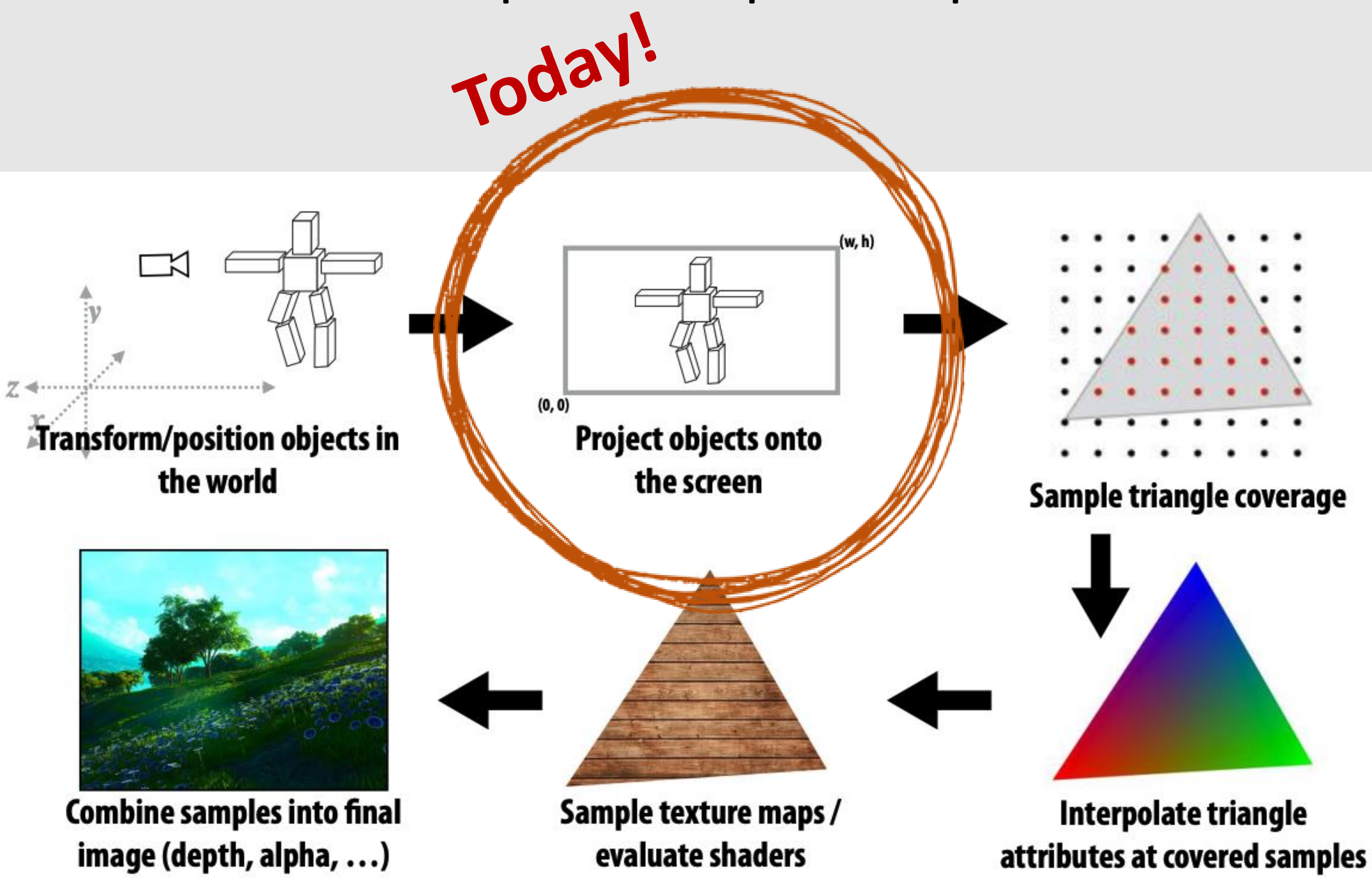
Perspective Projection & Rasterization

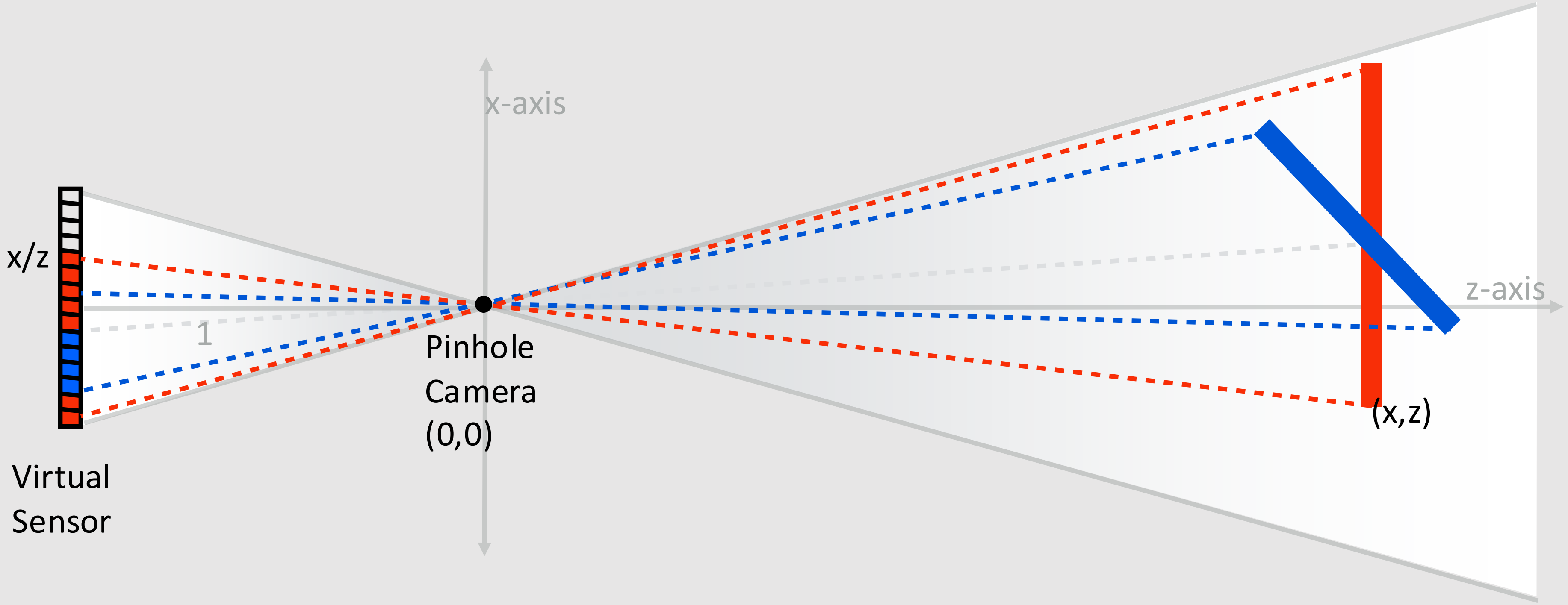
- Perspective Projection
- Drawing a Line
- Drawing a Triangle
- Supersampling

The “Simpler” Graphics Pipeline



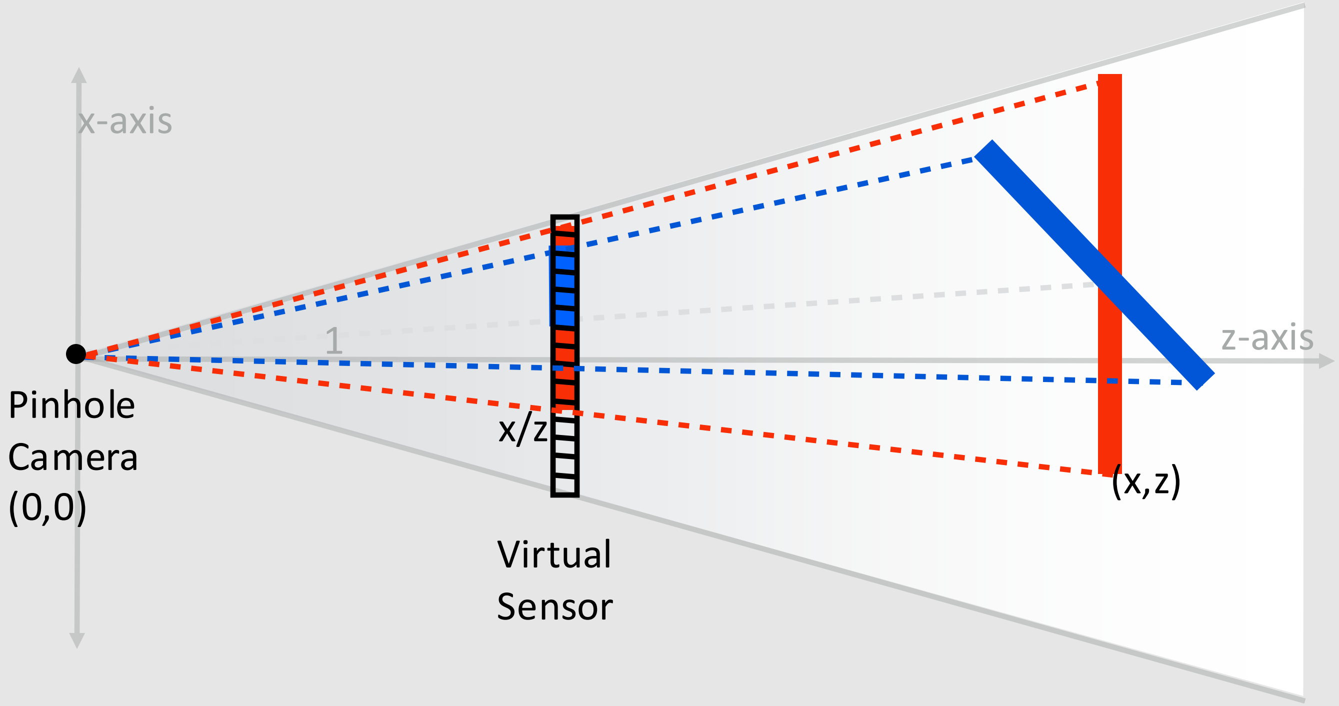
Perspective Projection

The Pinhole Camera



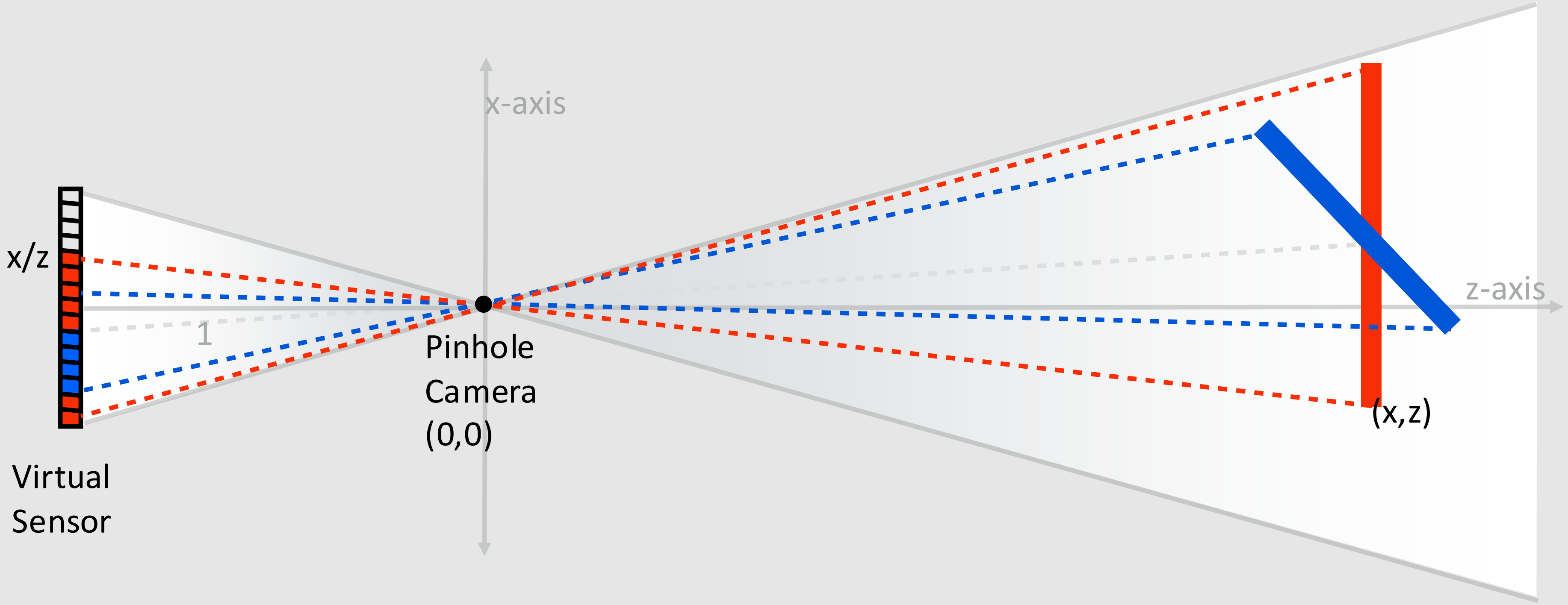
Our image seems to be upside down…

The Pinhole Camera



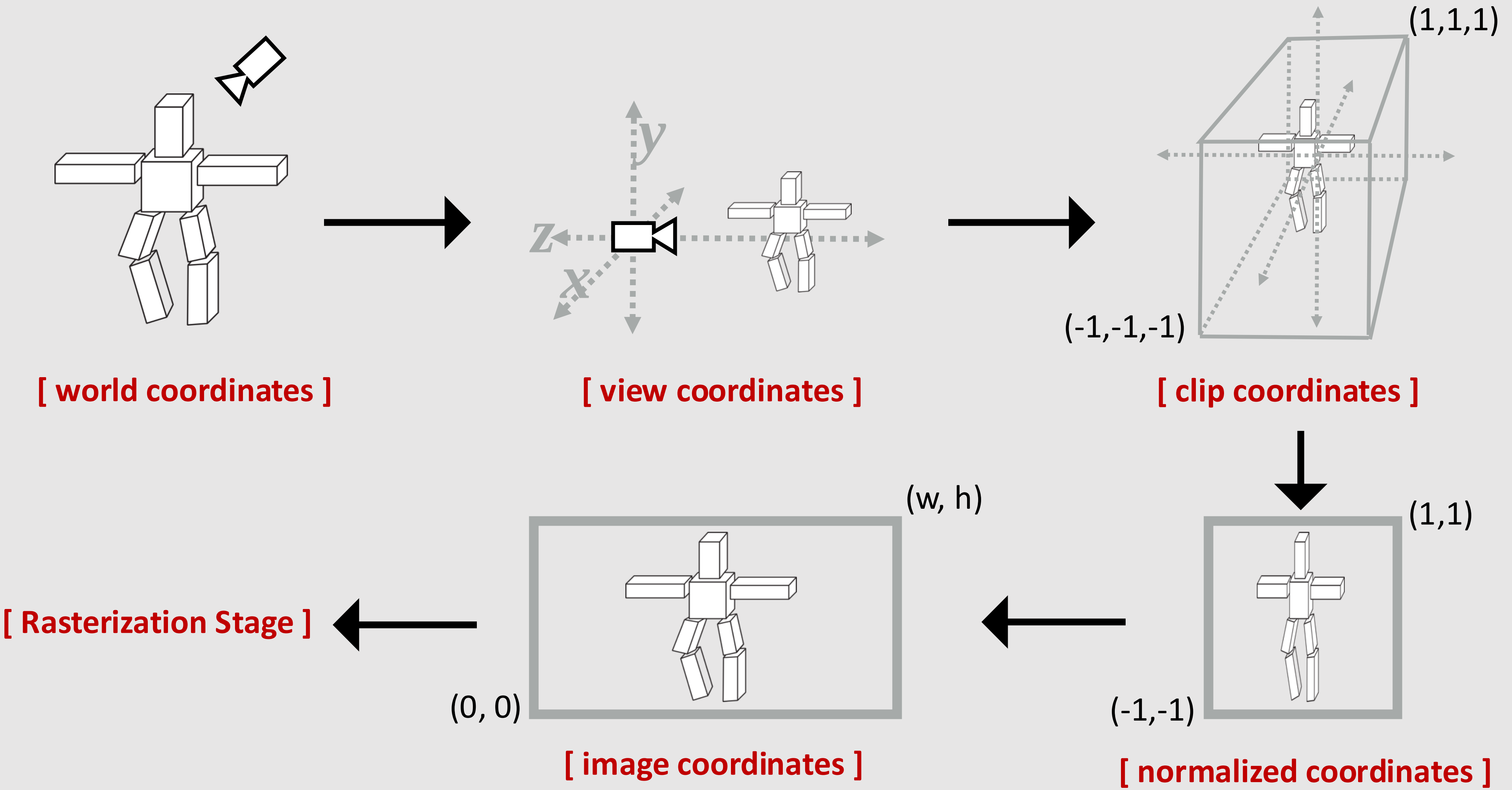
Better!...but what if part of our scene is closer that $z < 1$?

The Pinhole Camera

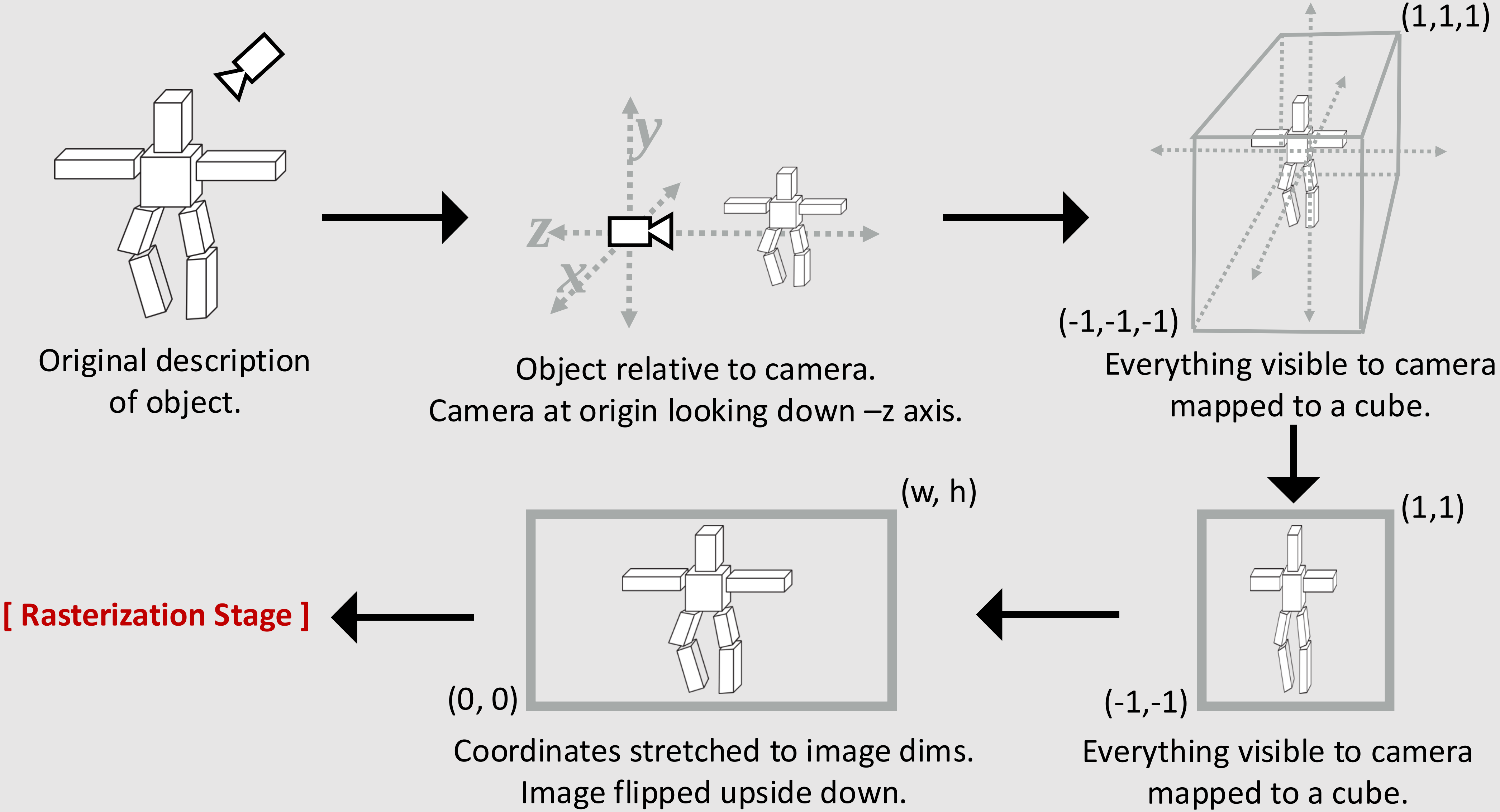


We'll just go back to capturing content like this
We can always flip the image at the end

