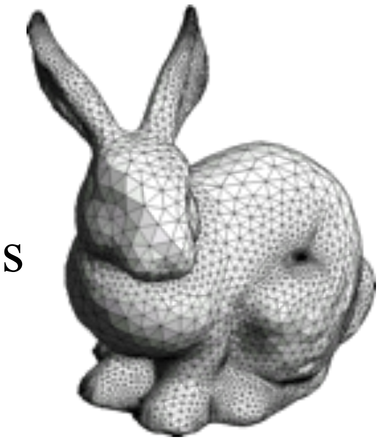
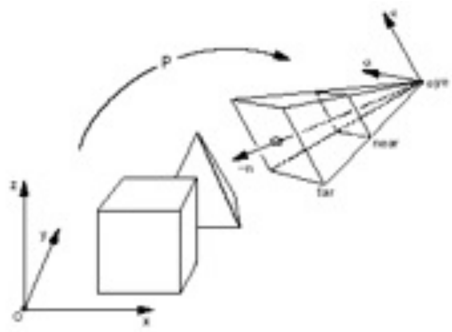
Modern Graphics Pipeline



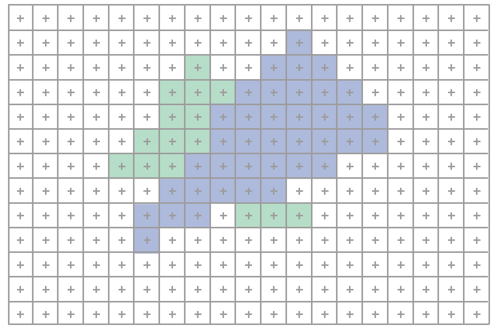
- Input
  - Geometric model
    - Triangle vertices, vertex normals, texture coordinates
  - Lighting/material model (shader)
    - Light source positions, colors, intensities, etc.
    - Texture maps, specular/diffuse coefficients, etc.
  - Viewpoint + projection plane
- Output
  - Color (+depth) per pixel

Colbert & Krivanek

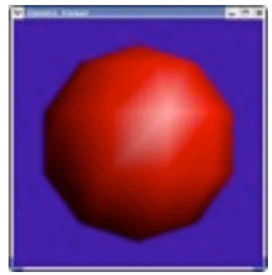
Modern Graphics Pipeline



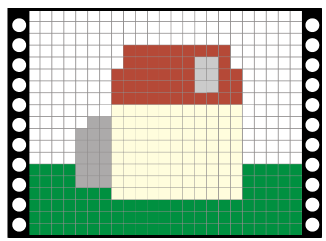
- Project vertices to 2D (image)

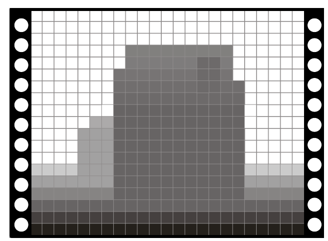


- Rasterize triangle: find which pixels should be lit

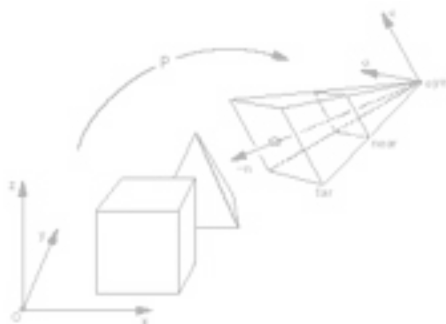


- Compute per-pixel color
- Test visibility (Z-buffer), update frame buffer

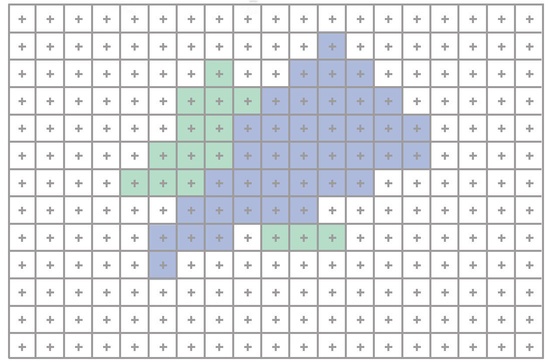




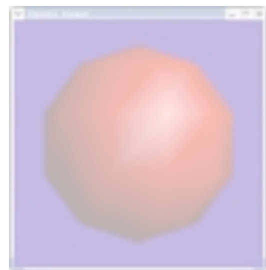
Modern Graphics Pipeline



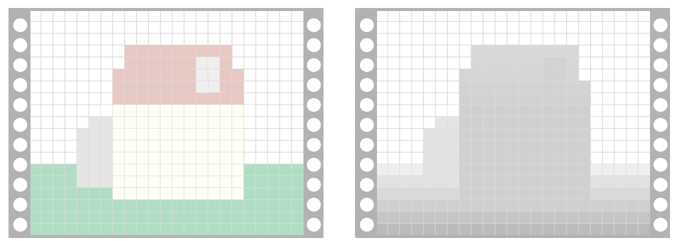
- Project vertices to 2D (image)
  - We now have screen coordinates



- Rasterize triangle: find which pixels should be lit



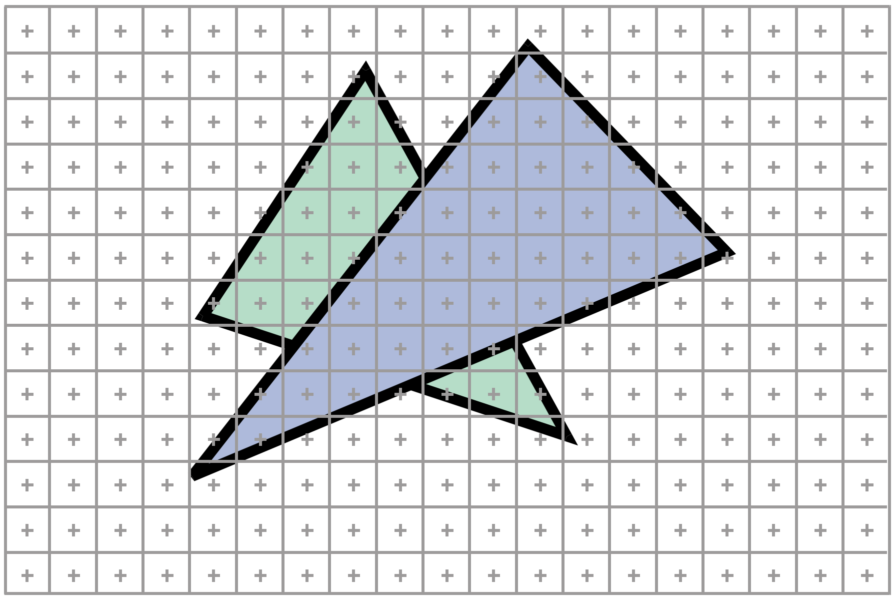
- Compute per-pixel color



- Test visibility (Z-buffer), update frame buffer

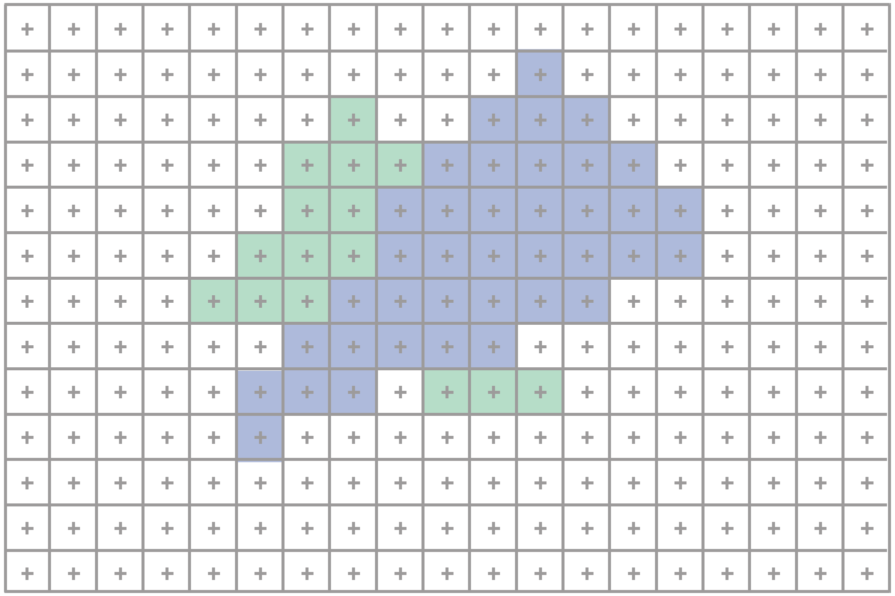
2D Scan Conversion

- Primitives are “continuous” geometric objects; screen is discrete (pixels)



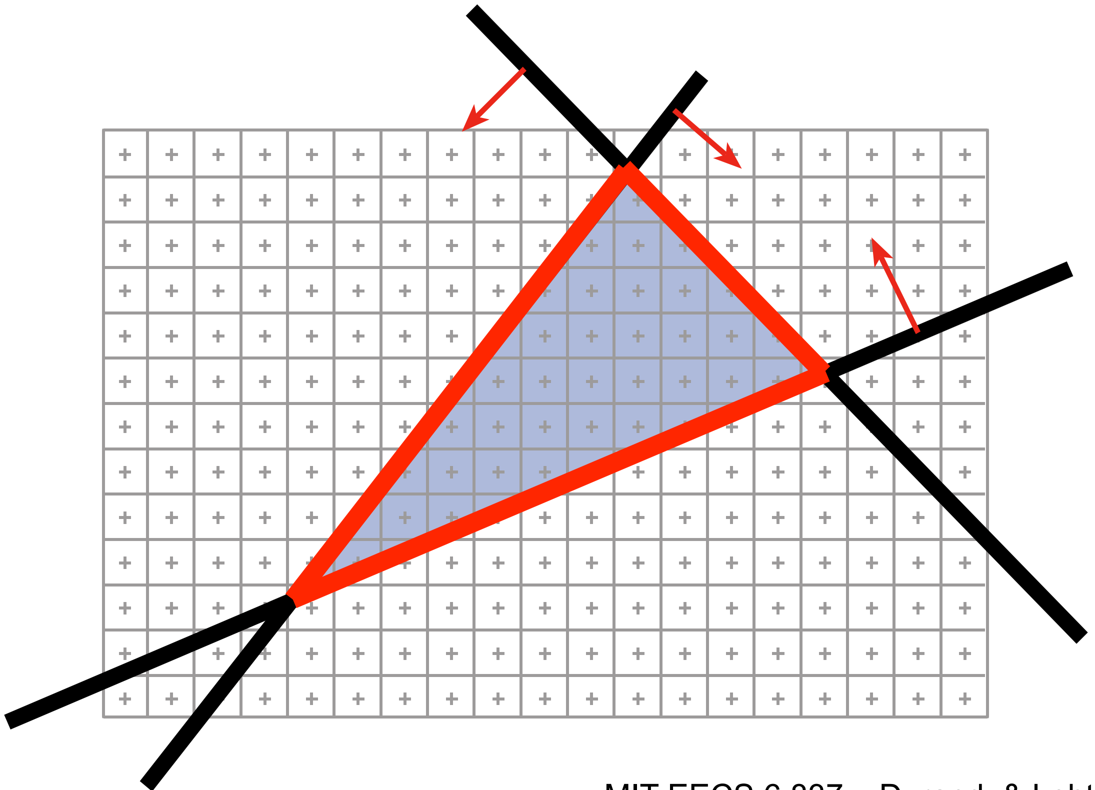
2D Scan Conversion

- Primitives are "continuous" geometric objects; screen is discrete (pixels)
- Rasterization computes a discrete approximation in terms of pixels **(how?)**



