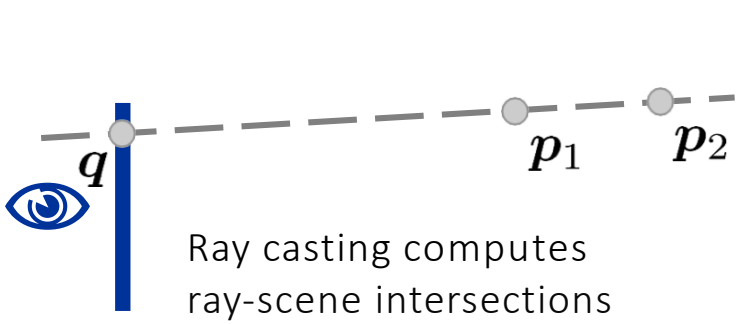
Lecture 19:

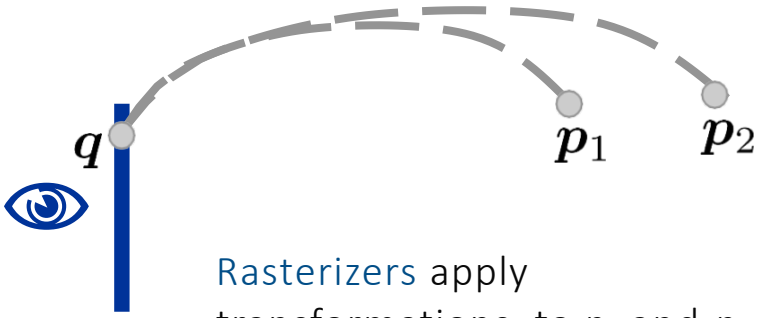
Rasterization

Contents

Visibility can be resolved by ray-tracing or by rasterization



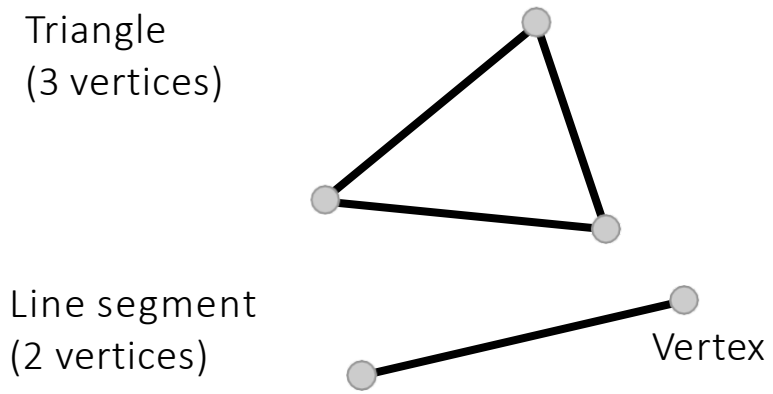
Ray casting computes ray-scene intersections to estimate q from p_1 and p_2.



Rasterizers apply transformations to p_1 and p_2 in order to estimate q.

- If more than one scene point p_i is mapped to the same sensor position q, the scene point closest to the viewer is selected

Computation of pixel positions in an image plane that represent a projected primitive



