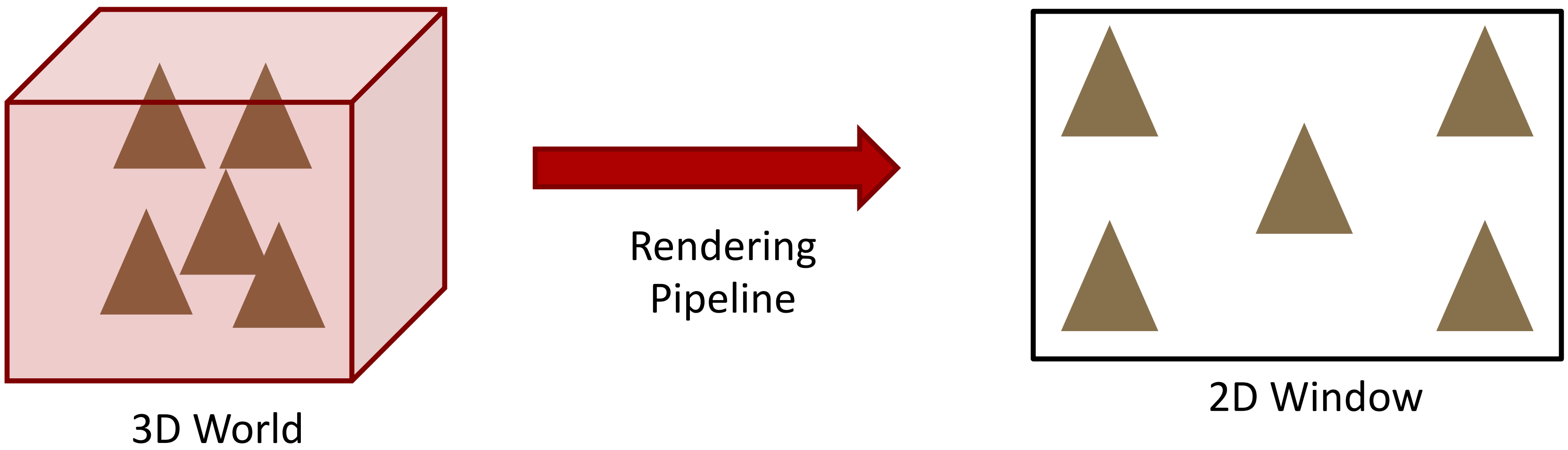
CENG 477
Introduction to Computer Graphics

Forward Rendering Pipeline
Clipping and Culling

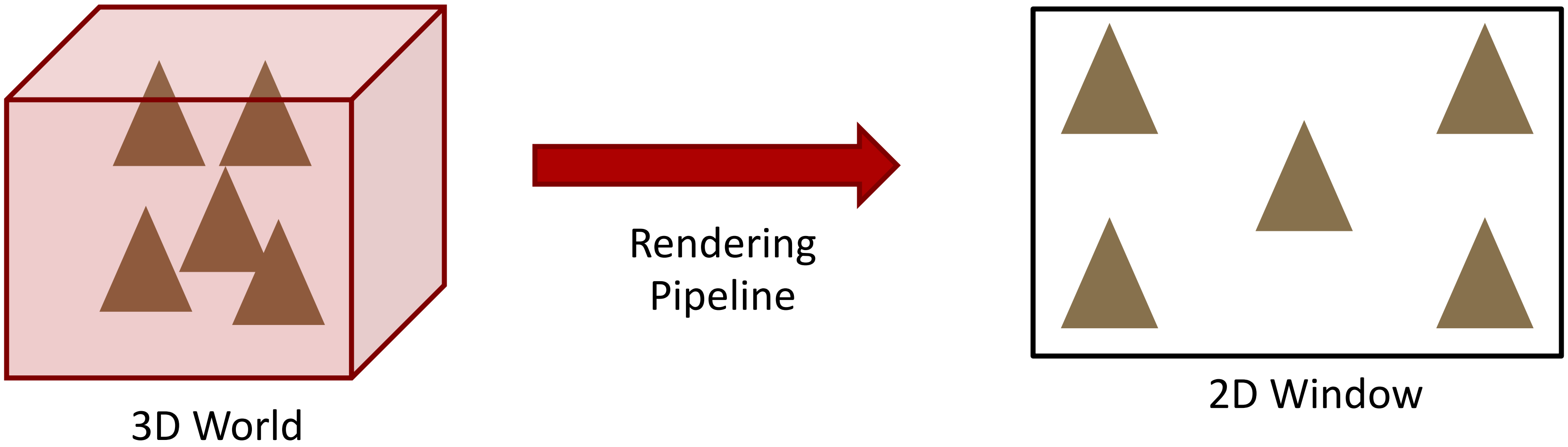
Rendering Pipeline

- Sequence of operations that are used to draw primitives defined in a 3D coordinate system on a 2D window
- Can be implemented on hardware or software
- Two notable APIs: OpenGL and D3D
- Not static: Constantly evolving to meet the demands of the industry

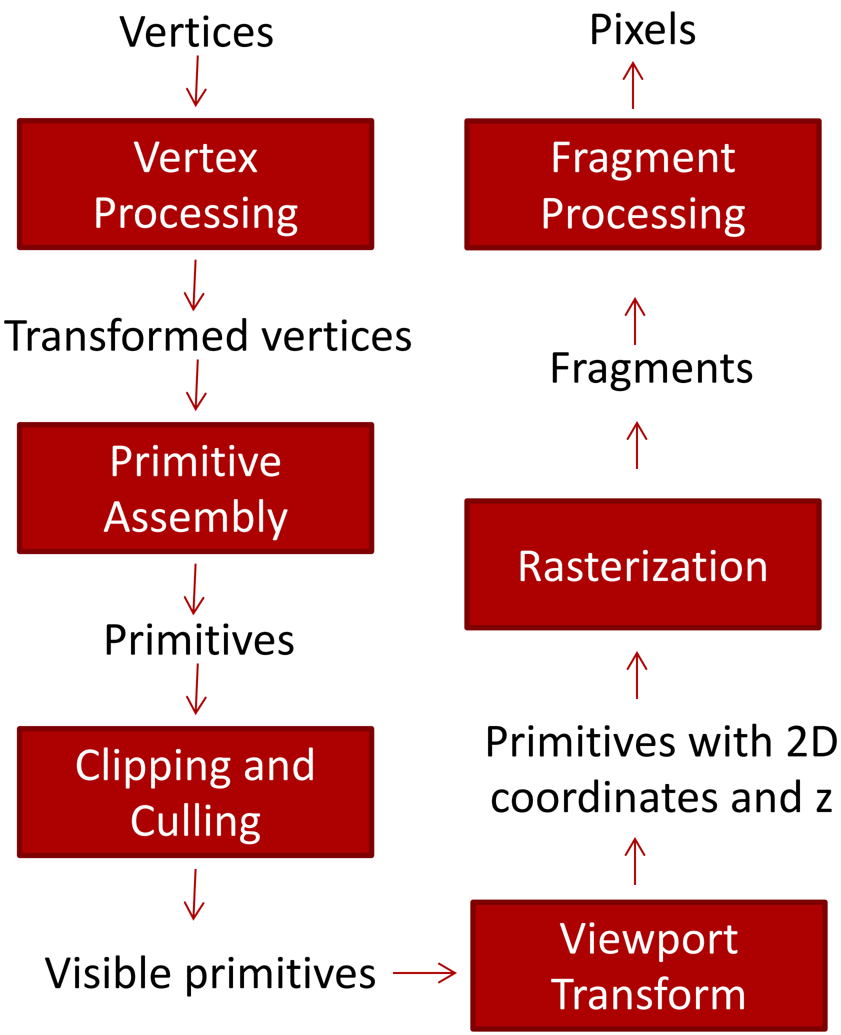


Rendering Pipeline

- Official versions of OpenGL released to date are 1.0, 1.1, 1.2, 1.2.1, 1.3, 1.4, 1.5, 2.0, 2.1, 3.0, 3.1, 3.2, 3.3, 4.0, 4.1, 4.2, 4.3, 4.4, 4.5, 4.6 (2017)

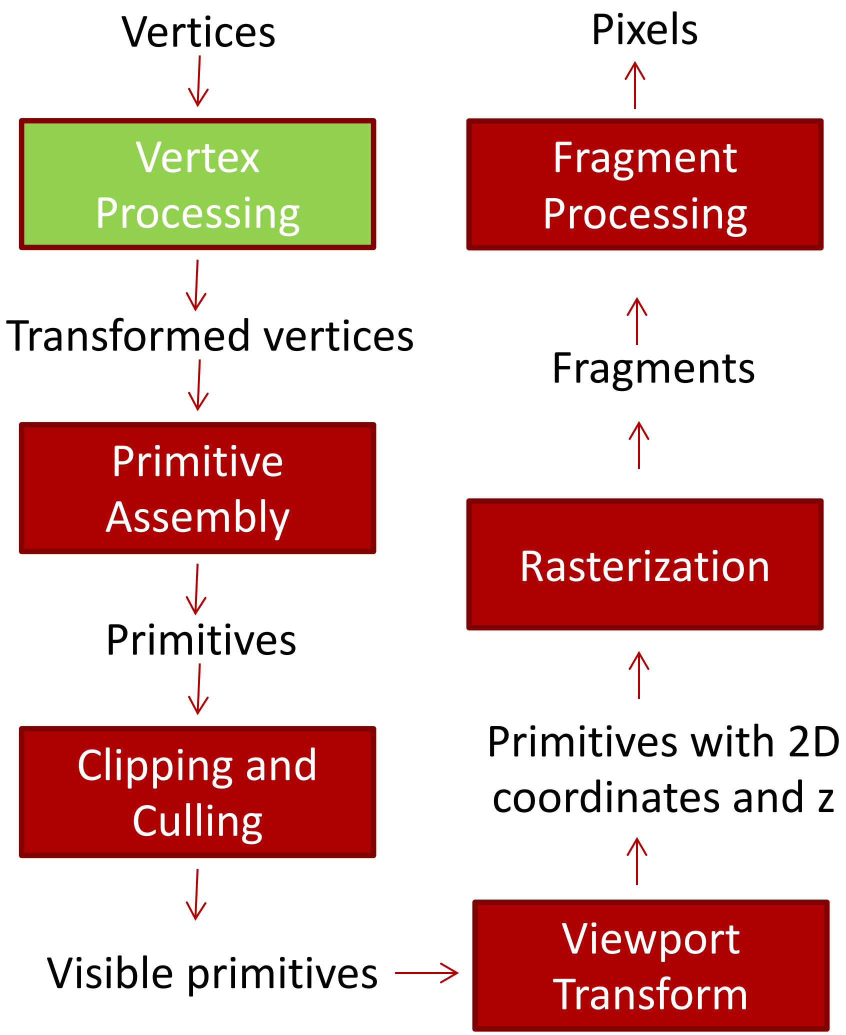


Rendering Pipeline – Overview



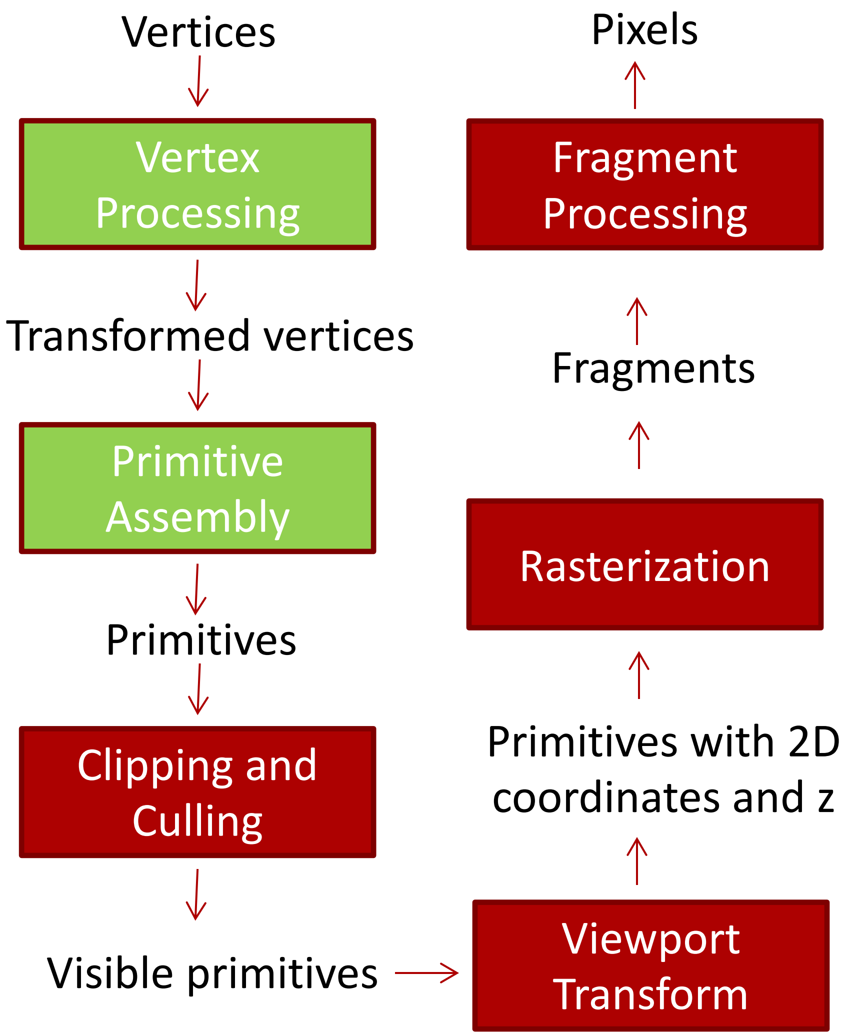
1. Get vertices in specific order and connectivity information
2. Process (transform) vertices
3. Create primitives from connected vertices
4. Clip and cull primitives to eliminate invisible ones
5. Transform primitives to screen space (preserve z)
6. Rasterize primitives to obtain fragments
7. Process fragments to obtain visible pixels with color

Rendering Pipeline – Overview



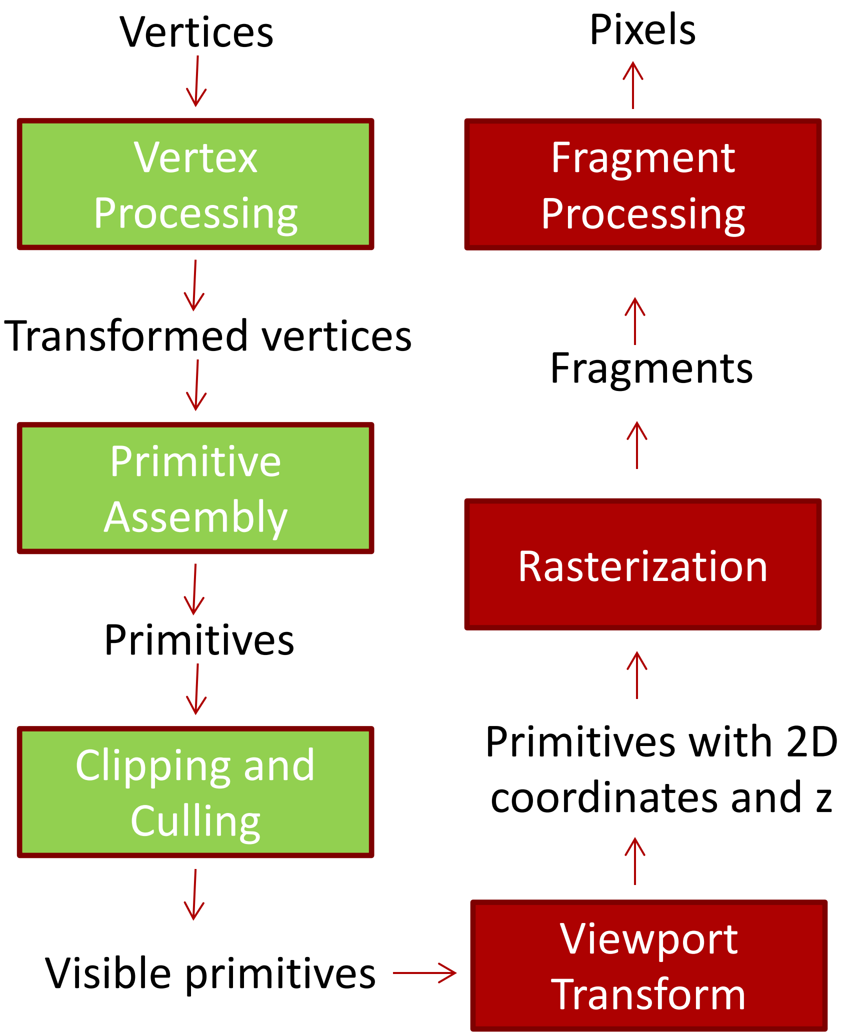
- We learned about vertex processing
  - Modeling transformations
  - Camera transformations
  - Projection transformations

