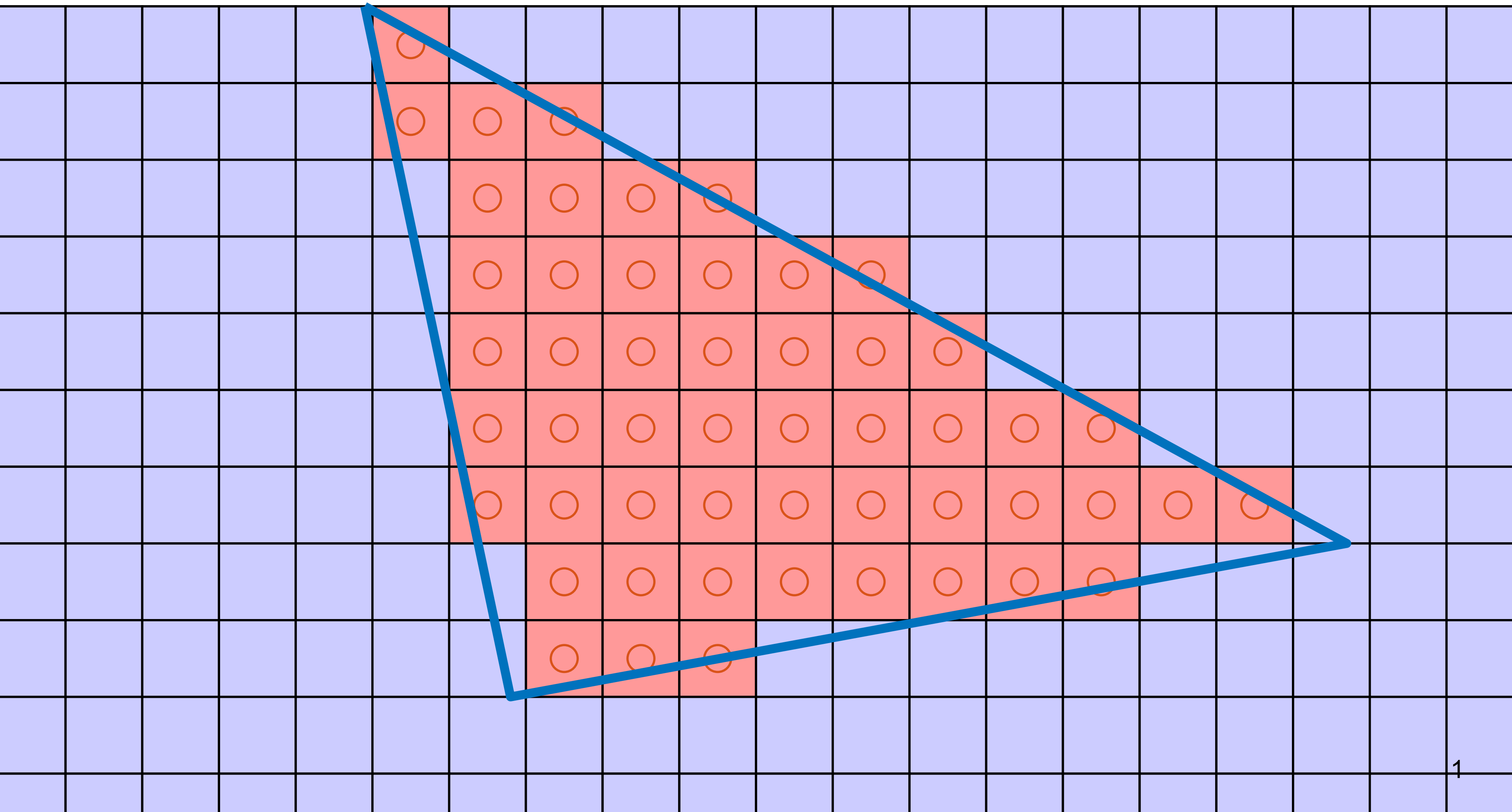
Rasterization & The Graphics Pipeline

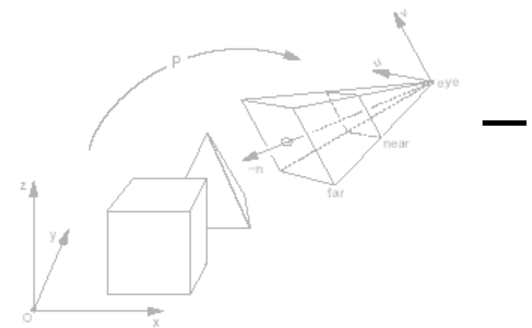
15.3 Rasterization using edge functions



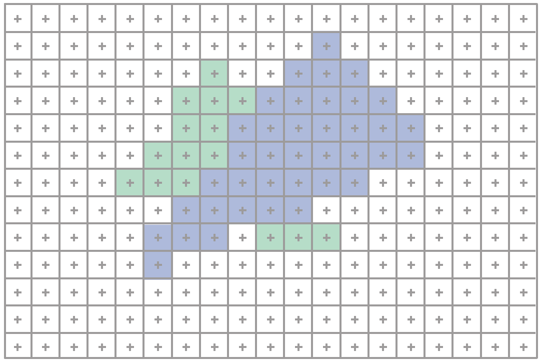
In This Video

- Edge functions: finding out which pixels are covered
  - and various optimisations
- Extra: Projection matrices
  - From 3D to 2D via homogeneous coordinates

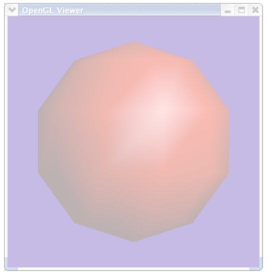
The Graphics Pipeline



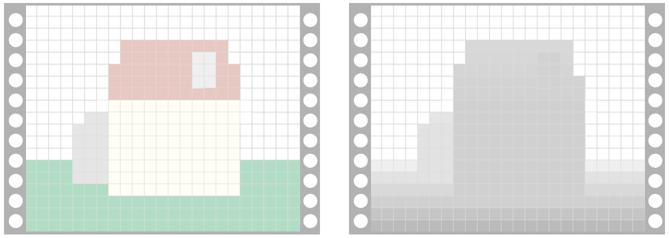
- Project vertices to 2D (image)
  - We now have screen coordinates