Primitives represented by vertices

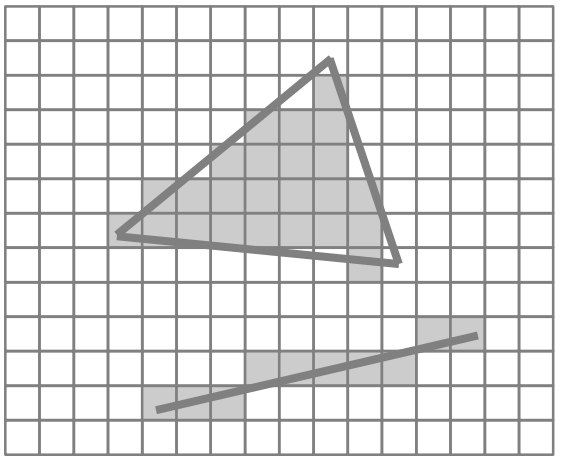
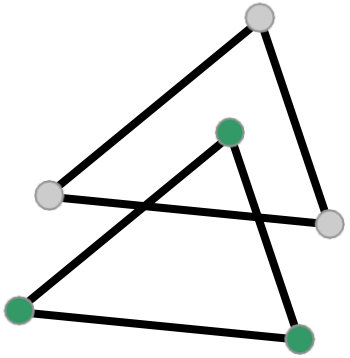
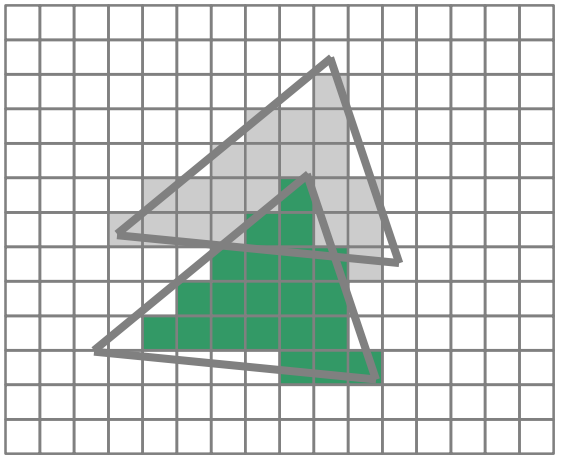


Image plane / 2D array of pixels

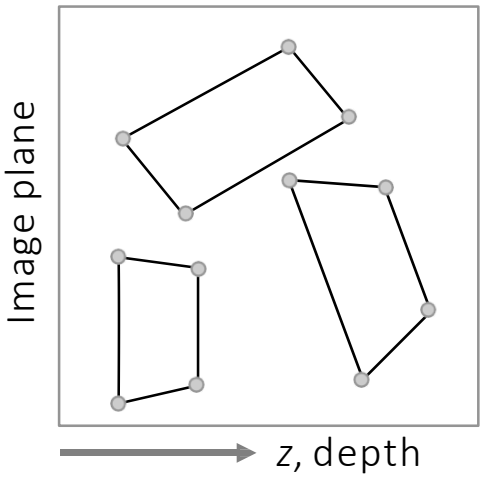
After rasterization, **visibility can be efficiently resolved per pixel position**

- Distances of primitives to the viewer, *i.e.* depth values, can be compared per pixel position

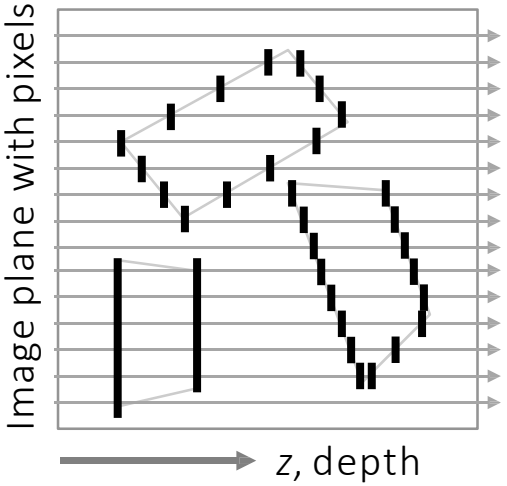




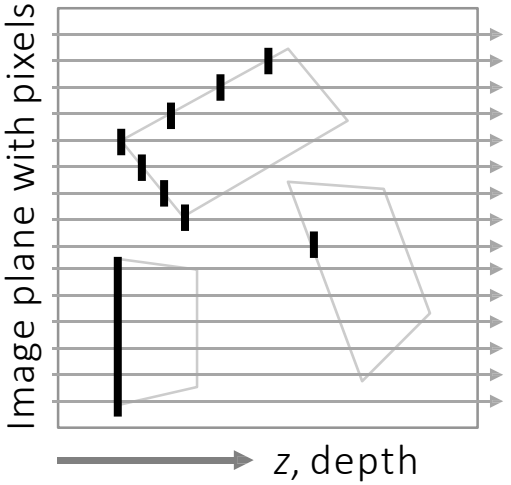
Rasterization is typically implemented for canonical view volumes



