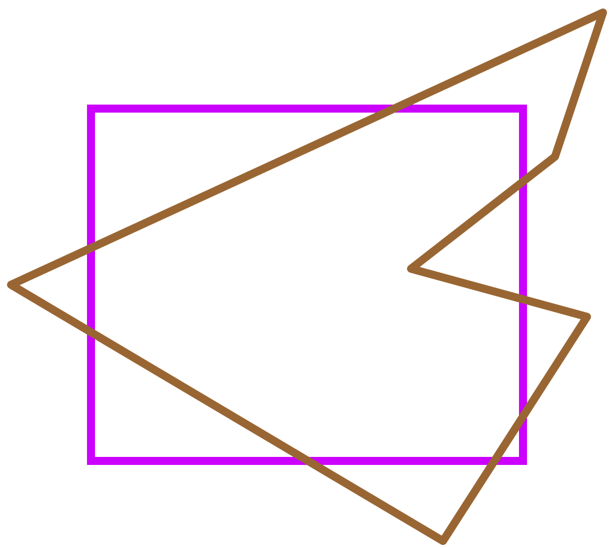
concave polygon→multiple polygons

Outline

- Review
- Clipping Basics
- Cohen-Sutherland Line Clipping
- Clipping Polygons
- Sutherland-Hodgman Clipping
- Perspective Clipping

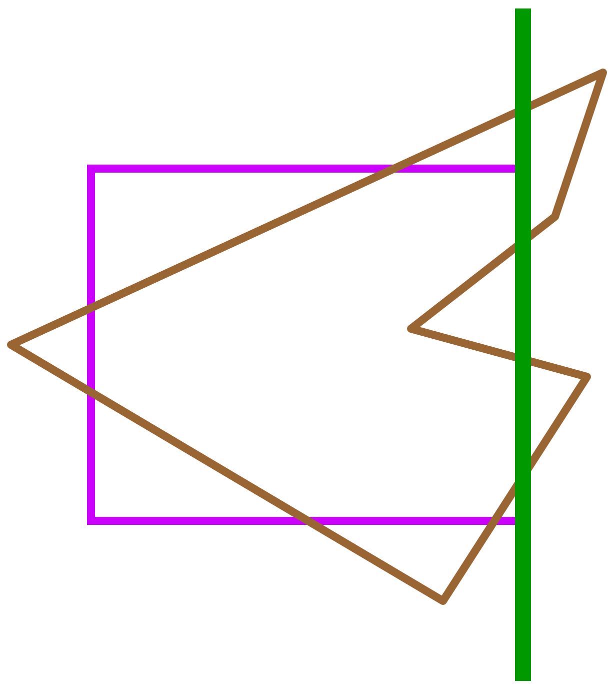
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped



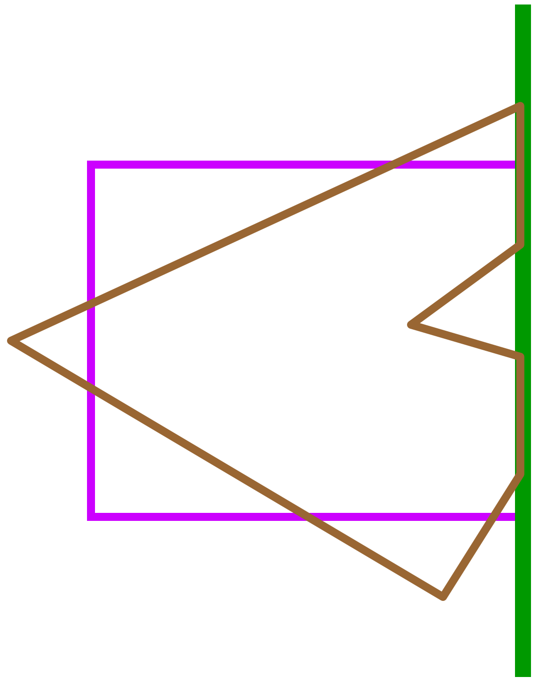
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped



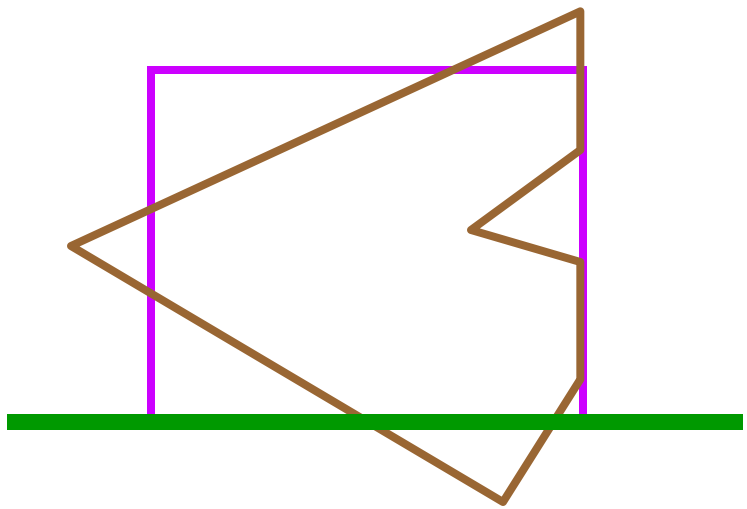
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped



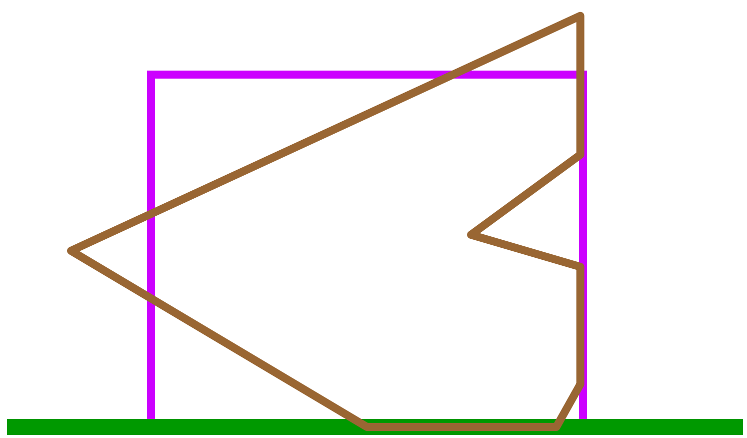
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped



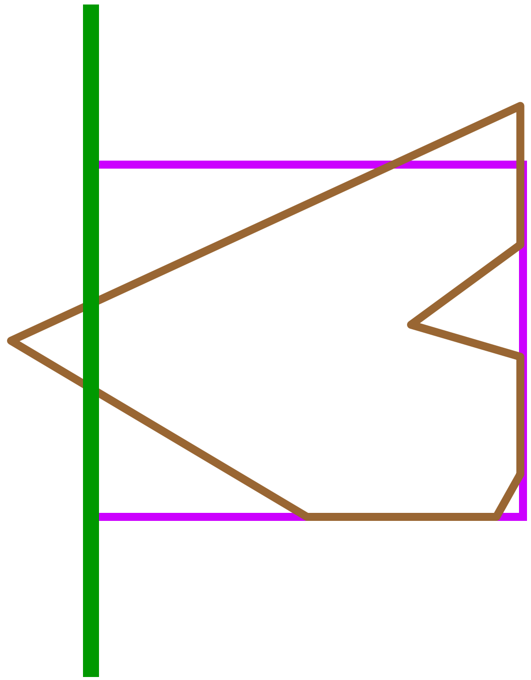
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped



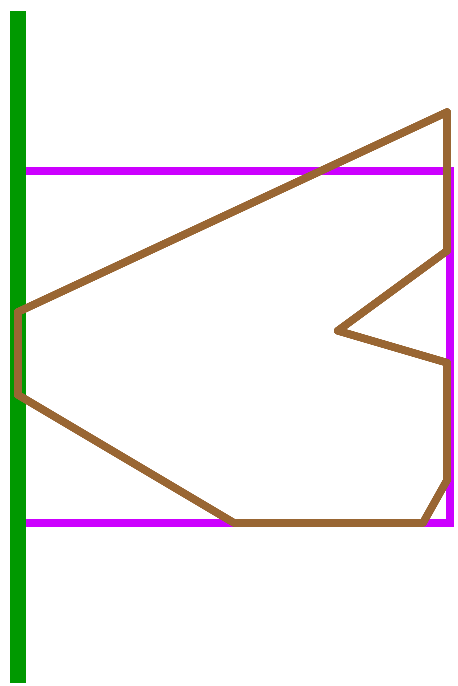
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped



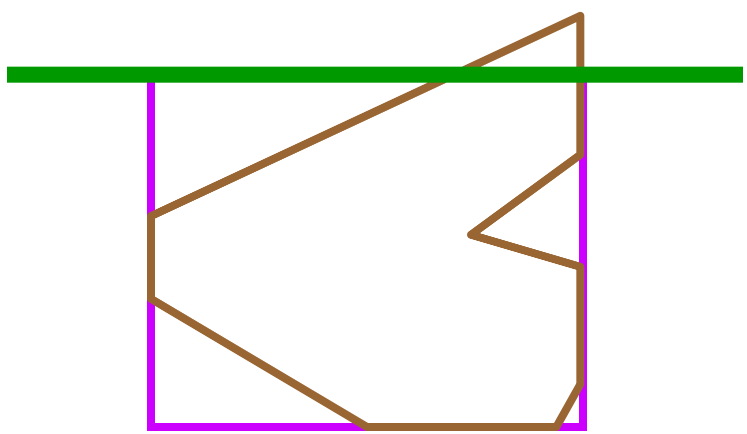
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped



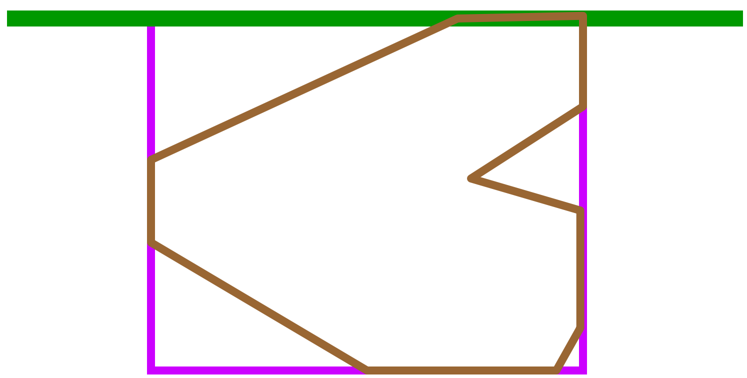
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped



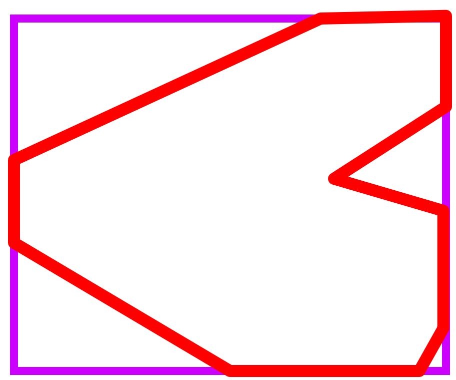
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped



Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation
  - After doing all planes, the polygon is fully clipped
- *Will this work for non-rectangular clip regions?*
- *What would 3-D clipping involve?*



Sutherland-Hodgman Clipping

- Input/output for algorithm:
  - Input: list of polygon vertices in order
  - Output: list of clipped polygon vertices consisting of old vertices (maybe) and new vertices (maybe)
- Note: this is exactly what we expect from the clipping operation against each edge