- Rasterize triangle: find which pixels should be lit



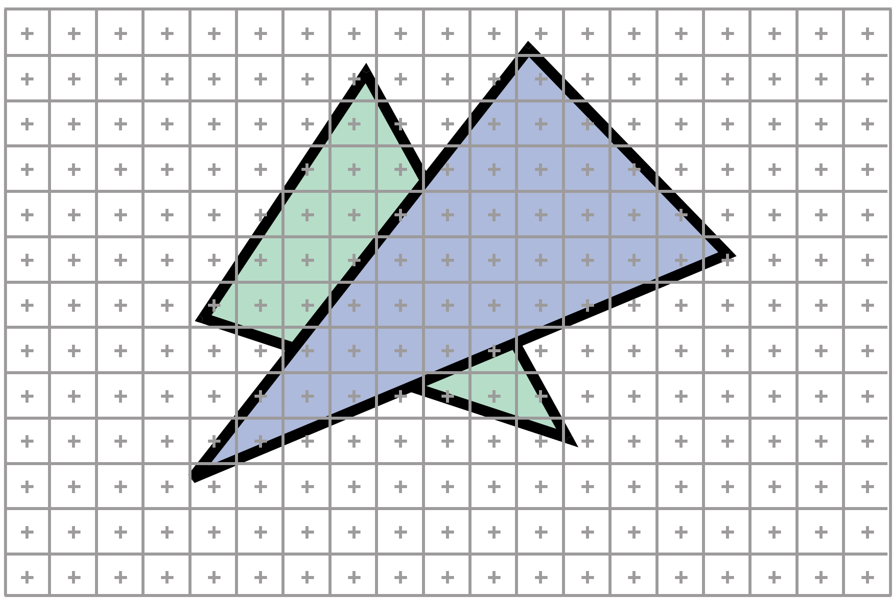
- Compute per-pixel color



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

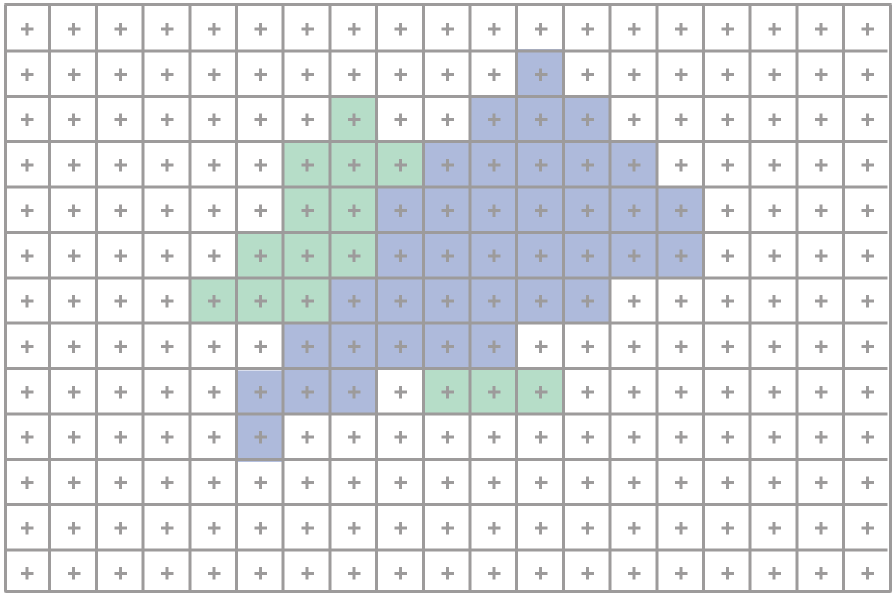
2D Scan Conversion

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

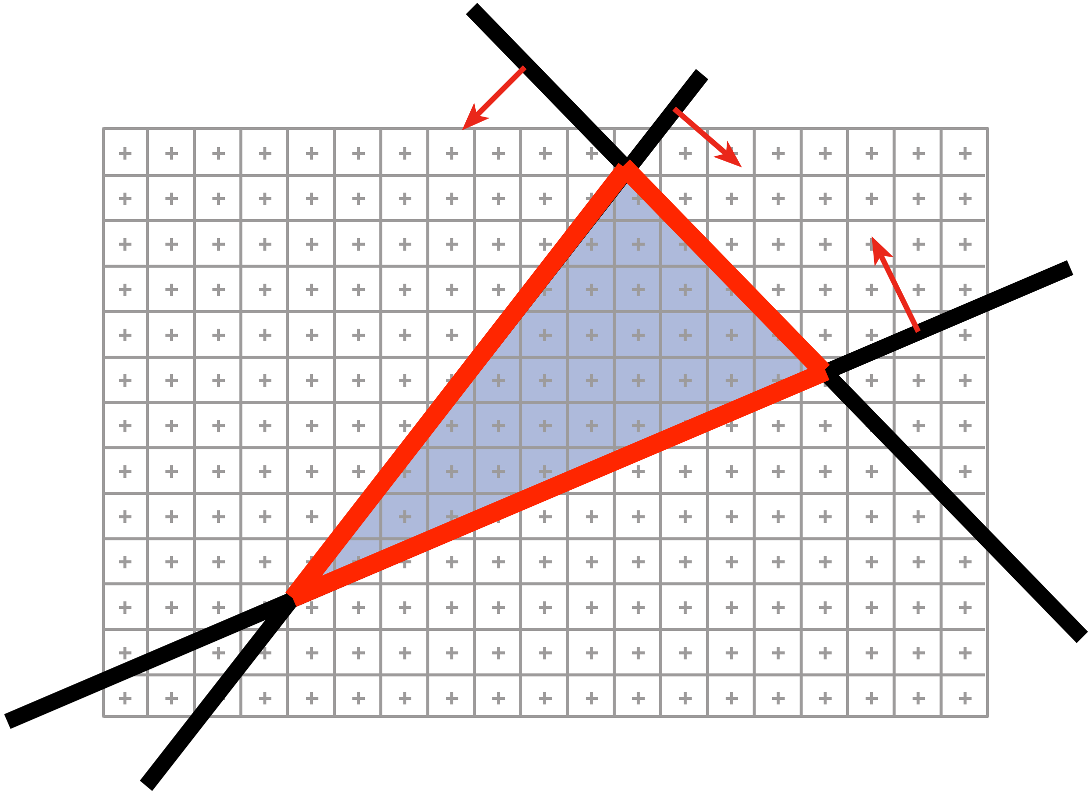


2D Scan Conversion

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

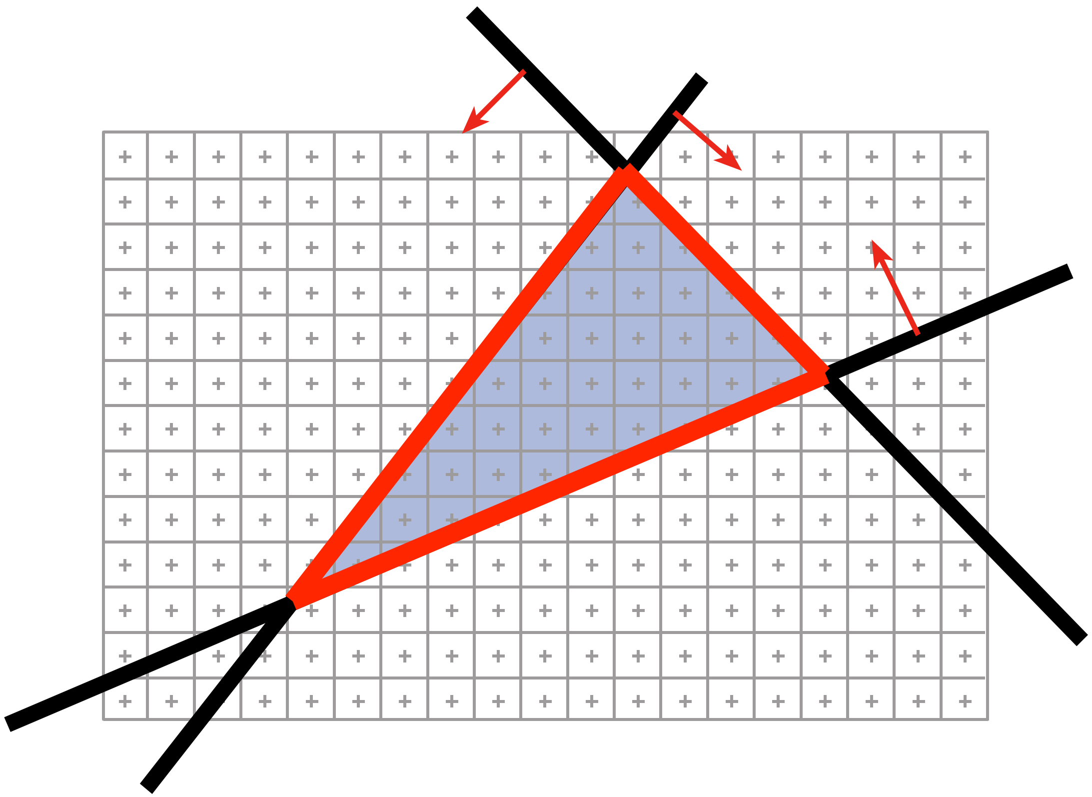


Edge Functions



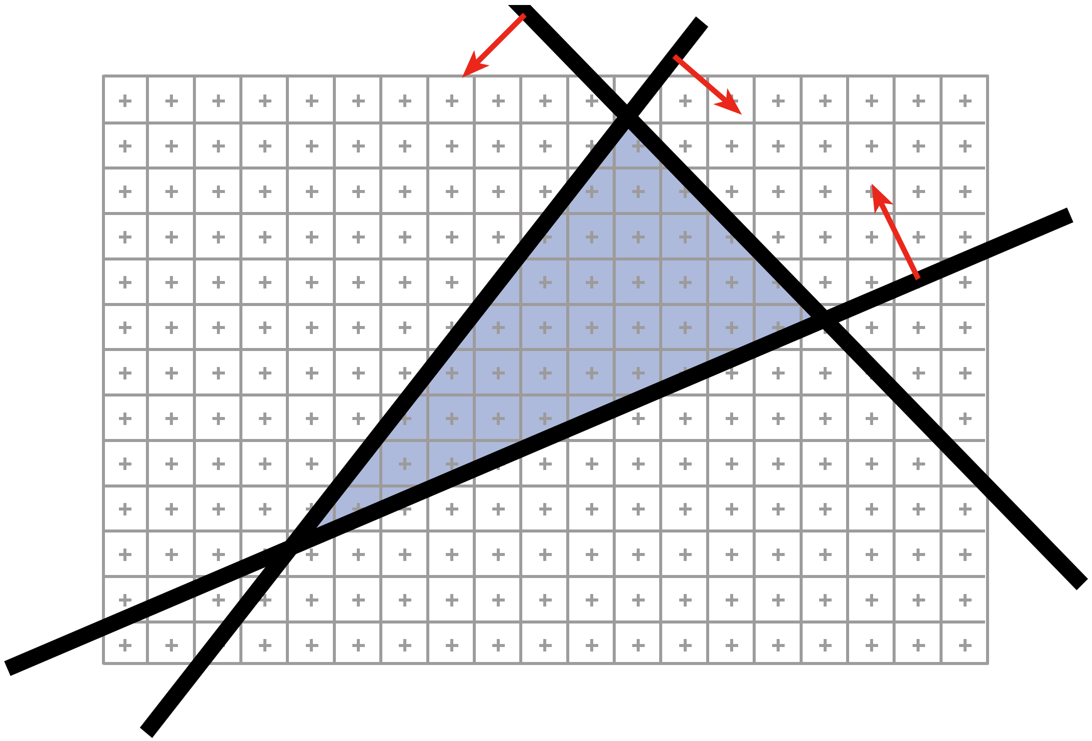
Edge Functions

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



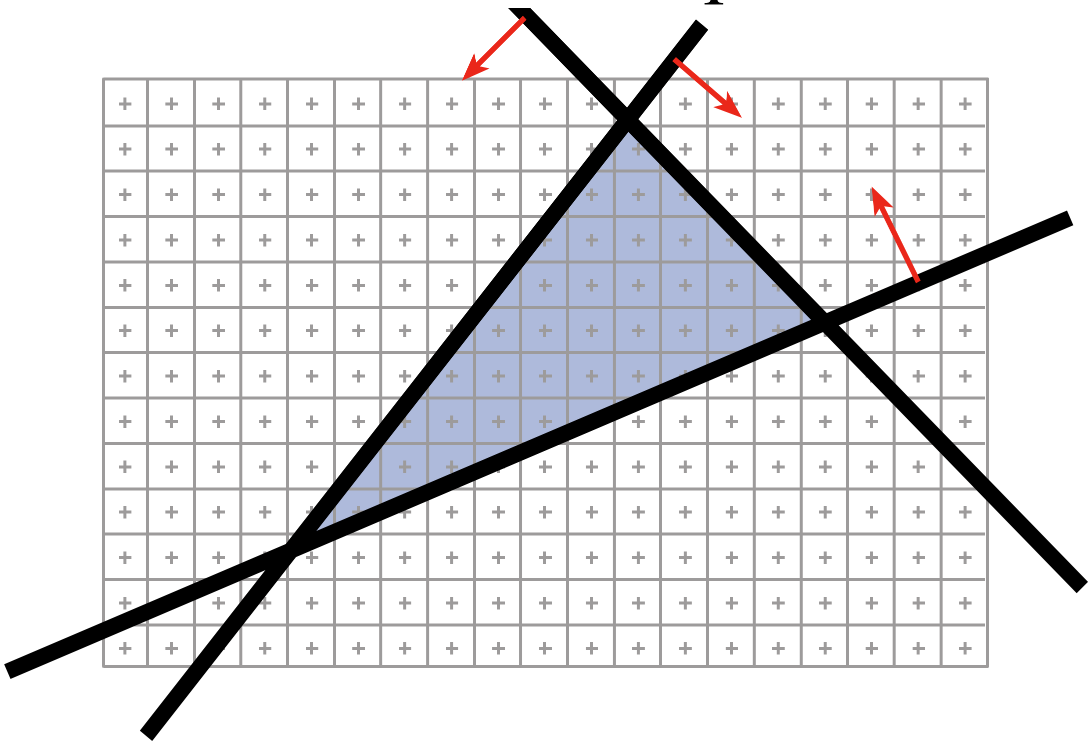
Edge Functions

- The triangle's 3D edges project to line segments in the image (thanks to planar perspective)



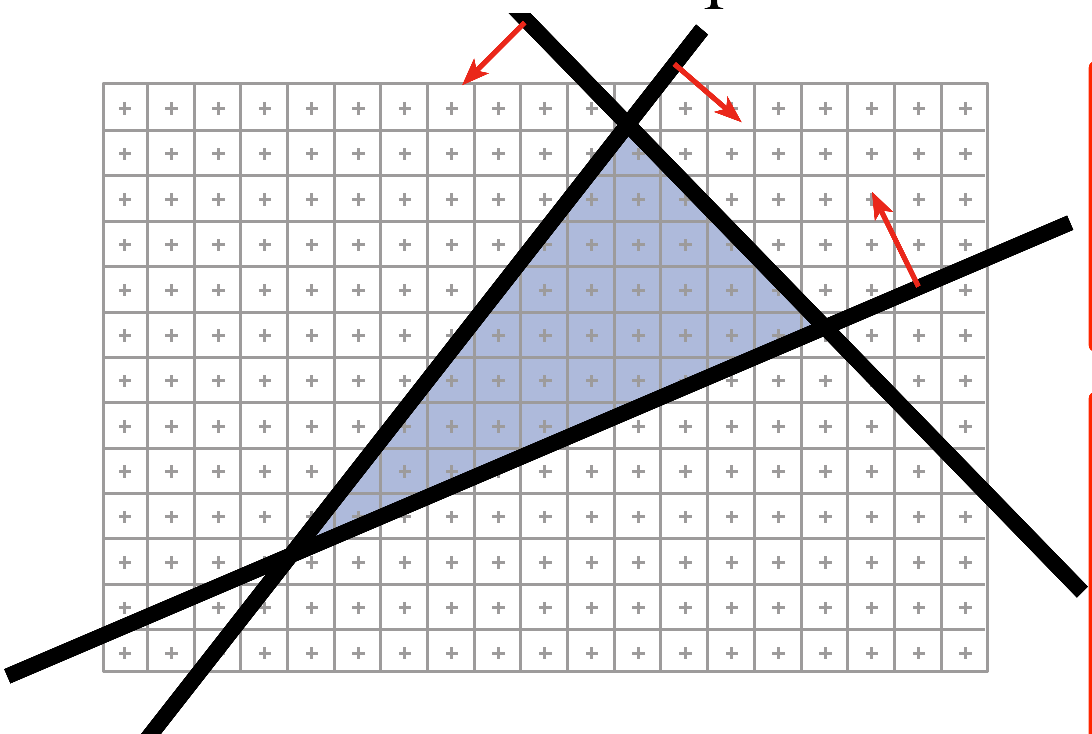
Edge Functions

- 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

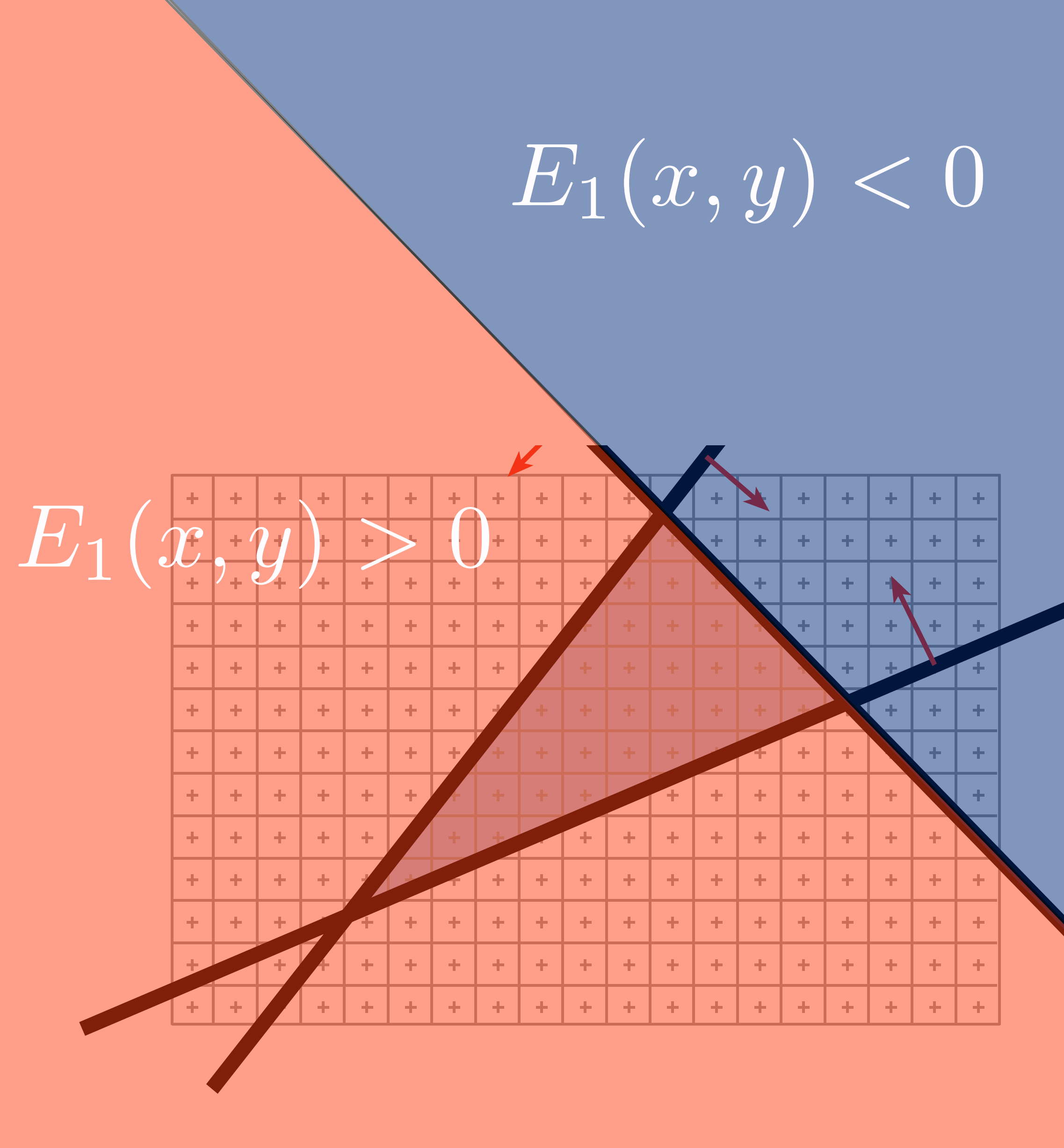
- The triangle's 3D edges project to line segments in the image (thanks to planar perspective)
- The interior of the triangle is the set of points that is inside all three halfspaces defined by these lines



