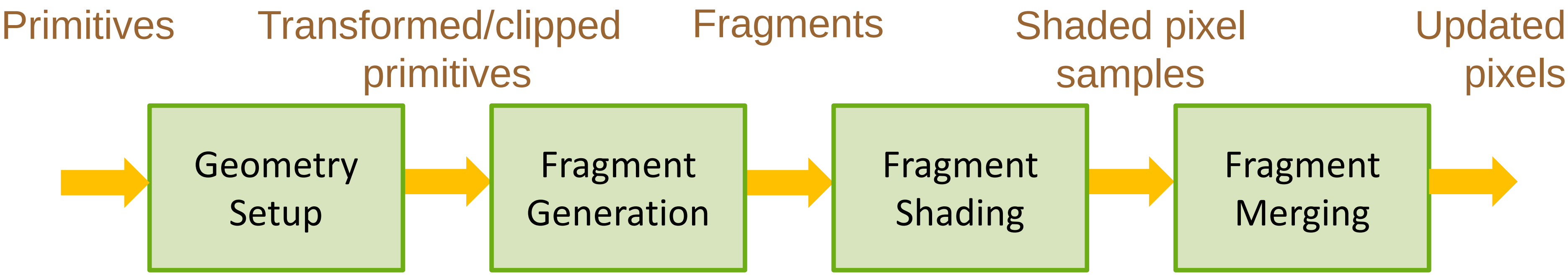
Rasterization Architectures

Georgios Papaioannou - 2014

A High Level Rasterization Pipeline



Geometry Setup
- Transformation
- Culling
- Primitive assembly
- Clipping

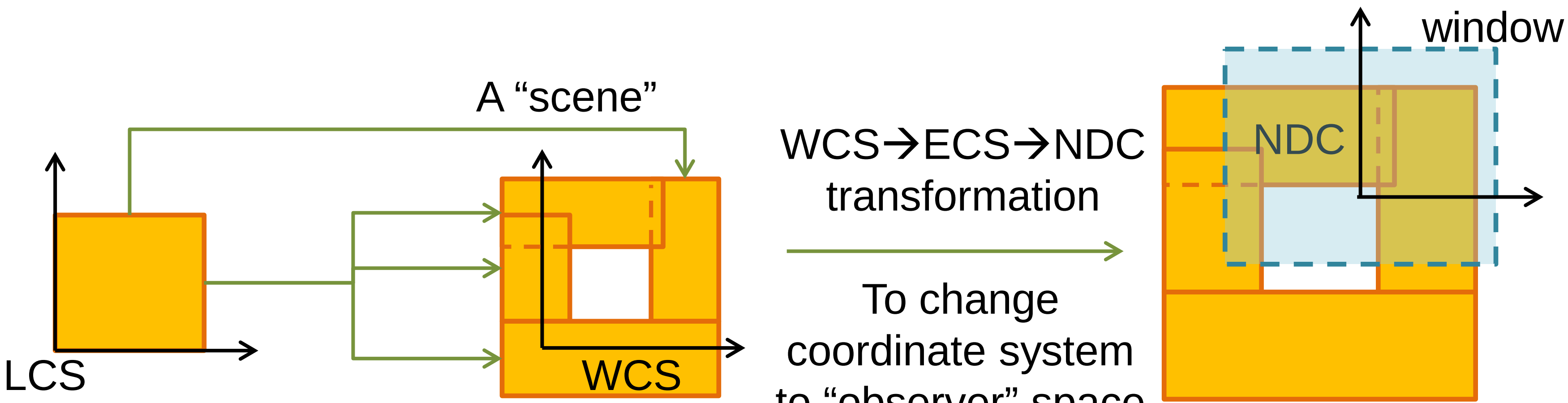
Fragment Generation
- Primitive sampling
- Attribute interpolation
- Pixel coverage estimation

Fragment Shading
- Pixel color determination
- Transparency
- …

Fragment Merging
- Visibility determination
- Blending
- Reconstruction filtering

Geometry Setup

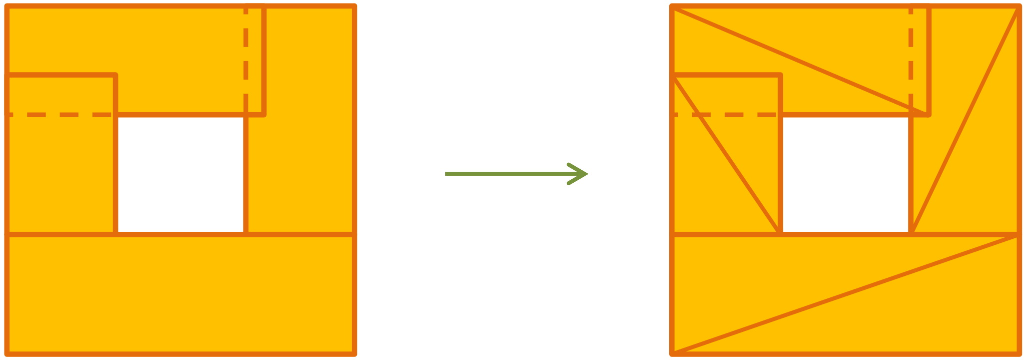
- Geometry must be transformed in order to:
  - Be expressed in the proper coordinate system for each operation to take place
  - Get modified according to the desired arrangement of primitives / objects to form a virtual world or scene



Various geometric transformations applied to original shape to build the desired outcome

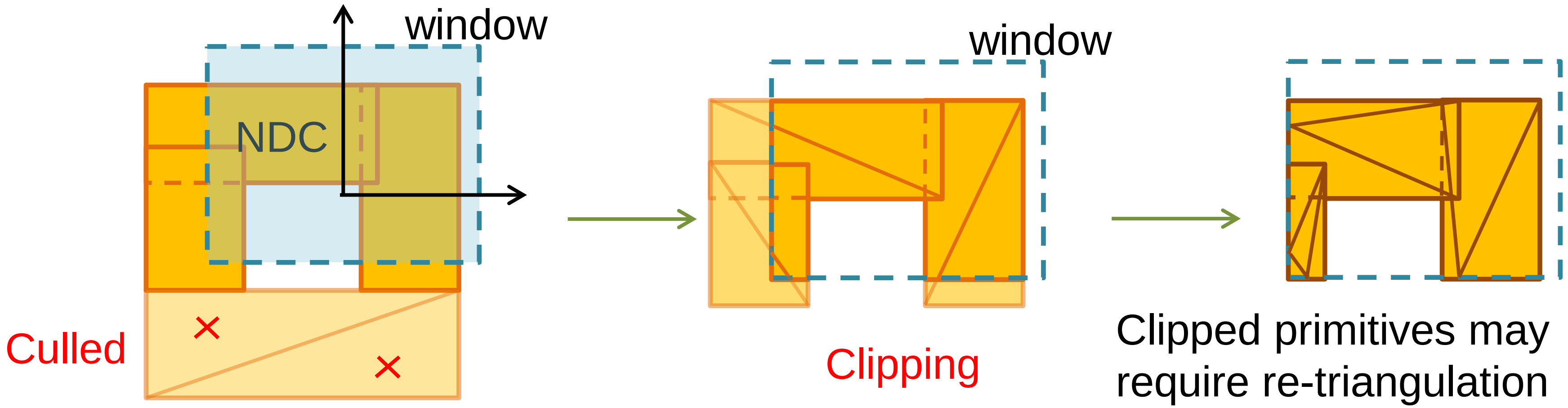
Geometry Setup (2)

- The vertices of the resulting primitives are then assembled into a form that can be efficiently sampled by the rasterizer (e.g. triangles):



Geometry Setup (3)

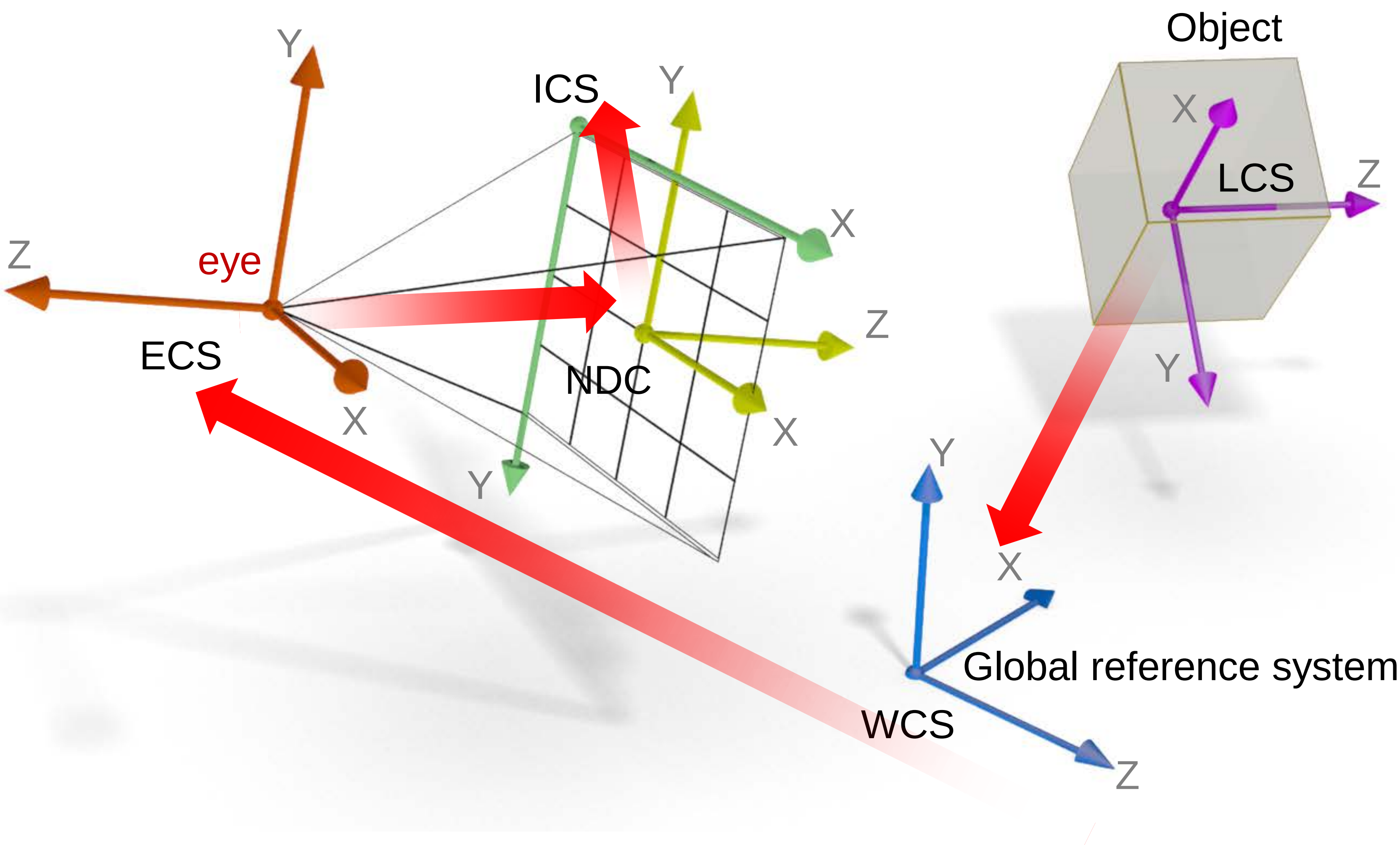
- Redundant geometry (invisible, unimportant etc.) is culled (removed) to reduce overhead
- To further reduce/split load and avoid degenerate / problematic geometry, primitives are clipped to the boundaries of NDC regions



3D Geometry Transformations

- All coordinates have to be:
  - Transformed from their native, object space ones to a global, common reference system
  - Then expressed relative to the camera and
  - Projected on the image plane
- All of these transformations are concatenated into a single matrix, which is applied to the vertices of each triangle
- Different objects may have different transformations

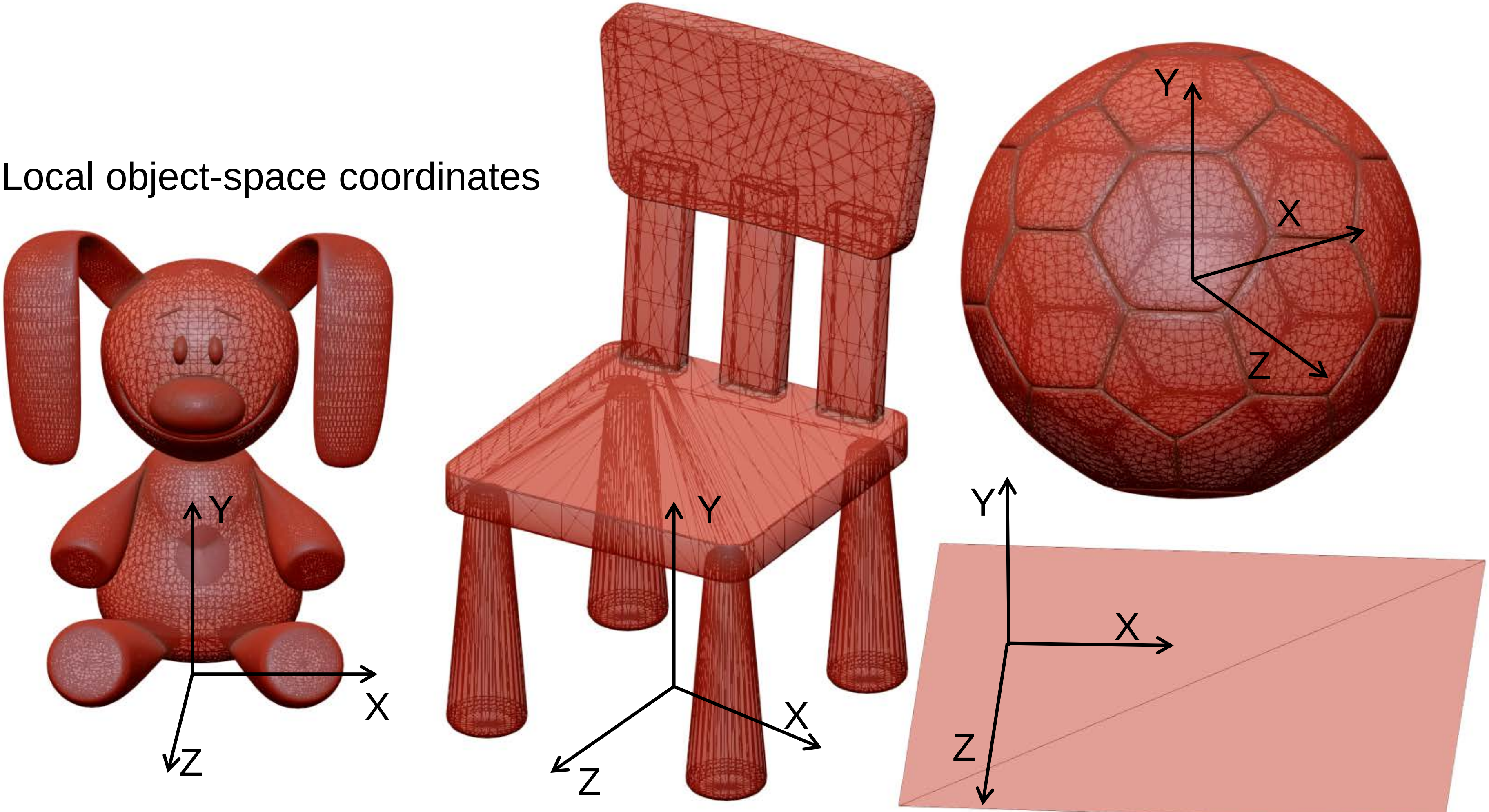
Geometric Transformation Sequence



3D Geometry Setup (1)

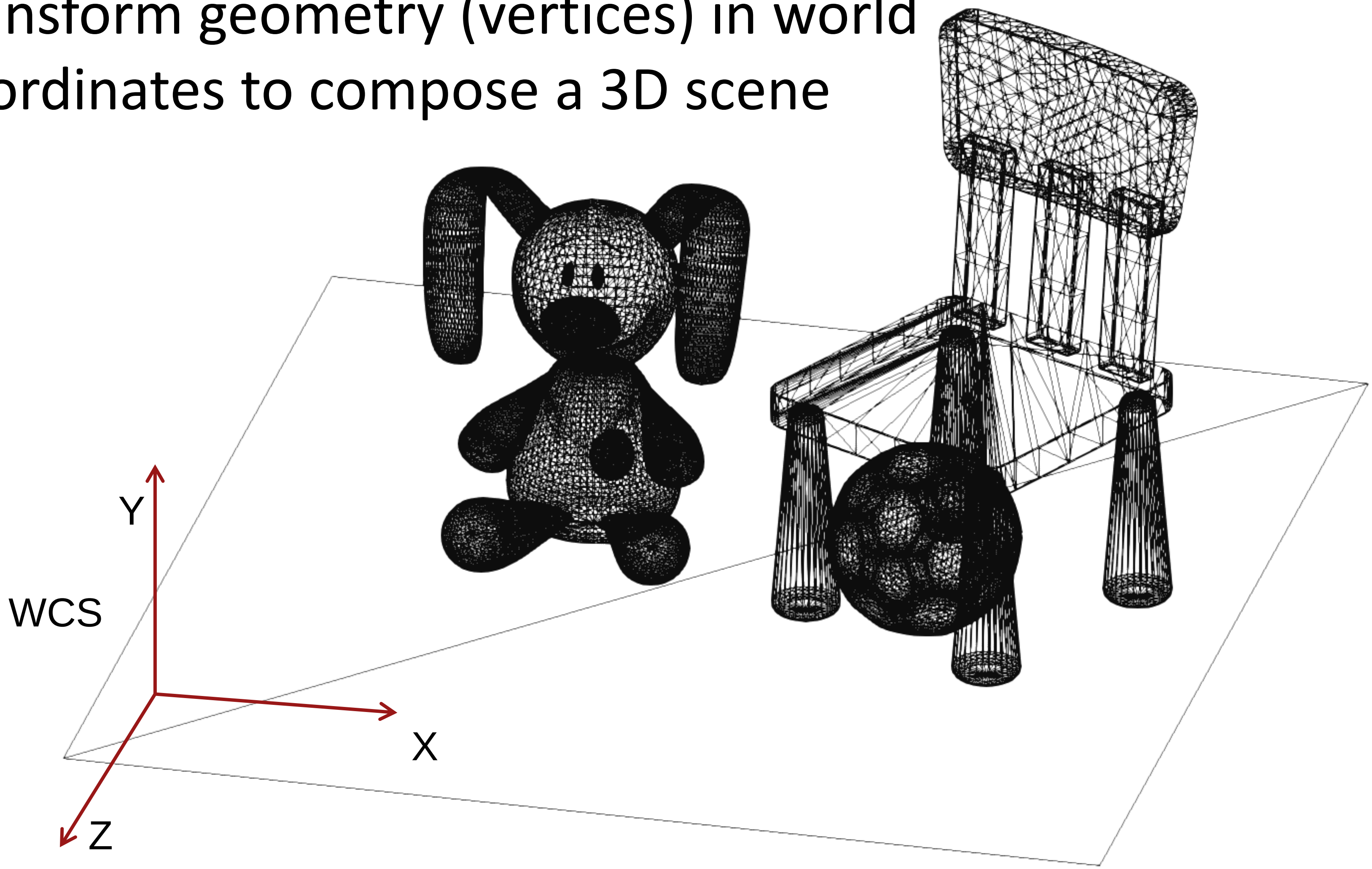
- Initial primitives (as defined/loaded by the application)

Local object-space coordinates



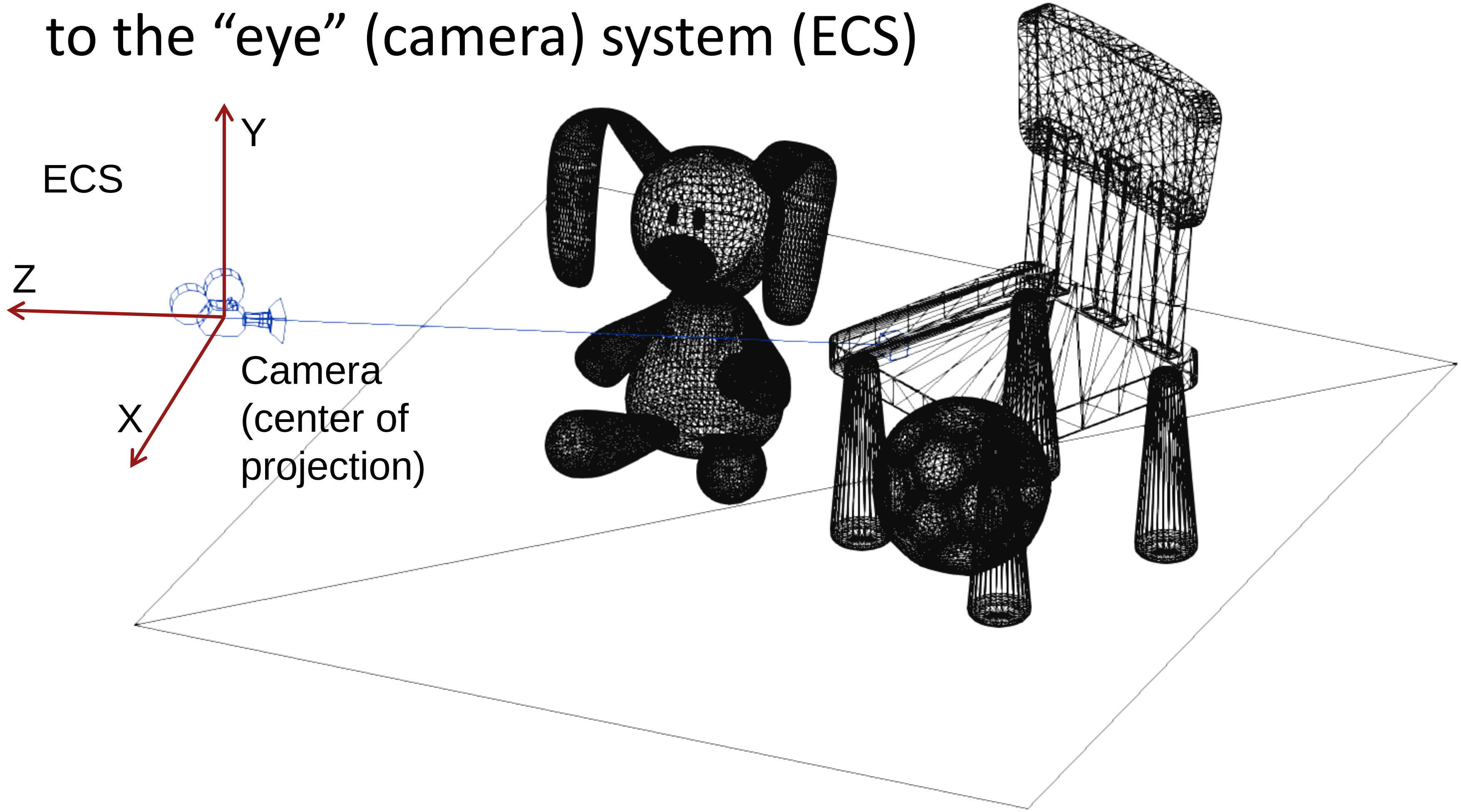
3D Geometry Setup (2)