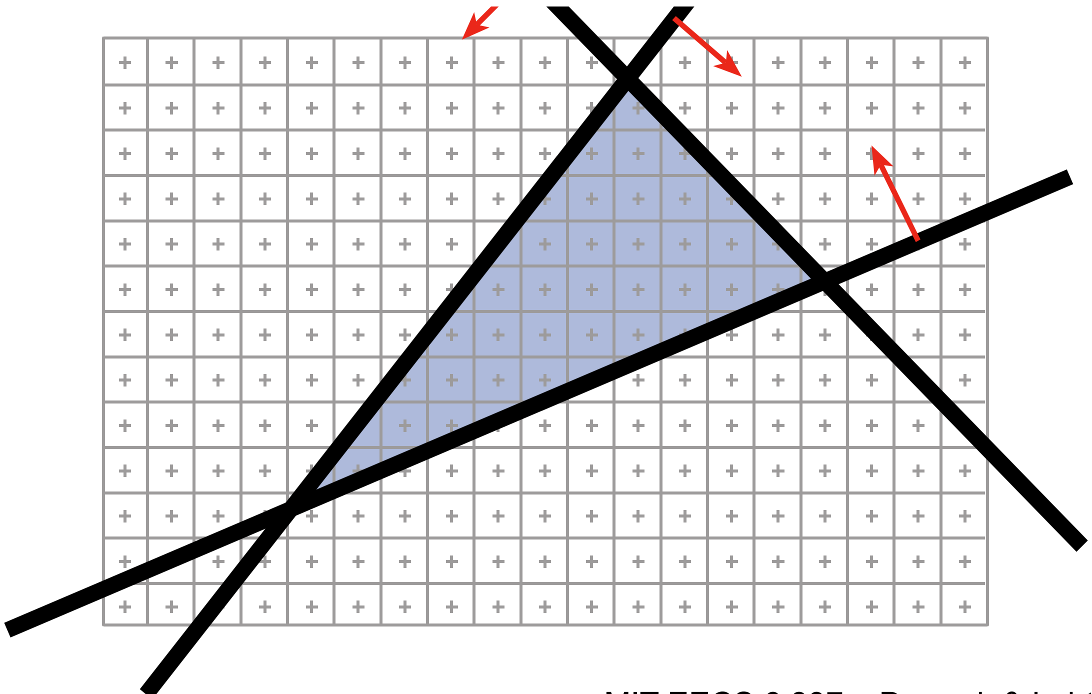
Edge Functions

- The triangle's 3D edges project to line segments in the image (thanks to planar perspective)
  - Lines map to lines, not curves



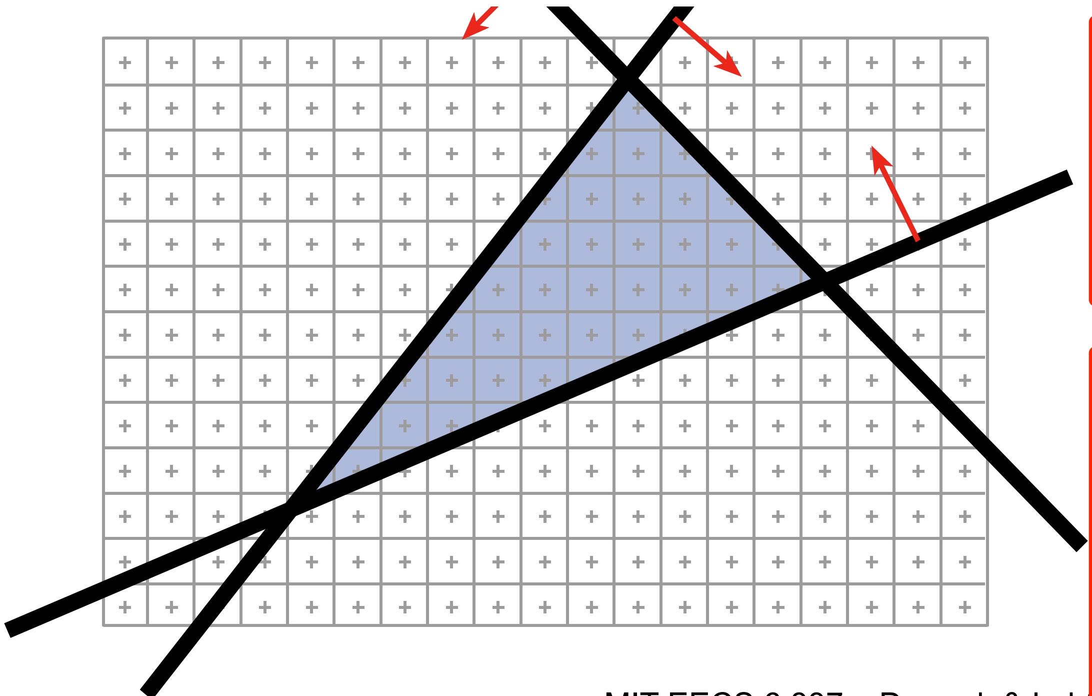
Edge Functions

- The triangle's 3D edges project to line segments in the image (thanks to planar perspective)
- The interior of the triangle is the set of points that is inside all three halfspaces defined by these lines



Edge Functions

- The triangle's 3D edges project to line segments in the image (thanks to planar perspective)
- The interior of the triangle is the set of points that is inside all three halfspaces defined by these lines

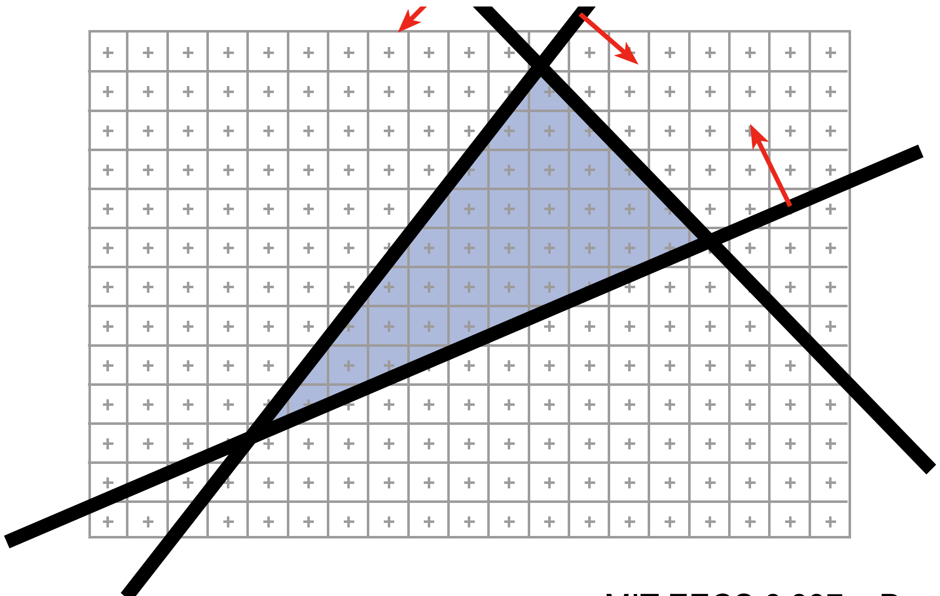


$$E_i(x, y) = a_i x + b_i y + c_i$$

$$(x, y) \text{ within triangle} \Leftrightarrow E_i(x, y) \geq 0, \quad \forall i = 1, 2, 3$$

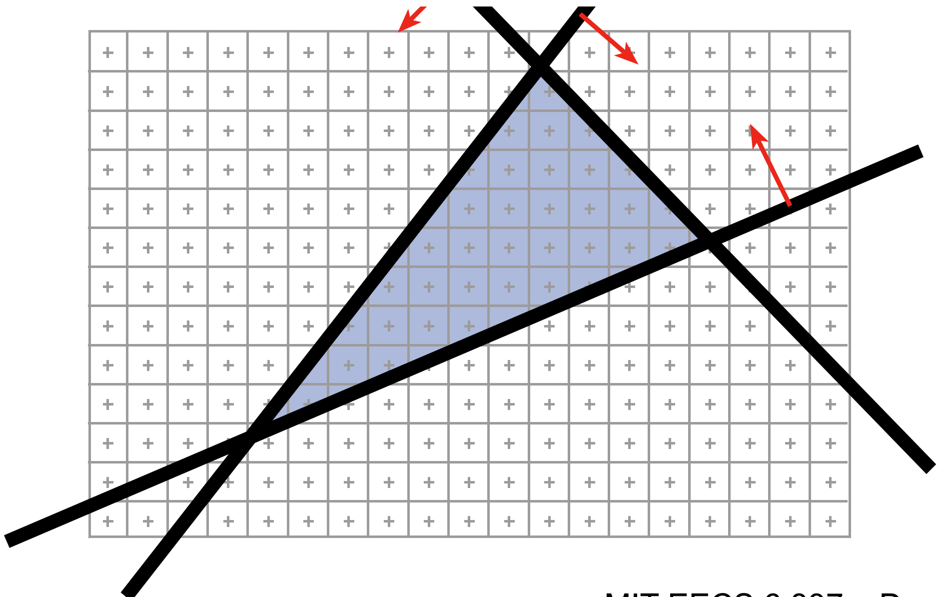
Brute Force Rasterizer

- Compute E_1, E_1 , E_3 coefficients from projected vertices
  - Called “triangle setup”, yields a_i, b_i, c_i for i=1,2,3