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

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

Edge Function 1

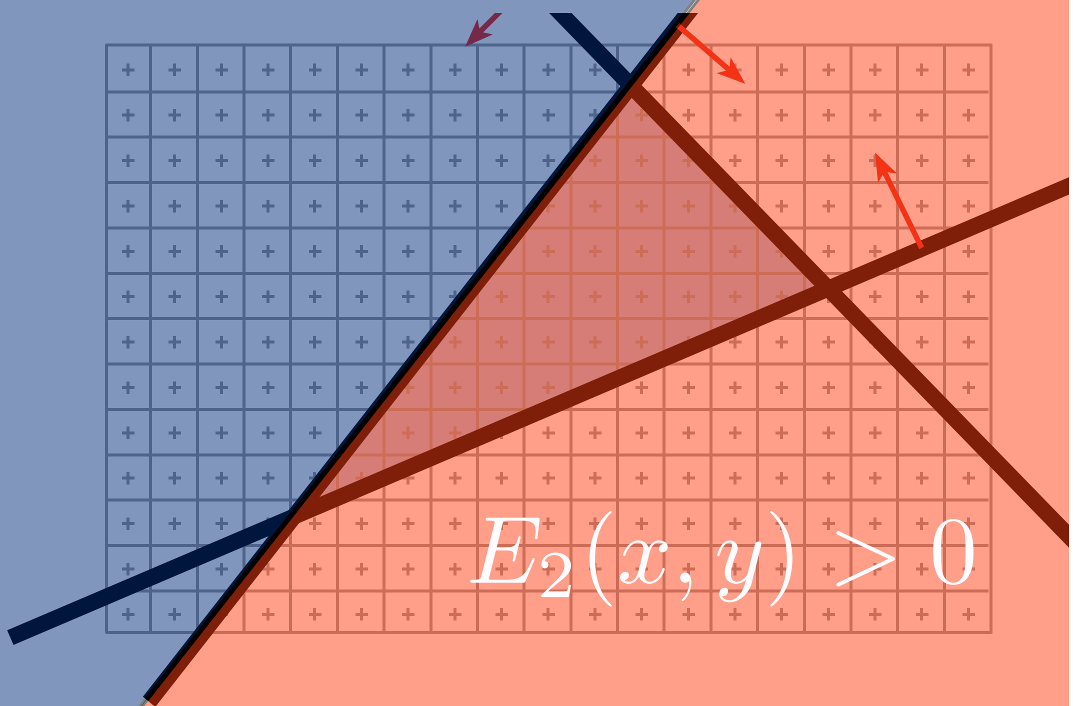


$$E_i(x, y) = a_i x + b_i y + c_i$$

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

Edge Function 2

$$E_2(x, y) < 0$$



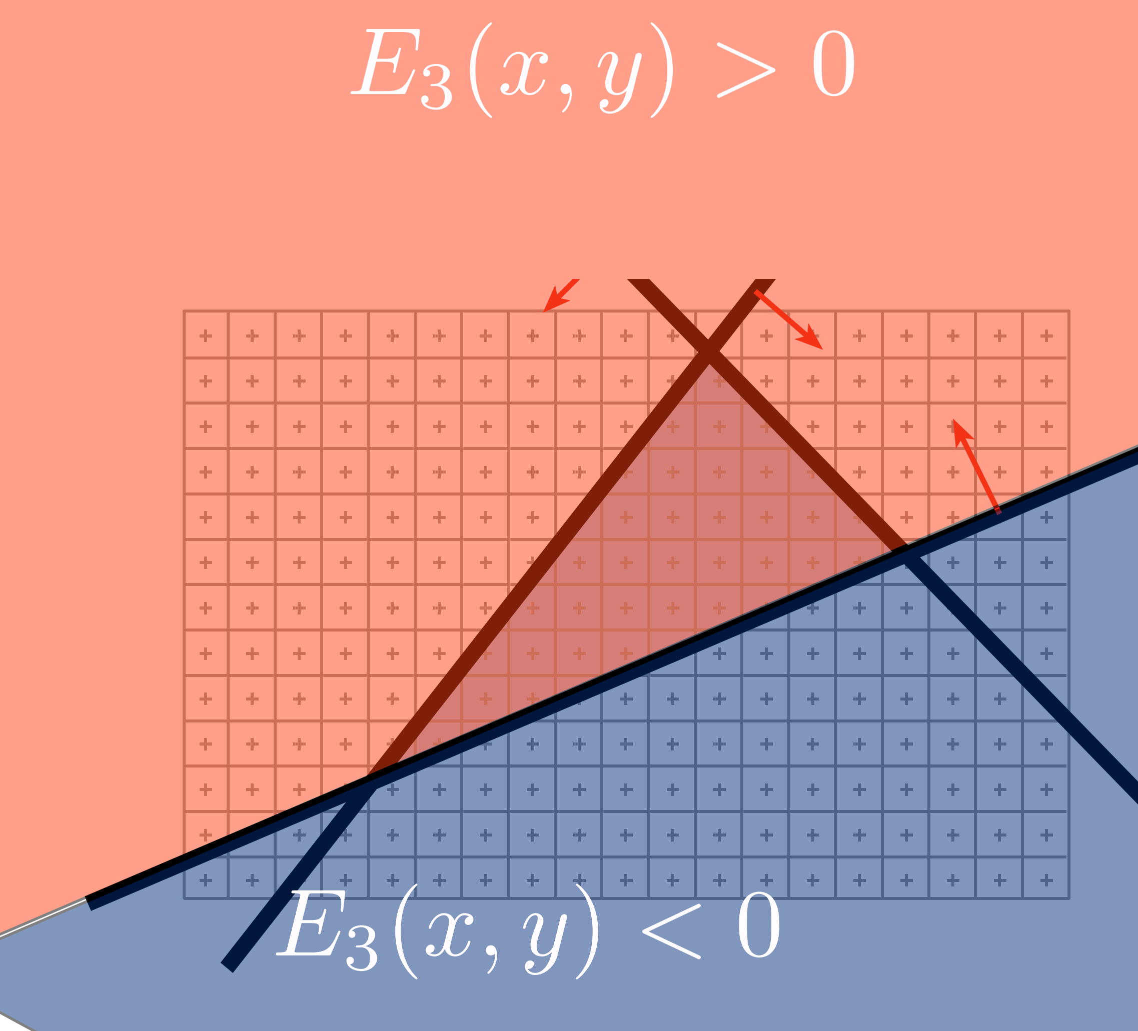
$$E_2(x, y) > 0$$

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

(x, y) within triangle

$$E_i(x, y) \geq 0, \quad \forall i = 1, 2, 3$$

Edge Function 3

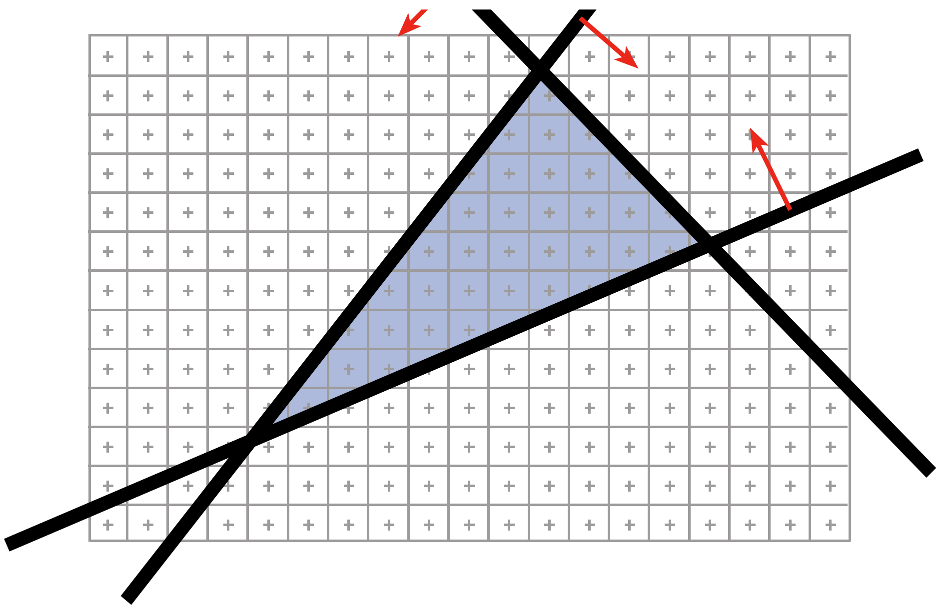


$$E_i(x, y) = a_i x + b_i y + c_i$$

(x, y) within triangle

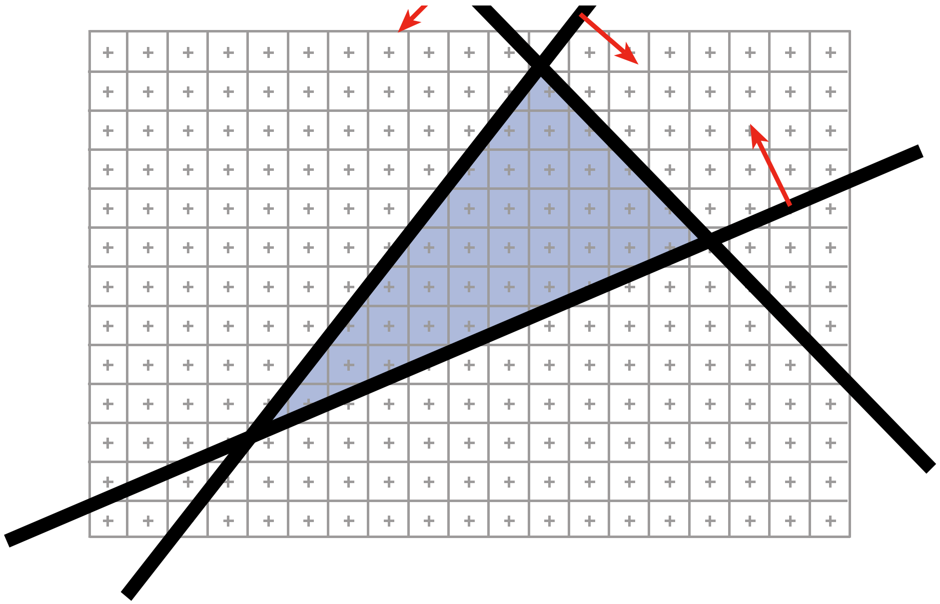
$$E_i(x, y) \geq 0, \quad \forall i = 1, 2, 3$$

