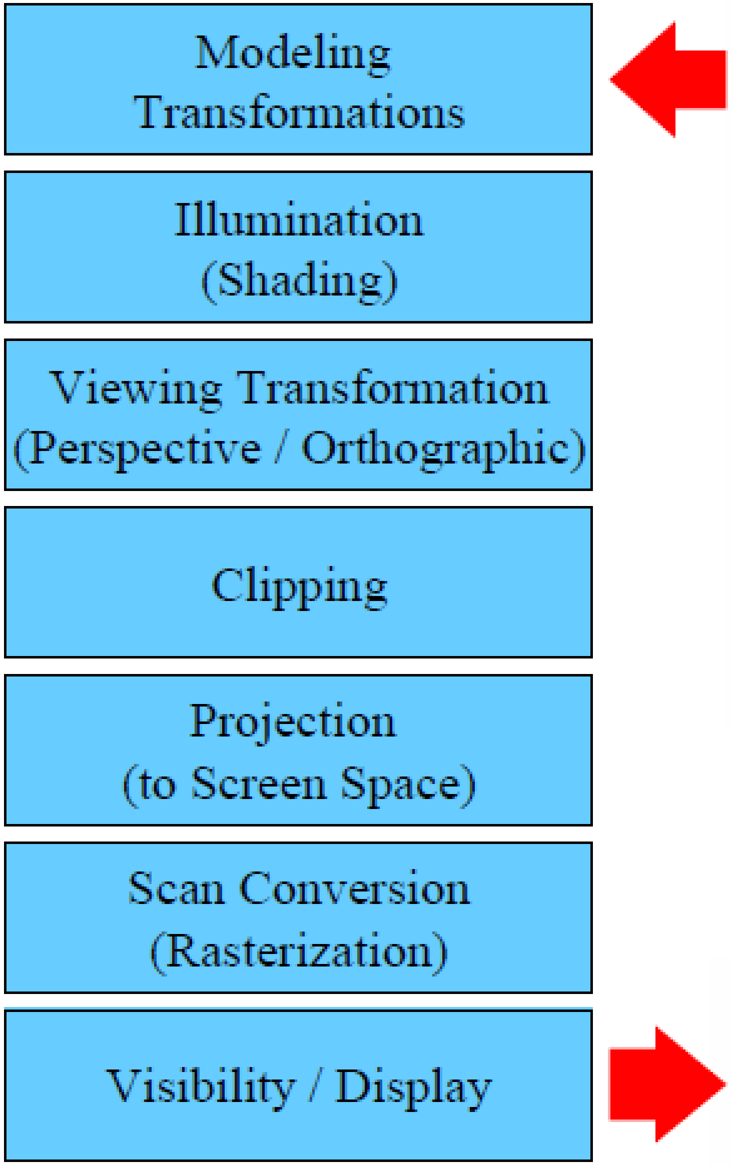
The Graphics Pipeline



Input:

Geometric model:
Description of all object, surface, and light source geometry and transformations

Lighting model:
Computational description of object and light properties, interaction (reflection)

Synthetic Viewpoint (or *Camera*):
Eye position and viewing frustum

Raster Viewport:
Pixel grid onto which image plane is mapped

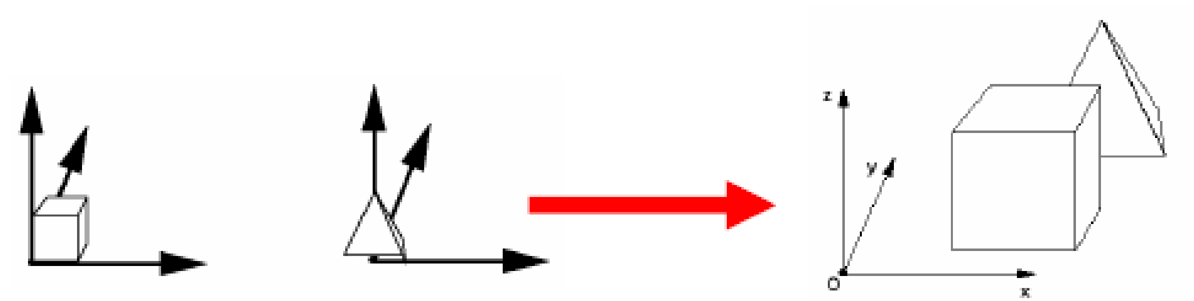
Output:

Colors/Intensities suitable for framebuffer display
(For example, 24-bit RGB value at each pixel)

Modeling Transformations

- **Modeling Transformations**
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- 3D models defined in their own coordinate system (object space)
- Modeling transforms orient the models within a common coordinate frame (world space)

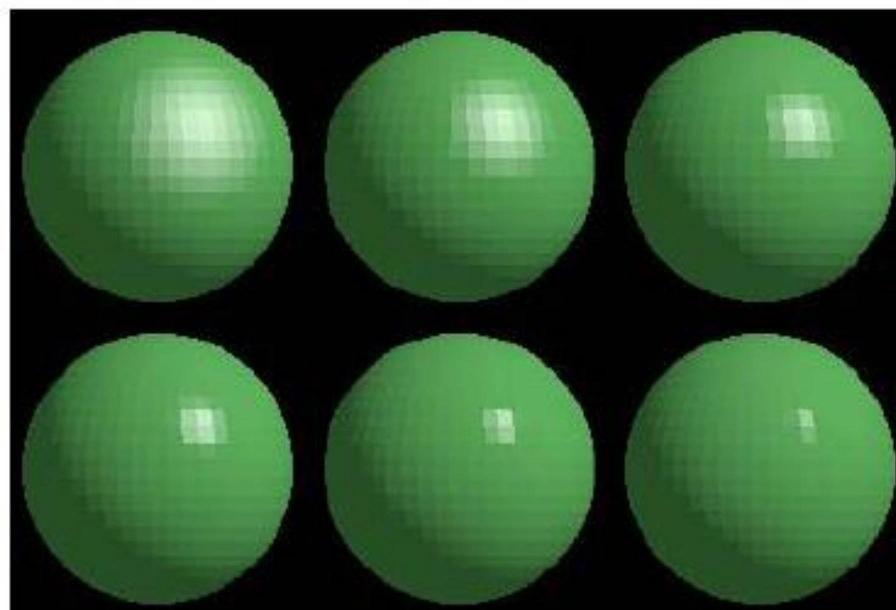


Object space World space

Illumination (Shading) (Lighting)

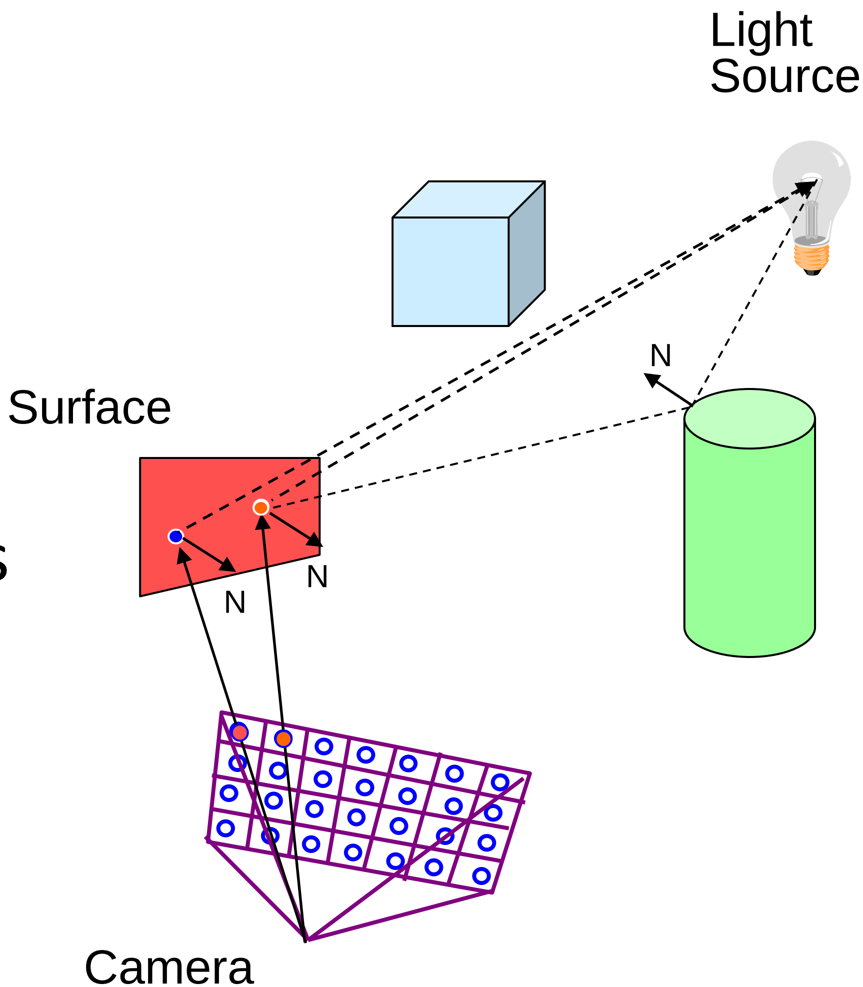
- Modeling Transformations
- **Illumination (Shading)**
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Vertices lit (shaded) according to material properties, surface properties (normal) and light sources
- Local lighting model (Diffuse, Ambient, Phong, etc.)



Lighting Simulation

- Direct illumination
  - Ray casting
  - Polygon shading
- Global illumination
  - Ray tracing
  - Monte Carlo methods
  - Radiosity methods



Viewing Transformation

- Modeling Transformations
- Illumination (Shading)
- **Viewing Transformation (Perspective / Orthographic)**
- Clipping
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

- Maps world space to eye space
- Viewing positions is transformed to original & direction is oriented along some axis (usually *z*)

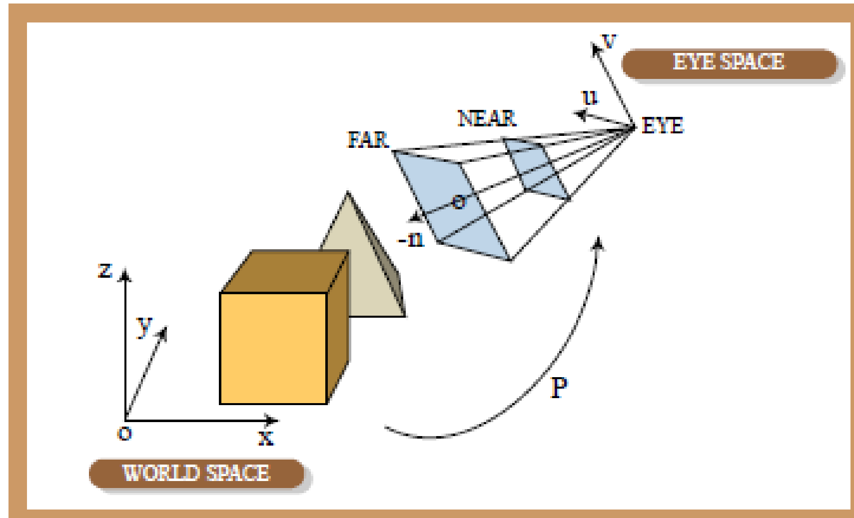
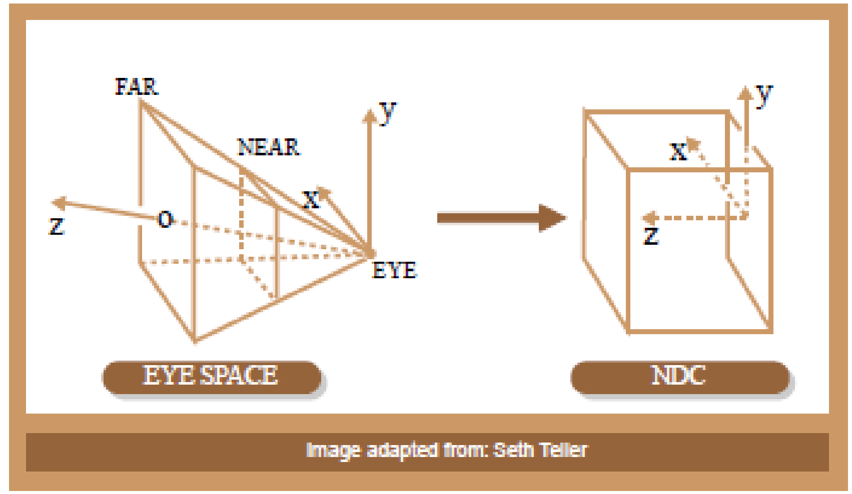


Image adapted from: Seth Teller

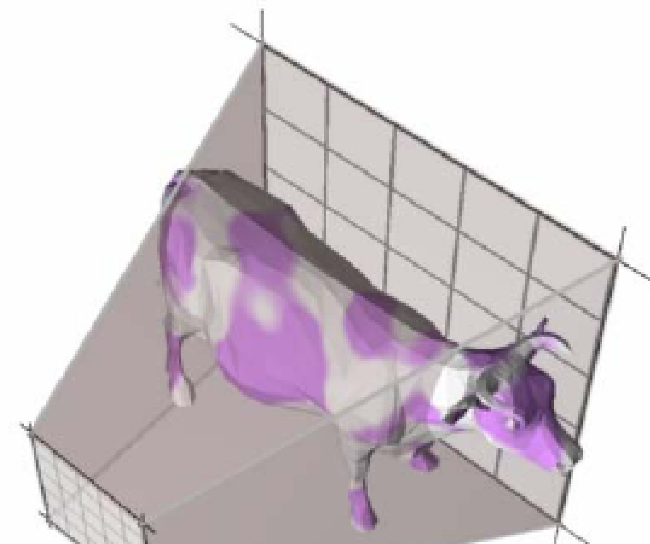
Clipping

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- **Clipping**
- Projection (to Screen Space)
- Scan Conversion (Rasterization)
- Visibility / Display

• Transform to Normalized Device Coordinates (NDC)



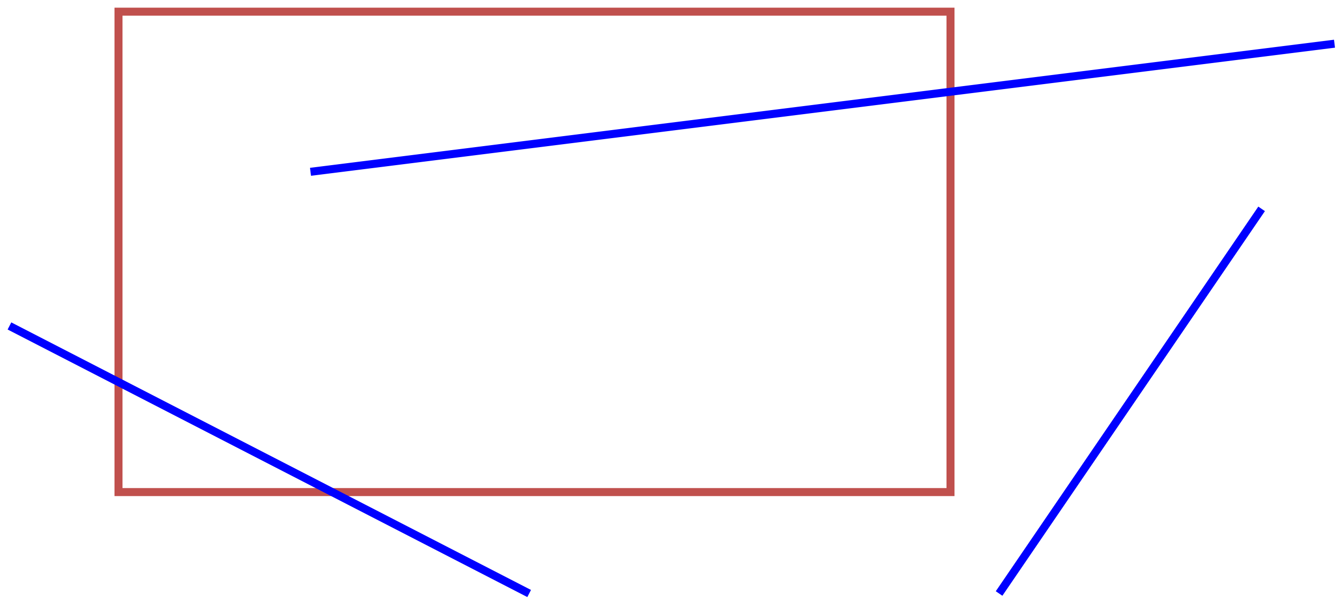
• Portions of the object outside the view volume (view frustum) are removed



Courtesy of Leonard McMillan, Computer Science at the University of North Carolina in Chapel Hill. Used with permission.

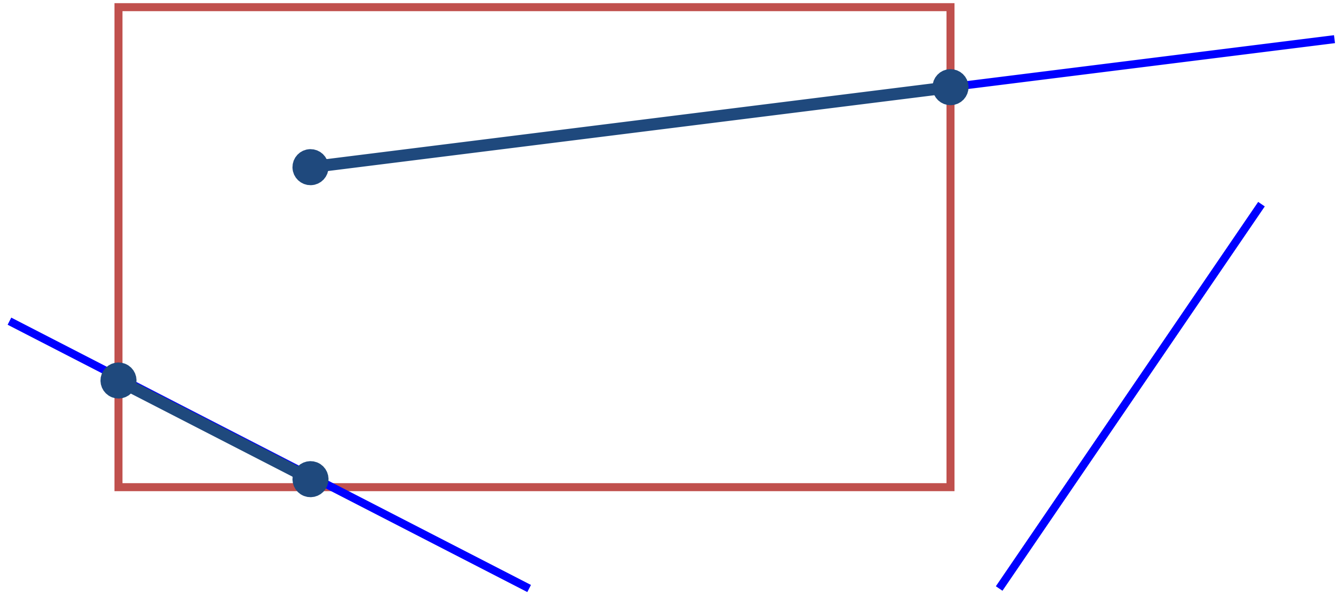
Why clip?

- We don’t want to waste time rendering objects that are outside the viewing window (or clipping window)

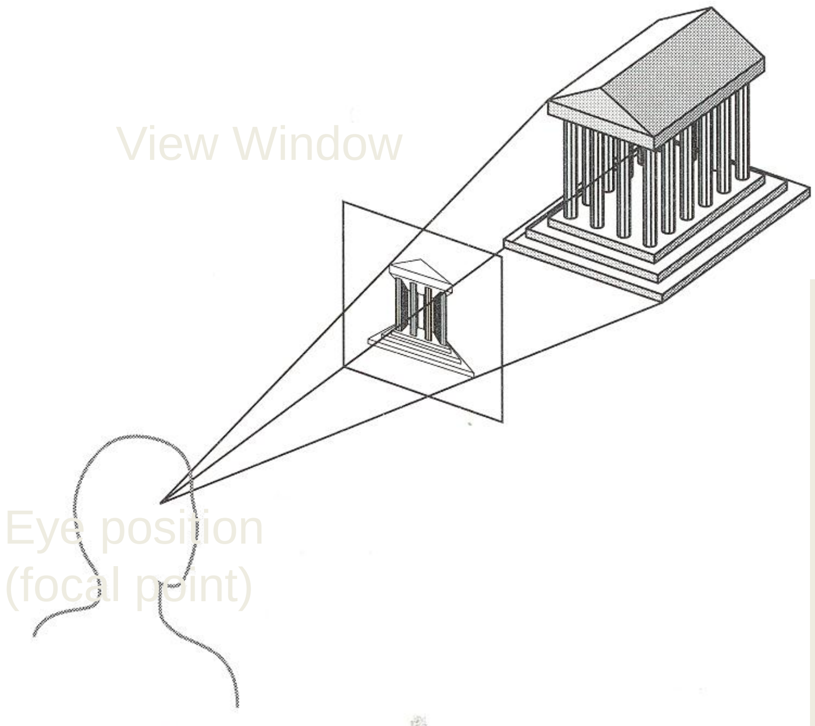


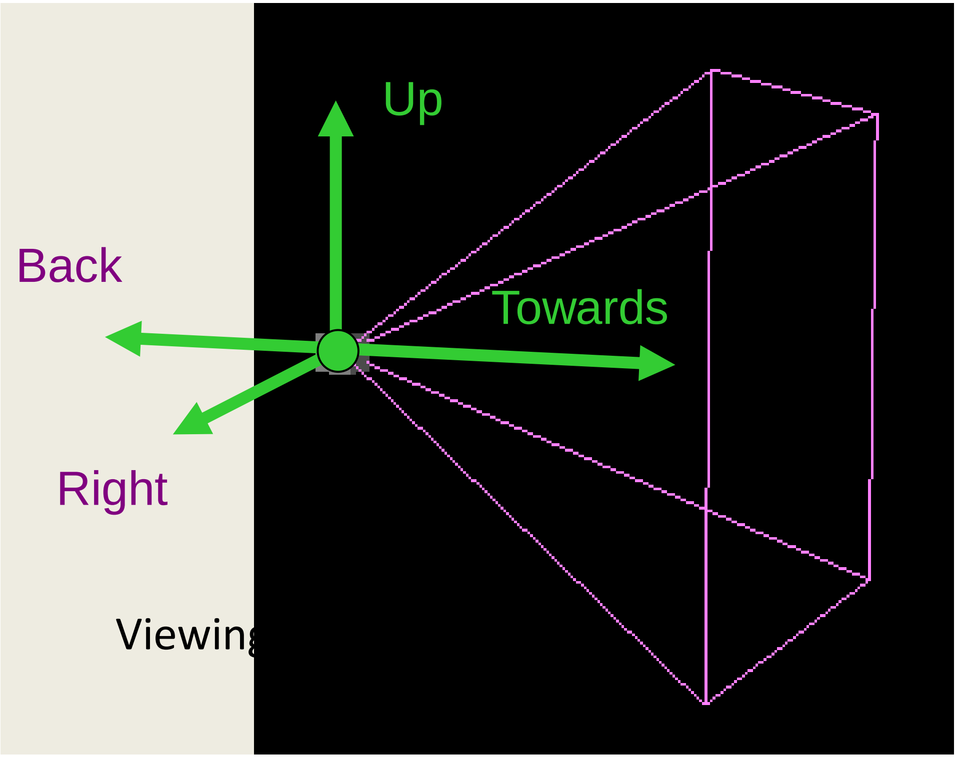
What is clipping?