Brute Force Rasterizer

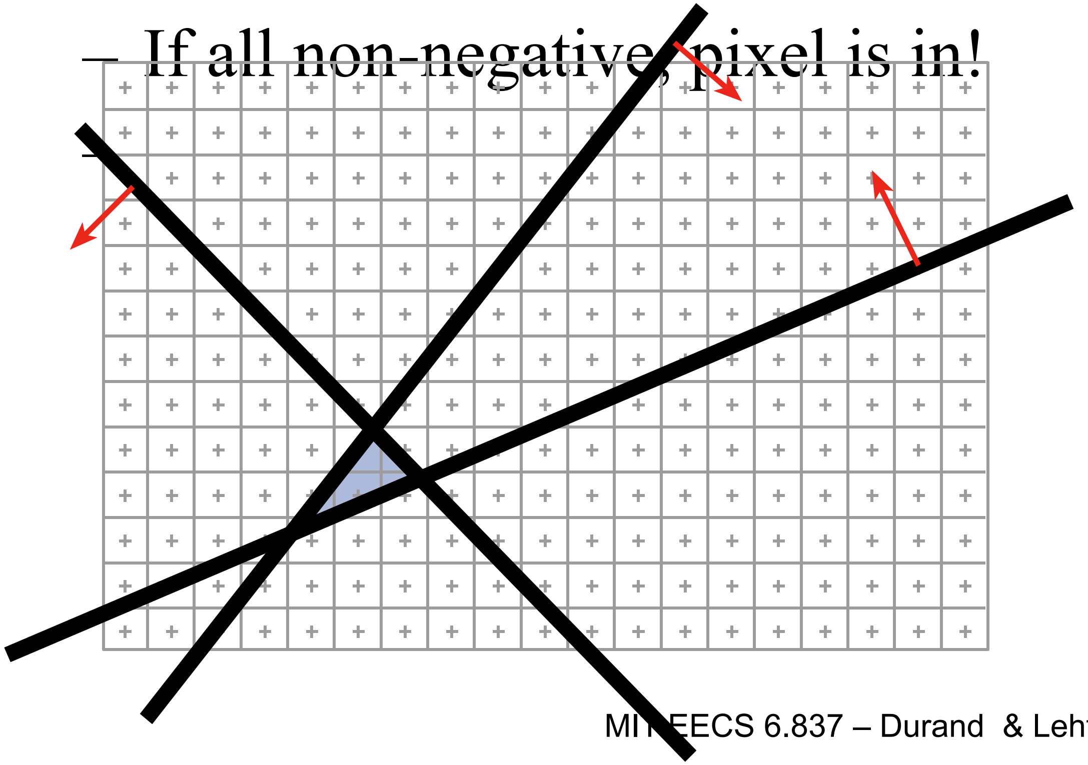
- Compute E_1, E_1 , E_3 coefficients from projected vertices
- For each pixel (x, y)
  - Evaluate edge functions at pixel center
  - If all non-negative, pixel is in!



Problem?

Brute Force Rasterizer

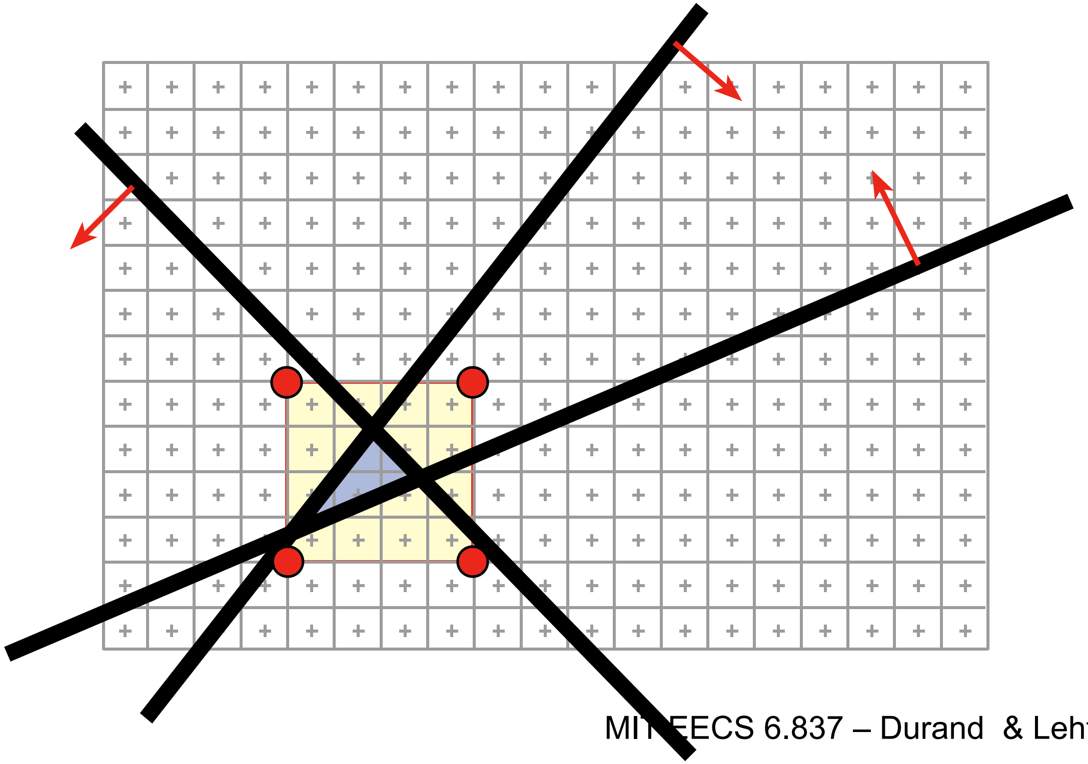
- Compute E_1, E_1 , E_3 coefficients from projected vertices
- For each pixel (x, y)
  - Evaluate edge functions at pixel center
  - If all non-negative, pixel is in!



If the triangle is small, lots of useless computation if we really test all pixels

Easy Optimization

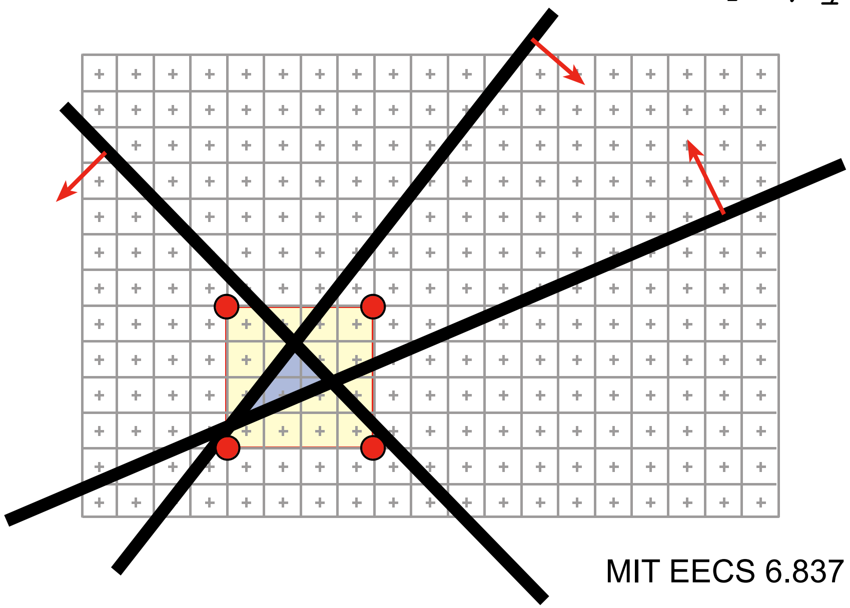
- Improvement: Scan over only the pixels that overlap the *screen bounding box* of the triangle
- How do we get such a bounding box?
  - X_{min}, X_{max}, Y_{min}, Y_{max} of the projected triangle vertices



Rasterization Pseudocode

Note: No visibility

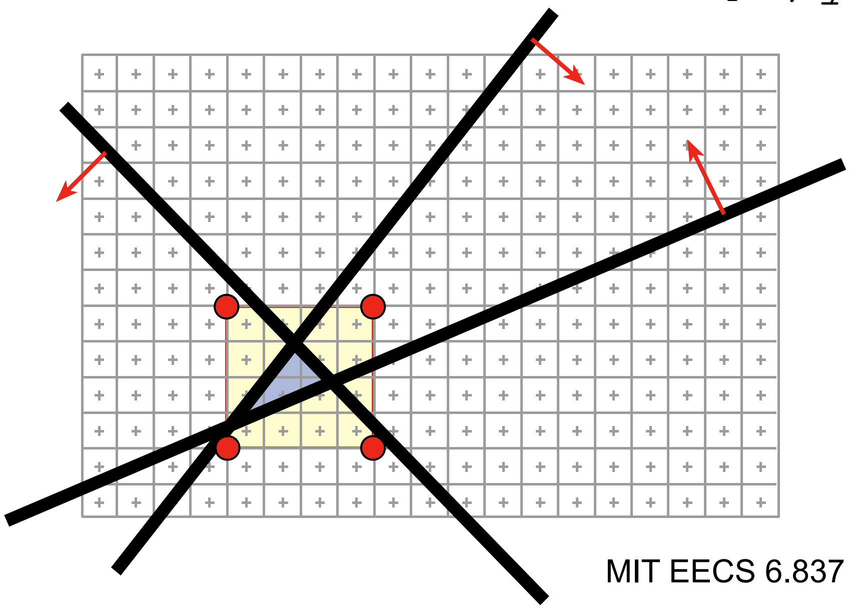
```
For every triangle
   Compute projection for vertices, compute the Ei
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Evaluate edge functions Ei
      If all > 0
          Framebuffer[x,y ] = triangleColor
```



Bounding box clipping is easy, just clamp the coordinates to the screen rectangle

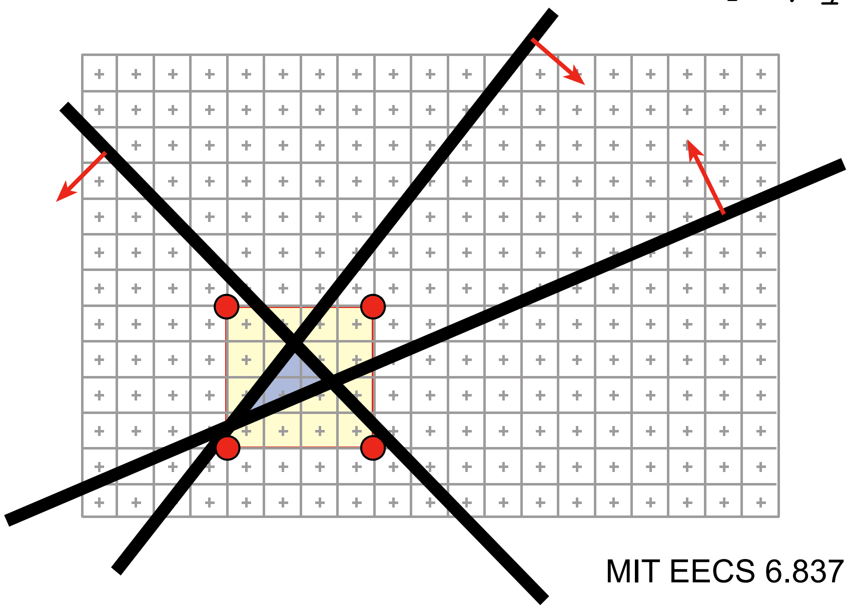
Can We Do Better?

```
For every triangle
   Compute projection for vertices, compute the Ei
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Evaluate edge functions aix + biy + ci
      If all > 0
         Framebuffer[x,y ] = triangleColor
```



Can We Do Better?

```
For every triangle
   Compute projection for vertices, compute the Ei
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Evaluate edge functions aix + biy + ci
      If all >0
          Framebuffer[x,y ] = triangleColor
```



These are linear functions of the pixel coordinates (x,y), i.e., they only change by a constant amount when we step from x to x+1 (resp. y to y+1)