Brute Force Rasterizer



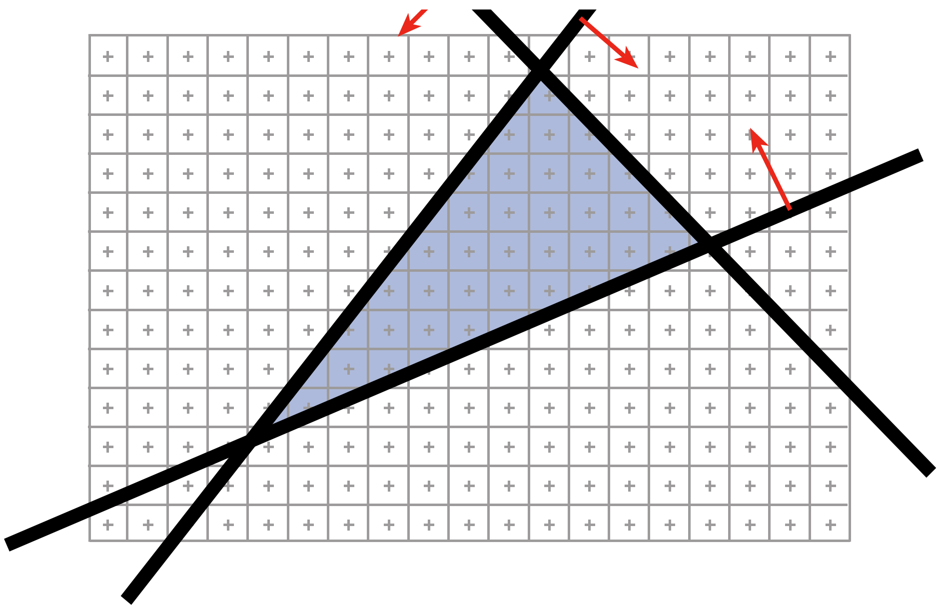
Brute Force Rasterizer

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



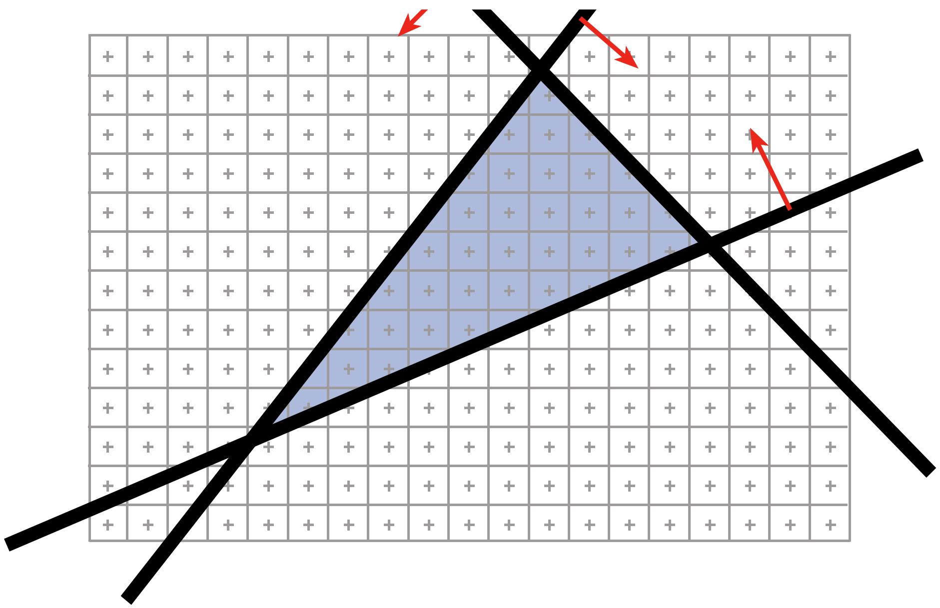
Brute Force Rasterizer

- Compute E_1-E_3 from projected vertices
  - Called "triangle setup", yields a_i, b_i, c_i for i=1,2,3
- For each pixel (x, y)
  - Evaluate edge functions at pixel center
  - If all non-negative, pixel is in!



Brute Force Rasterizer

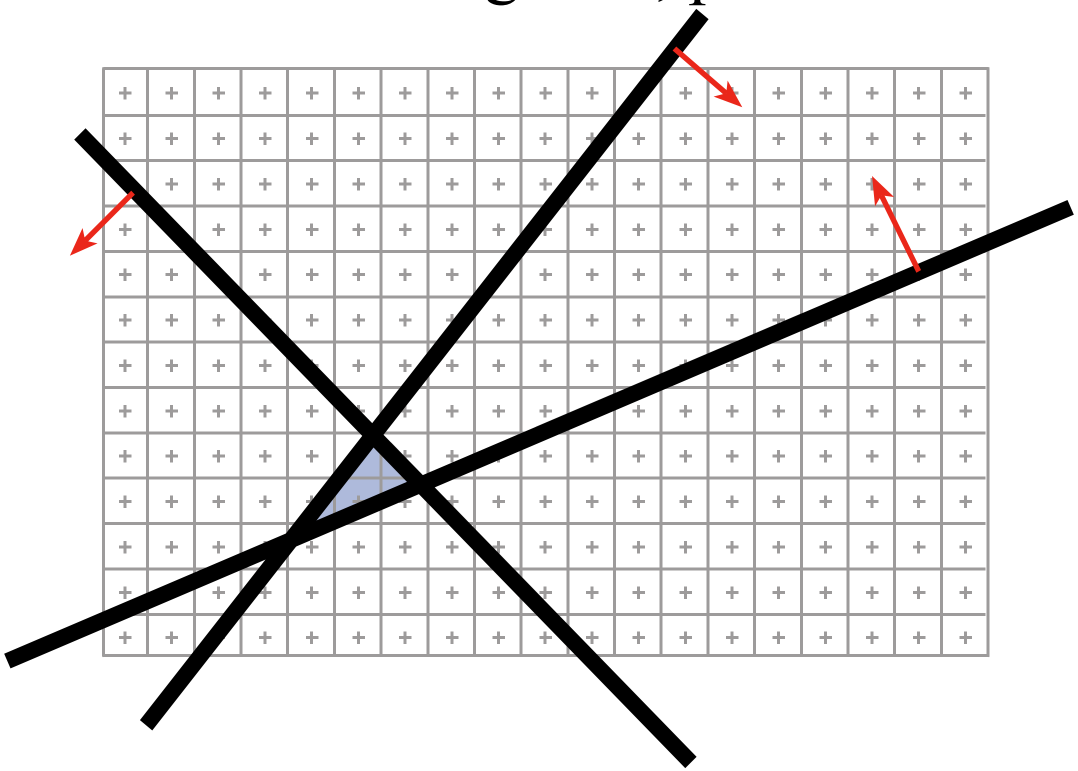
- Compute E_1-E_3 from projected vertices
  - Called "triangle setup", yields a_i, b_i, c_i for i=1,2,3
- For each pixel (x, y)
  - Evaluate edge functions at pixel center
  - If all non-negative, pixel is in!



Problem?

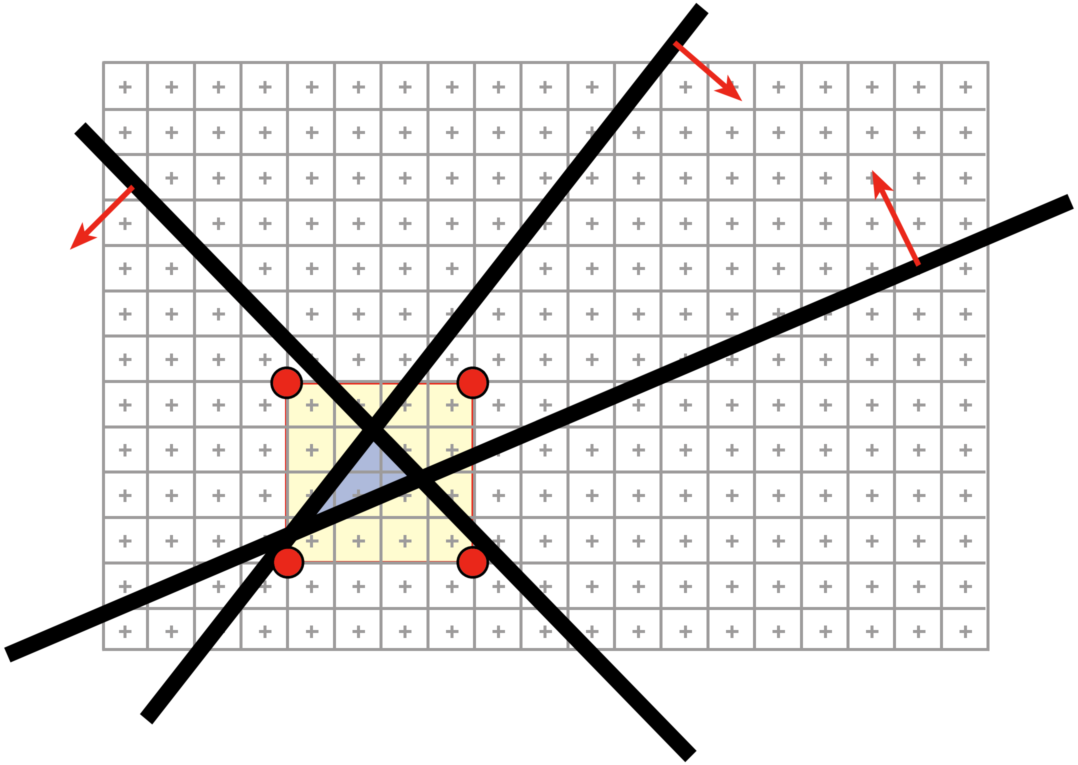
Brute Force Rasterizer

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



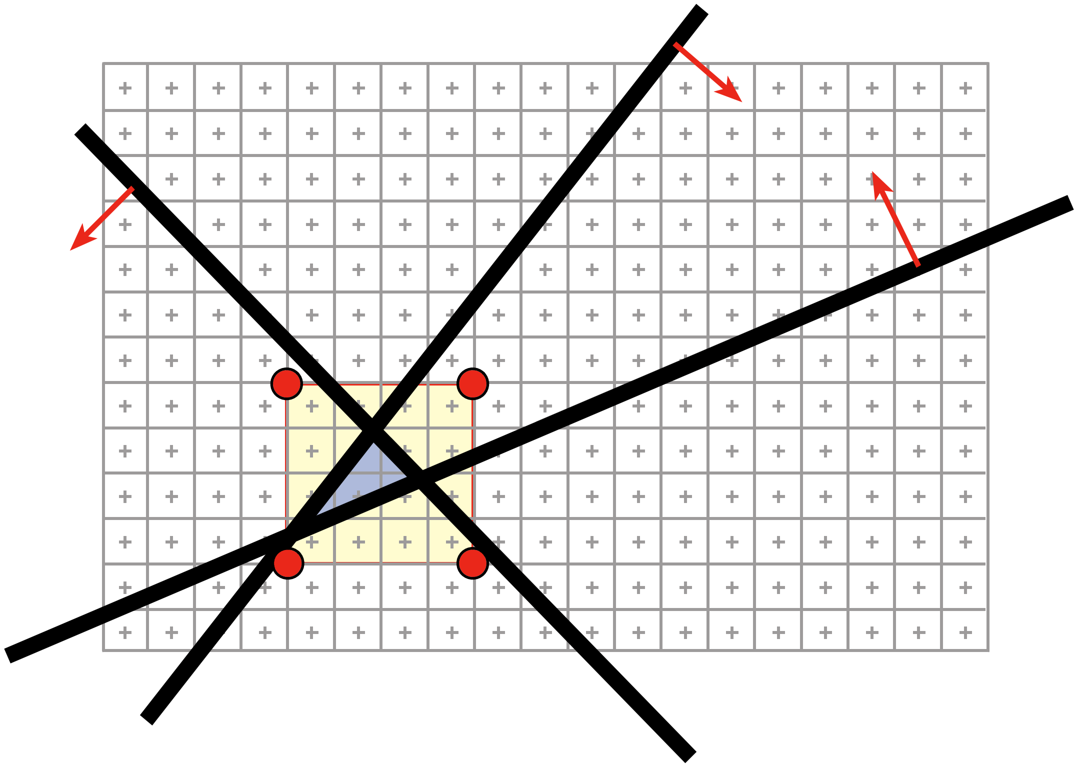
If the triangle is small, lots of useless computation if we really test all pixels

Easy Optimization



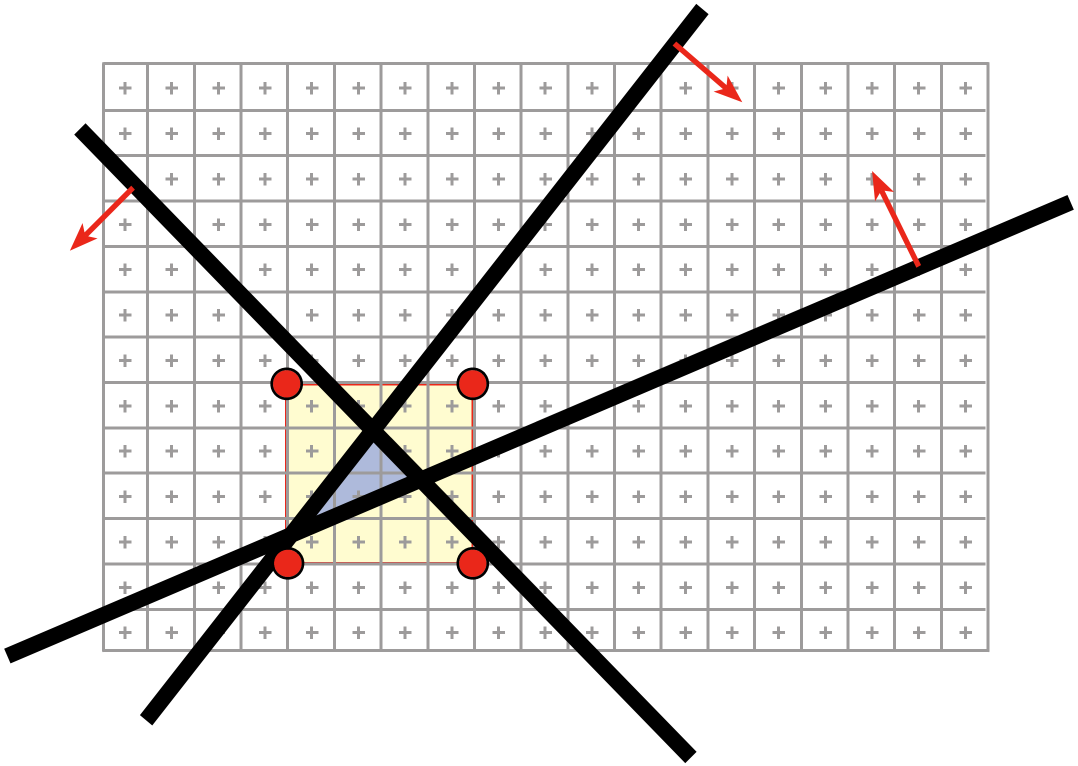
Easy Optimization

- Improvement: Scan over only the pixels that overlap the *screen bounding box* of the triangle