- Analytically calculating the portions of primitives within the view window

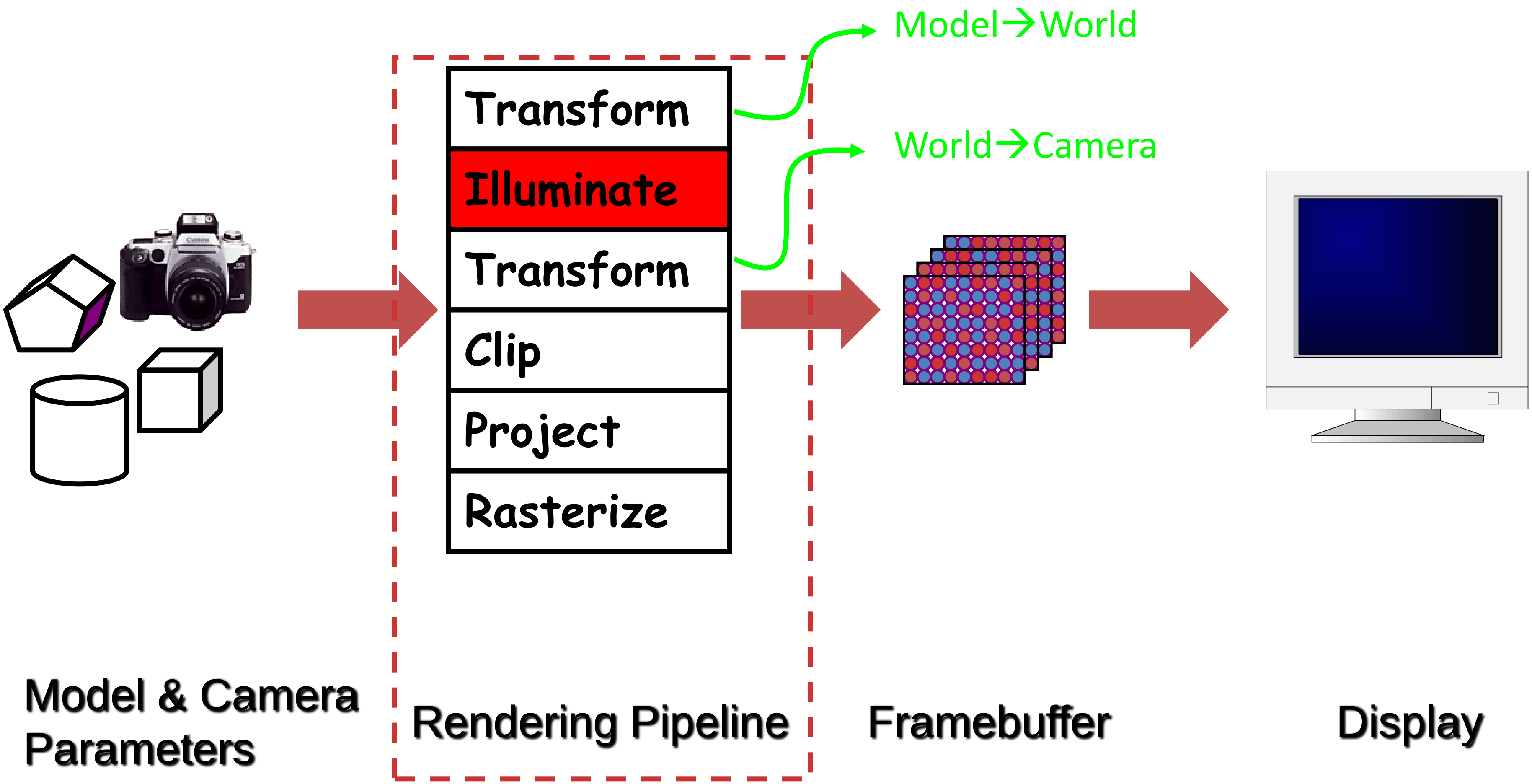


Clip to what?

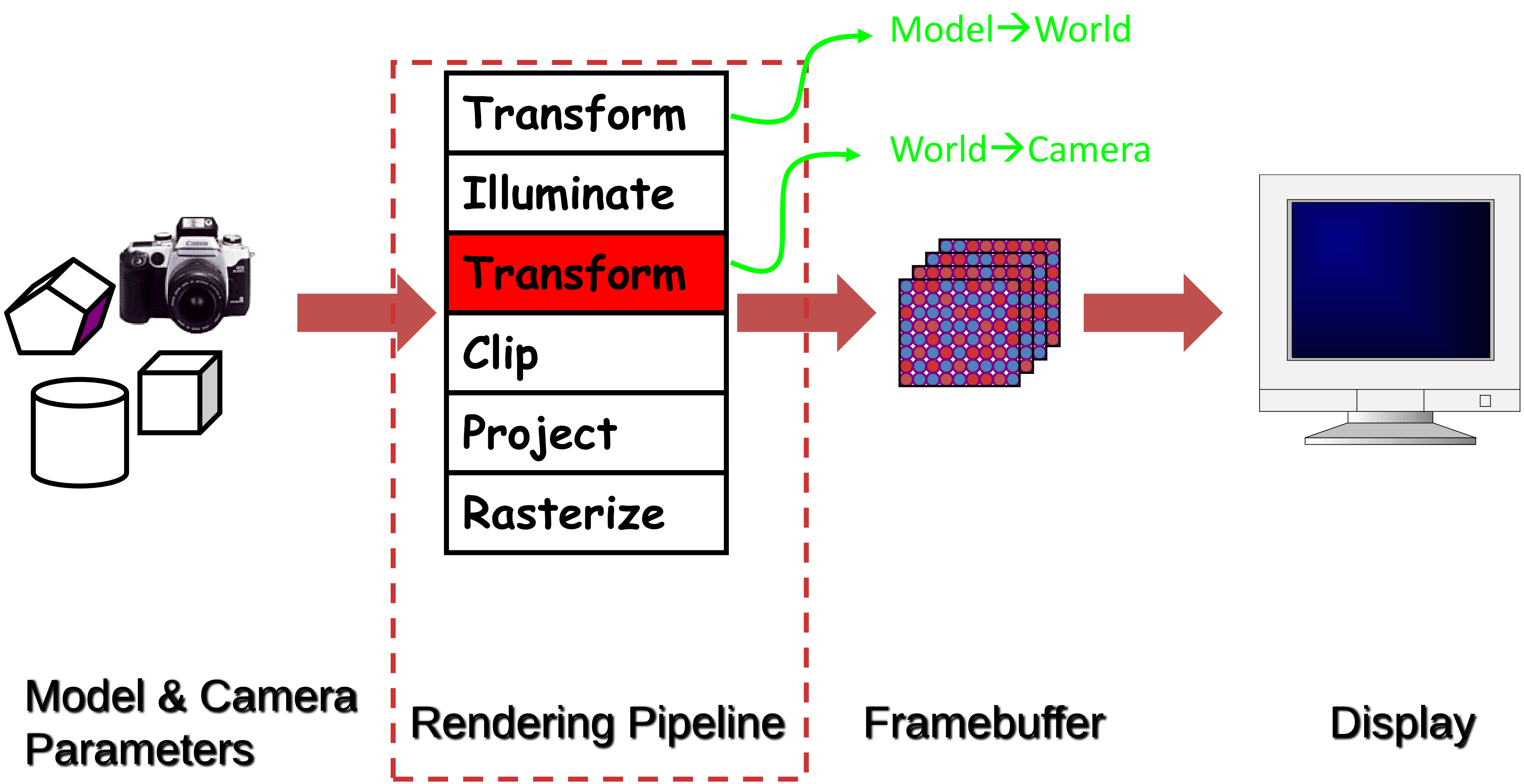

View Window
Eye position
(focal point)


Up
Back
Towards
Right
Viewing

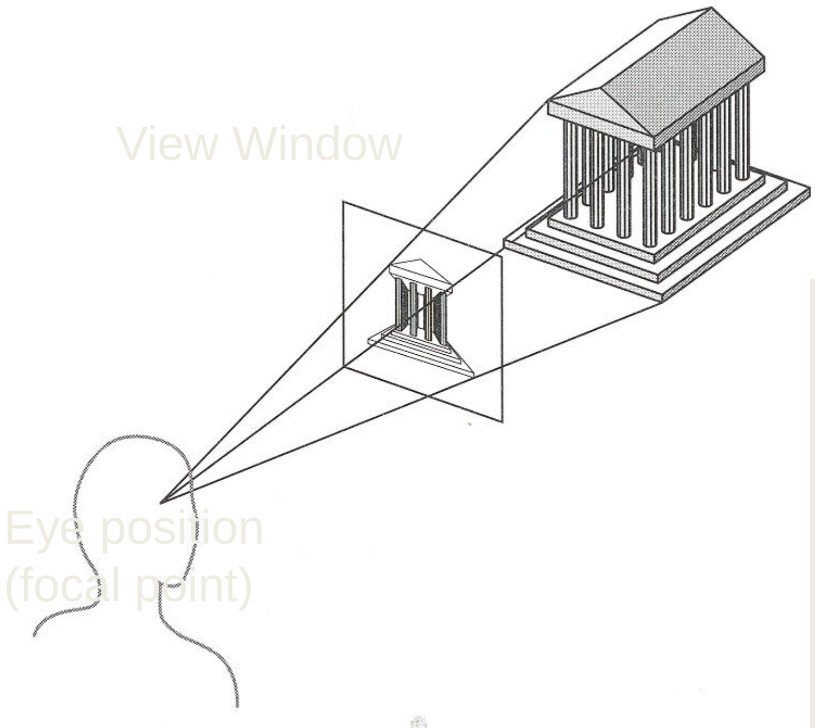
Why illuminate before clipping?

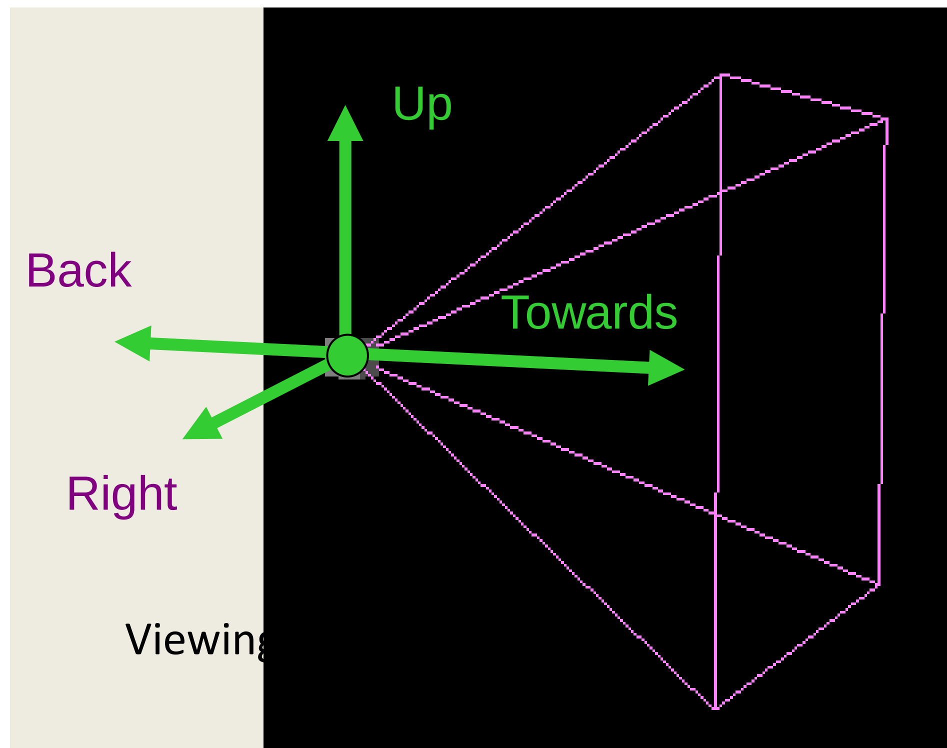


Why World→Camera before clipping?

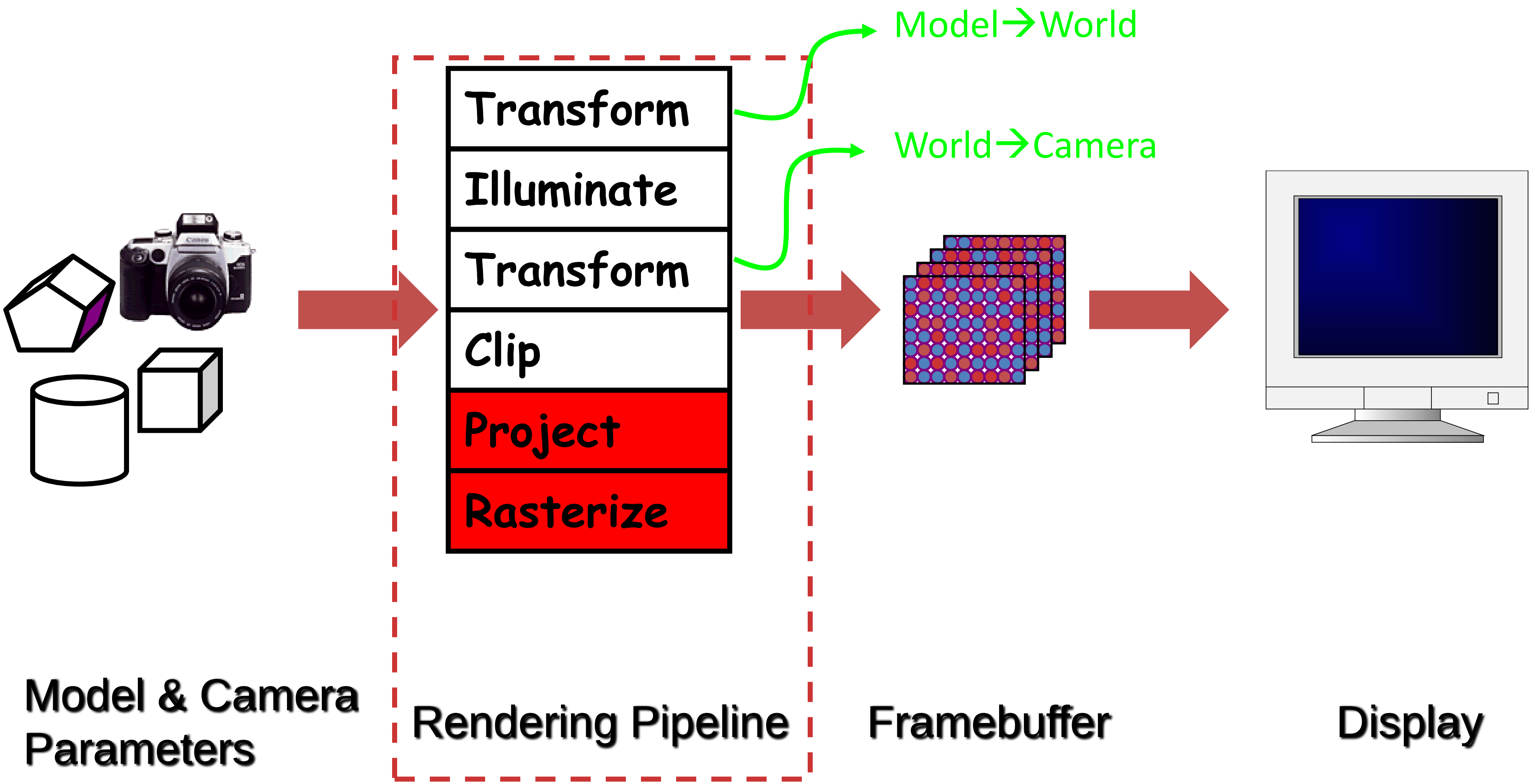


Clip to what?





Remind me why I care again



Why Clip?

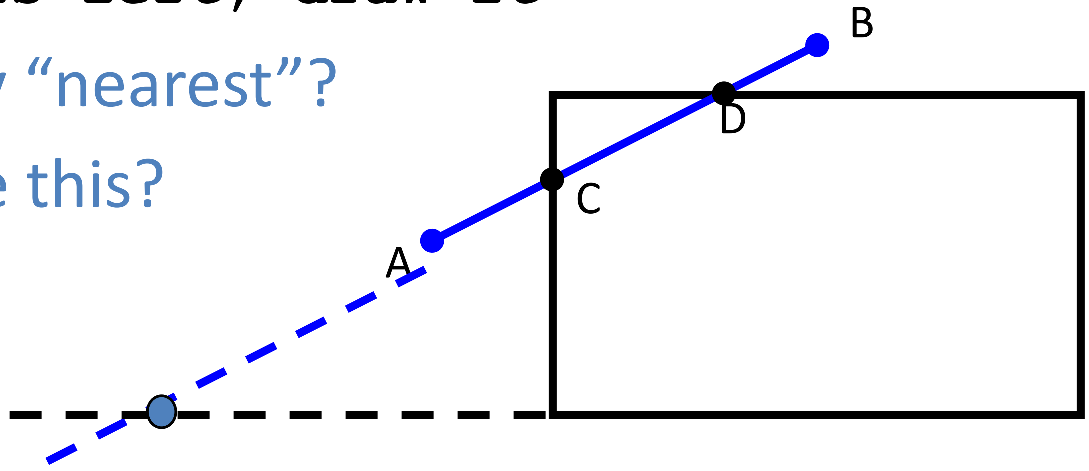
- Bad idea to rasterize outside of framebuffer bounds
- Also, don't waste time scan converting pixels outside window

Clipping

- The naïve approach to clipping lines:

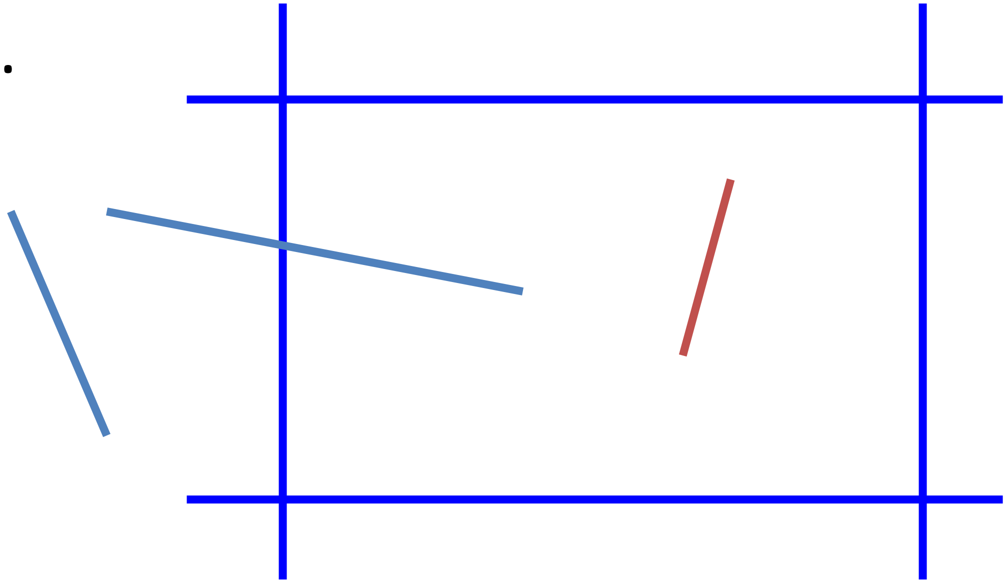
```
for each line segment
    for each edge of view_window
        find intersection point
        pick "nearest" point
    if anything is left, draw it
```

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



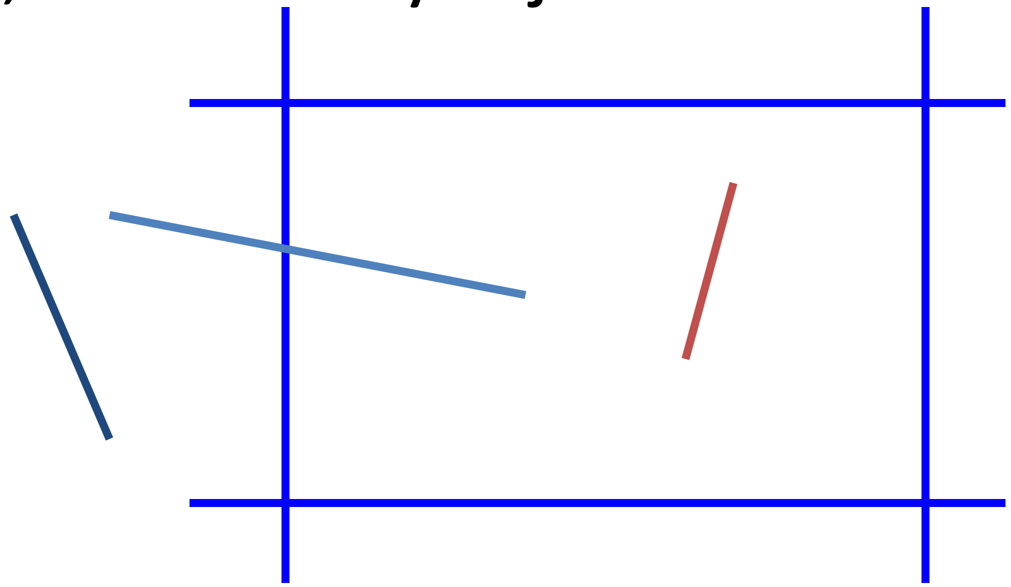
Trivial Accepts

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



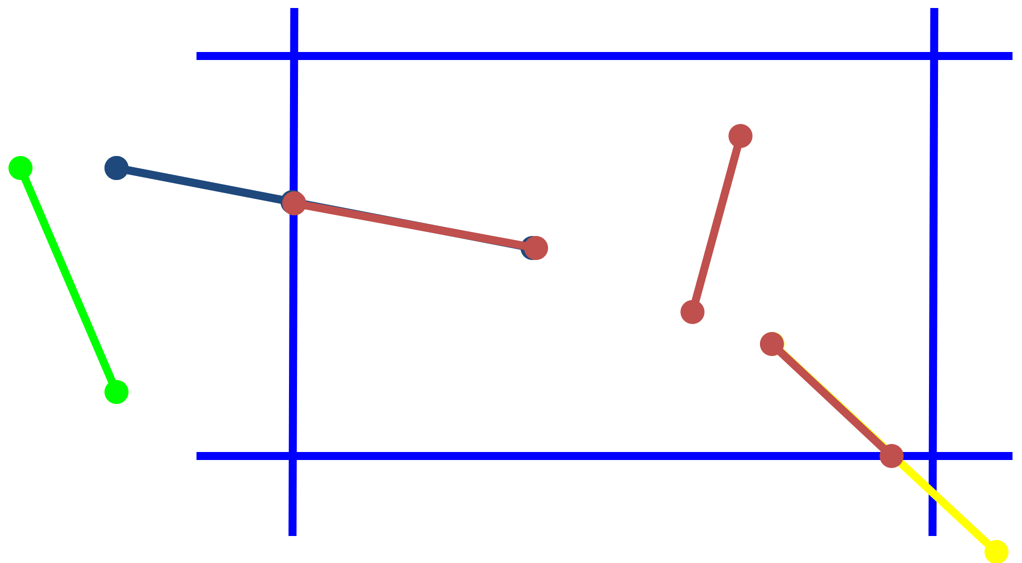
Trivial Rejects

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



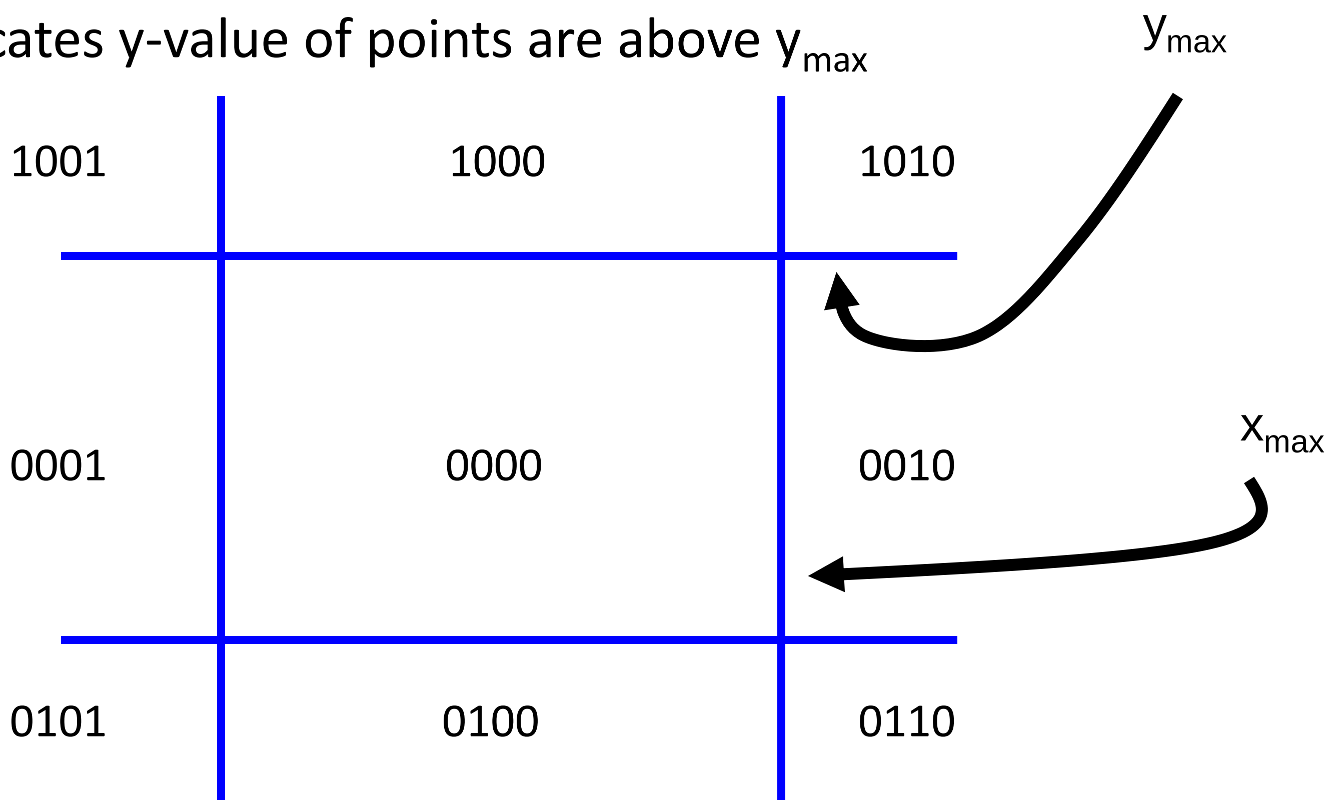
Clipping Lines To Viewport

- Combining trivial accepts/rejects
  - Trivially accept lines with both endpoints inside all edges of the view window
  - Trivially reject lines with both endpoints outside the same edge of the view window
  - Otherwise, reduce to trivial cases by splitting into two segments