Side view of a scene in
a canonical view volume

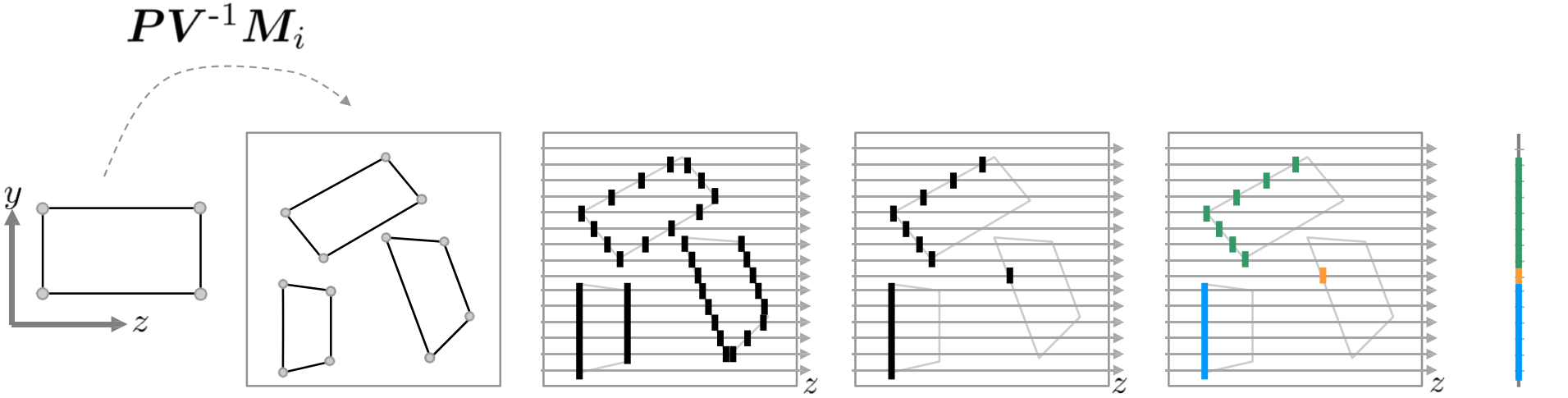


Rasterization result



Resolved visibility

Rasterization is typically implemented for canonical view volumes



Object in local space

Scene in the canonical view volume

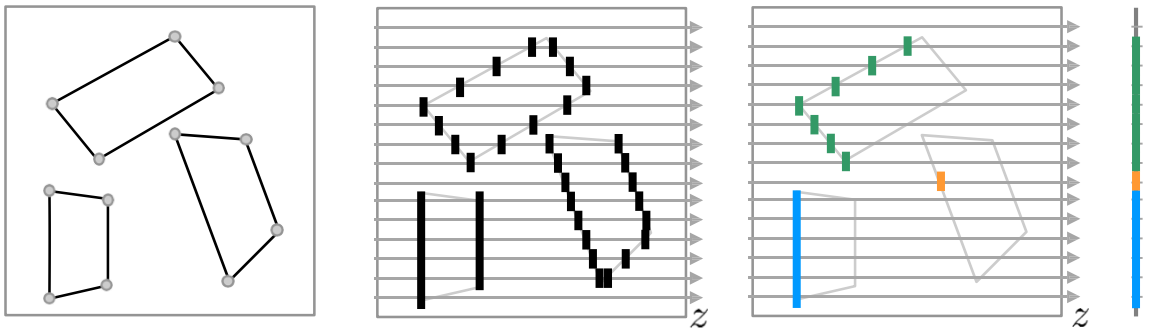
Rasterization

Visibility

Shading

Image

Terms – 2D Illustration



Vertex
Primitive

Object with four vertices and four primitives

Fragment

Pixel

Vertices: have positions and other attributes.

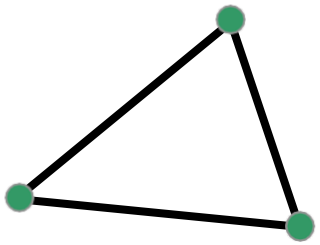
Primitives: are represented by vertices.

Fragments: are pixel candidates with pixel positions and other attributes.