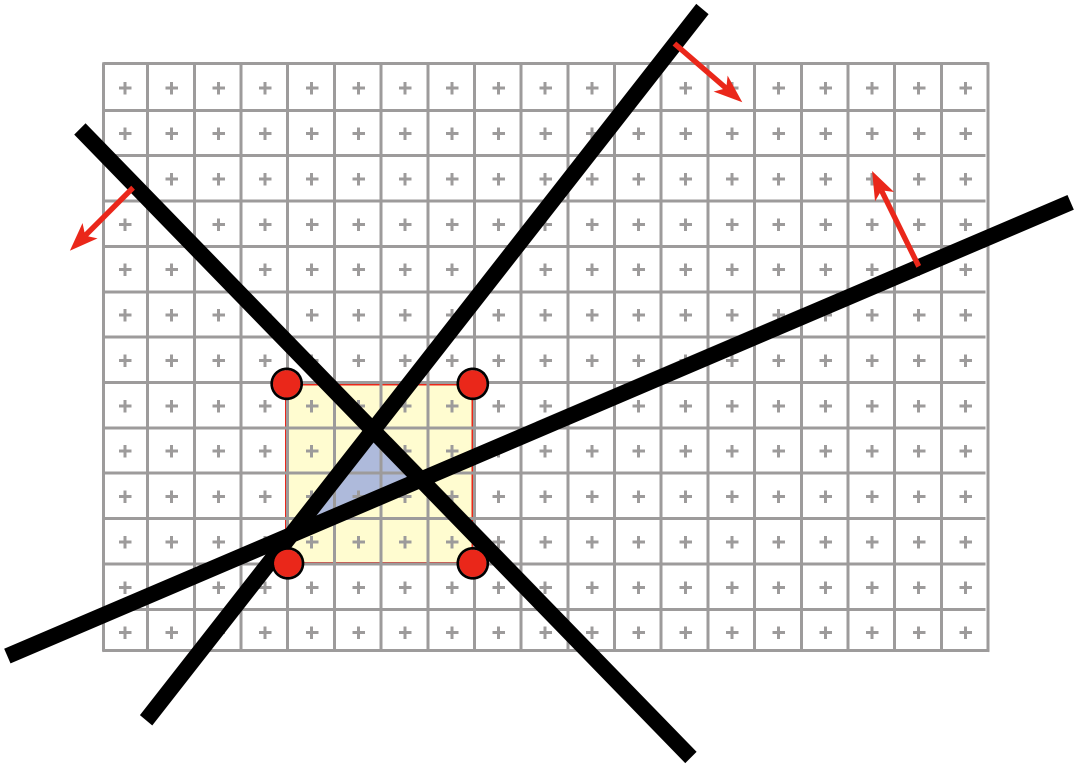
Easy Optimization

- Improvement: Scan over only the pixels that overlap the *screen bounding box* of the triangle
- How do we get such a bounding box?



Easy Optimization

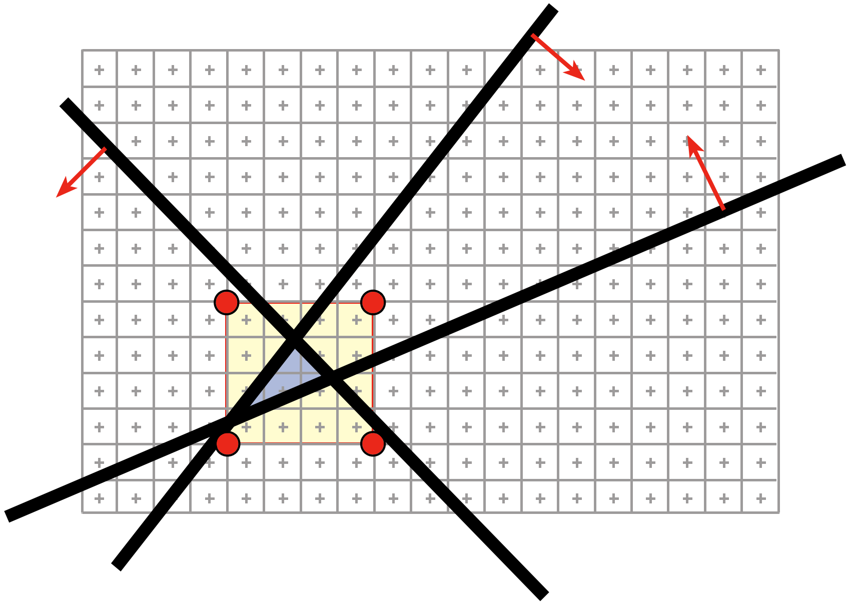
- Improvement: Scan over only the pixels that overlap the *screen bounding box* of the triangle
- How do we get such a bounding box?
  - X_{min}, X_{max}, Y_{min}, Y_{max} of the projected triangle vertices



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
```

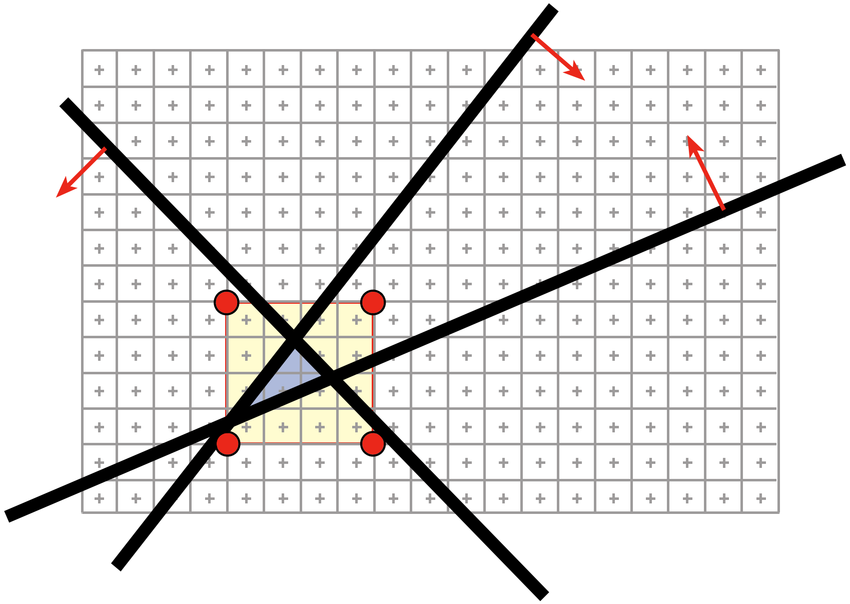


Bounding box clipping is easy, just clamp the coordinates to the screen rectangle

Can We Do Better?

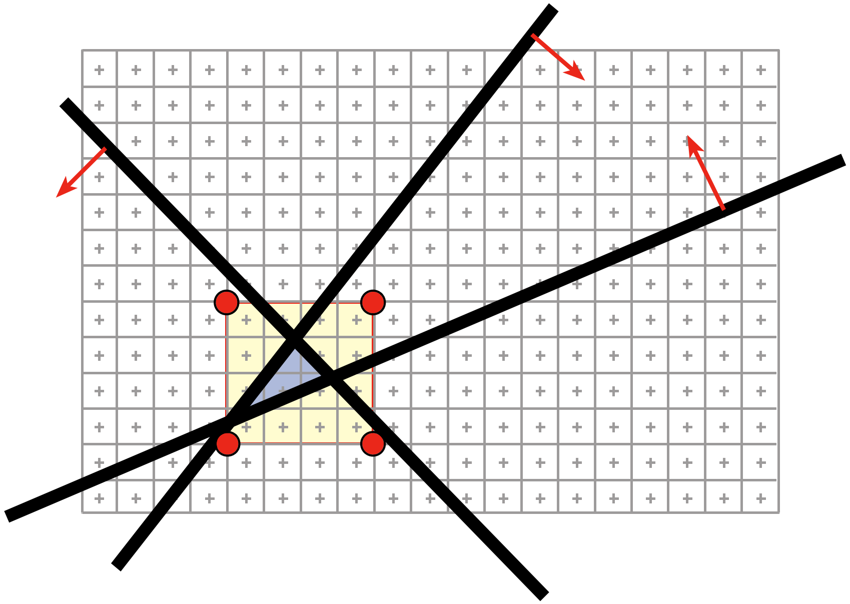
```
For every triangle
   Compute projection for vertices, compute the Ei
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Evaluate edge functions aix + biy + ci
      If all > 0
          Framebuffer[x,y ] = triangleColor
```

Evaluate edge functions $a_i x + b_i y + c_i$



Can We Do Better?

```
For every triangle
   Compute projection for vertices, compute the E_i
   Compute bbox, clip bbox to screen limits
   For all pixels in bbox
      Evaluate edge functions a_i x + b_i y + c_i
      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)

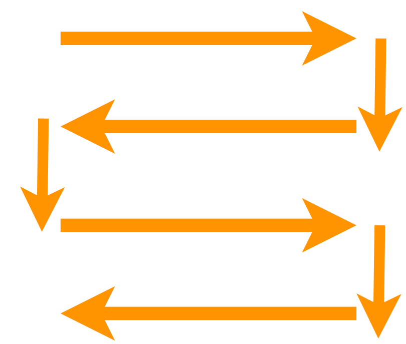
Incremental Edge Functions

```
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
```

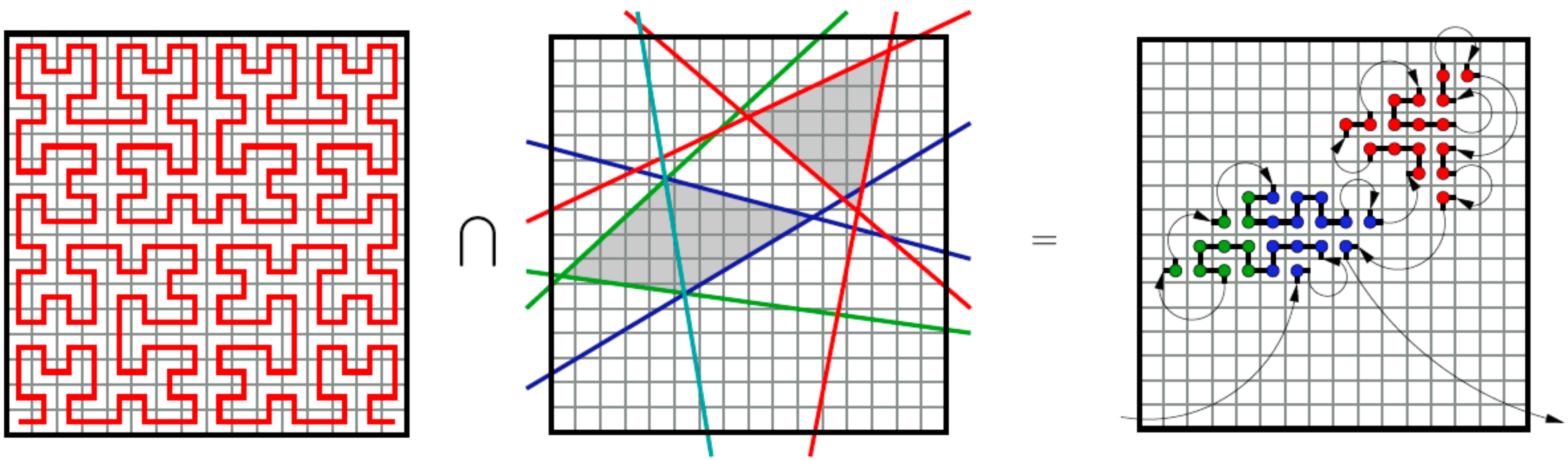


- We save ~two multiplications and two additions per pixel when the triangle is large

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

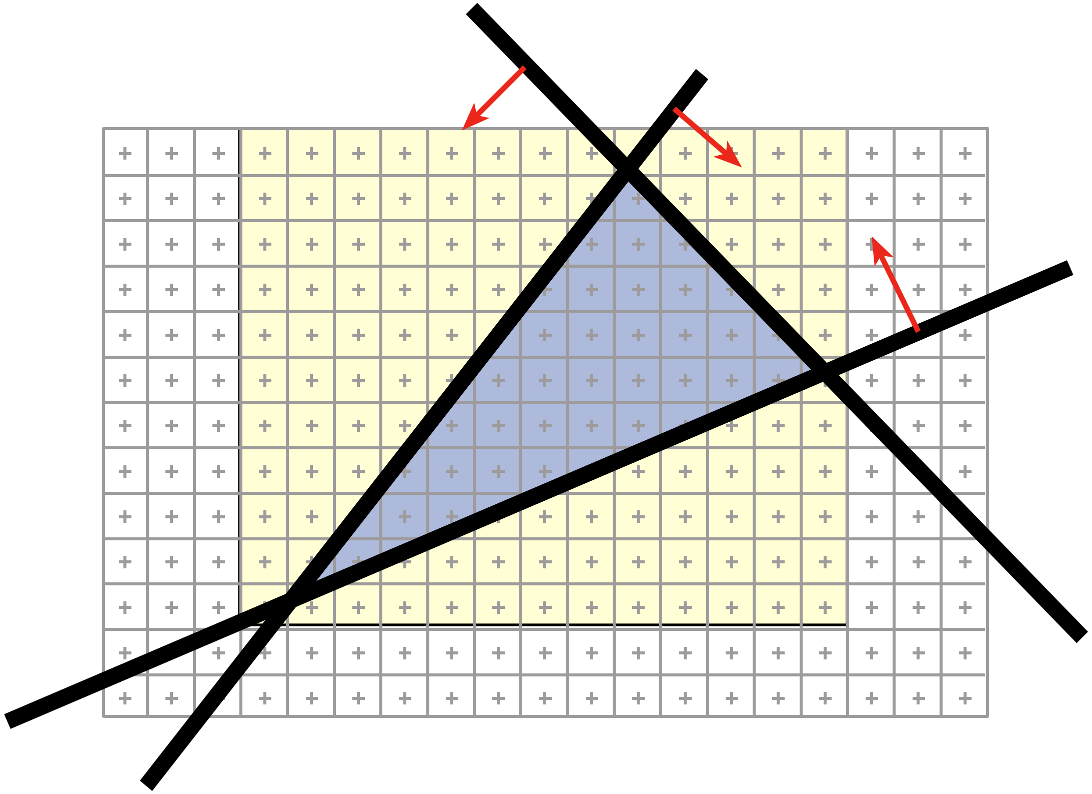
Questions?