Rendering Pipeline – Overview



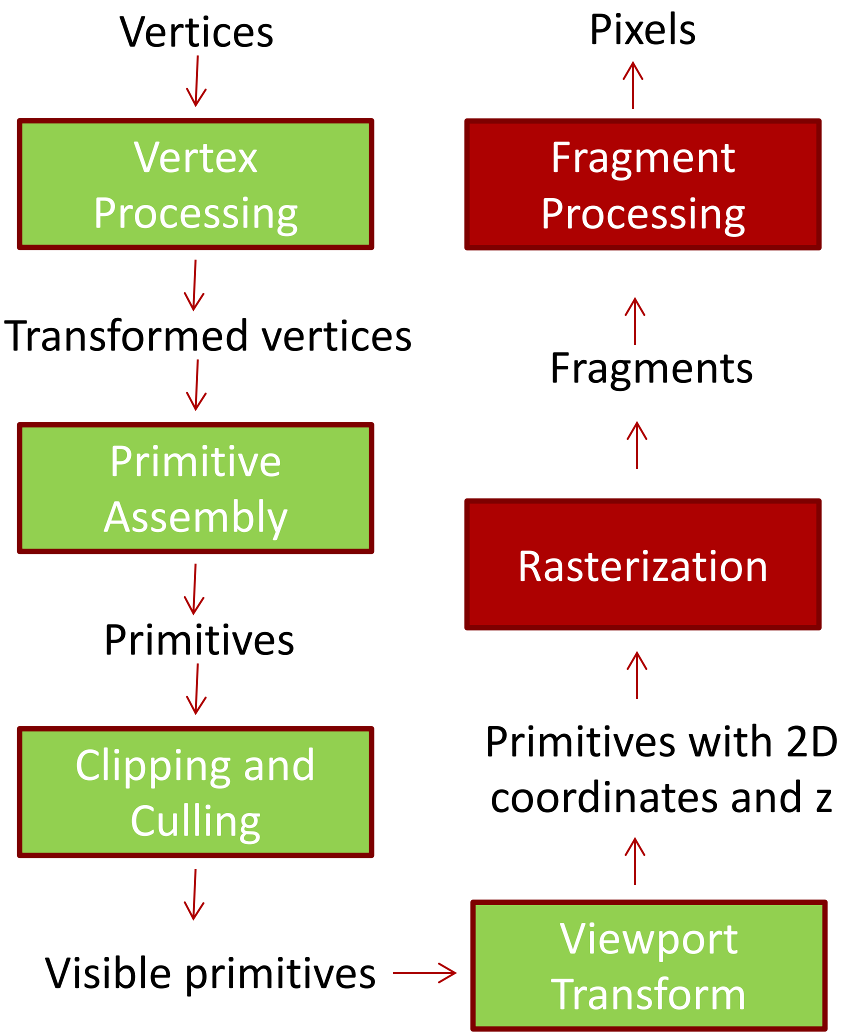
- Primitive assembly is the process of grouping of vertices to create primitives
  - Lines
  - Triangles
  - Quadrangles
  - …
- This is just logical grouping, no actual objects are visible at this point

Rendering Pipeline – Overview



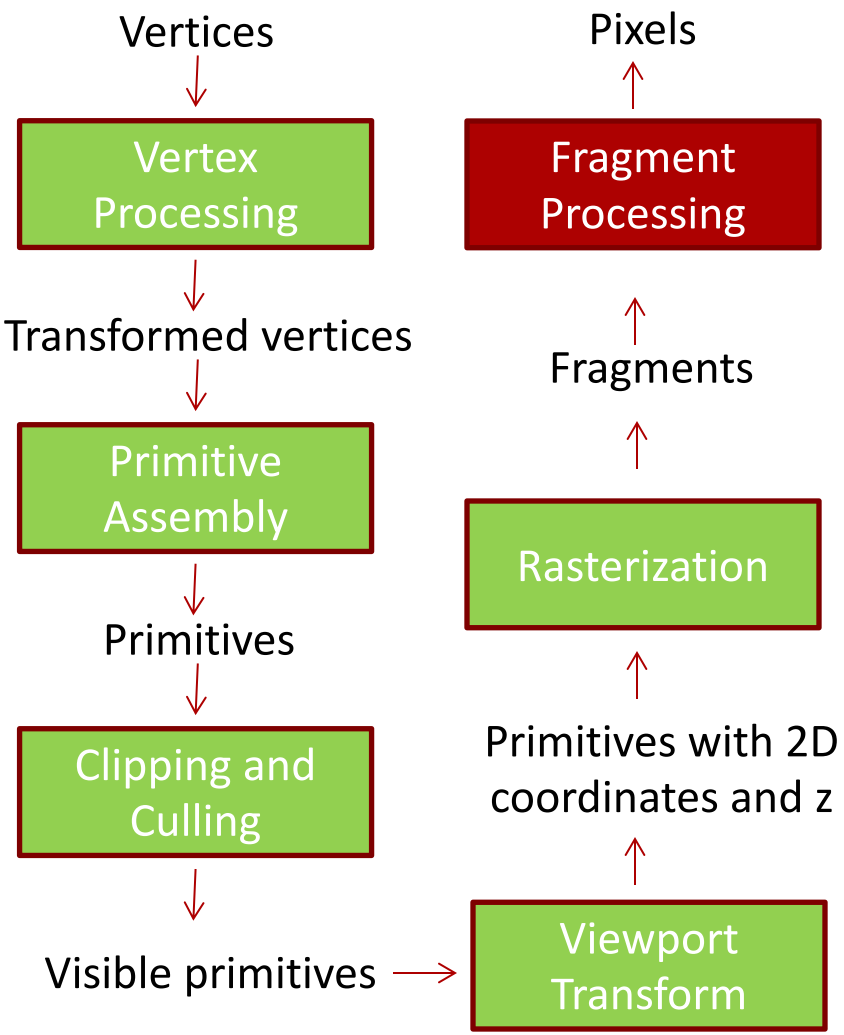
- During clipping and culling, primitives outside the CVV must be culled
- Primitives partially inside the CVV must be clipped
  - May produce new vertices

Rendering Pipeline – Overview



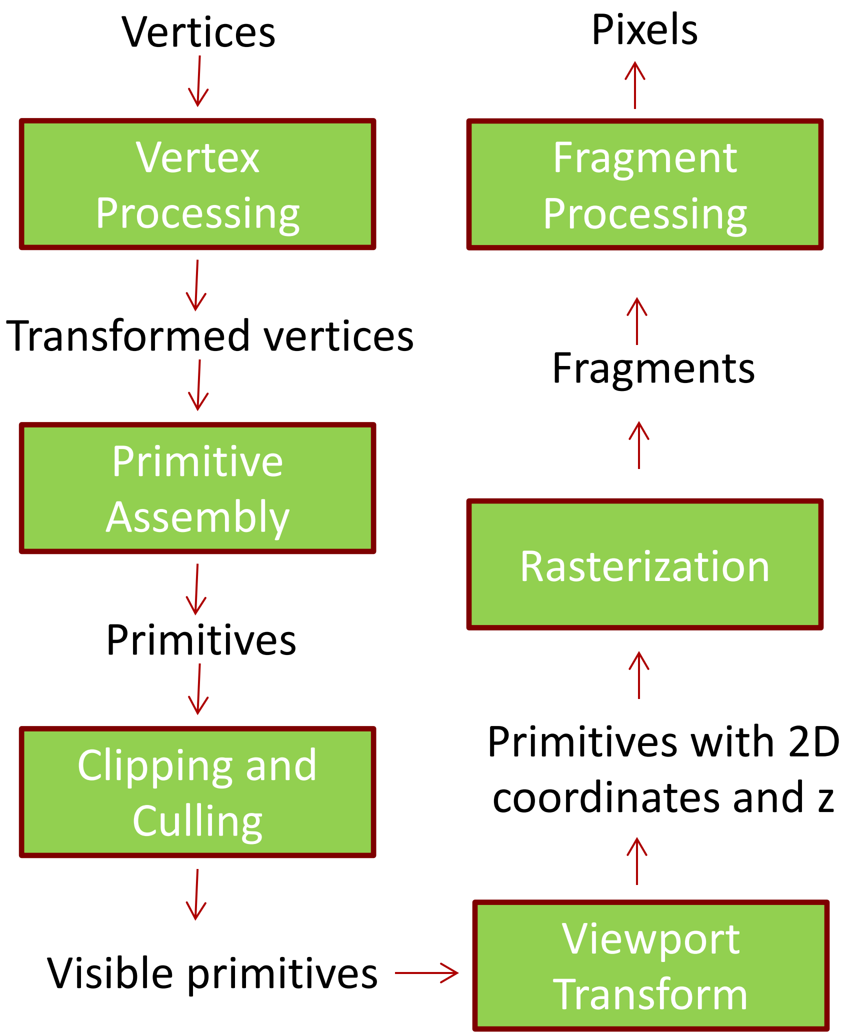
- With viewport transform, all surviving (and potentially clipped) primitives acquire viewport coordinates
  - We work on integer pixel coordinates from this point on
  - Depth can be fixed point or floating point number in [0, 1] range (unless changed by glDepthRange)

Rendering Pipeline – Overview



- Rasterization is the process of determining the pixels (fragments) that make up a primitive
  - Different algorithms for lines, triangles, etc.

Until Now



- Finally, each fragment may be further processed before being a visible pixel on the screen (or being written to a memory buffer)
  - Depth testing
  - Alpha blending
  - …
- This has a pipeline on its own

Clipping

- In modern graphics API, there are essentially three kinds of primitives: points, lines, and triangles
- Point clipping: straightforward
  - Reject a point if its coordinates are outside the viewing volume
- Line clipping
  - Cohen-Sutherland algorithm
  - Liang-Barsky algorithm
- Polygon clipping
  - Sutherland-Hodgeman algorithm

Clipping

- Clipping is done in the clip space which is a result of applying projection (orthographic or perspective) transformation
- After perspective transformation the *w* component of a point becomes equal to $-z$

$$M_{per} = \begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Clipping

- To find the actual point in the canonical viewing volume, we divide by this last component
- However, clipping is performed before dividing by *w* (that is **–** *z*) for several reasons:
  - *w* may be equal to 0 in which case division would be undefined
  - Instead of comparing $-1 \leq \frac{x}{w} \leq 1$ we can directly compare $-w \leq x \leq w$ thus avoiding an extra division for vertices that will be clipped
  - The same goes for y and z components
  - Finally division by w may make objects behind the viewer to come in-front of the viewer
  - That is why in the following we don't clip against -1 and 1 but against arbitrary numbers (it is also possible to define user clip planes which may have arbitrary values)

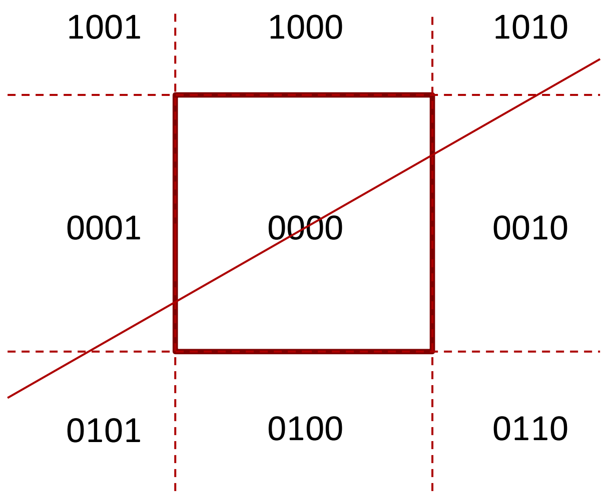
Clipping

- For simplicity, however, in the following we assume that clipping is performed against a 2D box with coordinates between [x_{min}, x_{max}] and [y_{min}, y_{max}]
- The same ideas can be easily generalized to 3D
- Line clipping:
  - Cohen-Sutherland Algorithm
  - Liang-Barsky Algorithm
- Polygon clipping:
  - Sutherland-Hodgeman Algorithm

Cohen-Sutherland Algorithm

- Assign outcodes to the end points of lines:
  - Bit0 = 1 if region is to the left of left edge, 0 otherwise
  - Bit1 = 1 if region is to the right of right edge, 0 otherwise
  - Bit2 = 1 if region is below the bottom edge, 0 otherwise
  - Bit3 = 1 if region is above the top edge, 0 otherwise

Outcodes for this line are: 0101 and 1010



How many regions would we have in 3D?

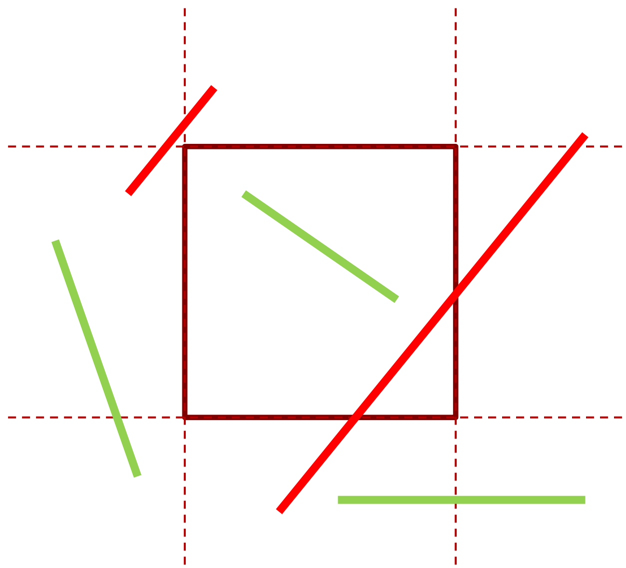
How bits would we need?