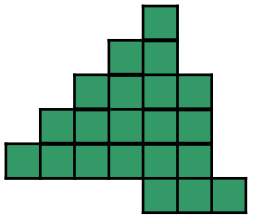
Pixels: have a position and other attributes, in particular color.

Framebuffer: consists of pixels.

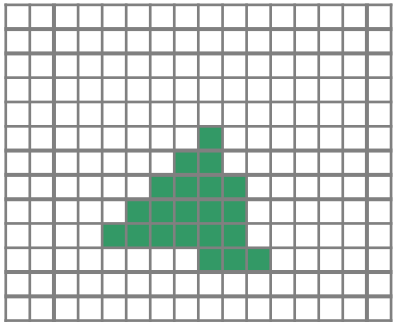
Terms - Illustration



Triangle 1 with three vertices

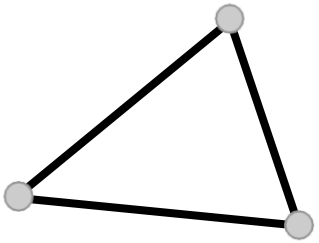


Rasterizer generates fragments.

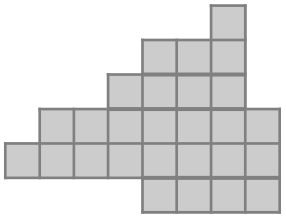


Pixels of the framebuffer

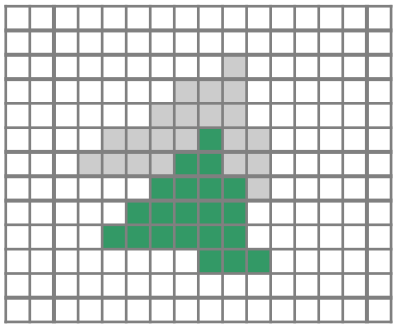
Fragment attributes are used to update pixel attributes in the framebuffer.



Triangle 2 with three vertices



Rasterizer generates fragments.



Pixels of the framebuffer

Framebuffer attributes can be updated. Fragments can be discarded.

Vertex processing

- **Input**: Vertices
- **Output**: Vertices
- Transformations
- Setting, computation, processing of vertex attributes, *e.g.* position, color (Phong), texture coordinates

Rasterization

- **Input**: Vertices and connectivity information
- **Output**: Fragments
- Primitive assembly
- Rasterization of primitives (generates fragments from vertices and connectivity information)
- Sets fragment attributes from vertex attributes, *e.g.* distance to viewer, color, texture coordinates