- For a really HC piece of rasterizer engineering, see the hierarchical Hilbert curve rasterizer by McCool, Wales and Moule.
  - (Hierarchical? We'll look at that next..)

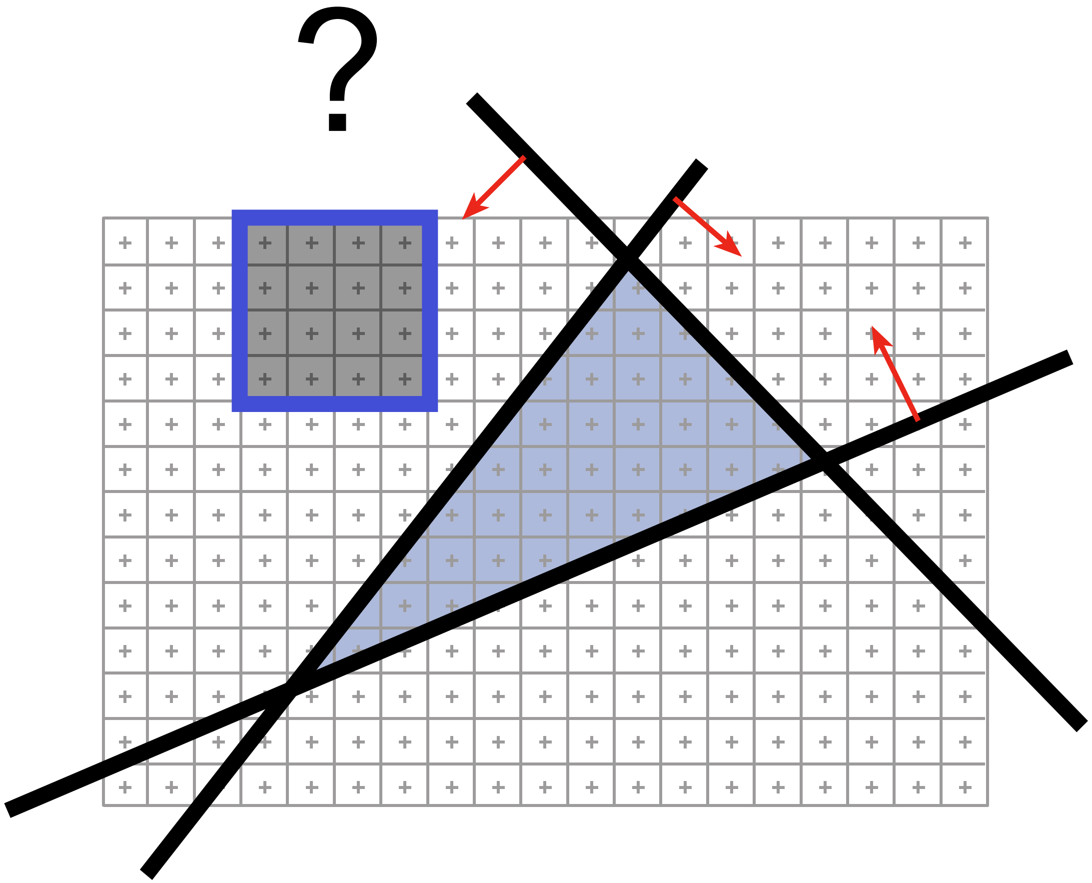


Can We Do Even Better?

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

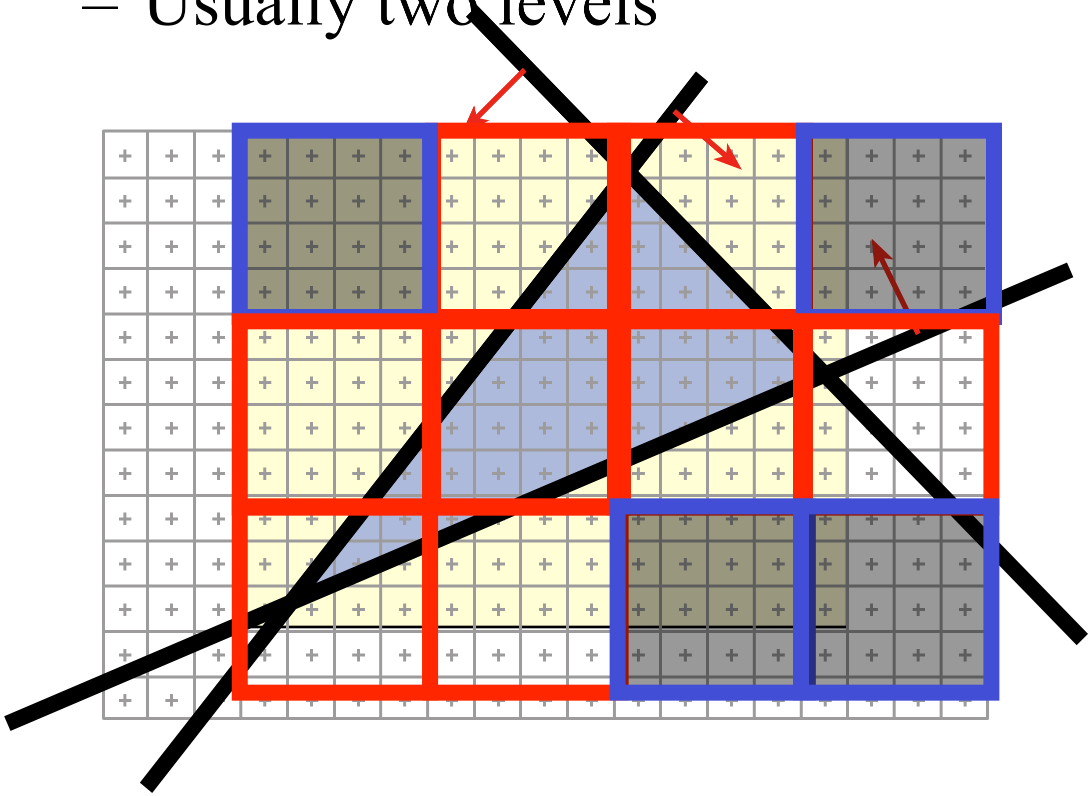


Indeed, We Can Be Smarter



Indeed, We Can Be Smarter

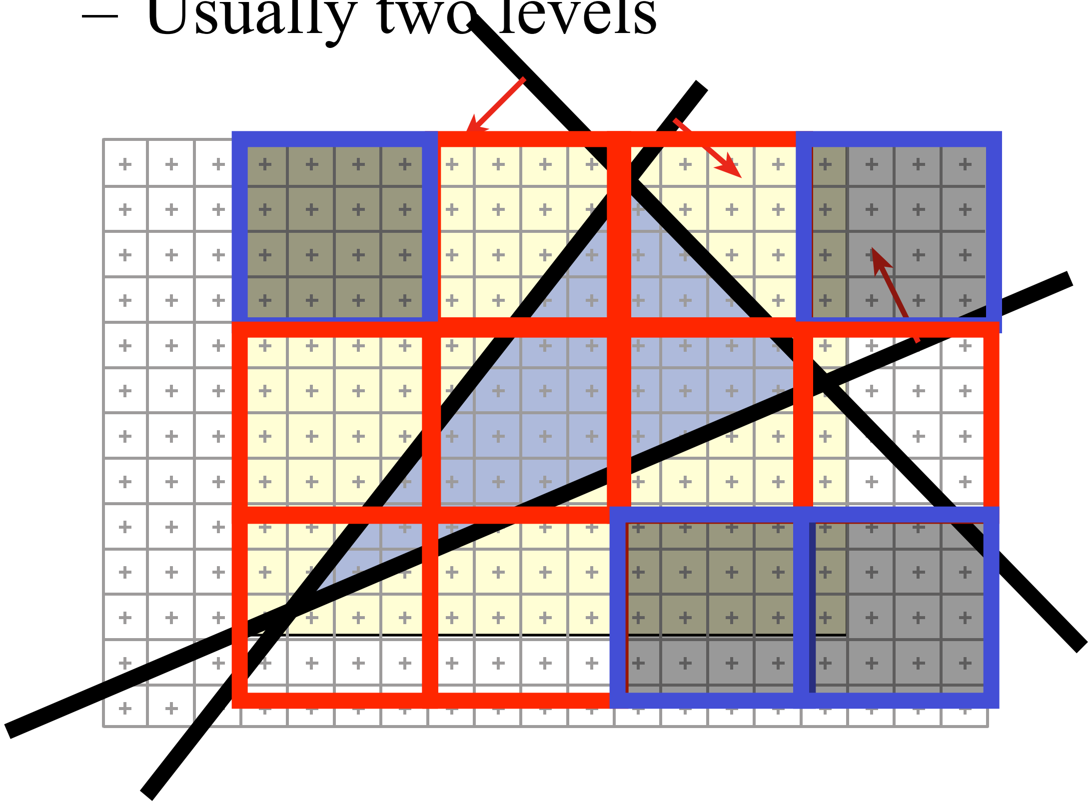
- Hierarchical rasterization!
  - Conservatively test **blocks of pixels** before going to per-pixel level (can skip large blocks at once)
  - Usually two levels



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

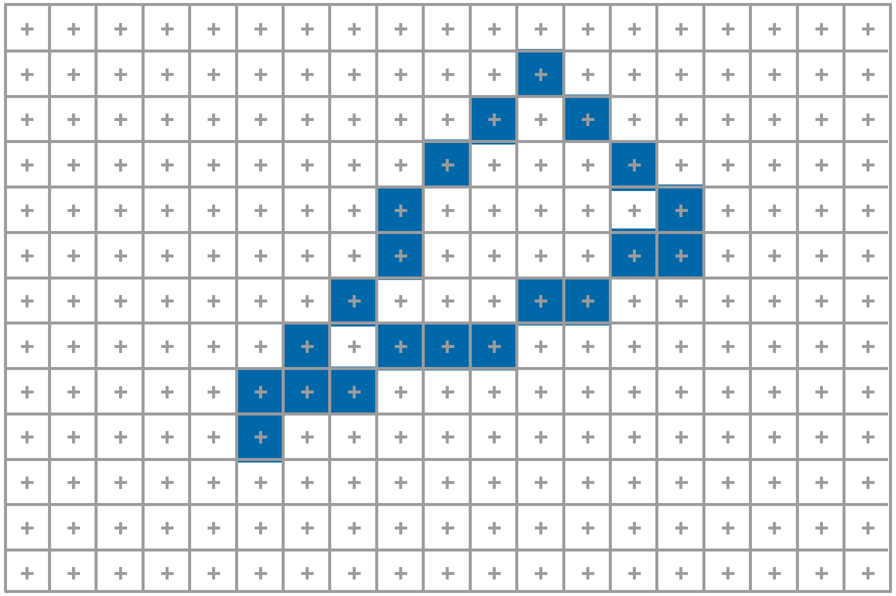
- Hierarchical rasterization!
  - Conservatively test **blocks of pixels** before going to per-pixel level (can skip large blocks at once)
  - Usually two levels



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.)