Cohen-Sutherland Algorithm

- Handle **trivial accept** and **trivial rejects** first:
  - If both outcodes are zero (i.e. their BITWISE OR is zero) accept the line as it is
  - If BITWISE AND of outcodes are non-zero, reject the line entirely

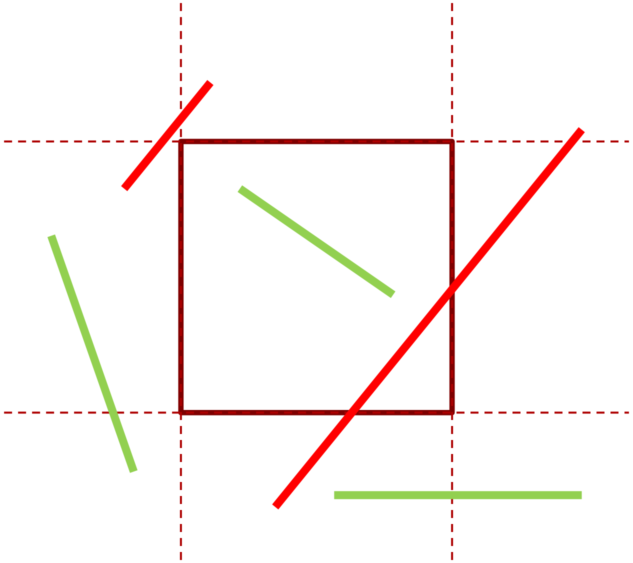
Trivial accept/reject lines

Non-trivial cases



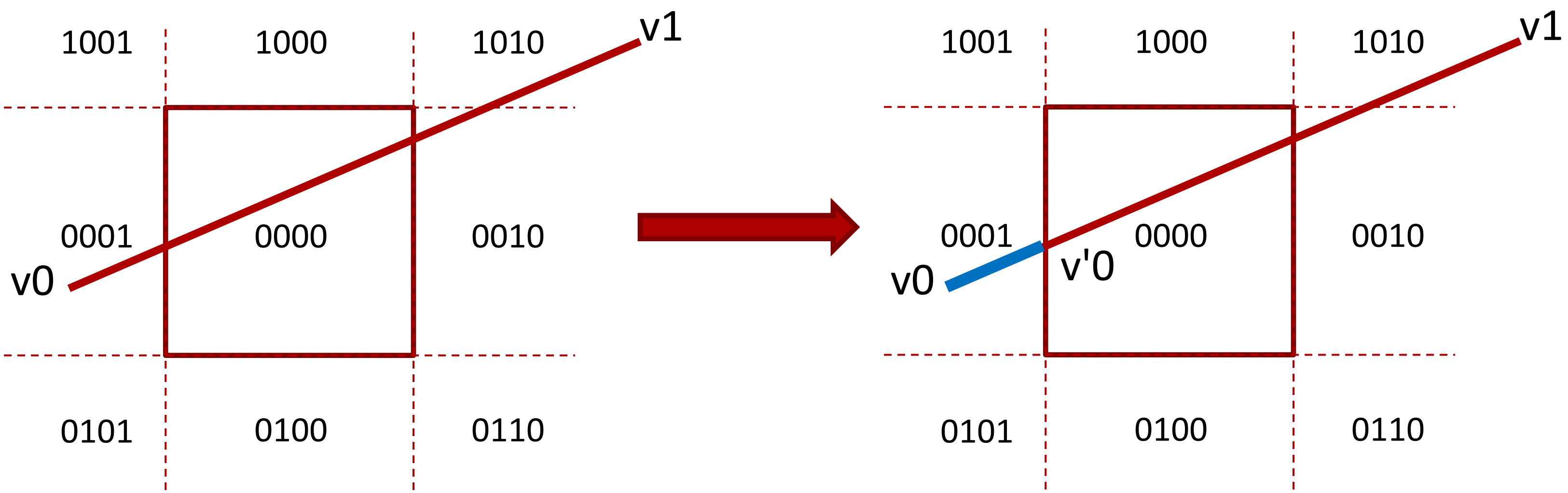
Cohen-Sutherland Algorithm

- For non-trivial cases, iteratively subdivide lines until all parts can be trivially accepted and rejected
  - Iteration follows a fixed order (e.g. left, right, bottom, top)

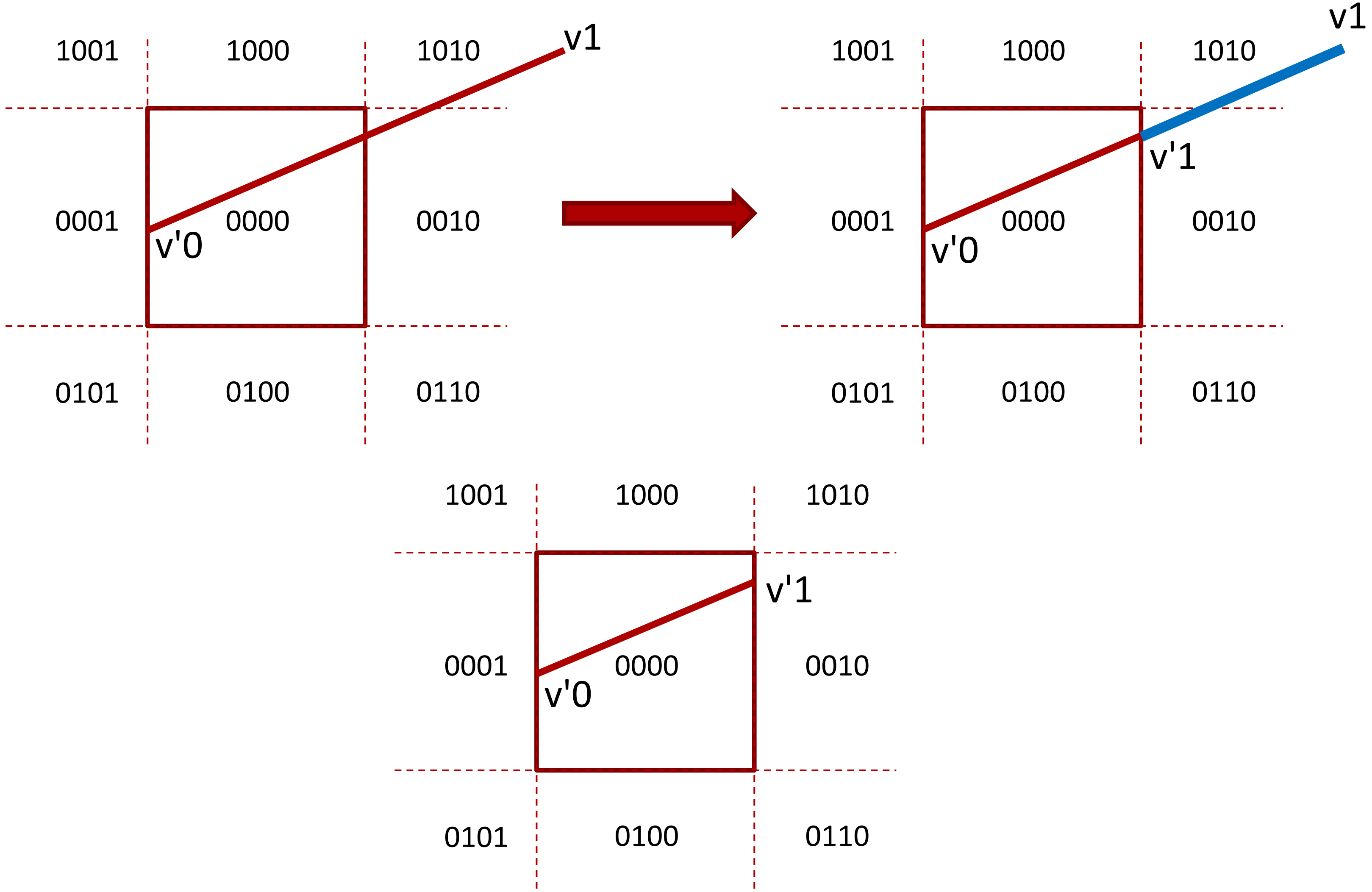


Cohen-Sutherland Algorithm

- Non-trivial cases:
  - **Step 1:** Pick an **outside** endpoint (v = v0 or v1)
  - **Step 2:** Pick an edge for which v is **outside**
  - **Step 3:** Intersect the line with that edge creating a new endpoint v’
  - **Step 4: Recompute** the outcode of v’ and go to step 1 unless trivial reject or trivial accept is possible

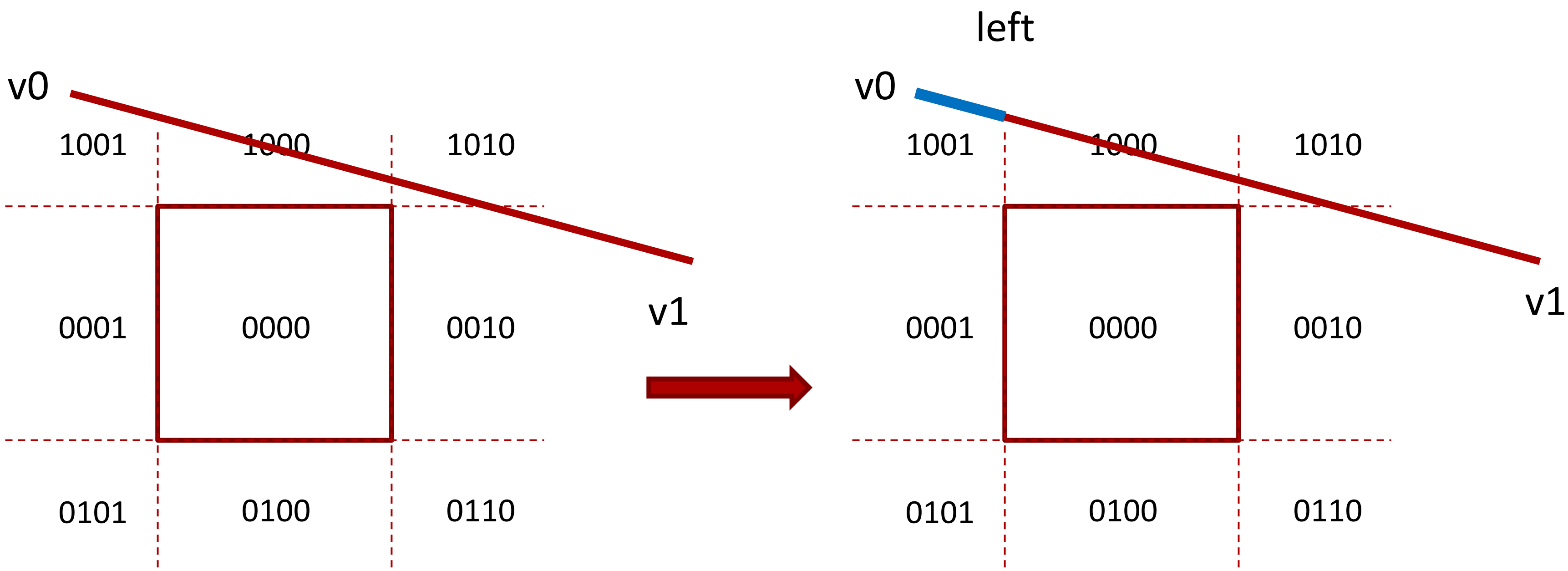


Cohen-Sutherland Algorithm



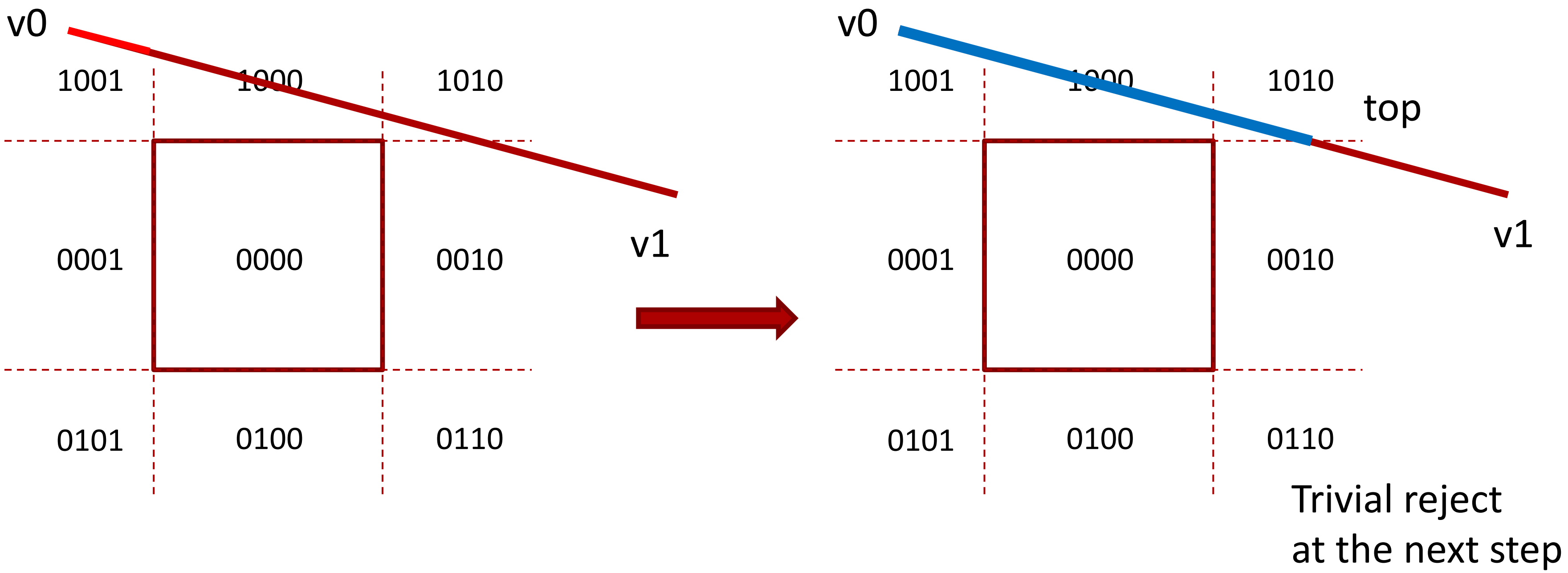
Cohen-Sutherland Algorithm

- May perform needless clipping:



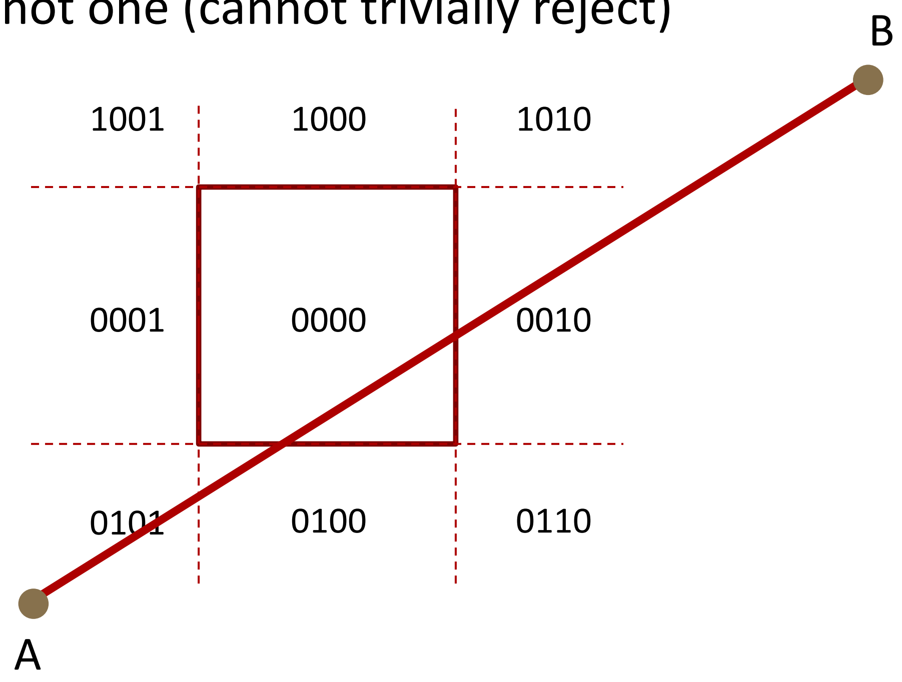
Cohen-Sutherland Algorithm

- May perform needless clipping:



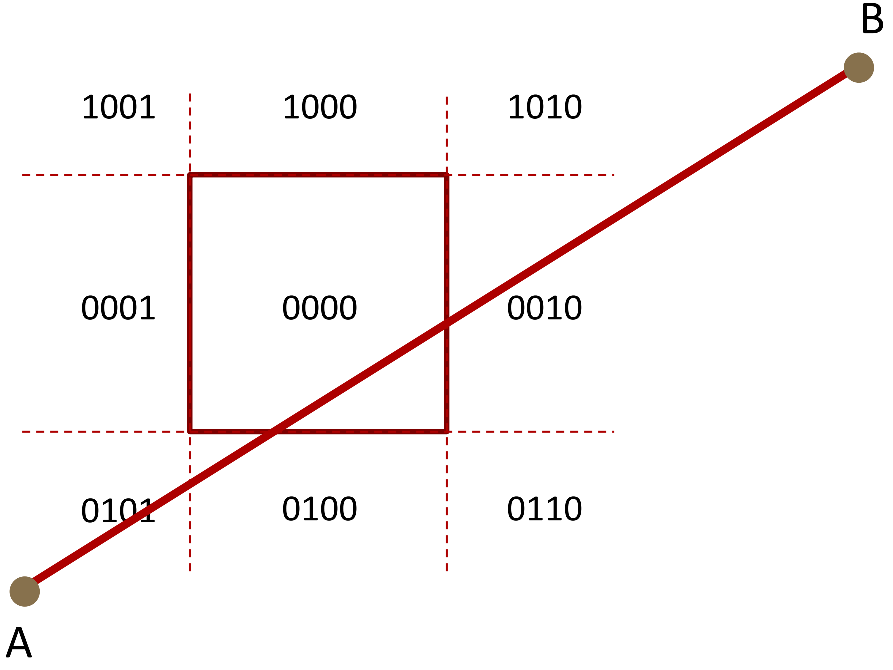
Cohen-Sutherland Algorithm

- Another example:
  - Bitwise OR not zero (cannot trivially accept)
  - Bitwise AND not one (cannot trivially reject)



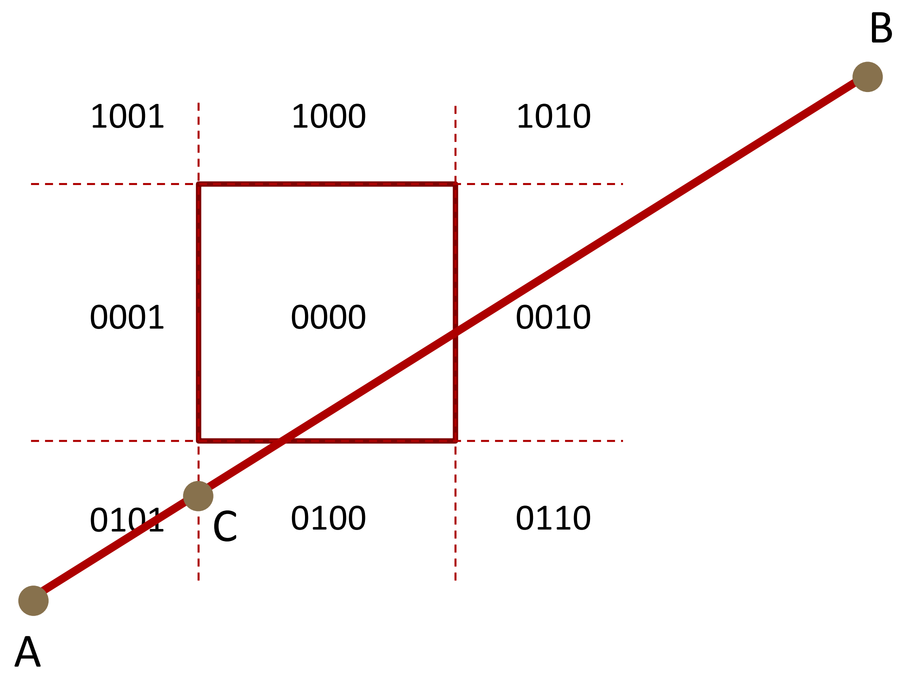
Cohen-Sutherland Algorithm

- Pick an outside vertex, A
- Pick an edge that is outside of: 010**1** => left



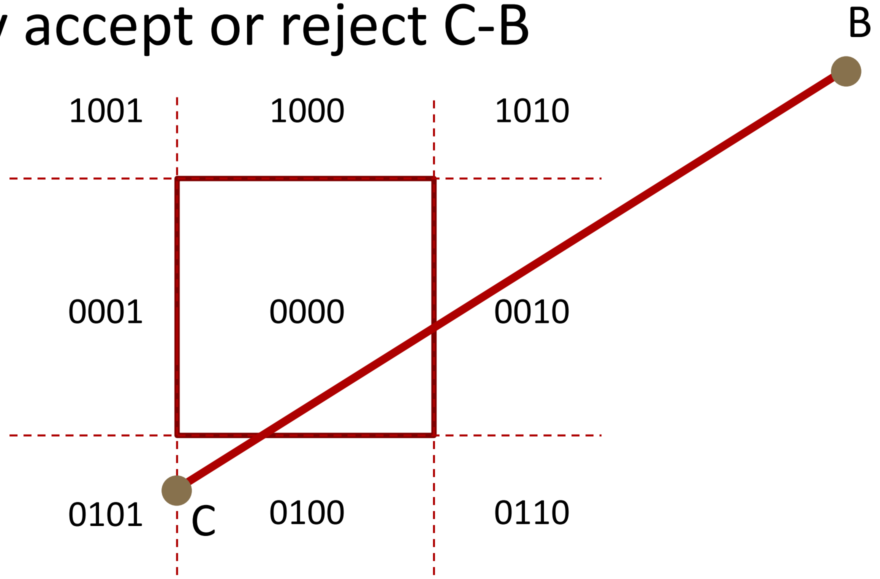
Cohen-Sutherland Algorithm

- Intersect to find C



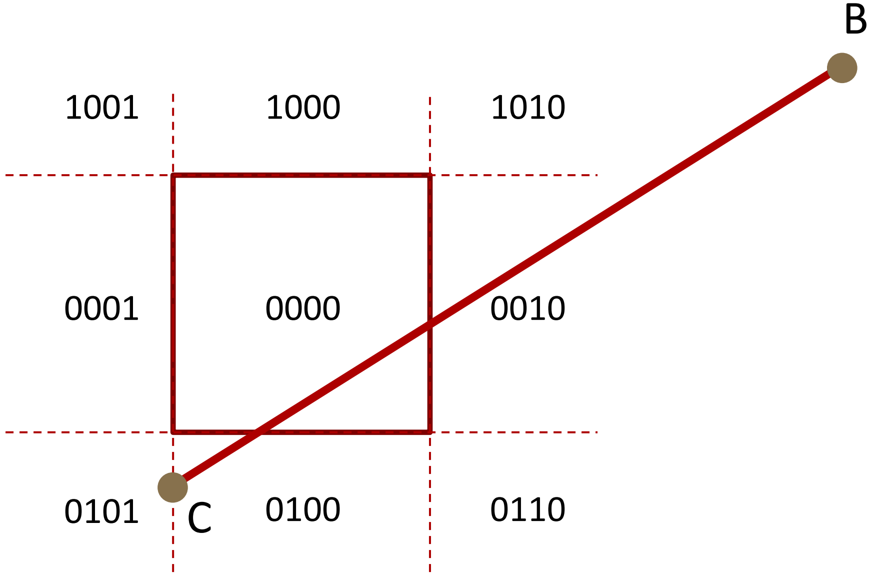
Cohen-Sutherland Algorithm

- Discard the A-C segment, C is your new endpoint
- Its outcode is 0100
- Cannot trivially accept or reject C-B



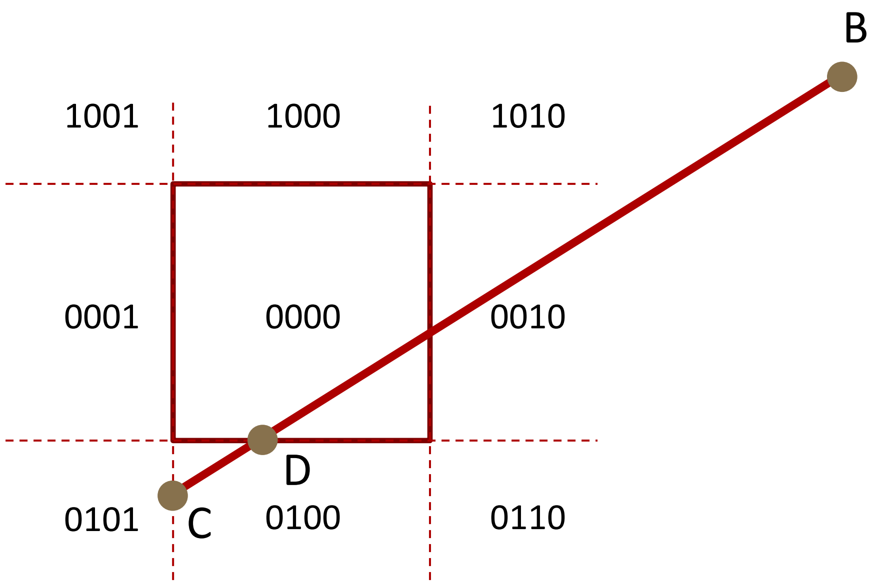
Cohen-Sutherland Algorithm

- Now pick an outside point, C for example
- Pick an edge that is outside of, 0**1**00 => bottom



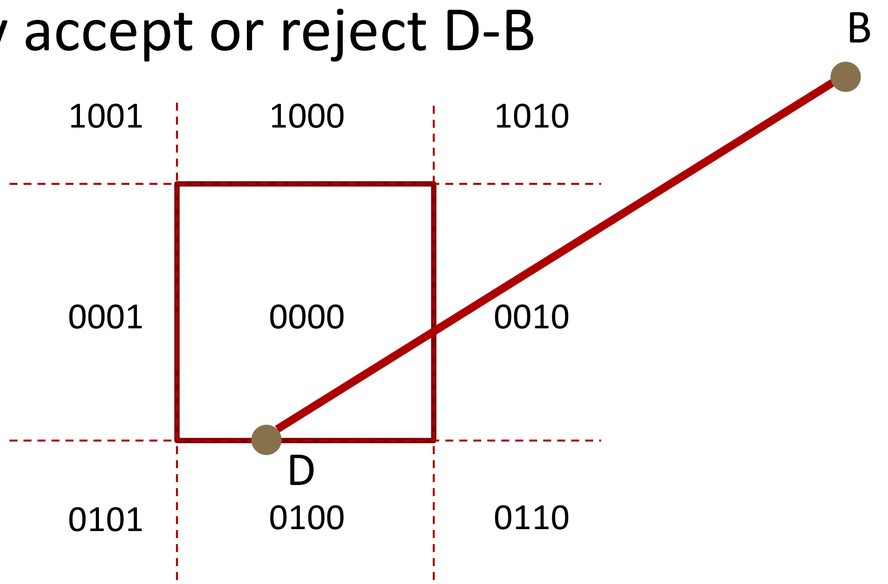
Cohen-Sutherland Algorithm

- Intersect to find D



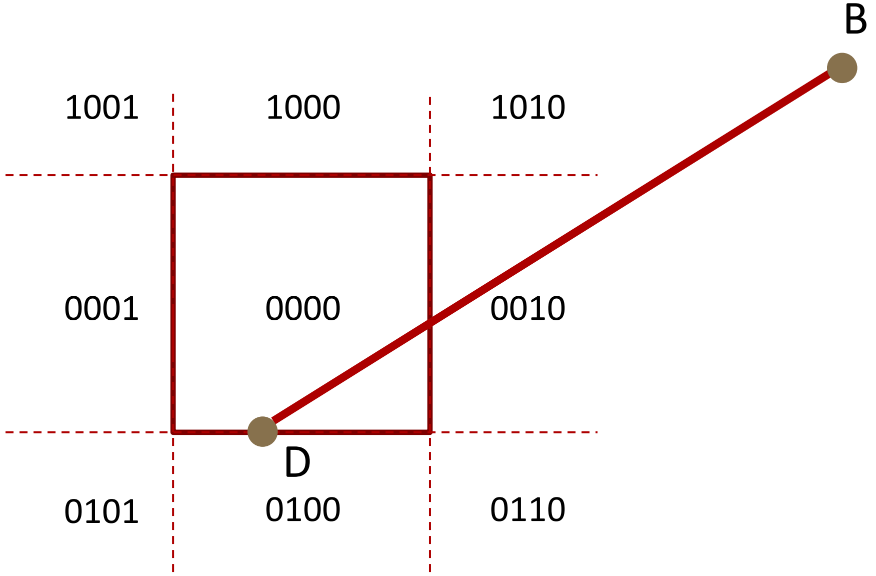
Cohen-Sutherland Algorithm

- Discard C-D, D is your new endpoint
- Its outcode is 0000
- Cannot trivially accept or reject D-B



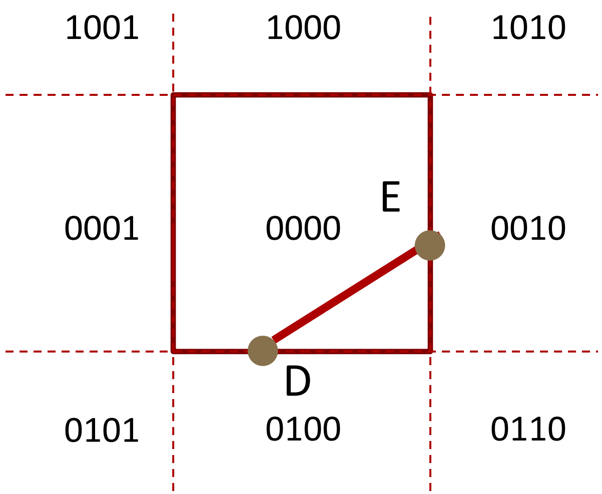
Cohen-Sutherland Algorithm

- Now pick B as your only outside point
- Pick an edge that is outside of, 10**1**0 => right



Cohen-Sutherland Algorithm

- Intersect to find E
- Discard E-B, E is your new endpoint
- E's outcode is 0000
- We can now trivially accept D-E as our clipped line



Line Intersections

- In 2D, we need to intersect a line with other lines
- In 3D, we need to intersect a line with planes
- We may use parametric form in both cases (similar to ray tracing)

$$v_0 = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \end{bmatrix}$$

$$v_1 = \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}$$

$$x(t) = x_0 + \boldsymbol{d}t$$

$$y(t) = y_0 + \boldsymbol{d}t$$

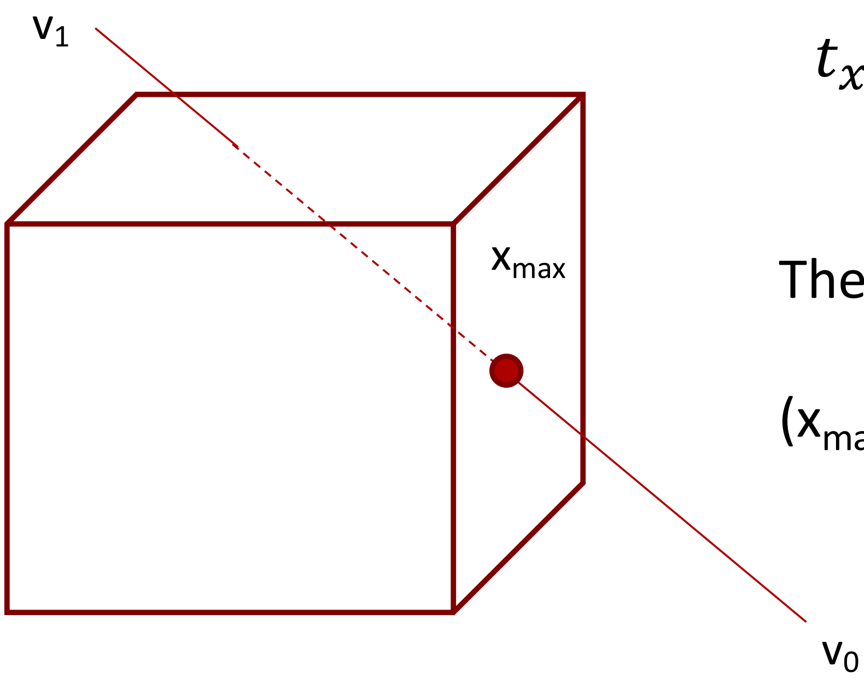
$$z(t) = z_0 + \boldsymbol{d}t$$

where

$$\boldsymbol{d} = \begin{bmatrix} x_1 - x_0 \\ y_1 - y_0 \\ z_1 - z_0 \end{bmatrix}$$