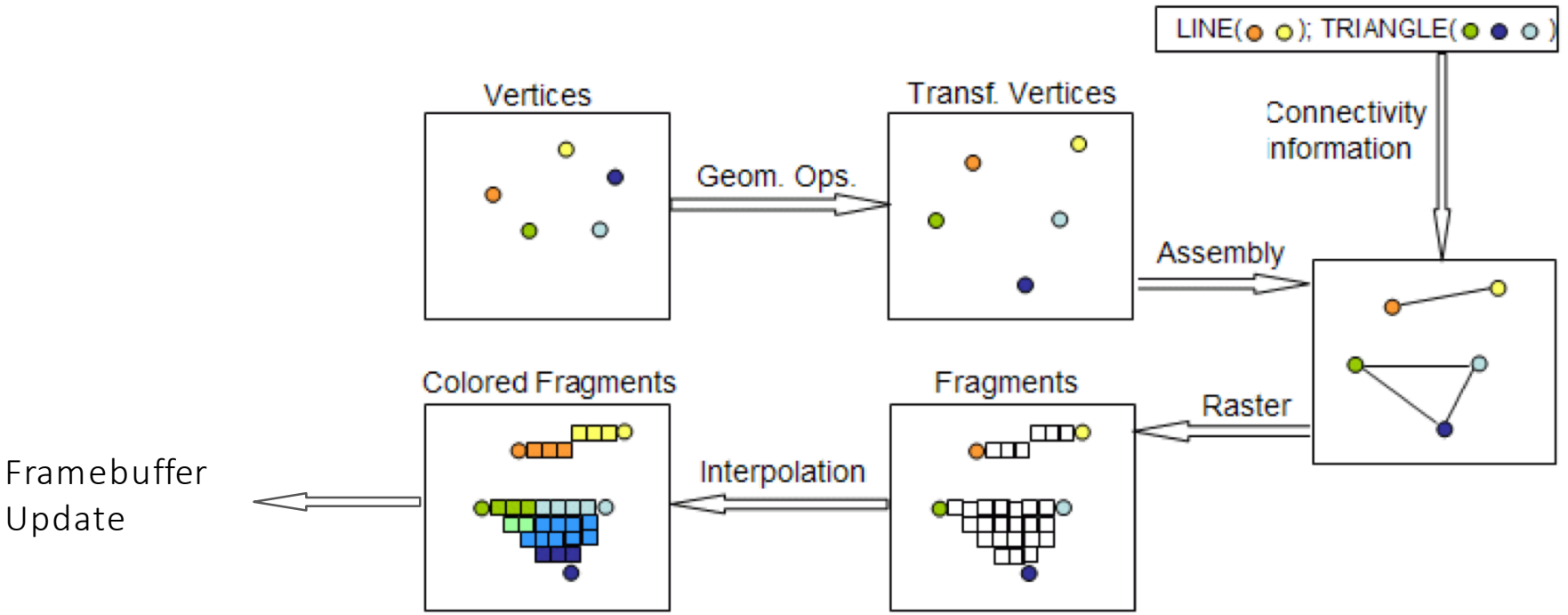
Fragment processing

- **Input**: Fragments
- **Output**: Fragments
- Fragment attributes are processed, *e.g.* color
- Fragments can be discarded

Framebuffer update

- **Input**: Fragments
- **Output**: Framebuffer attributes
- Fragment attributes update framebuffer attributes, *e.g.* color

Main Stages - Overview

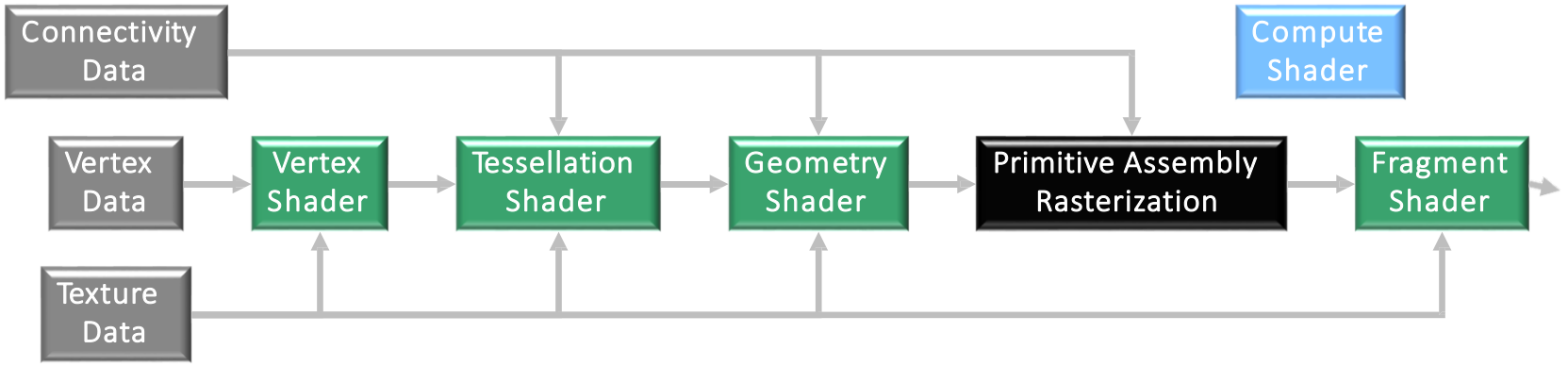


Concept motivated by computational efficiency

- Vertices and fragments are processed independently in the respective stages

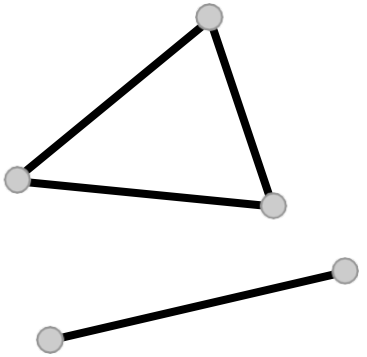
Stages are supported by graphics hardware GPU