Cohen-Sutherland Line Clipping

– Divide view window into regions defined by window edges
– Assign each region a 4-bit *outcode*:
  - Bit 1 indicates y-value of points are above y_{max}



Cohen-Sutherland Line Clipping

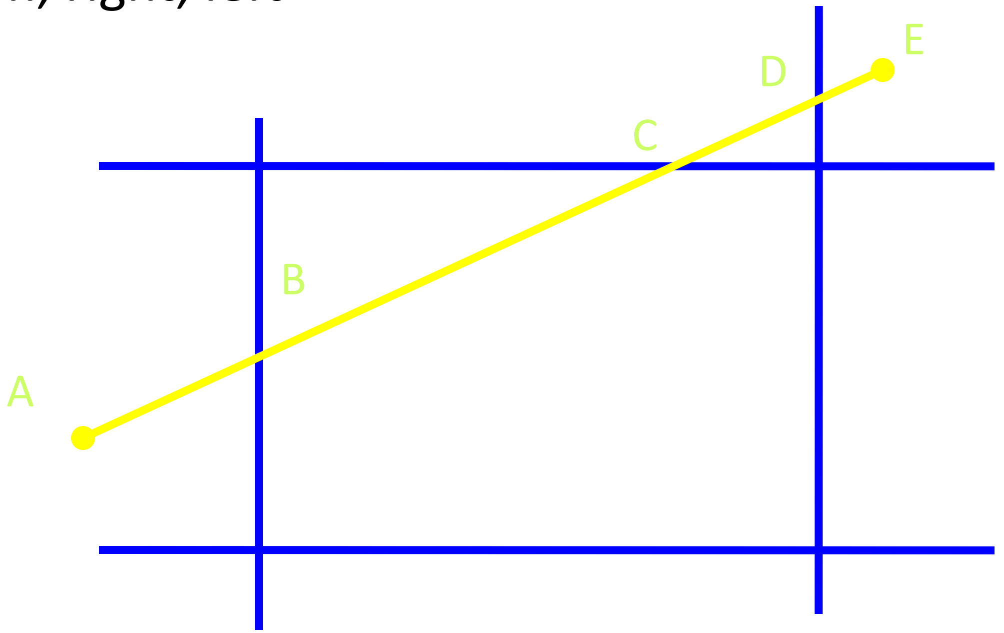
- For each line segment
  - Assign an outcode to each vertex
  - If both outcodes = 0, trivial accept
    - Same as performing `if (bitwise OR = 0)`
  - Else
    - bitwise *AND* vertex outcodes together
    - if result $\neq 0$, trivial reject

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 of view window that the line crosses (*how?*)
  - Intersect line with edge (*how?*)
  - Discard portion on wrong side of edge and assign new outcode to new vertex
  - Apply trivial accept/reject tests; repeat if necessary

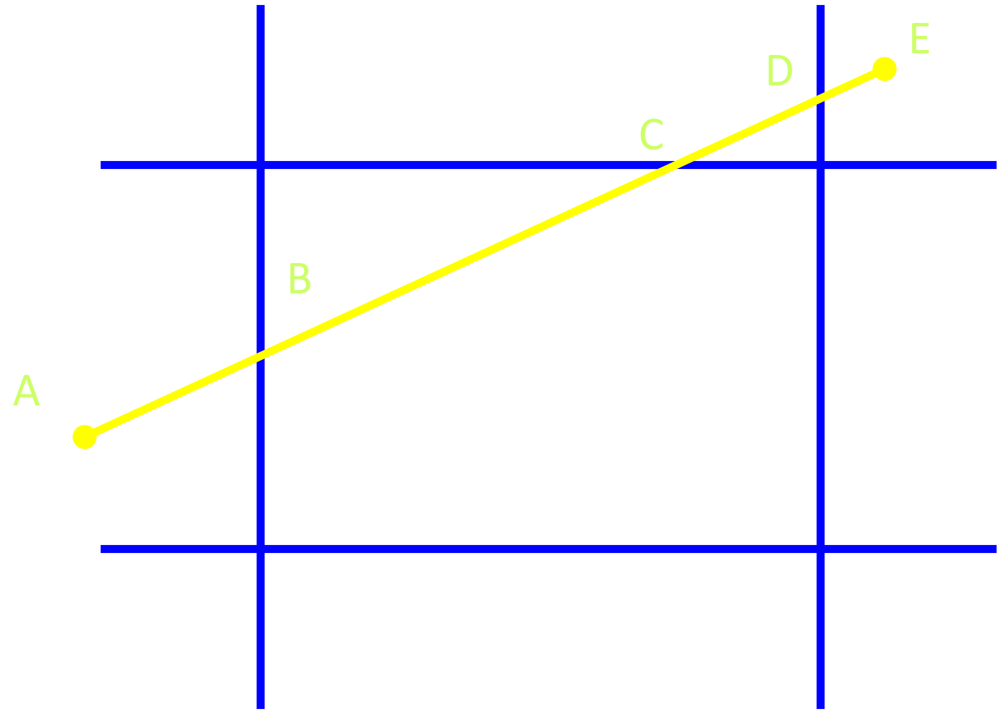
Cohen-Sutherland Line Clipping

- 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
  - Check against edges in same order each time
    - For example: top, bottom, right, left



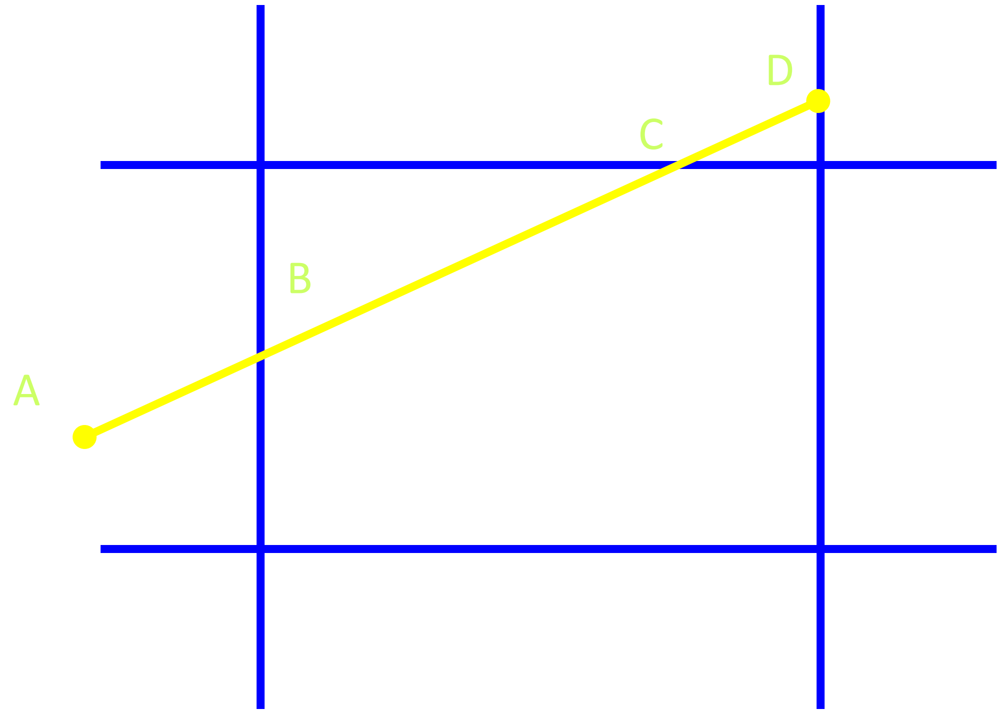
Cohen-Sutherland Line Clipping

- Intersect line with edge (how?)



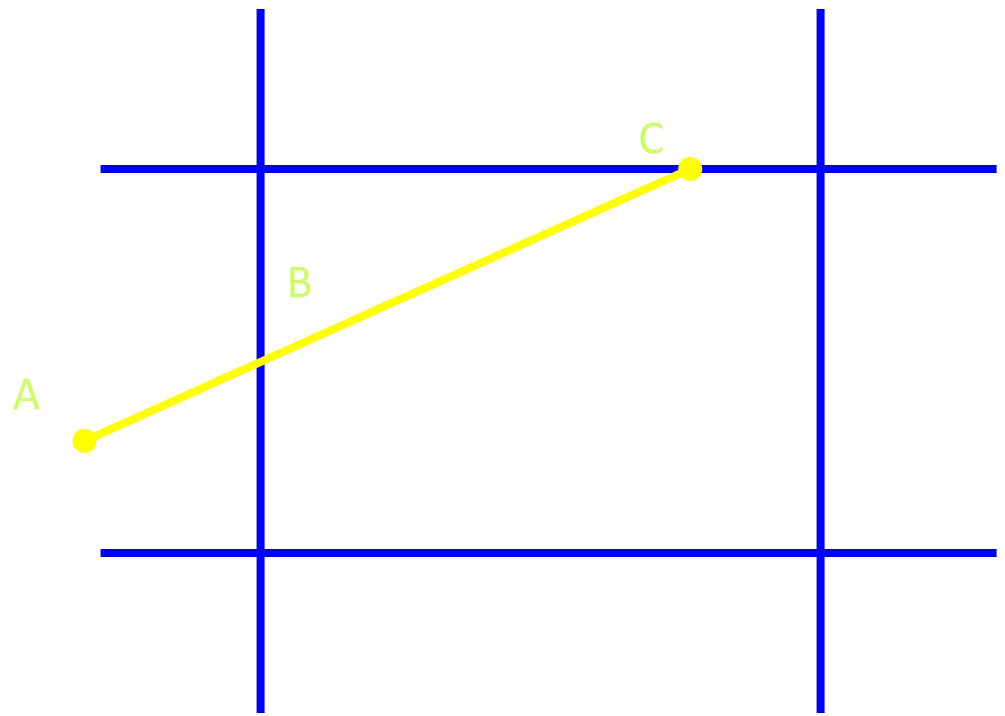
Cohen-Sutherland Line Clipping

- Discard portion on wrong side of edge and assign outcode to new vertex
- Apply trivial accept/reject tests and repeat if necessary



Cohen-Sutherland Line Clipping

- Discard portion on wrong side of edge and assign outcode to new vertex
- Apply trivial accept/reject tests and repeat if necessary



View Window Intersection Code

- $(x_1, y_1), (x_2, y_2)$ intersect with vertical edge at x_{right}
  - $y_{intersect} = y_1 + m(x_{right} - x1)$
    - where $m=(y_2-y_1)/(x_2-x_1)$

- $(x_1, y_1), (x_2, y_2)$ intersect with horizontal edge at y_{bottom}
  - $x_{intersect} = x_1 + (y_{bottom} - y1)/m$
    - where $m=(y_2-y_1)/(x_2-x_1)$

Cohen-Sutherland Review

- Use opcodes to quickly eliminate/include lines
  - Best algorithm when trivial accepts/rejects are common
- Must compute viewing window clipping of remaining lines
  - Non-trivial clipping cost
  - Redundant clipping of some lines
- More efficient algorithms exist

Solving Simultaneous Equations

- Equation of a line
  - Slope-intercept (explicit equation): $y = mx + b$
  - Implicit Equation: $Ax + By + C = 0$
  - Parametric Equation: Line defined by two points, P_0 and P_1
    - $\mathbf{P}(t) = \mathbf{P}_0 + (\mathbf{P}_1 - \mathbf{P}_0)\, t$, where $\mathbf{P}$ is a vector $[x, y]^T$

    - $x(t) = x_0 + (x_1 - x_0)\, t$
    - $y(t) = y_0 + (y_1 - y_0)\, t$

Parametric Line Equation

- Describes a finite line
- Works with vertical lines (like the viewport edge)
  - $0 <= t <= 1$
    - Defines line between P_0 and P_1
  - $t < 0$
    - Defines line before P_0
  - $t > 1$
    - Defines line after P_1

Parametric Lines and Clipping

- Define each line in parametric form:
  - $P_0(t) \ldots P_{n-1}(t)$
- Define each edge of view window in parametric form:
  - $P_L(t)$, $P_R(t)$, $P_T(t)$, $P_B(t)$
- Perform Cohen-Sutherland intersection tests using appropriate view window edge and line

Line / Edge Clipping Equations

- Faster line clippers use parametric equations
- Line 0:
  - $x^0 = x^0_0 + (x^0_1 - x^0_0)\ t^0$
  - $y^0 = y^0_0 + (y^0_1 - y^0_0)\ t^0$

View Window Edge L:

- $x^L = x^L_0 + (x^L_1 - x^L_0)\ t^L$
- $y^L = y^L_0 + (y^L_1 - y^L_0)\ t^L$

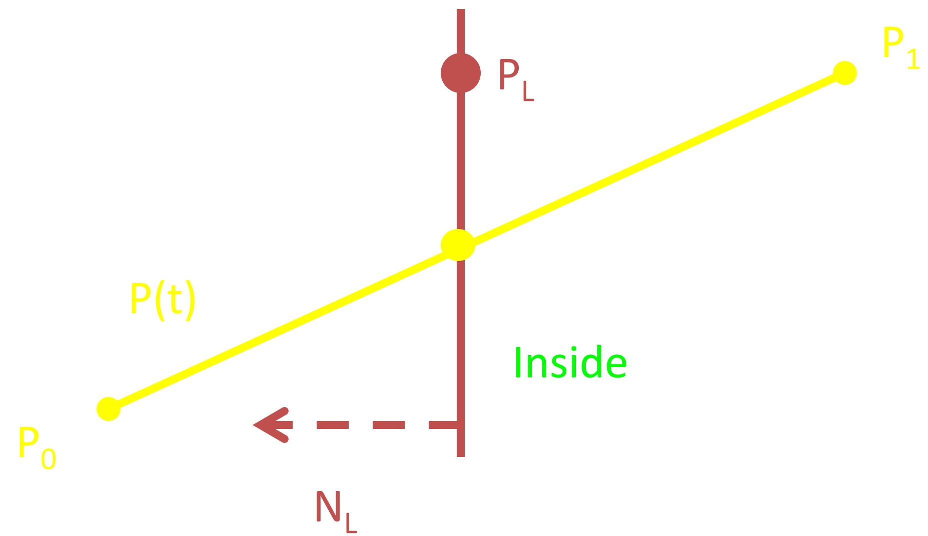
- $x^0_0 + (x^0_1 - x^0_0)\ t^0 = x^L_0 + (x^L_1 - x^L_0)\ t^L$
- $y^0_0 + (y^0_1 - y^0_0)\ t^0 = y^L_0 + (y^L_1 - y^L_0)\ t^L$
  - Solve for t^0 and/or t^L

Cyrus-Beck Algorithm

- We wish to optimize line/line intersection
  - Start with parametric equation of line:
    - $P(t) = P_0 + (P_1 - P_0)\ t$
  - And a point and normal for each edge
    - P_L, N_L

Cyrus-Beck Algorithm

- Find t such that
- $N_L \; [P(t) - P_L] = 0$



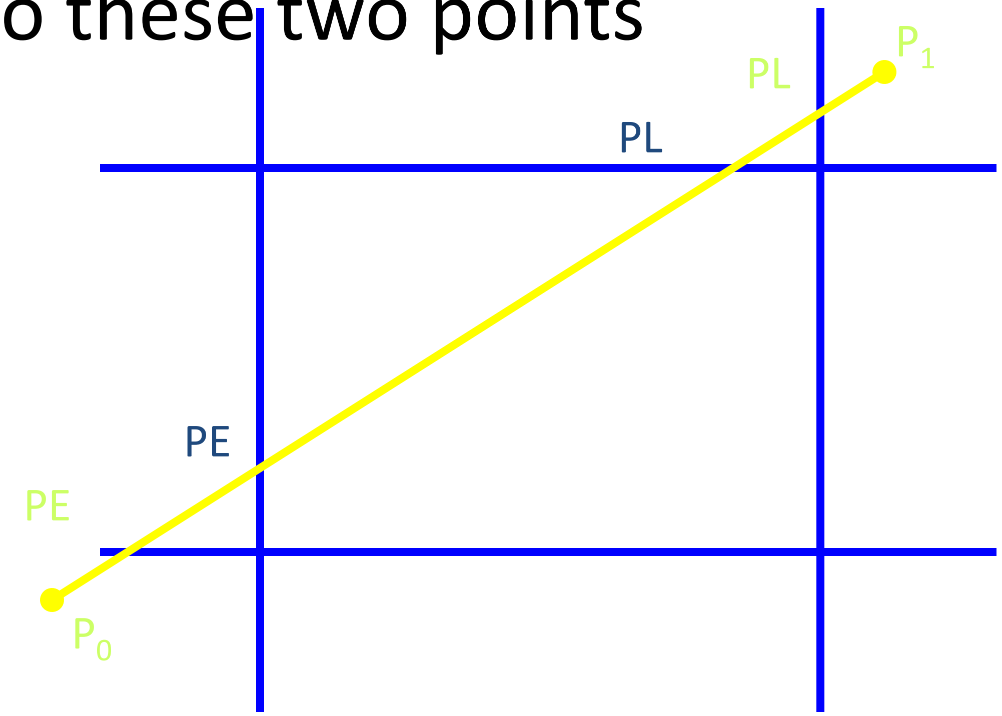
- Substitute line equation for P(t):
  - $N_L \; [P_0 + (P_1 - P_0)\, t - P_L] = 0$
- Solve for t
  - $t = N_L \; [P_L – P_0] \; / \; -N_L \; [P_1 - P_0]$

Cyrus-Beck Algorithm

- Compute t for line intersection with all four edges
- Discard all $(t < 0)$ and $(t > 1)$
- Classify each remaining intersection as
  - Potentially Entering (PE)
  - Potentially Leaving (PL)
- $N_L \; [P_1 - P_0] > 0$ implies PL
- $N_L \; [P_1 - P_0] < 0$ implies PE
  - Note that we computed this term when computing t so we can keep it around

Cyrus-Beck Algorithm

- Compute PE with largest t
- Compute PL with smallest t
- Clip to these two points



Cyrus-Beck Algorithm

- Because of horizontal and vertical clip lines:
  - Many computations reduce
- Normals: (-1, 0), (1, 0), (0, -1), (0, 1)
- Pick constant points on edges
- solution for t:
  - $-(x_0 - x_{left}) / (x_1 - x_0)$
  - $(x_0 - x_{right}) / -(x_1 - x_0)$
  - $-(y_0 - y_{bottom}) / (y_1 - y_0)$
  - $(y_0 - y_{top}) / -(y_1 - y_0)$

Comparison

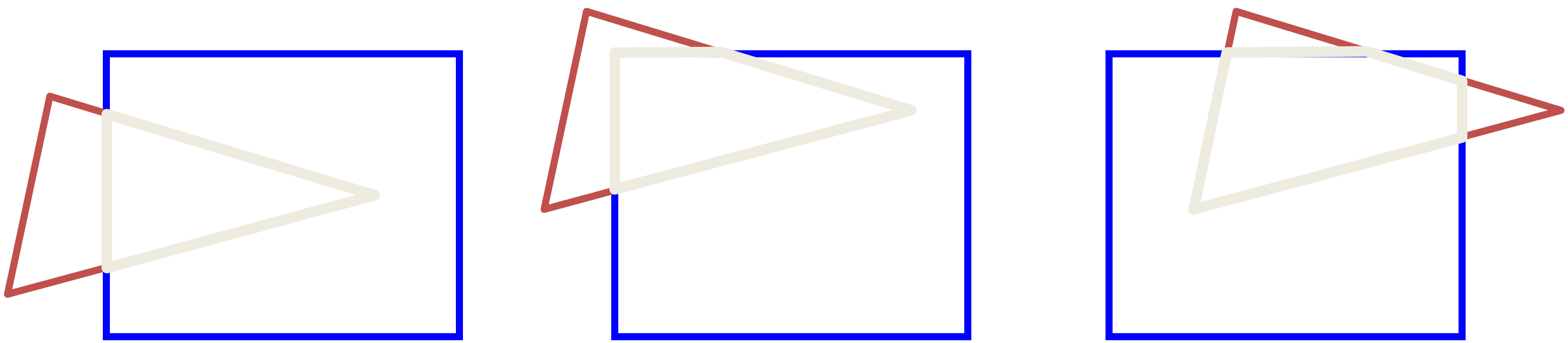
- Cohen-Sutherland
  - Repeated clipping is expensive
  - Best used when trivial acceptance and rejection is possible for most lines
- Cyrus-Beck
  - Computation of t-intersections is cheap
  - Computation of (x,y) clip points is only done once
  - Algorithm doesn’t consider trivial accepts/rejects
  - Best when many lines must be clipped
- Liang-Barsky: Optimized Cyrus-Beck
- Nicholl et al.: Fastest, but doesn’t do 3D

Clipping Polygons

- Clipping polygons is more complex than clipping the individual lines
  - Input: polygon
  - Output: original polygon, new polygon, or nothing
- The biggest optimizer we had was trivial accept or reject…
- 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:

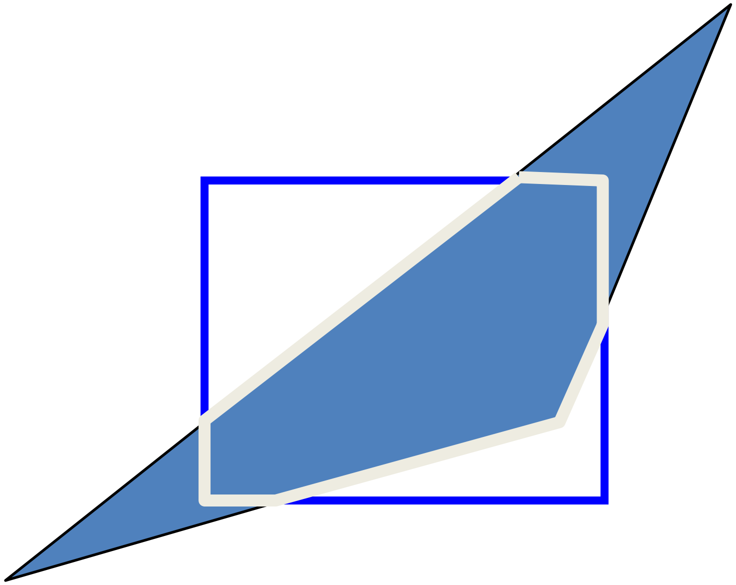


triangle ⇨ triangle triangle ⇨ quad triangle ⇨ 5-gon

How many sides can a clipped triangle have?