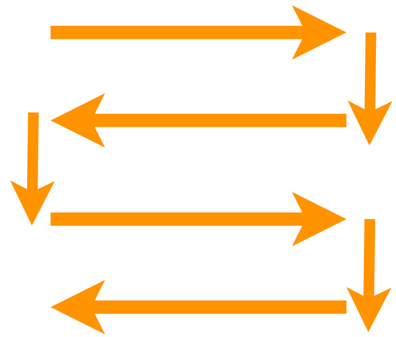
Incremental Edge Functions

```
For every triangle
   ComputeProjection
   Compute bbox, clip bbox to screen limits
   For all scanlines y in bbox
      Evaluate all Ei's at (x0,y): Ei = aix0 + biy + ci
      For all pixels x in bbox
         If all Ei>0
            Framebuffer[x,y ] = triangleColor
         Increment line equations: Ei += ai
```

- We save ~two multiplications and two additions per pixel when the triangle is large

Incremental Edge Functions

```
For every triangle
    ComputeProjection
    Compute bbox, clip bbox to screen limits
    For all scanlines y in bbox
        Evaluate all E_i's at (x0,y): E_i = a_i x0 + b_i y + c_i
        For all pixels x in bbox
            If all E_i>0
                Framebuffer[x,y ] = triangleColor
            Increment line equations: E_i += a_i
```

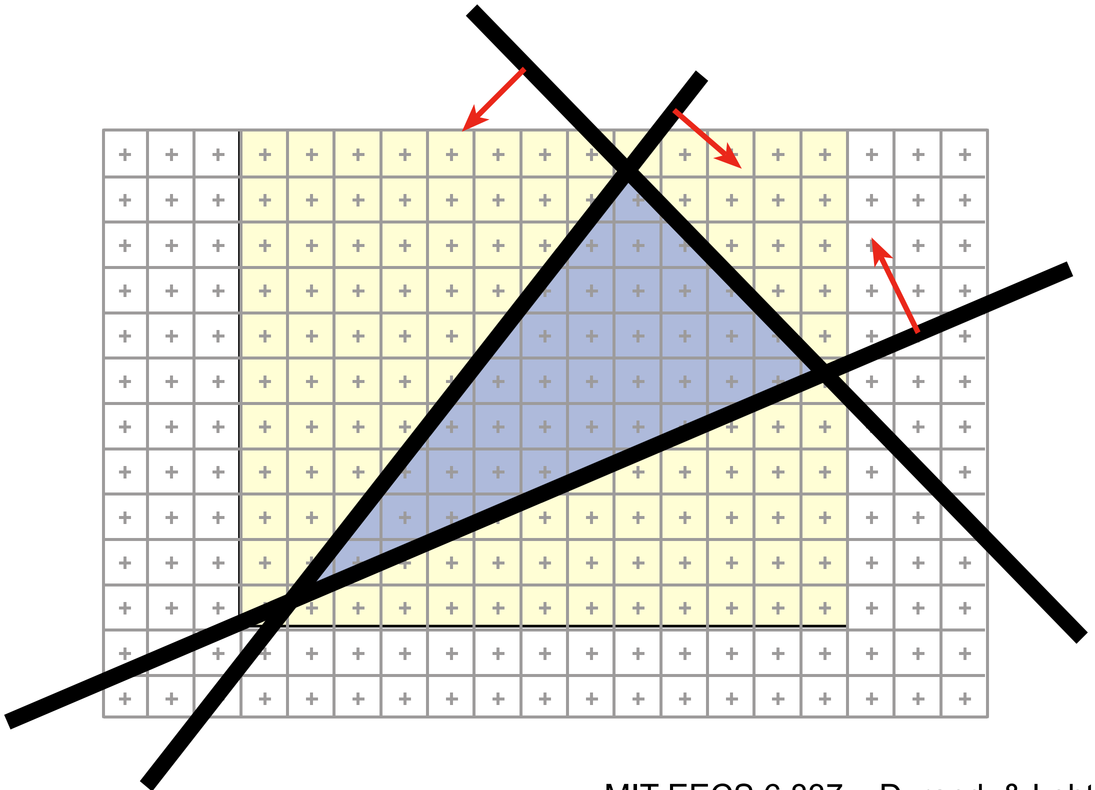


- We save ~two multiplications and two additions per pixel when the triangle is large

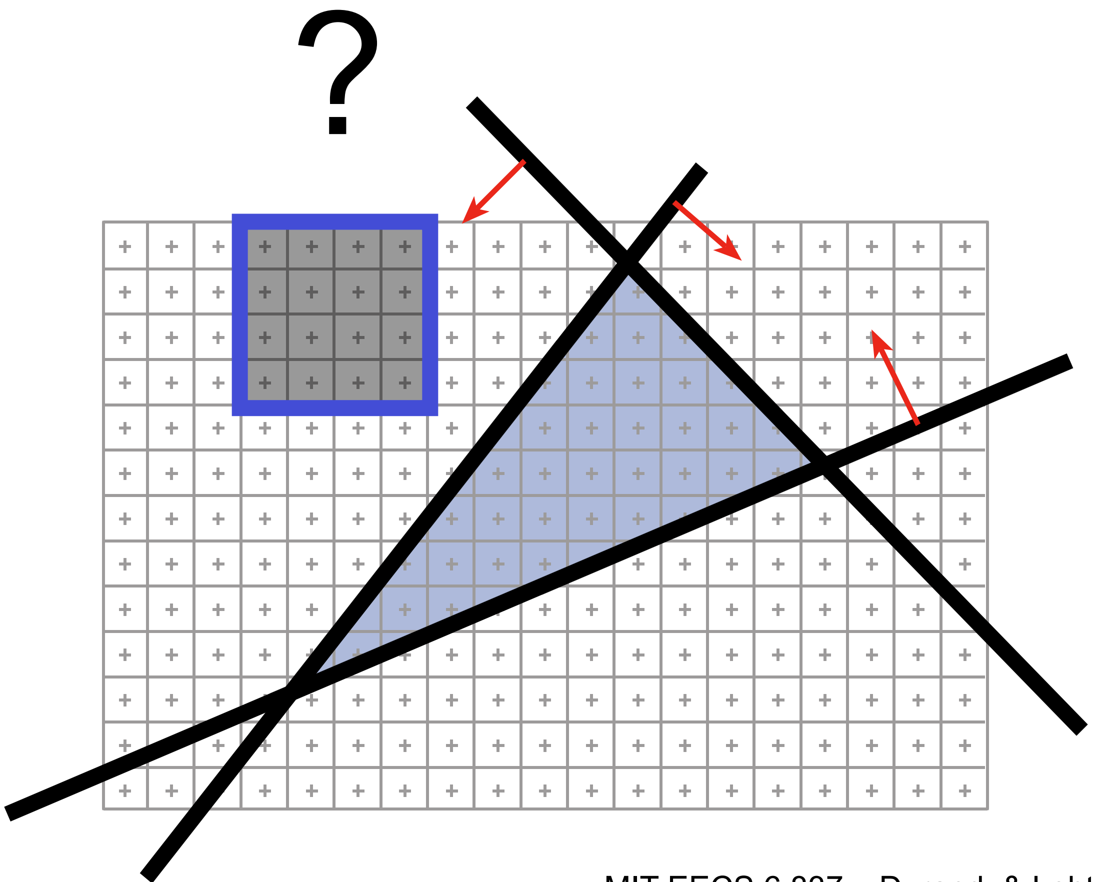
Can also zig-zag to avoid reinitialization per scanline, just initialize once at x0, y0

Can We Do Even Better?

- We compute the line equation for many useless pixels
- What could we do?