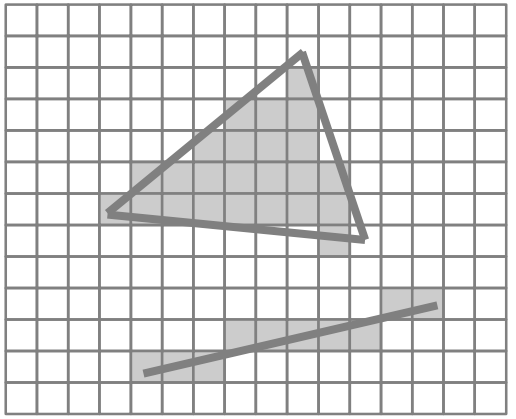
- OpenGL, DirectX, Vulkan are software interfaces to GPUs



Input

- Vertices with connectivity information and attributes:
- **Position**
  - Z-component in NDC space is referred to as depth value. Represents distance to the camera plane.
- **Color**
  - Can optionally be defined or computed with Phong, if surface normal, light and material properties are available
- **Texture coordinates**
  - For lookup and processing of additional data, *i.e.* textures

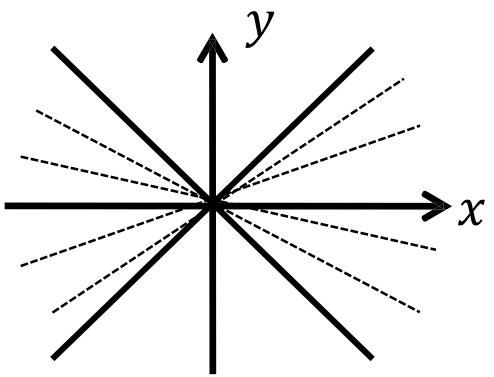




Output

- Fragments with attributes
  - Pixel position
  - Interpolated color, depth, texture coordinates

Assumptions



- Pixels are sample ***points*** on a 2D-integer-grid
  - **OpenGL**: cell bottom-left, integer-coordinate
  - **We**: at the cell center
- Simple raster operations
  - Just setting pixel values or not (binary decision)
    - More complex operations later: compositing / anti-aliasing
- Endpoints at (sub-) pixel coordinates
  - Simple and consistent computations with fixed-point arithmetic
- Limiting to lines with gradient / slope $|m| \leq 1$ (mostly horizontal)
  - Separate handling of horizontal and vertical lines
  - For mostly vertical, swap x and y ($|\frac{1}{m}| \leq 1$), rasterize, swap back
    - Special cases in SW, trivial in HW :-)
- Line width is one pixel
  - $|m| \leq 1$: 1 pixel per column (X-driving axis)
  - $|m| > 1$: 1 pixel per row (Y-driving axis)

Specification

- Initial and end points: $(x_o, y_o), (x_e, y_e)$
- The rational slope $m = {}^{(y_e - y_o)}/_{(x_e - x_o)} = {}^{dy}/_{dx}$
- Functional form: $y = mx + B$, where end points are integer coordinates

Goal

- Find those pixel per column whose distance to the line is smallest

Brute-force algorithm

- Assume that $+x$ is the driving axis → set pixel in every column

```
for (x_i = x_o; x_i < x_e; x_i++) {
    y_i = m * x_i + B
    setPixel(x_i, std::round(y_i))
}
```

Comments

- Variables m and thus y_i need to be calculated in floating-point
- Not well suited for direct HW implementation
  - A floating-point ALU is significantly larger in HW than integer

Digital Differential Analyzer (DDA)

- Origin of incremental solvers for simple differential equations (the Euler method)
- Per time-step: $x' = x + {}^{dx}/_{dt}$, $y' = y + {}^{dy}/_{dt}$