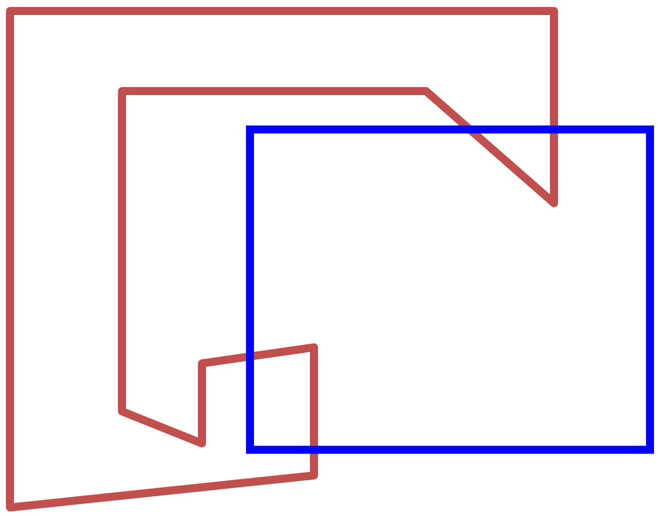
How many sides?

- Seven…



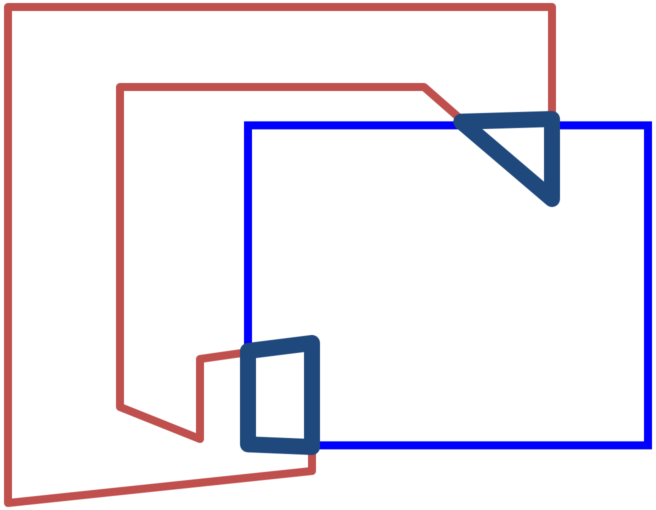
Why Is Clipping Hard?

- A really tough case:



Why Is Clipping Hard?

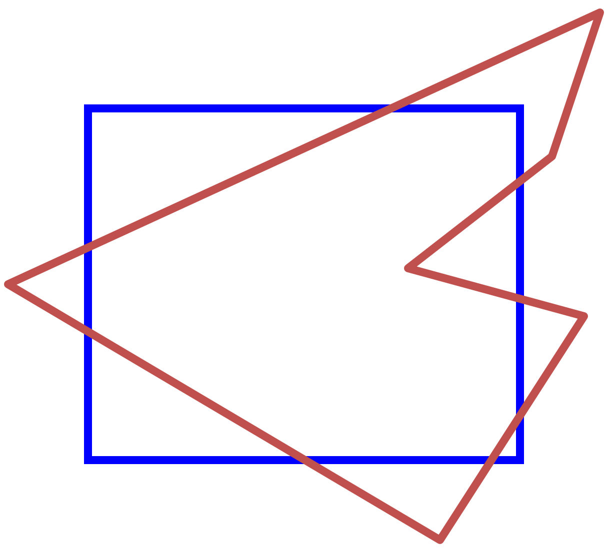
- A really tough case:



concave polygon ⇨ multiple polygons

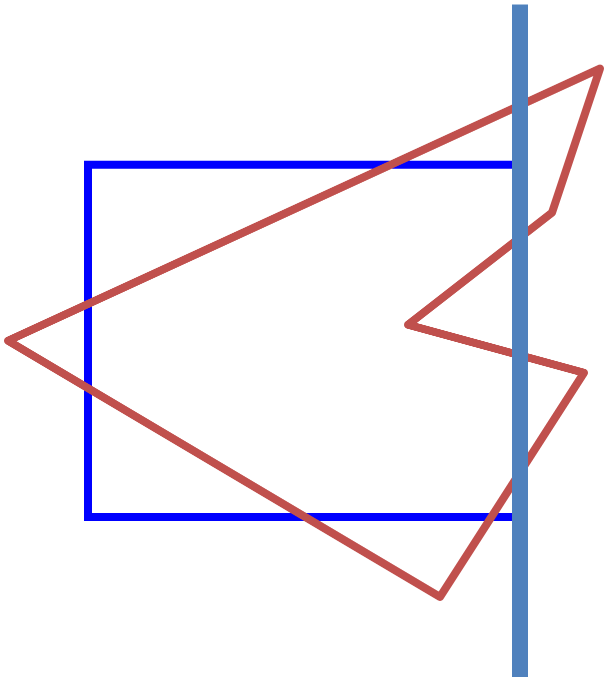
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the view window individually
  - Clip the polygon against the view window edge's equation



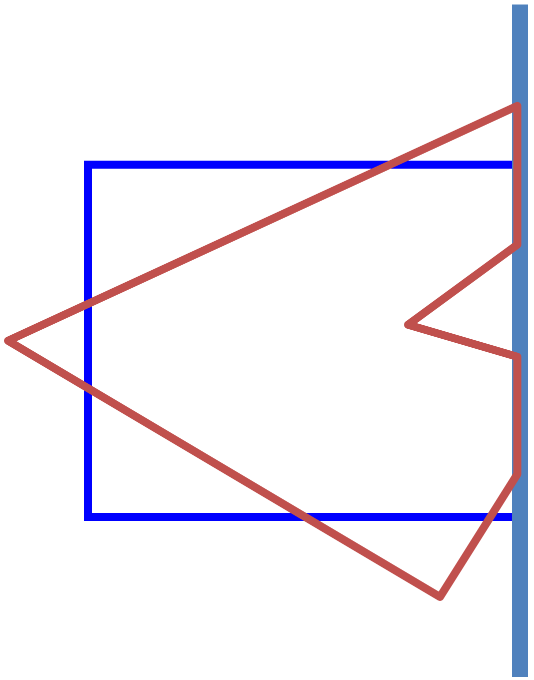
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation



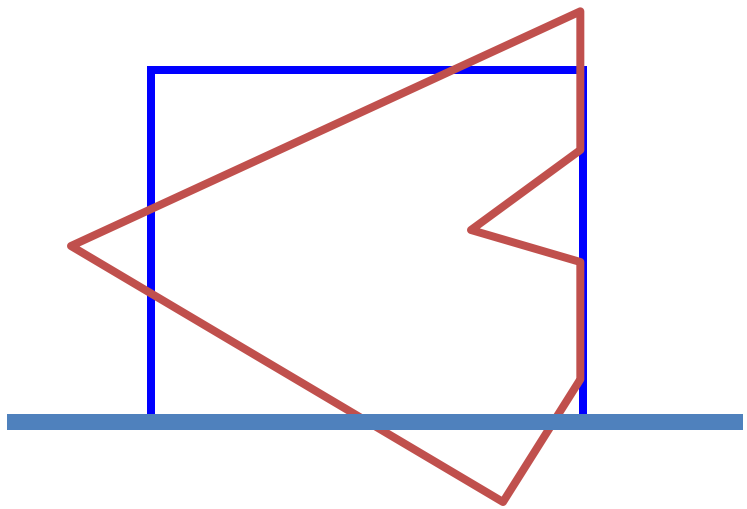
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation



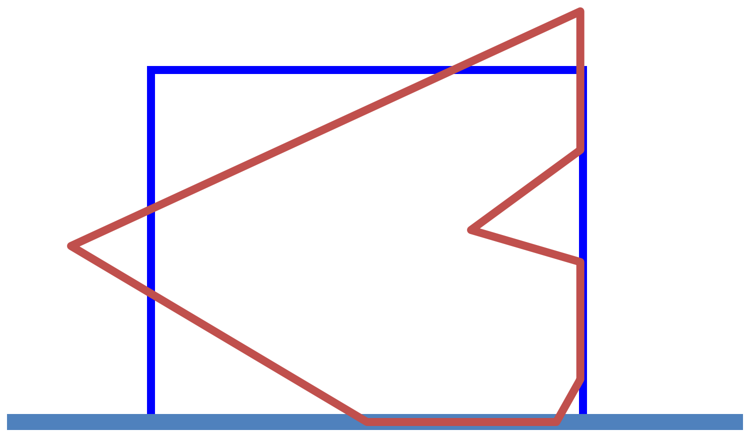
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation



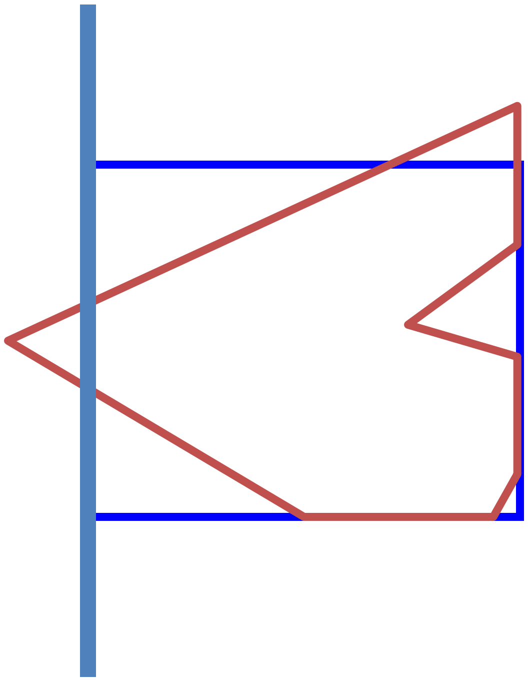
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation



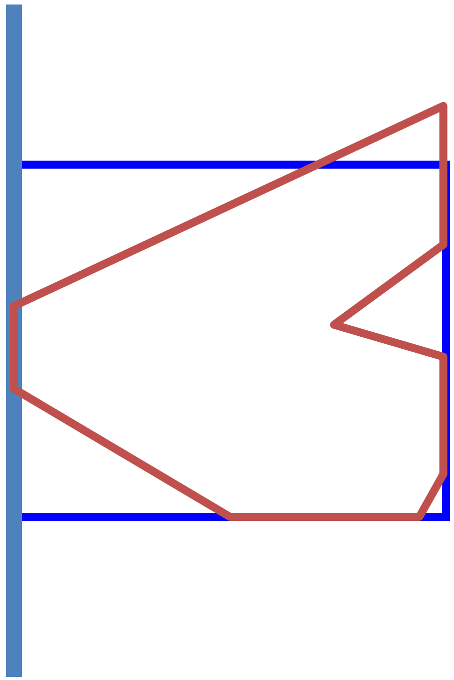
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation



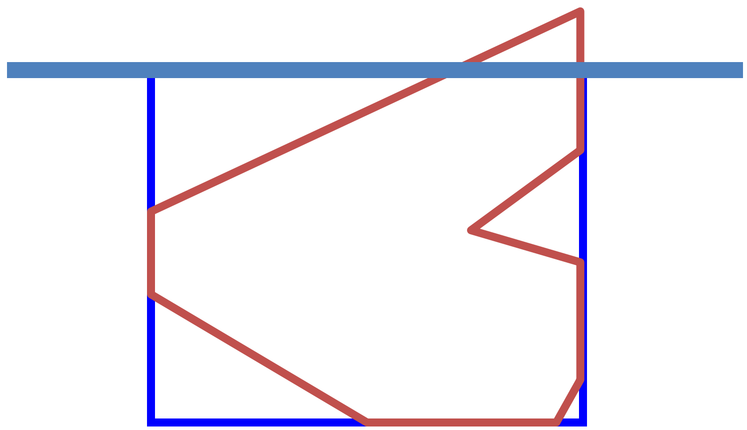
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation



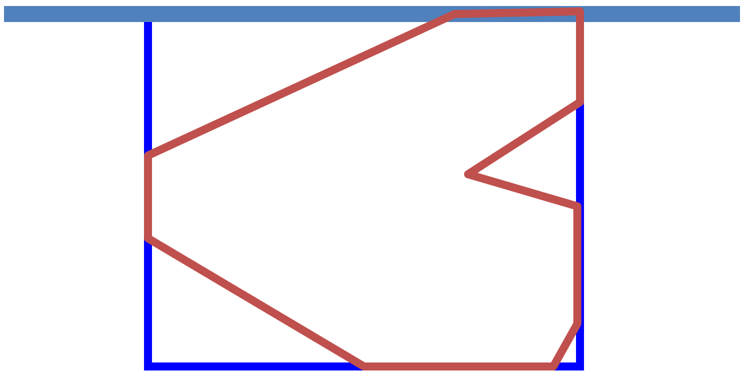
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation



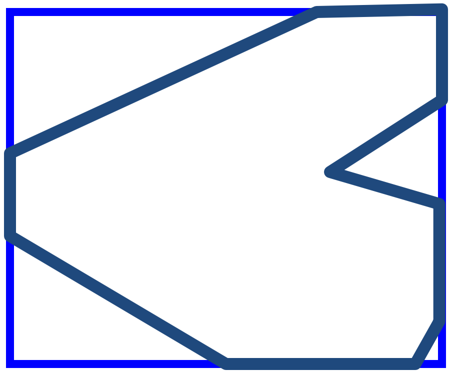
Sutherland-Hodgman Clipping

- Basic idea:
  - Consider each edge of the viewport individually
  - Clip the polygon against the edge equation



Sutherland-Hodgman Clipping

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



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

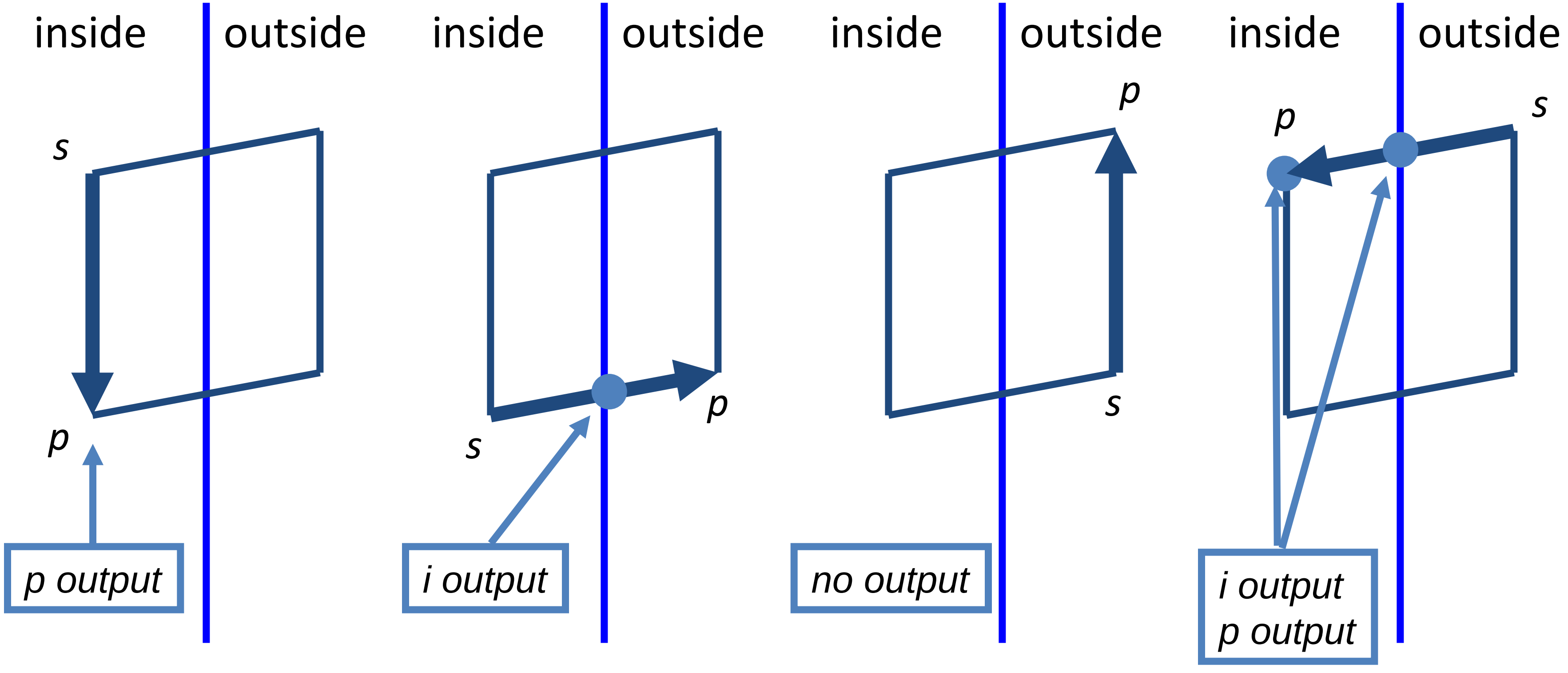
Sutherland-Hodgman Clipping

- 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:

  (Orange 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 P, defined by q and n:

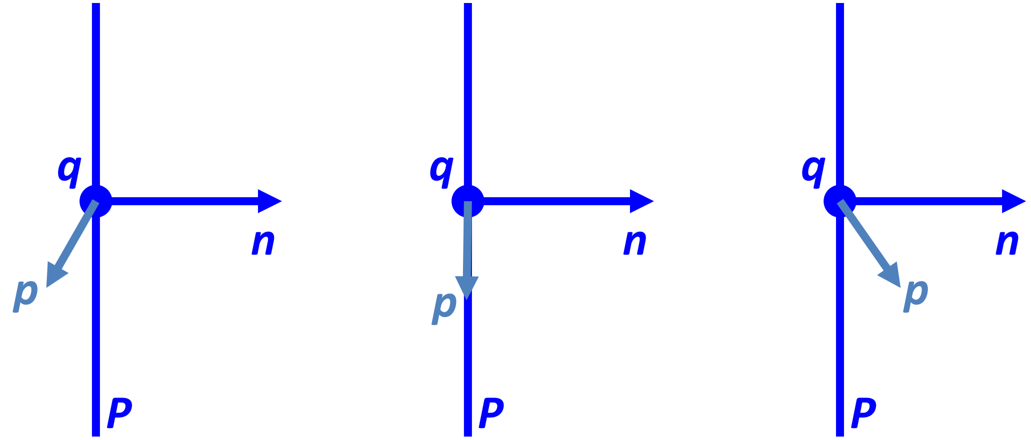
  $(p - q) \bullet n < 0$: p inside $\boldsymbol{P}$

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

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

Remember: $\boldsymbol{p} \bullet n = |p|\ |n| \cos(\theta)$

θ = angle between p and n



Finding Line-Plane Intersections

- Edge intersects plane P where E(t) is on P
  - $\boldsymbol{q}$ is a point on ***P***
  - $\boldsymbol{n}$ is normal to ***P***

$$(\boldsymbol{L}(t) - \boldsymbol{q}) \bullet \boldsymbol{n} = 0$$

$$(L_0 + (L_1 - L_0)\, t - \boldsymbol{q}) \bullet \boldsymbol{n} = 0$$

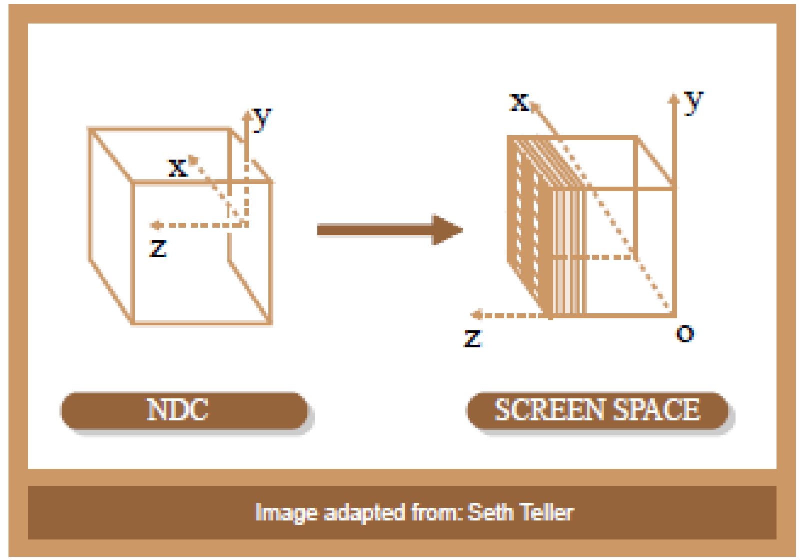
$$t = [(\boldsymbol{q} - \boldsymbol{L_0}) \bullet \boldsymbol{n}] \,/\, [(\boldsymbol{L_1} - \boldsymbol{L_0}) \bullet \boldsymbol{n}]$$

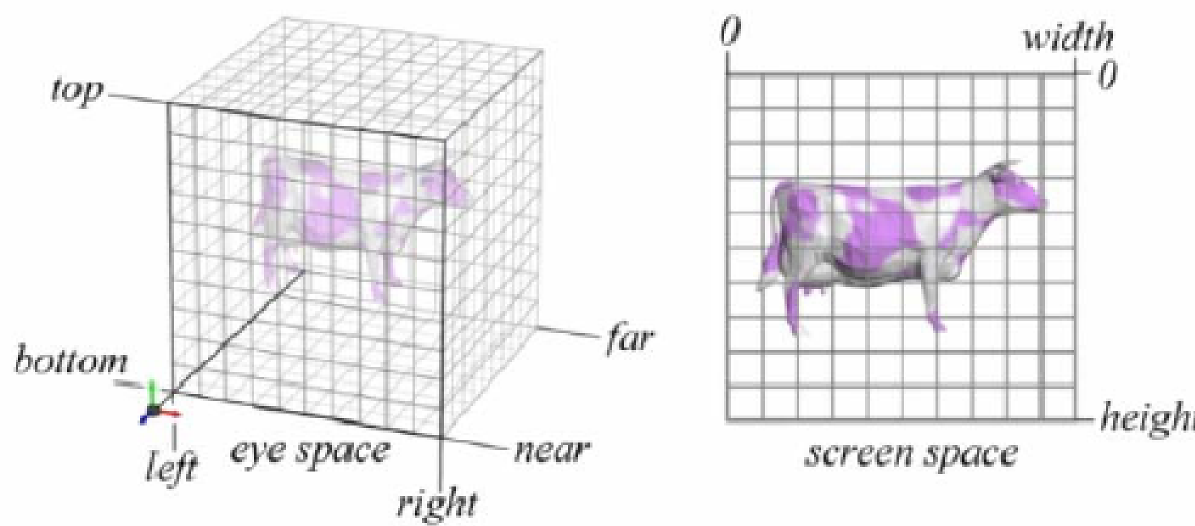
  - The intersection point $\boldsymbol{i} = \boldsymbol{L}(t)$ for this value of t

Projection

- Modeling Transformations
- Illumination (Shading)
- Viewing Transformation (Perspective / Orthographic)
- Clipping
- **Projection (to Screen Space)**
- Scan Conversion (Rasterization)
- Visibility / Display

- The objects are projected to the 2D image place (screen space)





Scan Conversion (Rasterization)

Modeling Transformations

Illumination (Shading)

Viewing Transformation (Perspective / Orthographic)

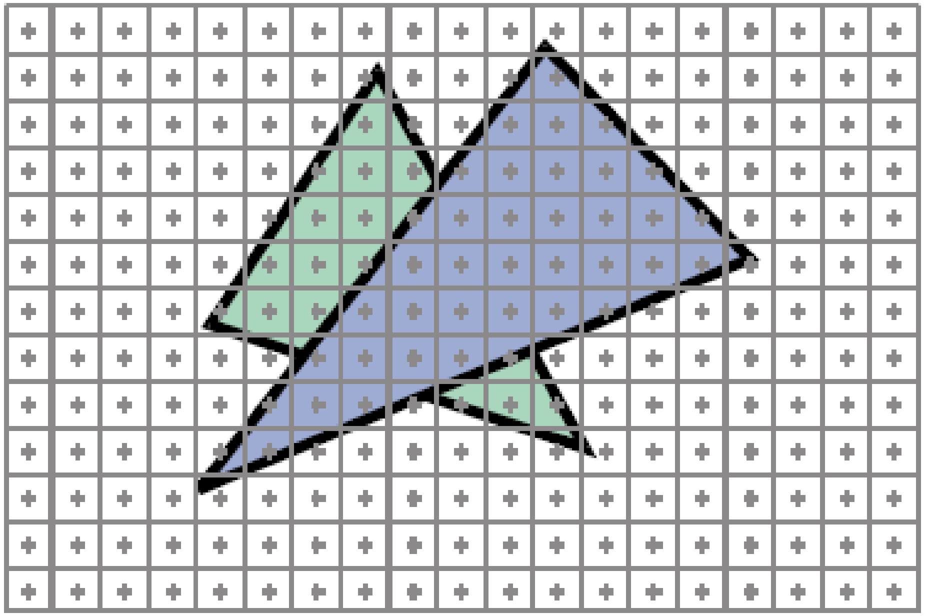
Clipping

Projection (to Screen Space)

Scan Conversion (Rasterization)

Visibility / Display

- Rasterizes objects into pixels
- Interpolate values as we go (color, depth, etc.)