Perspective Projection

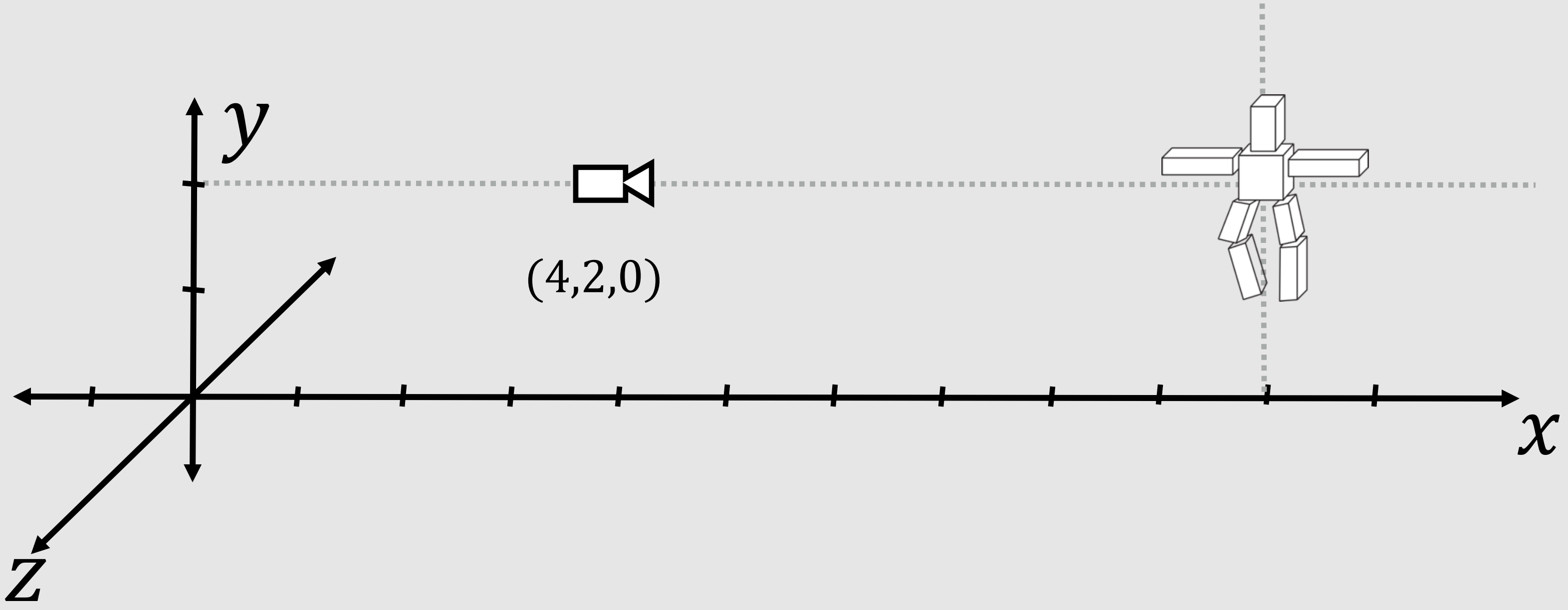


Perspective Projection



Camera Example

Consider camera at (4,2,0), looking down x-axis, object given in world coordinates:

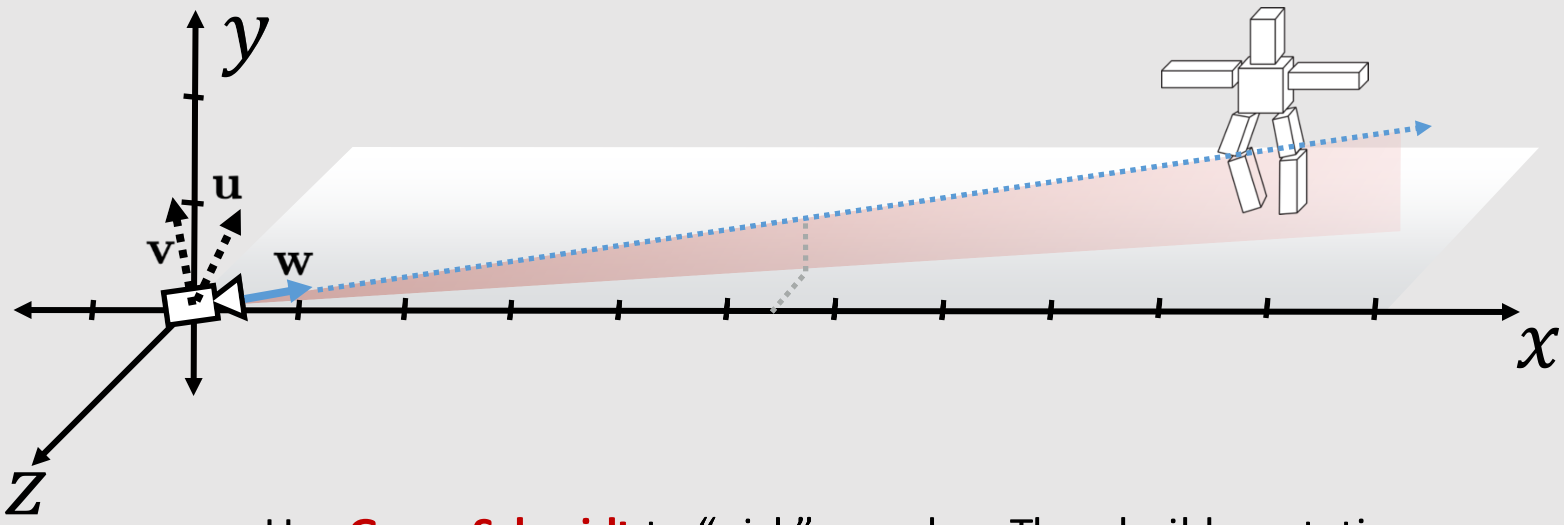


Goal: find a spatial transformation for the object in a coordinate system where the camera is at the origin, looking down the –z axis

1) Translate by (-4,-2,0)
2) Rotate by 90deg about the y-axis

Camera Example

Now consider a camera at the origin looking in a direction $\mathbf{w} \in \mathbb{R}$^3

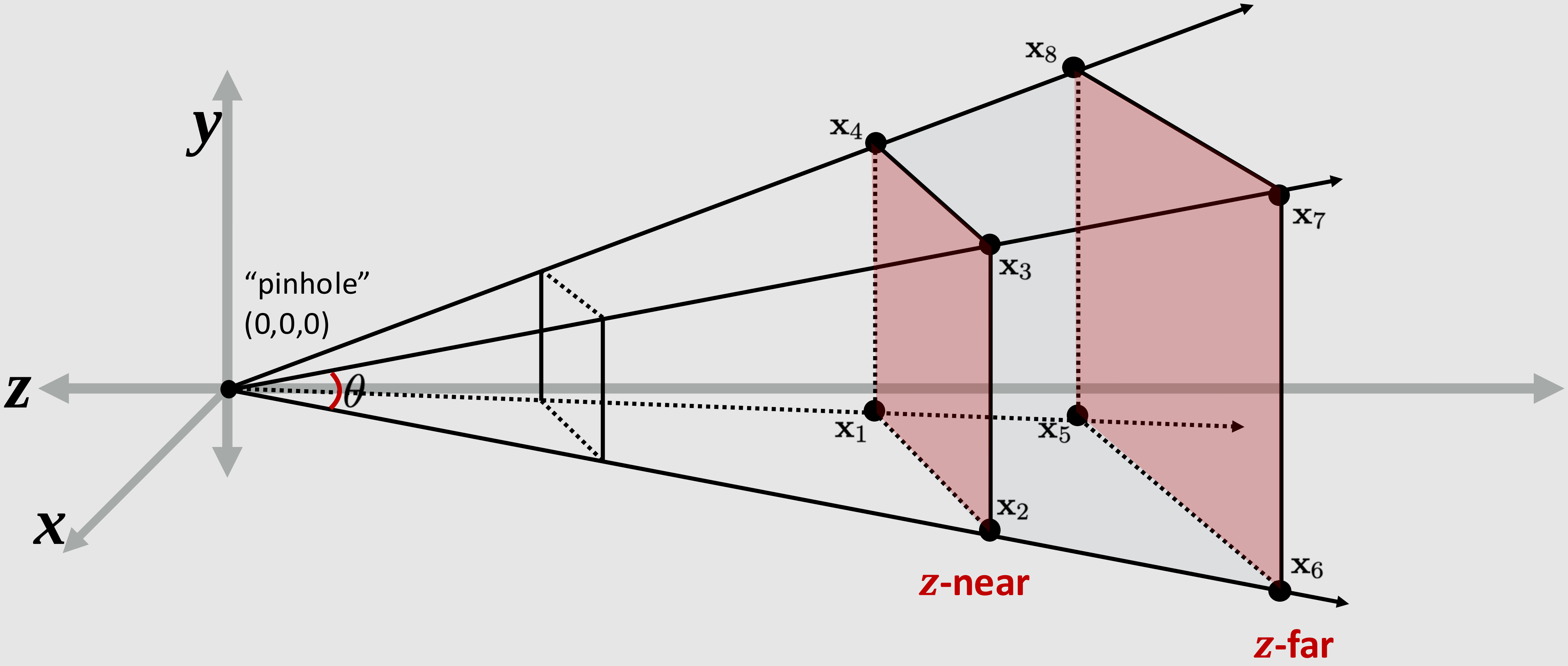


Use **Gram-Schmidt** to "pick" v and w. Then build a rotation matrix R and invert/transpose it to apply the transform

$$R = \begin{bmatrix} u_x & v_x & -w_x \\ u_y & v_y & -w_y \\ u_z & v_z & -w_z \end{bmatrix} \qquad R^{-1} = \begin{bmatrix} u_x & u_y & u_z \\ v_x & v_y & v_z \\ -w_x & -w_y & -w_z \end{bmatrix}$$

View Frustrum

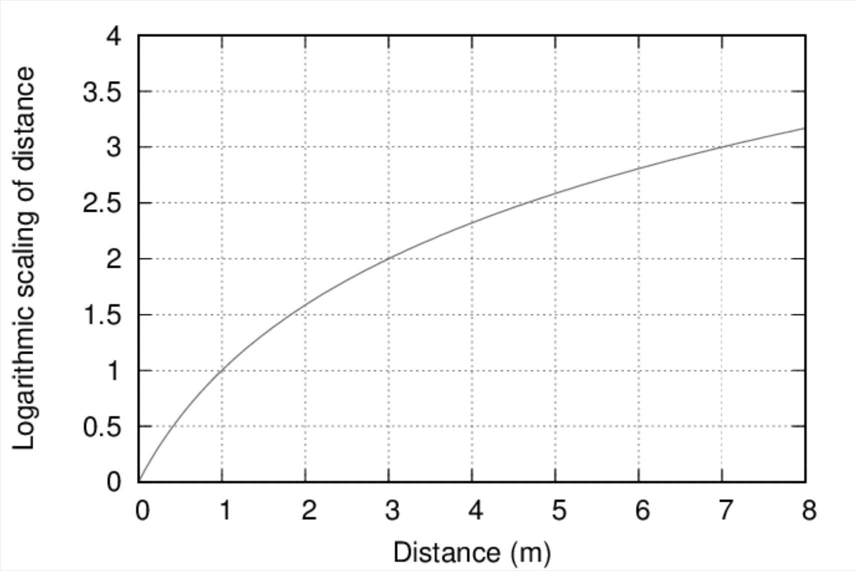
Also known as the “region the camera can see”

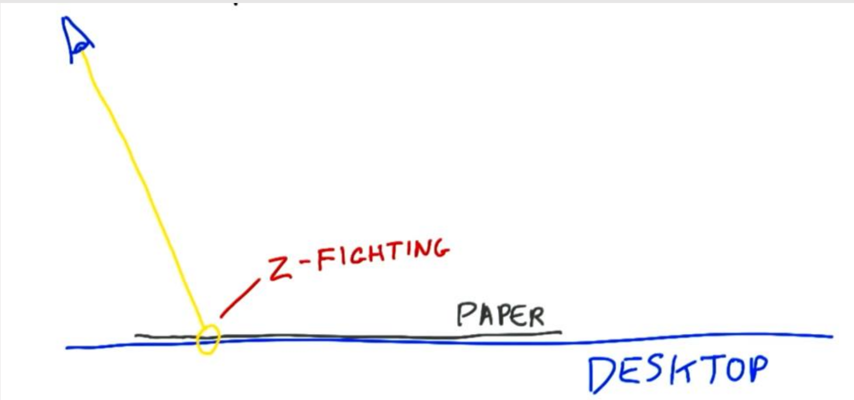


Q: Why is it important we have a z-near and z-far?

Logarithmic Distance

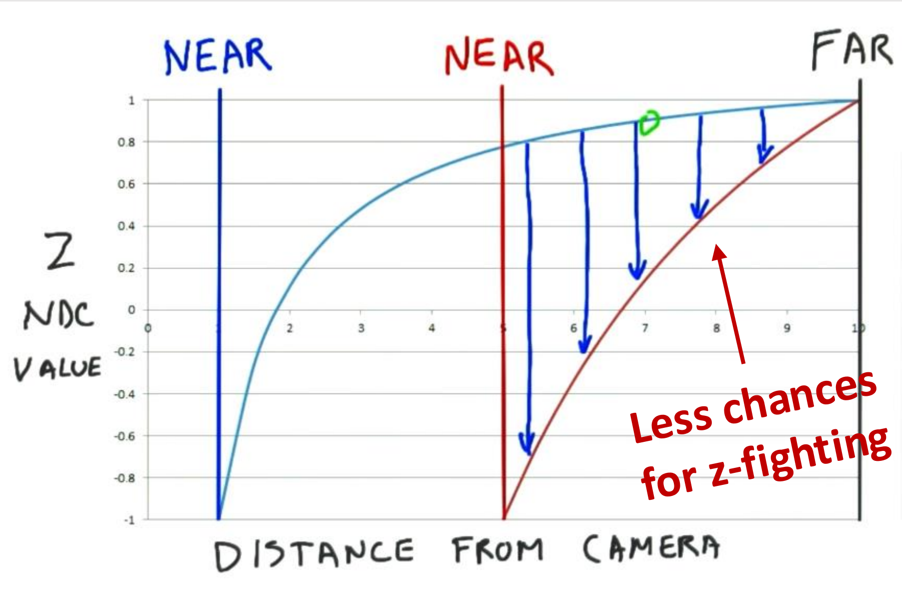
- Objects get smaller at a logarithmic rate as they move farther from our eyes
  - In this class, **eyes == cameras**
  - Little change in size for objects already far away as they get farther
- In computer graphics, we quantize everything:
  - Colors
  - Shapes
  - Depth
- Providing a fixed precision for depth (usually 32 bits) means objects very far away may share the same depth data
  - Limited representable depth values
  - Leads to unintentional clipping



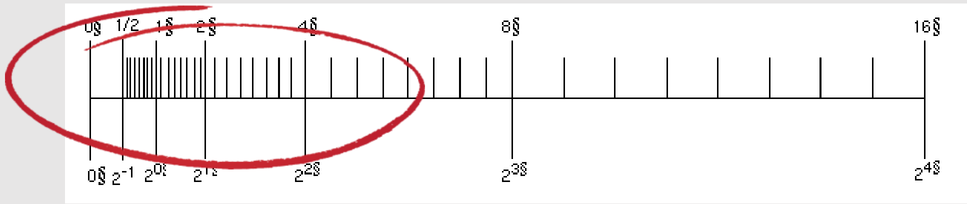


Near and Far Clipping (2015) Udacity

Near and Far Clipping Planes



Near and Far Clipping (2015) Udacity



floating point has more "resolution" near zero

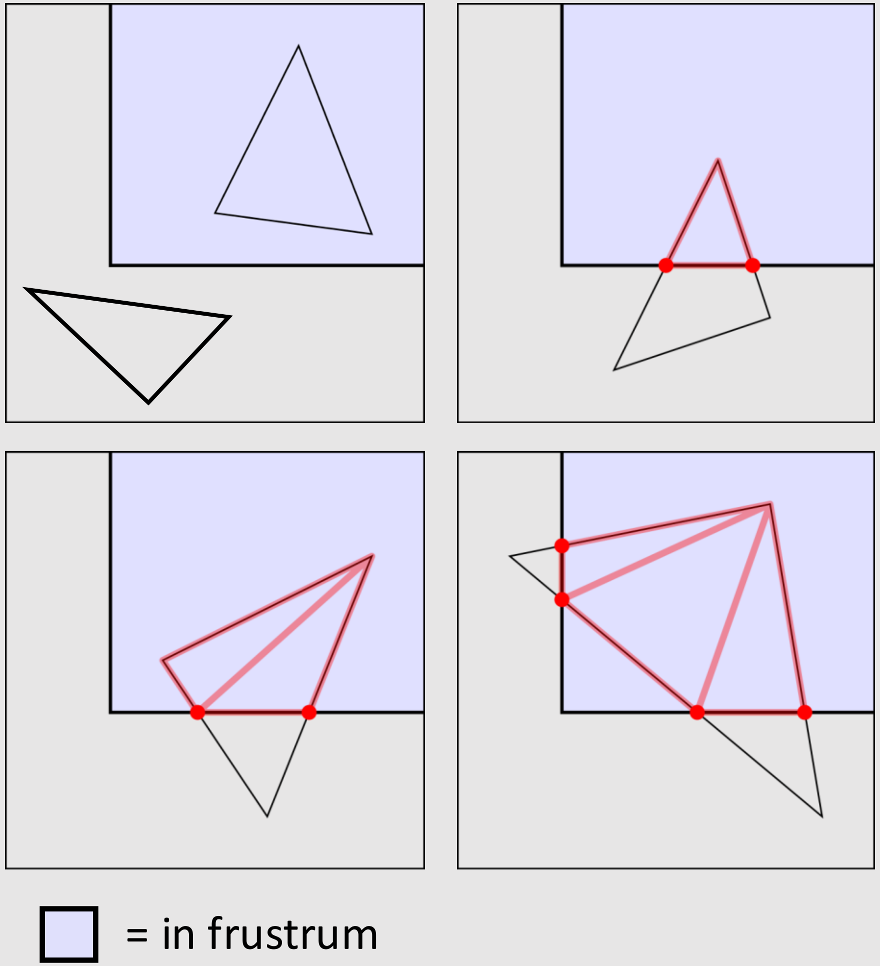
- **Idea:** set a smaller range for possible depth values
  - Min depth is the **near clipping plane**
  - Max depth is the **far clipping plane**
    - Logarithmic curve doesn't give many possible values for far objects…
- **Problem:** accidentally clip out objects important to our scene if range set too small
  - Near/Far clipping plane should encapsulate the most important objects closest/farthest to the camera
- **Advantage:** far clipping cuts out unimportant objects from your scene early in the pipeline
  - **Examples:** far-away trees in an already dense forest

Clipping

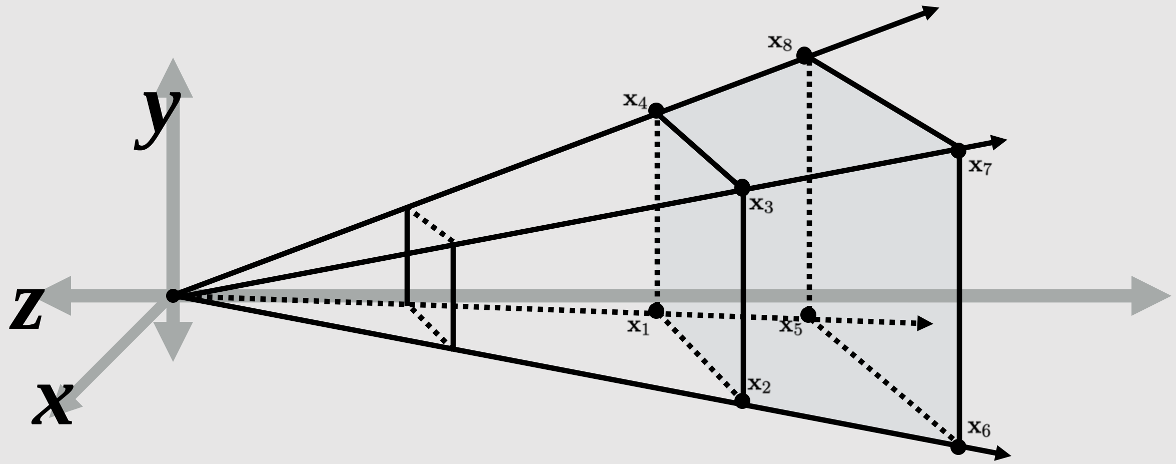
- **Clipping** eliminates triangles not visible to the camera (not in view frustum)
  - Don't waste time rasterizing primitives you can't see!
  - Discarding individual fragments is expensive
    - "Fine granularity"
  - Makes more sense to toss out whole primitives
    - "Coarse granularity"

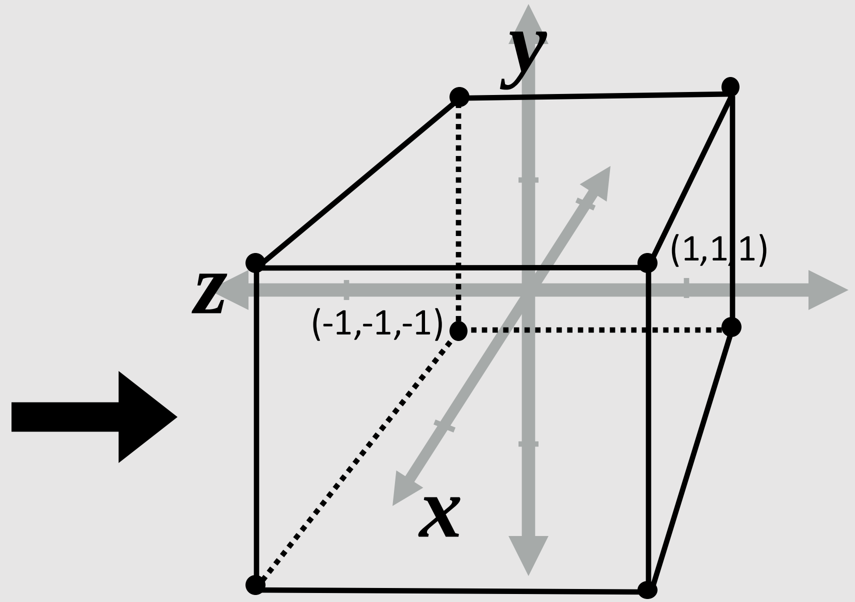
- What if a primitive is **partially clipped?**
  - Partially enclosed triangles are tessellated into smaller triangles in the frustrum