- Transform geometry (vertices) in world coordinates to compose a 3D scene



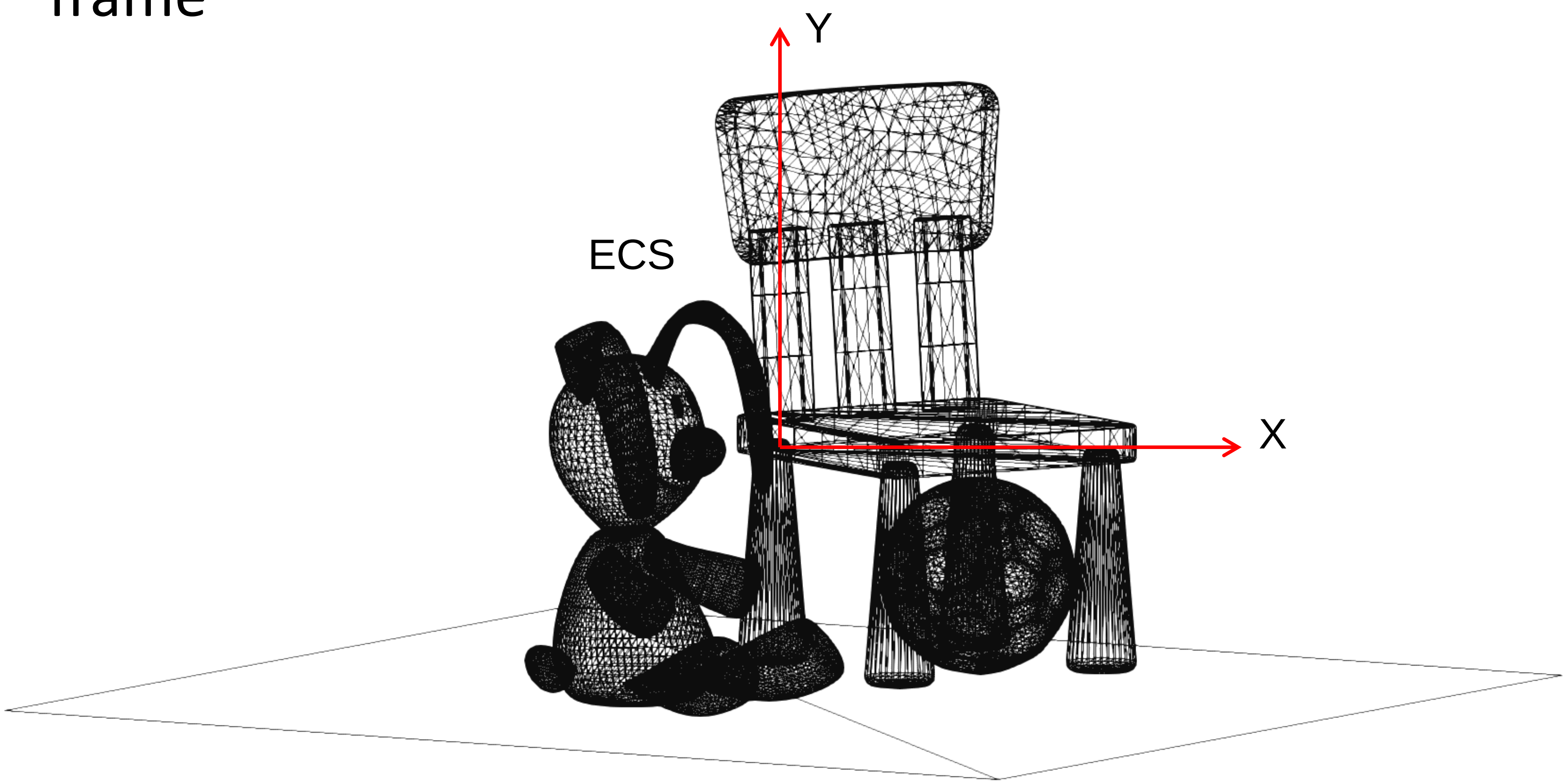
3D Geometry Setup (3)

- Transform geometry (vertices) relative to the "eye" (camera) system (ECS)



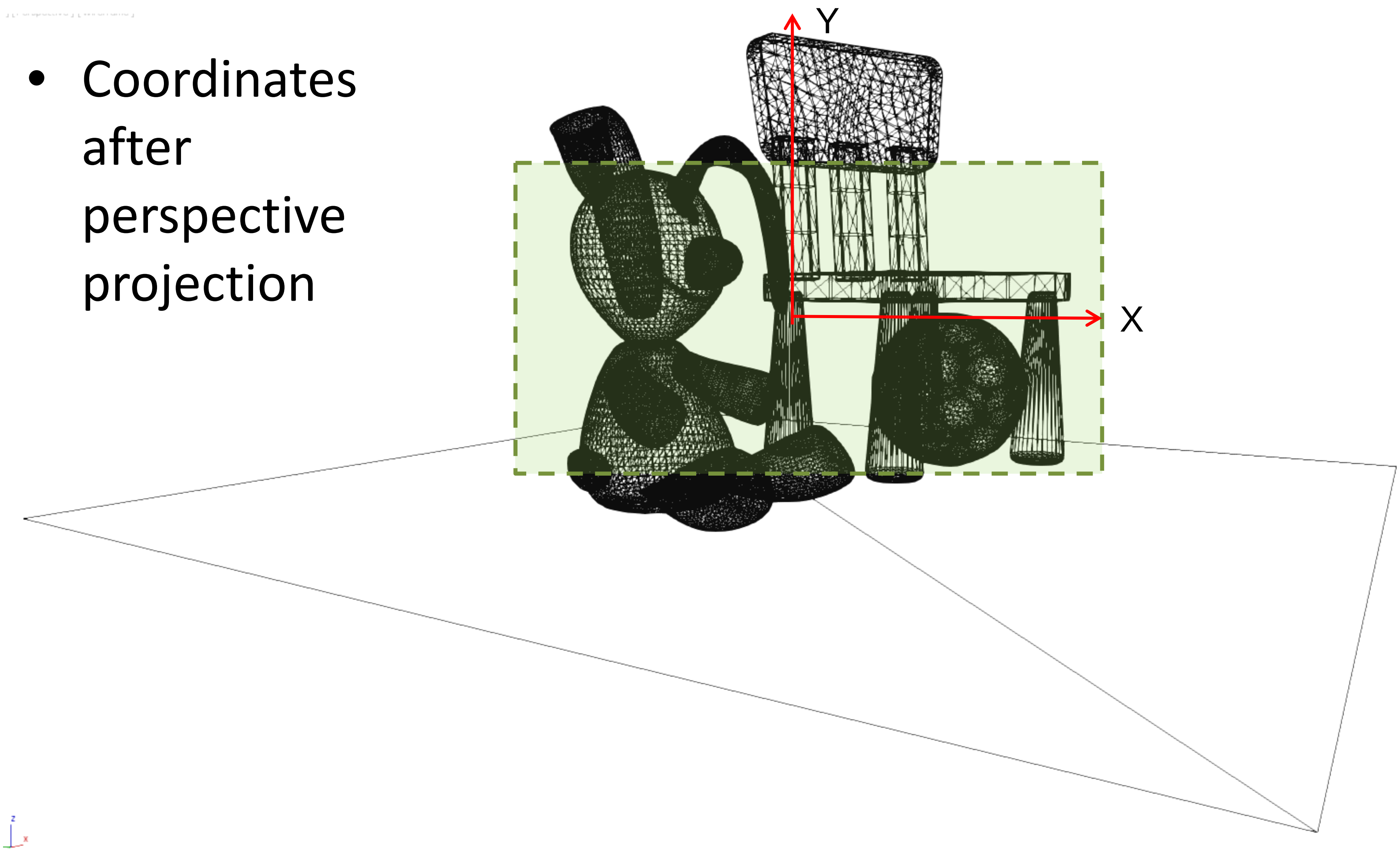
3D Geometry Setup (4)

- Coordinates as “seen” from the camera reference frame



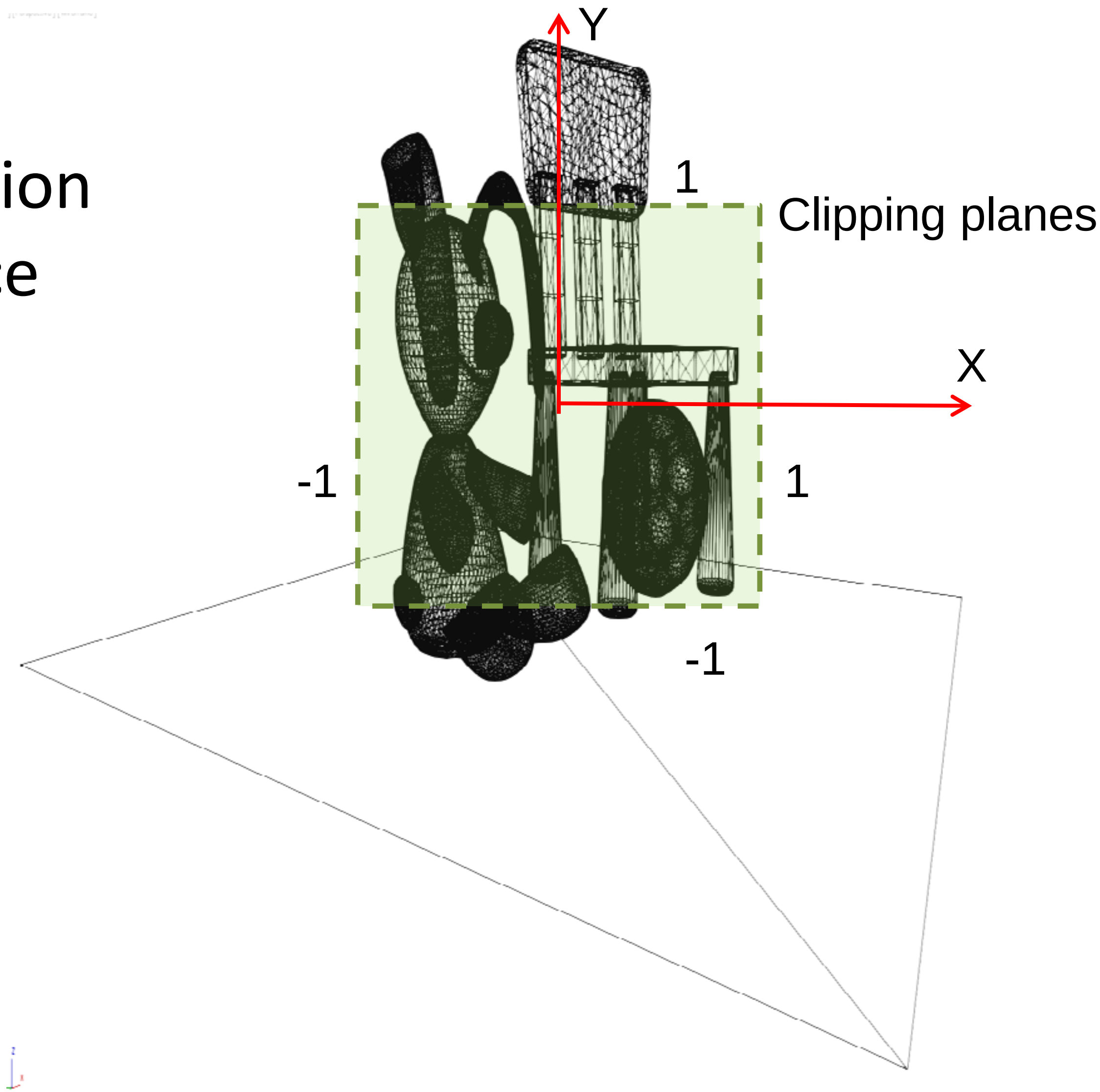
3D Geometry Setup (5)

- Coordinates after perspective projection



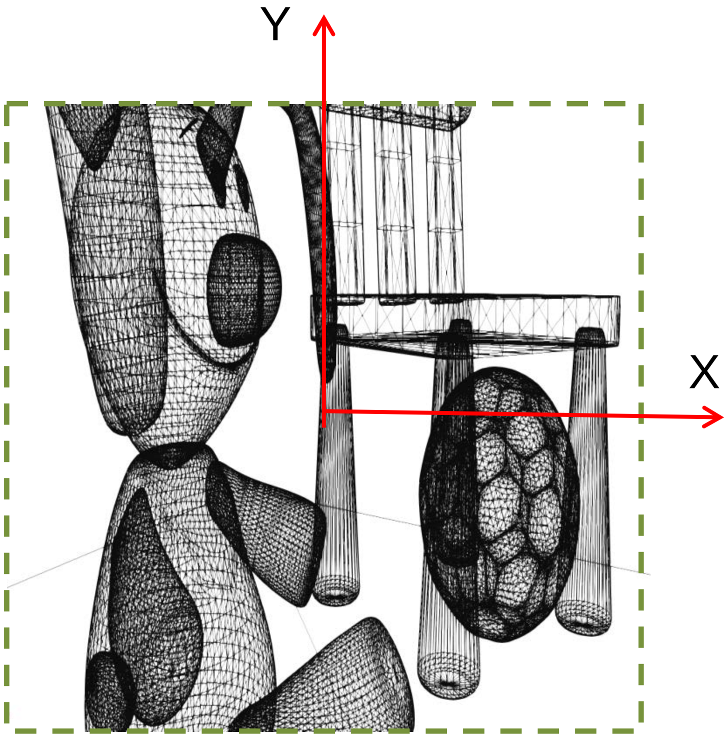
3D Geometry Setup (6)

- Coordinates after perspective projection in normalized device coordinates



3D Geometry Setup (7)

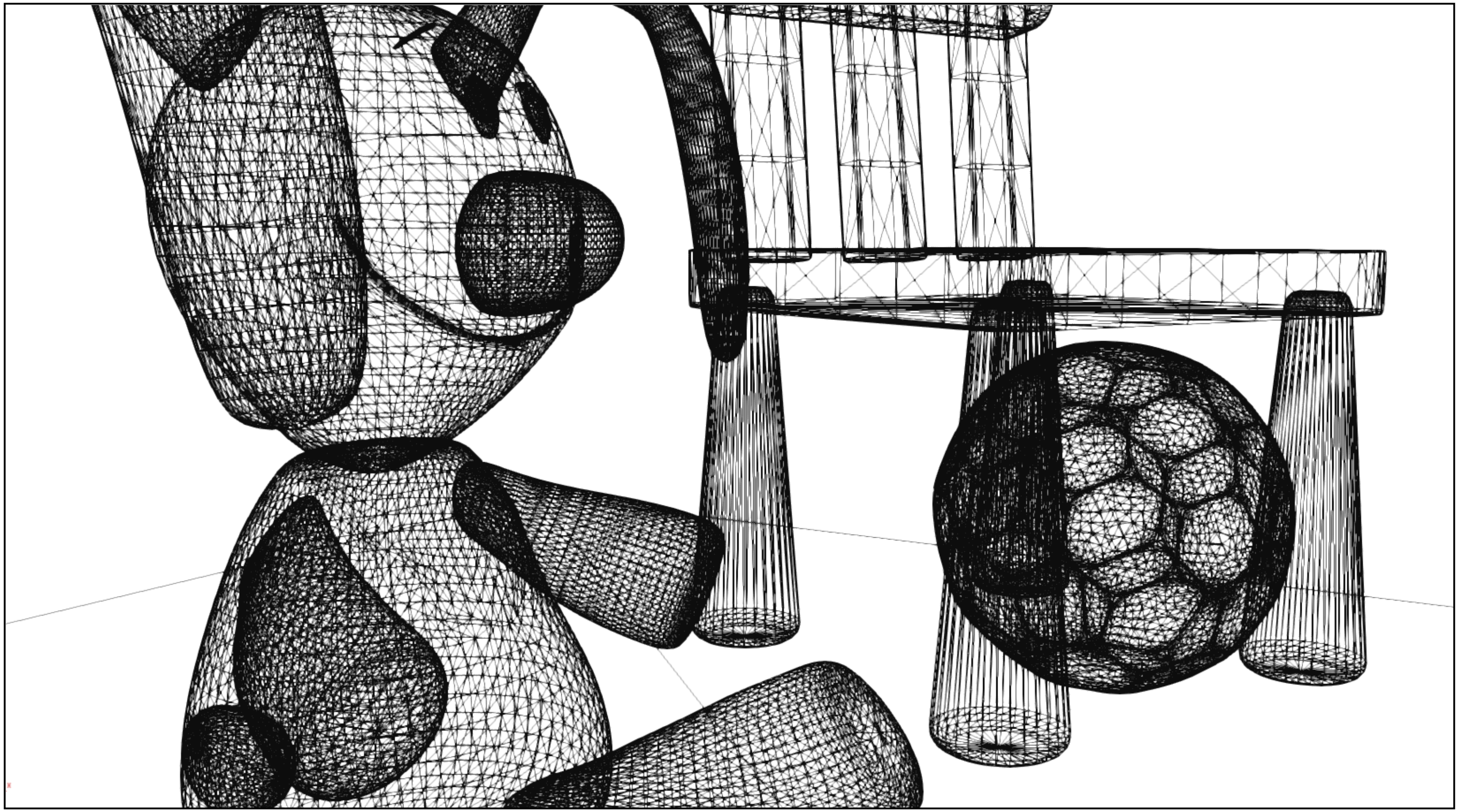
- Primitives after clipping (still in normalized device coordinates)



Clipped primitives

3D Geometry Setup (8)

- Coordinates of assembled primitives after window transformation (image space – pixel units)

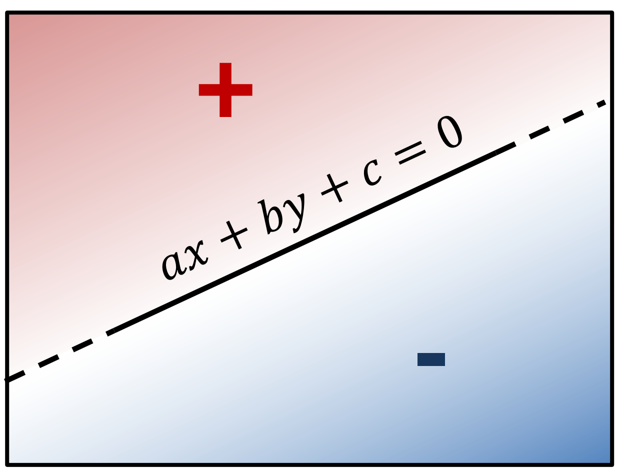


Clipping - General

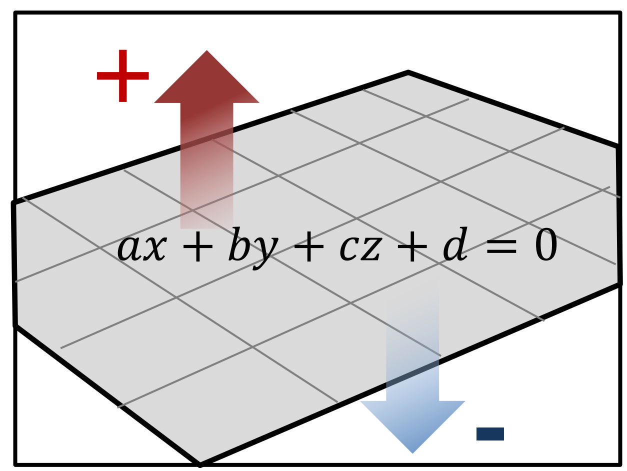
- With clipping we limit the extents of primitives to the viewing region
  - Avoid erroneous projection of geometry (see frustum clipping)
  - Discard invisible geometry
- In general, we clip lines and polygons in both 2D and 3D

Half-spaces

- A hyperplane in 2D (a line) or in 3D (a plane) divides space in two halves
- The corresponding equation is positive on one side, negative on the other and zero exactly on the hyperplane:



2D

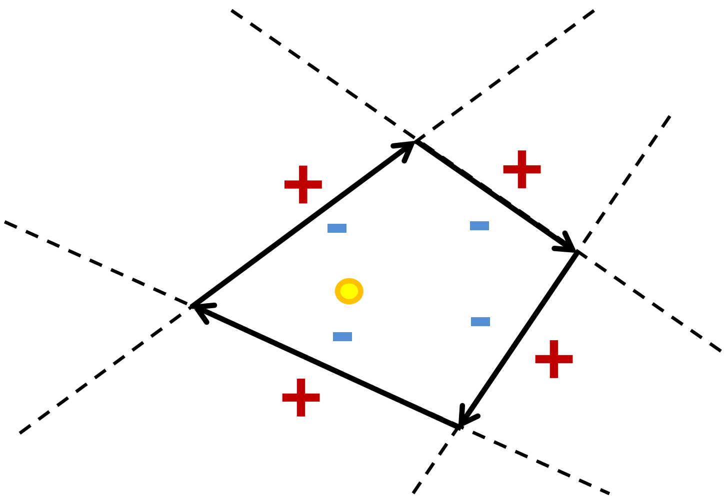


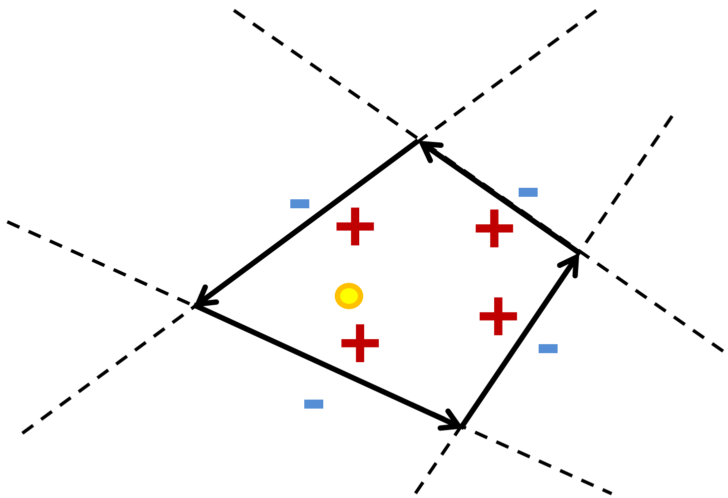
3D

Point Containment

- If a set of oriented hyperplanes f_i forms a convex region, then determining if a point $\mathbf{p}$ lies inside this region resolves to testing if:

$$sign(f_i(\mathbf{p})) = sign\left(f_j(\mathbf{p})\right), \forall i, j$$



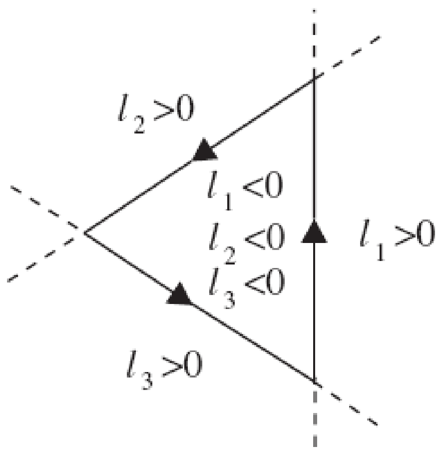


Point in Triangle Test

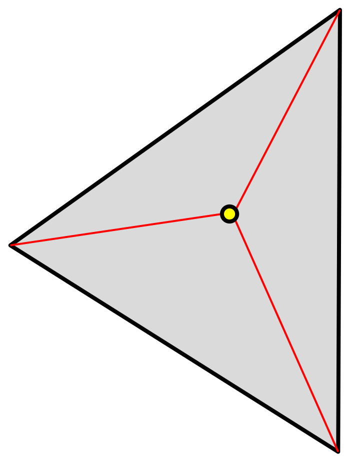
$$sign(y - s \cdot x - b)$$

$$s = \frac{y_n - y_1}{x_n - x_1} = \frac{\Delta y}{\Delta x}$$

$$b = \frac{y_1 x_n - y_n x_1}{x_n - x_1}$$

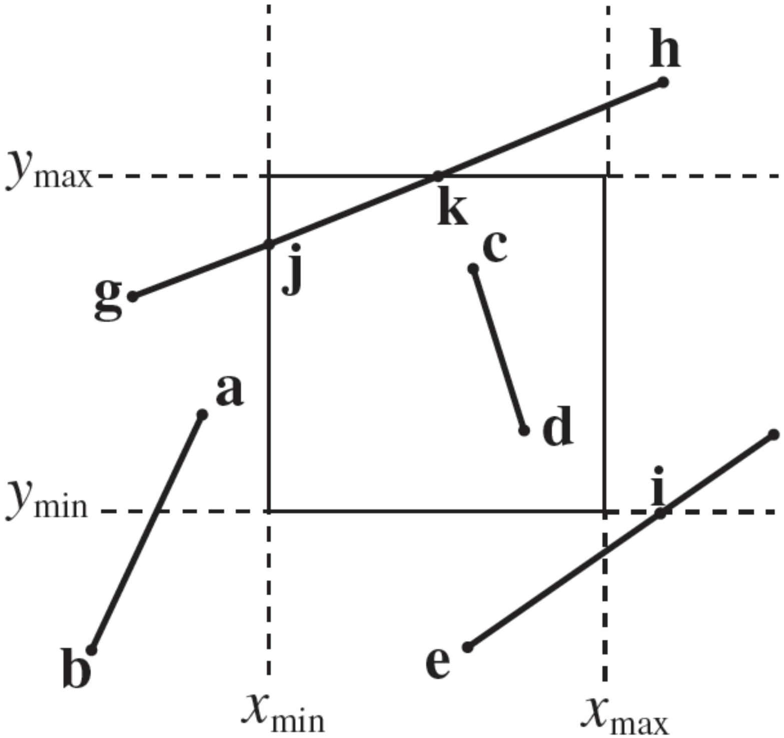


- Alternatively, we can check the barycentric coordinates of the the point w.r.t. the 3 vertices →
  - Inside: $u, v, w \geq 0$



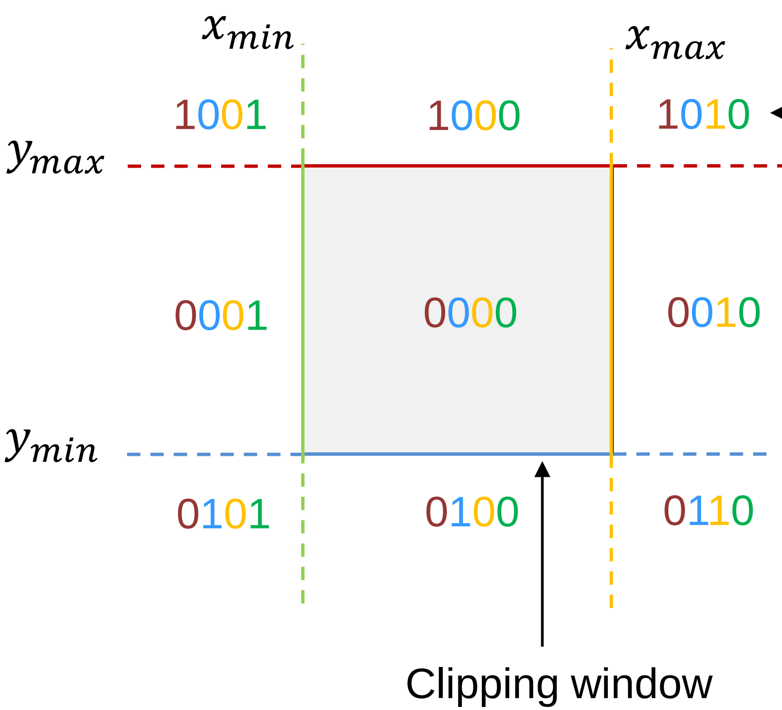
Line Clipping on Rectangular Bounds

- 3 cases:
  - Line segment entirely outside region
  - Line segment entirely inside region
  - Line segment intersects 1 or 2 boundary segments



A Simple Line Clipping Algorithm

- Cohen-Sutherland algorithm
  - Fast segment in/out detection via binary tests
  - Recursive splitting of intersecting segments



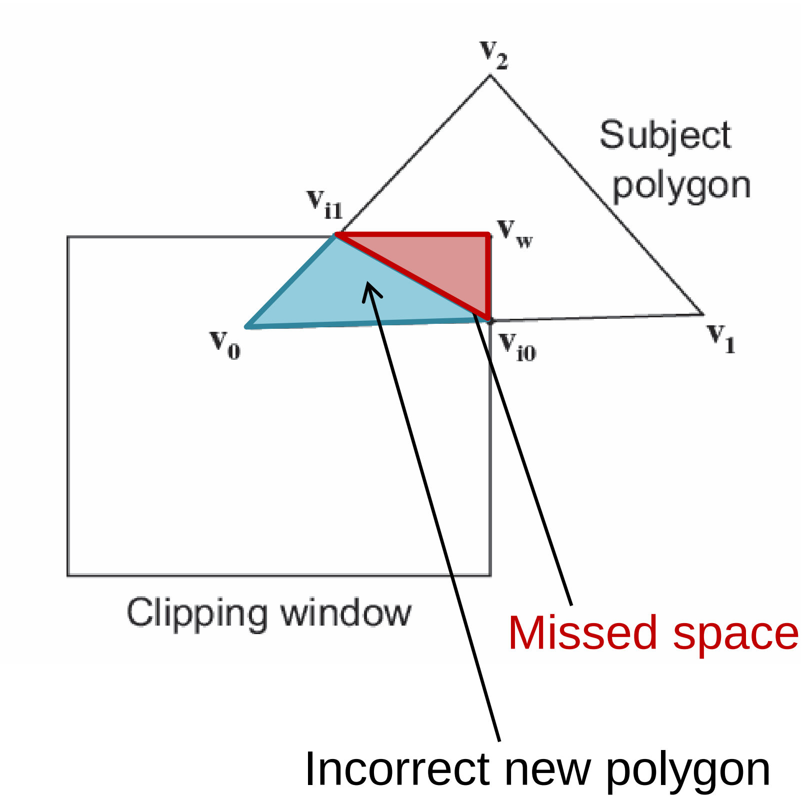
Encode the 9 tiles according to the sign of the 4 line equations

- *First bit.* Set to 1 for $y > y_{\max}$, else set to 0;
- *Second bit.* Set to 1 for $y < y_{\min}$, else set to 0;
- *Third bit.* Set to 1 for $x > x_{\max}$, else set to 0;
- *Fourth bit.* Set to 1 for $x < x_{\min}$, else set to 0.

CS Line Clipping Algorithm

```
void CS( vec3 * P1, vec3 * P2,
         float x_min, float x_max, float y_min, float y_max )
{
     unsigned char c1, c2;
     vec3 I;
     c1=Code(*P1);
     c2=Code(*P2);
     if ( ( c1|c2 == 0 ) || // both inside or
          ( c1&c2 !=0 ) )   // outside but on the same side of a
                            // clipping line (see figure)
                            // do nothing
     else
         {
             Intersect (P1,P2,&I,xmin,xmax,ymin,ymax);
             if ( IsOuside(*P1) )
                 *P1 = I;
             else
                 *P2 = I;
             CS(P1,P2,xmin,xmax,ymin,ymax);
         }
}
```

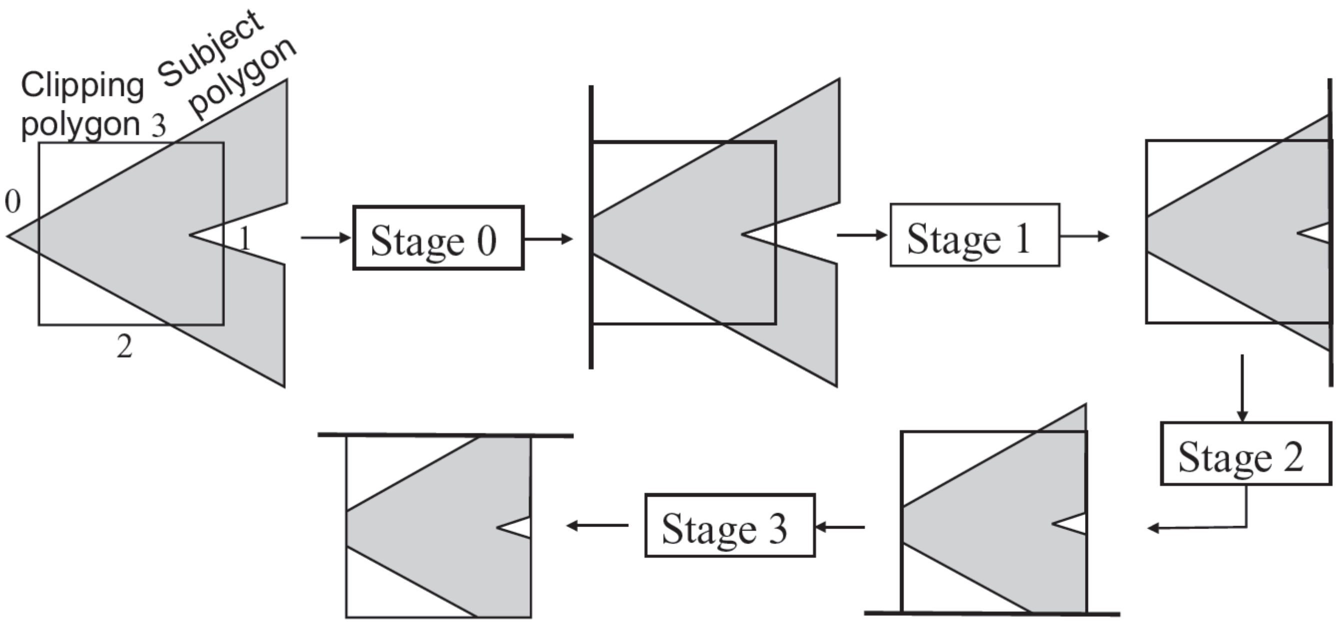
Polygon Clipping



- Polygon clipping cannot be regarded as multiple line clipping!
- Requires mutual edge + point containment and intersection testing

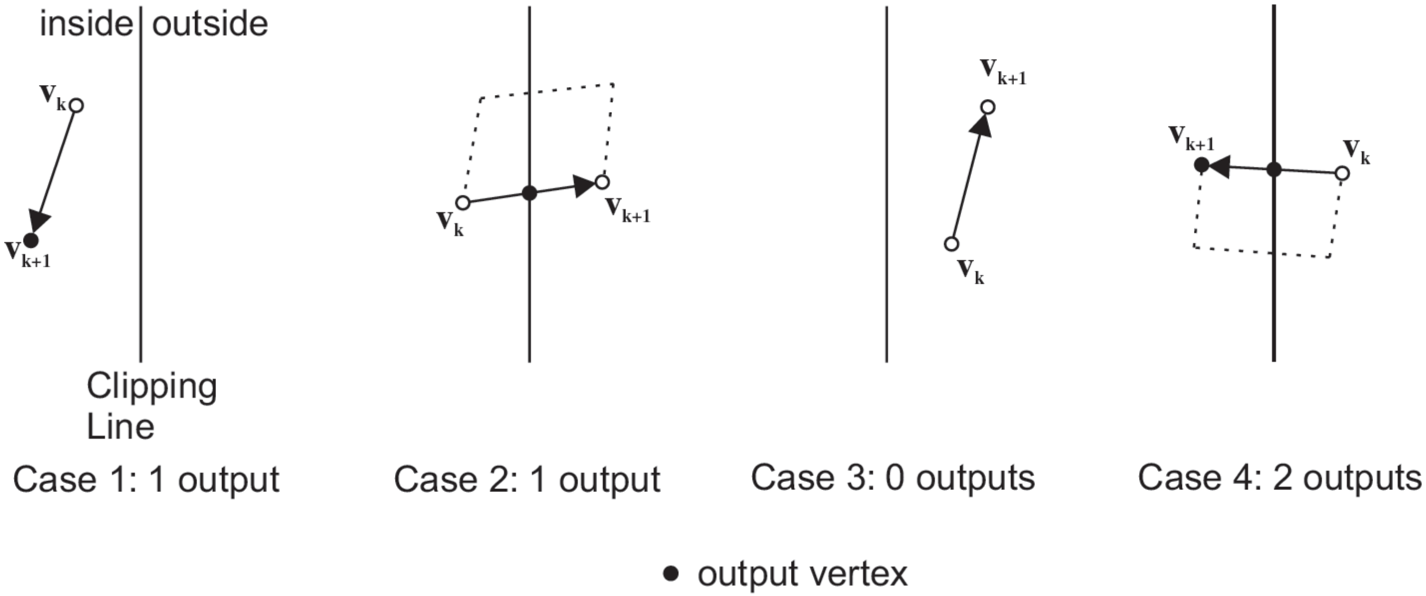
Sutherland-Hodgman Clipping Algorithm (1)

- Clips an arbitrary polygon against a convex clipping polygonal region
- Iteratively clips the input polygon against each one of the segments of the clipping region



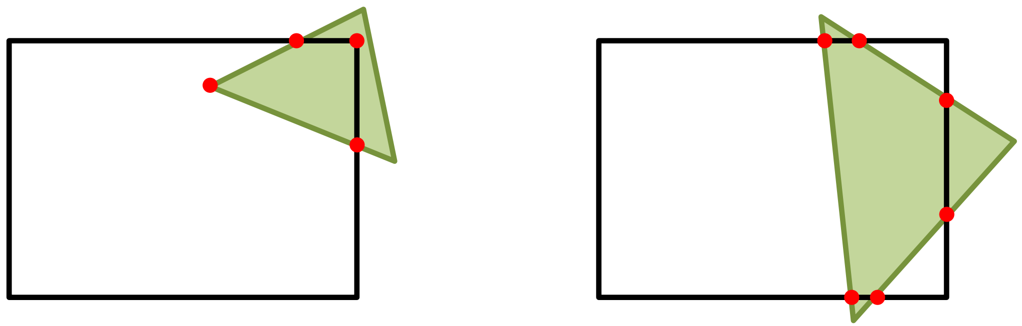
Sutherland-Hodgman Clipping Algorithm (2)

- For each clipping line:
  - For each vertex transition of the input polygon:
    - Determine what points to generate according to the following configurations
  - Join all sequentially generated vertices to form a polygon
  - Use this polygon as input to the next iteration

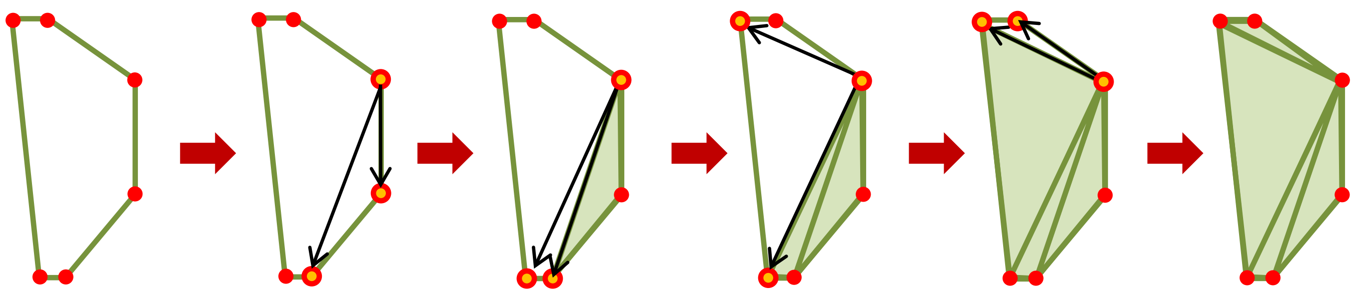


Convex Shape Re-triangulation

- Clipped triangles against the viewing window may require re-triangulation



- Triangulation of convex shapes is trivial:



Frustum Clipping (1)

- Before rasterizing the polygons, they must be clipped against the view frustum (see projections)
- Why?
  - Coordinates behind near plane get inverted and wrap beyond the far plane → degenerate, impossible “triangles”
  - Coordinates on z=0 → singularity in perspective division

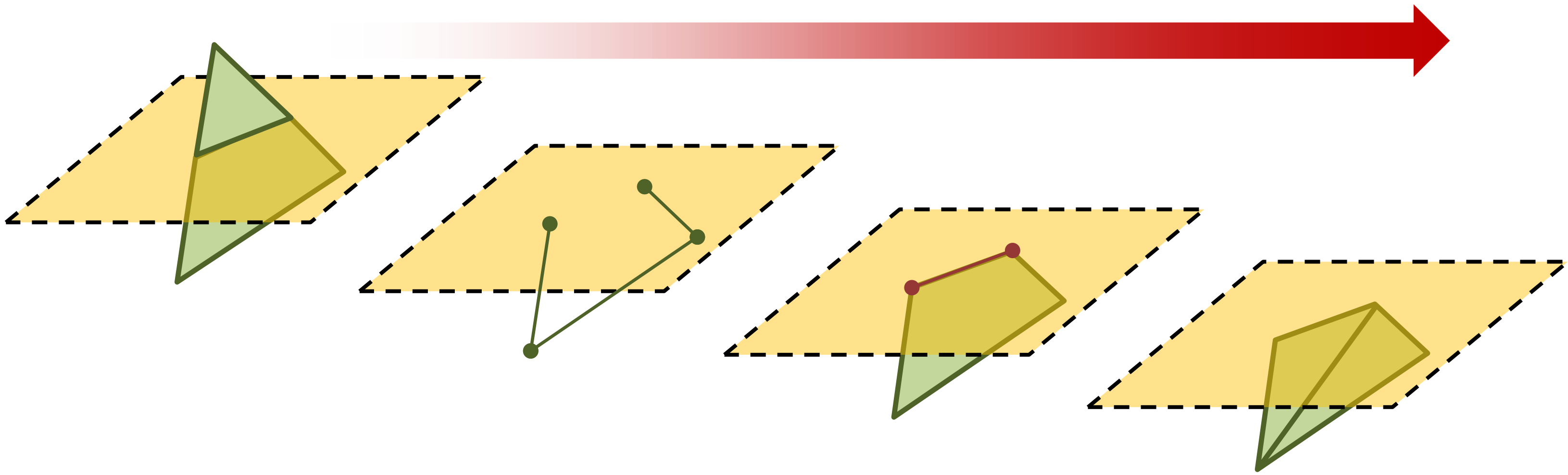
Frustum Clipping (2)

- Frustum clipping can be done with a Sutherland-Hodgman-style method for triangles/planes
- For a 6-plane frustum (i.e. the camera frustum), this is a 6-stage triangle/plane clipping pipeline

- Clipping is performed in the post-projective space, before the perspective division. Why?
  - In all projections (perspective, too), the frustum planes are axis aligned → simplified comparisons and equations (see Chapter 5.3 in [G&V]

Frustum Clipping (3)

- Triangle/plane clipping:
  - Perform 2 line-plane clipping steps
  - Join the open edges (if any)
  - Re-triangulate if necessary