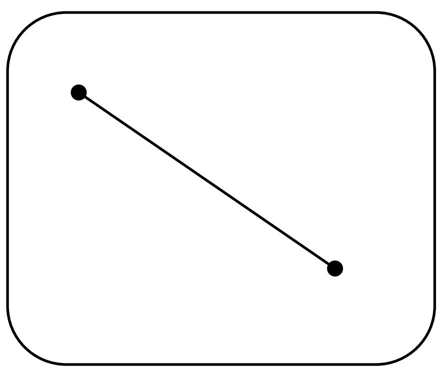


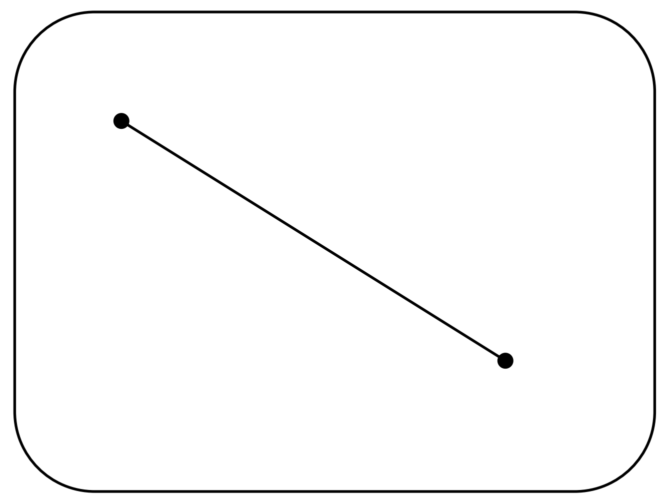
Vector Graphics

- How to generate an image using vectors
  - A line is represented by endpoints (10,10) to (90,90)
  - The points along the line are computed using a line equation
    - y = mx + b
  - If you want the image larger, no problem…

Cheap transmission

Computation required



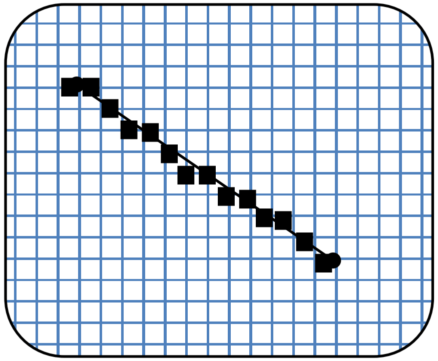


Raster Graphics

- How to generate a line using rasters
  - A line is represented by assigning some pixels a value of 1

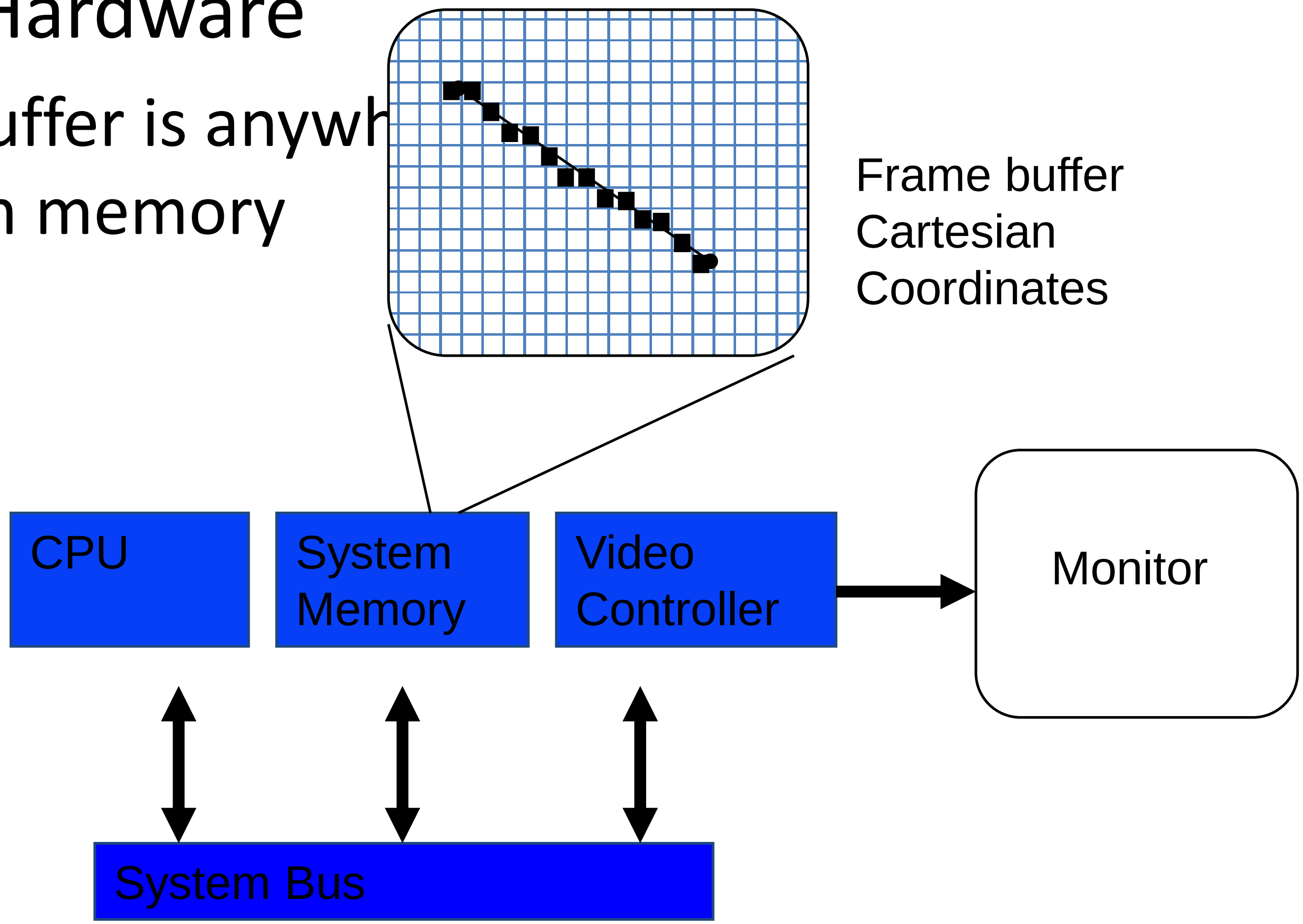
    Lot's of extra info to communicate

    No computation
  - The entire line is specified by the pixel values
    - What do we do to make image larger?



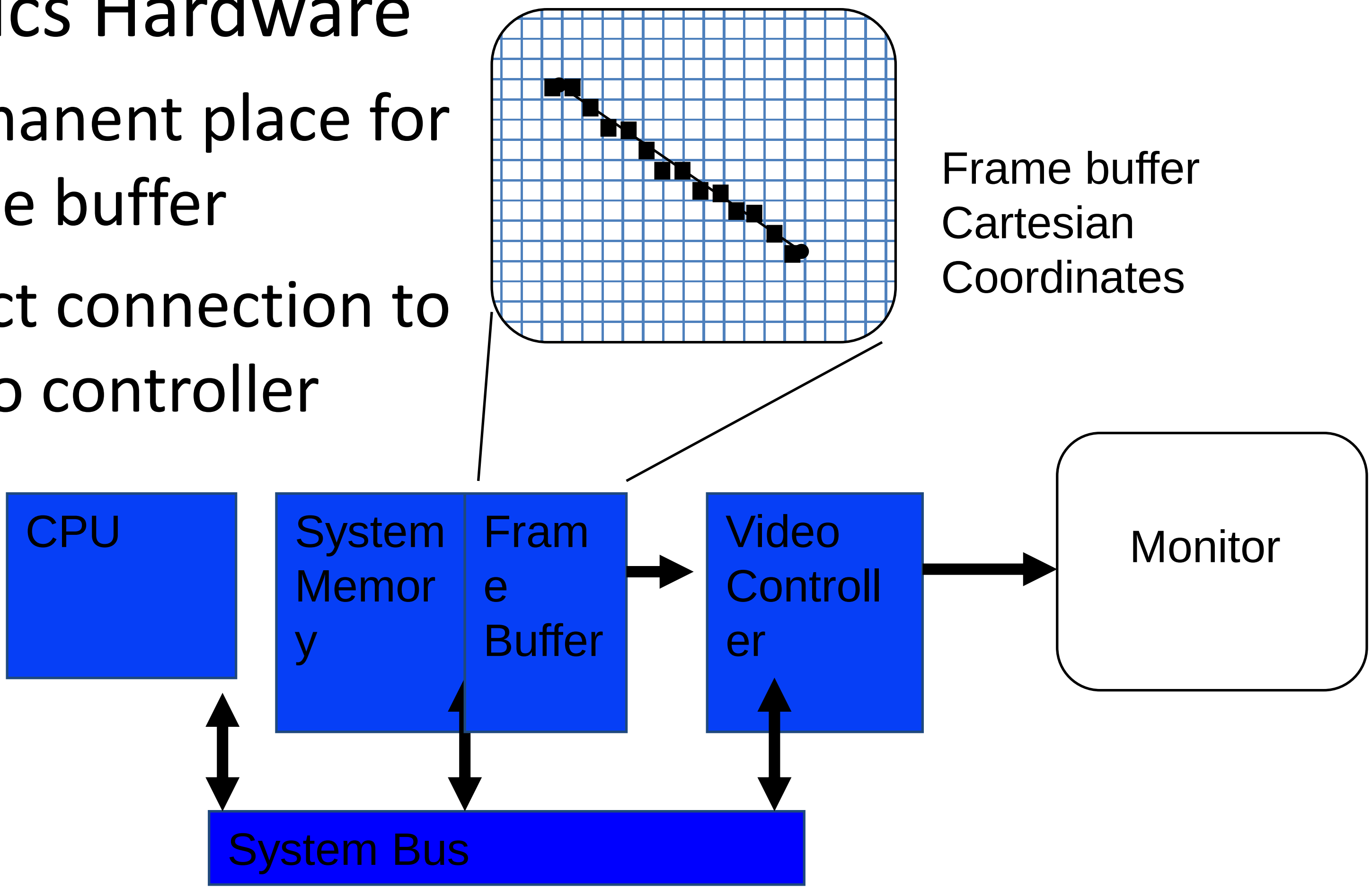
Video Controllers

- Graphics Hardware
  - Frame buffer is anywh
    in system memory



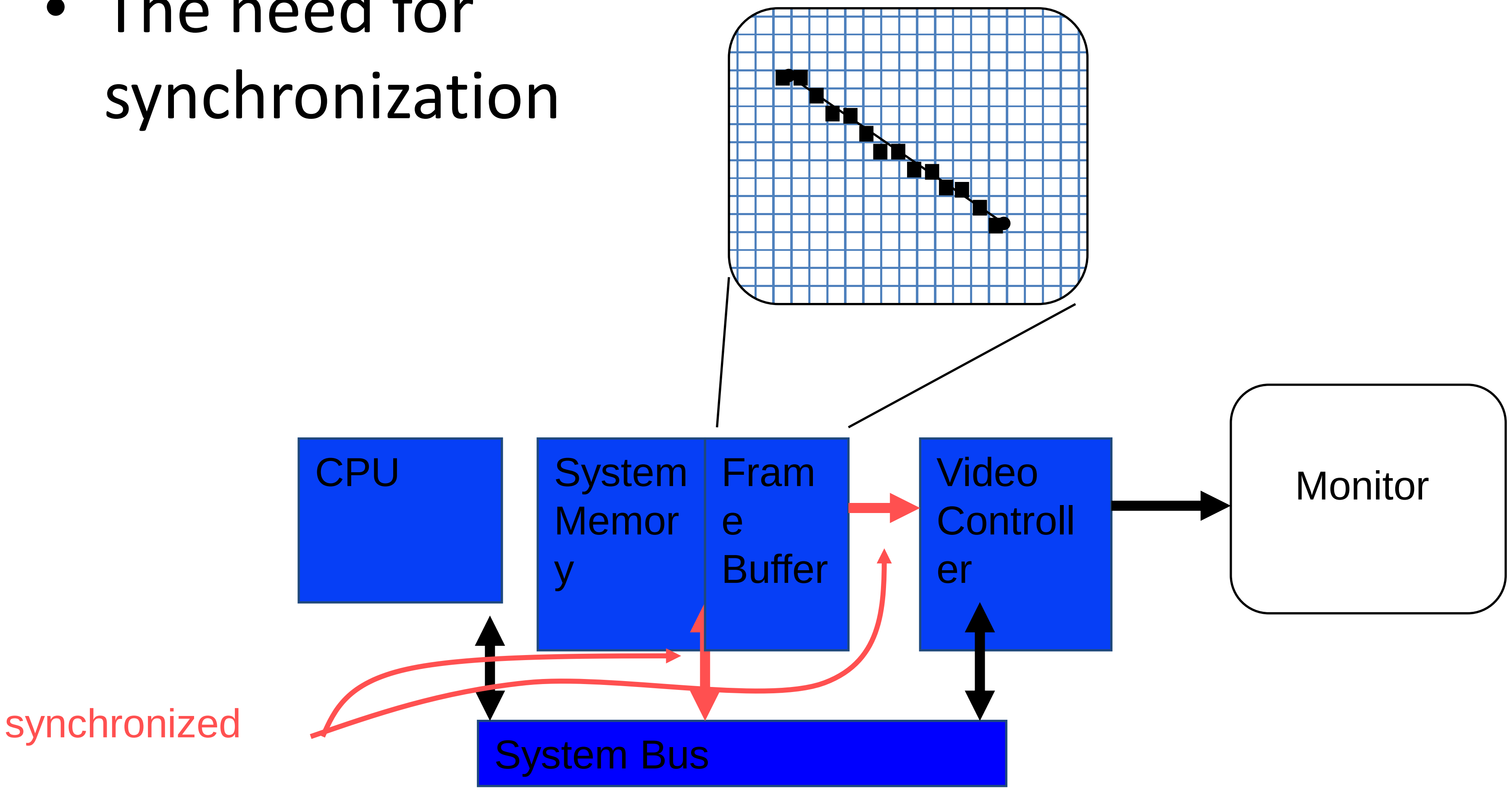
Video Controllers

- Graphics Hardware
  - Permanent place for frame buffer
  - Direct connection to video controller



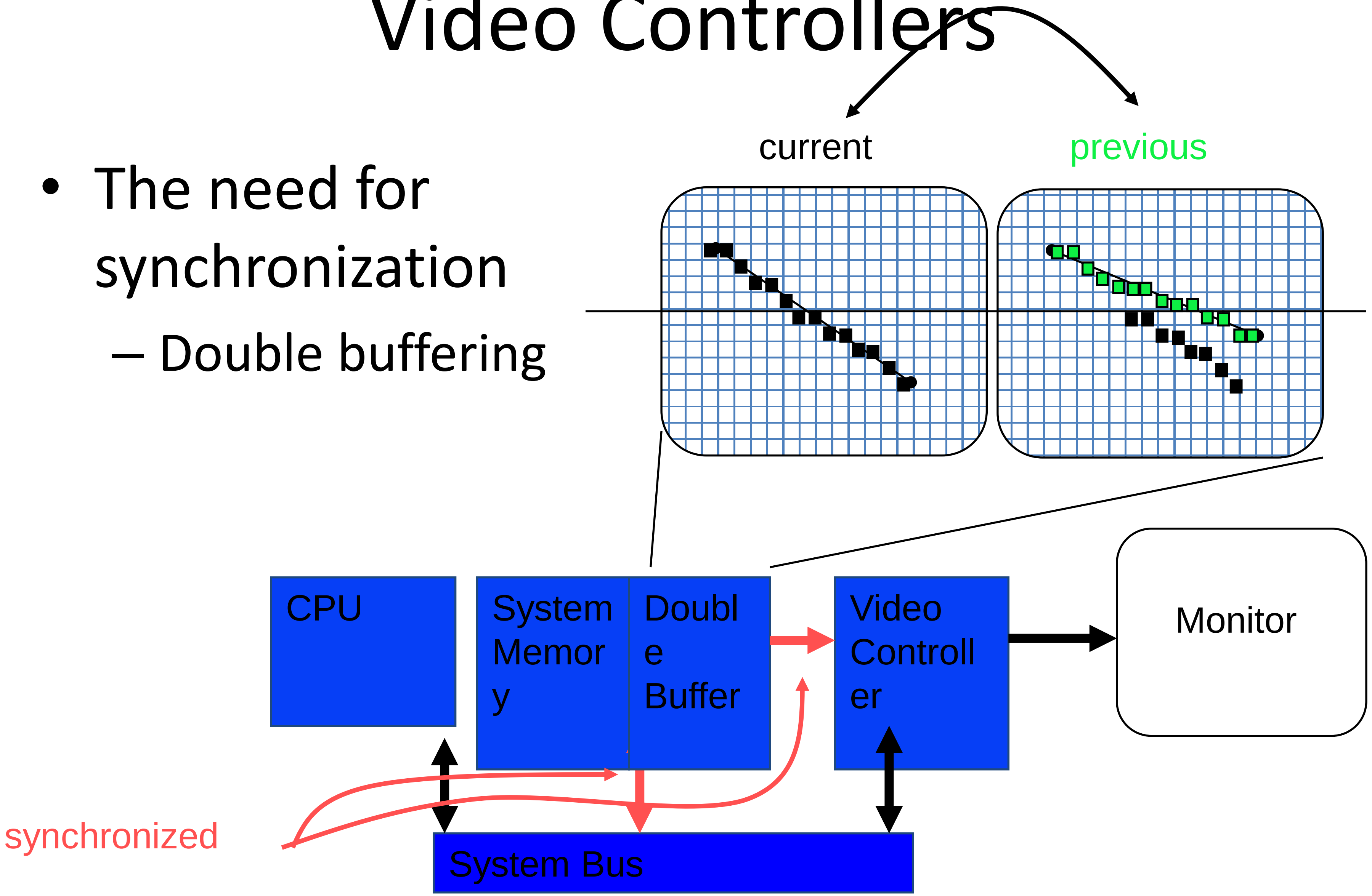
Video Controllers

- The need for synchronization



Video Controllers

- The need for synchronization
  - Double buffering



Frame Buffer

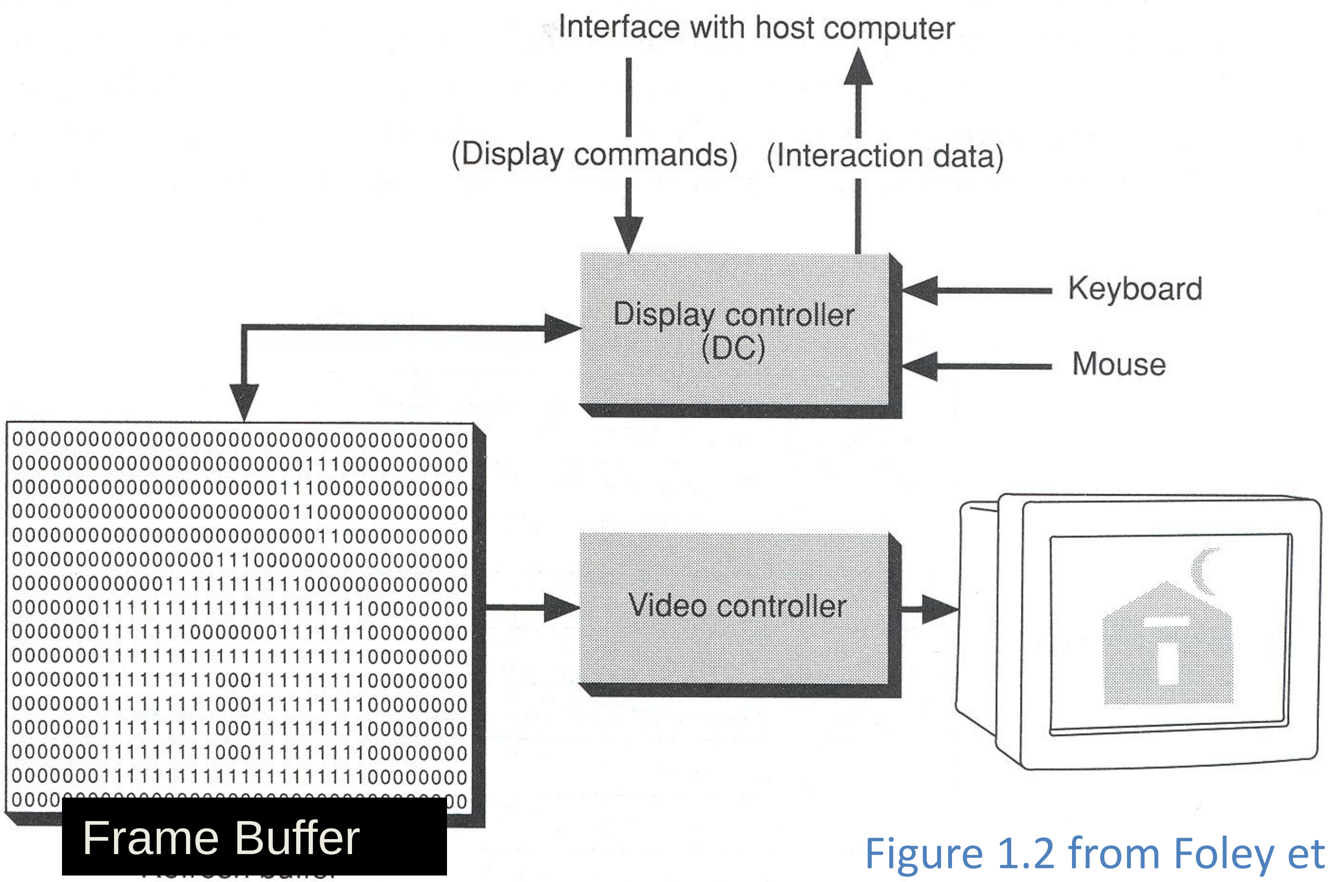
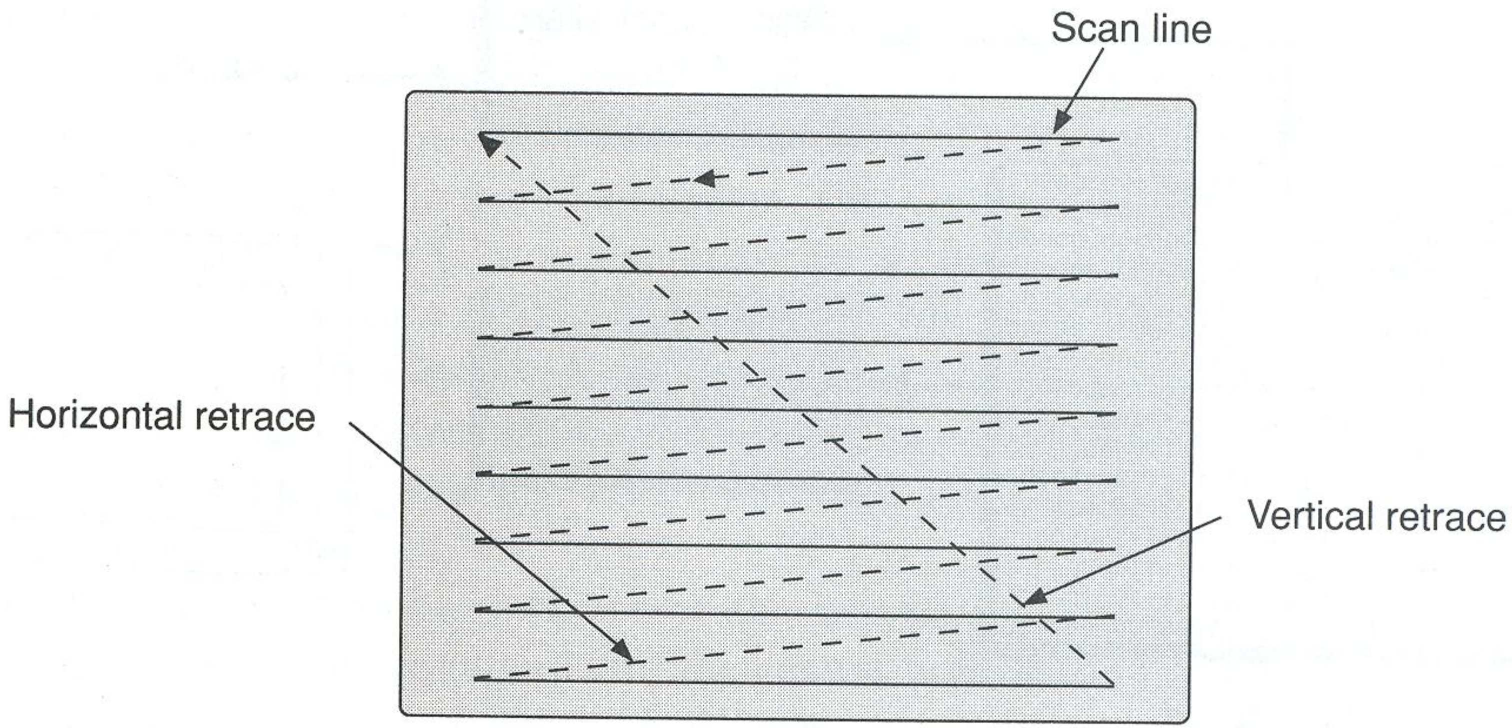


Figure 1.2 from Foley et al.