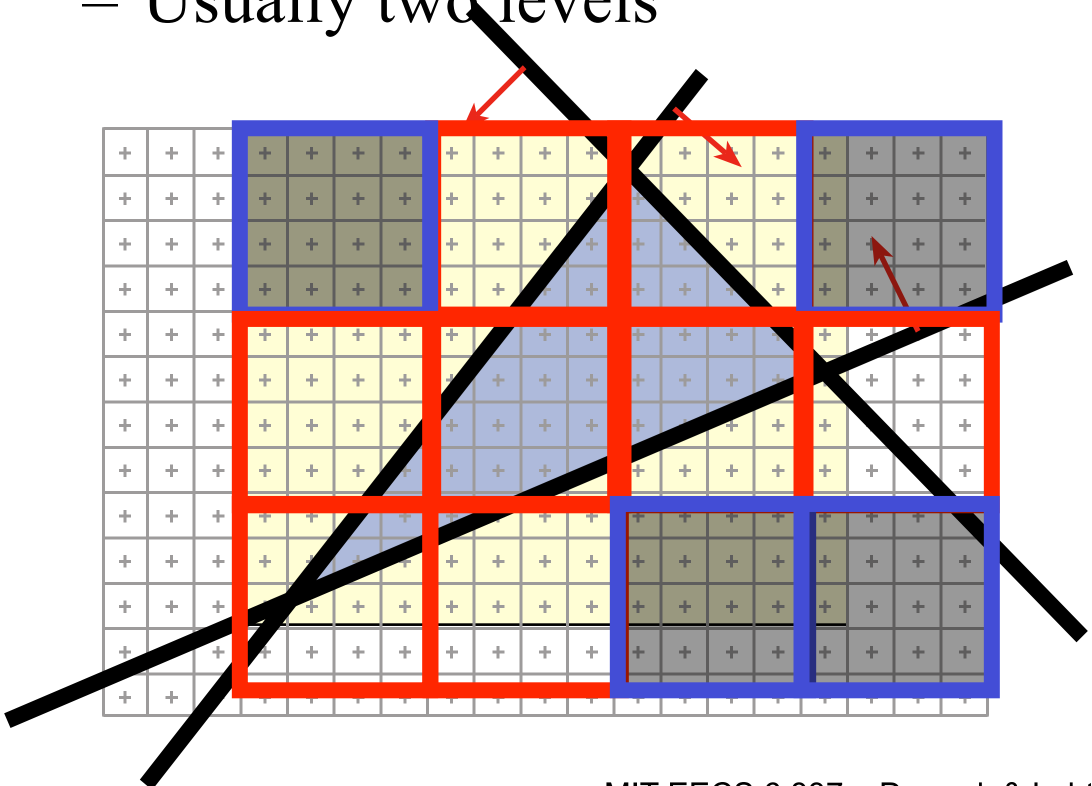


Indeed, We Can Be Smarter



Indeed, We Can Be Smarter

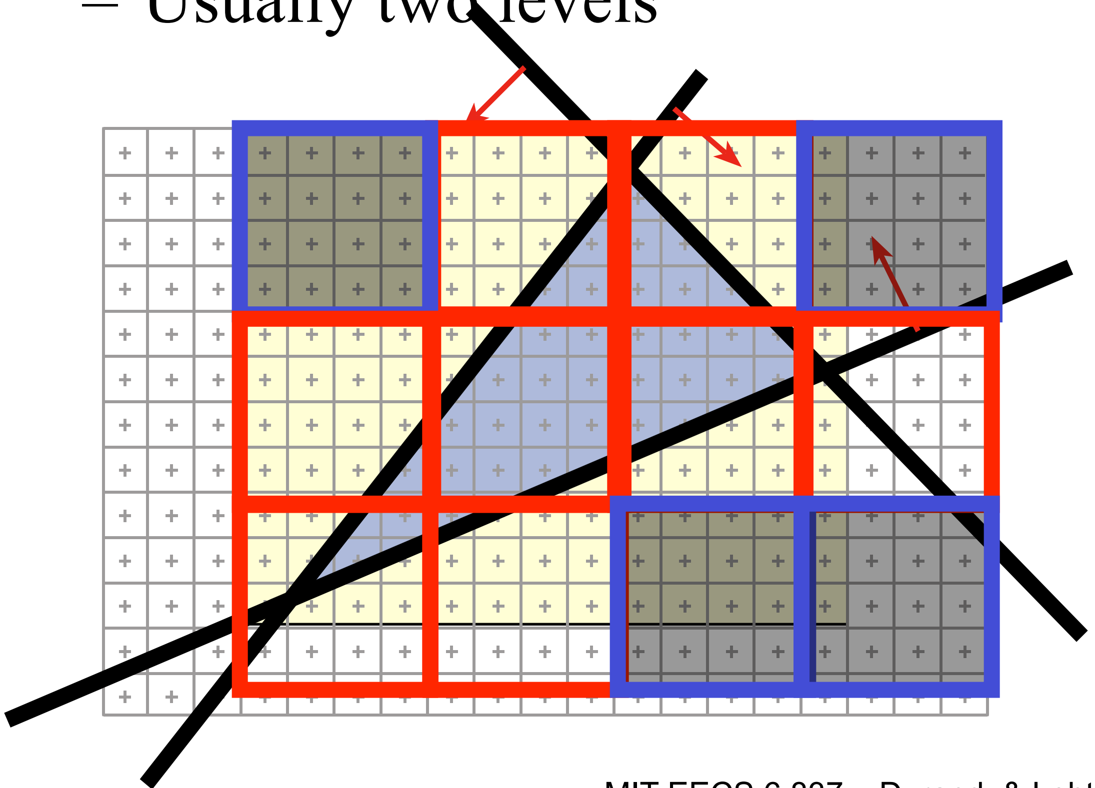
- Hierarchical rasterization!
  - Conservatively test **blocks of pixels** before going to per-pixel level (can skip large blocks at once)
  - Usually two levels



Conservative tests of axis-aligned blocks vs. edge functions are not very hard, thanks to linearity. See Akenine-Möller and Aila, Journal of Graphics Tools 10 (3), 2005.

Indeed, We Can Be Smarter

- Hierarchical rasterization!
  - Conservatively test **blocks of pixels** before going to per-pixel level (can skip large blocks at once)
  - Usually two levels



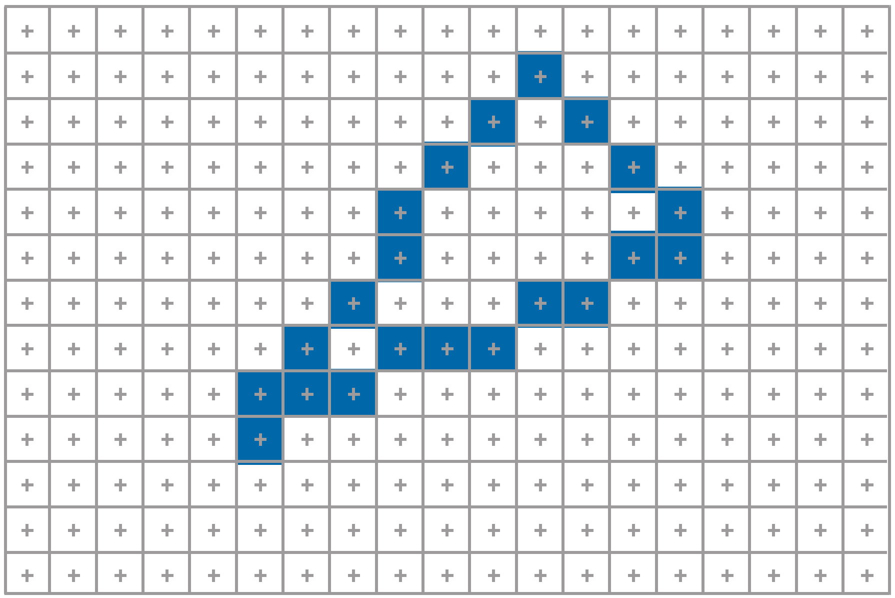
Can also test if an entire block is **inside** the triangle; then, can skip edge functions tests for all pixels for even further speedups. (Must still test Z, because they might still be occluded.)

Further References

- Henry Fuchs, Jack Goldfeather, Jeff Hultquist, Susan Spach, John Austin, Frederick Brooks, Jr., John Eyles and John Poulton, "Fast Spheres, Shadows, Textures, Transparencies, and Image Enhancements in Pixel-Planes", Proceedings of SIGGRAPH '85 (San Francisco, CA, July 22–26, 1985). In *Computer Graphics*, v19n3 (July 1985), ACM SIGGRAPH, New York, NY, 1985.
- Juan Pineda, "A Parallel Algorithm for Polygon Rasterization", Proceedings of SIGGRAPH '88 (Atlanta, GA, August 1–5, 1988). In *Computer Graphics*, v22n4 (August 1988), ACM SIGGRAPH, New York, NY, 1988. Figure 7: Image from the spinning teapot performance test.
- Marc Olano Trey Greer, "Triangle Scan Conversion using 2D Homogeneous Coordinates", Graphics Hardware 97 http://www.cs.unc.edu/~olano/papers/2dh-tri/2dh-tri.pdf

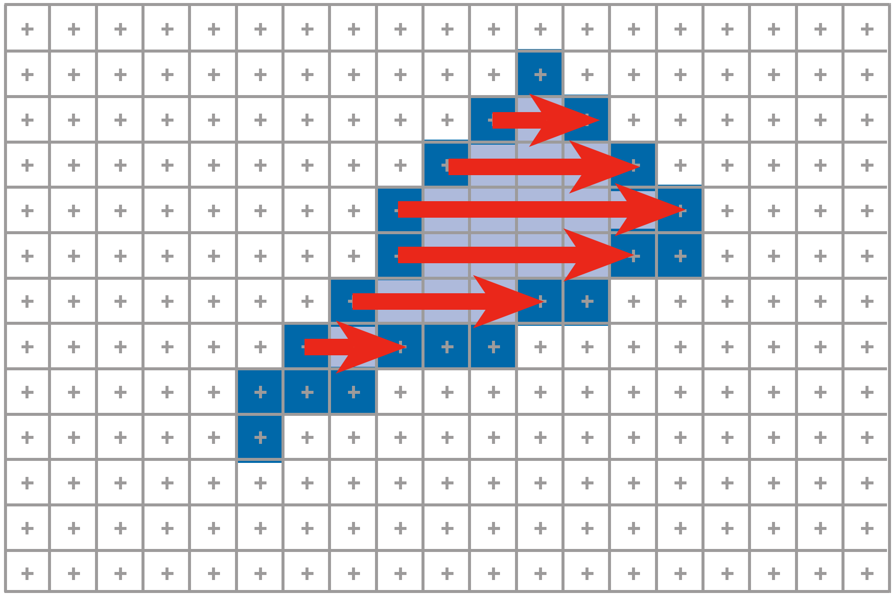
Oldskool Rasterization

- Compute the boundary pixels using line rasterization



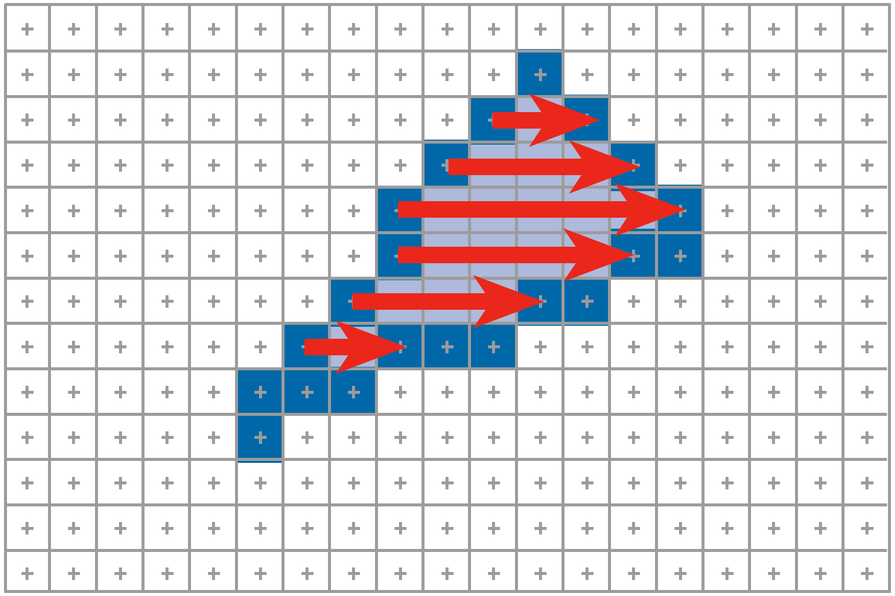
Oldskool Rasterization

- Compute the boundary pixels using line rasterization
- Fill the spans



Oldskool Rasterization

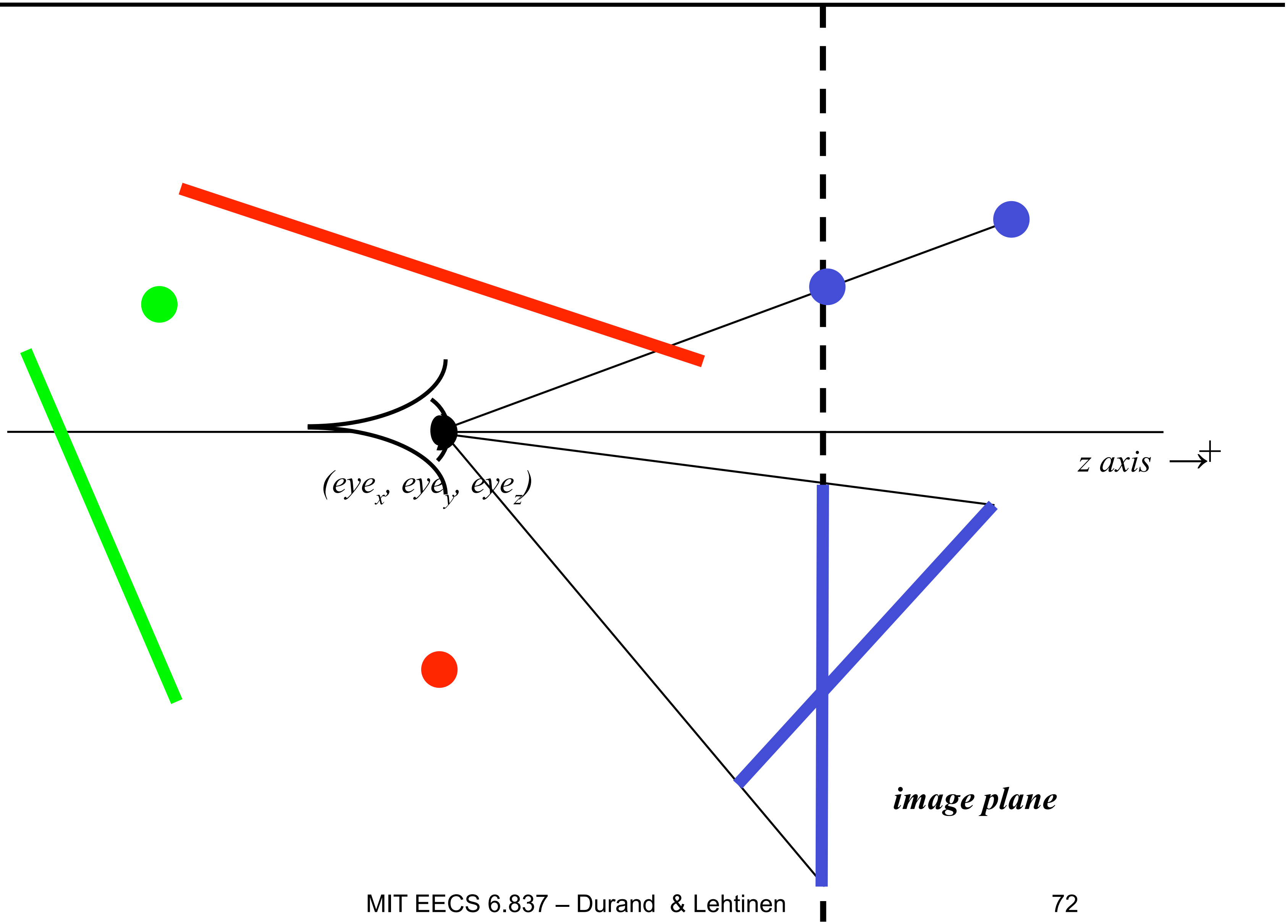
- Compute the boundary pixels using line rasterization
- Fill the spans