- If part of a triangle is outside the frustrum, it means at least one of its vertices are outside the frustrum
  - **Idea:** check if vertices in frustrum



Map Frustrum To Cube





l = left b = bottom n = near
r = right t = top f = far

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

$$\begin{aligned}
\mathbf{x}_1 &= \{l,b,n,1\} \\
\mathbf{x}_2 &= \{r,b,n,1\} \\
\mathbf{x}_3 &= \{r,t,n,1\} \\
\mathbf{x}_4 &= \{l,t,n,1\} \\
\mathbf{x}_5 &= \{l,b,f,1\} \\
\mathbf{x}_6 &= \{r,b,f,1\} \\
\mathbf{x}_7 &= \{r,t,f,1\} \\
\mathbf{x}_8 &= \{l,t,f,1\}
\end{aligned}$$

$$\begin{aligned}
\mathbf{y}_1 &= \{-1,-1,1,1\} \\
\mathbf{y}_2 &= \{1,-1,1,1\} \\
\mathbf{y}_3 &= \{1,1,1,1\} \\
\mathbf{y}_4 &= \{-1,1,1,1\} \\
\mathbf{y}_5 &= \{-1,-1,-1,1\} \\
\mathbf{y}_6 &= \{1,-1,-1,1\} \\
\mathbf{y}_7 &= \{1,1,-1,1\} \\
\mathbf{y}_8 &= \{-1,1,-1,1\}
\end{aligned}$$

Map Frustrum To Cube

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

[translate terms]

[scale terms]

- **Q:** why is the z-axis scalar term $\frac{2}{n-f}$?
  - Camera looks down –z axis, so we need to flip axis!

subtract the midpoint to center the coordinate

$$x - \frac{l+r}{2}$$

divide by the clipping range to normalize to [-0.5, 0.5]

$$\frac{x}{r-l} - \frac{l+r}{2(r-l)}$$

scale by 2 to expand range to [-1, 1]

$$\frac{2x}{r-l} - \frac{l+r}{r-l}$$

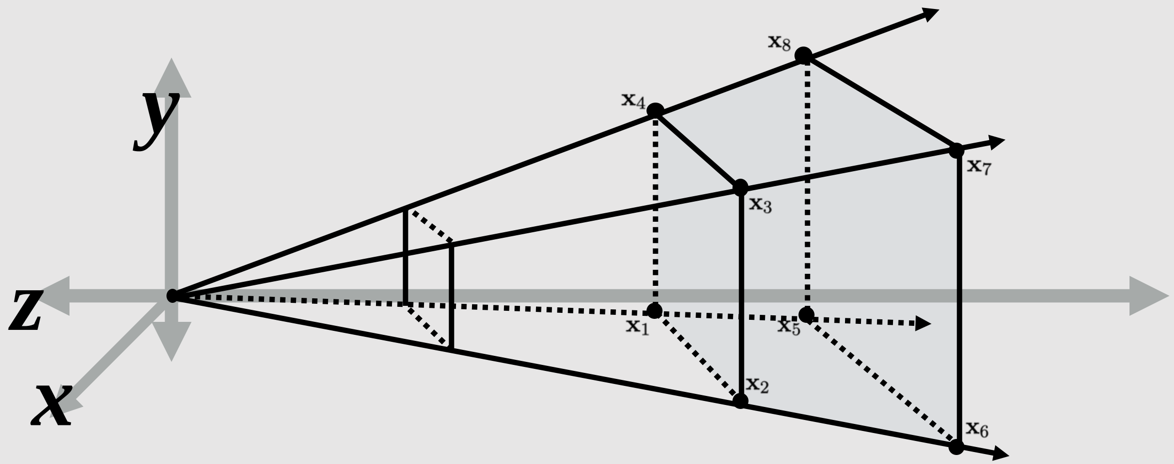
flip sign of second fraction to make translation additive

$$\frac{2}{r-l}x + \frac{l+r}{l-r}$$

Map A Harder Frustrum To Cube

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

With perspective projection, we end up dividing out the z coordinate.
Full perspective matrix takes geometry of view frustum into account:



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

Map A Harder Frustrum To Cube

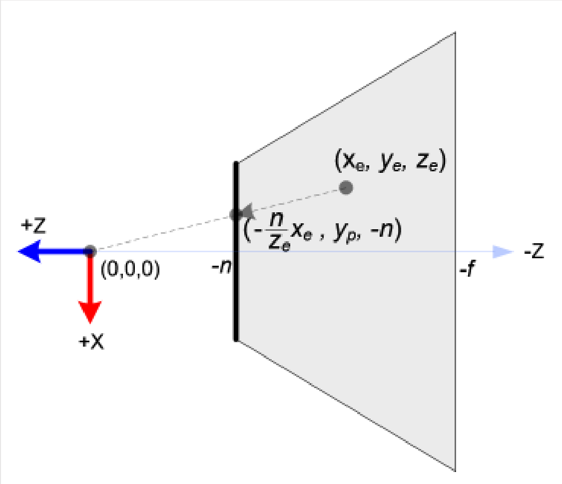
$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

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

Same idea as above: w divides out the depth, so we set it equal to the depth z
Small difference: we are looking down the –z axis, so we set $w = -z$

Map A Harder Frustrum To Cube

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



the projection of x linearly approaches 0 as it is projected closer to the camera

$$\frac{n}{-z}x$$

use the same equation as before, subbing in new projection

$$\frac{2(\frac{n}{-z}x)}{r-l} + \frac{r+l}{l-r}$$

simplify first term, multiply z/z to second term

$$\frac{2n}{(r-l)(-z)}x + \frac{(r+l)z}{(r-l)(-z)}$$

extract $-z$ from denominator

$$\frac{\left(\frac{2n}{(r-l)}x + \frac{(r+l)}{(r-l)}z\right)}{-z}$$

By setting $w = -z$, we will do this last division step when dividing out the depth

**see http://www.songho.ca/opengl/gl_projectionmatrix.html for a full derivation

Map A Harder Frustrum To Cube

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

$$\begin{aligned} \mathbf{x}_1 &= \{l,b,n,1\} \\ \mathbf{x}_2 &= \{r,b,n,1\} \\ \mathbf{x}_3 &= \{r,t,n,1\} \\ \mathbf{x}_4 &= \{l,t,n,1\} \\ \mathbf{x}_5 &= \{l,b,f,1\} \\ \mathbf{x}_6 &= \{r,b,f,1\} \\ \mathbf{x}_7 &= \{r,t,f,1\} \\ \mathbf{x}_8 &= \{l,t,f,1\} \end{aligned}$$

$$\begin{aligned} \mathbf{y}_1 &= \{-1,-1,1,1\} \\ \mathbf{y}_2 &= \{1,-1,1,1\} \\ \mathbf{y}_3 &= \{1,1,1,1\} \\ \mathbf{y}_4 &= \{-1,1,1,1\} \\ \mathbf{y}_5 &= \{-1,-1,-1,1\} \\ \mathbf{y}_6 &= \{1,-1,-1,1\} \\ \mathbf{y}_7 &= \{1,1,-1,1\} \\ \mathbf{y}_8 &= \{-1,1,-1,1\} \end{aligned}$$

the final normalized z_n is a function of the initial z and w, divided by the negative depth (projection):

$$z_n = \frac{Az + Bw}{-z}$$

to solve for A and B, solve for the fact that -n maps to -1 and -f maps to 1**

$$\frac{-An + B}{n} = -1$$

$$\frac{-Af + B}{f} = 1$$

2 equations, 2 unknowns, use your favorite linear solver

$$A = \frac{-(f+n)}{f-n}$$

$$B = \frac{-2fn}{f-n}$$

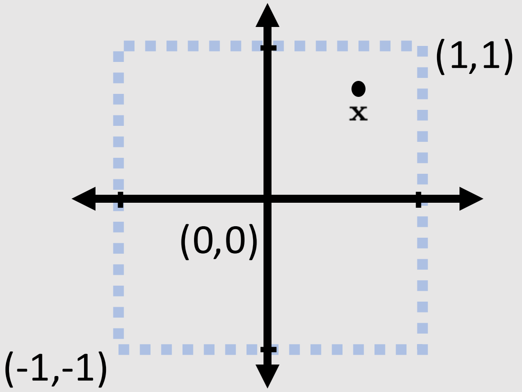
**remember w is a homogeneous coordinate, so w=1

Screen Transform

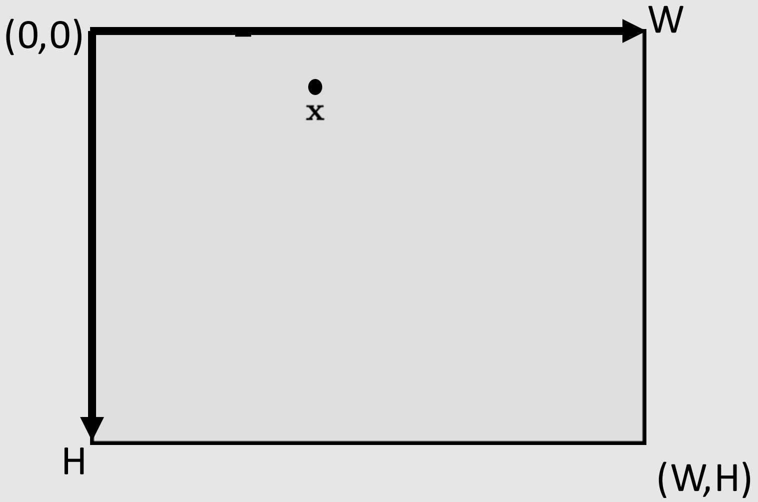
- We now have a way of going from camera view frustrum to normalized screen space:
  - Apply projection matrix
  - Divide out w-coordinate (set to –z)

- Last transform: image space
  - Take points from [-1,1] x [-1,1] to a W x H pixel image

- Step 1: reflect about x-axis
- Step 2: translate by (1,1)
- Step 3: scale by (W/2, H/2)

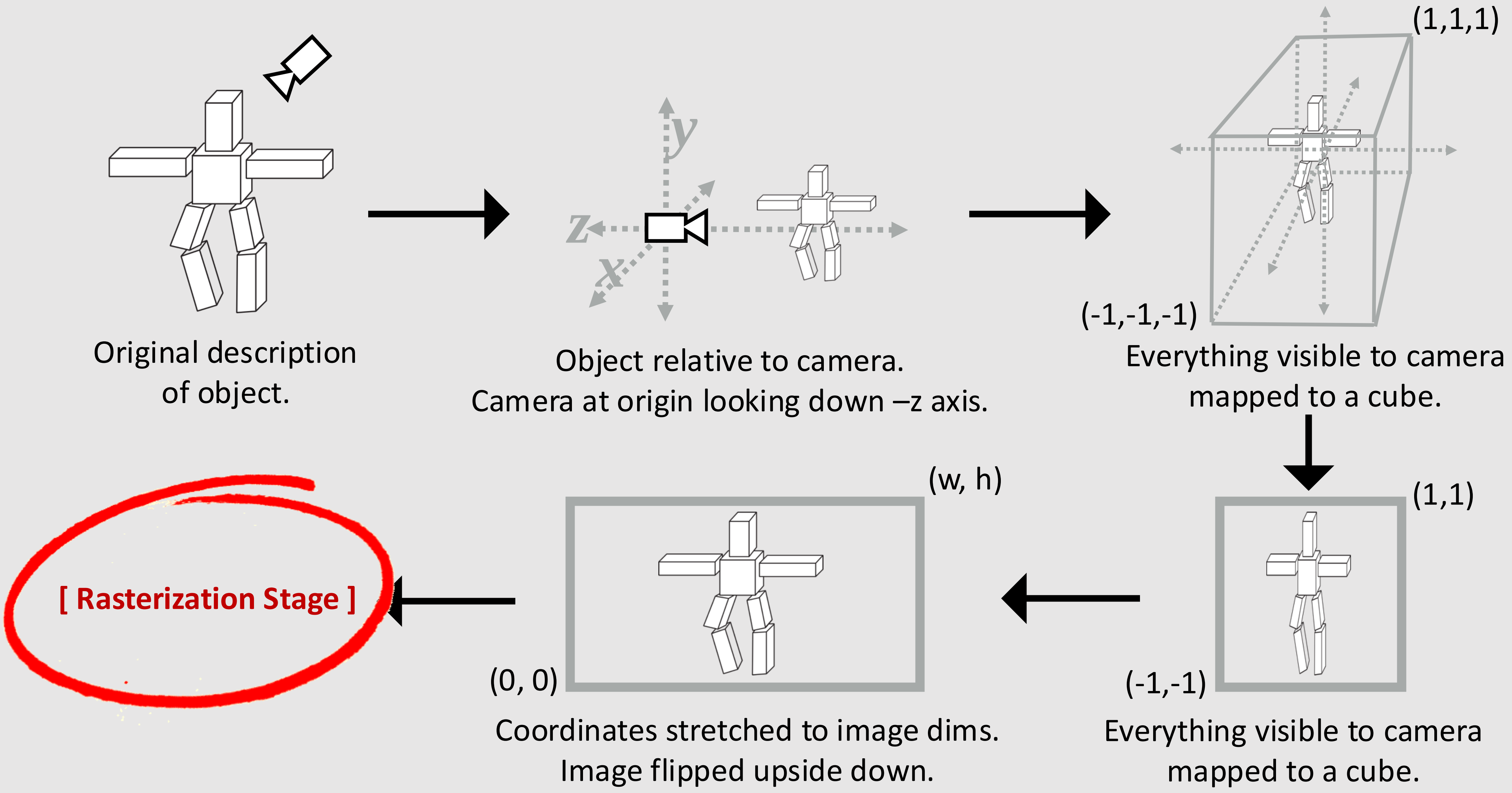


[normalized coordinates]



[image coordinates]

Perspective Projection



Rasterization

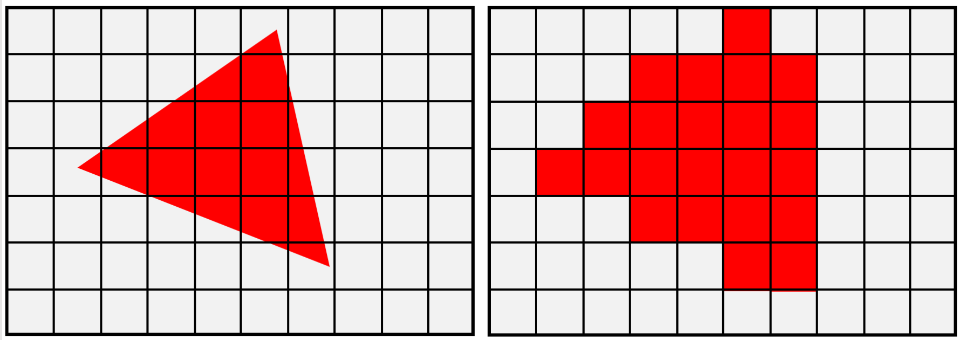
- **Problem:** displays don't know what a triangle is or how to display one
  - But they do know how to display a buffer of pixels!
- **Goal:** convert draw instructions into an image of pixels to show on the display
  - Example:

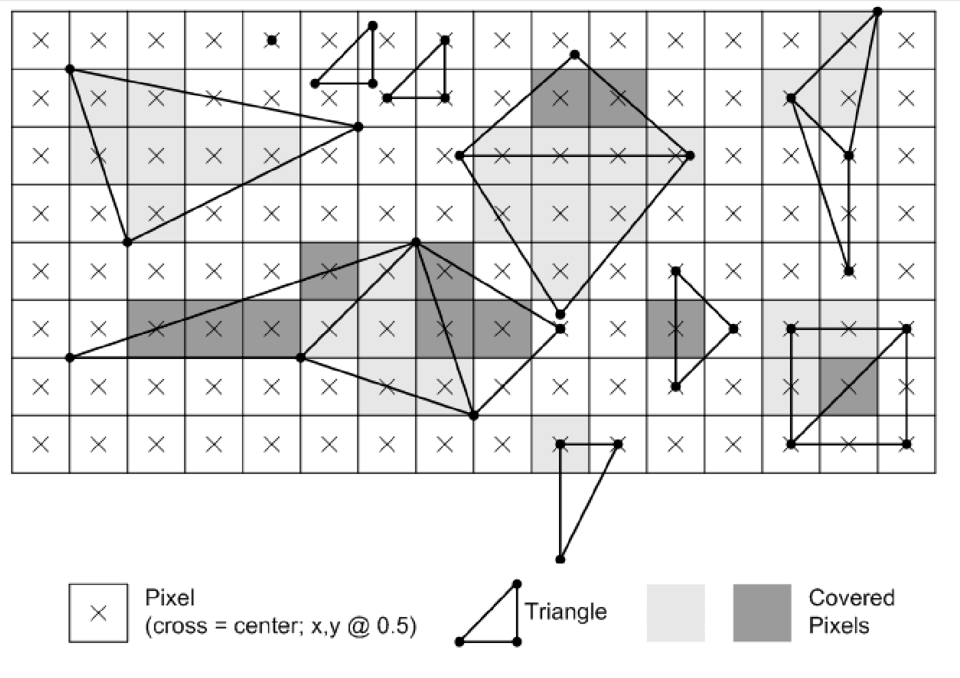
color

<polygon fill="#ED18ED"
points="464.781,631.819 478.417,309.091 471.599,642.045 "/>

3 x (2D points)

  - The above is a valid svg instruction
- Requires turning shapes into pixels
  - Need to check which shapes overlap which pixels



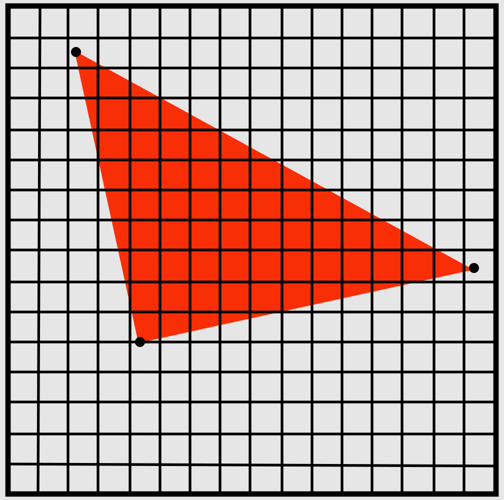


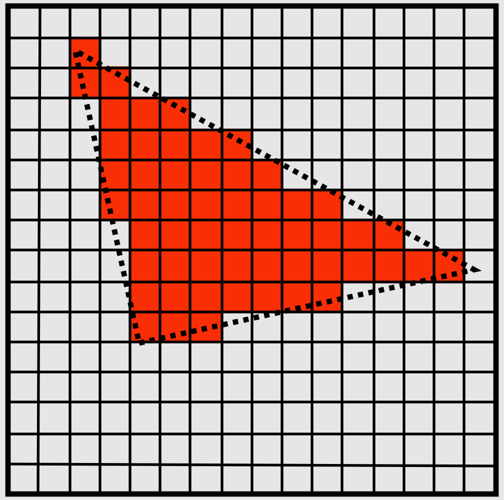
Direct3D Documentation (2020) Microsoft

Rasterization

For Each **Triangle**:
    For Each **Pixel**:
        If **Pixel** In **Triangle**:
            Pixel Color = Triangle Color

- How to check if a pixel is inside a triangle?
- A pixel is a little square, check if the square exists inside the triangle**
    - Expensive/hard to compute!
- A pixel is a point, check if the point exists inside the triangle
    - Put the point at the pixel's center
    - We will refer to these as half-integer coordinates (Ex: [1.5, 4.5])





**"*A pixel is not a little square*" Alvy Ray Smith

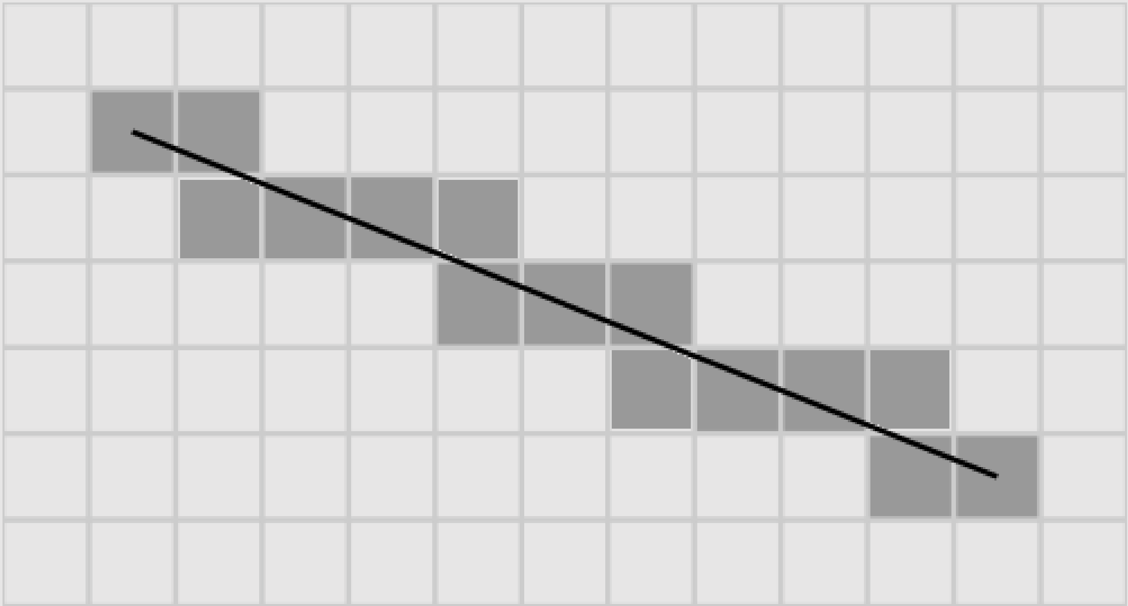
- ~~Perspective Projection~~
- Drawing a Line
- Drawing a Triangle
- Supersampling

Before that,
Let’s learn how to draw a line!