Incremental algorithm

- Choose $dt = dx$, then per pixel

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

$y_{i+1} = m * (x_i + 1) + B = y_i + m$

$setPixel(x_{i+1}, std::round(y_{i+1}))$

Remark

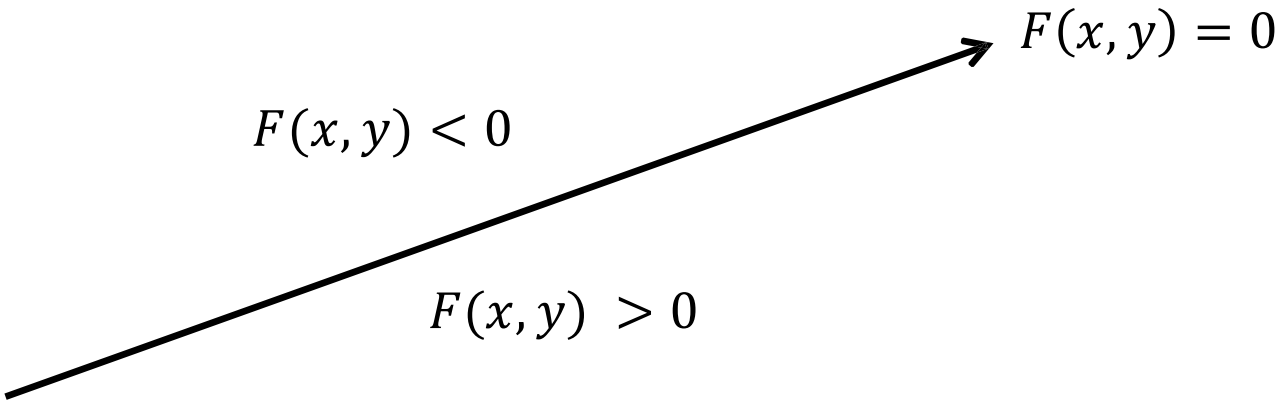
- Utilization of coherence through incremental calculation
  - Avoids the “costly” multiplication
- Accumulates error over length of the line
- Floating point calculations may be moved to fixed point
  - Must control accuracy of fixed point representation

DDA analysis

- Critical point: decision whether rounding up or down

Idea

- Integer-based decision through implicit functions
- Implicit line equation
  - $F(x,y) = ax + by + c = 0$
- Here with $y = mx + B = \frac{dy}{dx}x + B \quad \Rightarrow \quad 0 = xdy - ydx + Bdx$
  - $a = dy, b = -dx, c = Bdx$
- Results in
  - $F(x,y) = xdy - ydx + Bdx = 0$



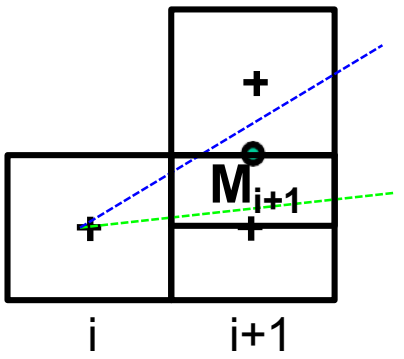
Decision variable d (the midpoint formulation)

- Assume we are at $x = i$, calculating next step at $x = i + 1$
- Measures the vertical distance of midpoint from line:

$$d_{i+1} = F(M_{i+1}) = F\left(x_i + 1, y_i + \frac{1}{2}\right) = a(x_i + 1) + b\left(y_i + \frac{1}{2}\right) + c$$

Preparations for the next pixel

```
if (d_{i+1} <= 0) {                    // Increment in x only
    d_{i+2} = d_{i+1}  + a = d_{i+1} + dy      // Incremental calculation
} else {                               // Increment in x and y
    d_{i+2} = d_{i+1} + a + b = d_{i+1} + dy – dx
    y = y + 1
}
x = x + 1
```



Initialization

- $d_1 = F\left(x_o + 1, y_o + \frac{1}{2}\right) = a(x_o + 1) + b\left(y_o + \frac{1}{2}\right) + c =$
$$= ax_o + by_o + c + a + \frac{b}{2} = F(x_o, y_o) + a + \frac{b}{2} = a + \frac{b}{2}$$
- Because $F(x_o, y_o)$ is zero by definition (line goes through (x_o, y_o))
  - Pixel is always set (but check consistency rules → later)