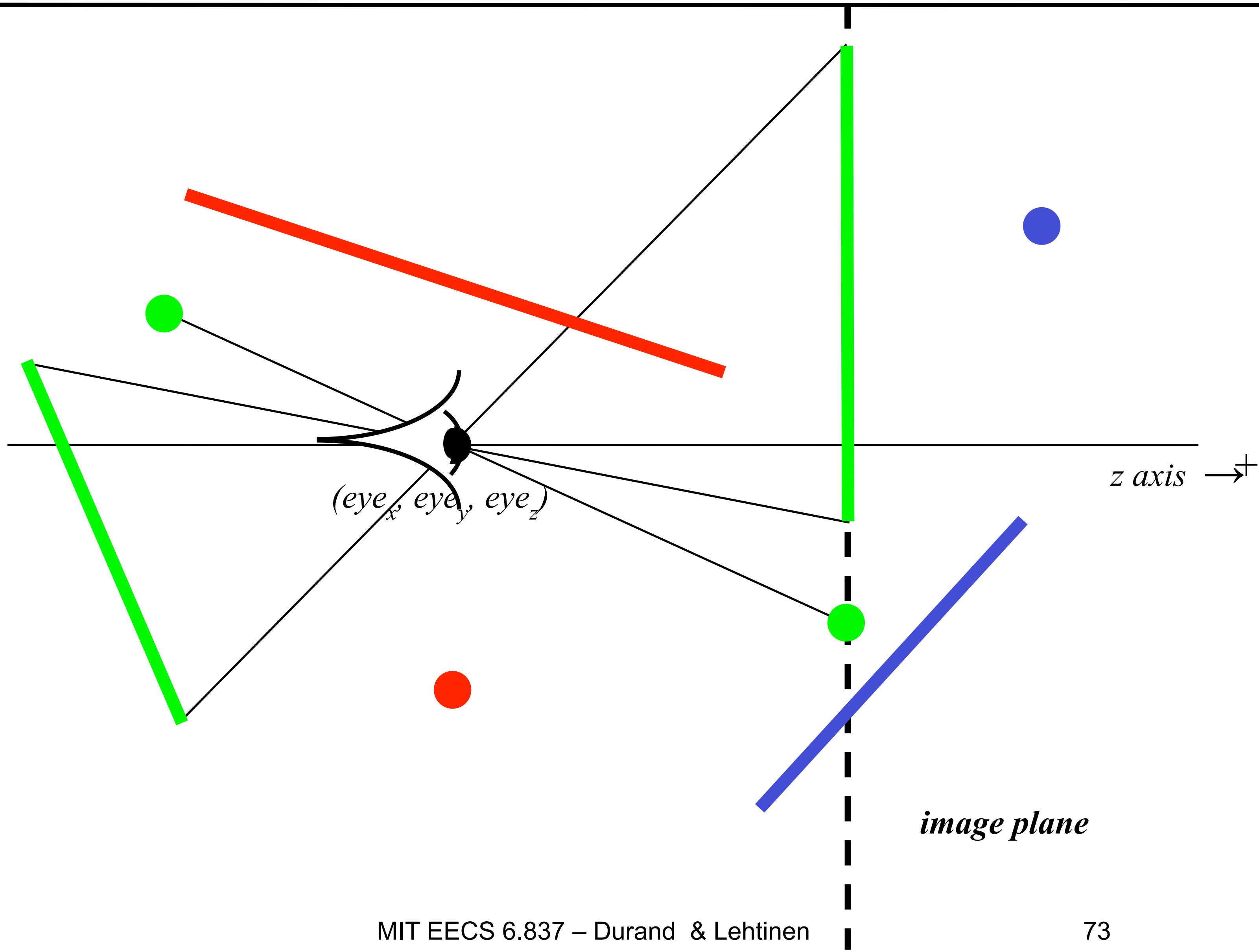
More annoying to implement than edge functions

Not faster unless triangles are huge

What if the p_z is > eye_z?

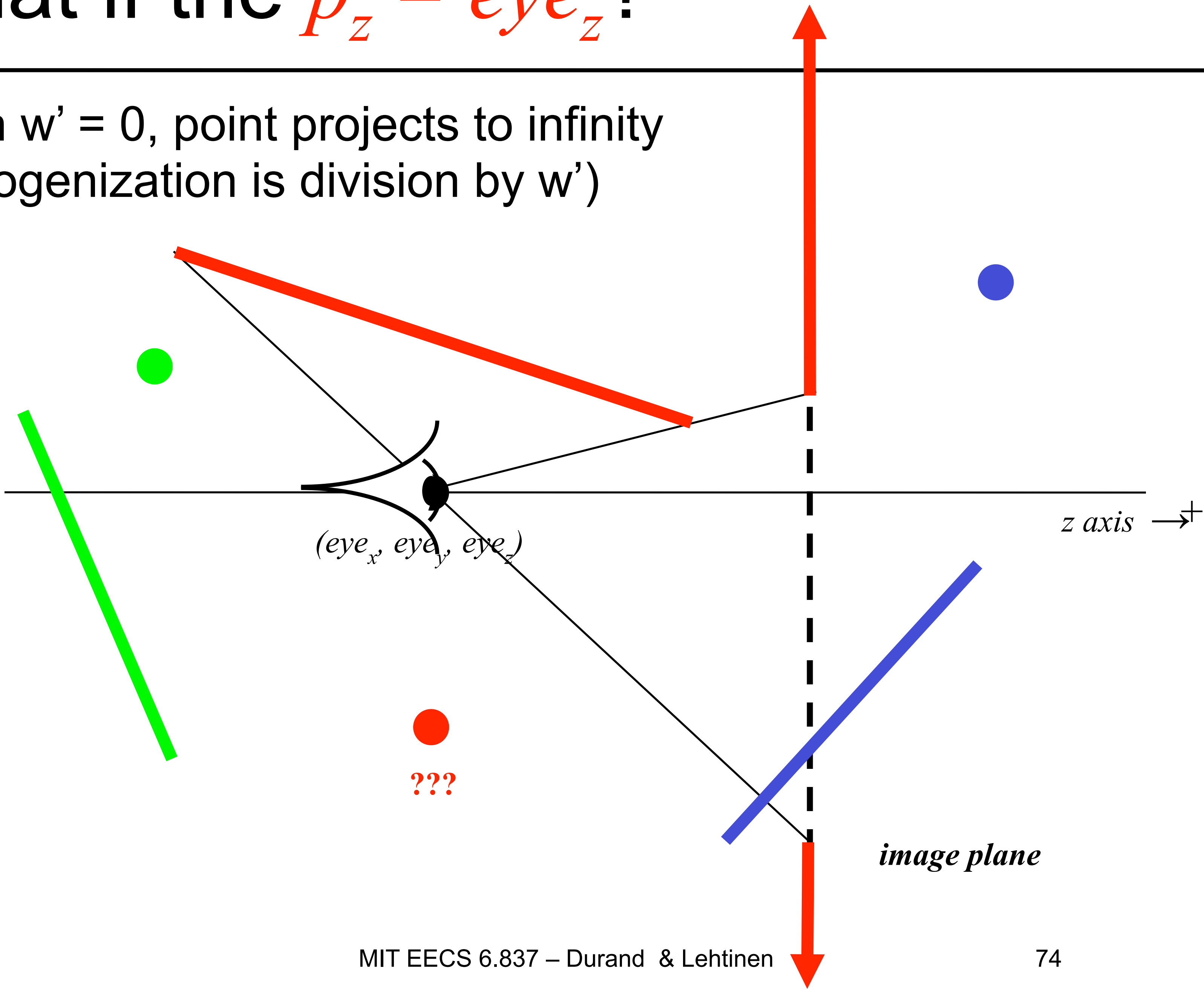


What if the p_z is < eye_z?