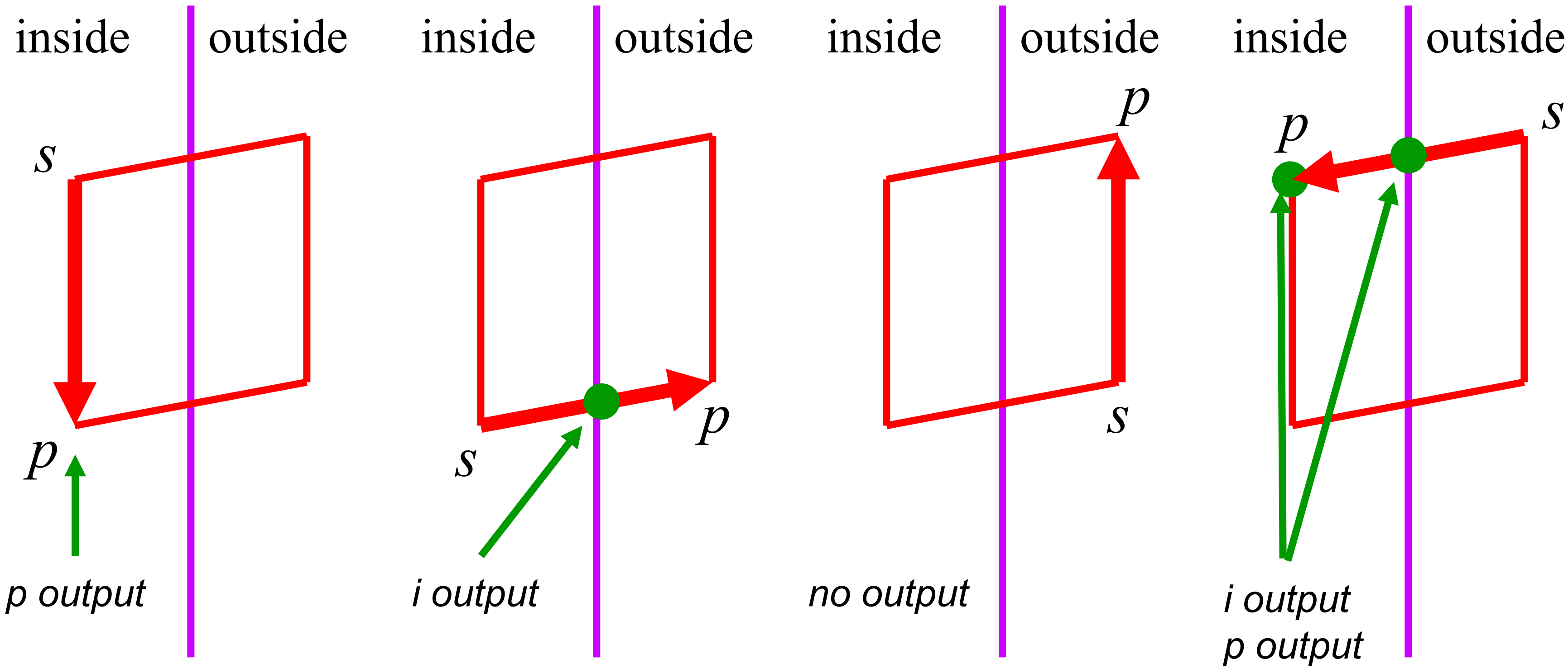
- This algorithm generalizes to 3-D
  - Show movie…

Sutherland-Hodgman Clipping

- We need to be able to create clipped polygons from the original polygons
- Sutherland-Hodgman basic routine:
  - Go around polygon one vertex at a time
  - Current vertex has position *p*
  - Previous vertex had position *s*, and it has been added to the output if appropriate

Sutherland-Hodgman Clipping

- Edge from *s* to *p* takes one of four cases:
  (Purple line can be a line or a plane)