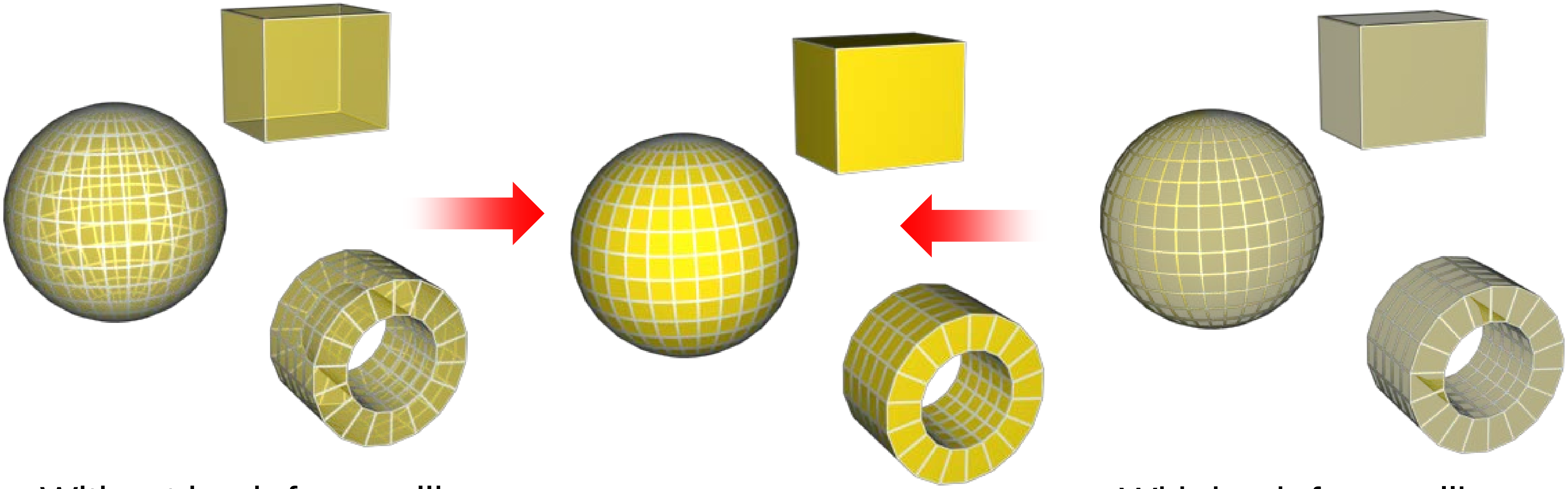


Pixel-level Clipping

- It is possible to perform clipping at a pixel level (or pixel block level, for hierarchical implementations)
- Pixel-level clipping boils down to discarding values outside the usable range (i.e. within the 2D/3D clipping region)
  - Saves on H/W and power consumption (less circuitry)
  - Naïve implementation: Not very fast – many samples to discard
  - Hierarchical / block-based implementation: efficient

NVIDIA patent EP1756769 B1

Optimizations – Back-face Culling (1)

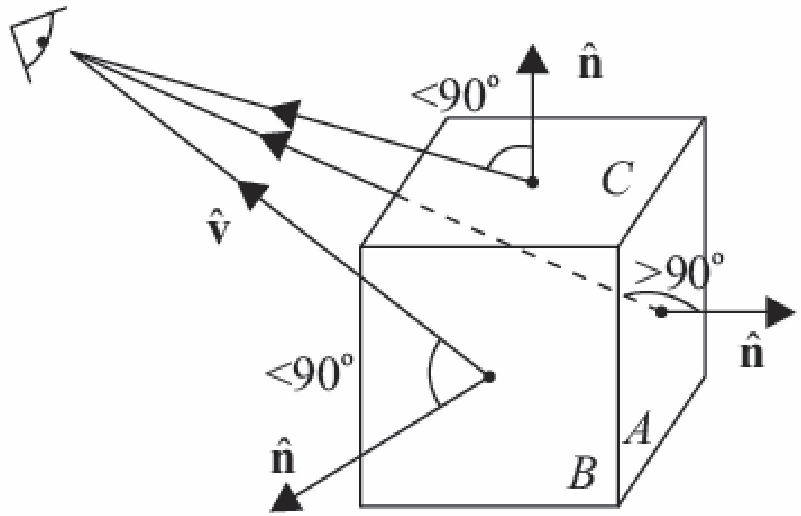


Without back-face culling

With back-face culling
(~50% fewer triangles)

- Back-face culling can dramatically reduce the rasterization load by effectively discarding all polygons facing off the eye direction
- Transparent shapes should not be BF culled

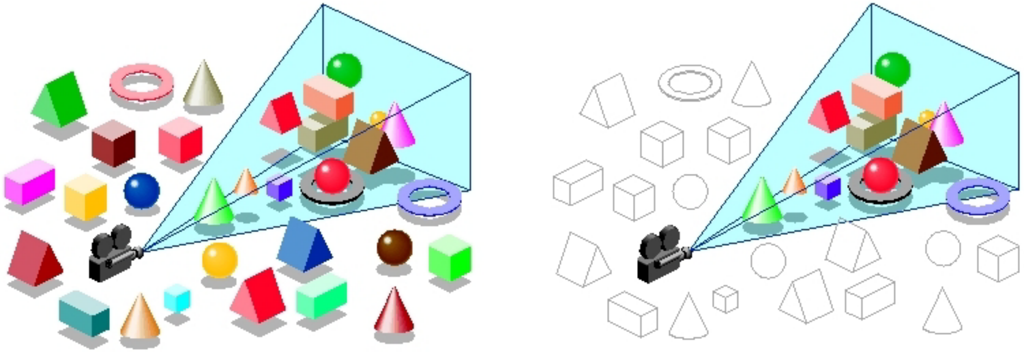
Optimizations – Back-face Culling (2)



- Back-face culling rejects polygons whose normal deviates more than 90 degrees from the viewing direction

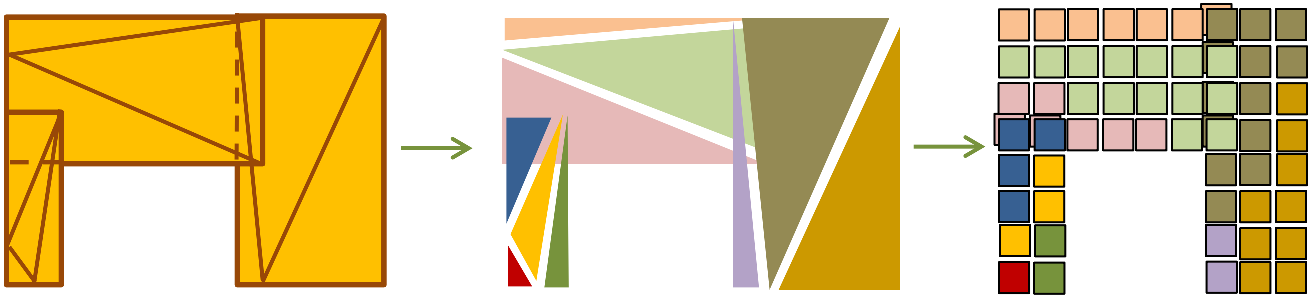
Optimizations - Frustum Culling

- Conservatively discards entire objects early on, before clipping by:
  - Checking the extents (bounding box) of an object against the bounds of the frustum
- This test is very simple in post-projective space:
  - if all projected bounding box corners are outside the frustum → cull the object
  - Can be extended to non-camera frusta to cull hidden objects



Rasterization

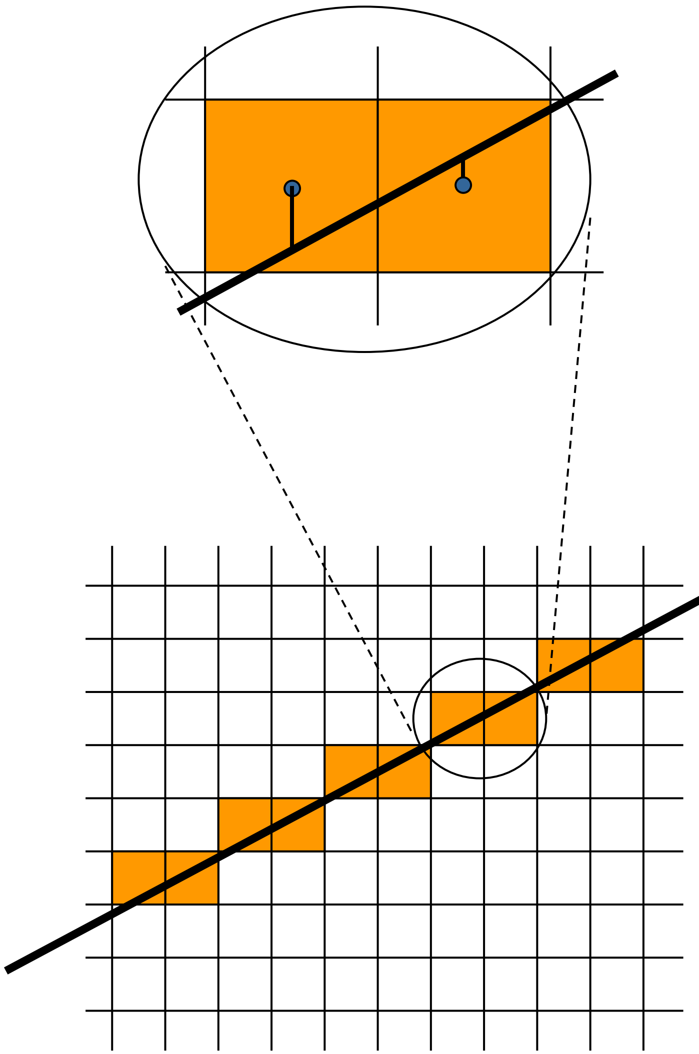
- Rasterization is the process that generates the pixel-based samples on the stream of primitives
- Before rasterization occurs, it is convenient to transform the primitives in screen coordinates (i.e. pixel units) – see rasterization slides
- Each primitive is processed independently!



Fragments from different primitives may overlap → Ordering must be resolved (see next slides)

Line Rasterization

- Must:
  - Approximate the mathematical line as close as possible (min. error)
  - Not leave any gaps
  - Maintain a constant width
  - Be efficient



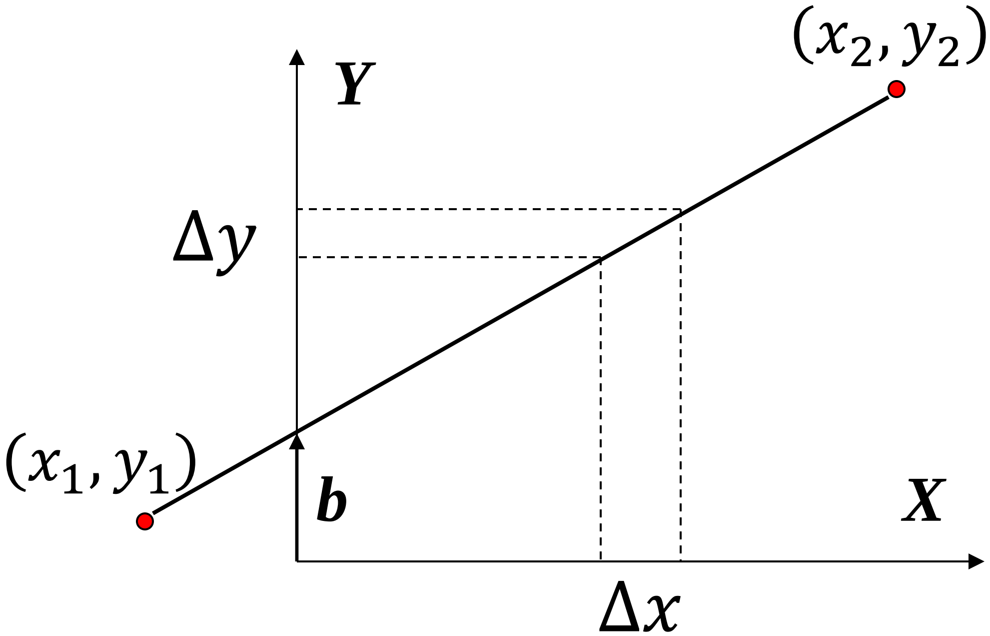
Approximating the Line Equation (1)

- Given a line segment in the first octant $(x_1, y_1) \rightarrow (x_2, y_2)$, the line passing through the endpoints is defined as:

$$y = s \cdot x + b$$

$$s = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

$$b = \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1}$$

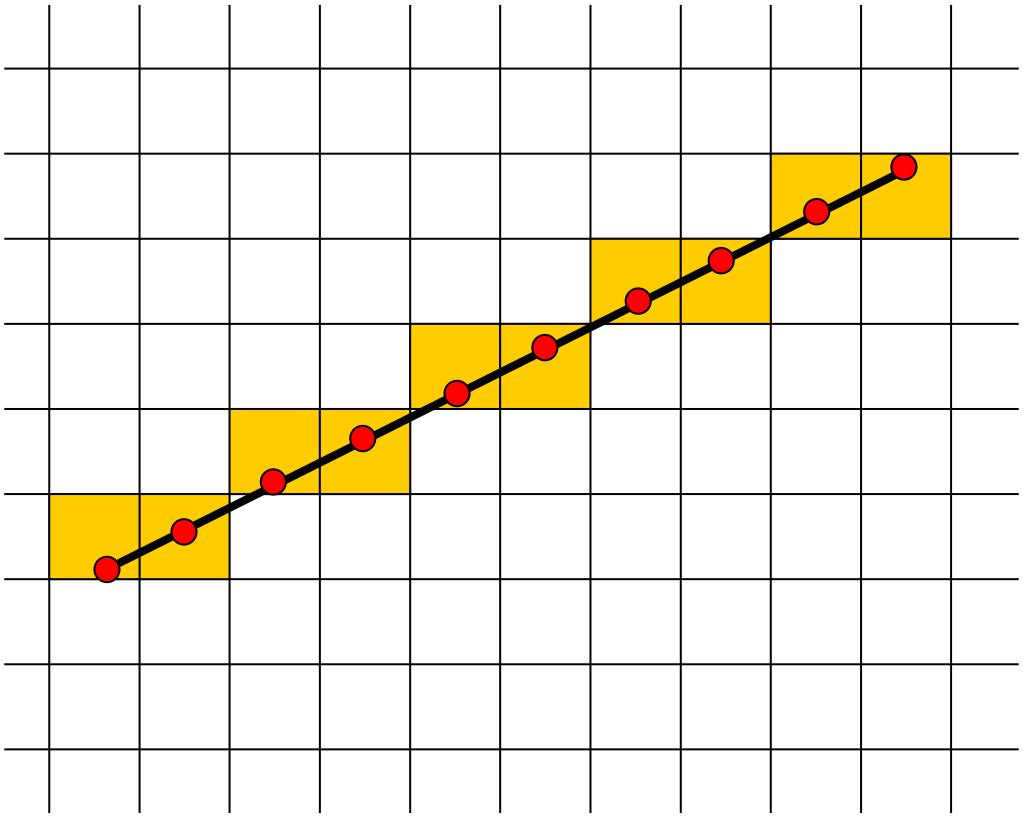


Approximating the Line Equation (2)

```
void Line1( float x1, float y1, float x2, float y2 )
{
    float s, b, y;
    float x;
    s = (y2-y1) / (x2-x1);
    b = (y1*x2 – y2*x1) / (x2-x1);
    for ( x = x1; x <= x2; x+=1.0f )
    {
        y = s*x + b;
        SetPixel( floor(x+0.5f), floor(y+0.5f) );
    }
}
```

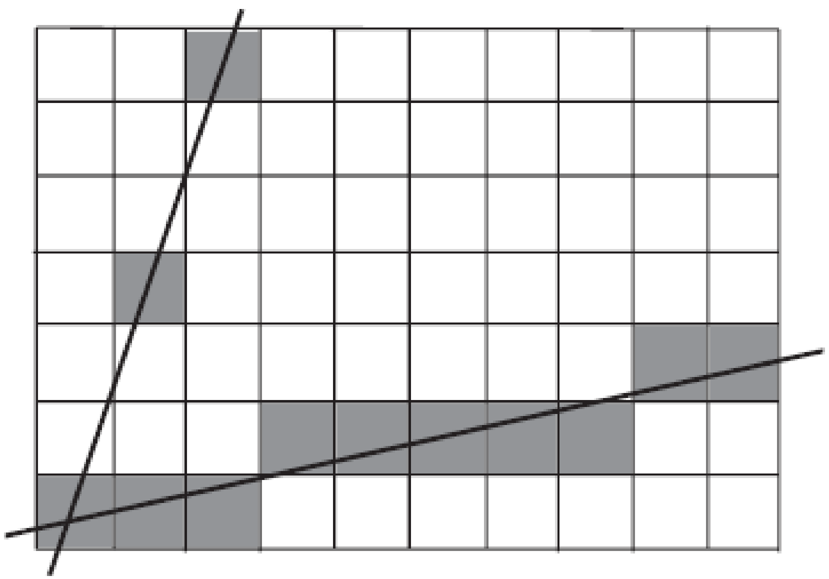
Result of the Line1 Algorithm

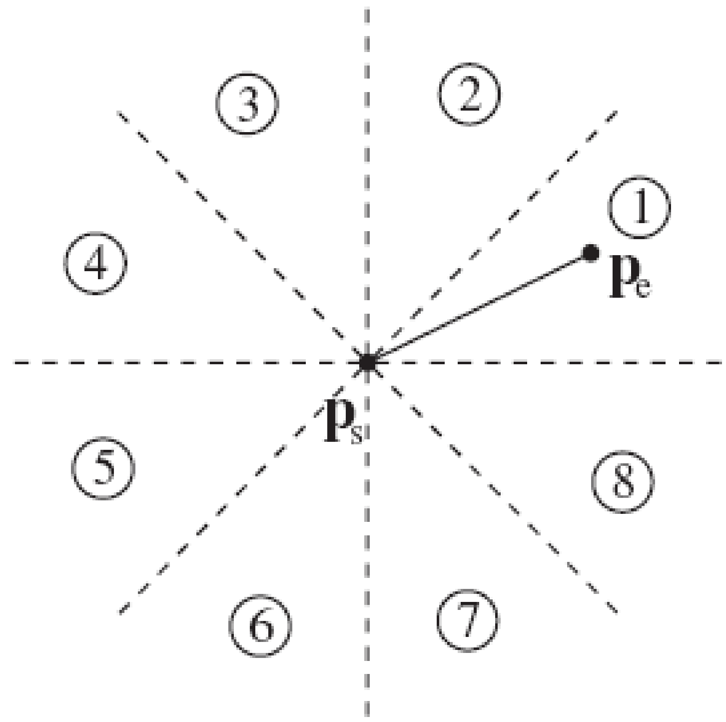
- Y values are eventually rounded to the nearest integer cell



Incremental Line Algorithm (1)

- Y values are computed for fixed and positive X increments
- The described algorithm (Line1) is valid only for octant 1:





Incremental Line Algorithm (2)

- The multiplication inside the loop can be simplified, since:

$$x_{i+1} = x_i + 1$$

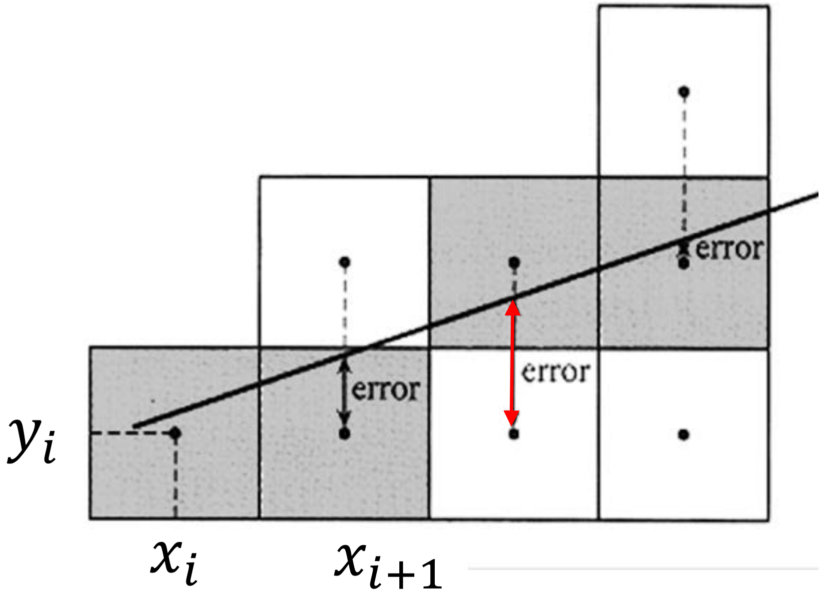
$$y_{i+1} = sx_{i+1} + b = sx_i + b + s = y_i + s$$

Incremental Line Algorithm (3)

```
void Line2( float x1, float y1, float x2, float y2 )
{
    float s, y;
    float x;
    s = (y2-y1) / (x2-x1);
    y = y1;
    for ( x = x1; x <= x2; x+=1.0f )
    {
        SetPixel( floor(x+0.5f), floor(y+0.5f) );
 →      y = y+s;
    }
}
```

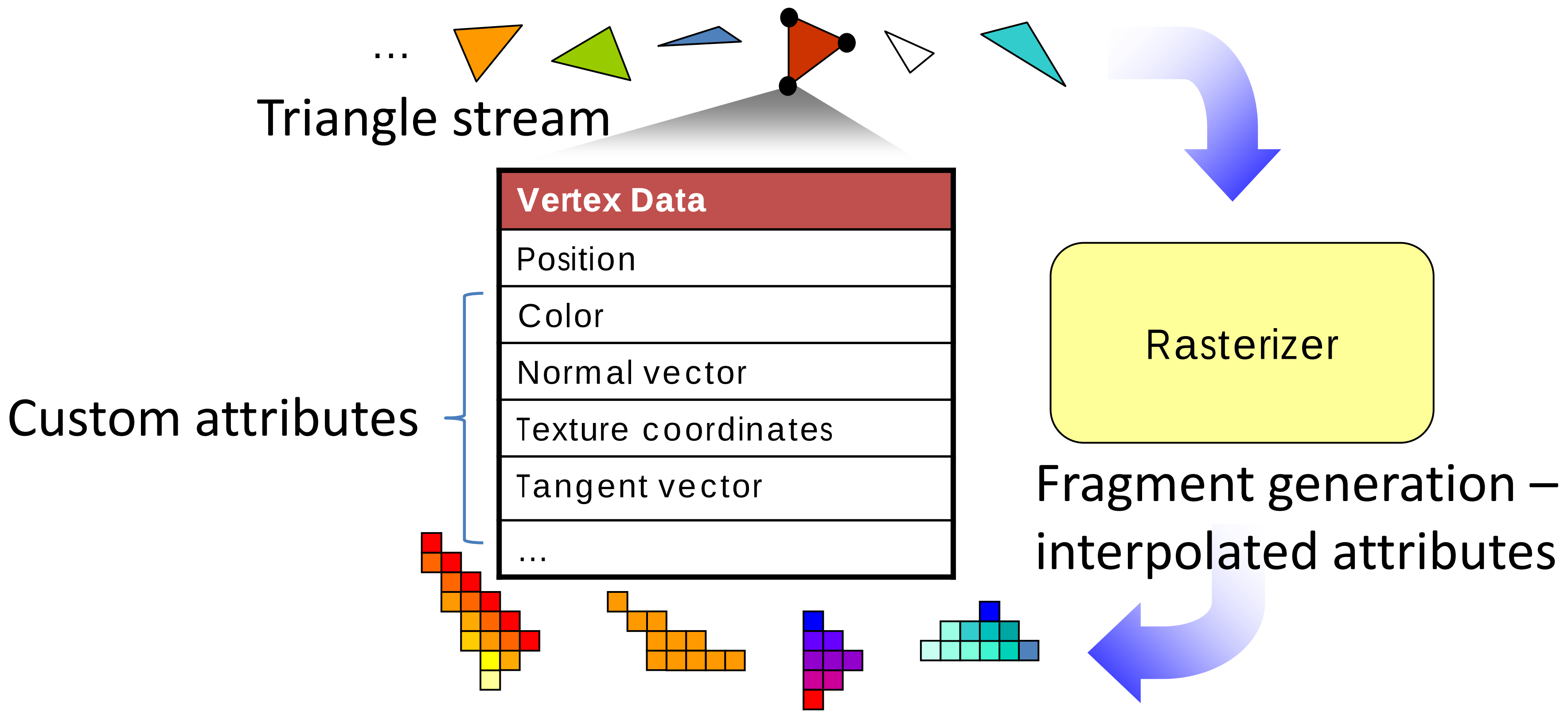
Integer Variants of Line Drawing

- If all coordinates are integer values, there are several improvements to be made to save calculations:
  - Drop the rounding, by stepping to the next Y value if the increment becomes larger than 1/2 pixel
  - Scaling all comparisons by Δx to dispense with the division