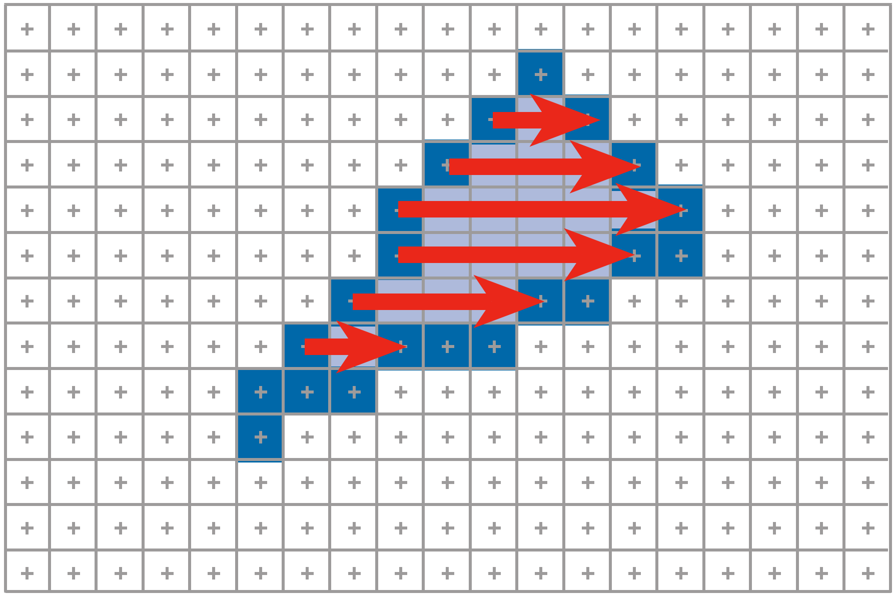
Oldskool Rasterization

- Compute the boundary pixels using line rasterization



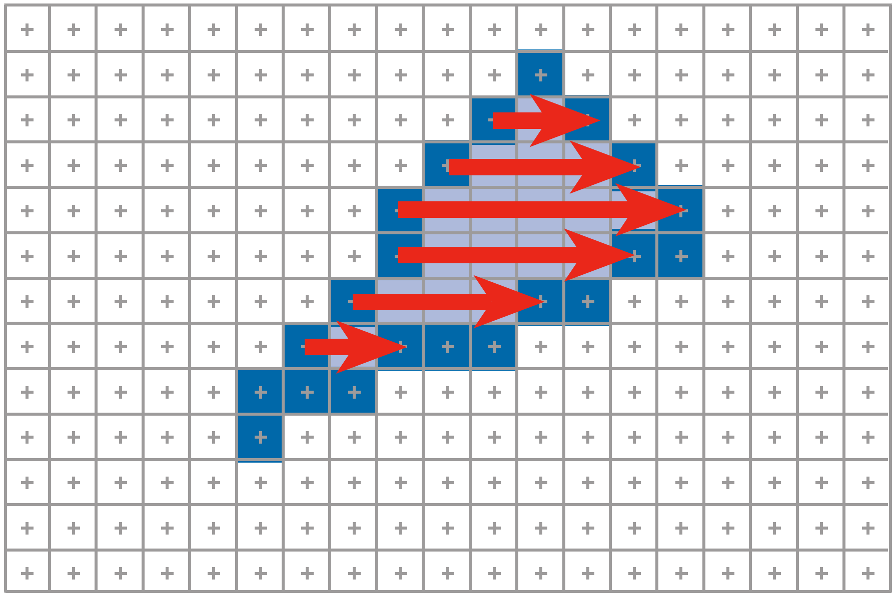
Oldskool Rasterization

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



Oldskool Rasterization

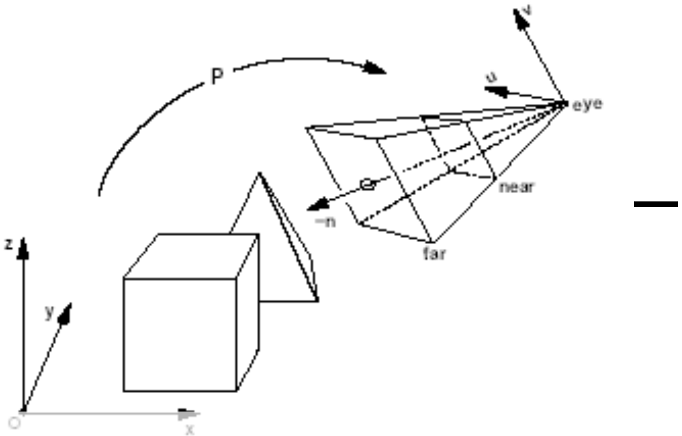
- Compute the boundary pixels using line rasterization
- Fill the spans



MUCH MORE annoying to implement than edge functions

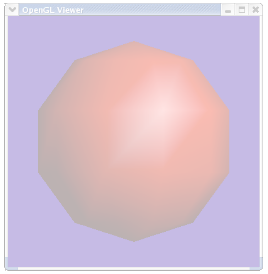
Projection

- Project vertices to 2D (image)

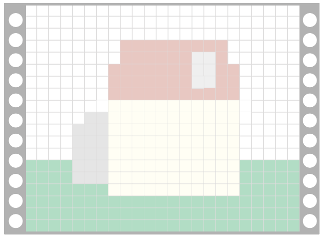
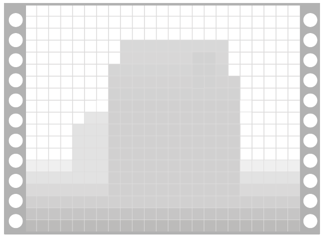


- Rasterize triangle: find which pixels should be lit

- Compute per-pixel color

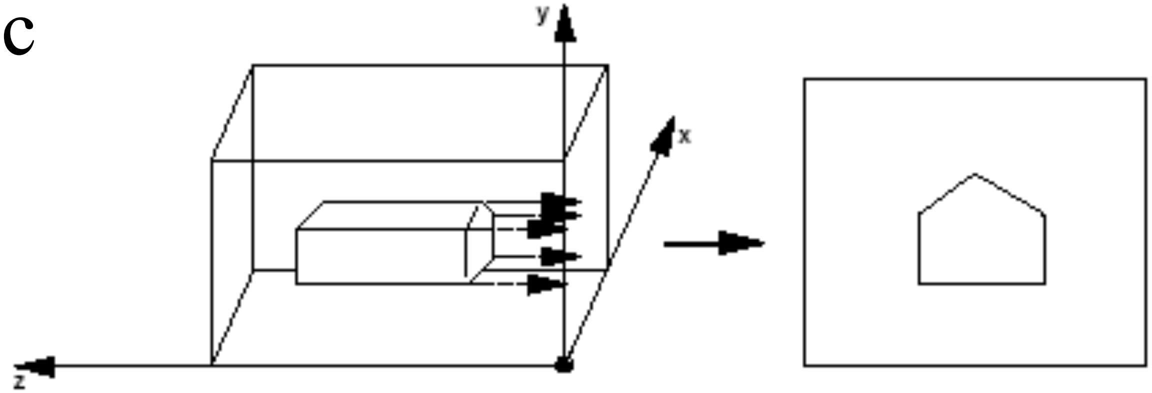


- Test visibility (Z-buffer), update frame buffer

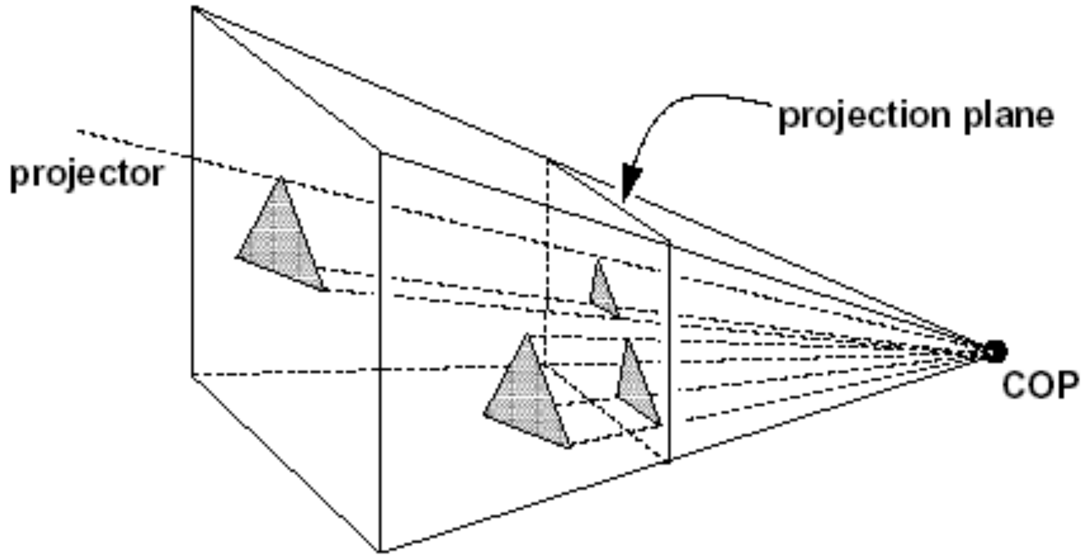



Orthographic vs. Perspective

- Orthographic

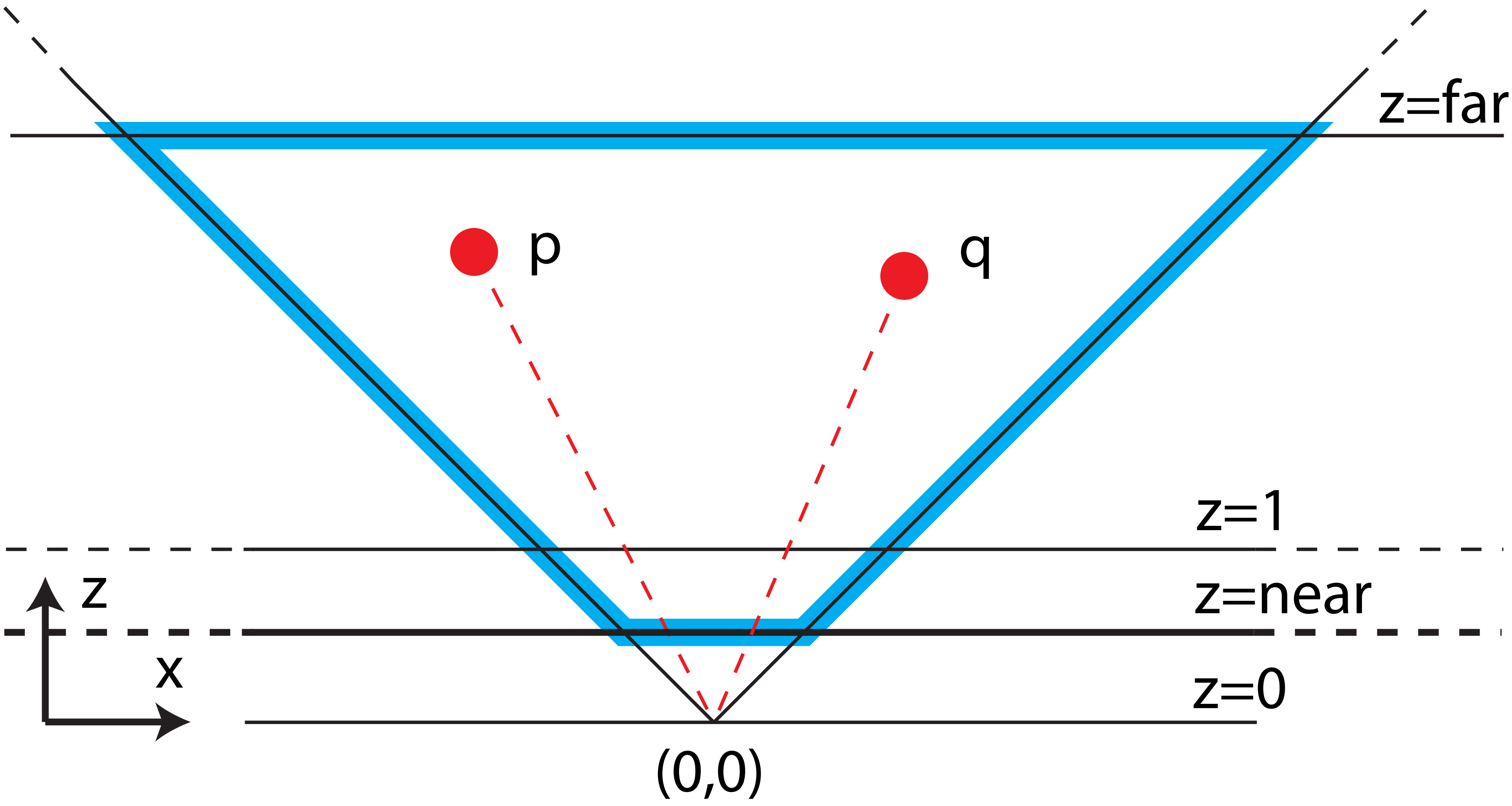


- Perspective



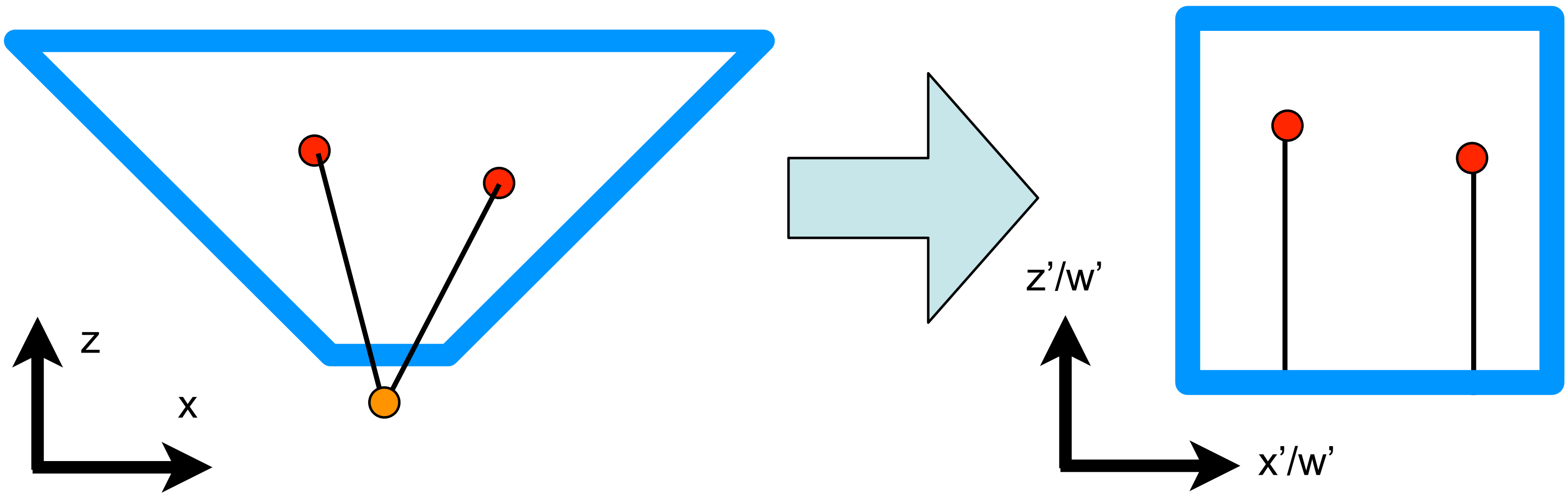
The View Frustum in 2D

- (In 3D this is a truncated pyramid.)