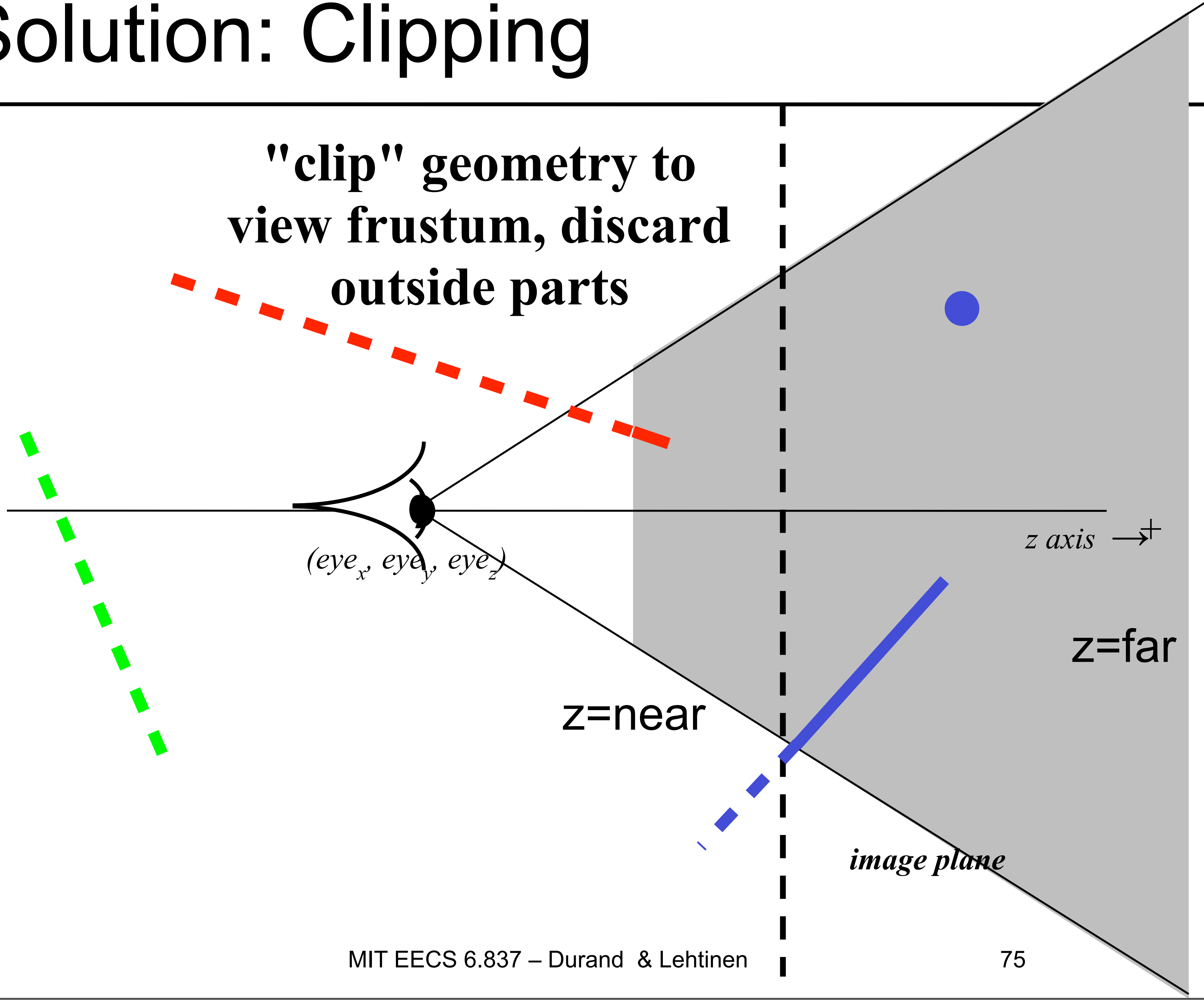


What if the $p_z = eye_z$?

When w' = 0, point projects to infinity
(homogenization is division by w')

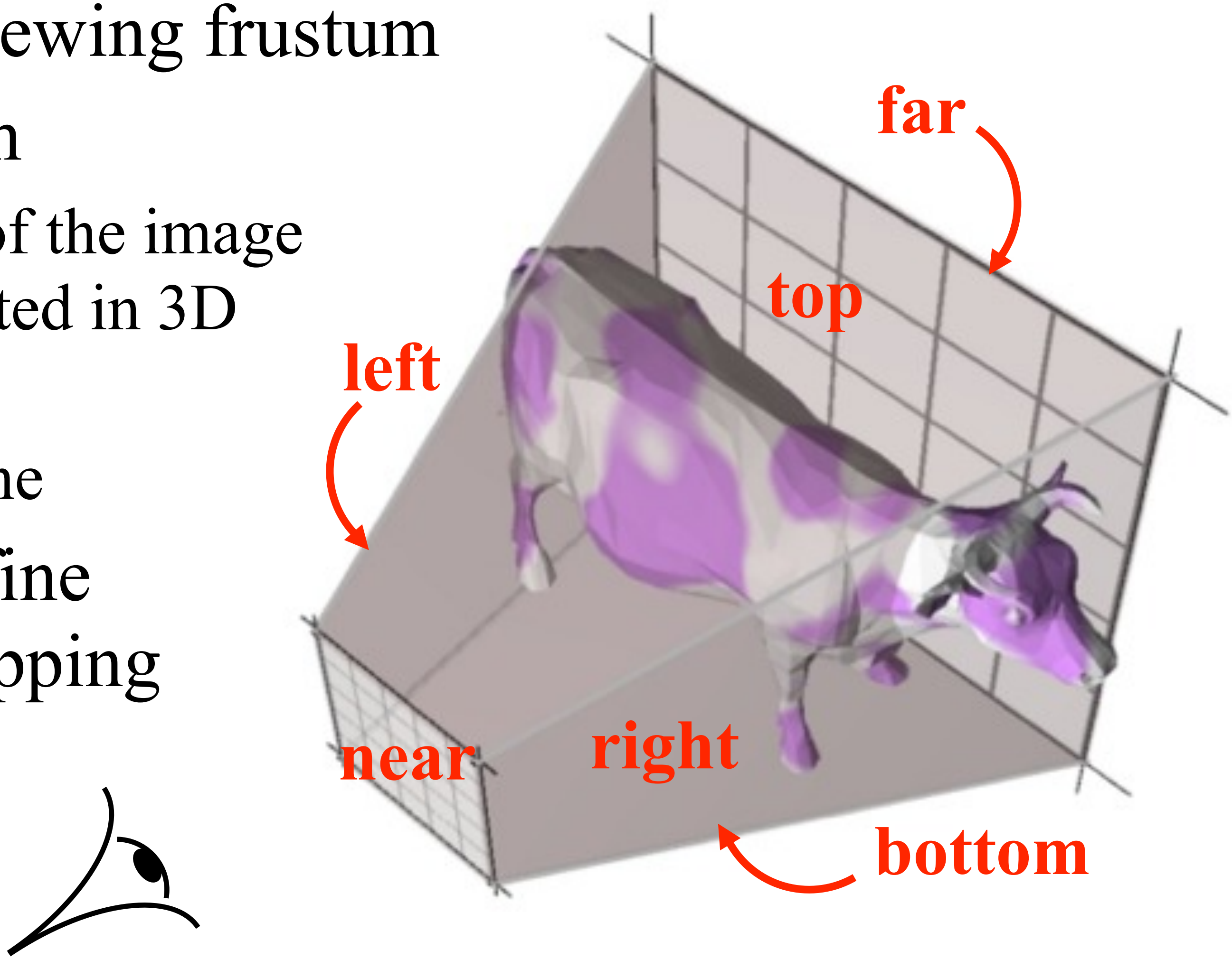


A Solution: Clipping



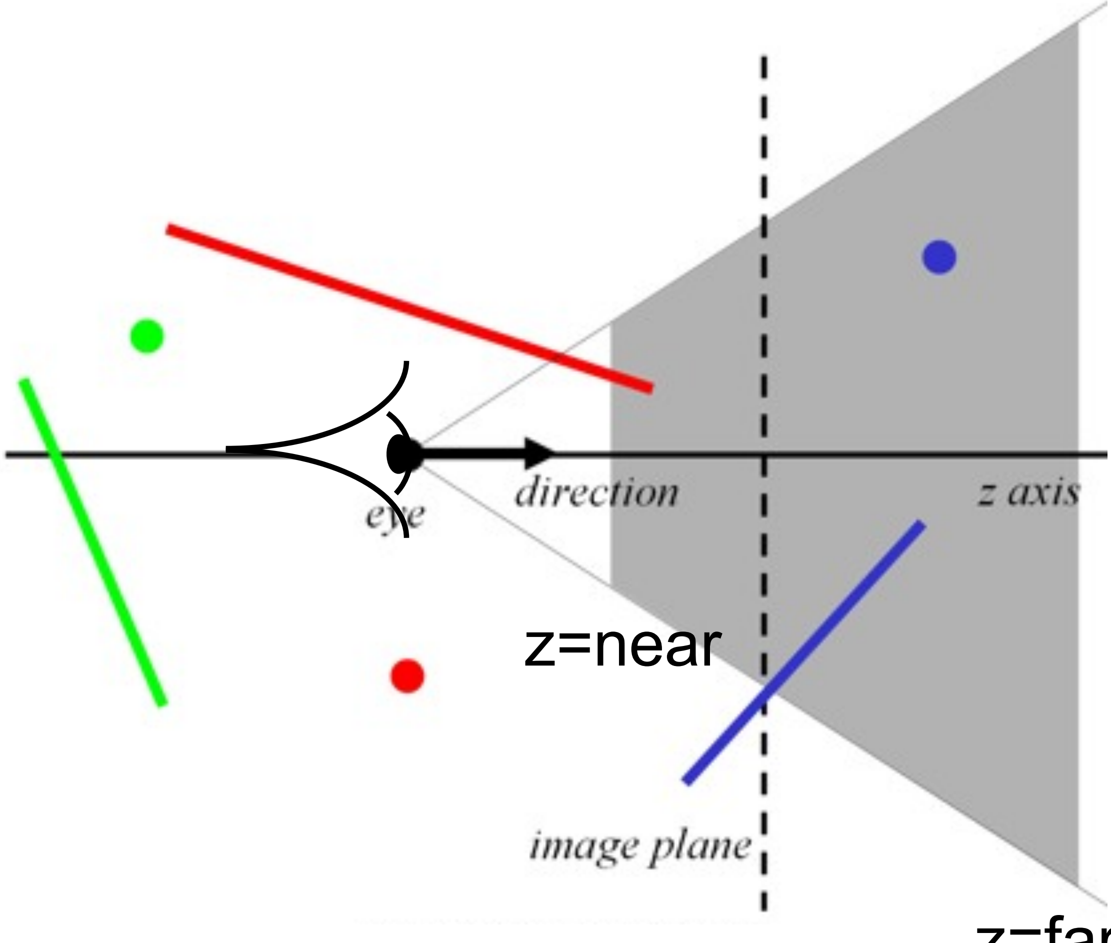
Clipping

- Eliminate portions of objects outside the viewing frustum
- View Frustum
  - boundaries of the image plane projected in 3D
  - a near & far clipping plane
- User may define additional clipping planes



Why Clip?