Surely it can’t be difficult…it’s just a line

Introduction To The Line

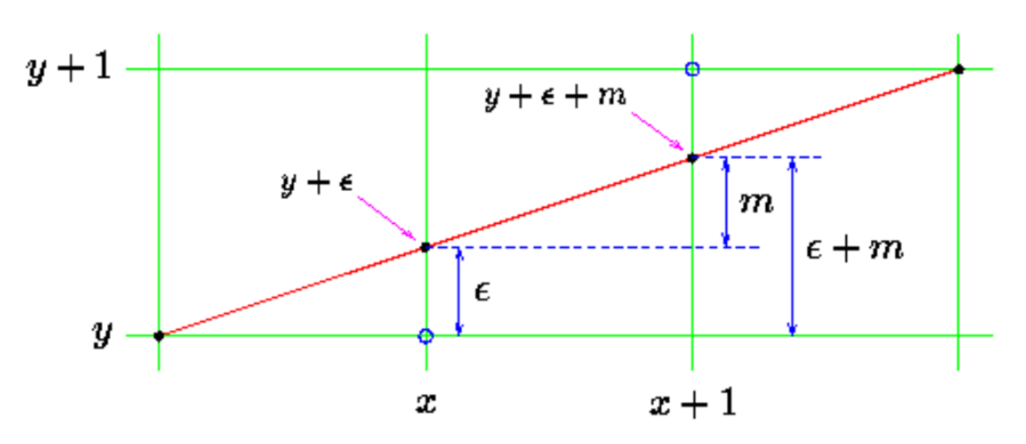
- A line is defined by $(x_1, y_1), (x_2, y_2)$
  - Slope given as $\mathrm{m} = \frac{y_2 - y_1}{x_2 - x_1}$
- What does it mean for a line to overlap a pixel?
  - A pixel is just a point
  - A line has no thickness
    - Neither have a notion of area
- Instead, we will reinterpret pixels as squares
  - A pixel lights up if the line intersects it
    - Checking if a line intersects a pixel can be expensive!
- Find a linear algorithm ~O(n) where n is the number of output fragments
  - Everything we check should be everything in the output



The Bresenham Line Algorithm

- Consider the case when m is in range $[0,1]$
  - Implies $\Delta x \geq \Delta y$

- We will traverse up the x-axis
  - Each step of x we take, decide if we keep y the same or move y up one step
    - Since $0 < m < 1$, a positive move in x causes a positive move in y



[pseudocode]

Ensure the x-coordinate of (x_1, y_1) is smaller
Let y' be our current vertical component along the line
Let y be the initial y_1
For each x value in range $[x_1, x_2]$ with step 1:
    Shade (x, y)
    Add m to y' (if x takes step 1, y' takes step m)
    If the new y' is closer to the row of pixels above:
        Add 1 to y

[code]

If $x_1 > x_2$:
    $\text{Swap}(x_1, x_2)$, $\text{Swap}(y_1, y_2)$
$\varepsilon \leftarrow 0$, $y \leftarrow y_1$
For $x \leftarrow x_1$ to x_2 do:
    Shade(x, y)
    If $(|\varepsilon + m| > 0.5)$:
        $\varepsilon \leftarrow \varepsilon + m - 1$, $y \leftarrow y + 1$
    Else:
        $\varepsilon \leftarrow \varepsilon + m$

The Bresenham Line Algorithm

- What if m is in range $[0,1]$?

$\varepsilon \leftarrow 0, \quad y \leftarrow y_1$
For $x \leftarrow x_1$ to x_2 do:
  Shade(x, y)
  If $(|\varepsilon + m| > 0.5)$:
    $\varepsilon \leftarrow \varepsilon + m - 1, \quad y \leftarrow y + 1$
  Else:
    $\varepsilon \leftarrow \varepsilon + m$

- What if m is in range $[-1,0]$?

$\varepsilon \leftarrow 0, \quad y \leftarrow y_1$
For $x \leftarrow x_1$ to x_2 do:
  Shade(x, y)
  If $(|\varepsilon + m| > 0.5)$:
    $\varepsilon \leftarrow \varepsilon + m + 1, \quad y \leftarrow y - 1$
  Else:
    $\varepsilon \leftarrow \varepsilon + m$

- What if $m > 1$?

$\varepsilon \leftarrow 0, \quad x \leftarrow x_1$
For $y \leftarrow y_1$ to y_2 do:
  Shade(x, y)
  If $(|\varepsilon + 1/m| > 0.5)$:
    $\varepsilon \leftarrow \varepsilon + 1/m - 1, \quad x \leftarrow x + 1$
  Else:
    $\varepsilon \leftarrow \varepsilon + 1/m$

- What if $m < -1$?

$\varepsilon \leftarrow 0, \quad x \leftarrow x_1$
For $y \leftarrow y_1$ to y_2 do:
  Shade(x, y)
  If $(|\varepsilon + 1/m| > 0.5)$:
    $\varepsilon \leftarrow \varepsilon + 1/m + 1, \quad x \leftarrow x - 1$
  Else:
    $\varepsilon \leftarrow \varepsilon + 1/m$

**When traversing x-axis, x1 must be smaller. When traversing y-axis, y1 must be smaller

That’s kinda complicated…
Can we make it easier somehow?

The [Nicer] Bresenham Line Algorithm

$a = < x_1, y_1 >, \qquad b = < x_2, y_2 >$
$\Delta x \leftarrow |x_2 - x_1|, \qquad \Delta y \leftarrow |y_2 - y_1|$

setup coordinates

If ($\Delta x > \Delta y$):
    $i \leftarrow 0, \quad j \leftarrow 1$
If ($\Delta x < \Delta y$):
    $i \leftarrow 1, \quad j \leftarrow 0$

compute the longer axis i
and the shorter axis j

If ($a_i > b_i$):
    $swap(a, b)$

the starting coordinate should be the smaller value along the longer axis

$t_1 \leftarrow floor(a_i), \qquad t_2 \leftarrow floor(b_i)$

compute long axis bounds

For $u \leftarrow t_1$ to t_2 do:
    $w \leftarrow \frac{(u+0.5)-a_i}{(b_i-a_i)}$
    $v \leftarrow w * \left(b_j - a_j\right) + a_j$

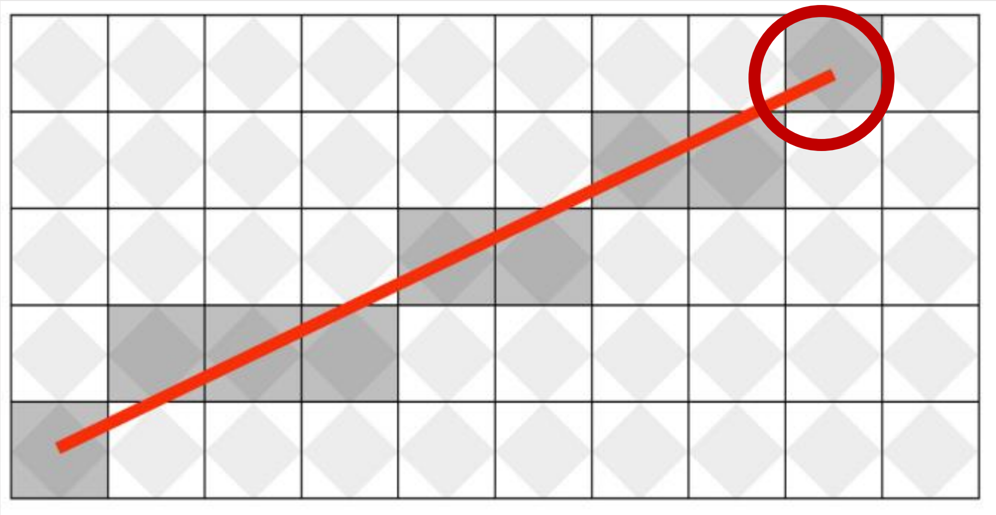
    Shade($floor(u) + 0.5, \; floor(v) + 0.5$)

for each step taken along the longer axis, compute the percent distance traveled w and project that percentage onto the shorter axis. Then convert to half-integer coordinates

Introduction To The Line

- Bresenham algorithm only works if both the start and end coordinates lie on half-integer coordinates
- Instead we will consider a line to intersect a pixel if the line intersects the diamond inside the pixel
    - $|x - p_x| + |y - p_y| < \frac{1}{2}$
        - Checks if point (x, y) lies in the diamond of pixel p
- Still the same idea as before! The only difference is that we need to check if the endpoints correctly intersect the last pixels

In OpenGL/Scotty3D, line needs to fully go through diamond!



The [Even Nicer] Bresenham Line Algorithm

$a = < x_1, y_1 >, \quad b = < x_2, y_2 >$
$\Delta x \leftarrow |x_2 - x_1|, \quad \Delta y \leftarrow |y_2 - y_1|$

If ($\Delta x > \Delta y$):
    $i \leftarrow 0, \quad j \leftarrow 1$
If ($\Delta x < \Delta y$):
    $i \leftarrow 1, \quad j \leftarrow 0$

If ($a_i > b_i$):
    $swap(a, b)$

$t_1 \leftarrow floor(a_i), \quad t_2 \leftarrow floor(b_i)$

For $u \leftarrow t_1$ to t_2 do:
    $w \leftarrow \frac{(u+0.5)-a_i}{(b_i-a_i)}$
    $v \leftarrow w * (b_j - a_j) + a_j$

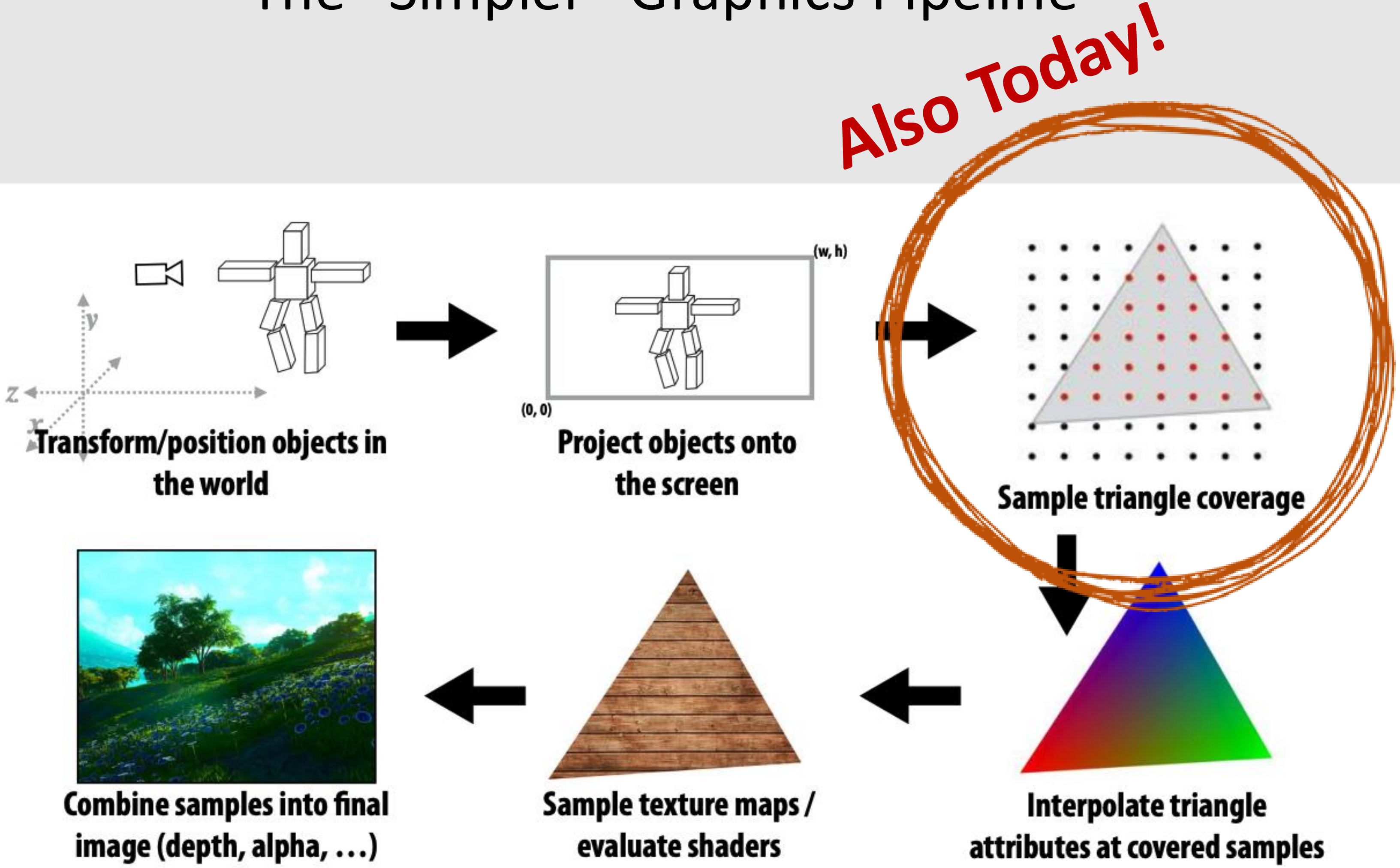
    Shade($floor(u) + 0.5, \ floor(v) + 0.5$)

TODO: fix t_1 and t_2 to properly account for OR discard the two edge fragments if the endpoints a and b are inside the 'diamond' of the edge fragments

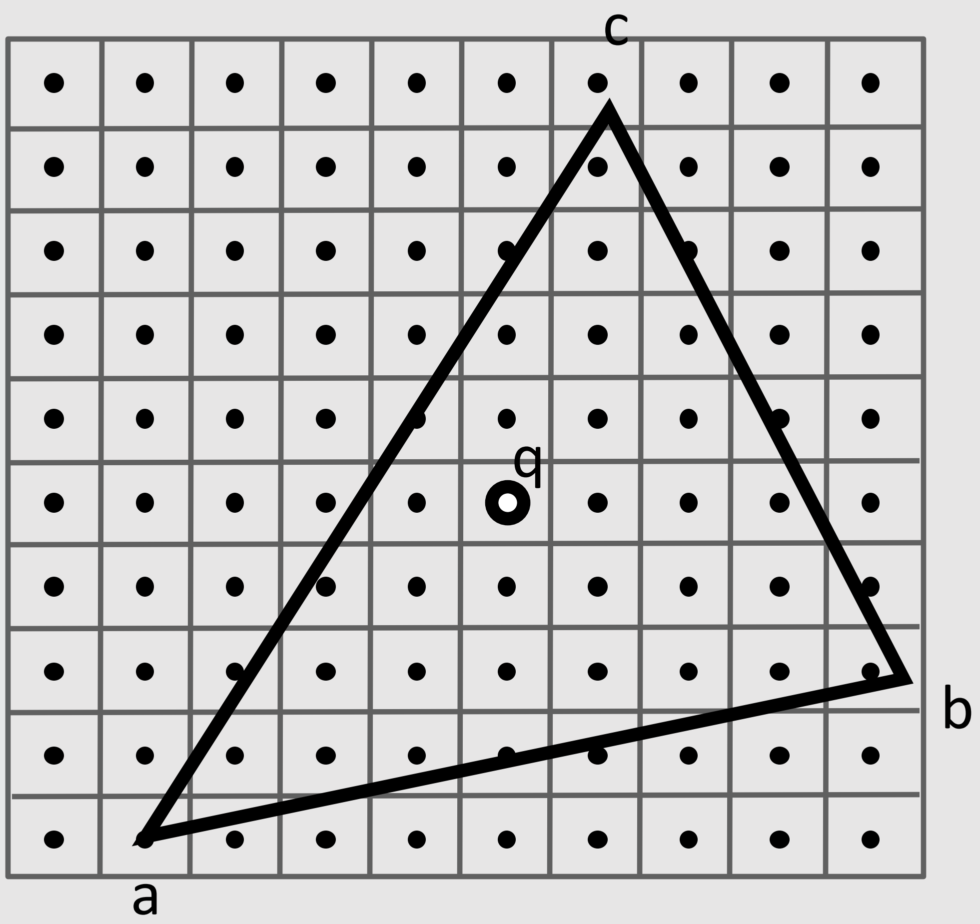
Remember: $|x - p_x| + |y - p_y| < \frac{1}{2}$

- ~~Perspective Projection~~
- ~~Drawing a Line~~
- Drawing a Triangle
- Supersampling

The "Simpler" Graphics Pipeline

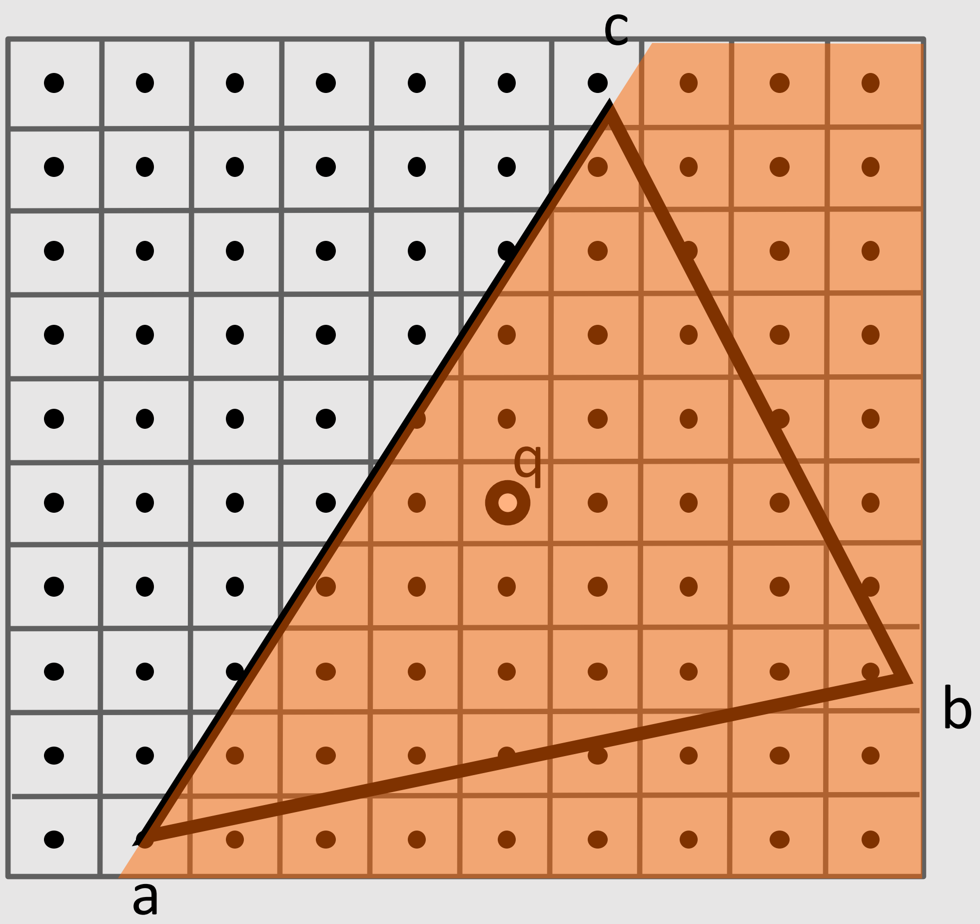


Point-In-Triangle Test



- Which points do we check?
  - **Idea 1:** check all points q in the image
    - For large images (1080p), we're checking hundreds of thousands of points per triangle!
  - **Idea 2:** check all points q in the bounding box of the triangle:
    - $x_{min} = \min(a_x, b_x, c_x)$
    - $y_{min} = \min(a_y, b_y, c_y)$
    - $x_{max} = \max(a_x, b_x, c_x)$
    - $y_{max} = \max(a_y, b_y, c_y)$
- How to check if a point is inside a triangle?