Sutherland-Hodgman Clipping

- 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*

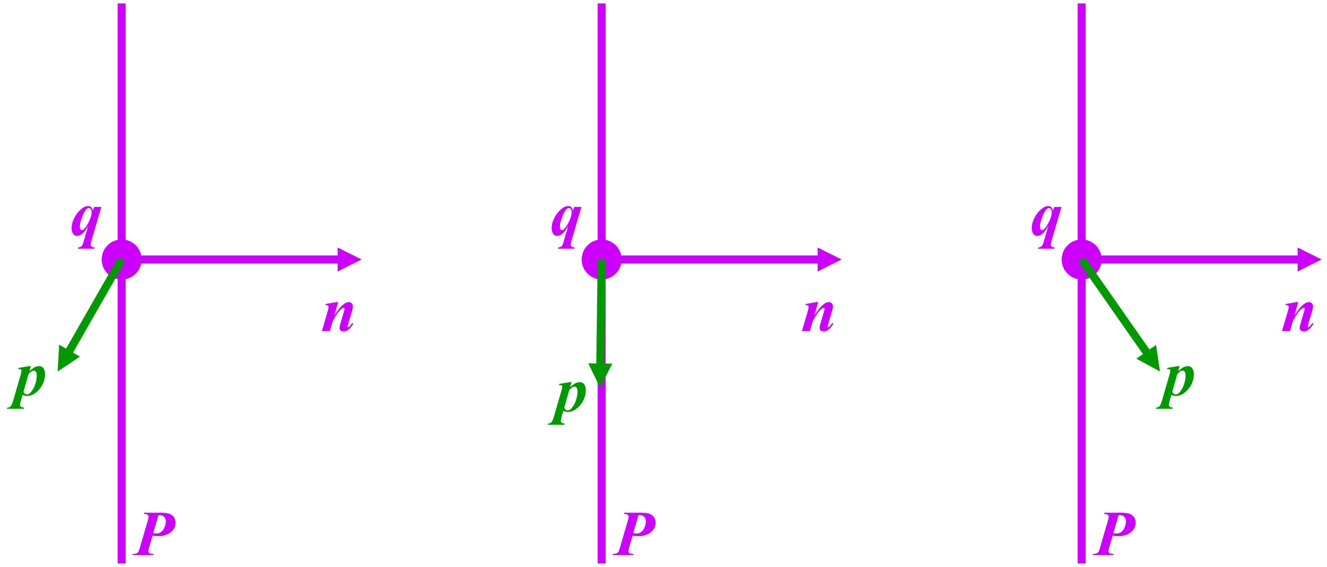
Point-to-Plane test

- A very general test to determine if a point p is "inside" a plane $\boldsymbol{P}$, defined by q and n:

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

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

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



Point-to-Plane Test

- Dot product is relatively expensive
  - 3 multiplies
  - 5 additions
  - 1 comparison (to 0, in this case)
- Think about how you might optimize or special-case this

Finding Line-Plane Intersections

- Use parametric definition of edge:

  $\boldsymbol{E}(t) = \boldsymbol{s} + t(\boldsymbol{p} - \boldsymbol{s})$

  - If t = 0 then $\boldsymbol{E}(t) = \boldsymbol{s}$
  - If t = 1 then $\boldsymbol{E}(t) = \boldsymbol{p}$
  - Otherwise, $\boldsymbol{E}(t)$ is part way from $\boldsymbol{s}$ to $\boldsymbol{p}$

Finding Line-Plane Intersections

- Edge intersects plane ***P*** where ***E****(t)* is on ***P***
  - ***q*** is a point on ***P***
  - ***n*** is normal to ***P***

$$(\boldsymbol{E}(t) - \boldsymbol{q}) \cdot \boldsymbol{n} = 0$$

$$(\boldsymbol{s} + t(\boldsymbol{p} - \boldsymbol{s}) - \boldsymbol{q}) \cdot \boldsymbol{n} = 0$$

$$t = [(\boldsymbol{q} - \boldsymbol{s}) \cdot \boldsymbol{n}] / [(\boldsymbol{p} - \boldsymbol{s}) \cdot \boldsymbol{n}]$$

  - The intersection point ***i* = *E****(t)* for this value of *t*

Line-Plane Intersections

- Note that the length of ***n*** doesn't affect result:

$$t = [(\boldsymbol{q} - \boldsymbol{s}) \cdot \boldsymbol{n}] / [(\boldsymbol{p} - \boldsymbol{s}) \cdot \boldsymbol{n}]$$

- Again, lots of opportunity for optimization

Outline

- Review
- Clipping Basics
- Cohen-Sutherland Line Clipping
- Clipping Polygons
- Sutherland-Hodgman Clipping
- Perspective Clipping