Frame Buffer Refresh

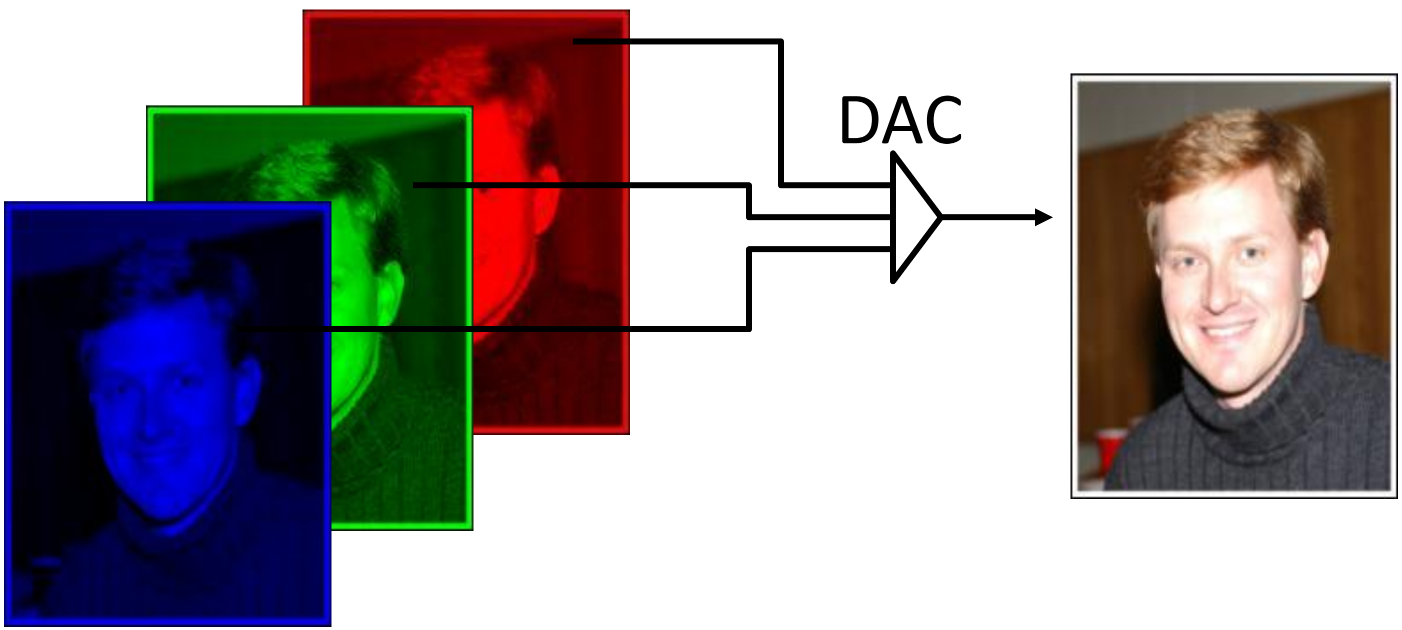


Refresh rate is usually 30-75Hz

Figure 1.3 from FvDFH

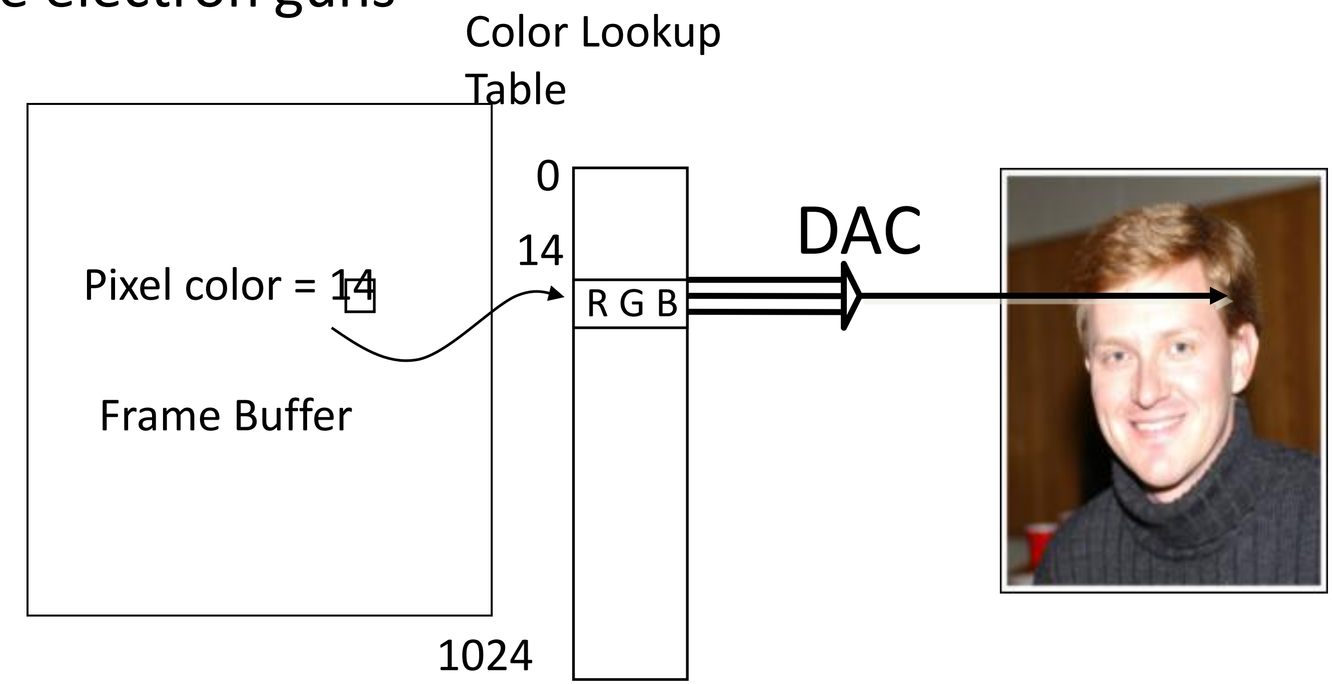
Direct Color Framebuffer

- Store the actual intensities of R, G, and B individually in the framebuffer
- 24 bits per pixel = 8 bits red, 8 bits green, 8 bits blue
  - 16 bits per pixel = ? bits red, ? bits green, ? bits blue

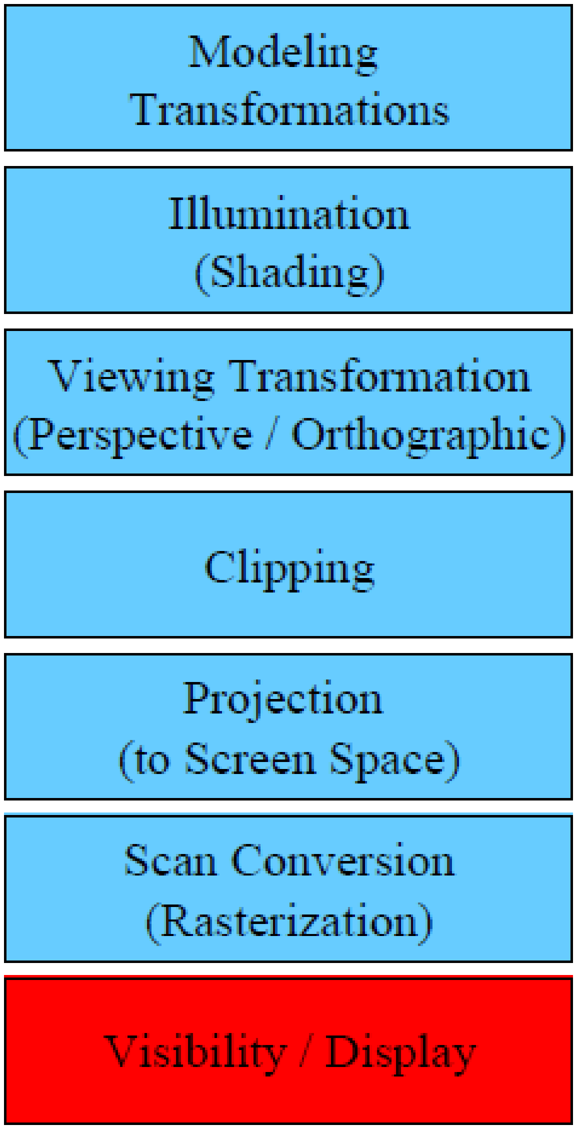


Color Lookup Framebuffer

- Store indices (usually 8 bits) in framebuffer
- Display controller looks up the R,G,B values before triggering the electron guns



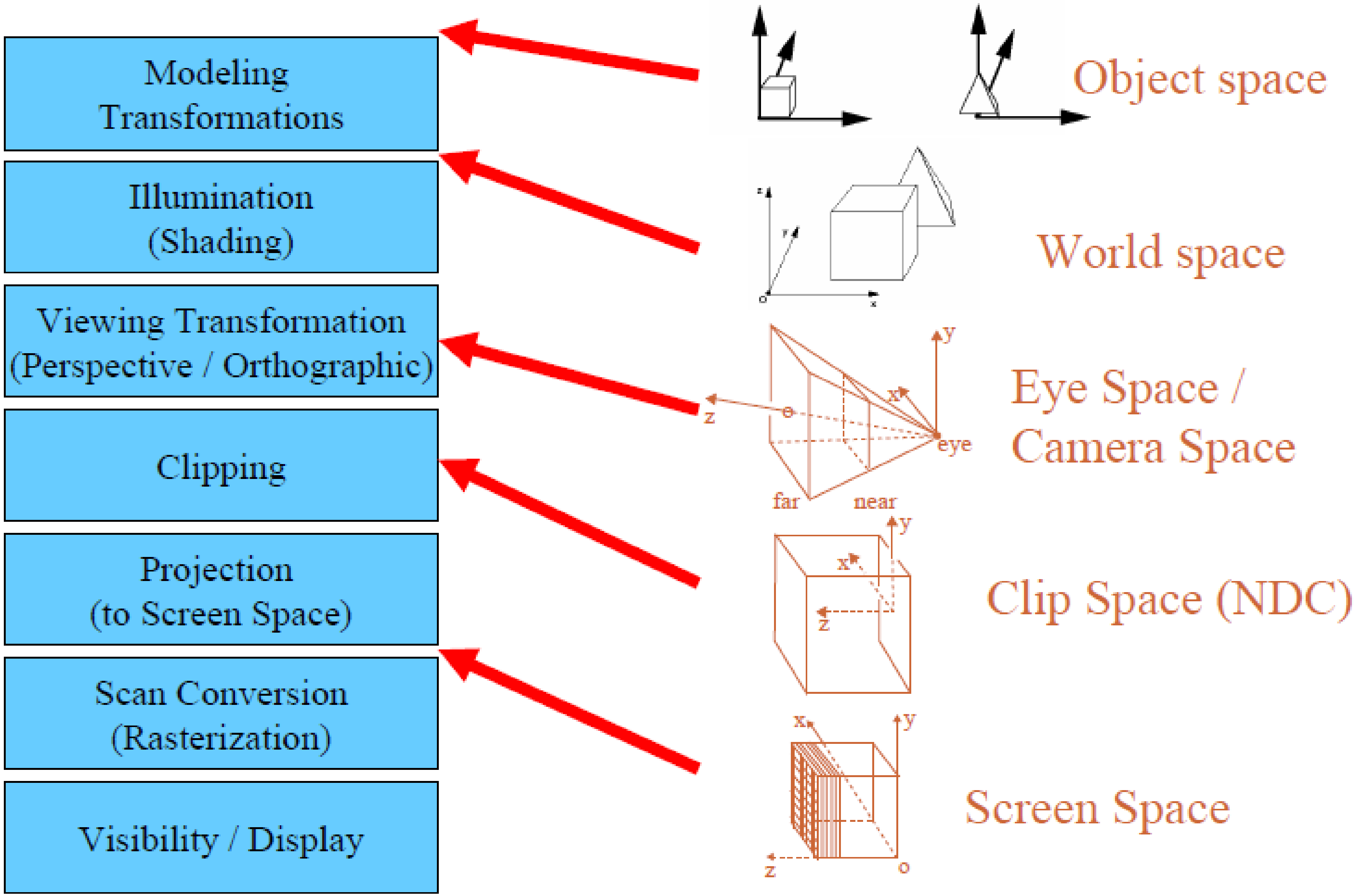
Visibility / Display



- Each pixel remembers the closest object (depth buffer)

- Almost every step in the graphics pipeline involves a change of coordinate system. Transformations are central to understanding 3D computer graphics.

Coordinate Systems in the Pipeline



Recap: Rendering Pipeline

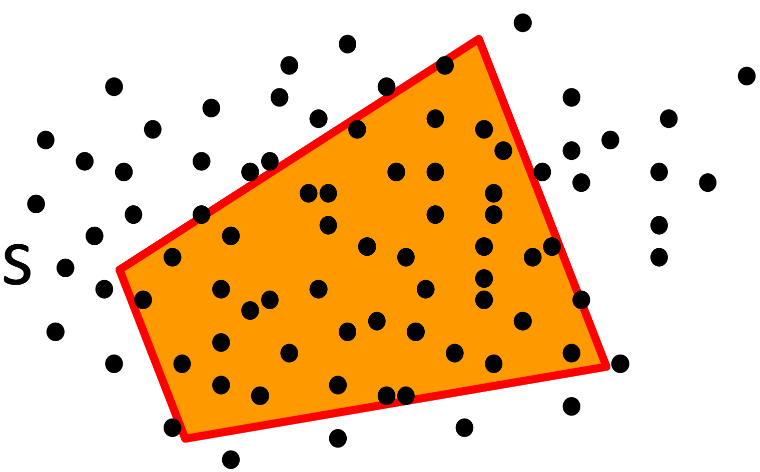
  - Modeling transformations
  - Viewing transformations
  - Projection transformations
  - Clipping
  - Scan conversion
- We now know everything about how to draw a polygon on the screen, except visible surface determination

Invisible Primitives

- Why might a polygon be invisible?
  - Polygon outside the *field of view*
  - Polygon is *backfacing*
  - Polygon is *occluded* by object(s) nearer the viewpoint
- For efficiency reasons, we want to avoid spending work on polygons outside field of view or backfacing
- For efficiency and correctness reasons, we need to know when polygons are occluded

View Frustum Clipping

- Remove polygons entirely outside frustum
  - Note that this includes polygons “behind” eye (actually behind near plane)
- Pass through polygons entirely inside frustum
- Modify remaining polygons to include only portions intersecting view frustum



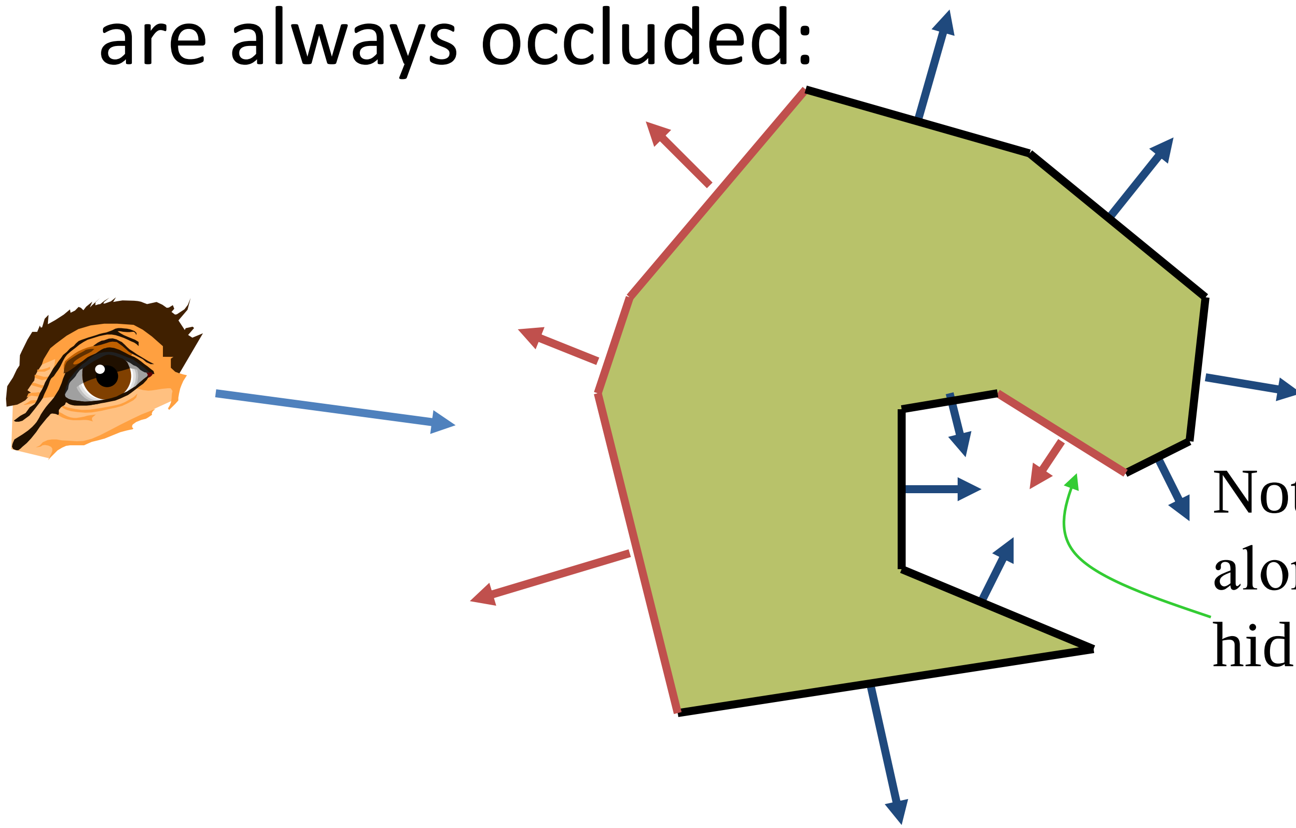
Back-Face Culling

- Most objects in scene are typically “solid”
- More rigorously: closed, orientable manifolds
  - Must not cut through itself
  - Must have two distinct sides
    - A sphere is orientable since it has two sides, 'inside' and 'outside'.
    - A Mobius strip or a Klein bottle is not orientable
  - Cannot “walk” from one side to the other
    - A sphere is a closed manifold whereas a plane is not

www.kleinbottle.com

Back-Face Culling

- On the surface of a closed manifold, polygons whose normals point away from the camera are always occluded:



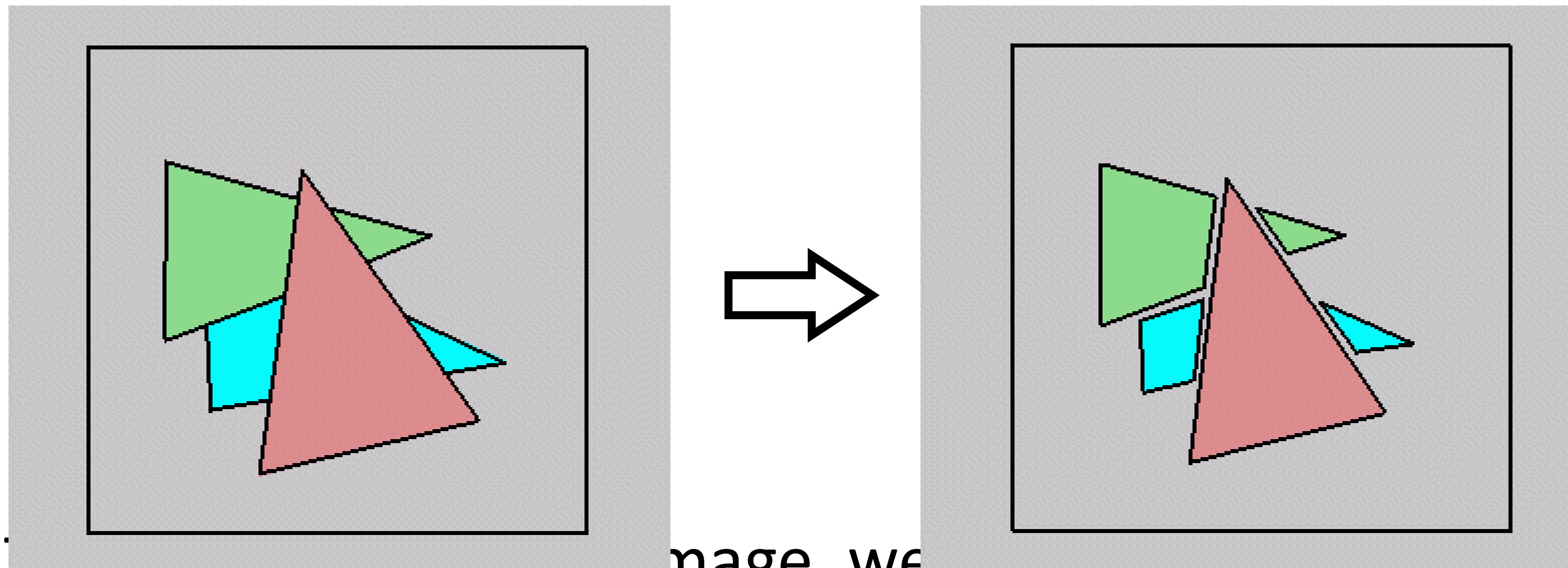
Note: backface culling alone doesn't solve the hidden-surface problem!