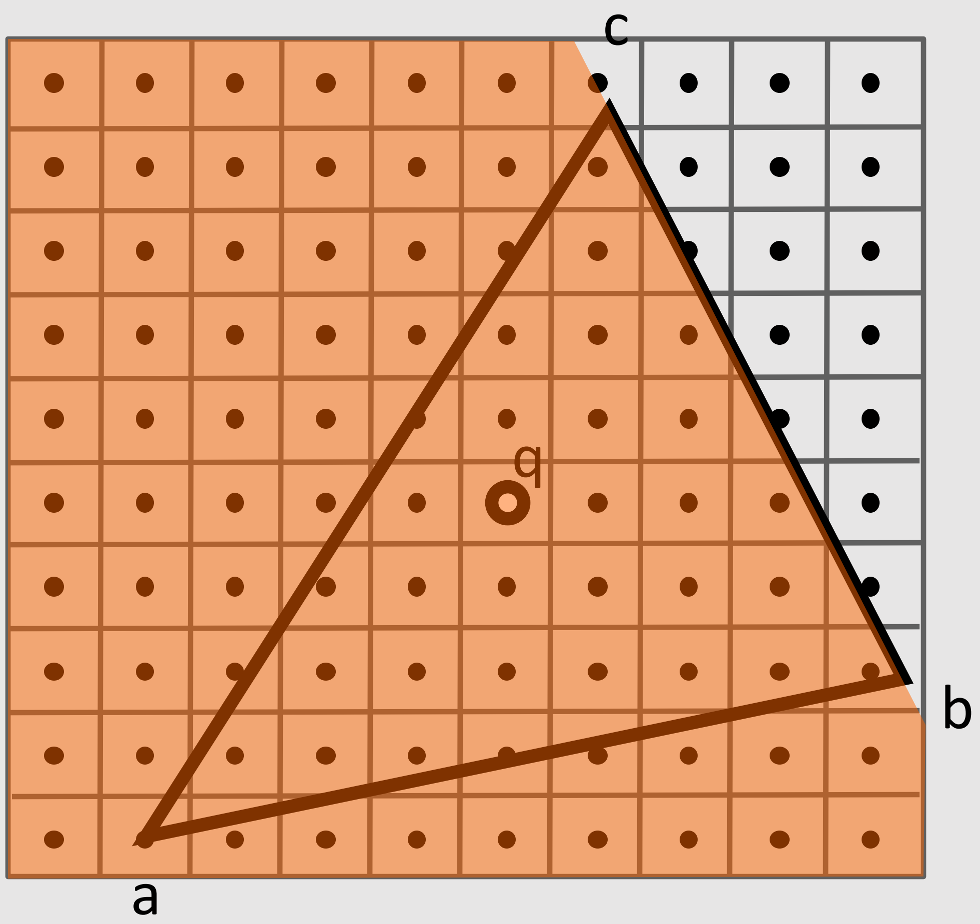
Point-In-Triangle Test



- How to check if a point is inside a triangle?
- Check that q is on the b side of $\overrightarrow{ac}$

$$\left(\overrightarrow{ac} \times \overrightarrow{ab}\right) \cdot \left(\overrightarrow{ac} \times \overrightarrow{aq}\right) > 0$$

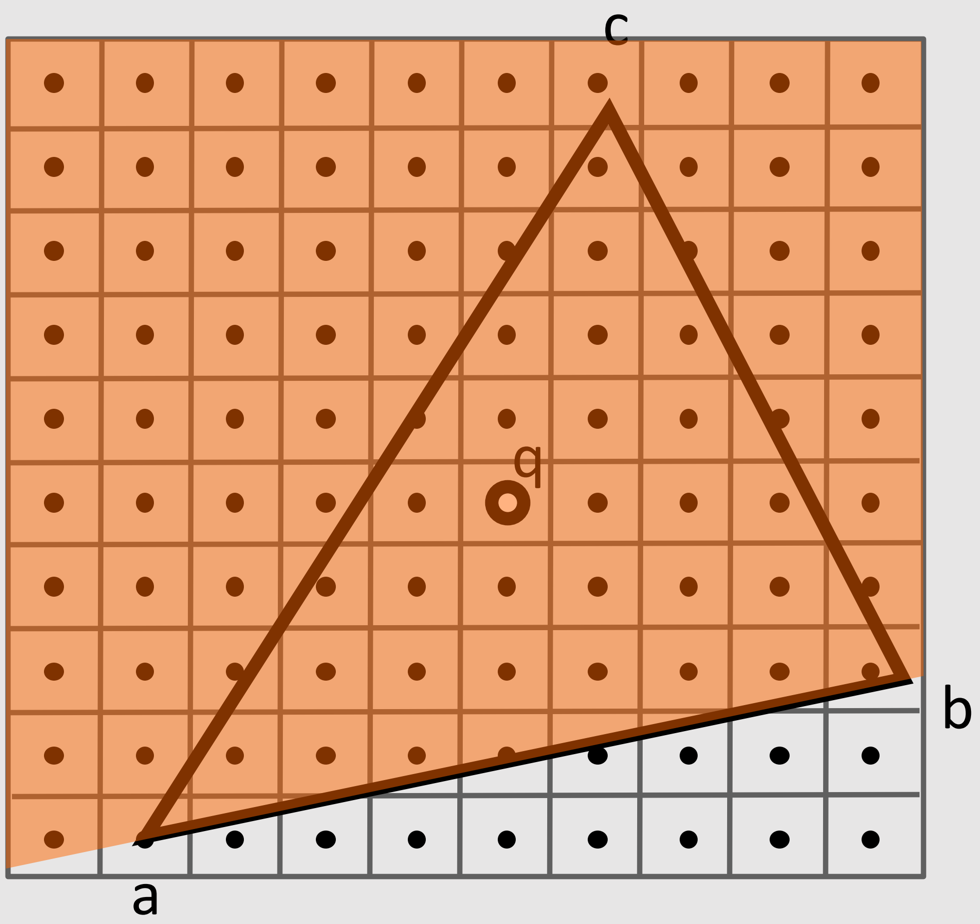
Point-In-Triangle Test



- How to check if a point is inside a triangle?
- Check that q is on the a side of $\overrightarrow{cb}$

$$(\overrightarrow{cb} \times \overrightarrow{ca}) \cdot (\overrightarrow{cb} \times \overrightarrow{cq}) > 0$$

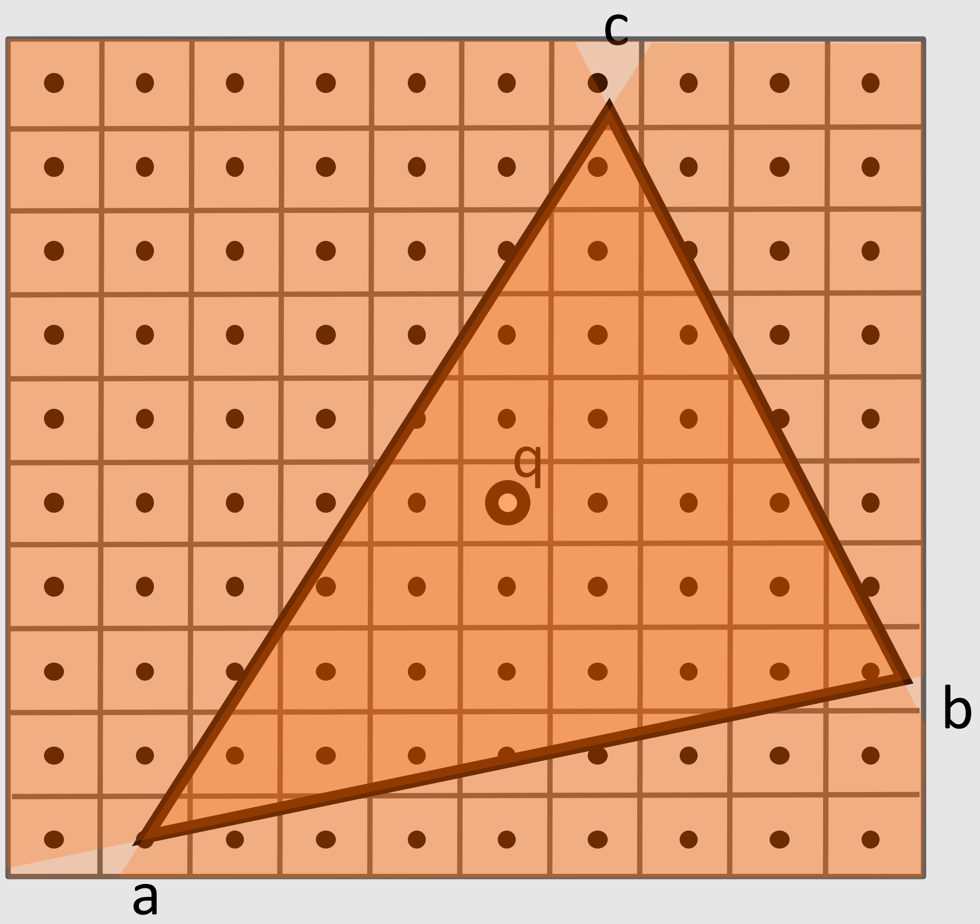
Point-In-Triangle Test



- How to check if a point is inside a triangle?
- Check that q is on the c side of $\overrightarrow{bc}$

$$(\overrightarrow{ba} \times \overrightarrow{bc}) \cdot (\overrightarrow{ba} \times \overrightarrow{bq}) > 0$$

Point-In-Triangle Test



- How to check if a point is inside a triangle?

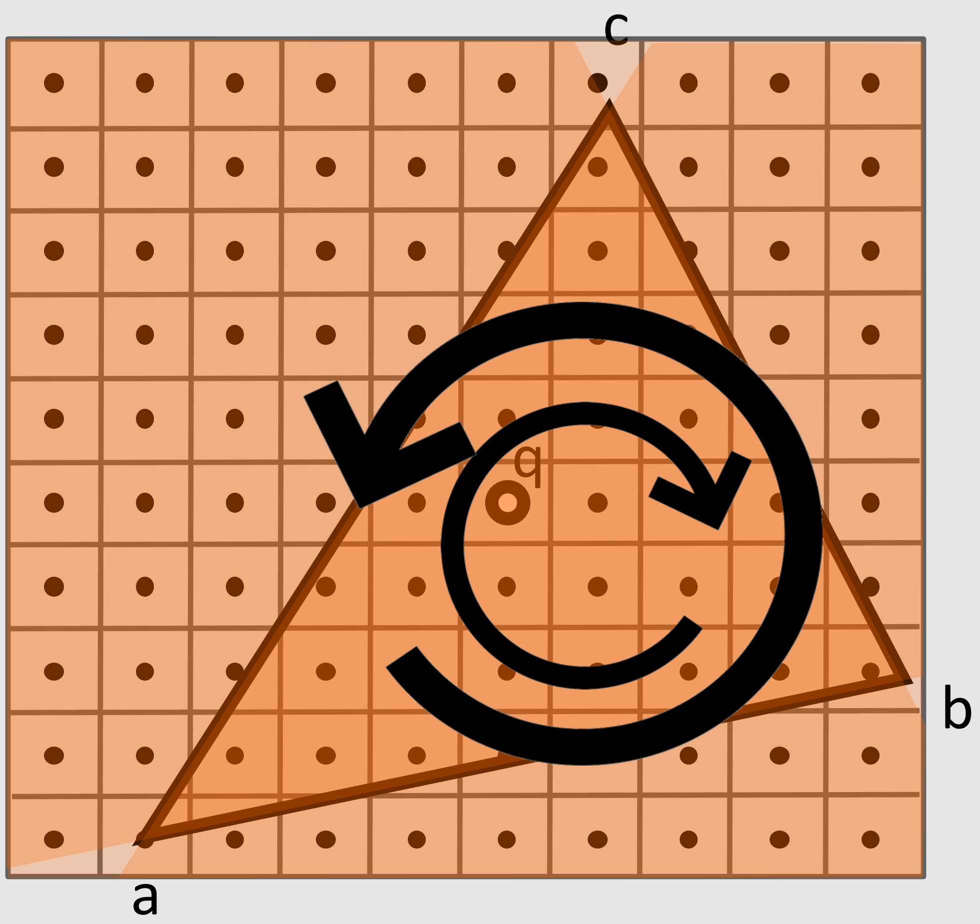
$$\left(\overrightarrow{ac} \times \overrightarrow{ab}\right) \cdot \left(\overrightarrow{ac} \times \overrightarrow{aq}\right) > 0 \;\&\&$$
$$\left(\overrightarrow{cb} \times \overrightarrow{ca}\right) \cdot \left(\overrightarrow{cb} \times \overrightarrow{cq}\right) > 0 \;\&\&$$
$$\left(\overrightarrow{ba} \times \overrightarrow{bc}\right) \cdot \left(\overrightarrow{ba} \times \overrightarrow{bq}\right) > 0$$

- What if b and c were swapped?

$$\left(\overrightarrow{ab} \times \overrightarrow{ac}\right) \cdot \left(\overrightarrow{ac} \times \overrightarrow{aq}\right) < 0$$

- Orientation matters!

Point-In-Triangle Test



- **Measurements must all either be positive or negative** for point to be in triangle

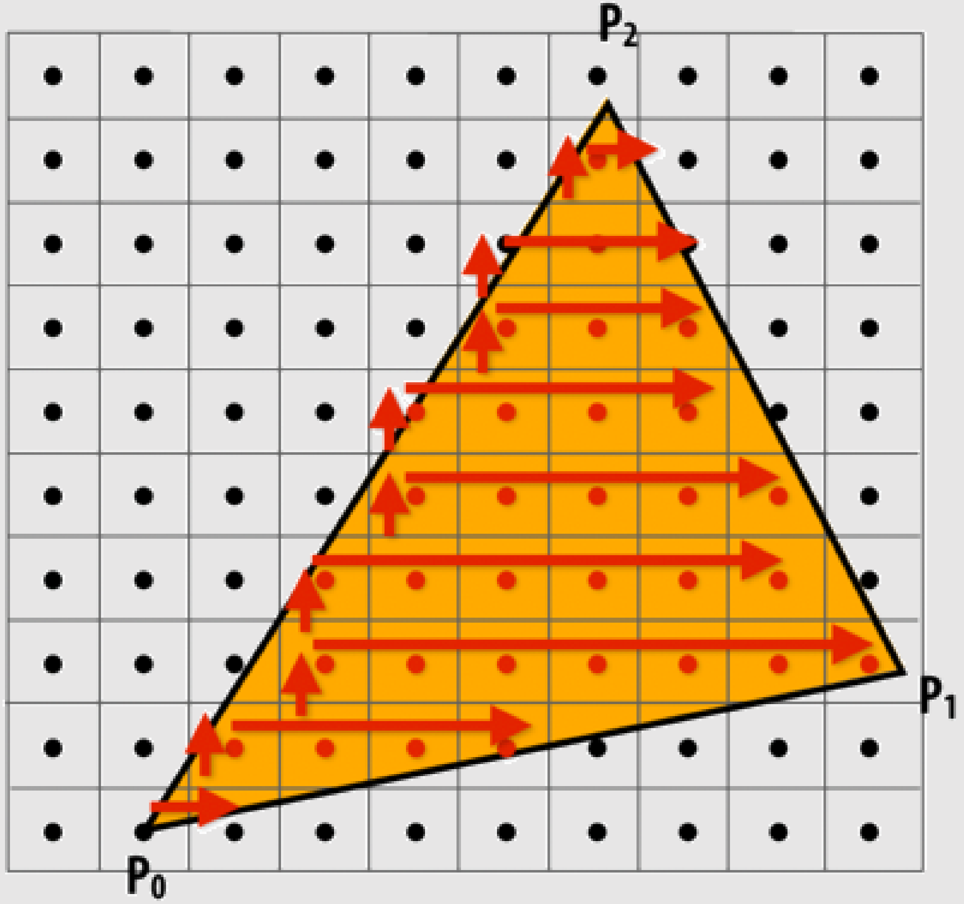
$$\left(\overrightarrow{ac} \times \overrightarrow{ab}\right) \cdot \left(\overrightarrow{ac} \times \overrightarrow{aq}\right) > 0 \;\&\&$$
$$\left(\overrightarrow{cb} \times \overrightarrow{ca}\right) \cdot \left(\overrightarrow{cb} \times \overrightarrow{cq}\right) > 0 \;\&\&$$
$$\left(\overrightarrow{ba} \times \overrightarrow{bc}\right) \cdot \left(\overrightarrow{ba} \times \overrightarrow{bq}\right) > 0$$

OR

$$\left(\overrightarrow{ab} \times \overrightarrow{ac}\right) \cdot \left(\overrightarrow{ac} \times \overrightarrow{aq}\right) < 0 \;\&\&$$
$$\left(\overrightarrow{ca} \times \overrightarrow{cb}\right) \cdot \left(\overrightarrow{cb} \times \overrightarrow{cq}\right) < 0 \;\&\&$$
$$\left(\overrightarrow{bc} \times \overrightarrow{ba}\right) \cdot \left(\overrightarrow{ba} \times \overrightarrow{bq}\right) < 0$$

- Orientation no longer matters
  - Just be consistent!

Incremental Triangle Traversal



$P_i = (x_i/w_i \; y_i/w_i \; z_i/w_i) = (X_i \; Y_i \; Z_i)$

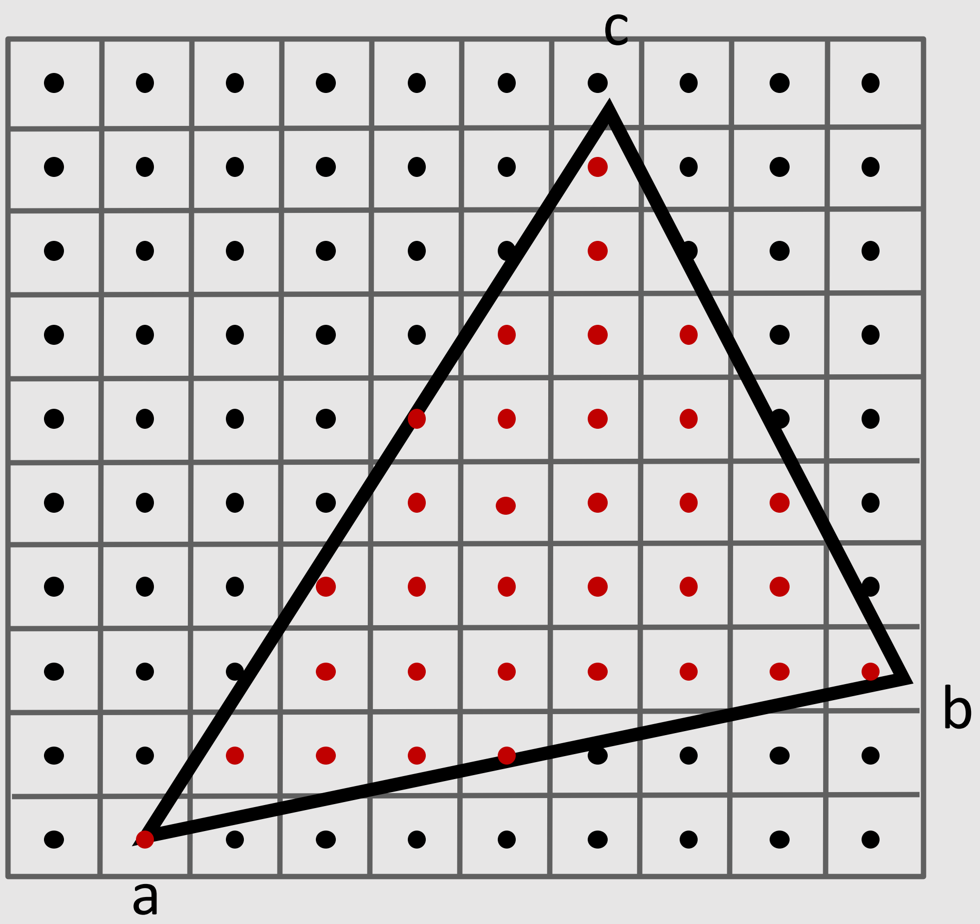
$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

$E_i(x, y) = (x - X_i)dY_i - (y - Y_i)dX_i$

$E_i(x, y) = 0$: point on edge
$E_i(x, y) > 0$: point outside edge
$E_i(x, y) < 0$: point inside edge

$dE_i(x+1, y) = E_i(x, y) + dY_i$
$dE_i(x, y+1) = E_i(x, y) + dX_i$

Parallel Coverage Tests

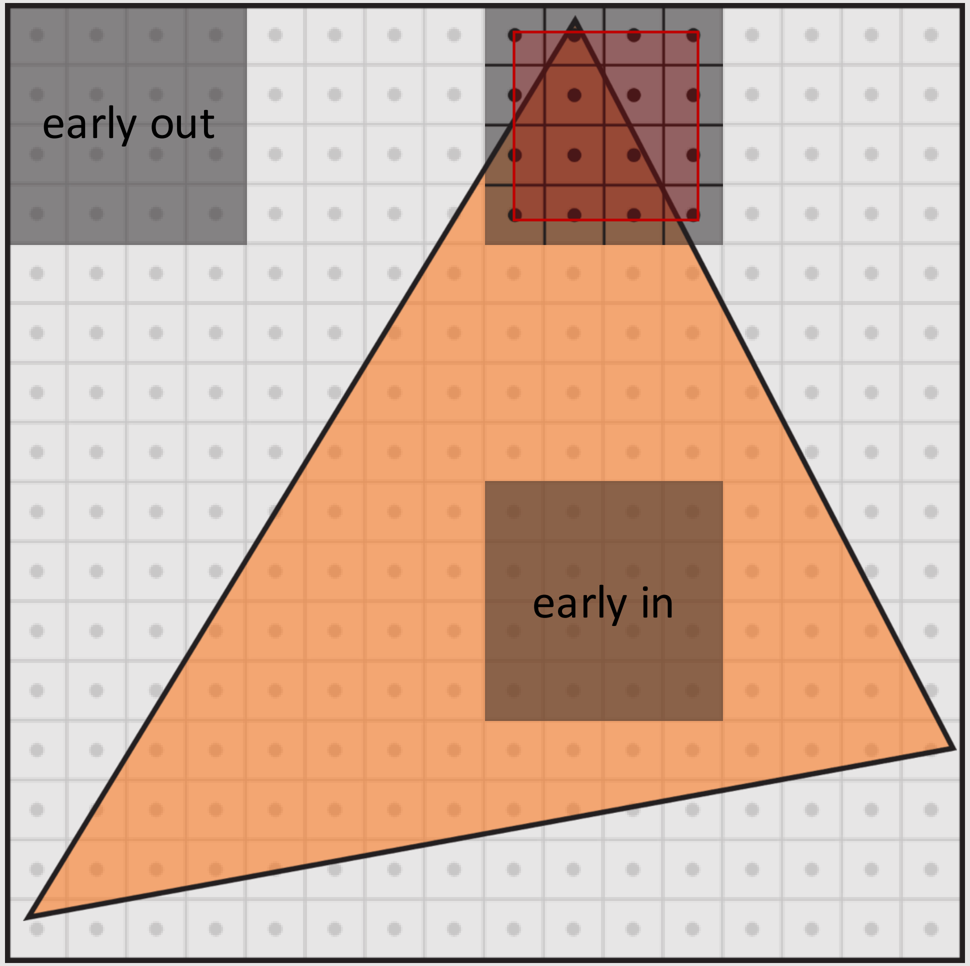


- Incremental traversal is very serial; modern hardware is highly parallel
  - Test all samples in triangle bounding box in parallel