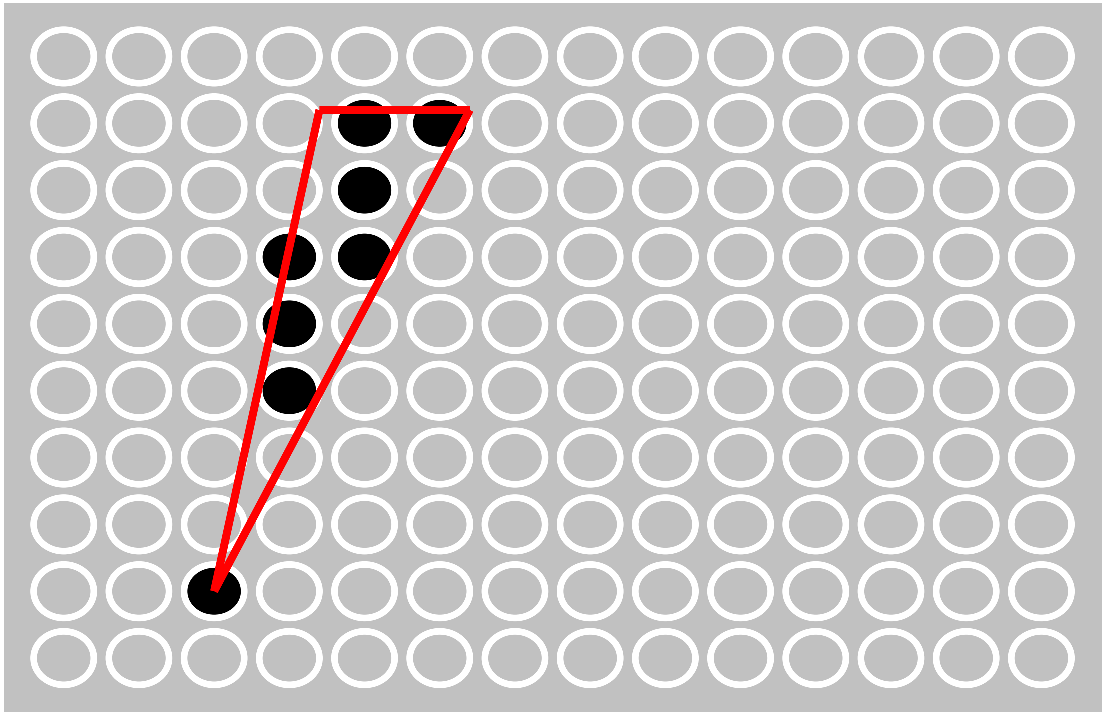
Rasterization – Triangle Traversal (1)

- Sampling the triangles involves traversing their interior and edges and generating a set of fragments per pixel (typically one)



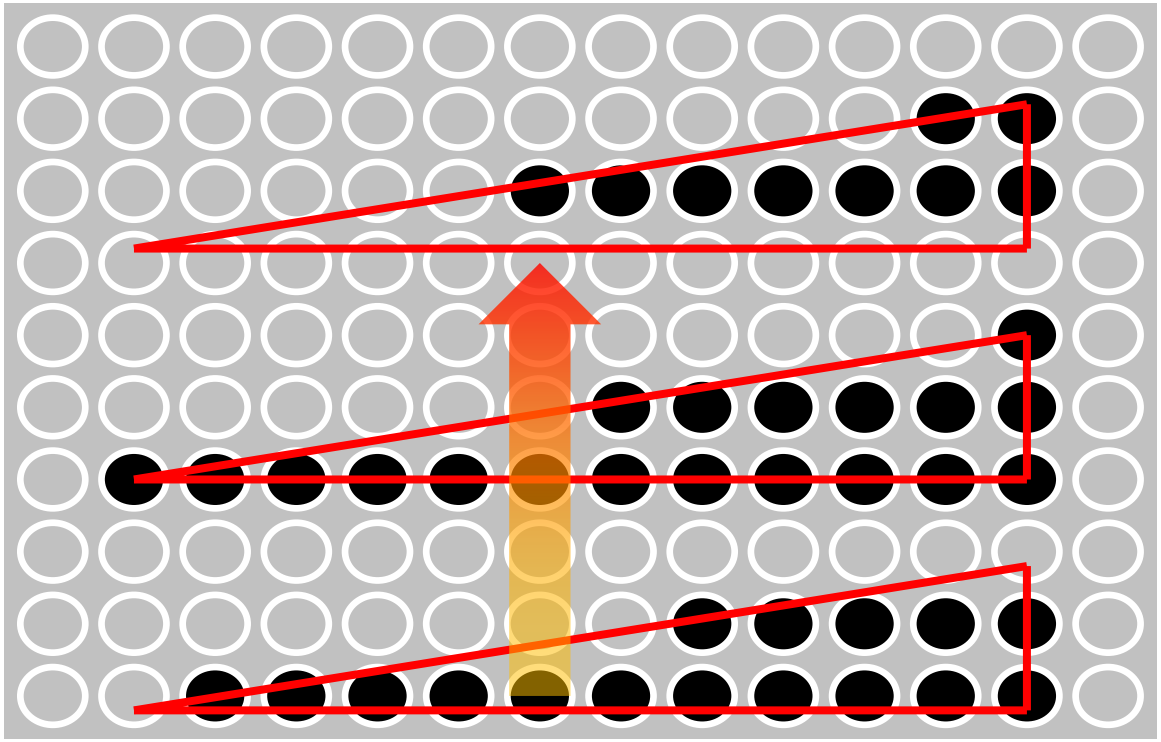
Triangle Rasterization Issues (1)

- Similar to lines, triangle rasterization must not leave gaps, for thin triangles:



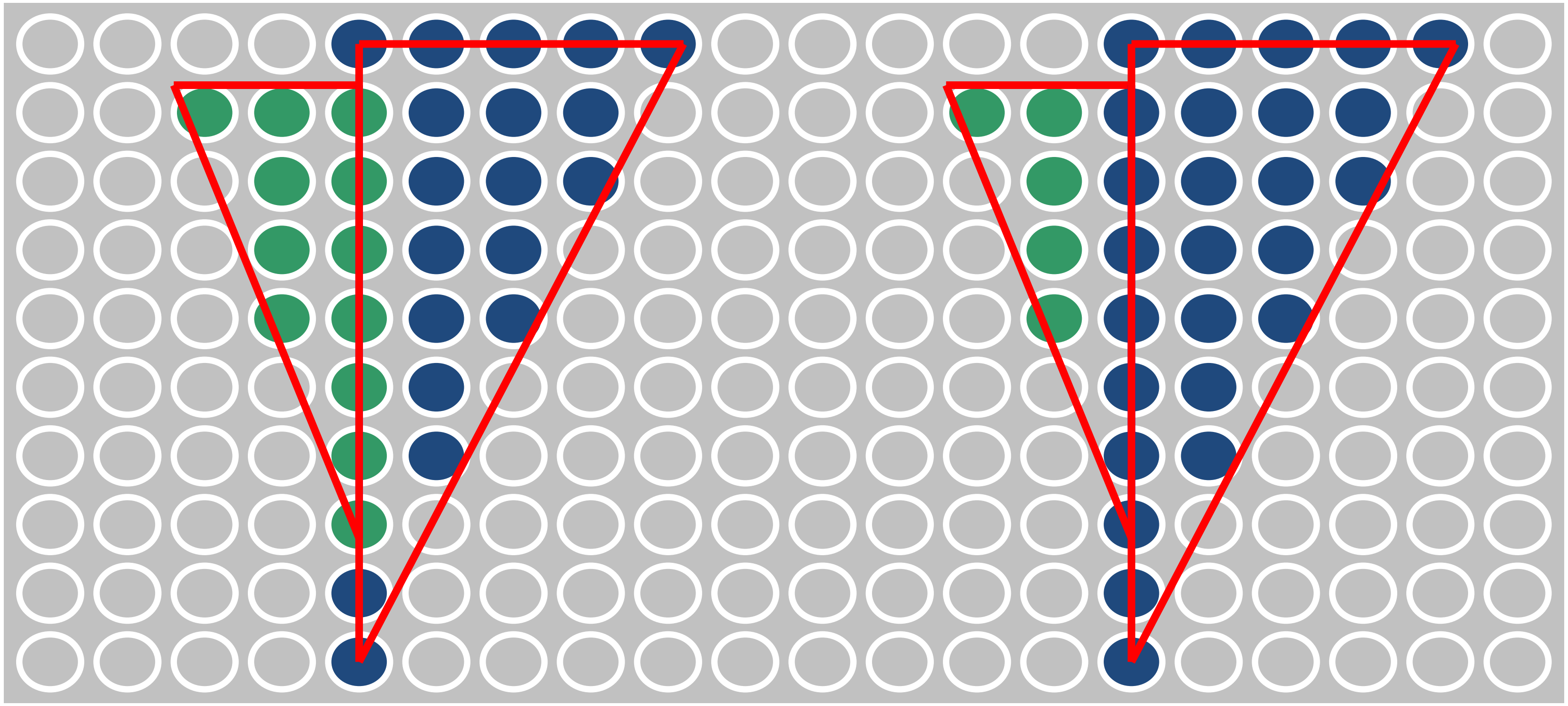
Triangle Rasterization Issues (2)

- Appearance must be as consistent as possible under slight sampling offsets (motion) – see antialiasing



Triangle Rasterization Issues (3)

- What is the priority of shared edges?



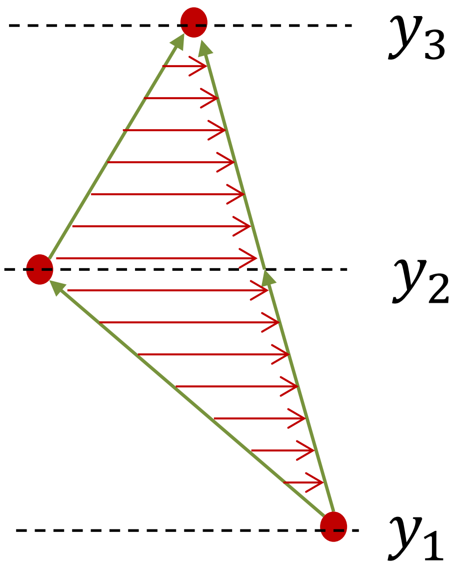
Triangle Traversal Algorithms

- Two dominant methods:
  - Edge Walking: Vertically follows edges and draws the corresponding scan line spans
  - Edge Equation: Tests the pixels for containment inside the triangle boundaries. Can be efficiently implemented in a divide and conquer manner

Edge Walking – Basic Idea

(AKA: Triangle Digital Differential Analyzer)

- Follow edges vertically
- Interpolate attributes down edges
- Fill in horizontal spans for each scanline
  - For each pixel of a scanline, interpolate edge attributes across span

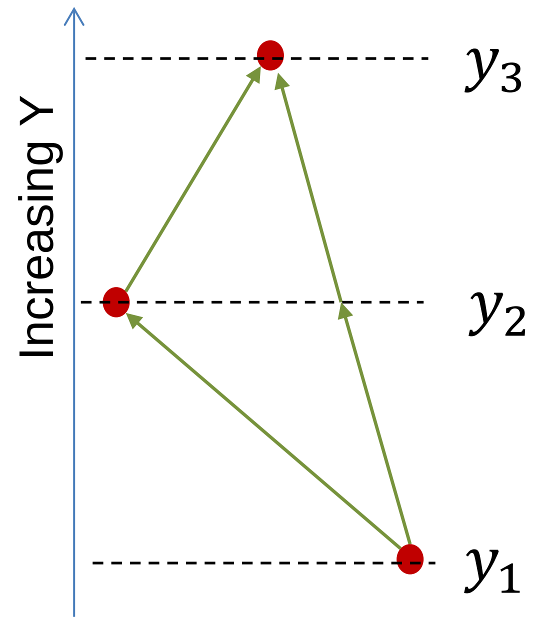


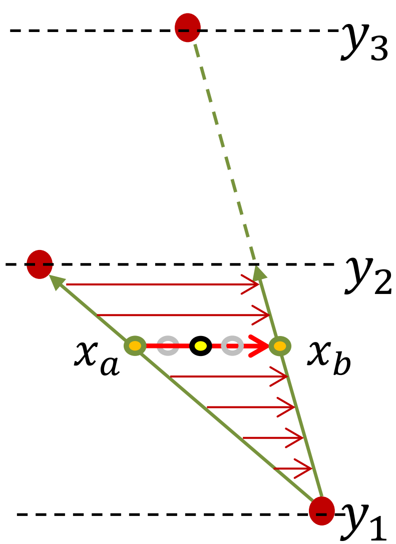
Edge Walking – Procedure

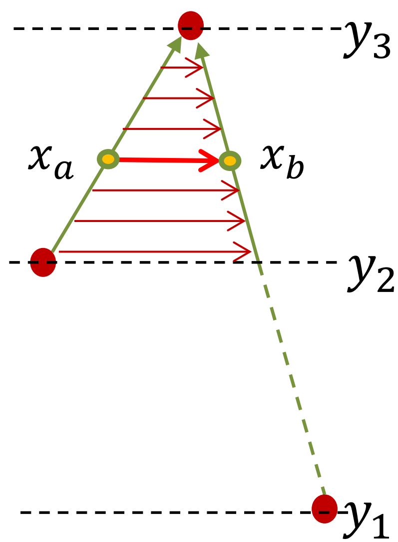
Sort Vertices by Y value

Scan Convert 2 sub-triangles:

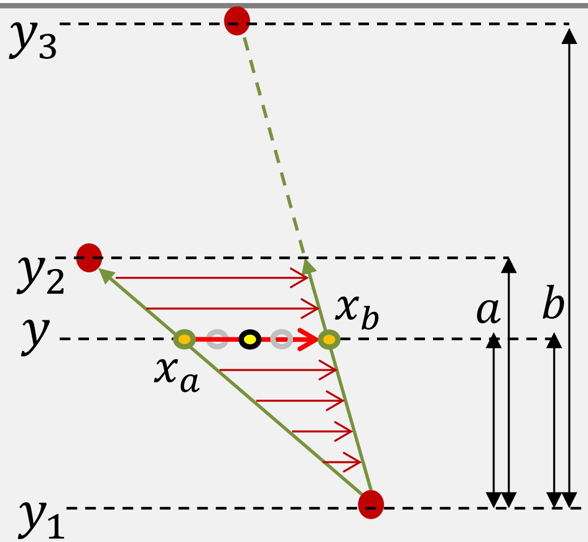
- For $y_1 \leq y < y_2$:
  - Interpolate x (x_a, x_b) and other values along edges
  - For $x_a \leq x < x_b$: interpolate values along spans
- For $y_2 \leq y < y_3$:
  - Interpolate x (x_a, x_b) and other values along edges
  - For $x_a \leq x < x_b$: interpolate values along spans







Edge Walking – Attribute Interpolation

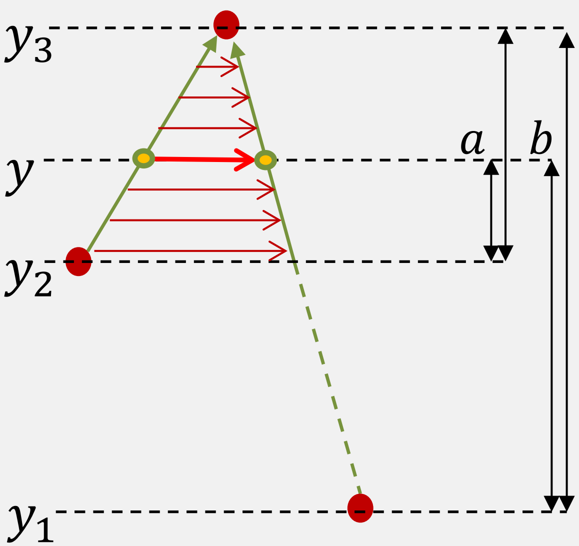


$$a = \frac{y - y_1}{y_2 - y_1}$$

$$b = \frac{y - y_1}{y_3 - y_1}$$

$$x_a = x_1 + a(x_2 - x_1)$$

$$x_b = x_1 + b(x_3 - x_1)$$



$$a = \frac{y - y_2}{y_3 - y_2}$$

$$b = \frac{y - y_1}{y_3 - y_1}$$

$$x_a = x_2 + a(x_3 - x_2)$$

$$x_b = x_1 + b(x_3 - x_1)$$

Inner loop (x)

$$s = \frac{x - x_a}{x_b - x_a}$$

$$z = z_a + s(z_b - z_a)$$

$$\xi_1 = \xi_{1a} + s(\xi_{1b} - \xi_{1a})$$

$$\xi_2 = \xi_{2a} + s(\xi_{2b} - \xi_{2a})$$

$$\vdots$$

$$\xi_n = \xi_{na} + s(\xi_{nb} - \xi_{na})$$

Any attribute ξ_k is similarly interpolated

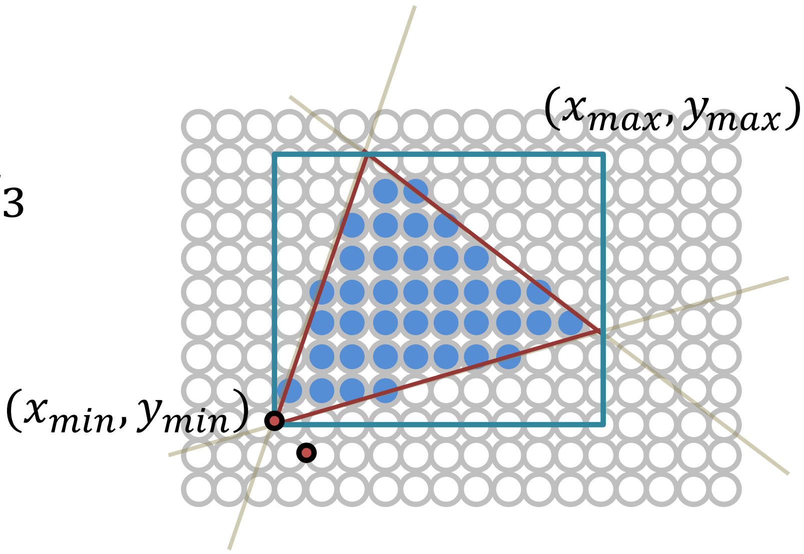
Ok, We Have a Traversal, Why Go for Another One?

- Scanline-style edge walking is reasonably good provided that you don't care about:
  - Aligned (coherent) memory access
  - Parallelism: multiple rows at a time
  - Variable sample positions
  - Ability to harness wide SIMD or build efficient hardware for it
- The above become really problematic especially in the case of thin, elongated triangles

Edge Equation Traversal – Basic Idea

- Triangle setup:
  - Find the bounding box of the triangle
  - Find the edge (line) equations of the oriented edges
  - Find triangle differentials
- For all pixels in the grid:
  - Find edge equation values $\varepsilon_1, \varepsilon_2, \varepsilon_3$
  - If $(\varepsilon_1 > 0) \wedge (\varepsilon_2 > 0) \wedge (\varepsilon_3 > 0)$
    - Interpolate attributes
    - Issue Fragment

Embarrassingly parallel!