Back-Face Culling

- Not rendering backfacing polygons improves performance
  - *By how much?*
    - *Reduces by about half the number of polygons to be considered for each pixel*
      - *Every front-facing polygon must have a corresponding rear-facing one*

Occlusion

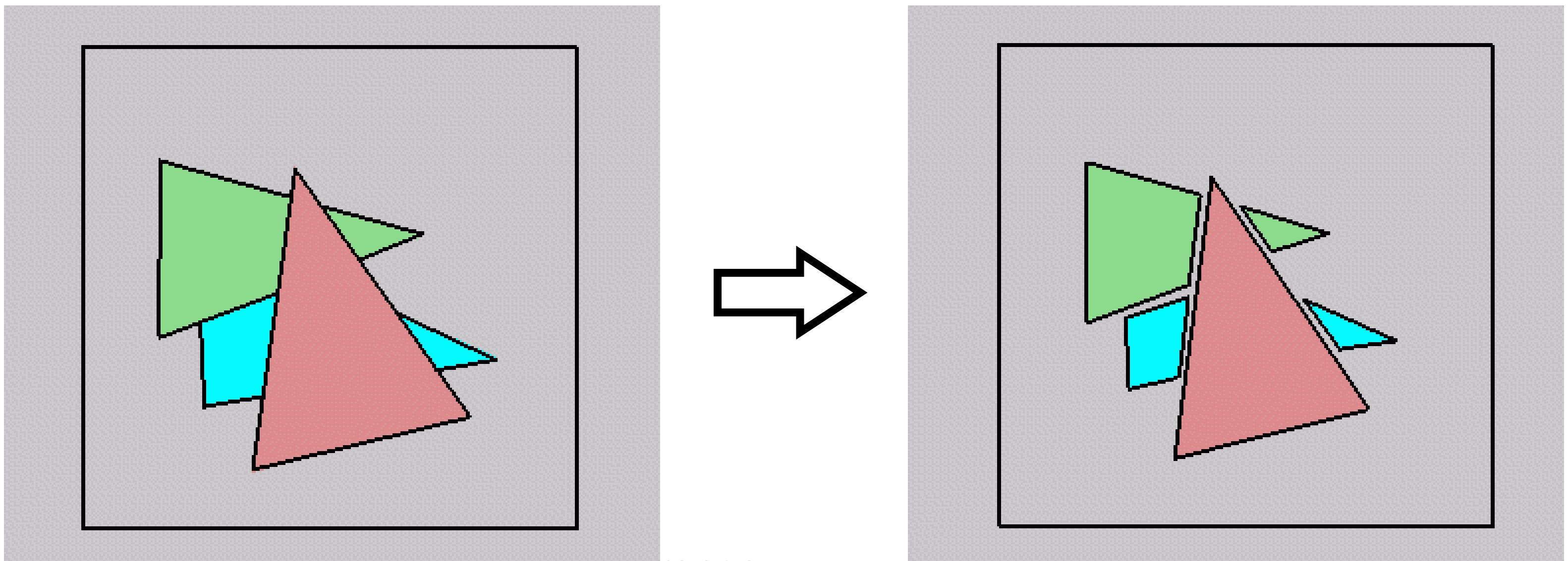
- For most interesting scenes, some polygons will overlap:



- To render the correct image, we need to determine which polygons occlude which

Painter’s Algorithm

- Simple approach: render the polygons from back to front, “painting over” previous polygons:

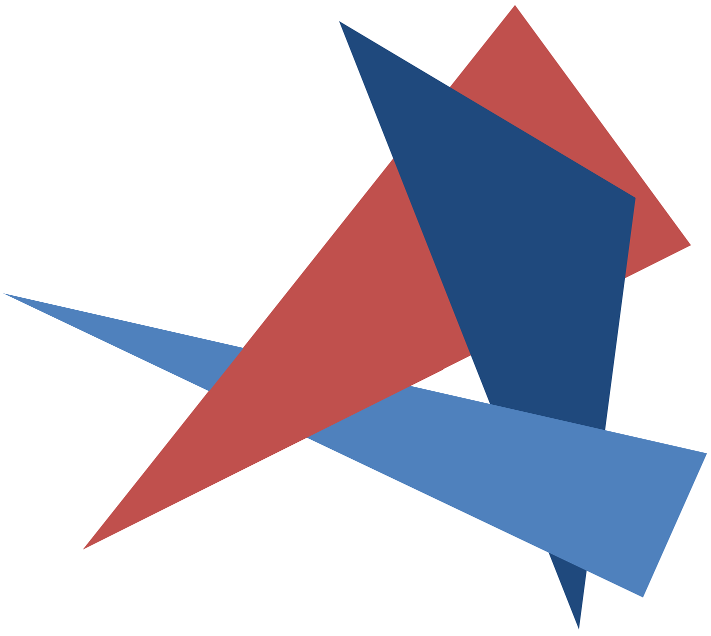


  - Draw blue, then green, then orange

- Will this work in the general case?

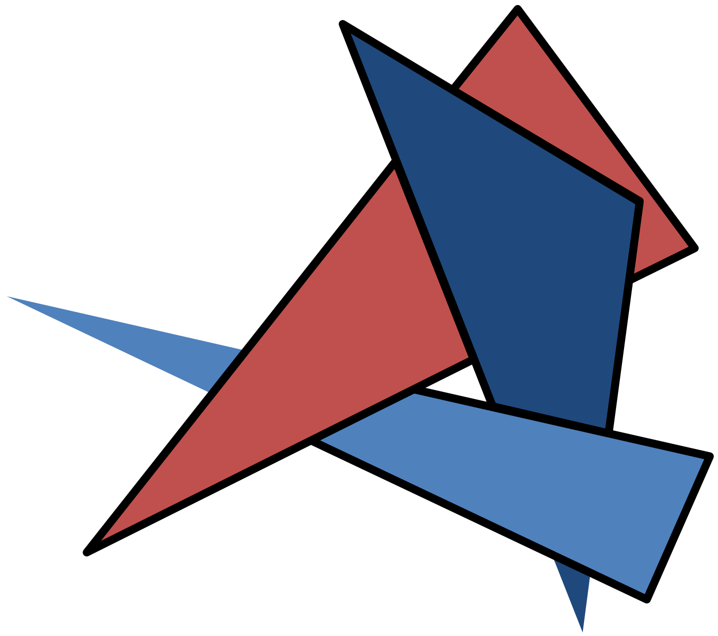
Painter's Algorithm: Problems

- Intersecting polygons present a problem
- Even non-intersecting polygons can form a cycle with no valid visibility order:



Analytic Visibility Algorithms

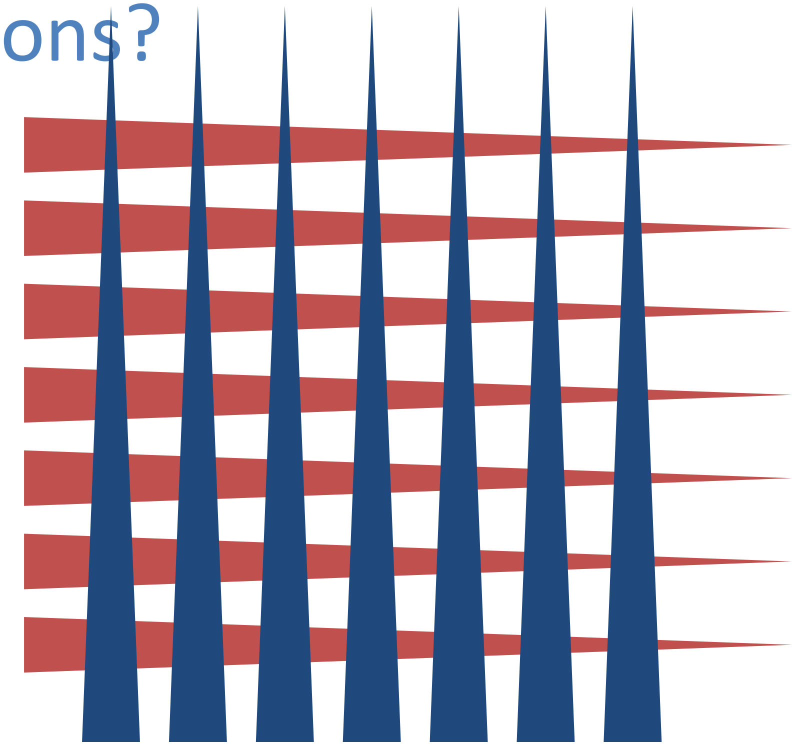
- Early visibility algorithms computed the set of visible polygon fragments directly, then rendered the fragments to a display:



Analytic Visibility Algorithms

- What is the minimum worst-case cost of computing the fragments for a scene composed of n polygons?
- Answer:
  $O(n^2)$

- What's your opinion
- of $O(n^2)$?



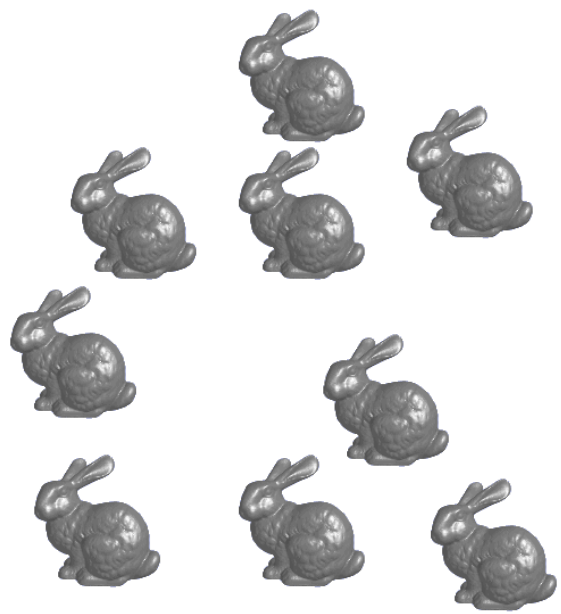
Analytic Visibility Algorithms

- So, for about a decade (late 60s to late 70s) there was intense interest in finding efficient algorithms for hidden surface removal
- We'll talk about two:
  - *Binary Space-Partition (BSP) Trees*
  - *Warnock's Algorithm*

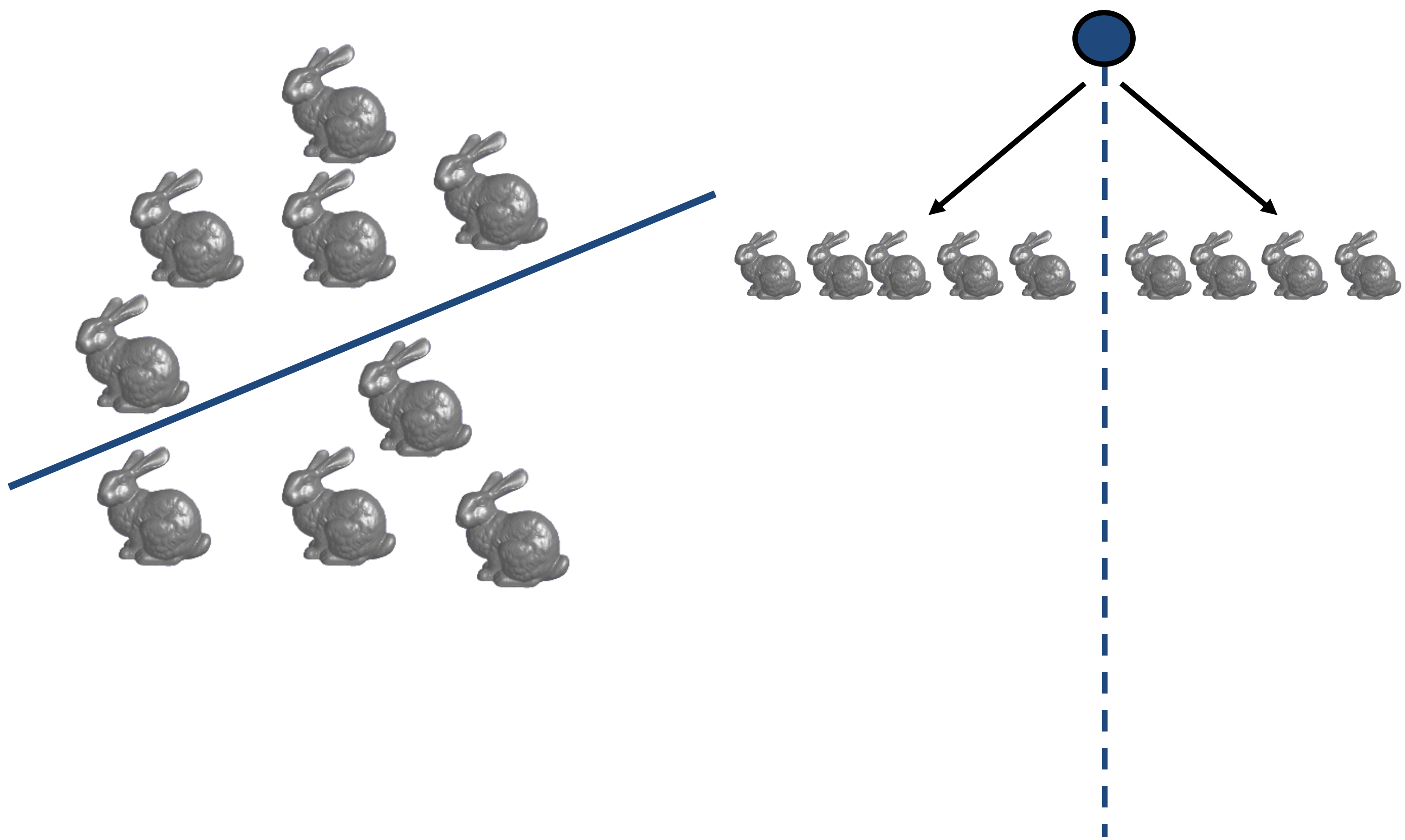
Binary Space Partition Trees (1979)

- BSP tree: organize all of space (hence partition) into a binary tree
  - *Preprocess*: overlay a binary tree on objects in the scene
  - *Runtime*: correctly traversing this tree enumerates objects from back to front
  - Idea: divide space recursively into half-spaces by choosing *splitting planes*
    - Splitting planes can be arbitrarily oriented