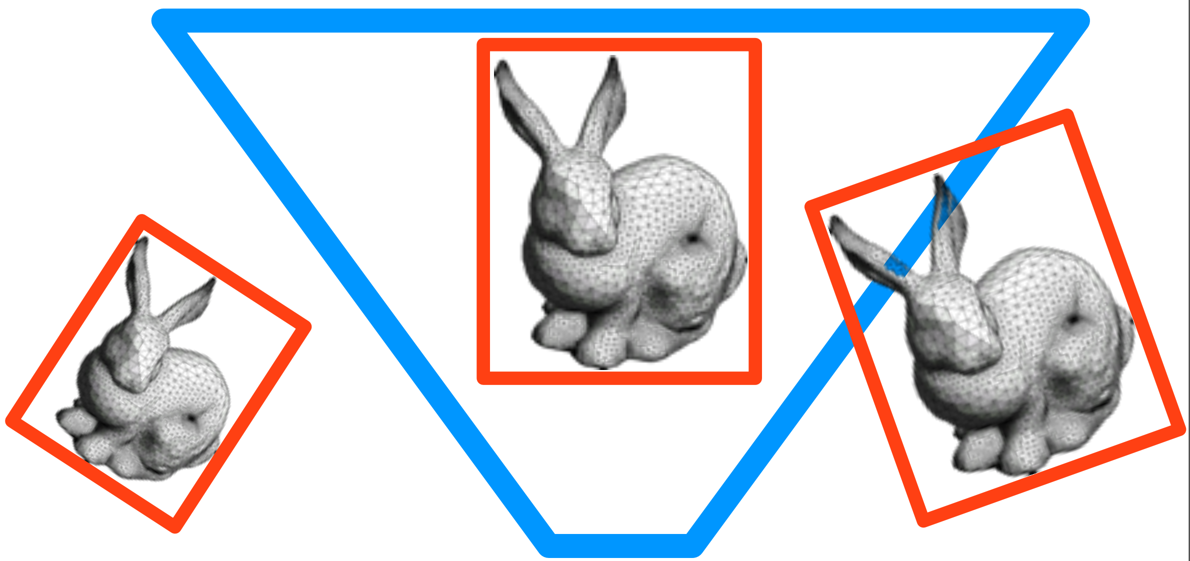
- Avoid degeneracies
  - Don't draw stuff behind the eye
  - Avoid division by 0 and overflow



Related Idea

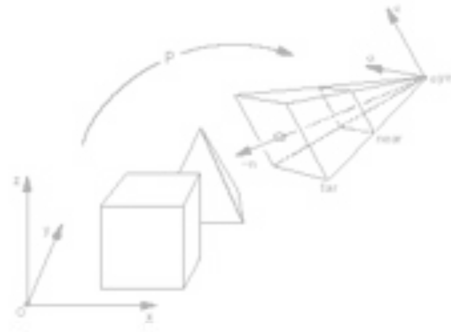
- "View Frustum Culling"
  - Use bounding volumes/hierarchies to test whether any part of an object is within the view frustum
    - Need "frustum vs. bounding volume" intersection test
    - Crucial to do hierarchically when scene has *lots* of objects!
    - Early rejection (different from clipping)

See e.g. Optimized view frustum culling algorithms for bounding boxes, Ulf Assarsson and Tomas Möller, journal of graphics tools, 2000.



Modern Graphics Pipeline

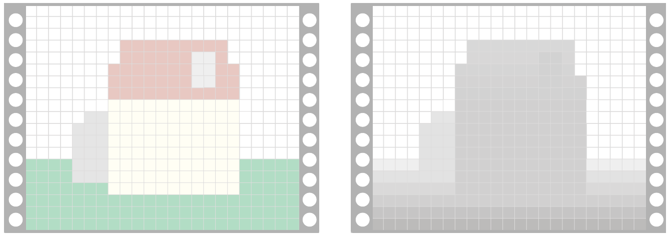
- Perform projection of vertices



- Rasterize triangle: find which pixels should be lit

- Compute per-pixel color

- Test visibility, update frame buffer

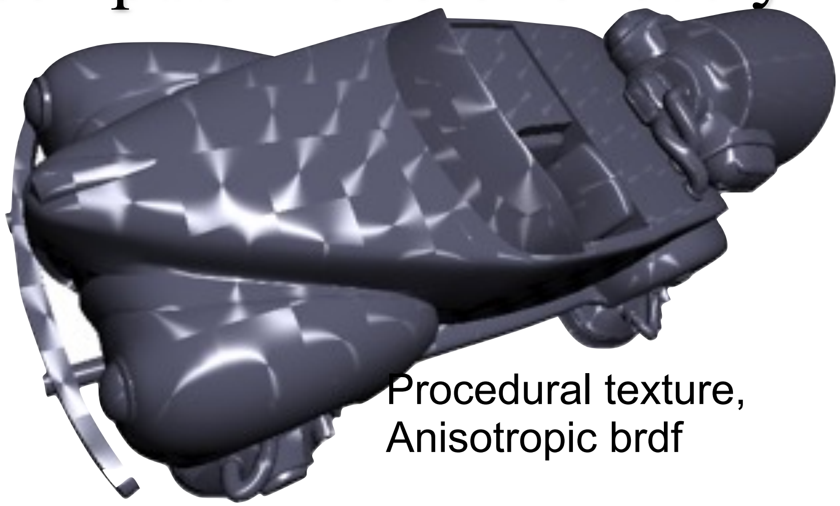


Pixel Shaders

- Modern graphics hardware enables the execution of rather complex programs to compute the color of every single pixel
  - More later

Translucence

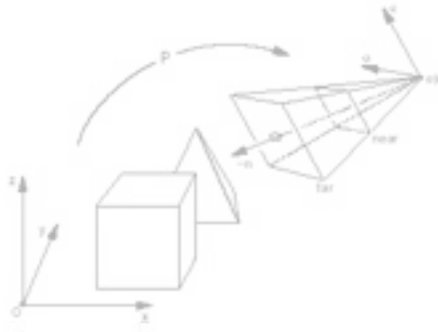
Backlighting



Procedural texture,
Anisotropic brdf

iridescence

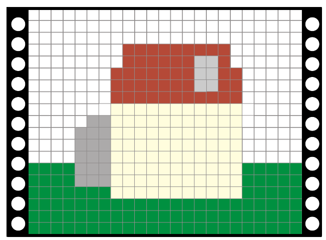
Modern Graphics Pipeline

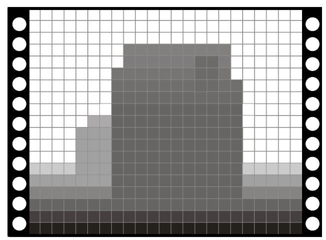


- Perform projection of vertices

- Rasterize triangle: find which pixels should be lit

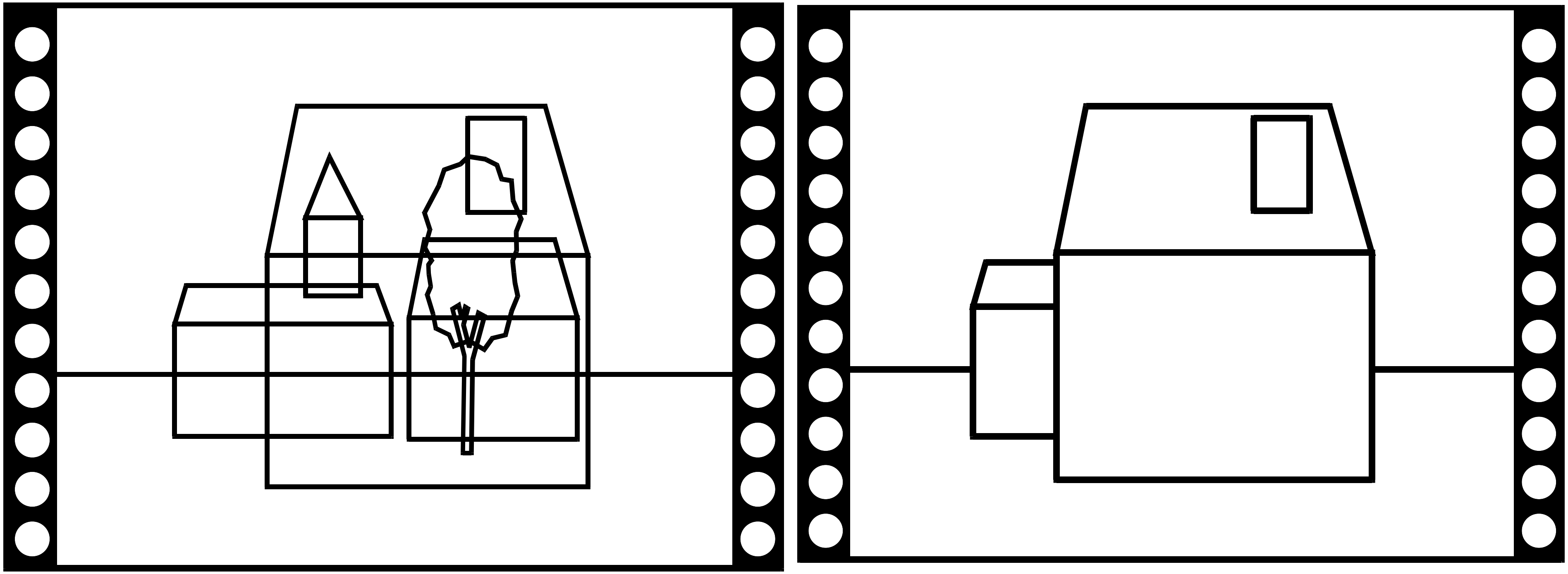
- Compute per-pixel color
- Test visibility, update frame buffer





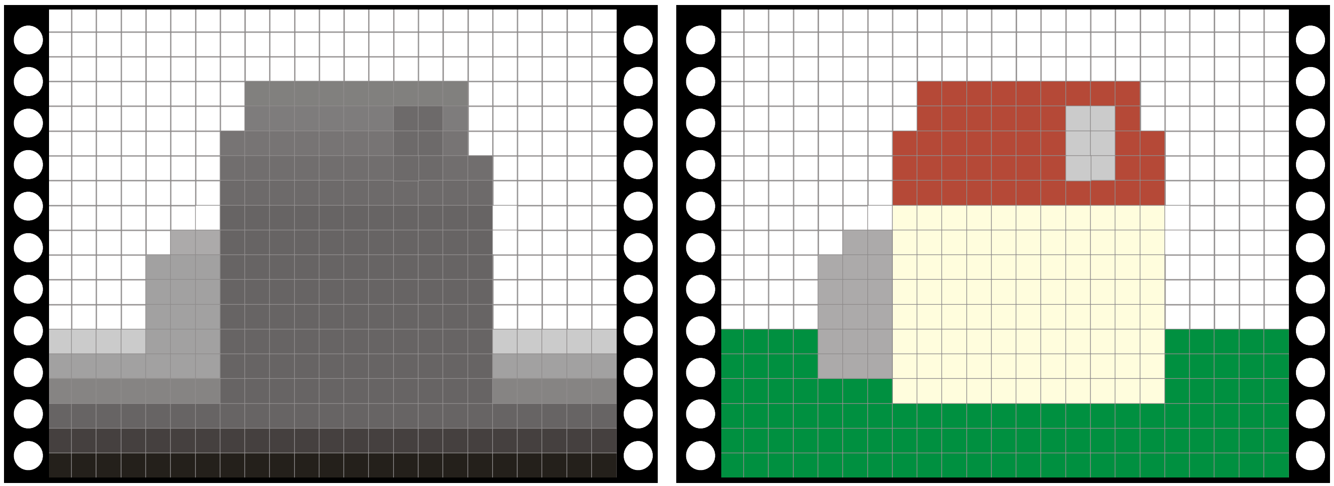
Visibility

- How do we know which parts are visible/in front?



Z buffer

- In addition to frame buffer (R, G, B)
- Store distance to camera (z-buffer)
- Pixel is updated only if new z is closer than z-buffer value



Z-buffer pseudo code

```
For every triangle
   Compute Projection, color at vertices
   Setup line equations
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Increment line equations
      Compute curentZ
      Increment currentColor
      If all line equations>0  //pixel [x,y] in triangle
         If currentZ<zBuffer[x,y]  //pixel is visible
               Framebuffer[x,y]=currentColor
            zBuffer[x,y]=currentZ
```

Works for hard cases!