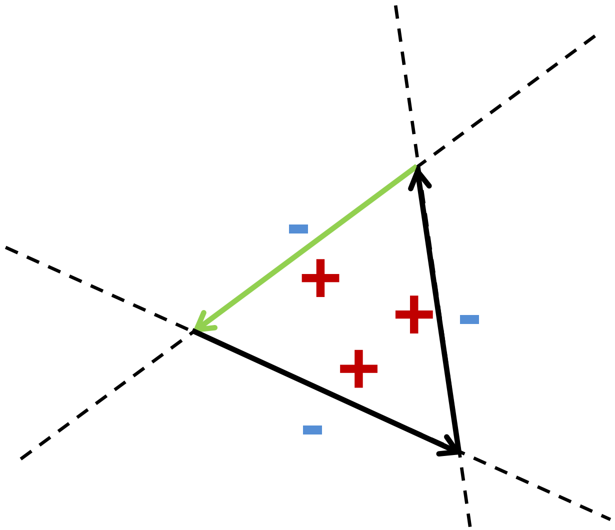


Edge Equation Values

$$y = s \cdot x + b \Longrightarrow \boxed{e = sx - y + b}$$

$$s = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

$$b = \frac{y_1 x_2 - y_2 x_1}{x_2 - x_1}$$

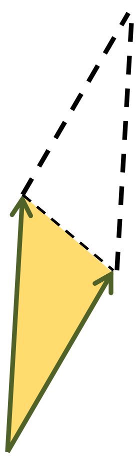


Value Interpolation

- Use barycentric coordinates!
- Can I incrementally construct the barycentric coordinates per pixel?
  - YES!
  - We can also incrementally update the edge equations per pixel

Edge Equation Traversal – Revisited (1)

- Given two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$, the following determinant calculates the signed area of the formed parallelogram:
$$\mathrm{A}_p(\mathbf{v}_1, \mathbf{v}_2) = \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}$$
- Or the signed area of the triangle formed by $\mathbf{v}_1$ and $\mathbf{v}_2$:
$$\mathrm{A}_t(\mathbf{v}_1, \mathbf{v}_2) = \frac{1}{2}\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix}$$
- Remember, these quantities are signed
- The sign is determined by the order of the two vectors

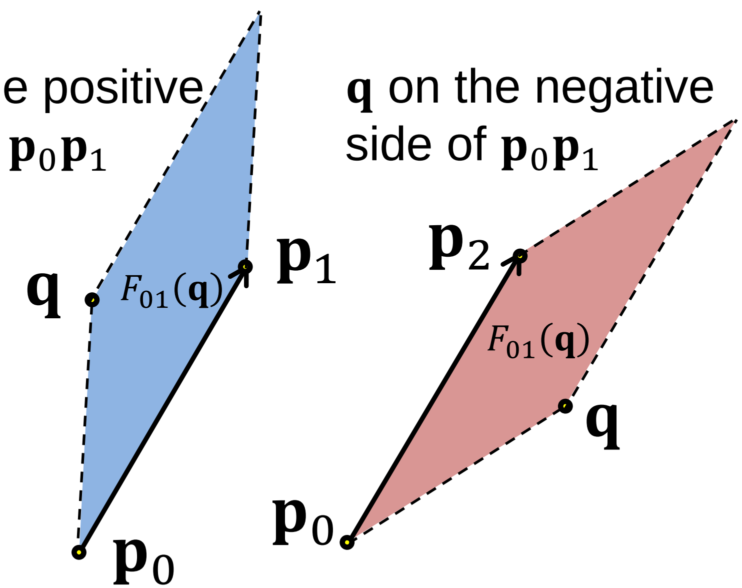


Edge Equation Traversal – Revisited (2)

- Now consider an edge $\mathbf{p}_0\mathbf{p}_1$ of a triangle and an arbitrary point $\mathbf{q}$
- Using as vectors $\mathbf{v}_1 = \mathbf{p}_0\mathbf{p}_1$ and $\mathbf{v}_2 = \mathbf{p}_0\mathbf{q}$ the determinant defines an edge function of $\mathbf{q}$ w.r.t. edge $\mathbf{p}_0\mathbf{p}_1$:

$\mathbf{q}$ on the positive side of $\mathbf{p}_0\mathbf{p}_1$

$\mathbf{q}$ on the negative side of $\mathbf{p}_0\mathbf{p}_1$



$$F_{01}(\mathbf{q}) = \begin{vmatrix} x_1 - x_0 & x_q - x_0 \\ y_1 - y_0 & y_q - y_0 \end{vmatrix}$$

Edge Equation Traversal – Revisited (3)

- Expanding and rearranging $F_{01}(\mathbf{q})$ we get:

$$F_{01}(\mathbf{q}) = \begin{vmatrix} x_1 - x_0 & x_q - x_0 \\ y_1 - y_0 & y_q - y_0 \end{vmatrix} \Leftrightarrow$$

$$F_{01}(\mathbf{q}) = (y_0 - y_1)x_q + (x_1 - x_0)y_q + (x_0 y_1 - y_0 x_1)$$

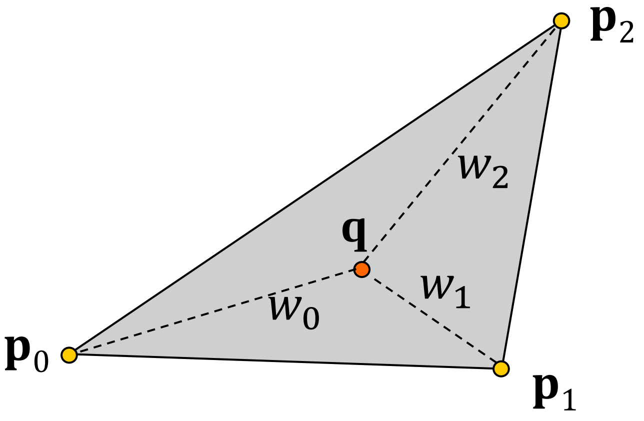
- Equivalently, for the other triangle edges:

$$F_{12}(\mathbf{q}) = (y_1 - y_2)x_q + (x_2 - x_1)y_q + (x_1 y_2 - y_1 x_2)$$
$$F_{20}(\mathbf{q}) = (y_2 - y_0)x_q + (x_0 - x_2)y_q + (x_2 y_0 - y_2 x_0)$$

Source: http://fgiesen.wordpress.com/2013/02/06/the-barycentric-conspirac/

Edge Equation Traversal – Revisited (4)

- Remember that $F_{01}(\mathbf{q})$ is related to the area of the triangle $\mathbf{p}_0\mathbf{p}_1\mathbf{q}$
- But so is the barycentric coordinate of $\mathbf{q}$ from $\mathbf{p}_2$!
- It is easy to see that if w_0, w_1, w_2 are the 3 barycentric coordinates, then:



$$w_0 = F_{12}(\mathbf{q})/w$$
$$w_1 = F_{20}(\mathbf{q})/w$$
$$w_2 = F_{01}(\mathbf{q})/w$$
$$w = F_{01}(\mathbf{q}) + F_{12}(\mathbf{q}) + F_{20}(\mathbf{q})$$

Incremental Traversal (1)

- Lets take the edge function and simplify it:

$$F_{01}(\mathbf{q}) = (y_0 - y_1)x_q + (x_1 - x_0)y_q + (x_0 y_1 - y_0 x_1) =$$
$$A_{01}x_q + B_{01}y_q + C_{01}$$

- The terms A_{01}, B_{01}, C_{01} as well as the respective terms of the other edge functions are constant per triangle
  - Can be computed once in the triangle setup phase

Incremental Traversal (2)

- Let's look now what happens for adjacent pixel coordinates:

$$F_{01}(x_q+1, y_q) = A_{01}(x_q+1) + B_{01}y_q + C_{01} = F_{01}(x_q, y_q) + A_{01}$$
$$F_{01}(x_q, y_q+1) = A_{01}x_q + B_{01}(y_q+1) + C_{01} = F_{01}(x_q, y_q) + B_{01}$$

- So, shifting the calculation to 1 pixel ahead in either direction only involves the addition of a constant term!

Source: http://fgiesen.wordpress.com/2013/02/10/optimizing-the-basic-rasterizer/

Parallel Traversal

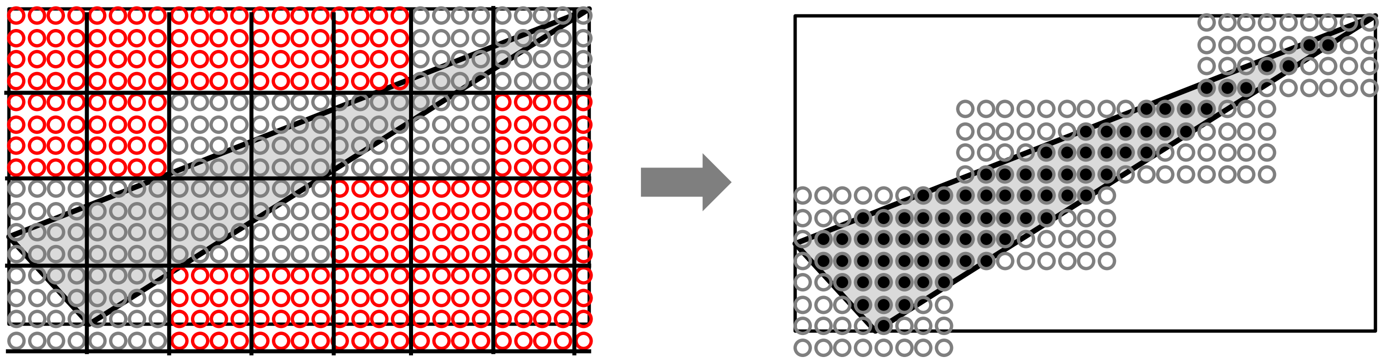
- More importantly, for parallel (vectorized) computations:

$$F_{ij}(x_{UL} + n, y_{UL} + m) = F_{ij}(x_{UL}, y_{UL}) + nA_{ij} + mB_{ij}$$

- where (x_{UL}, y_{UL}) is the upper-left corner of the bounding box
- The barycentric coordinates (interpolation variables) are computed from F_{ij} → These are independently and cheaply computed, too!

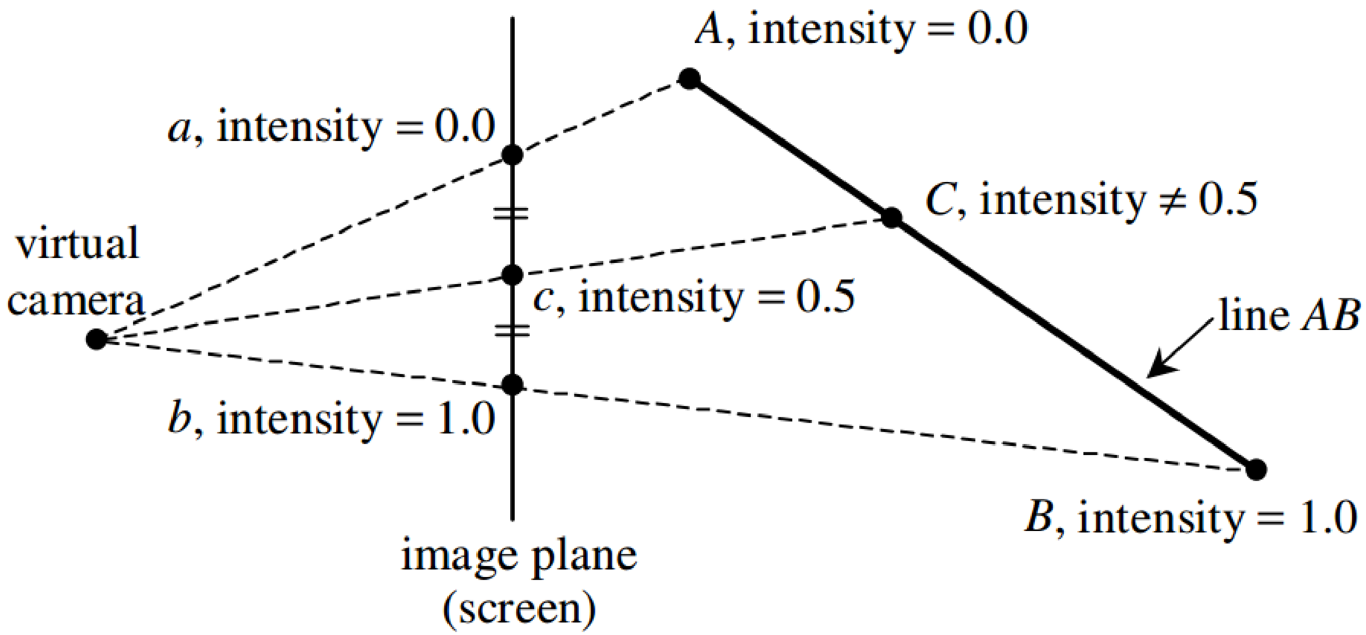
Edge Equation Traversal – Optimization (1)

- We can effectively reduce further the computations if we process the bounding box in blocks and discard entire blocks
  - Block discard: all block corners outside the triangle
  - Can be done hierarchically



Perspective and Interpolation (1)

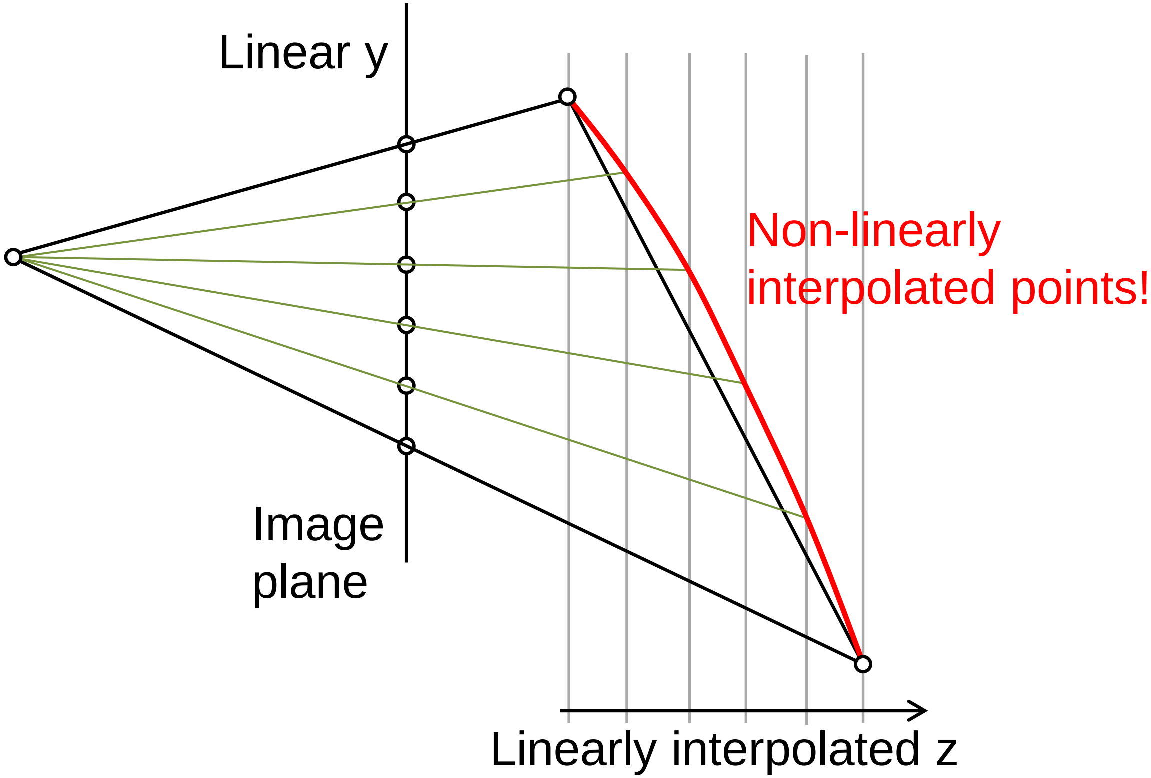
- Is there a problem with interpolating in perspective?
  - Screen-space interpolation does not correctly interpolate perspectively projected values:



Source: Kok-Lim Low, Perspective-Correct Interpolation, Tech. Rep. 2002

Perspective and Interpolation (2)

- Linear in screen space → Non-linear in eye space!



Perspective and Interpolation (3)

- Fortunately, we can derive functions that correctly perform this interpolation
- For the perspectively correct z:

$$z_s = \frac{1}{\frac{1}{z_1} + s\left(\frac{1}{z_2} - \frac{1}{z_1}\right)}$$

- i.e., interpolate 1/z values and invert the result

- For the derivation procedure see: Kok-Lim Low, Perspective-Correct Interpolation, Tech. Rep. 2002

Perspective and Interpolation (3)

- For the perspectively correct fragment attributes:

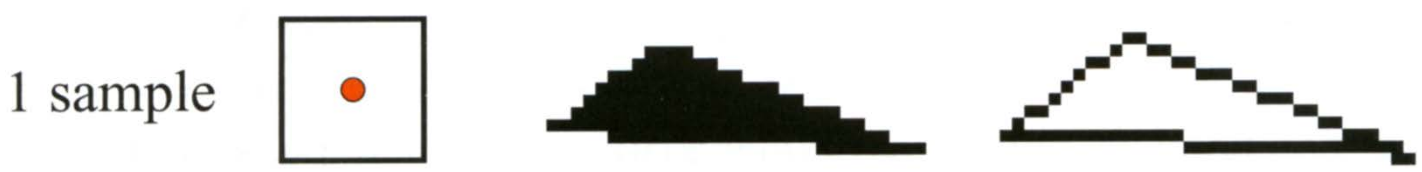
$$a_s = z_s \left(\frac{a_1}{z_1} + s \left(\frac{a_2}{z_2} - \frac{a_1}{z_1} \right) \right)$$

- i.e., divide vertex attributes by the corresponding z and multiple interpolated result by interpolated z

- For the derivation procedure see: Kok-Lim Low, Perspective-Correct Interpolation, Tech. Rep. 2002

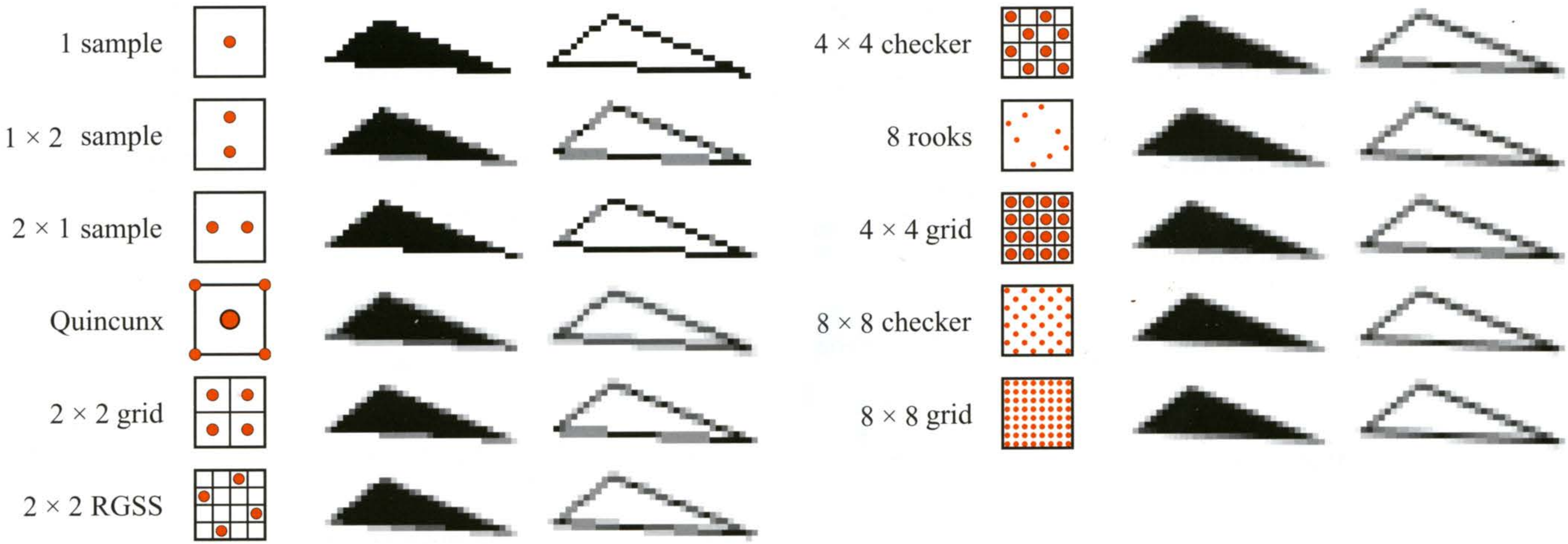
Geometry Antialiasing

- Aliasing in geometry boundaries due to fixed-rate sampling is a common artifact manifested as "pixelization"
  - Blocky appearance
  - Improper representation of thin structures
  - Temporal artifacts



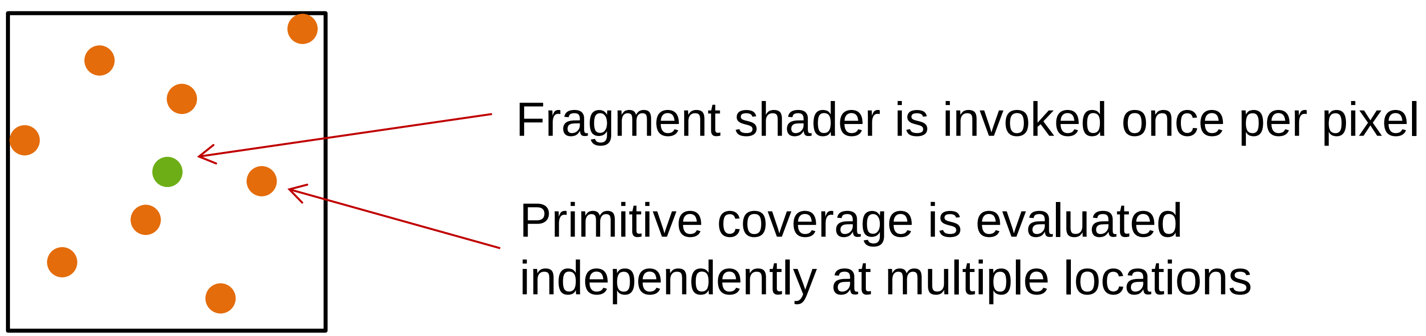
Super-sampling the Geometry

- The problem is alleviated by mitigating the sampling issues to a higher sampling frequency by super-sampling each pixel

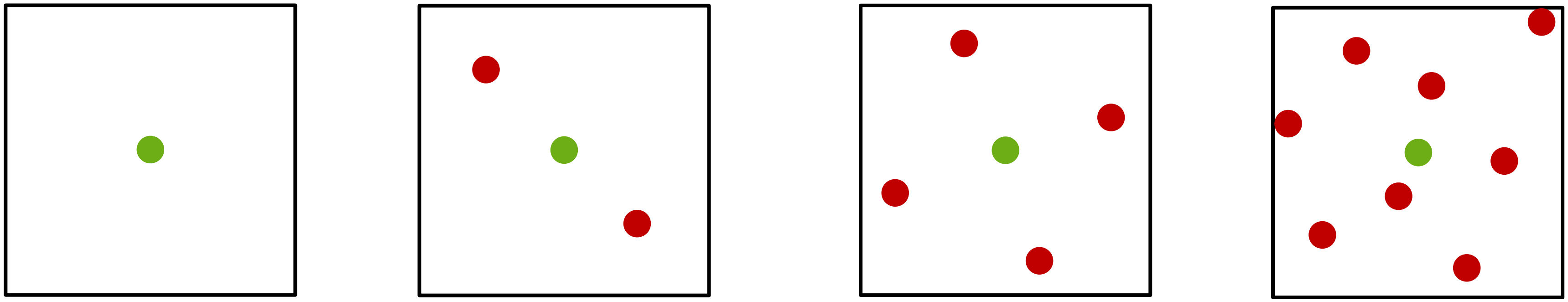


Practical Antialiasing - MSAA

- Supersampling the pixel normally implies evaluating the shading at all samples taken →
  - Cost: × number of samples!
- Solution: Evaluate the shading at a single location and take multiple coverage samples independently → MSAA (Multi-Sampled Anti-Aliasing)