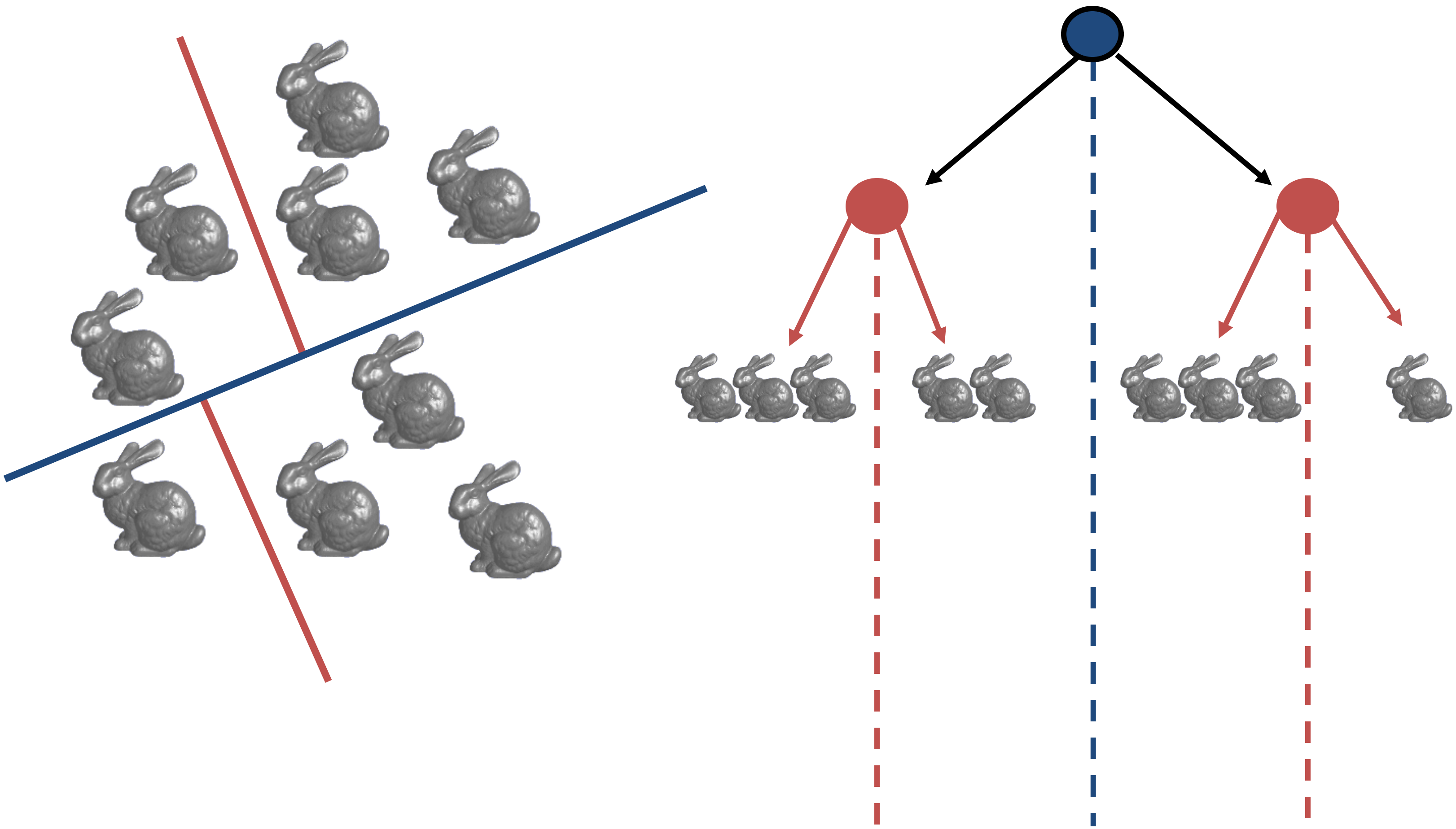
BSP Trees: Objects



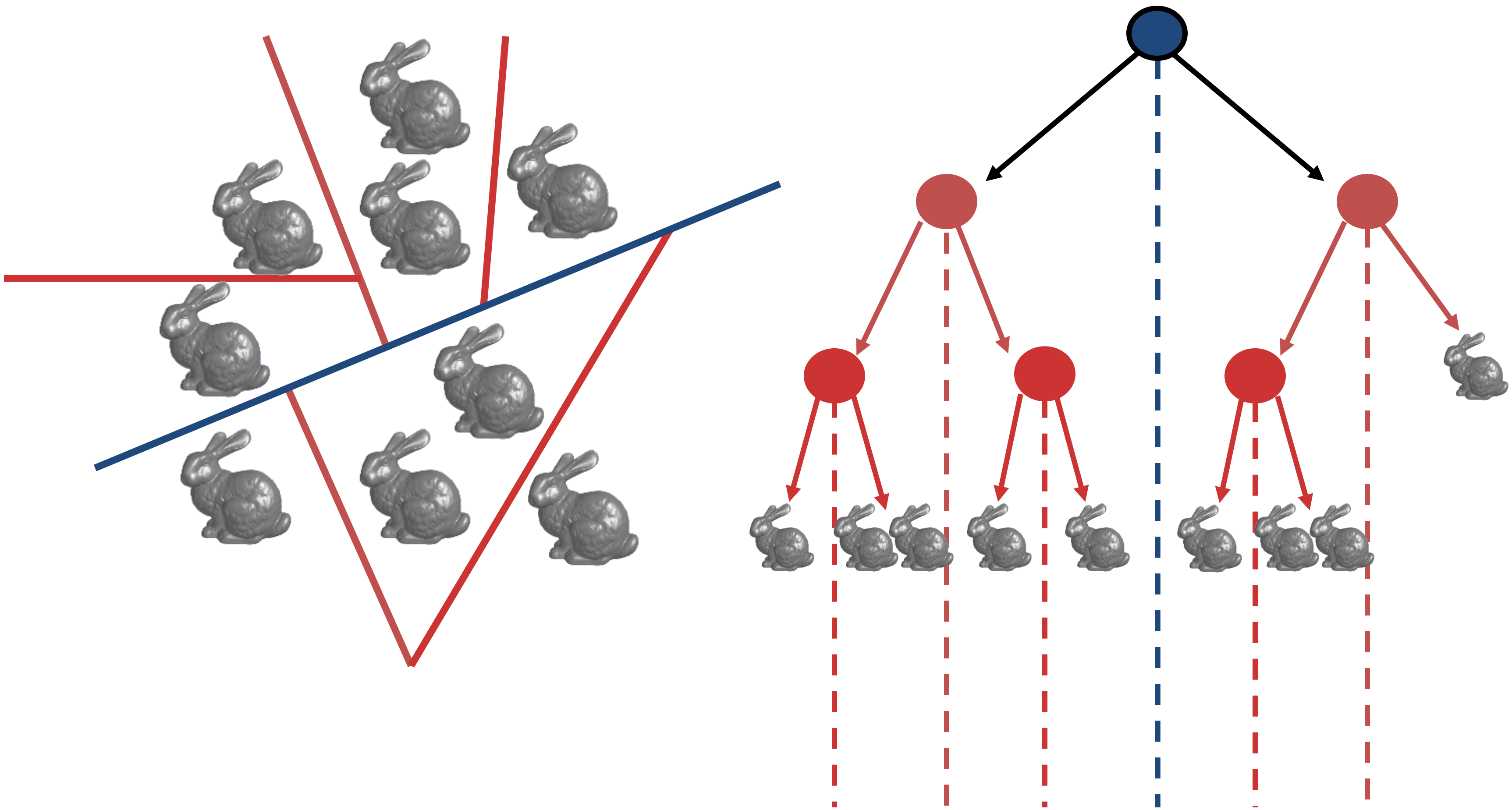
BSP Trees: Objects



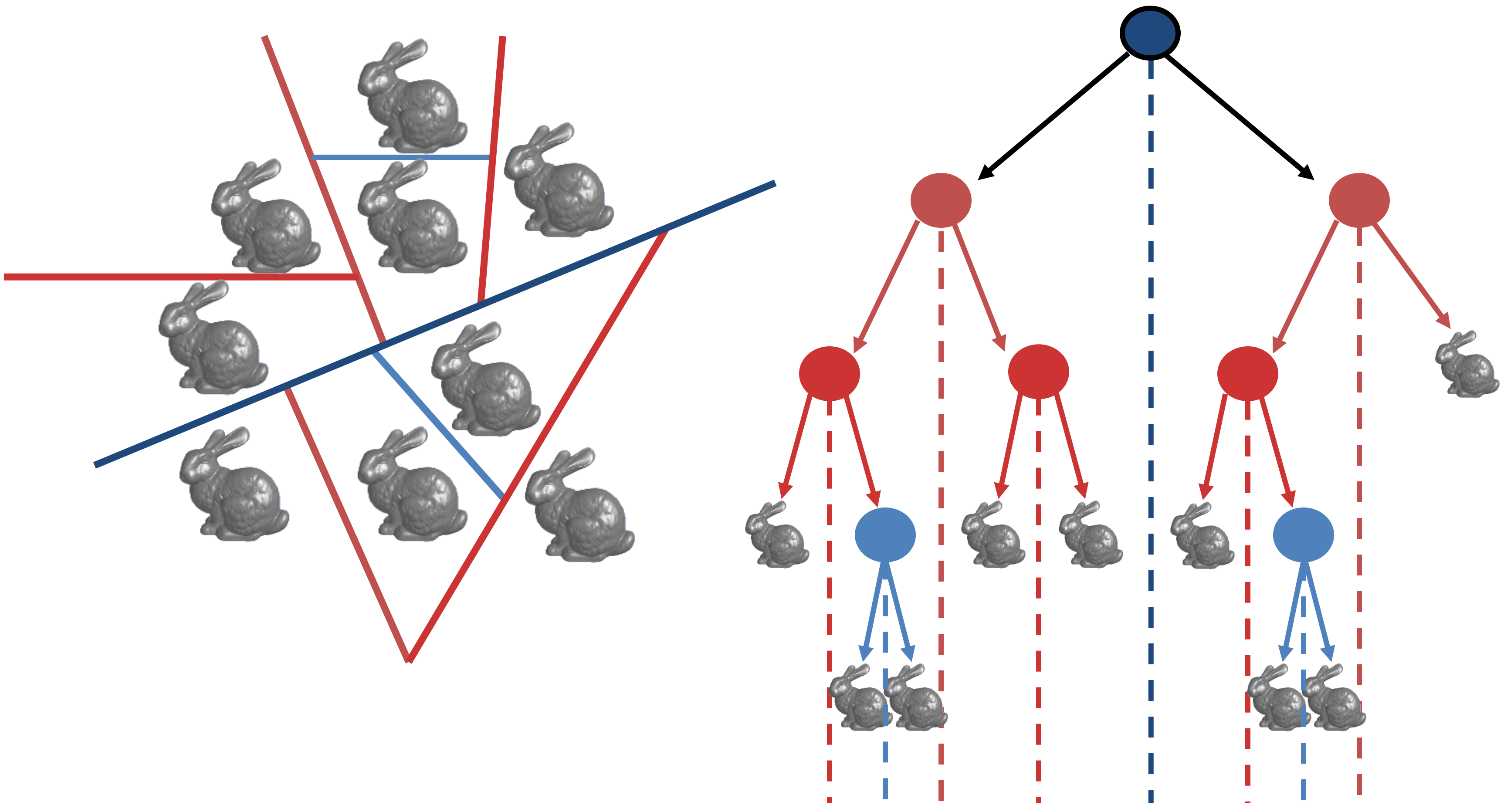
BSP Trees: Objects



BSP Trees: Objects



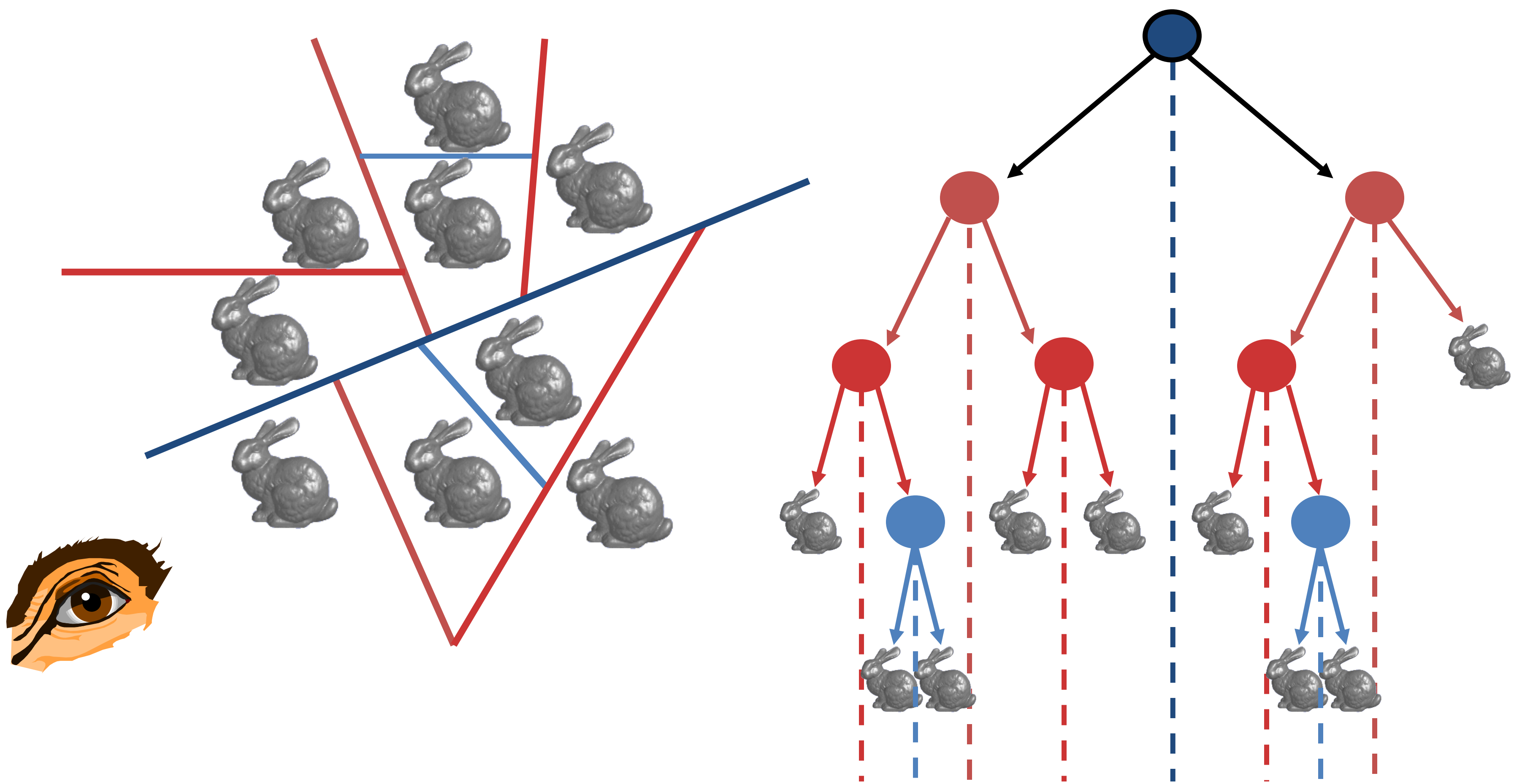
BSP Trees: Objects



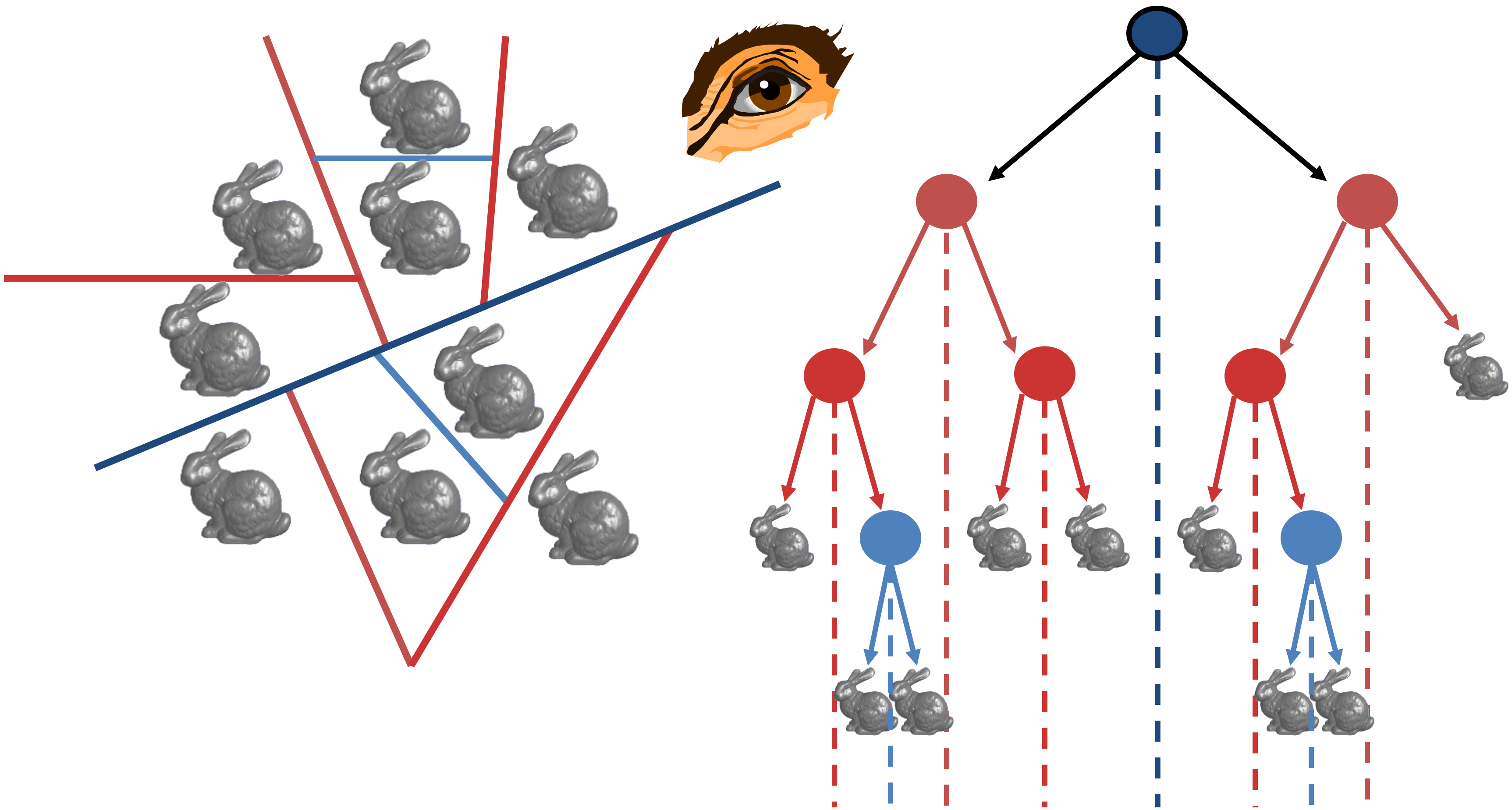
Rendering BSP Trees

- `renderBSP(BSPtree *T)`
- `    BSPtree *near, *far;`
- `    if (eye on left side of T->plane)`
- `        near = T->left; far = T->right;`
- `    else`
- `        near = T->right; far = T->left;`
- `    renderBSP(far);`
- `    if (T is a leaf node)`
- `        renderObject(T)`
- `  renderBSP(near);`

Rendering BSP Trees



Rendering BSP Trees

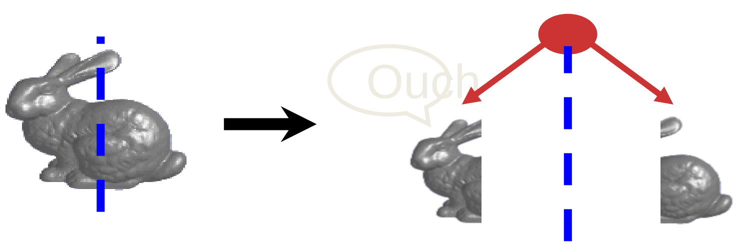


Polygons: BSP Tree Construction

- Split along the plane defined by any polygon from scene
- Classify all polygons into positive or negative half-space of the plane
  - If a polygon intersects plane, split polygon into two and classify them both
- Recurse down the negative half-space
- Recurse down the positive half-space

Discussion: BSP Tree Cons

- No bunnies were harmed in my example
- But what if a splitting plane passes through an object?
  - Split the object; give half to each node

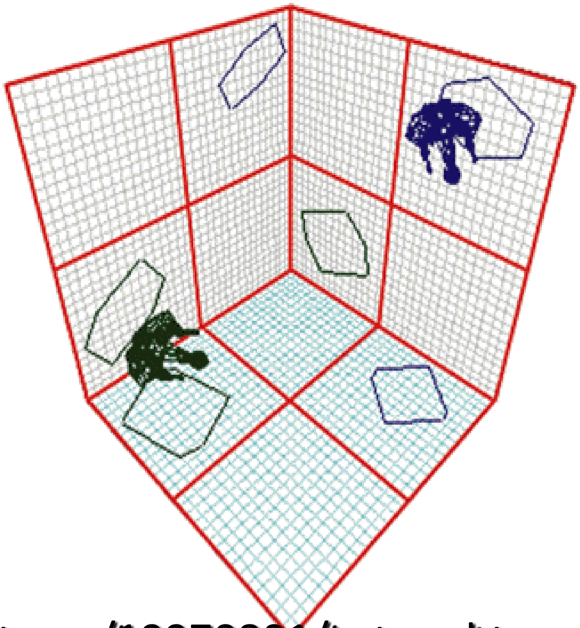


Summary: BSP Trees

- Pros:
  - Simple, elegant scheme
  - Only writes to framebuffer (no reads to see if current polygon is in front of previously rendered polygon, i.e., painters algorithm)
    - Thus very popular for video games (but getting less so)
- Cons:
  - Computationally intense preprocess stage restricts algorithm to static scenes
  - Slow time to construct tree
  - Splitting increases polygon count

Octrees

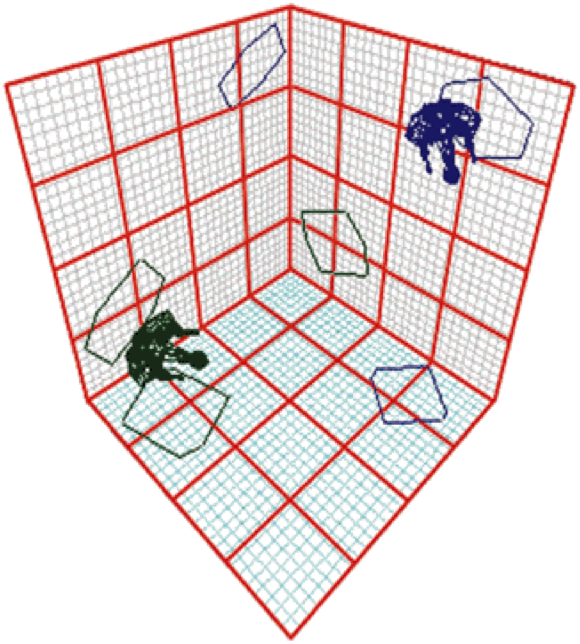
- Frequently used in modern video games
  - A BSP tree subdivides space into a series of half-spaces using single planes
  - An octree subdivides space into eight voxels using three axis-aligned planes
    - A voxel is labeled as having polygons inside it or not



www.gamasutra.com/features/19970801/octree.htm

Octrees

– A voxel may have geometry inside it or subdivide
  - Can have as many as eight children
– Thus we partition 3-D space into 3-D cells
– Checking visibility with polygons now faster due to only checking particular cells
– Quadtrees are a 2-D variant

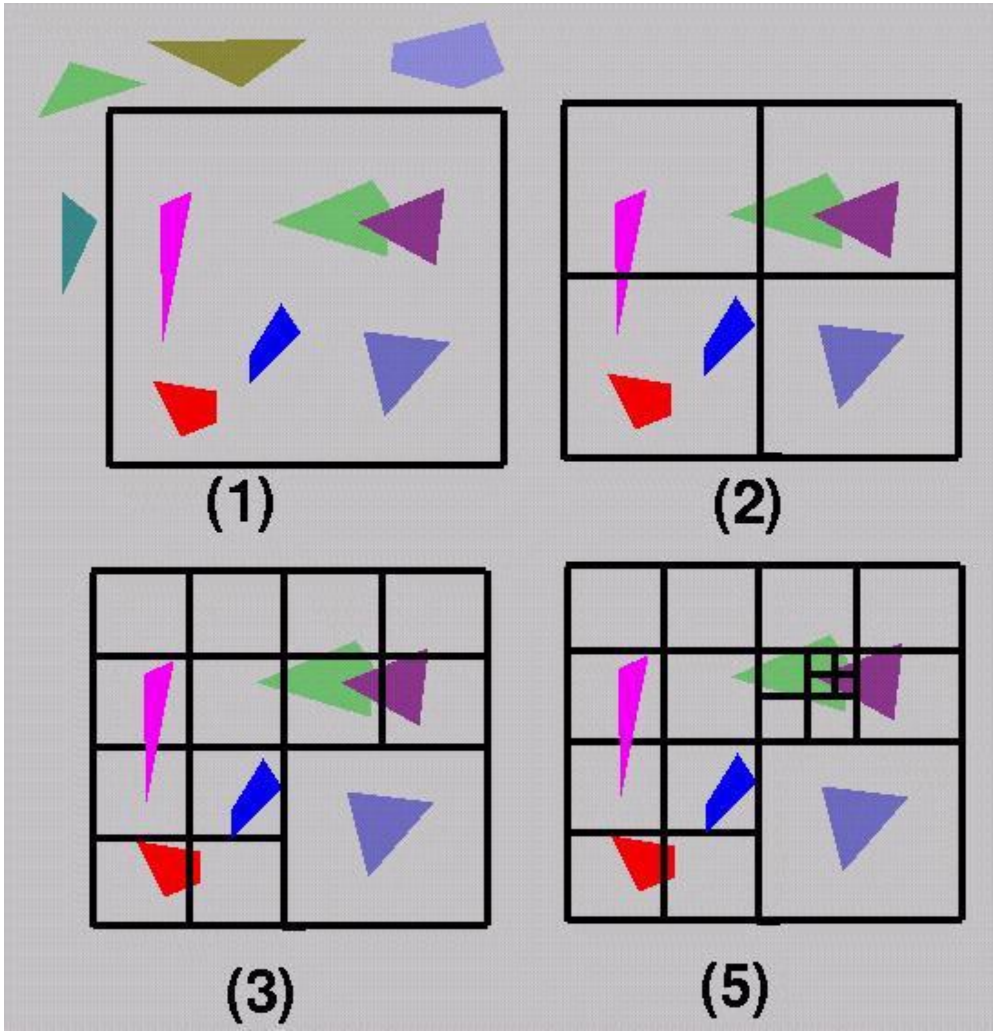


Warnock's Algorithm (1969)

- Elegant scheme based on a powerful general approach common in graphics: if the situation is too complex, subdivide
  - Start with a *root viewport* and a list of all primitives (polygons)
  - Then recursively:
    - Clip objects to viewport
    - If number of objects incident to viewport is zero or one, visibility is trivial
    - Otherwise, subdivide into smaller viewports, distribute primitives among them, and recurse

Warnock's Algorithm

- What is the terminating condition?
- How to determine the correct visible surface in this case?



Warnock's Algorithm

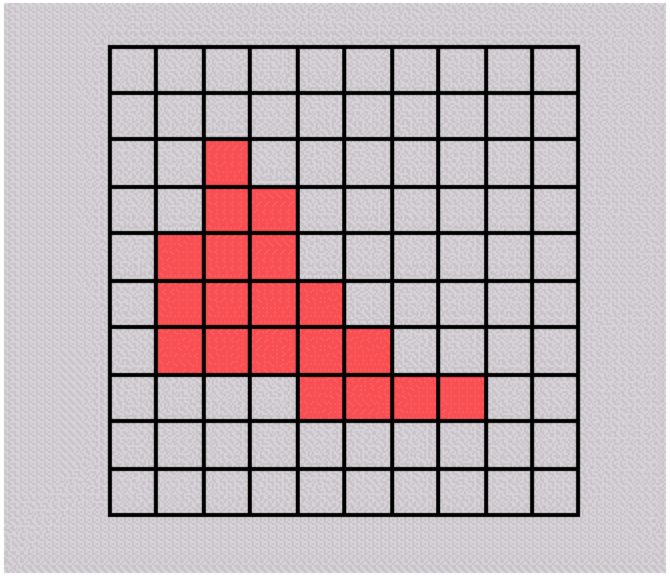
- Pros:
  - Very elegant scheme
  - Extends to any primitive type
- Cons:
  - Hard to embed hierarchical schemes in hardware
  - Complex scenes usually have small polygons and high *depth complexity*
    - Thus most screen regions come down to the single-pixel case

The Z-Buffer Algorithm

- Both BSP trees and Warnock's algorithm were proposed when memory was expensive
  - Example: first 512x512 framebuffer > $50,000!
- Ed Catmull (mid-70s) proposed a radical new approach called z-buffering.
- The big idea: resolve visibility independently at each pixel

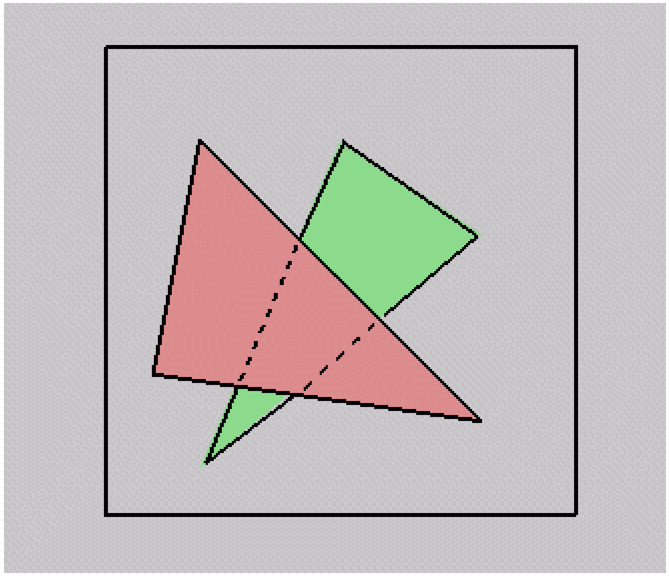
The Z-Buffer Algorithm

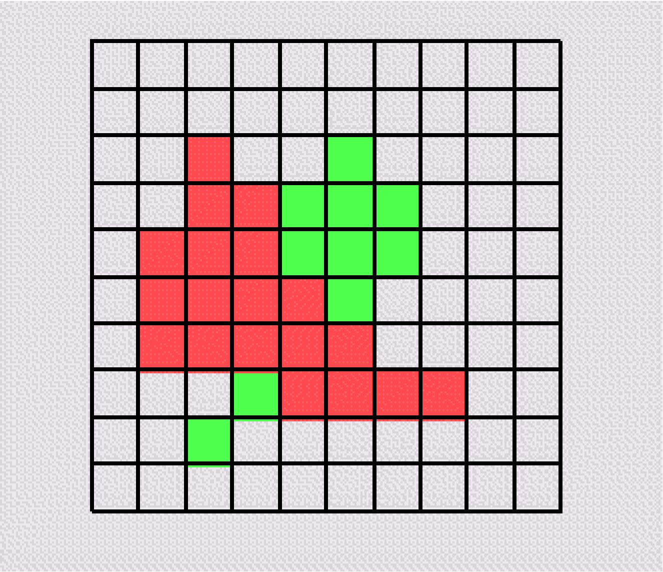
- We know how to rasterize polygons into an image discretized into pixels:



The Z-Buffer Algorithm

- What happens if multiple primitives occupy the same pixel on the screen? Which is allowed to paint the pixel?





The Z-Buffer Algorithm

- Idea: retain depth (Z in eye coordinates) through projection transform
  - Use canonical viewing volumes
  - Each vertex has z coordinate (relative to eye point) intact

The Z-Buffer Algorithm

- Augment framebuffer with Z-buffer or depth buffer which stores Z value at each pixel
  - At frame beginning, initialize all pixel depths to ∞
  - When rasterizing, interpolate depth (Z) across polygon and store in pixel of Z-buffer
  - Suppress writing to a pixel if its Z value is more distant than the Z value already stored there

The Z-Buffer Algorithm

- How much memory does the Z-buffer use?
- Does the image rendered depend on the drawing order?
- Does the time to render the image depend on the drawing order?
- How does Z-buffer load scale with visible polygons? With framebuffer resolution?

Z-Buffer Pros

- Simple!!!
- Easy to implement in hardware
- Polygons can be processed in arbitrary order
- Easily handles polygon interpenetration
- Enables deferred shading
  - Rasterize shading parameters (e.g., surface normal) and only shade final visible fragments

Z-Buffer Cons

- Lots of memory (e.g. 1280x1024x32 bits)
  - With 16 bits cannot discern millimeter differences in objects at 1 km distance
- Read-Modify-Write in inner loop requires fast memory
- Hard to do analytic antialiasing
  - We don’t know which polygon to map pixel back to
- Shared edges are handled inconsistently
  - ***Ordering dependent***
- Hard to simulate translucent polygons
  - We throw away color of polygons behind closest one