- All tests share some 'setup' calculations
  - Computing $\overrightarrow{ac}$, $\overrightarrow{cb}$, $\overrightarrow{ba}$

- Modern GPUs have special-purpose hardware for efficiently performing point-in-triangle tests
  - Same set of instructions, regardless of which coordinate q we are dealing with

Hierarchical Coverage Tests



- **Idea:** work coarse-to-fine
  - Check if large blocks are inside the triangle
    - **Early-in:** every pixel is covered
    - **Early-out:** every pixel is not covered
    - **Else:** test each pixel coverage individually

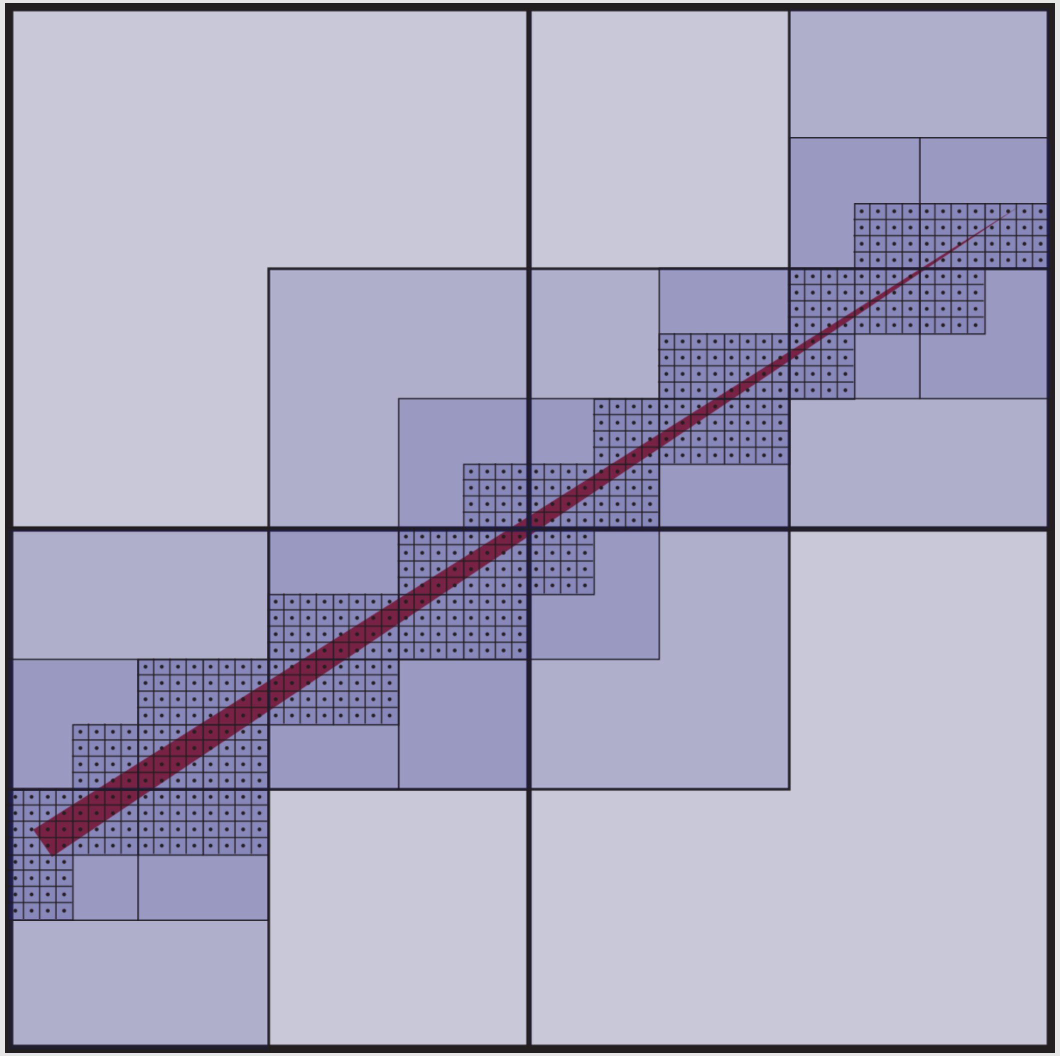
- **Early-in:** if all 4 corners of the block are inside the triangle
- **Else:** if a triangle line intersects a block line
- **Early-out:** if neither **Early-in** nor **Else**

- **Careful!** Best to represent block as smallest bounding box to pixel samples, not the pixels themselves!

Hierarchical Coverage Tests

- What is the right block size?
  - **Too big:** very difficult to get an **Early-in** or **Early-out**
  - **Too small:** blocks are too similar to pixels

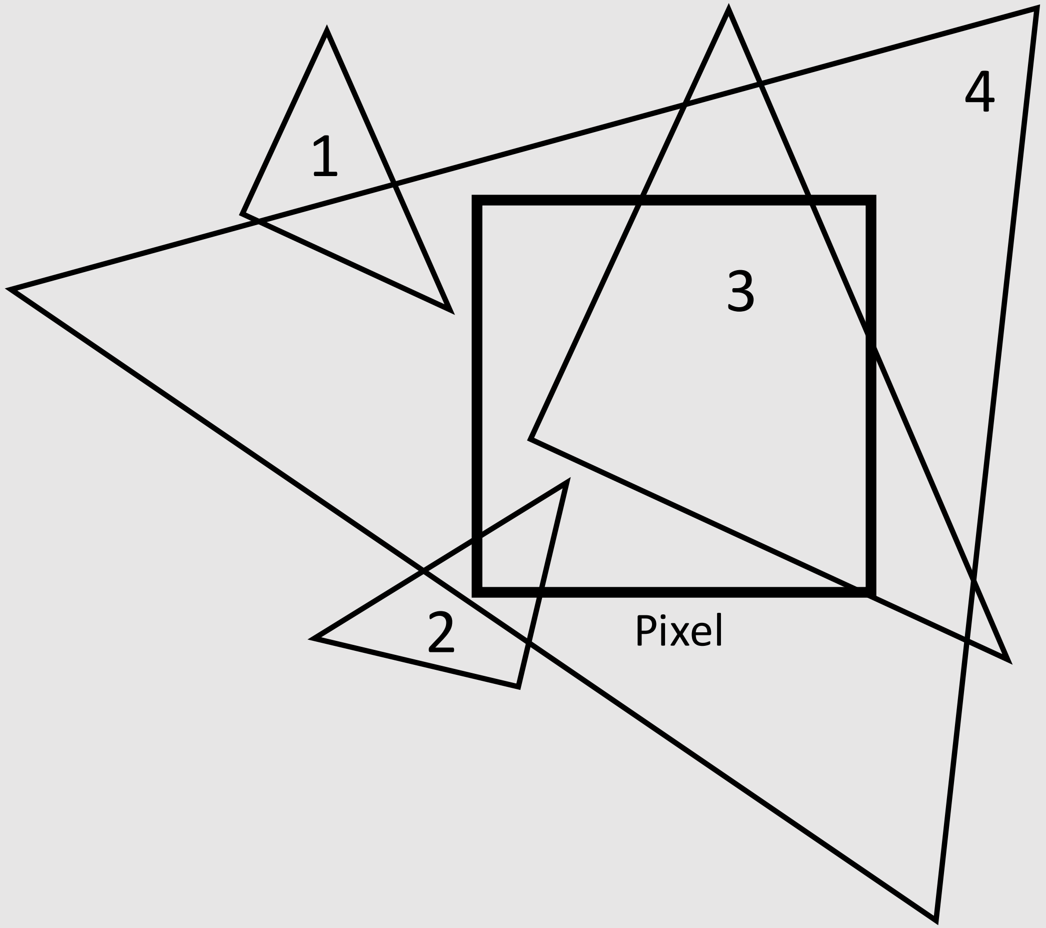
- **Idea:** create a hierarchy of block sizes
  - When entering the **Else** case, just drop down to the next smallest block size
  - Checking coverage reduced to logarithmic (We will learn why in a future lecture)



- ~~Perspective Projection~~
- ~~Drawing a Line~~
- ~~Drawing a Triangle~~
- Supersampling

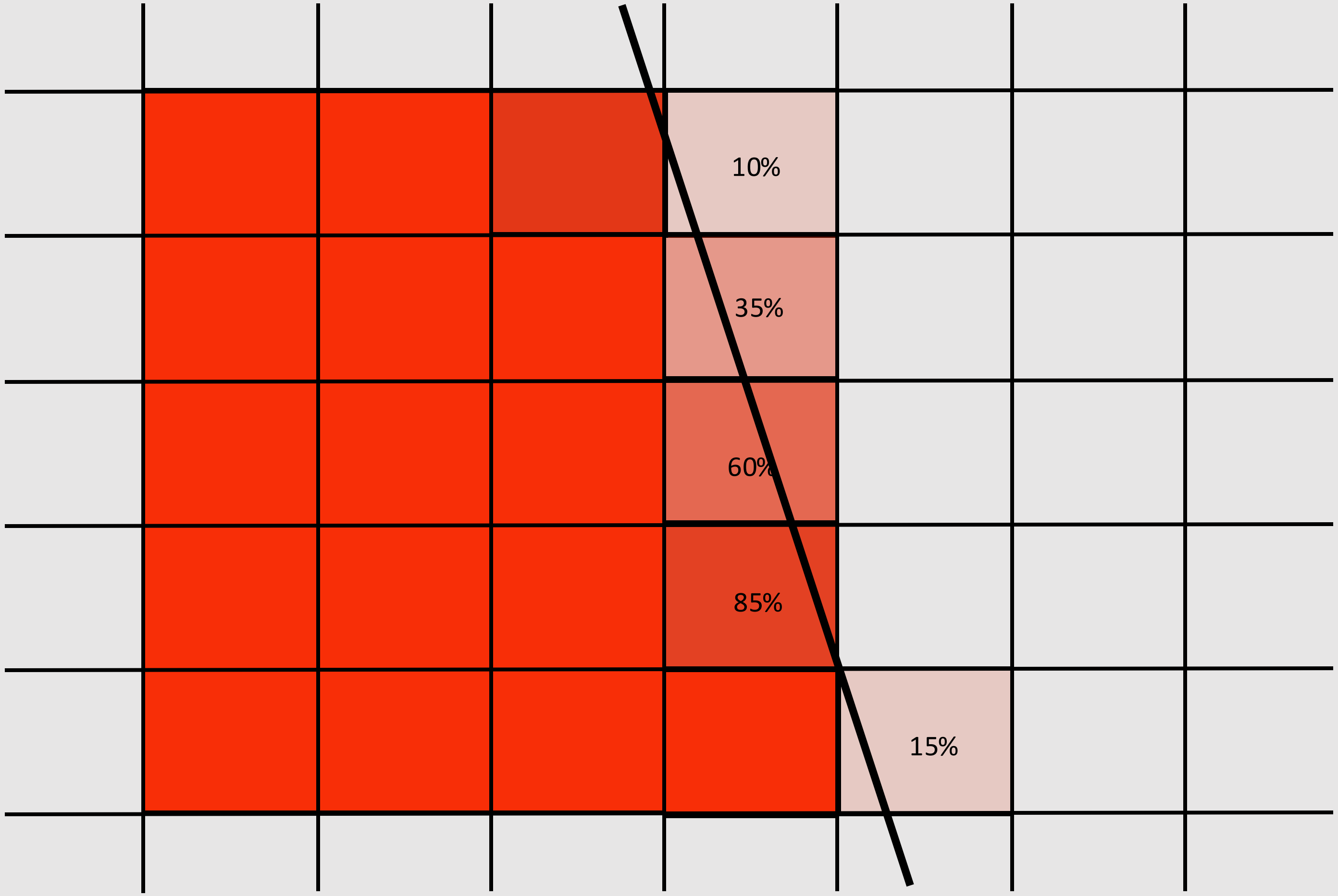
Pixel Coverage

Which triangles “cover” this pixel?

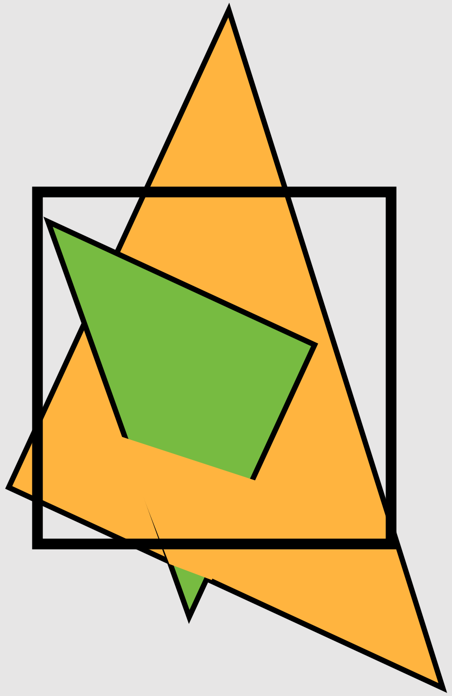


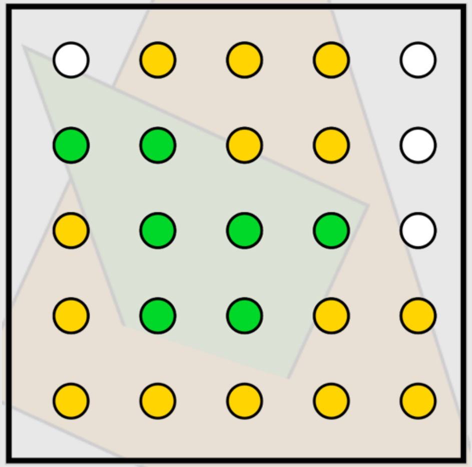
Pixel Coverage

- Compute fraction of pixel area covered by triangle, then color pixel according to this fraction
  - **Ex:** a red triangle that covers 10% of a pixel should be 10% red
- Difficult to compute area of box covered by triangle
  - Instead, consider coverage as an approximation



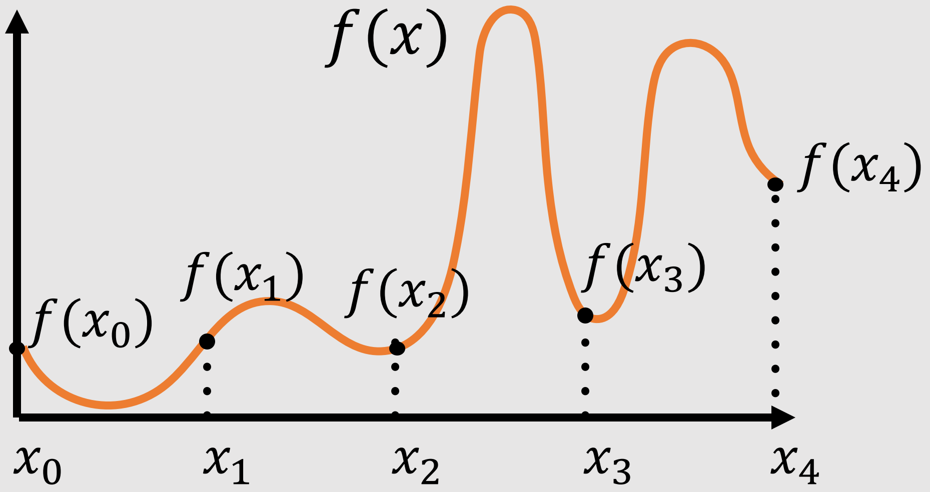
Coverage Via Samples

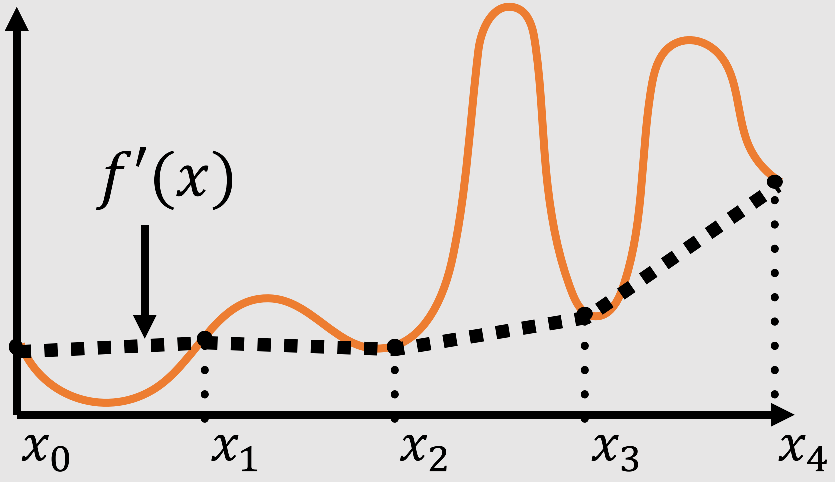




- A **sample** is a discrete measurement of a signal
  - Used to **convert continuous data to discrete**, but we can also take **samples of discrete data** too
- The more samples we take, the more accurate the image becomes
  - Same idea as using a larger sensor to take a better-quality photo
- **Problem:** each sample adds more work
  - What is the best way to use the least amount of samples to best approximate the original scene?
    - Main idea of **sample theory**

Sampling in 1D

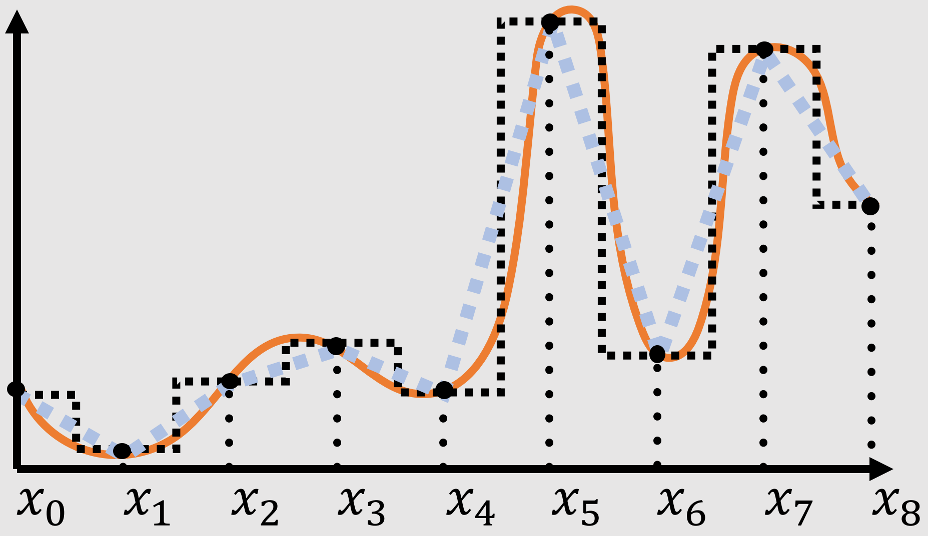




- **Idea:** take 5 random samples along the domain and evaluate $f(x)$
  - Many different ways to interpolate points:
    - Piecewise
    - Linear
    - Cubic

- Where is the best place to put 5 samples?
  - We know the answer because we can see the entire function f
    - f has been evaluated over the entire domain
  - What if we cannot see all of f?
  - What if f is expensive to evaluate?