MSAA - Example

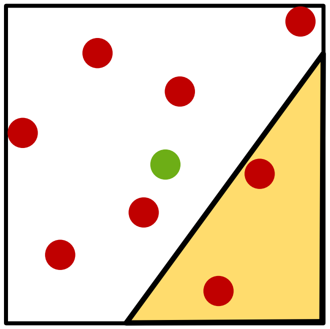


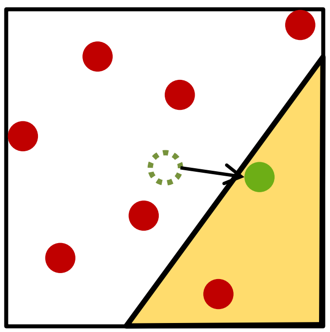
1X (no MSAA), 2X, 4X and 8X coverage samples on an NVIDIA 780Ti graphics card

Fragment shader evaluation location

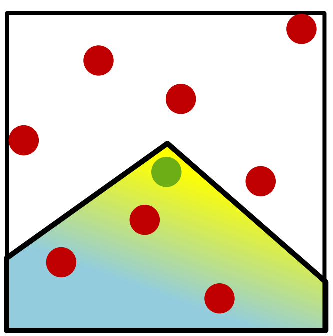
Coverage sample

MSAA - Deficiencies





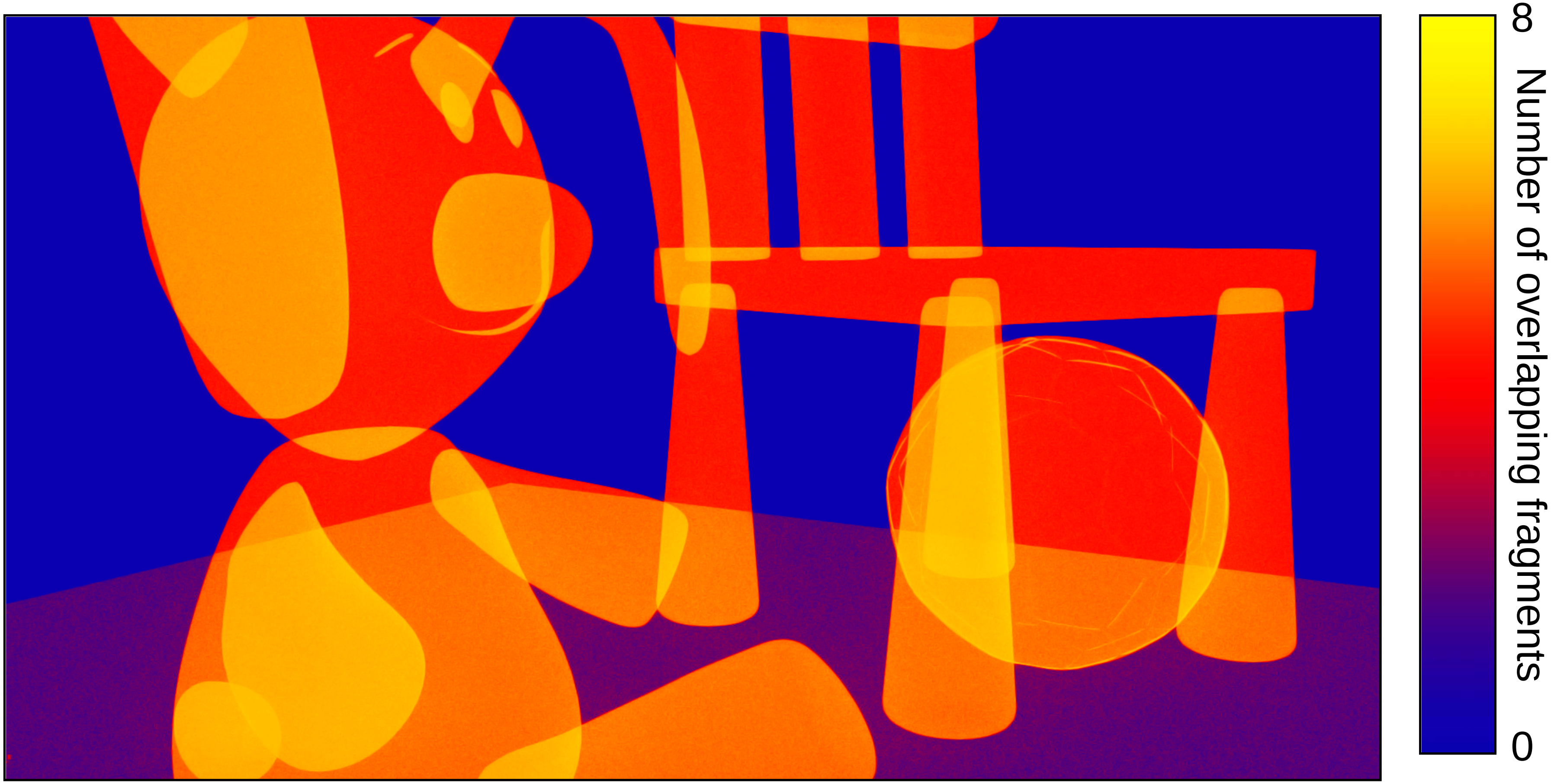
- Shader computations may be performed for locations outside the geometry!
  - Can be fixed by moving the shading to the covered sample closest to the center



- Attributes evaluated at the pixel center my not be representative of the covered area

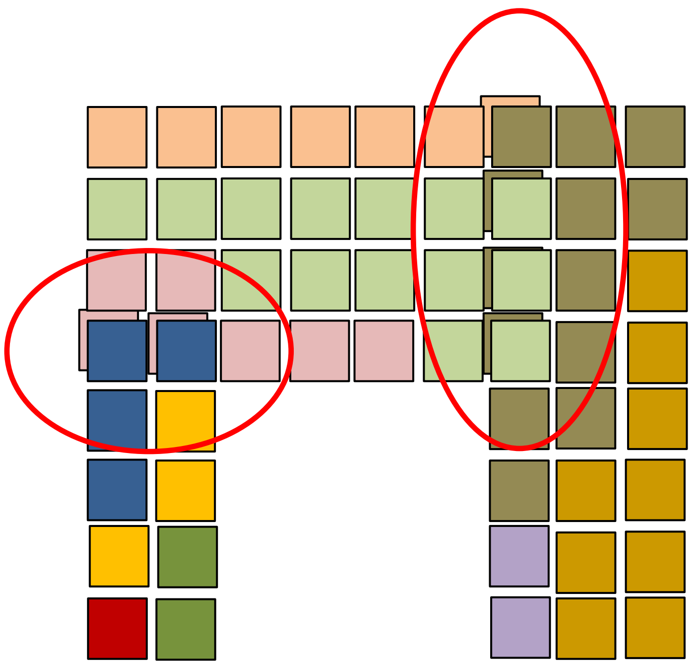
Triangle Rasterization - Overdraw

- Rasterized fragments overlap with previously drawn fragments from other triangles – not yet sorted



Sorting (1)

- The fragments of a primitive typically overlap fragments from other primitives

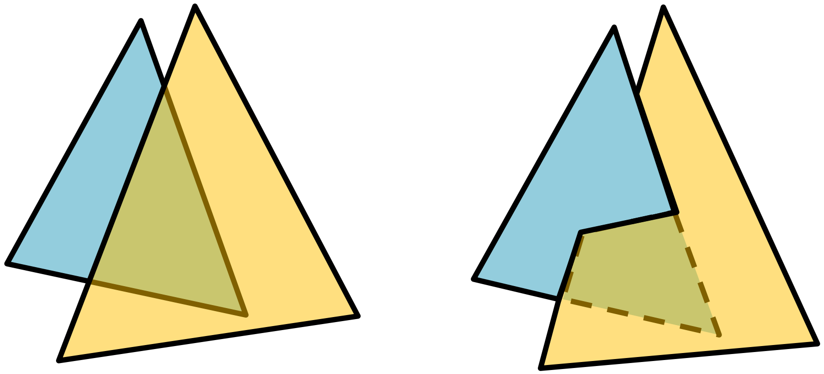


- There are many strategies to resolve the ordering of the rasterized primitives as they appear on screen
- Simplest:
  - Explicit order (FIFO)
- 3D: More elaborate schemes required (see 3D rasterization)

Sorting (2)

- Sorting can occur in various stages of the pipeline, depending on the type of primitives:
  - E.g., flat 2D polygons and lines can be trivially pre-sorted according to “z order” and then rasterized back to front
  - Conversely, intersecting or self-overlapping shapes may require a (post-) sorting strategy, at a fragment level (see 3D)

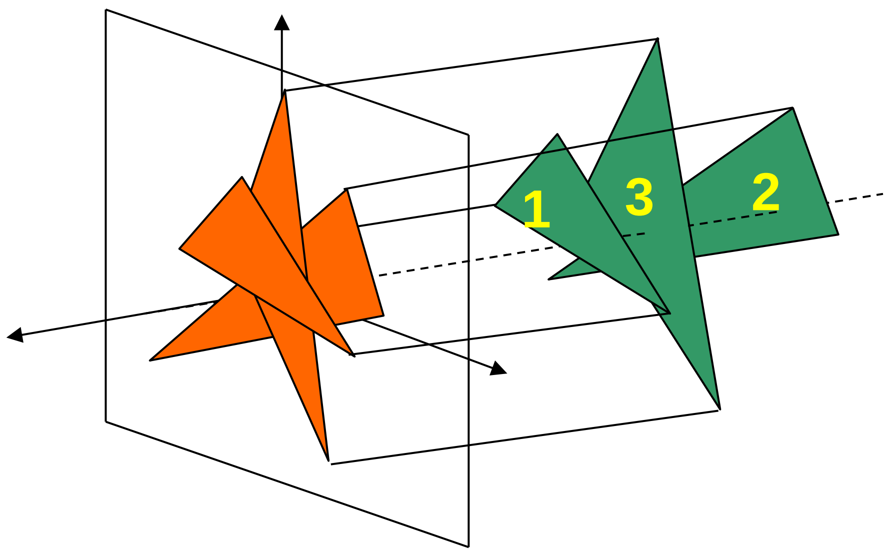
Can be resolved by primitive sorting



Cannot be resolved by primitive sorting – requires sorting at fragment level

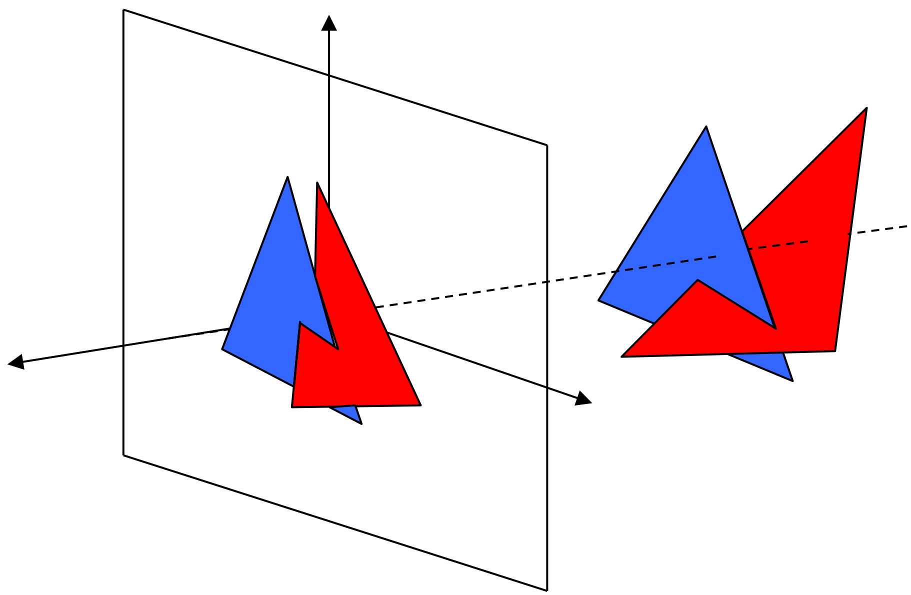
Rasterization and HSE in 3D

- After projecting the primitives in NDC, we must retain only surfaces visible to the camera (HSE) →
  - Surface parts must be sorted according to depth
  - And not according to order of appearance (it is arbitrary)



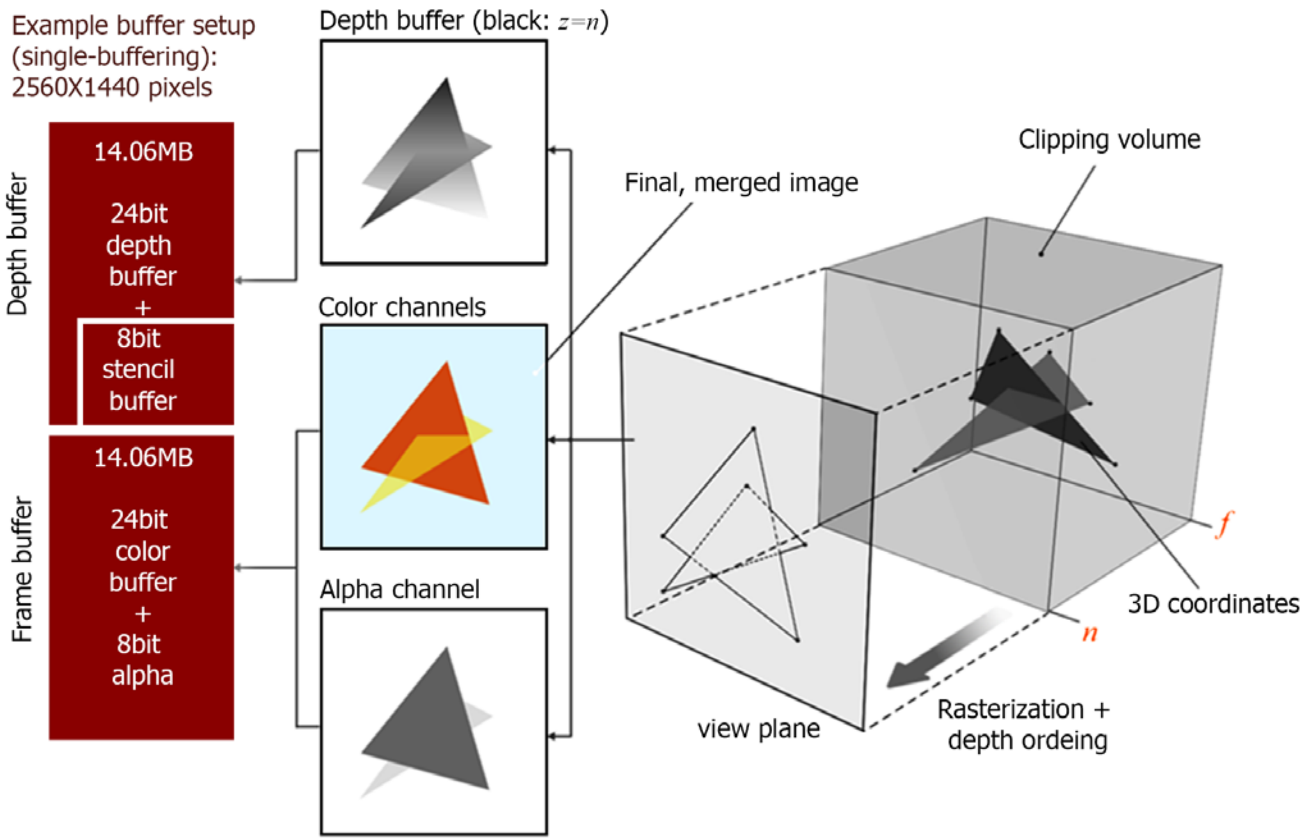
HSE – Per Pixel

- Even if polygons were depth-sorted according to some reference point on them (e.g. centroid), there is no guarantee that they do not overlap →
- Sorting must be performed per pixel



The Depth Buffer

- Separate buffer, same resolution as frame buffer
- Stores the nearest normalized depth values

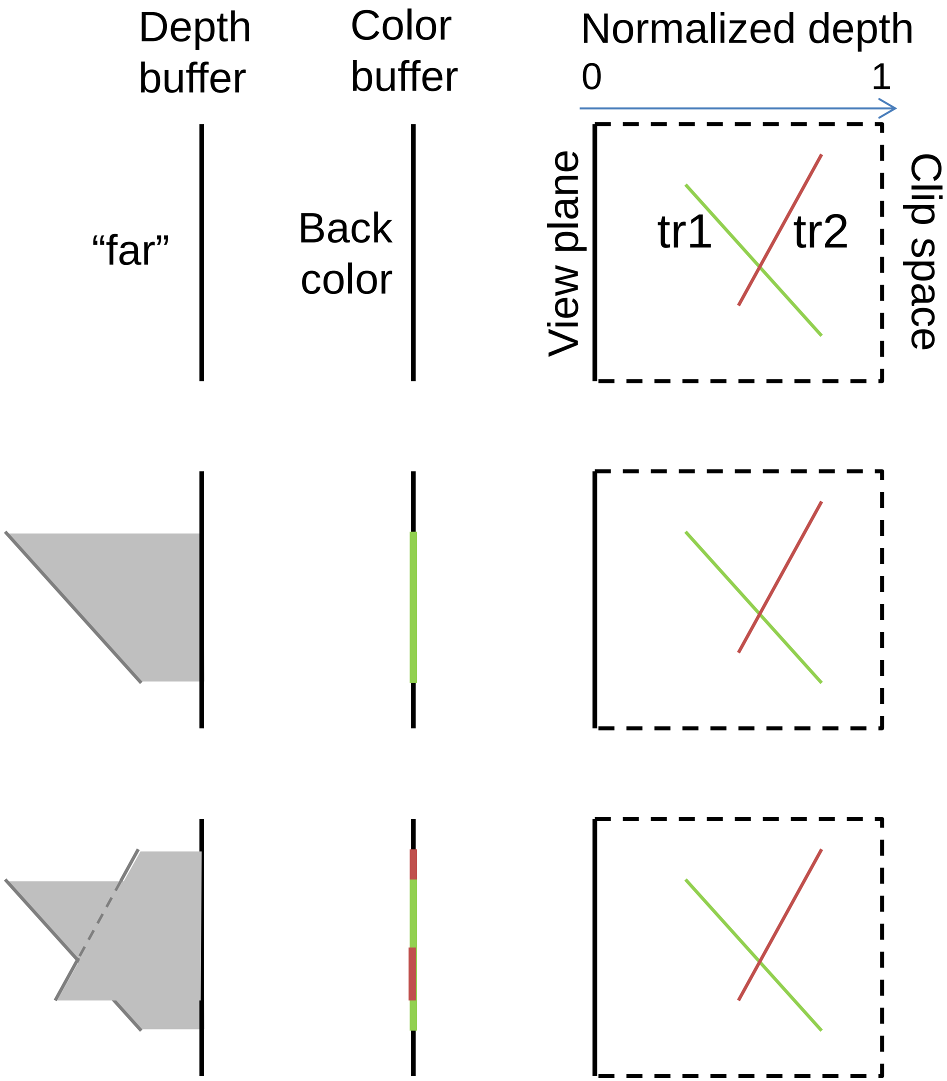


The Z-Buffer Algorithm

- The Z-Buffer algorithm uses the depth buffer to compare each generated fragment at location (i,j) with the previous "visible" (nearest) fragment
- If the new fragment is closest to the view plane:
  - Replace the z in the depth buffer
  - Forward the fragment to the merging stage
- Else (if fragment fails the depth test)
  - Discard the fragment
- Remarks:
  - The depth test may be $<$, $\leq$ or other comparison operand
  - Depth buffer is usually initialized to the "far" value

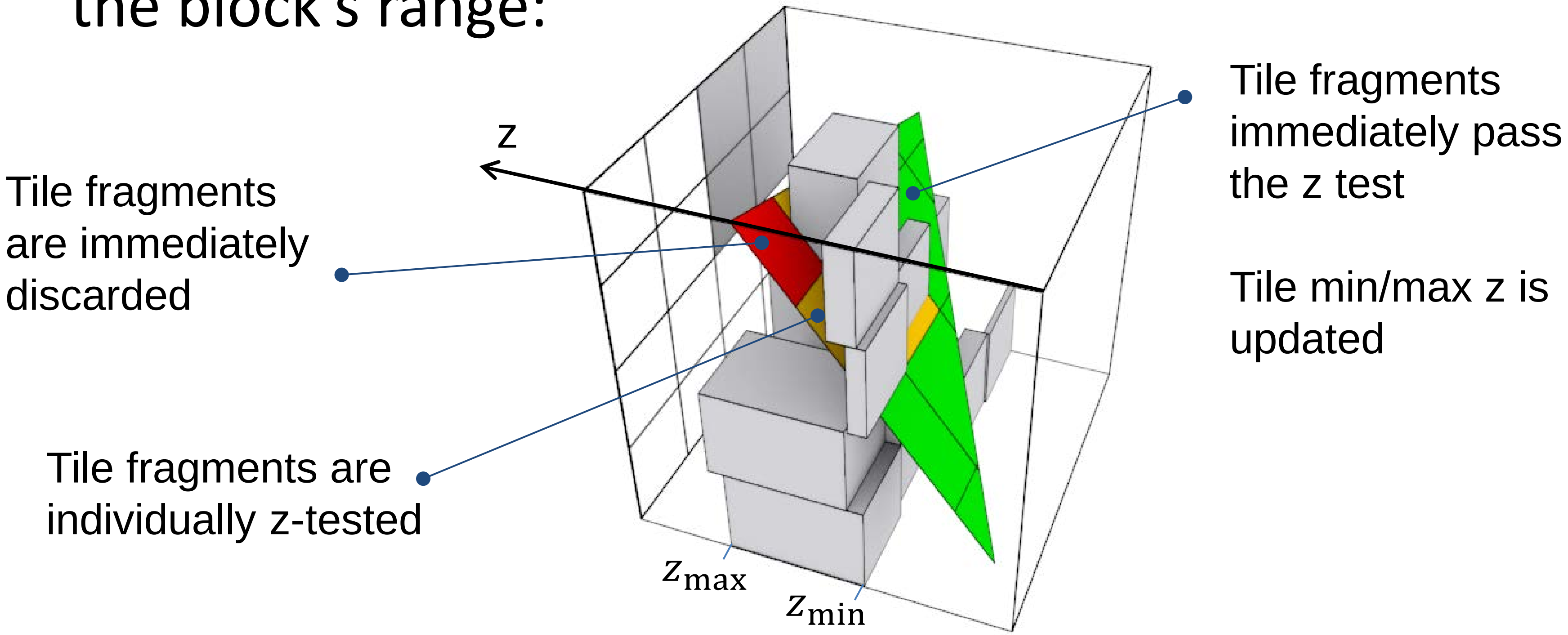
The Z-Buffer: A Simple Example

- Initialize the buffers
- Rasterize the 1st triangle: All z values are in front of the "far" depth
- Rasterize the 2nd triangle: not all z values pass the depth test