Line Intersections

- To find the intersection point, compute t corresponding to the given edge (or face) and then find the remaining values:



$$t_{xmax} = \frac{x_{max} - x_0}{x_1 - x_0}$$

The intersection point is at:

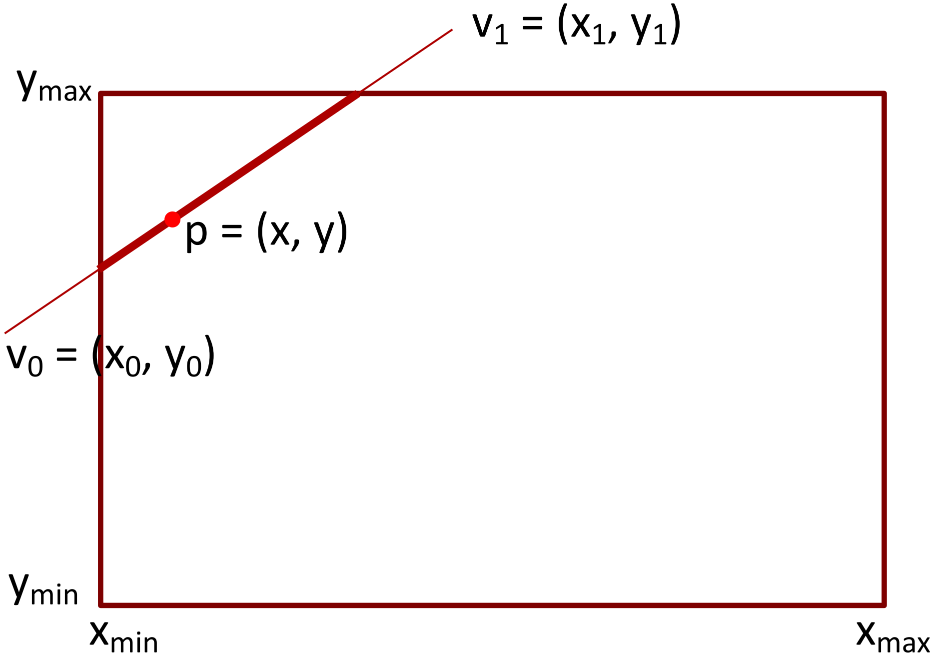
(x_{max}, $y(t_{xmax})$, $z(t_{xmax})$)

Cohen-Sutherland Algorithm

- Advantages:
  - If the chances of trivial accept/reject are high, this is a very fast algorithm
  - This can happen if the clipping rectangle is very large or very small
- Disadvantages:
  - Non-trivial lines can take several iterations to clip
  - Because testing and clipping are done in a fixed order, the algorithm will sometimes perform needless clipping

Liang-Barsky Algorithm

- Uses the idea of parametric lines
- Classifies lines as potentially entering and potentially leaving to speed up computation (approximately 40% speed-up over Cohen-Sutherland Alg.)

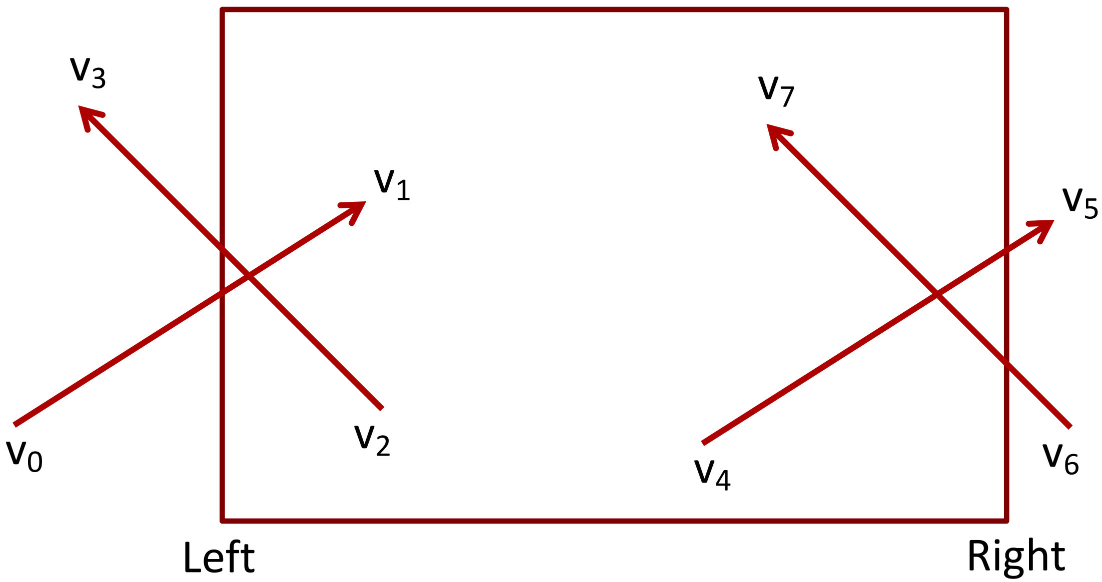


Goal: Given the line v0, v1 determine:
- The part of the line is inside the viewing rectangle.

Note: p = v0 + (v1-v0)t

Liang-Barsky Algorithm

- Potentially entering (PE) and leaving (PV):
- Why do we say potentially?



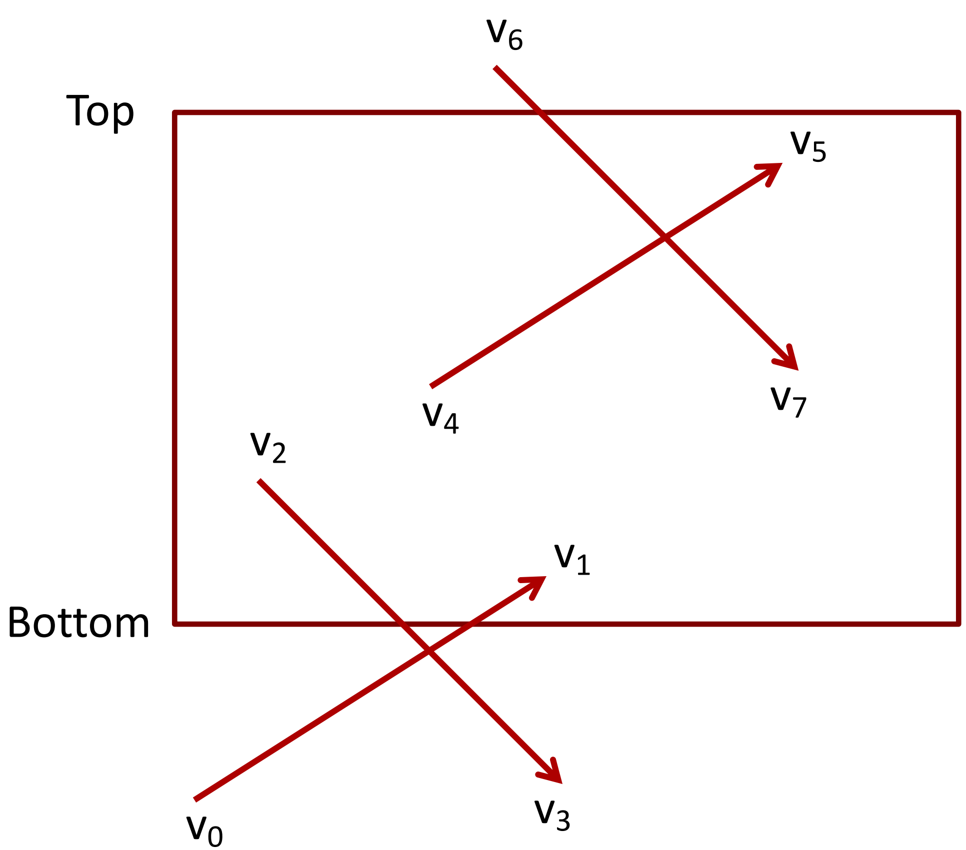
- v_0,v_1 is potentially entering the **left** edge as $x_1 - x_0 > 0$
- v_2,v_3 is potentially leaving the **left** edge as $x_3 - x_2 < 0$

The situation is reversed for the right edge:

- v_4,v_5 is potentially leaving the **right** edge as $x_5 - x_4 > 0$
- v_6,v_7 is potentially entering the **right** edge as $x_7 - x_6 < 0$

Liang-Barsky Algorithm

- Similar for bottom and top edges:



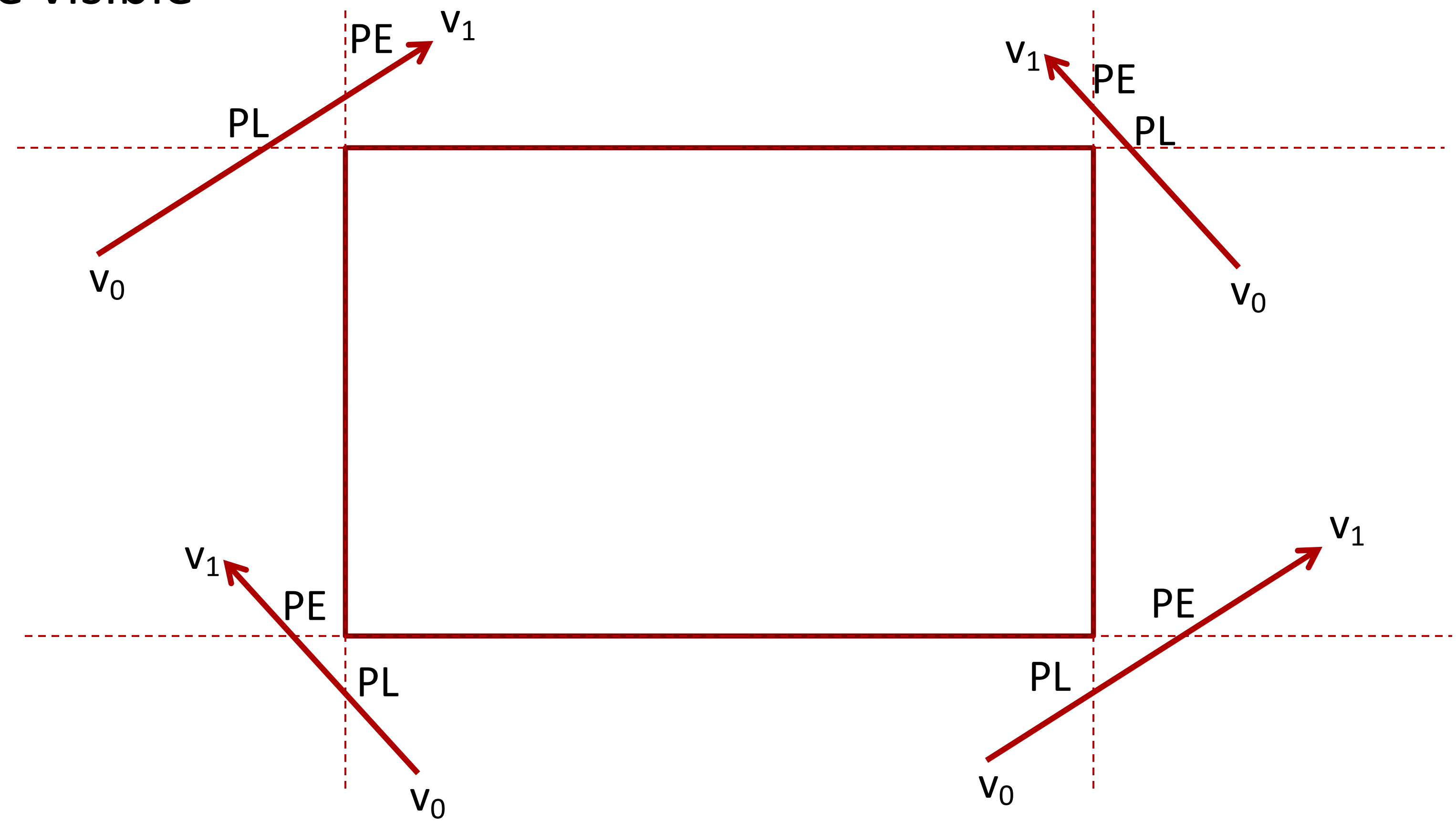
- v_0,v_1 is potentially entering the **bottom** edge as $y_1 - y_0 > 0$
- v_2,v_3 is potentially leaving the **bottom** edge as $y_3 - y_2 < 0$

The situation is reversed for the top edge:

- v_4,v_5 is potentially leaving the **top** edge as $y_5 - y_4 > 0$
- v_6,v_7 is potentially entering the **top** edge as $y_7 - y_6 < 0$

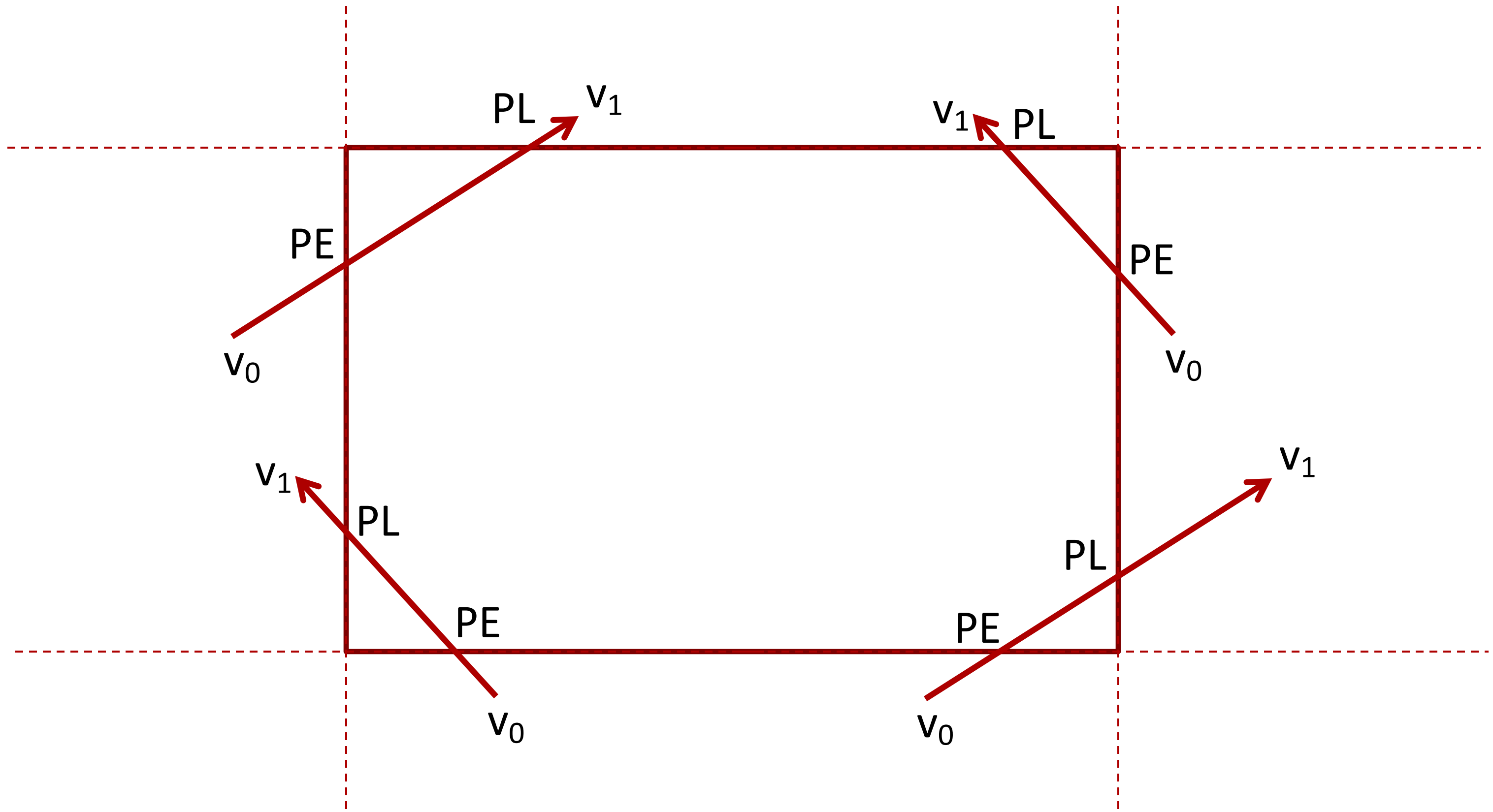
Liang-Barsky Algorithm

- **Observation:** If a line is first leaving then entering, it cannot be visible