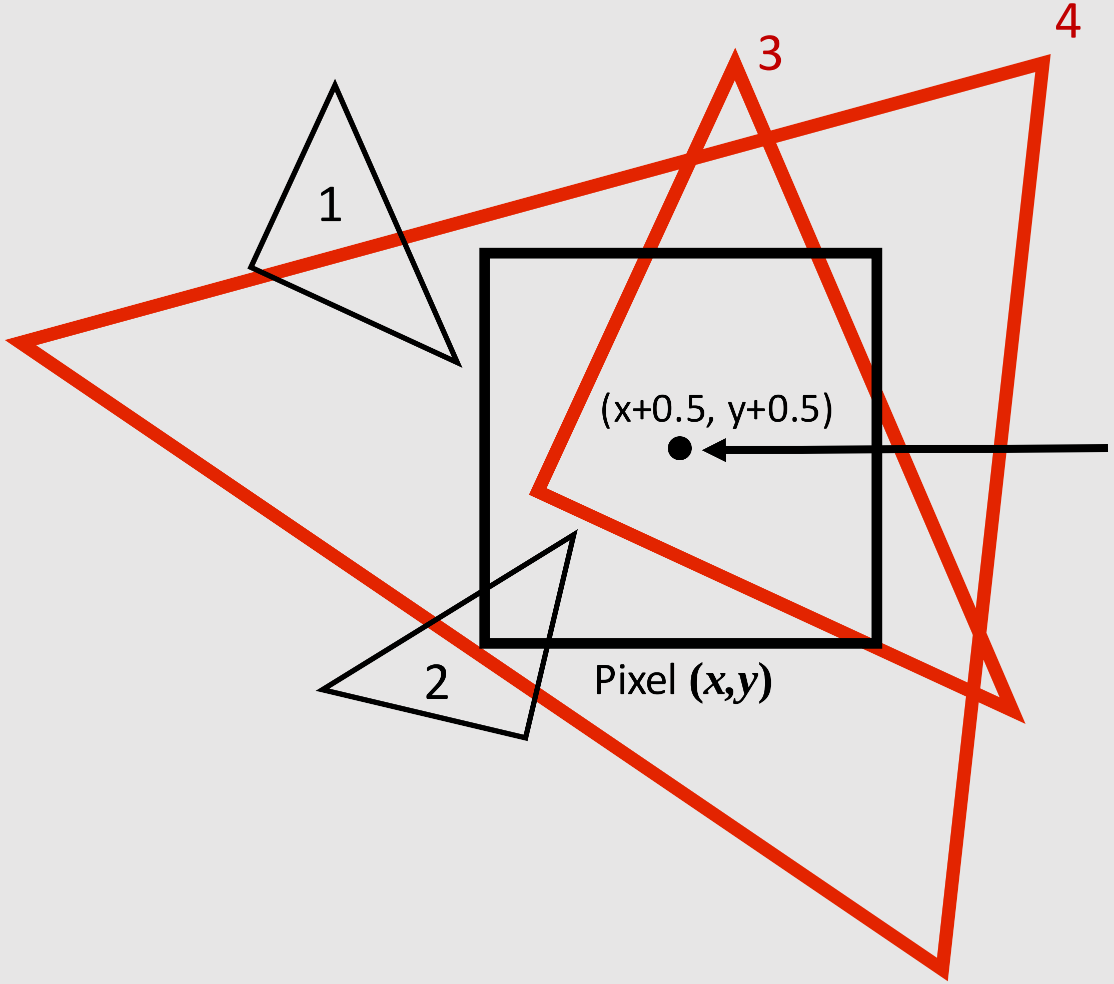
Sampling in 1D



- **Idea:** take more than 5 random samples along the domain and evaluate $f(x)$
  - Gets a better reconstruction of f but…
    - More evaluation calls needed
    - More memory to save
- Still don't know the best way to interpolate samples
  - Need to guess based on the behavior of f
  - Can consider things like gradients and such…

Pixel Coverage

Which triangles "cover" this pixel?

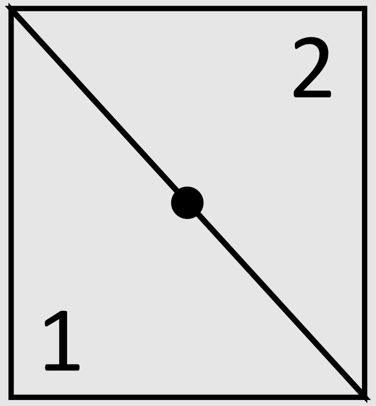


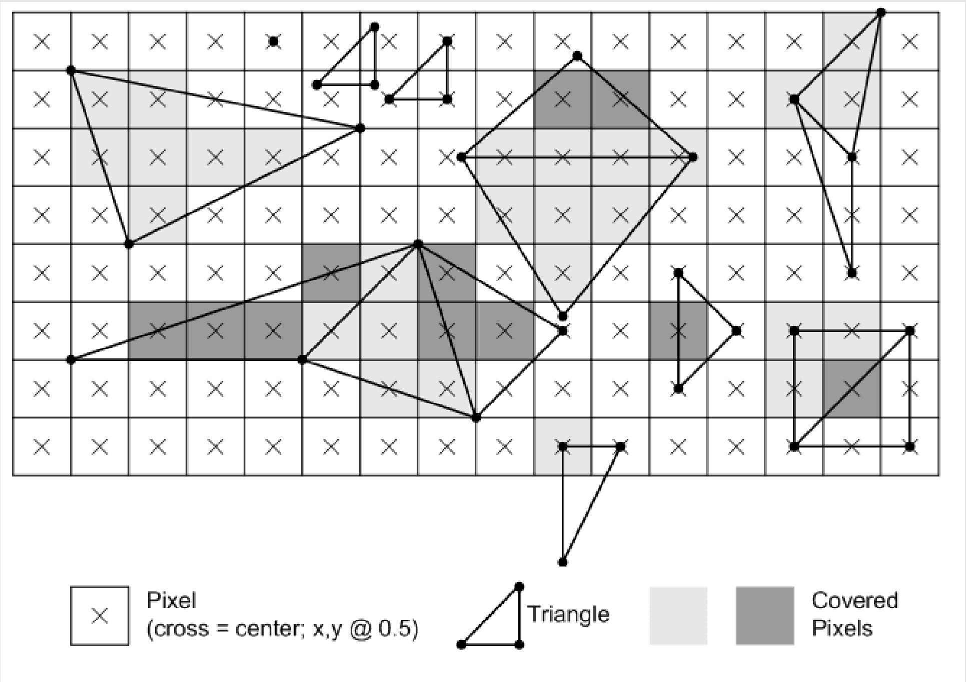
Here I chose the coverage sample point to be at a point corresponding to the pixel center

= triangle

= triangle but with a red outline

Edge Case



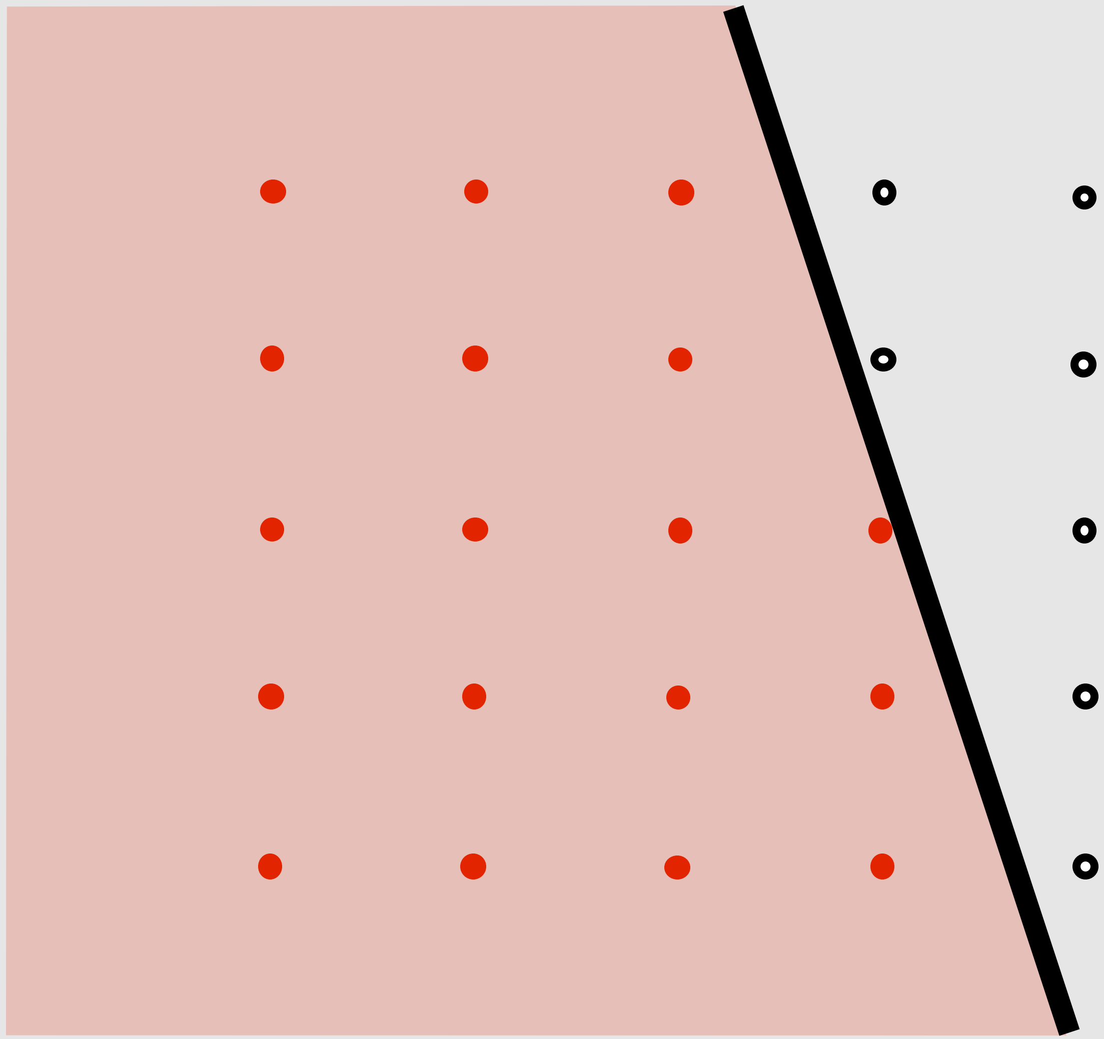


Direct3D Documentation (2020) Microsoft

- When edge falls directly on a screen sample, the sample is classified as within triangle if the edge is a “top edge” or “left edge”
  - **Top edge:** horizontal edge that is above all other edges
  - **Left edge:** an edge that is not exactly horizontal and is on the left side of the triangle
    - Triangle can have one or two left edges
- This is known as **edge ownership**

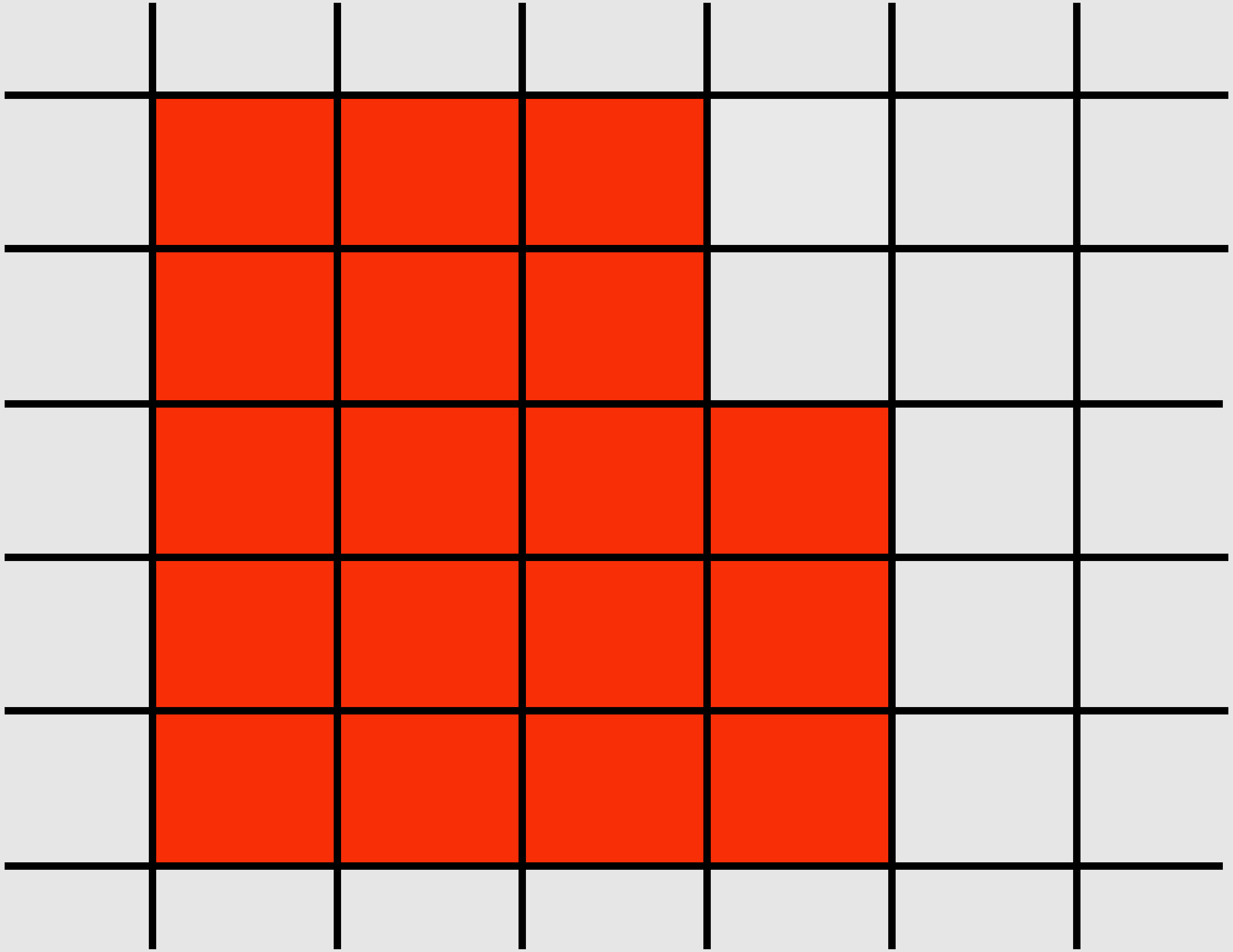
So how many samples do we take?

Sampling Per Pixel



Idea: take as many samples as there are pixels on screen

Sampling Per Pixel

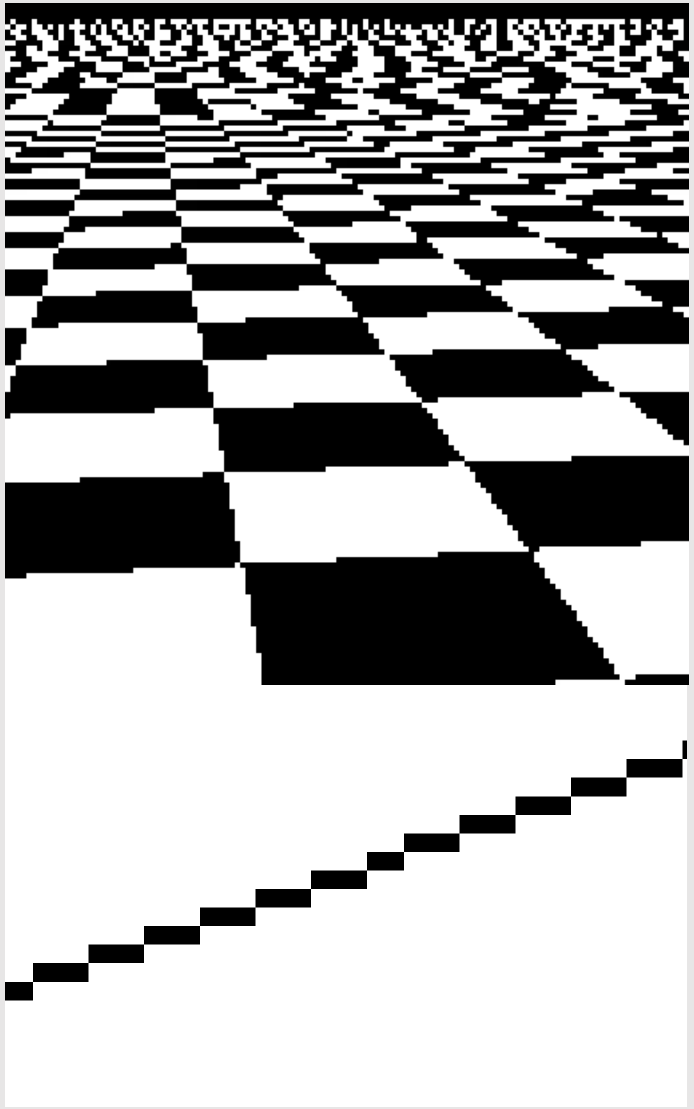


Problem: Results look blocky against edges
(let’s take more samples!)

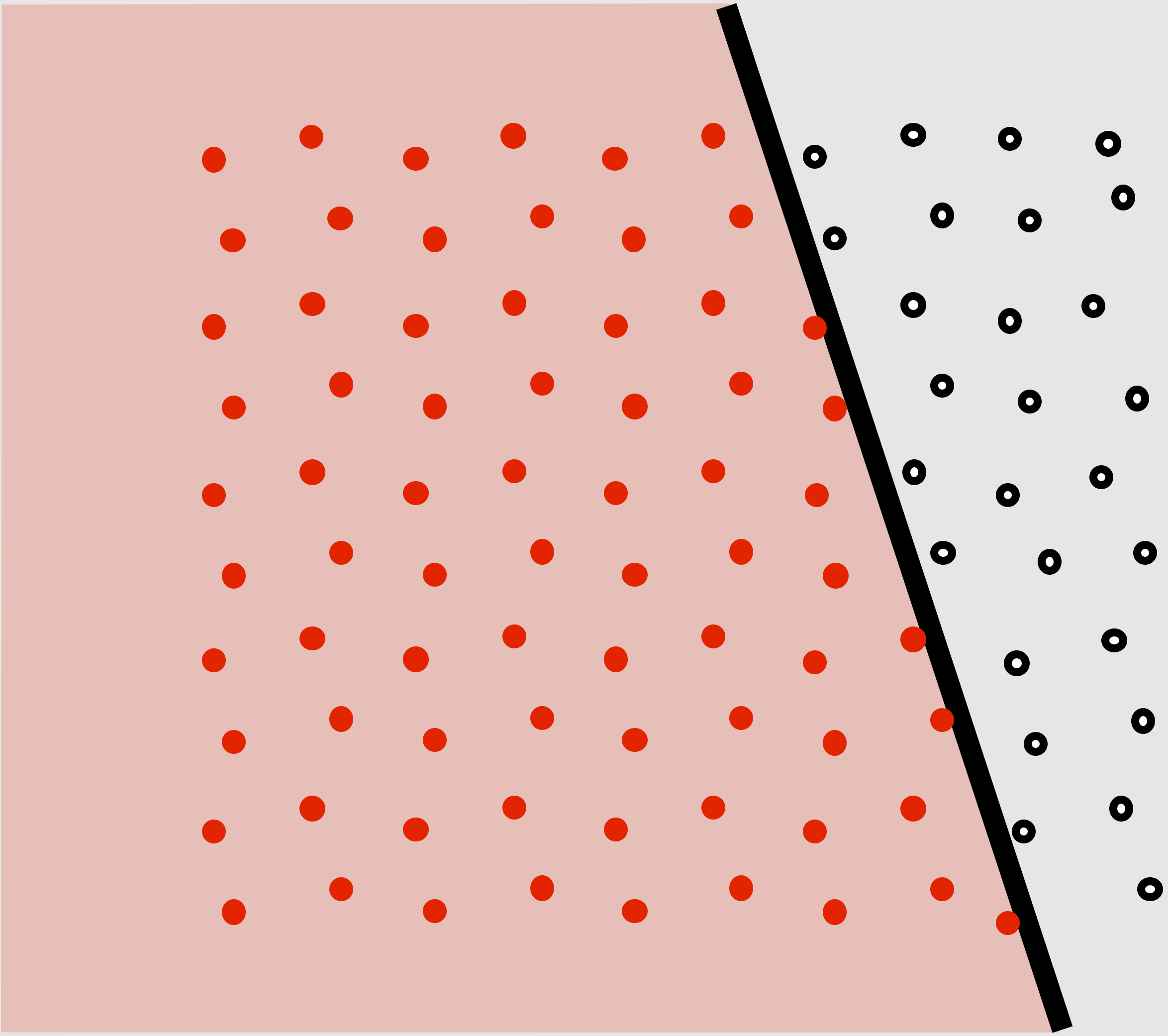
Aliasing Artifacts

- Imperfect sampling + imperfect reconstruction leads to image artifacts
  - Jagged edges
  - Moiré patterns

- Does this remind you of old school video games?
  - Old games took few samples and took few steps to prevent aliasing
    - Expensive to take more samples
    - Not enough compute to do filtering to interpolate samples
    - Not enough memory to take more samples