Z-Buffer – Optimization: Z Cull

- Split buffer into blocks (can use rasterization tiling)
- For each block maintain: $z_{\min}$, $z_{\max}$
- Compare the min/max z of an incoming triangle to the block's range:

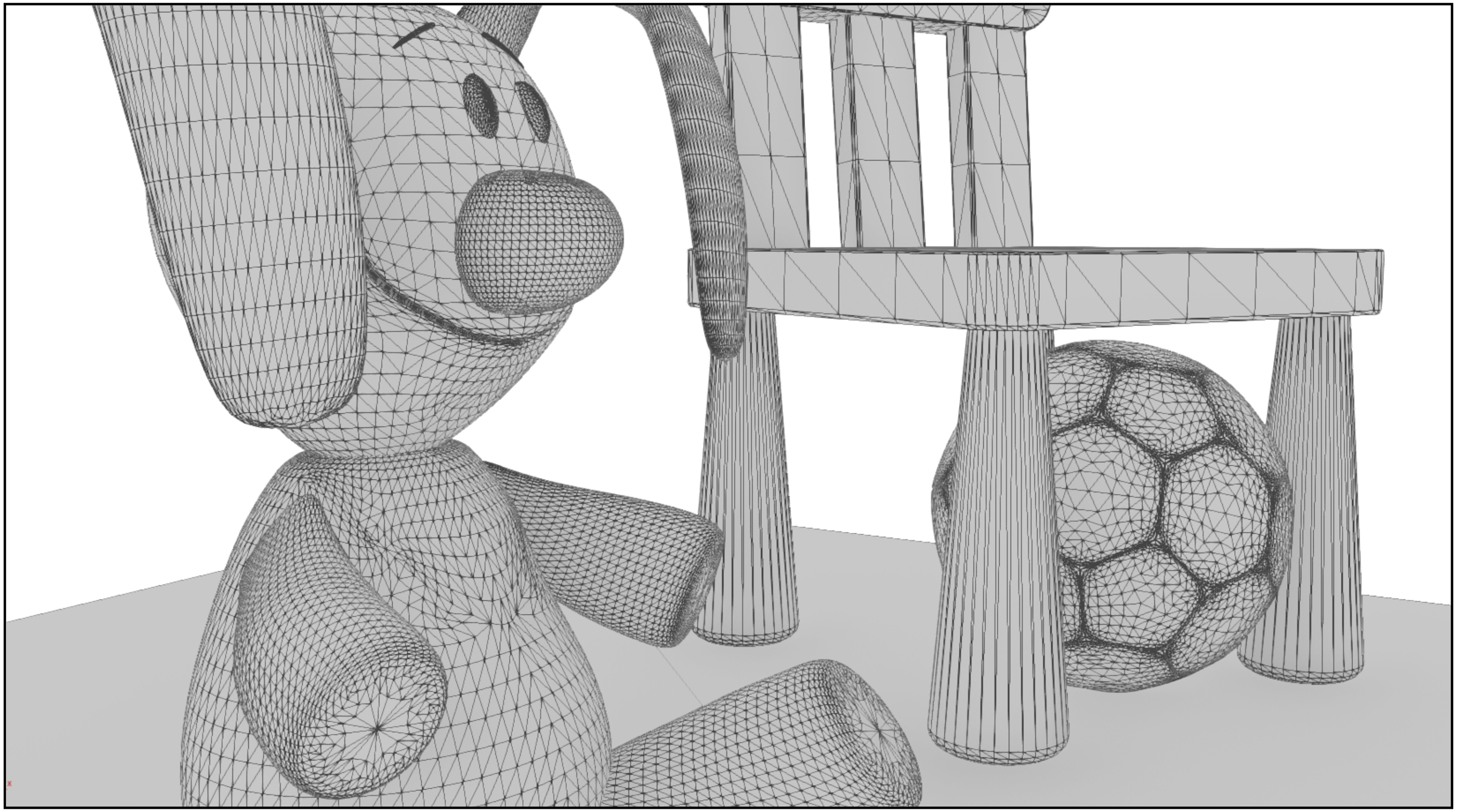


Shading

- In general, the fragment (pixel) shading process defines a color and transparency value for each generated geometry fragment
  - In the simplest case of a flat-colored primitive, e.g. a 2D polygon fill, a predetermined color is assigned to the fragments
  - More elaborate shading algorithms are required for lit and textured 3D surfaces (see texturing and shading chapters)

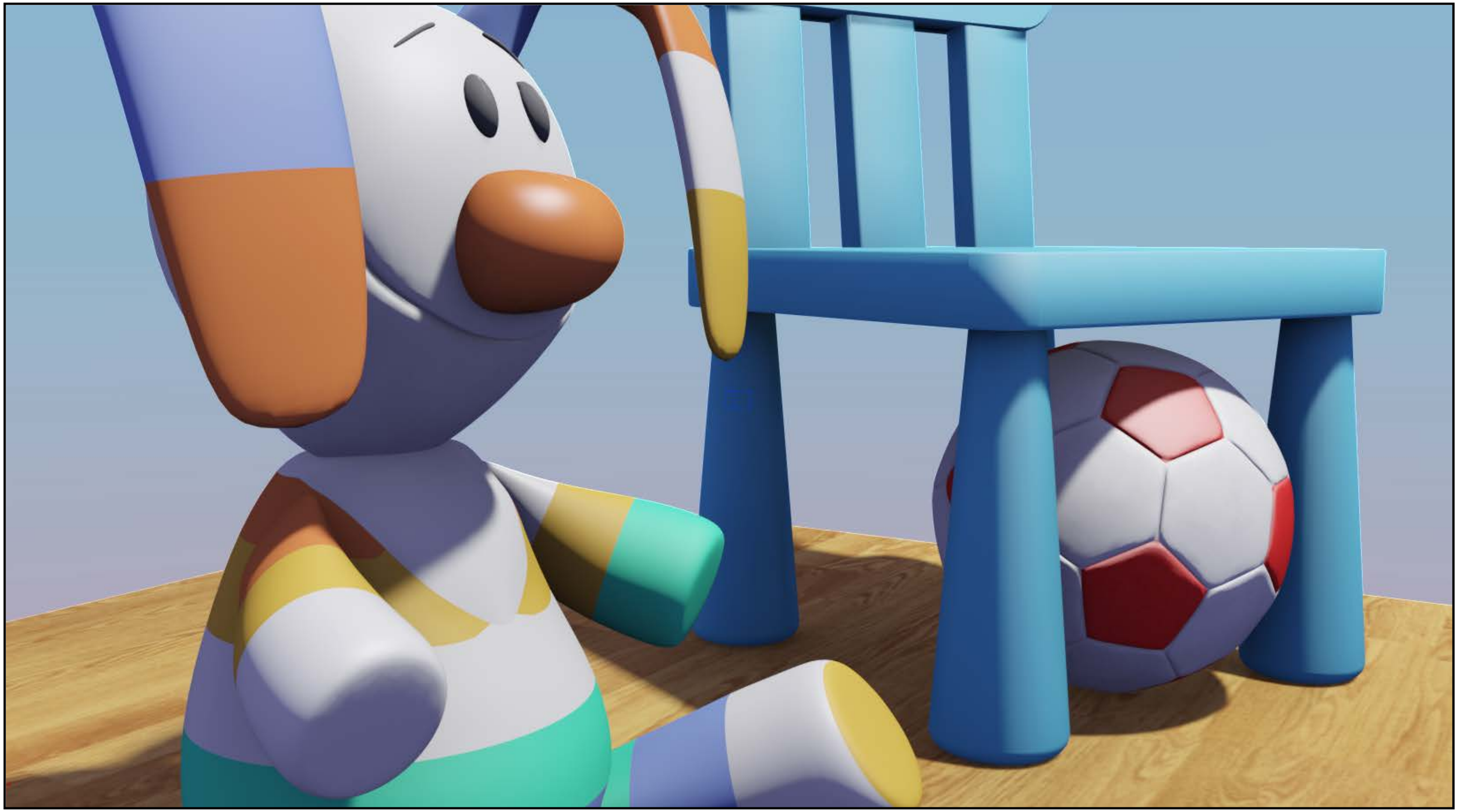
Triangle Rasterization – HSE

- Triangle Fragments with correct order after z-buffer testing



Shaded Fragments

- Triangle fragments after shading and merging



Merging Stage

- Shaded fragments that successfully passed the depth test must contribute to the image in the frame buffer
- In general:
  - Each fragment contributes to the image pixel according to coverage
  - The color is blended with any existing one in the same pixel coordinates. This is especially true for transparent pixels
- All typical rasterization pipelines allow for a number of blending functions to be applied to the incoming fragments

Fragment Merging and Transparency (1)

- When transparency values are generated, these can control the mixing of fragments
- The value controlling this blending is the alpha value, i.e. the “opacity” (or 1-transparency)

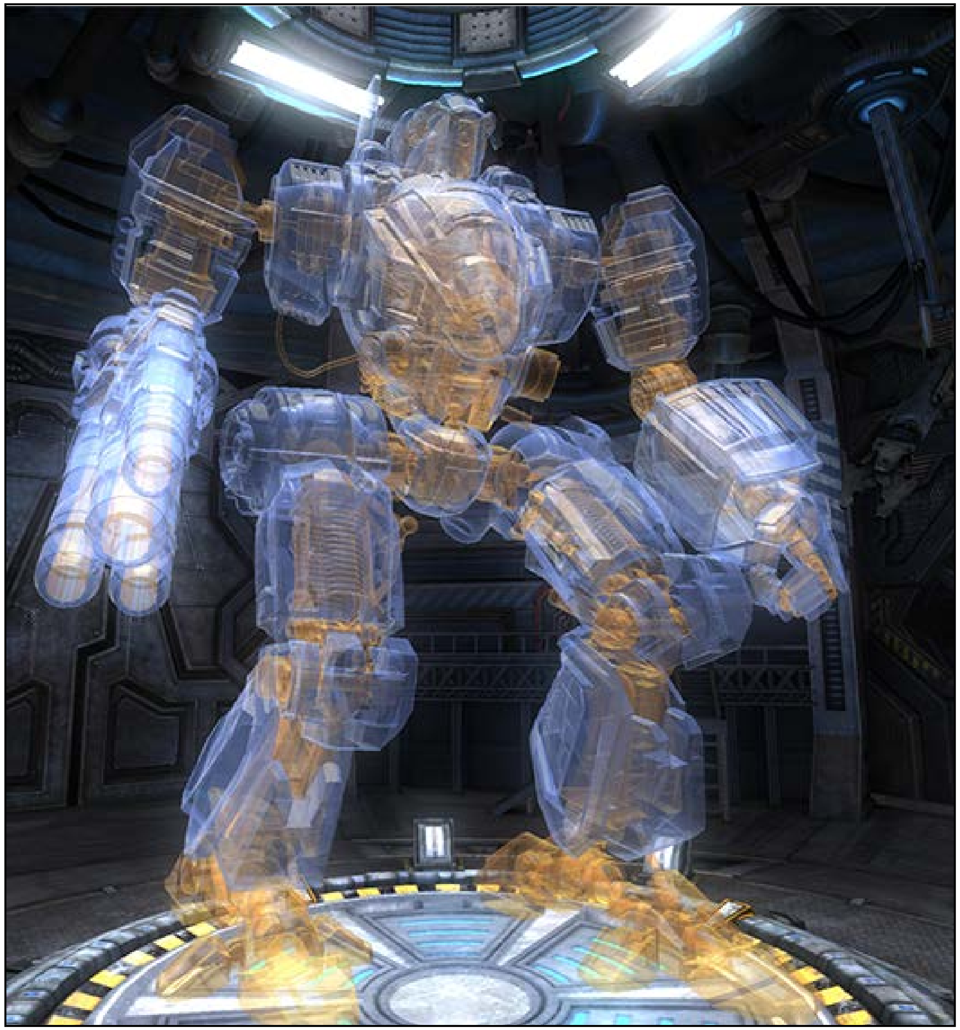
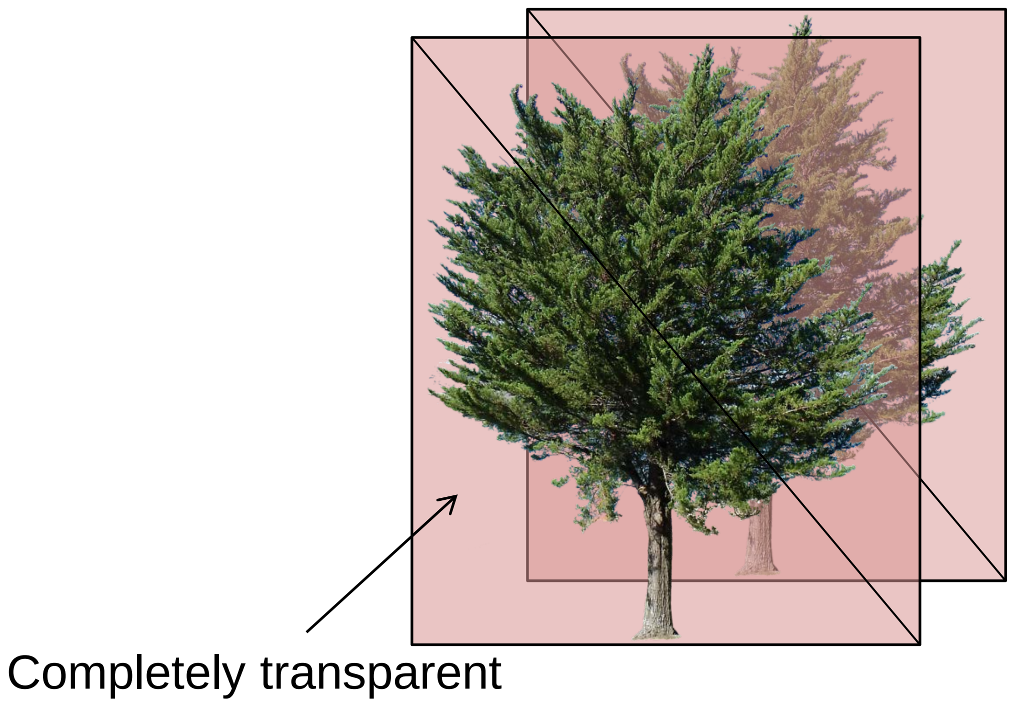


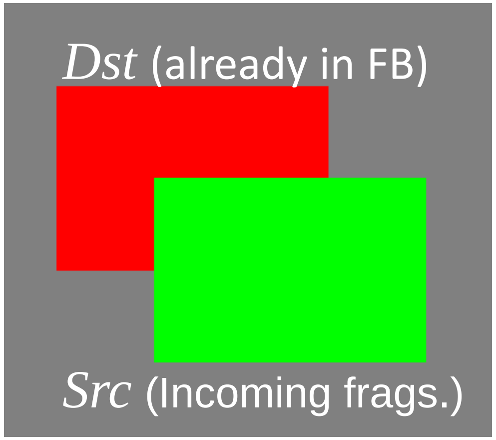
Image source: http://developer.amd.com

Fragment Merging and Transparency (2)

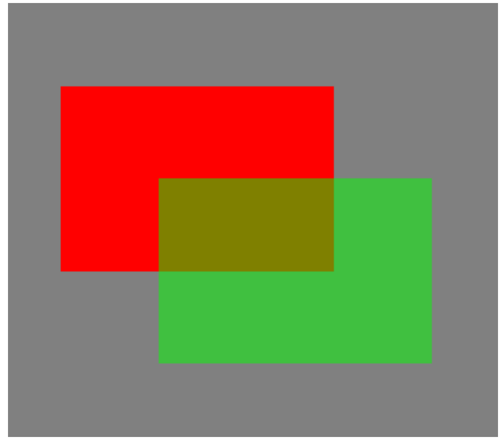
- Extreme values (1,0), can make fragments "pass through" or opaque, to display elaborate "perforated patterns" (see texturing)



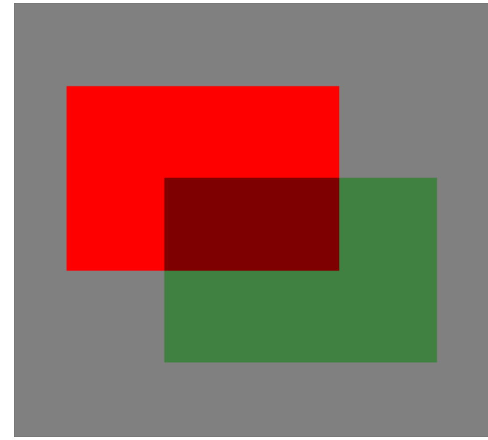
Compositing: Simple Examples



$1 \cdot Src + 0 \cdot Dst$
(replace)

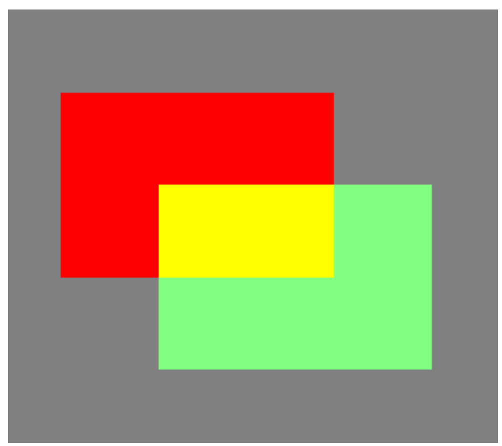


$a \cdot Src + (1 - a) \cdot Dst$
(linear mix)



$a \cdot Dst \cdot Src + (1 - a) \cdot Dst$
(multiply)

$Dst + a \cdot Src$
additive blend

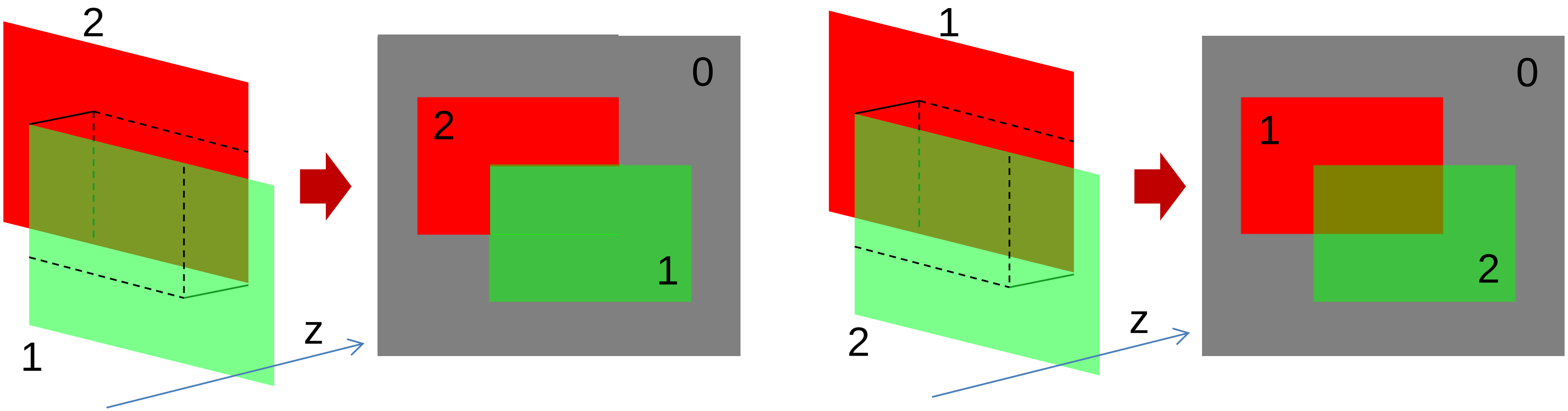


$Dst + Src$
color add

$\max\{0, Dst - Src\}$
color subtract

Z-Buffer and Transparency (1)

- Transparency is not handled well by the Z-Buffer algorithm:
  - Result depends on the order of occurrence of the fragments: Depth test discards fragments behind transparent surfaces if the latter are already rendered



Z-Buffer and Transparency (2)

- Solution 1:
  - Render all opaque geometry first
  - Render transparent geometry next
- Still:
  - Blending of transparent surfaces is still order (and view) dependent

Image source: AMD Mecha Demo

The A-Buffer (1)

- Is a generic antialiased fragment resolve technique, with full support for order-independent transparency
- Instead of a single (nearest) depth value, it maintains a sorted list of all fragments intersecting the pixel
- Stores per fragment transparency and coverage
- Merging:
  - Fragments are resolved front to back according to coverage (via a binary coverage mask) and their transparency

The A-Buffer (2)

- Expensive technique:
  - Must maintain a dynamic list per pixel (fragment bin)
  - Must contain additional data per fragment
  - Must sort contents in each fragment bin
  - Uses indirection (pointers) to access next datum
- H/W implementations?
  - Various optimized variants (or cut-down versions) implemented as shaders
  - Most popular variation: the k-Buffer
    - Fixed-size fragment buckets (arrays)
    - Sorting is still required

Contributors

- Georgios Papaioannou

- Sources:
  - [RTR] T. Akenine-Möller, E. Haines, N. Hoffman, Read-time Rendering (3rd Ed.), AK Peters, 2008
  - [G&V] T. Theoharis, G. Papaioannou, N. Platis, N. M. Patrikalakis, Graphics & Visualization: Principles and Algorithms, CRC Press
  - [OBR] http://fgiesen.wordpress.com/2013/02/10/optimizing-the-basic-rasterizer/