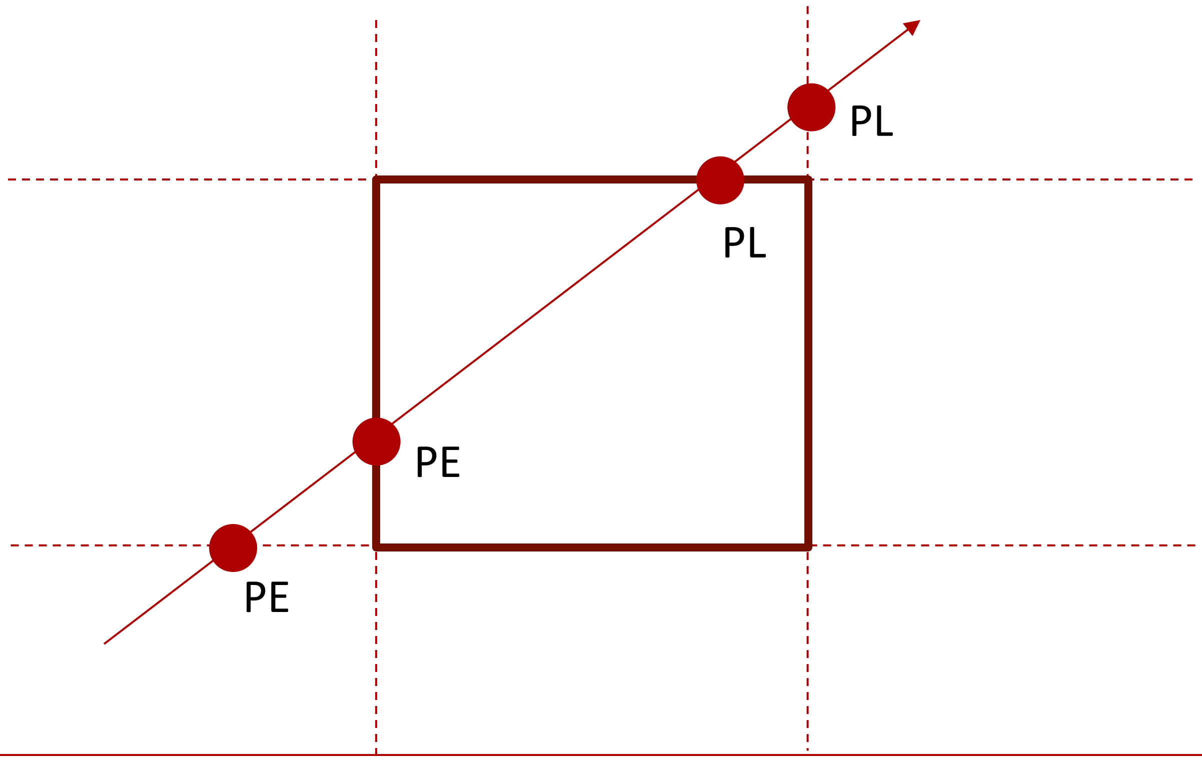
Liang-Barsky Algorithm

- Visible lines are first entering then leaving:



Liang-Barsky Algorithm

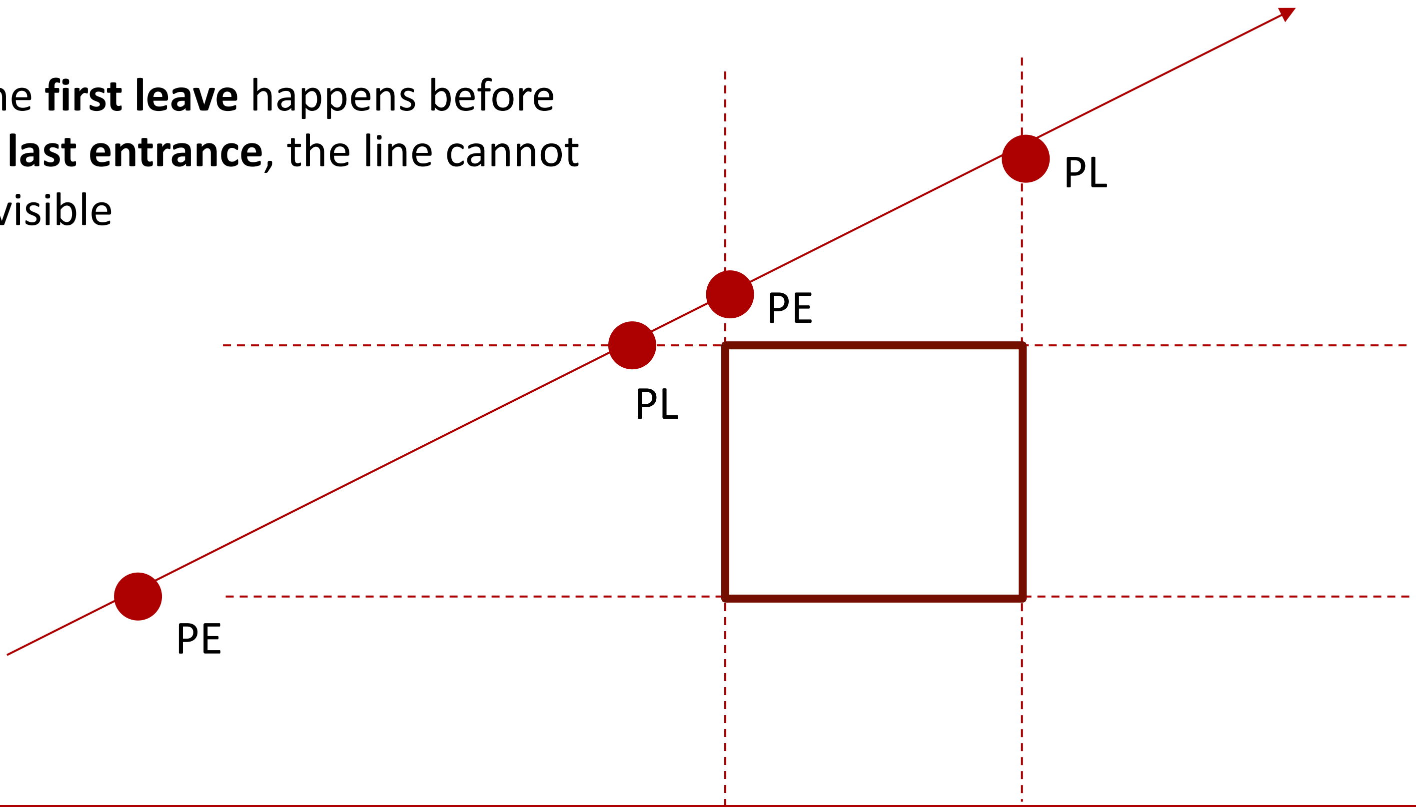
- Also note that for a 2D clipping rectangle, each line will enter and leave twice:



Liang-Barsky Algorithm

- Also note that for a 2D clipping rectangle, each line will enter and leave twice:

If the **first leave** happens before the **last entrance**, the line cannot be visible



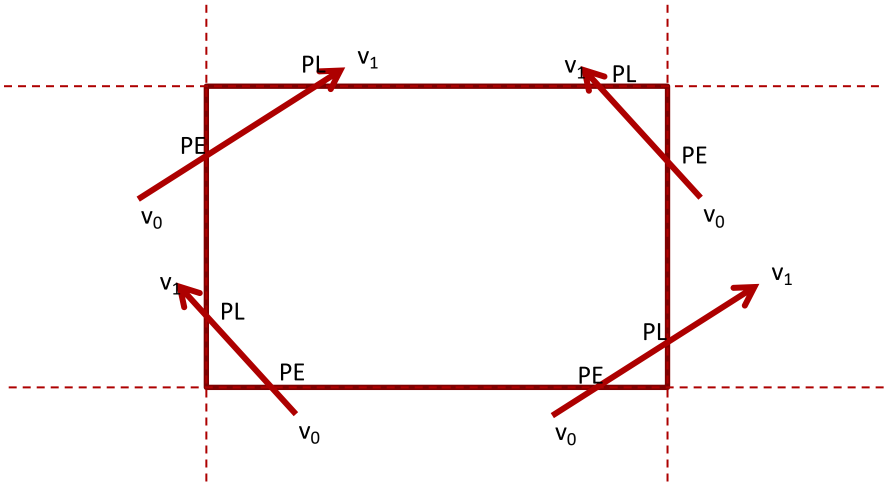
Liang-Barsky Algorithm

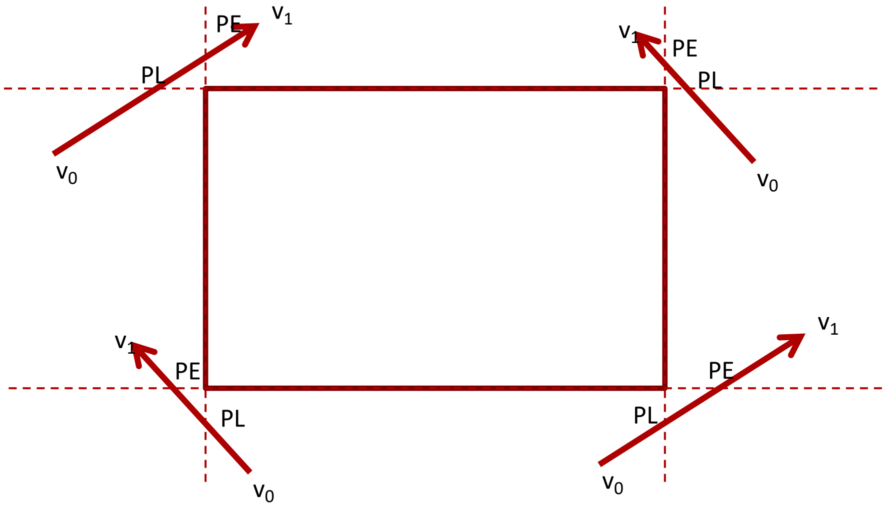
- Mathematical interpretation:

if ($t_{PL} < t_{PE}$):
    visible = **false**;

where t_{PL} is the t value for the **first** leaving intersection and t_{PE} is the t value for the **last** entering intersection

- So at intersection points, we need to compute the **t** value as well as whether the line is **PE** or **PL** at that point





Liang-Barsky Algorithm

- Computing **t** value at every edge:

x_{left} = x0 + (x1-x0)t ➔ t = (x_{left} – x0) / (x1 – x0)

x_{right} = x0 + (x1-x0)t ➔ t = (x_{right} – x0) / (x1 – x0)

y_{bottom} = y0 + (y1-y0)t ➔ t = (y_{bottom} – y0) / (y1 – y0)

y_{top} = y0 + (y1-y0)t ➔ t = (y_{top} – y0) / (y1 – y0)

- But this does not help us to know if line is entering or leaving at that point. **Solution:** look at the sign of dx, dy:

  - v_0,v_1 is potentially entering the **left** edge if dx = (x_1 – x_0) > 0
  - v_0,v_1 is potentially entering the **right** edge if dx = (x_1 – x_0) < 0 or –dx > 0
  - v_0,v_1 is potentially entering the **bottom** edge if dy = (y_1 – y_0) > 0
  - v_0,v_1 is potentially entering the **top** edge if dy = (y_1 – y_0) < 0 or –dy > 0

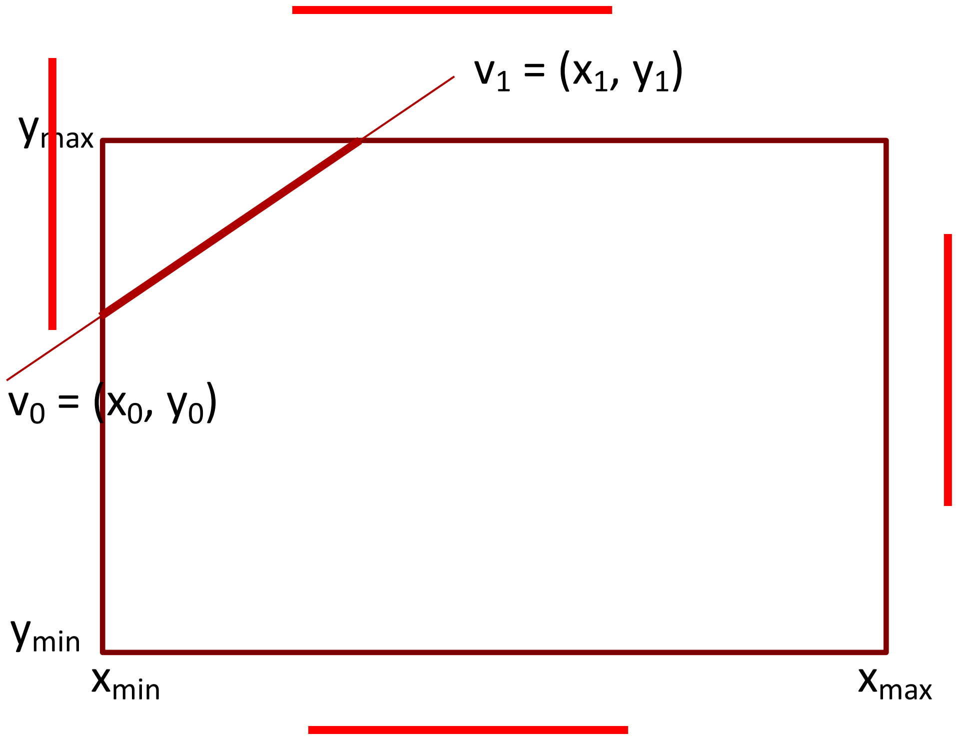
Liang-Barsky Algorithm

- Finding intersection type:
  - Entering left edge if $dx > 0$.
  - Entering right edge if $-dx > 0$.
  - Entering bottom edge if $dy > 0$.
  - Entering top edge if $-dy > 0$.
- Finding t:
  - For left edge: $t = (x_{left} – x_0) / (x_1 – x_0) = (x_{left} – x_0) / dx$
  - For right edge: $t = (x_{right} – x_0) / (x_1 – x_0) = (x_{right} – x_0) / dx$
    $= (x0 – x_{right}) / (-dx)$
  - For bottom edge: $t = (y_{bottom} – y_0) / (y_1 – y_0) = (y_{left} – y_0) / dy$
  - For top edge: $t = (y_{top} – y_0) / (y_1 – y_0) = (y_{top} – y_0) / dy$
    $= (y0 – y_{top}) / (-dy)$

Liang-Barsky Algorithm

- For lines parallel to edges:

if $d_x == 0$ and $x_{min} – x_0 > 0$: // left
  reject;
else if $d_x == 0$ and $x_0 - x_{max} > 0$: // right
  reject;
else if $d_y == 0$ and $y_{min} – y_0 > 0$: // bottom
  reject;
else if $d_y == 0$ and $y_0 - y_{max} > 0$: // top
  reject;



Liang-Barsky Algorithm

- Putting it all together:

```
tE = 0; tL = 1;
visible = false;
if visible(dx, xmin – x0, tE, tL): // left
    if visible (-dx, x0 – xmax, tE, tL): // right
        if visible (dy, ymin – y0, tE, tL): // bottom
            if visible (-dy, y0 – ymax, tE, tL): // top
                visible = true;
                if (tL < 1):
                    x1 = x0 + dx tL;
                    y1 = y0 + dy tL;
                if (tE > 0):
                    x0 = x0 + dx tE;
                    y0 = y0 + dy tE;
```

```
bool visible(den, num, tE, tL):
    if (den > 0): // potentially entering
        t = num / den;
        if (t > tL):
            return false;
        if (t > tE)
            tE = t;
    else if (den < 0): // potentially leaving
        t = num / den;
        if (t < tE):
            return false;
        if (t < tL)
            tL = t;
    else if num > 0: // line parallel to edge
            return false;
    return true;
```