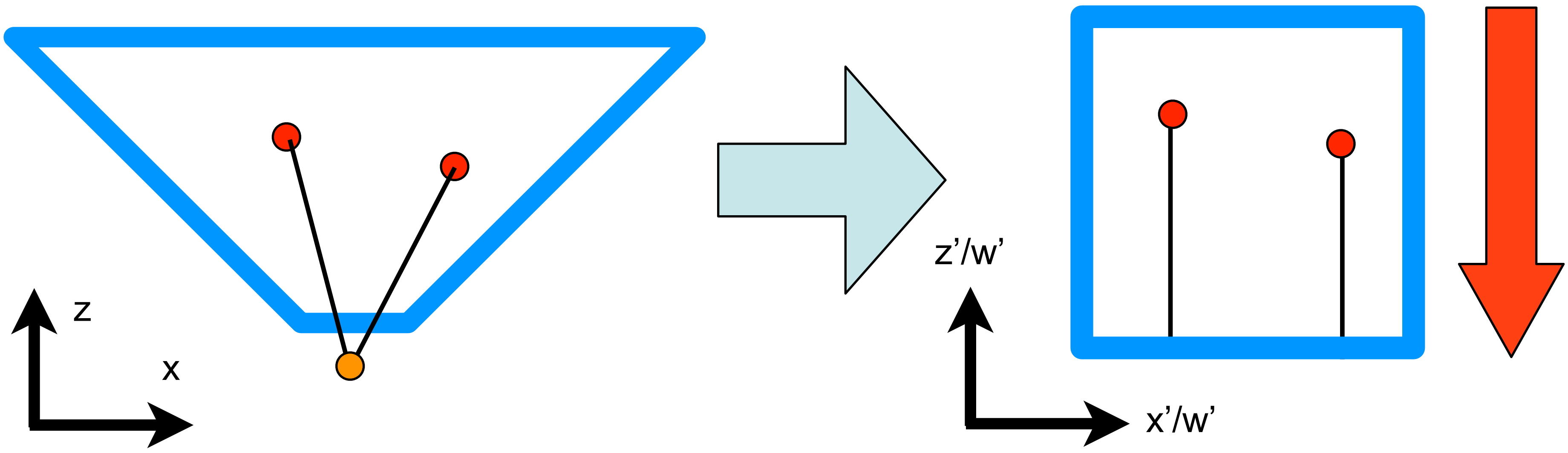
The View Frustum in 2D

- We can transform the frustum by a modified projection in a way that makes it a square (cube in 3D) after division by *w*’.



The View Frustum in 2D

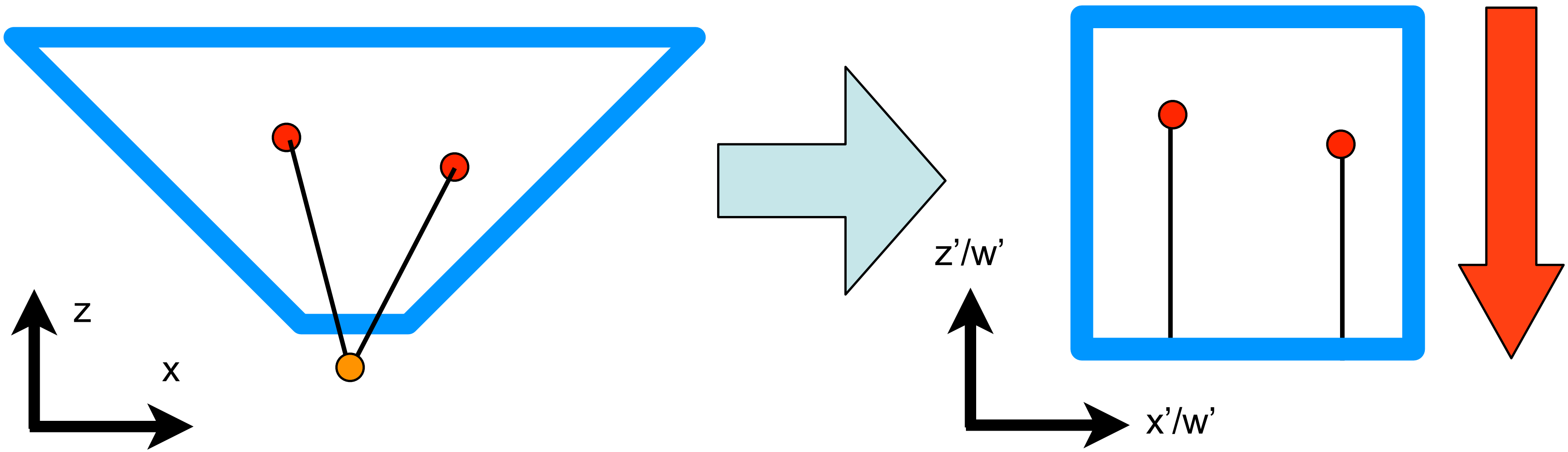
- We can transform the frustum by a modified projection in a way that makes it a square (cube in 3D) after division by *w*'.



The final image is obtained by merely dropping the z coordinate after projection (orthogonal projection)

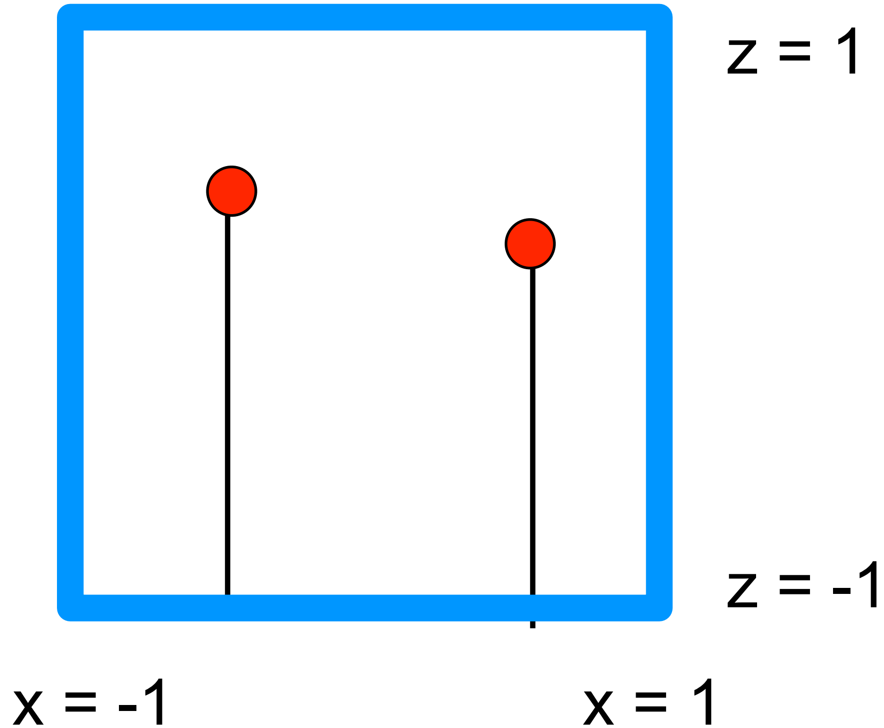
The View Frustum in 2D

- We can transform the frustum by a modified projection in a way that makes it a square (cube in 3D) after division by *w'*.



- The x' coordinate does not change w.r.t. the usual flattening projection, i.e., x'/w' stays the same

The Canonical View Volume



- Point of the exercise: This gives screen coordinates and depth values for Z-buffering with unified math
  - Caveat: OpenGL and DirectX define Z differently [0,1] vs. [-1,1]

OpenGL Form of the Projection

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{\text{far}+\text{near}}{\text{far}-\text{near}} & -\frac{2*\text{far}*\text{near}}{\text{far}-\text{near}} \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

Homogeneous coordinates within canonical view volume

Input point in view coordinates

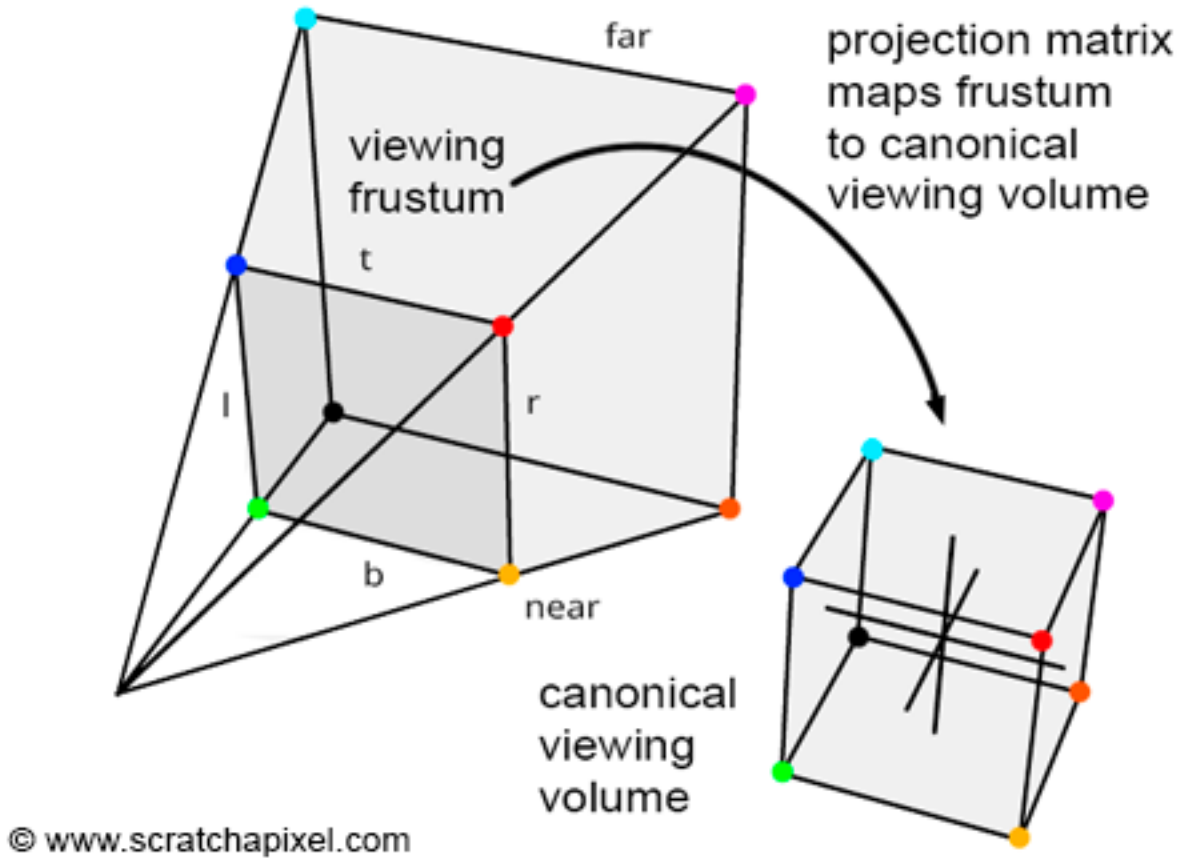
- Details/more intuition in handout in MyCourses
  - "Understanding Projections and Homogenous Coordinates"

Recap: Projection

- Perform rotation/translation/other transforms to put viewpoint at origin and view direction along z axis

- Combine with projection matrix (perspective or orthographic)
  - Homogenization achieves foreshortening

- **Corollary**: The entire transform from object space to canonical view volume $[-1,1]^3$ is a single matrix


far
projection matrix
maps frustum
to canonical
viewing volume
viewing
frustum
t
l
r
b
near
canonical
viewing
volume
© www.scratchapixel.com