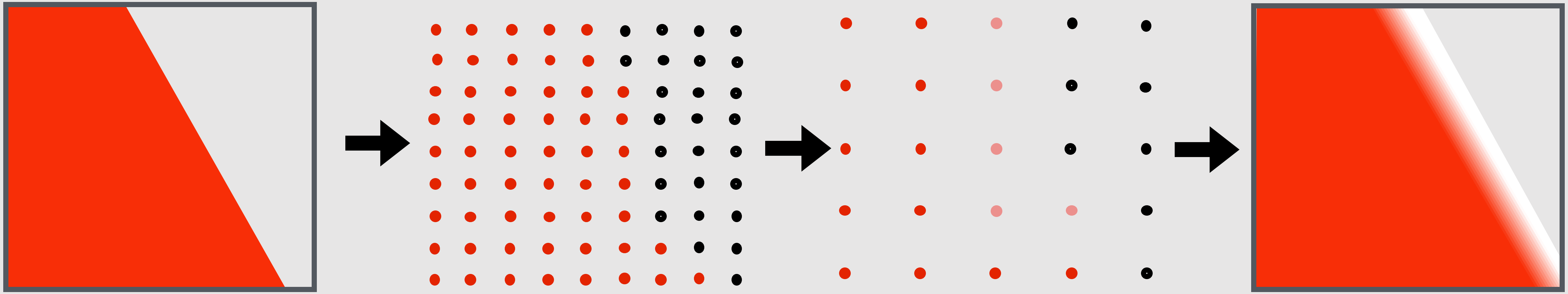


Supersampling Per Pixel



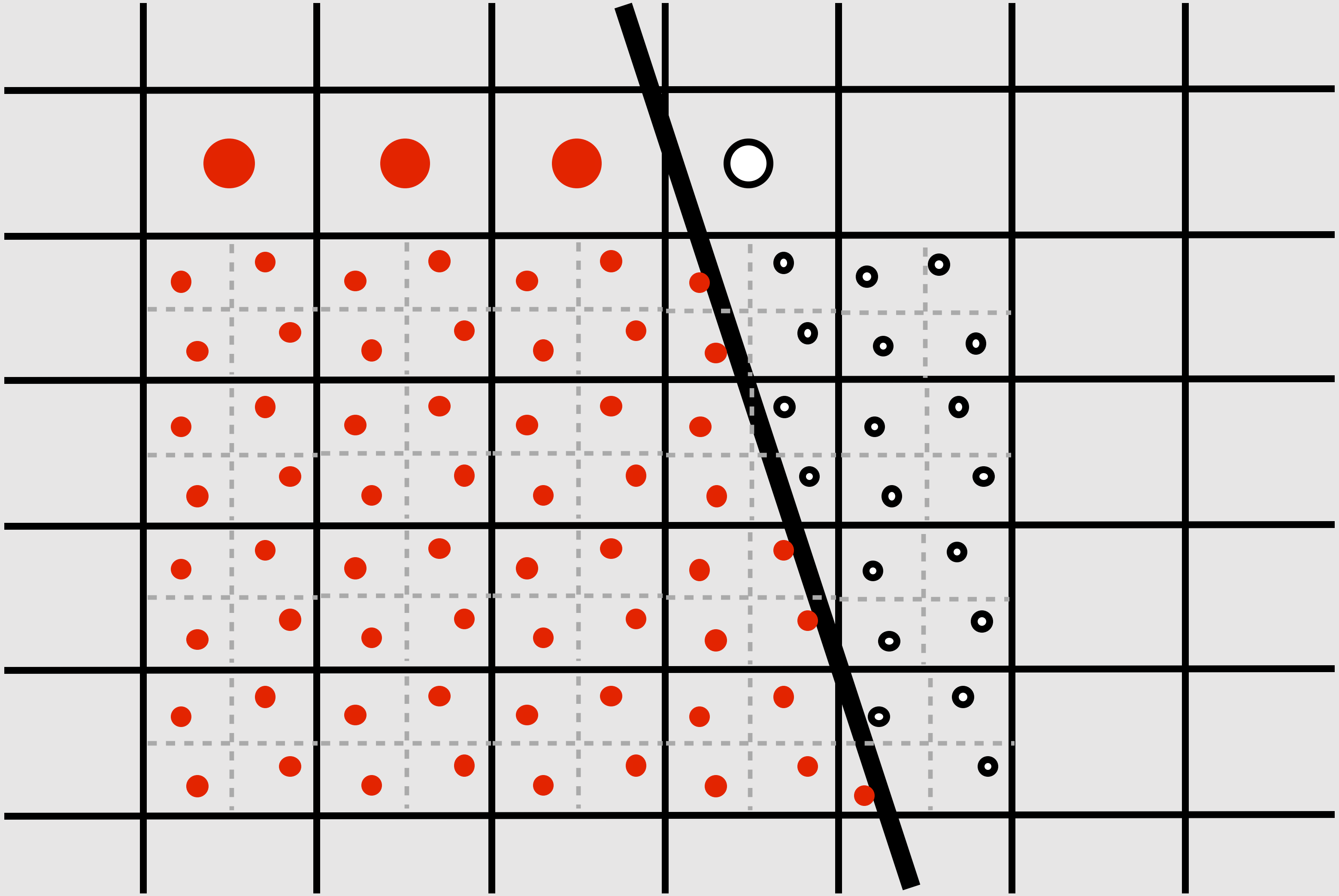
Idea: take many more samples than there are pixels on screen

Resampling

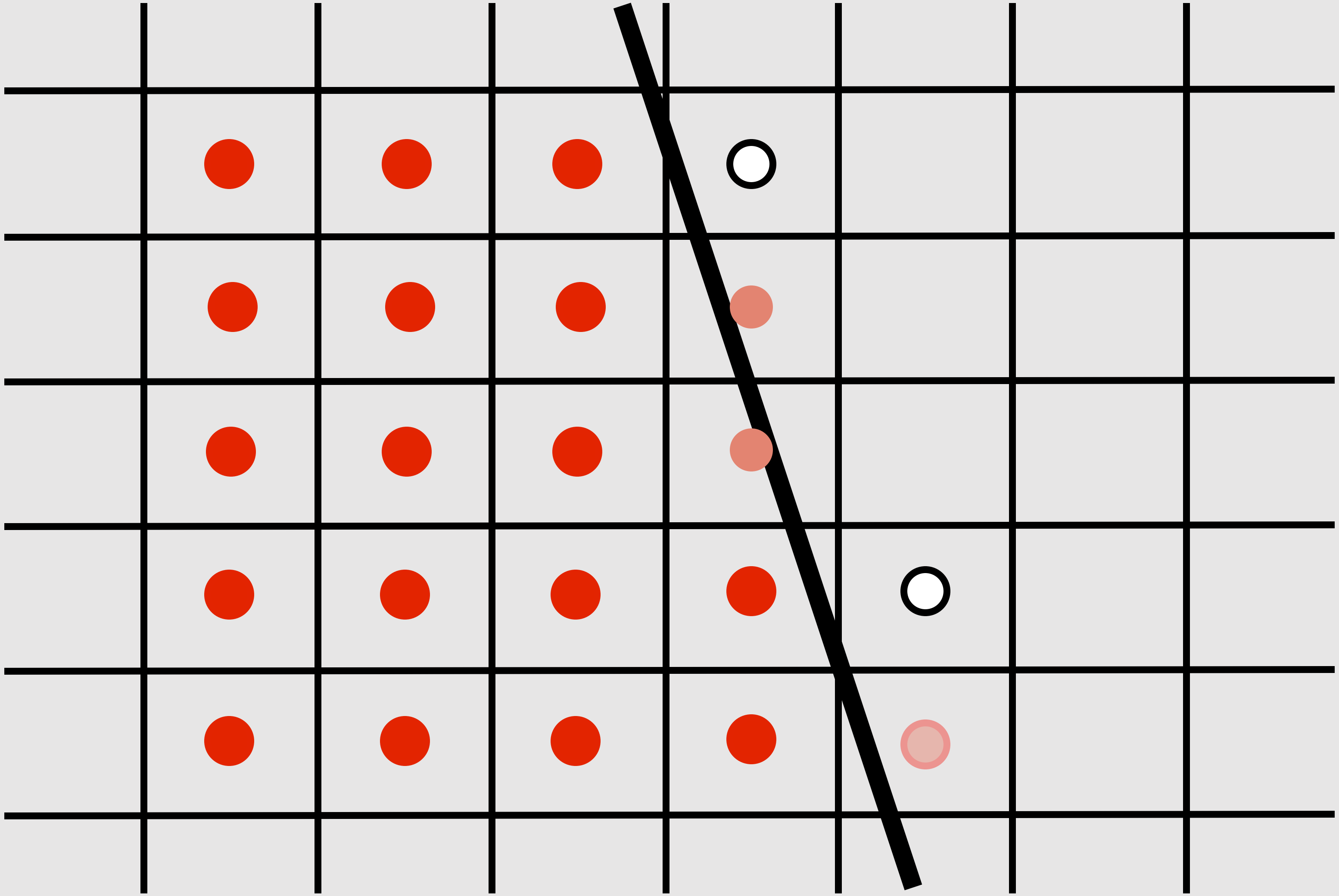


Each pixel now holds **n** samples.
Average the **n** samples together to get **1** sample per pixel **(1spp).**

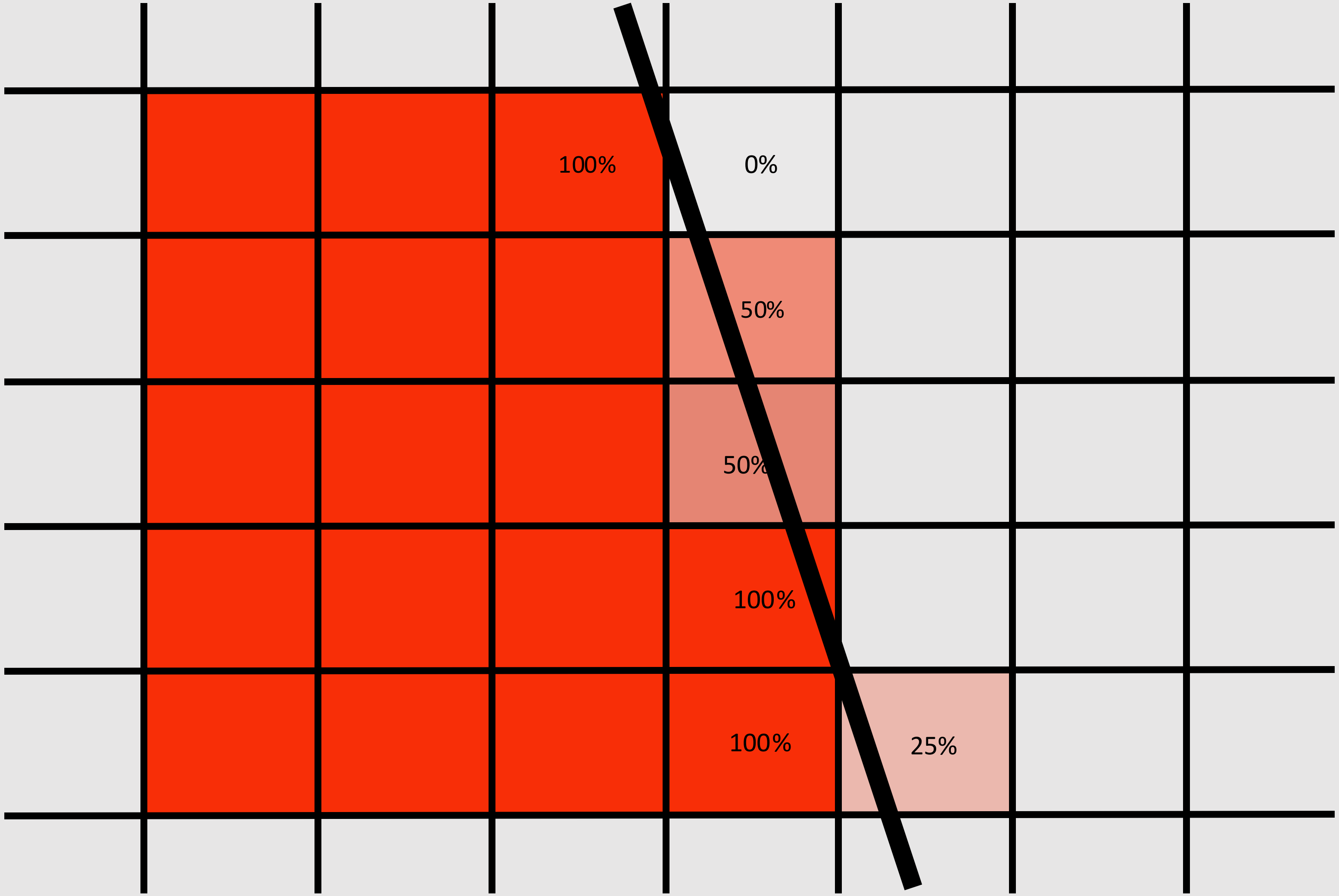
Resampling



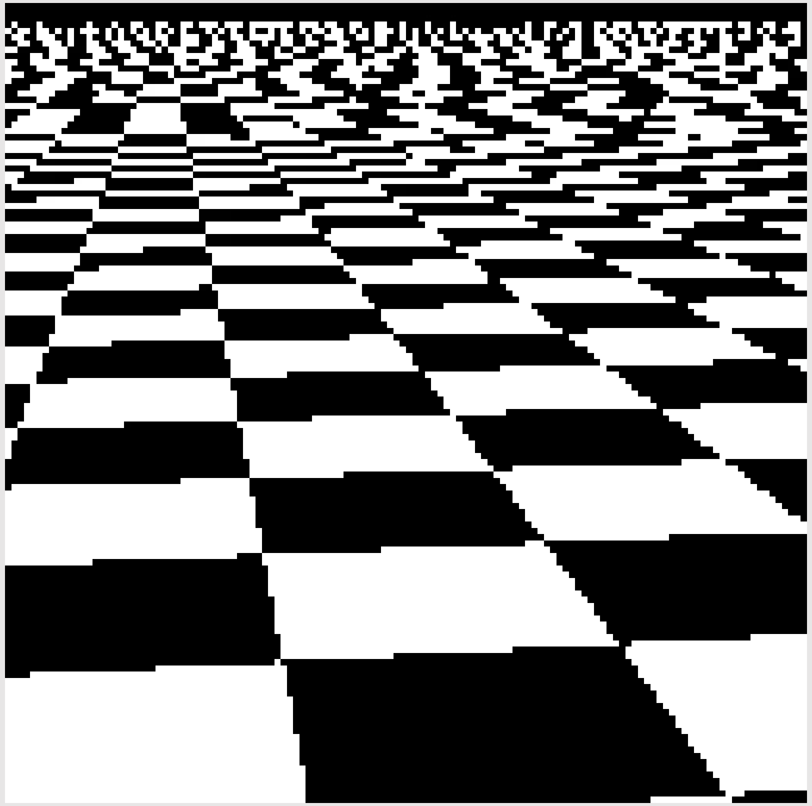
Resampling



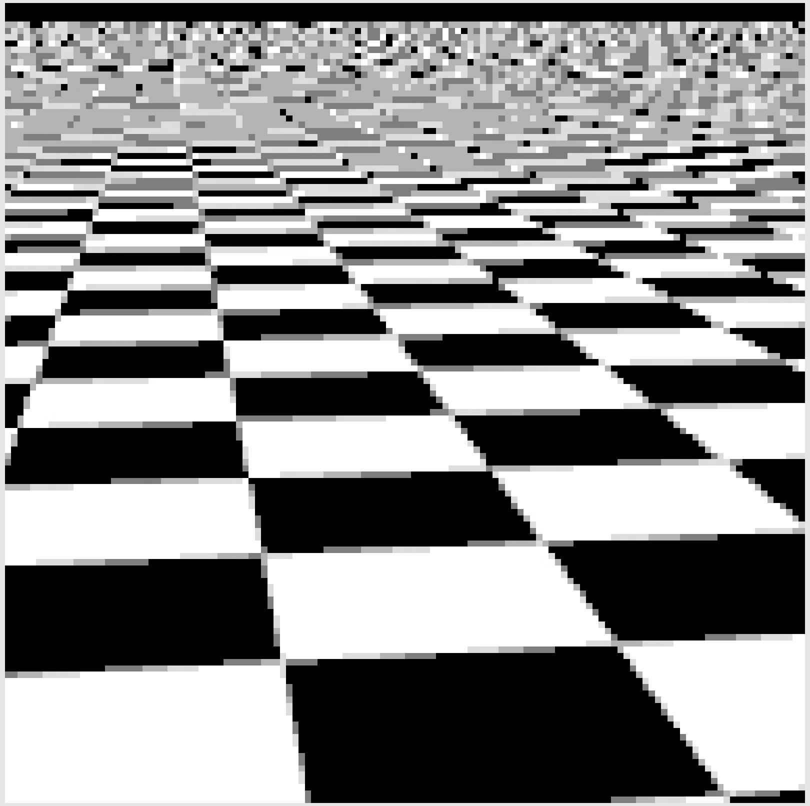
Resampling



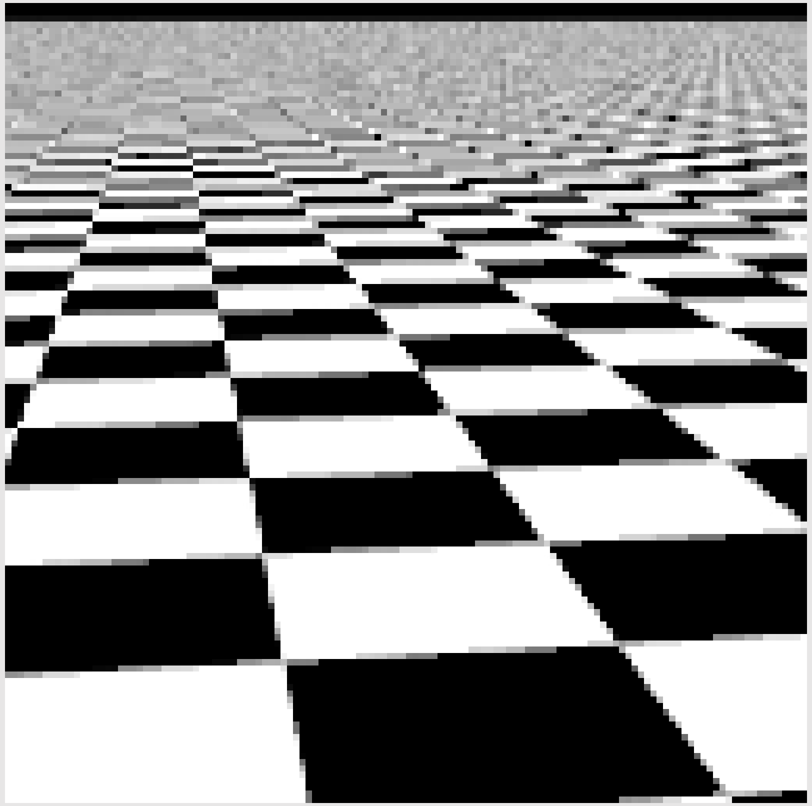
Supersampling Artifacts



[1x1spp]

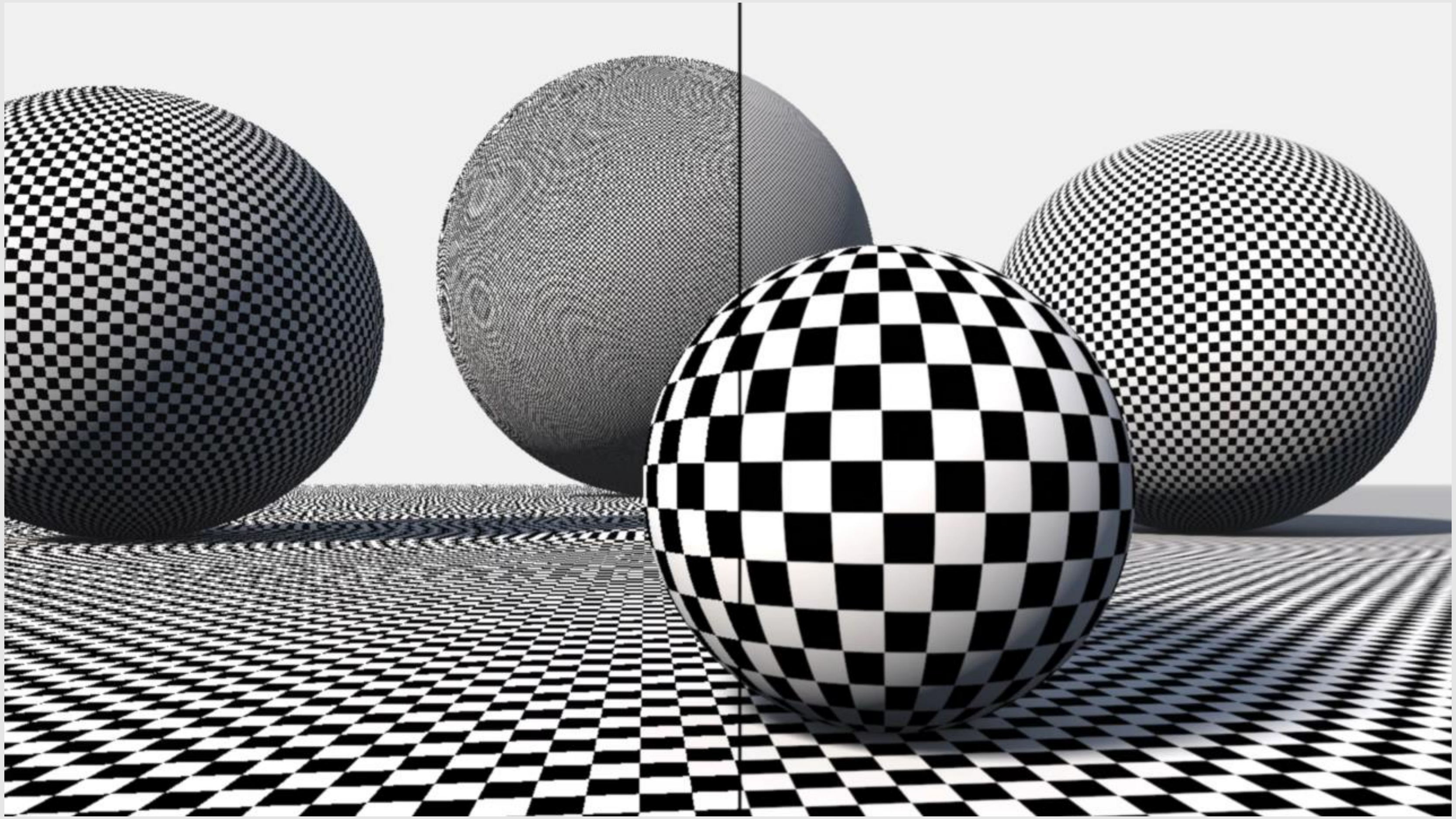


[4x4spp]



[32x32spp]

Supersampling Artifacts



In special cases, we can compute the exact coverage.
This occurs when what we are sampling matches our sampling pattern – **very rare!**