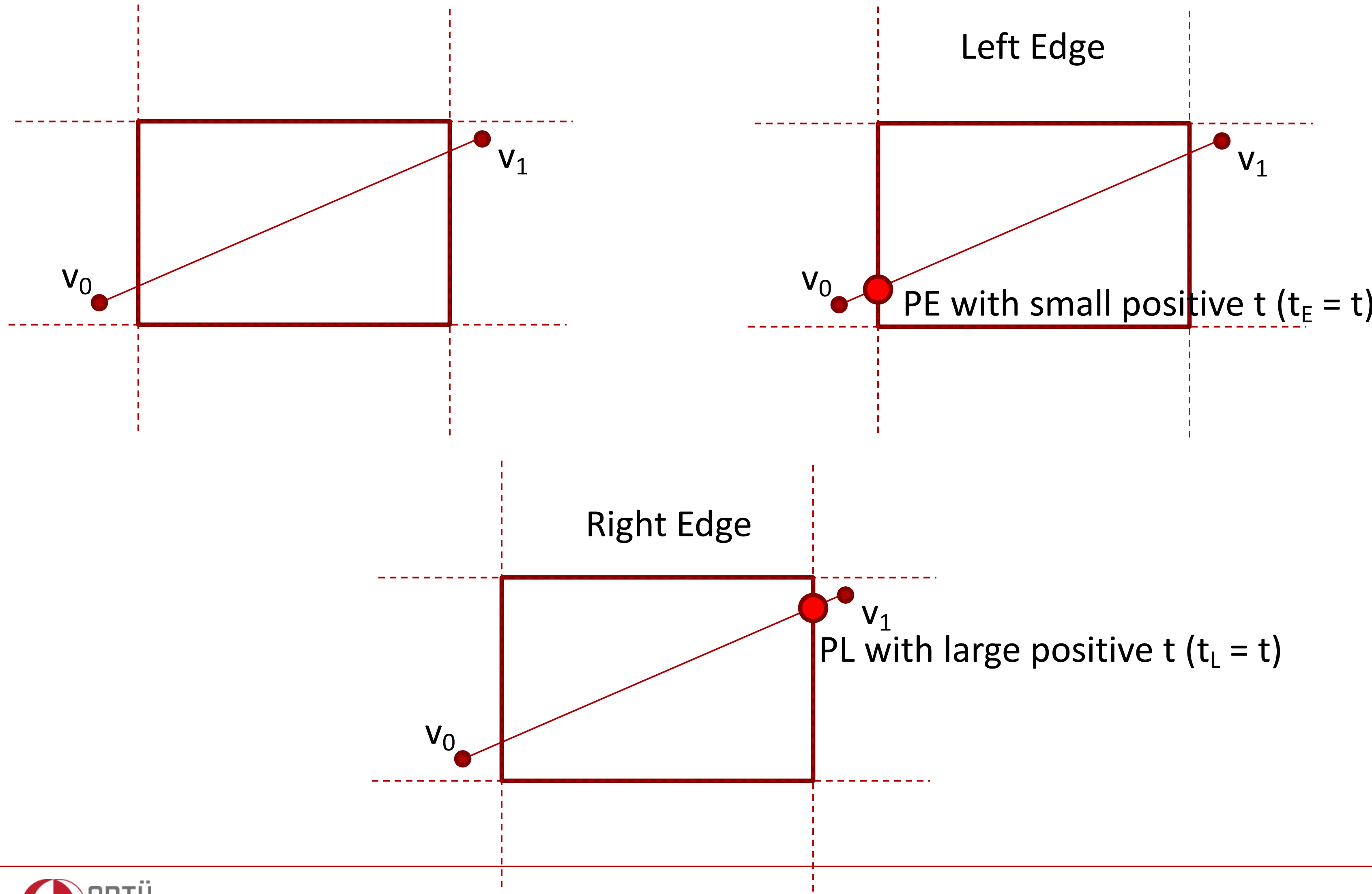
Liang-Barsky Algorithm

- 3D extension is straightforward:

$t_E = 0$; $t_L = 1$;
visible = false;
if visible(d_x, $x_{min} - x_0$, t_E, t_L): // left
  if visible ($-d_x$, $x_0 - x_{max}$, t_E, t_L): // right
    if visible (d_y, $y_{min} - y_0$, t_E, t_L): // bottom
      if visible ($-d_y$, $y_0 - y_{max}$, t_E, t_L): // top
        if visible (d_z, $z_{min} - z_0$, t_E, t_L): // front
          if visible ($-d_z$, $z_0 - z_{max}$, t_E, t_L): // back
            visible = true;
            if ($t_L < 1$):
              $x_1 = x_0 + d_x t_L$; $y_1 = y_0 + d_y t_L$; $z_1 = z_0 + d_z t_L$;
            if ($t_E > 0$):
              $x_0 = x_0 + d_x t_E$; $y_0 = y_0 + d_y t_E$; $z_0 = z_0 + d_y t_E$;

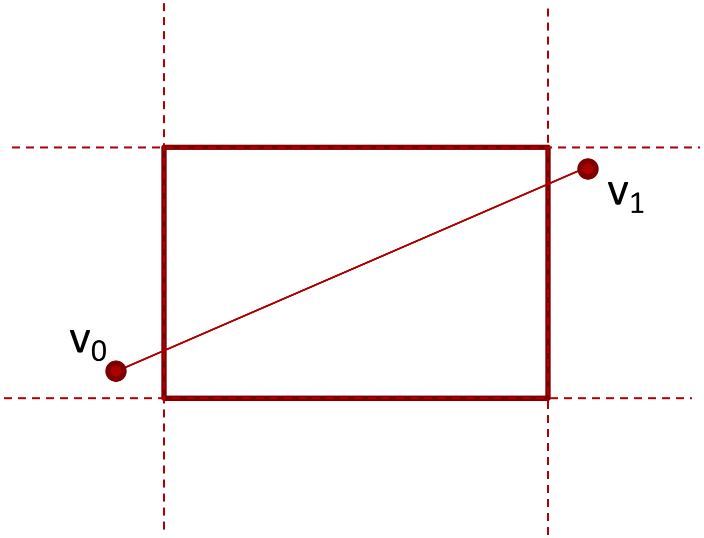
This part is used for efficient ray-bounding volume intersections in acceleration structures we learned earlier!

Example

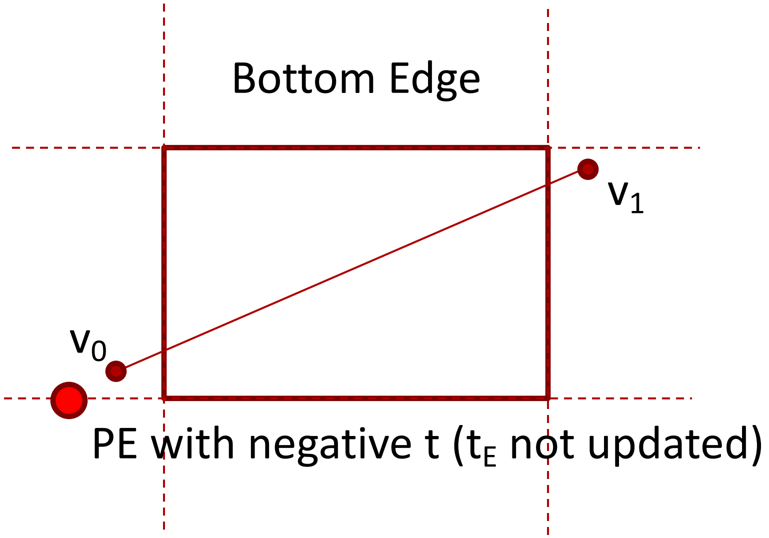


v_1
v_0
Left Edge
v_1
v_0
PE with small positive t (t_E = t)
Right Edge
v_1
PL with large positive t (t_L = t)
v_0

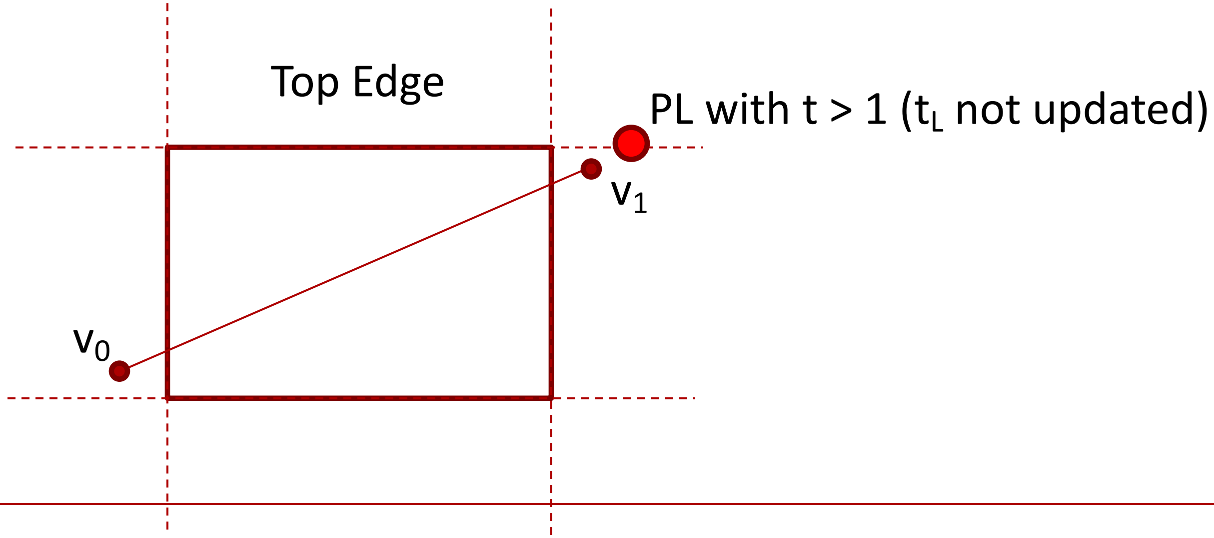
Example



v0
v1

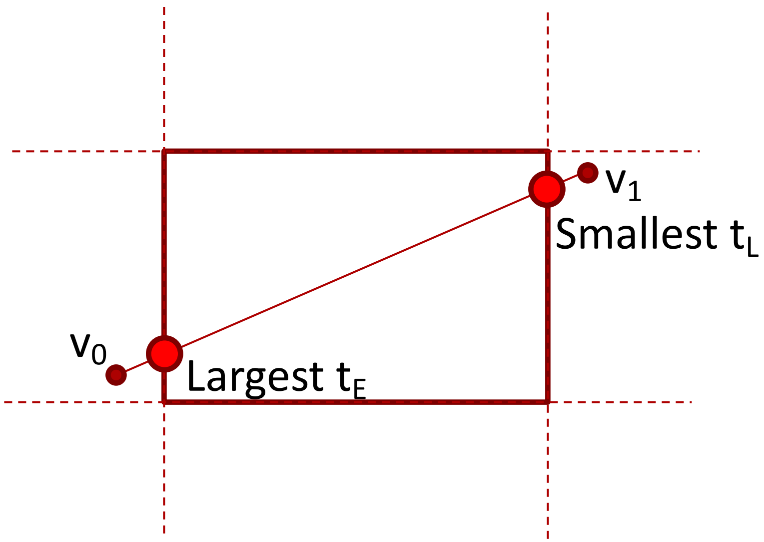


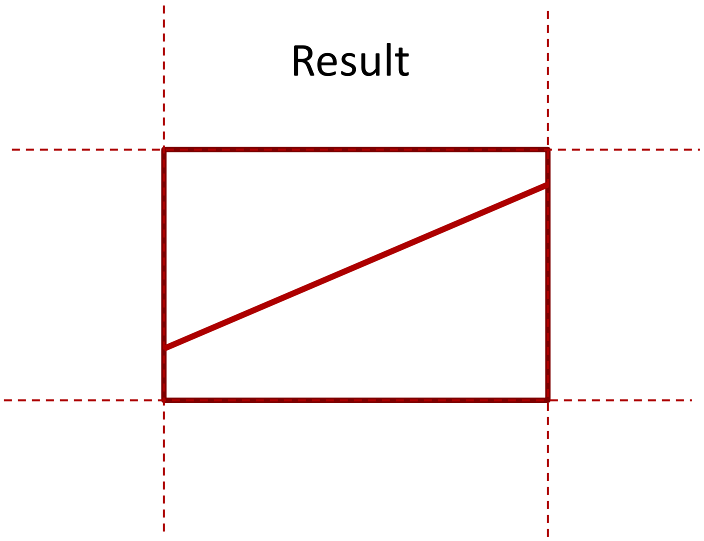
Bottom Edge
v0
v1
PE with negative t (tE not updated)



Top Edge
PL with t > 1 (tL not updated)
v0
v1

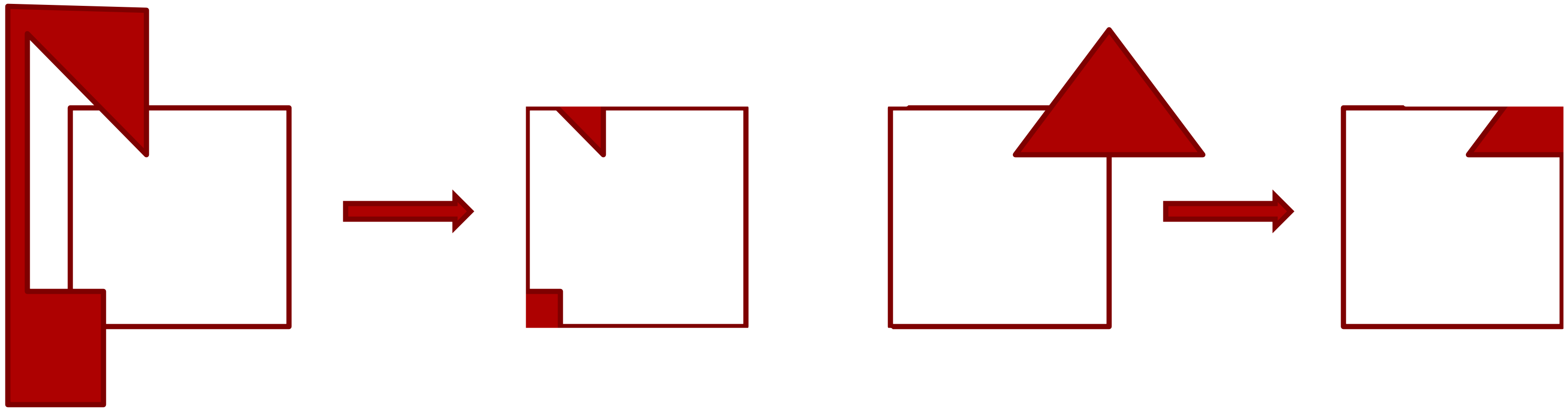
Example





Polygon Clipping – Sutherland Hodgeman Algorithm

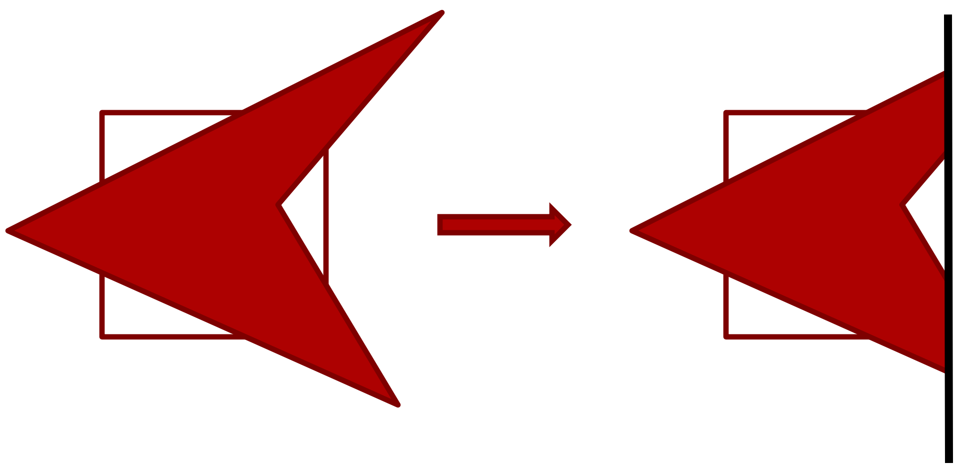
- Difficult problem as we need to deal with many cases:



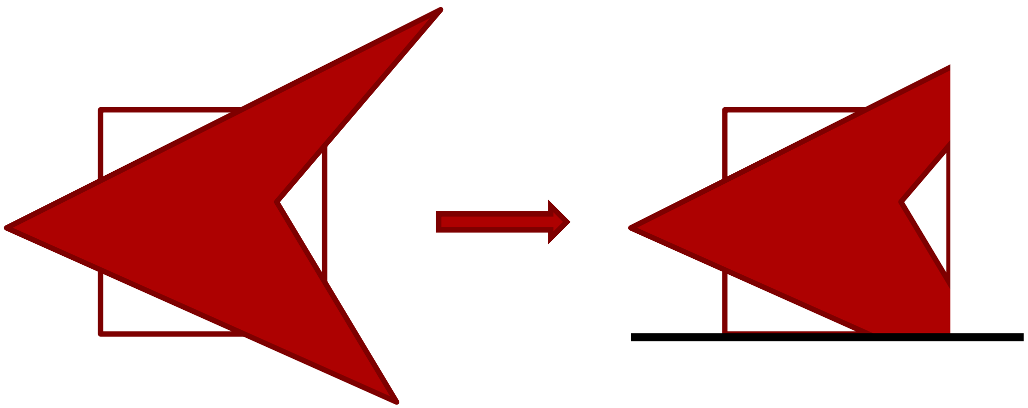
Sutherland Hodgeman Algorithm

- Divide and conquer approach makes it manageable:
  - Solve a series of simple and identical problems
  - When combined, the overall problem is solved
- Here, the simple problem is to clip a polygon against a single clip edge:

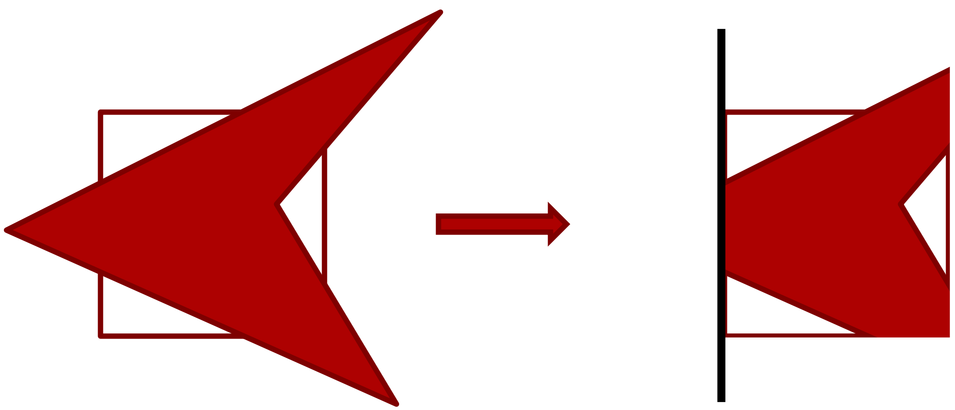
Clip against right edge



Sutherland Hodgeman Algorithm



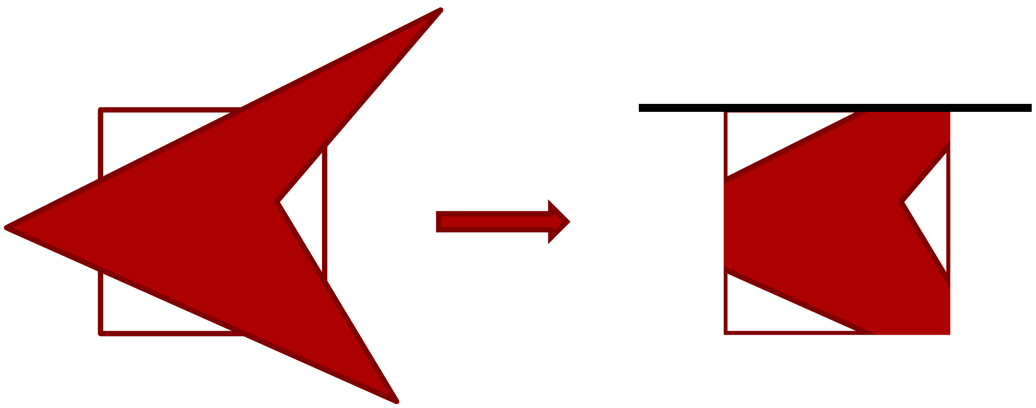
Clip against bottom edge



Clip against left edge

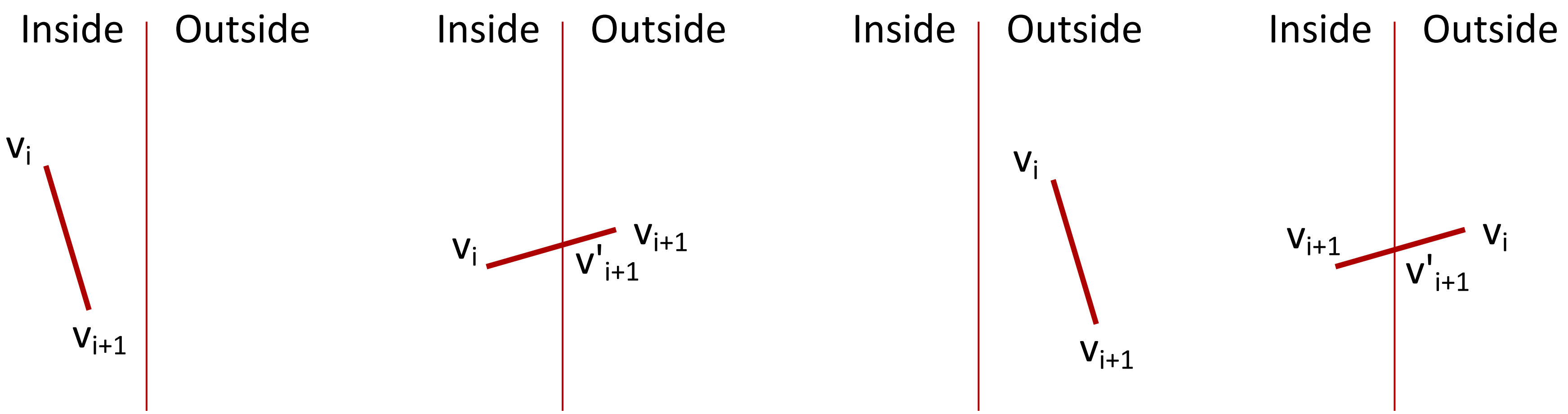
Sutherland Hodgeman Algorithm

Clip against top edge



Sutherland Hodgeman Algorithm

- This is accomplished by visiting the input vertices from v_0 to v_N and then back to v_0 for each clip boundary
- At every step we add 0, 1, or 2 vertices to the output:



Add v_{i+1} to output

Add v'_{i+1} to output

Add nothing to output

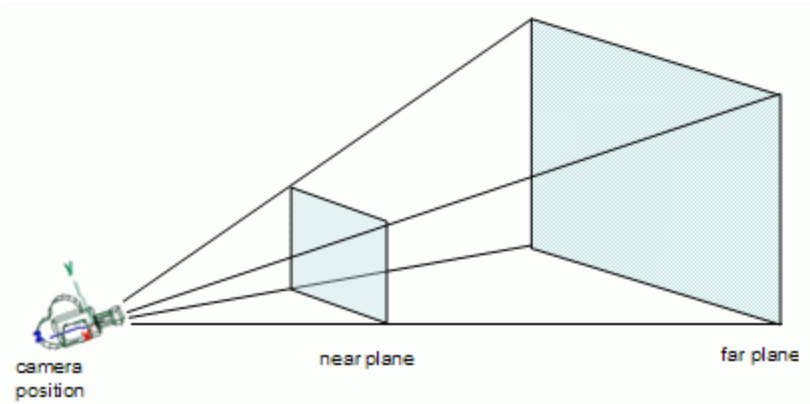
Add v'_{i+1} and v_{i+1} to output

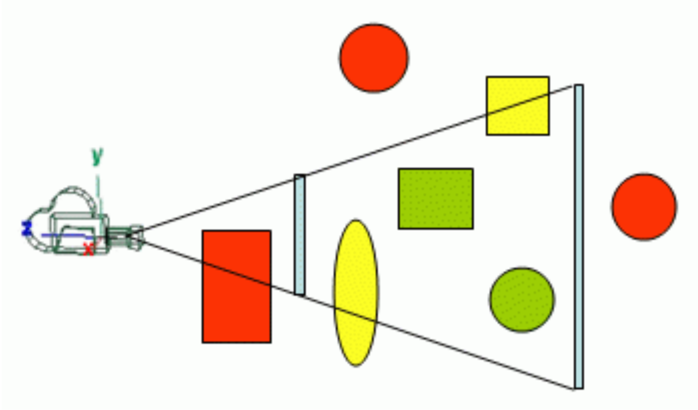
Culling

- Complex scenes contains many objects
- Objects closer to the camera occlude objects further away
- Rendering time can be saved if these invisible objects are culled (i.e. eliminated, discarded, thrown away)
- Three common culling strategies are:
  - View volume (frustum) culling
  - Backface culling
  - Occlusion culling

View Volume (Frustum) Culling

- The removal of geometry outside the viewing volume
- **No OpenGL support**: it is the programmer's responsibility to cull what is outside.





From lighthouse3d.com

View Volume (Frustum) Culling

- First determine the equations of the planes that make up the boundary of the view volume (6 planes):

Plane equation: $(p - a).\boldsymbol{n} = 0$

- Here, *a* is a point on the plane and ***n*** is the normal (pointing outside from the face)
- Plug the vertices of each primitive for *p.* If we get:

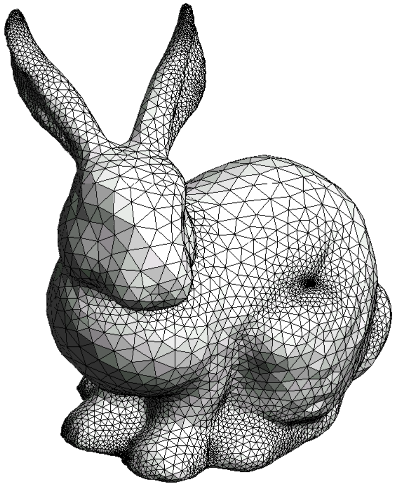
$$(p - a).\boldsymbol{n} > 0$$

for any plane, the vertex is outside

- If all vertices are outside w.r.t. the same plane, then the primitive is outside and can be culled
- Using a bounding box or bounding sphere for complex models is a better solution.

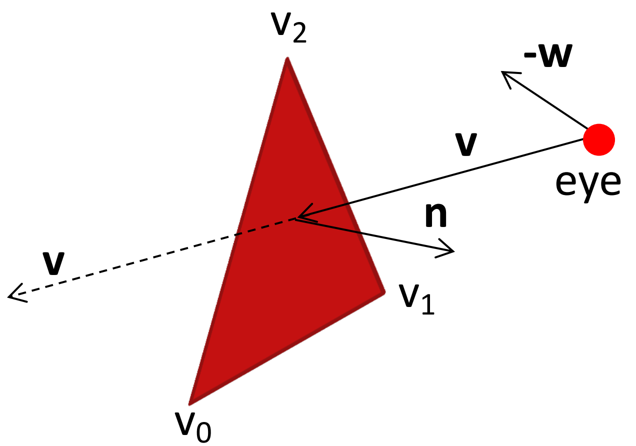
Backface Culling

- For **closed** polygon models, back facing polygons are guaranteed to be occluded by front facing polygons (so they don't need to be rendered)
- OpenGL supports backface culling: glCullFace(GL_BACK) and glEnable(GL_CULL_FACE)

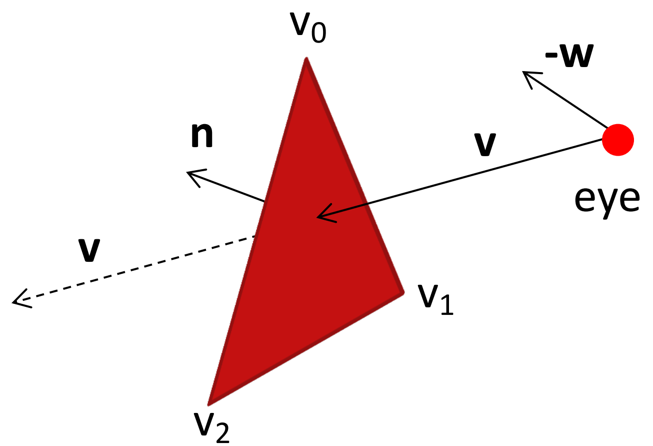


Backface Culling

- Polygons whose normals face away from the eye are called back facing polygons



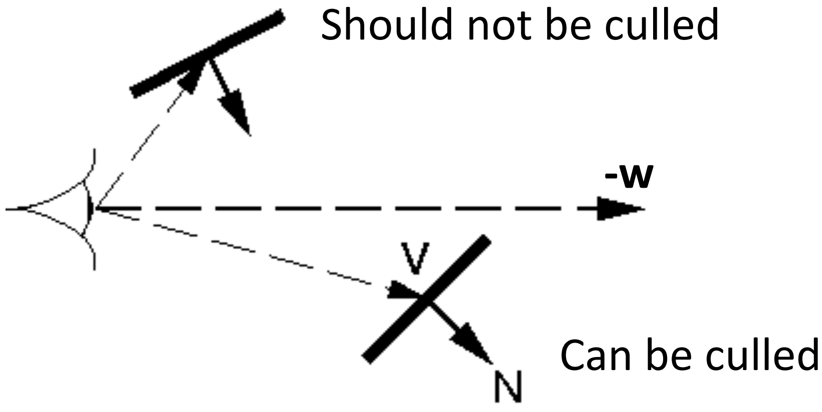
Front facing triangle
n.v < 0



Back facing triangle
n.v > 0

Backface Culling

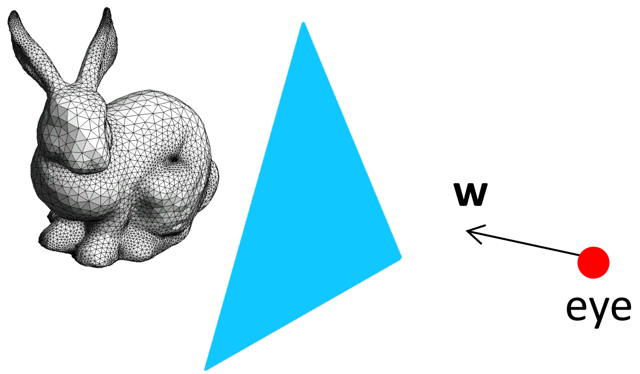
- Note the **v** is the vector from the eye to any point on the polygon (you can take the polygon center). **You cannot use the view vector**!



From http://omega.di.unipi.it

Occlusion Culling

- The removal of geometry that is within the view volume but is occluded by other geometry closer to the camera:



Red triangle is occluded by the blue triangle

- OpenGL supports occlusion queries to assist the user in occlusion culling
- By a fast rendering pass, it counts how many pixels of the tested object will be rendered
- This is commonly used in games