Elimination of fractions

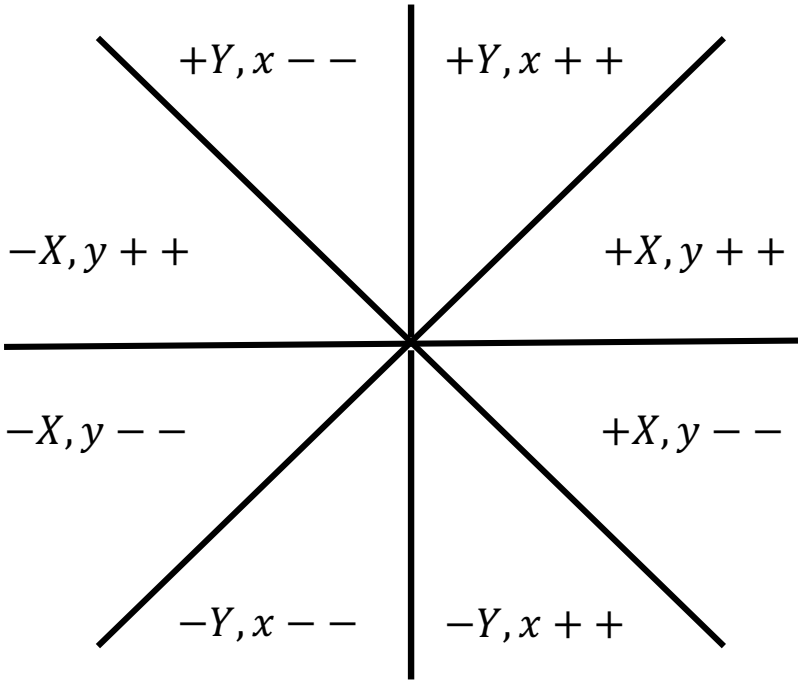
- Any positive scale factor maintains the sign of $F(x, y)$
  - $2F(x_o, y_o) = 2(\mathrm{a}x_o + by_o + c) \rightarrow d_{start} = 2a + b$

Observation:

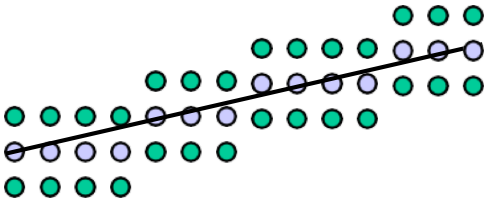
- When the start and end points have integer coordinates then $b = -dx$ and $a = dy$ are also integers
  - Floating point computation can be eliminated
- No accumulated error

8 different cases

- Driving (active) axis: ±X or ±Y
- Increment / decrement of y or x, respectively

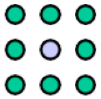


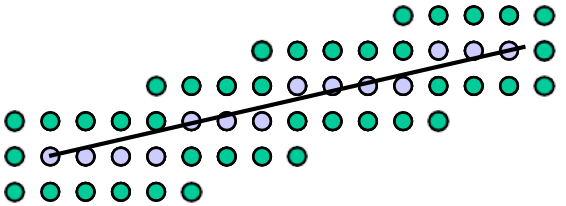
Pixel replication



- Problems with even-numbered widths
- Varying intensity of a line as a function of slope

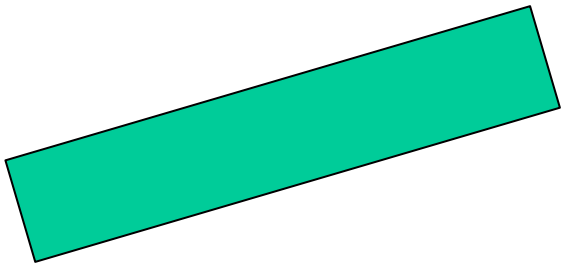
The moving pen





- For some pen footprints the thickness of a line might change as a function of its slope
- Should be as “round” as possible

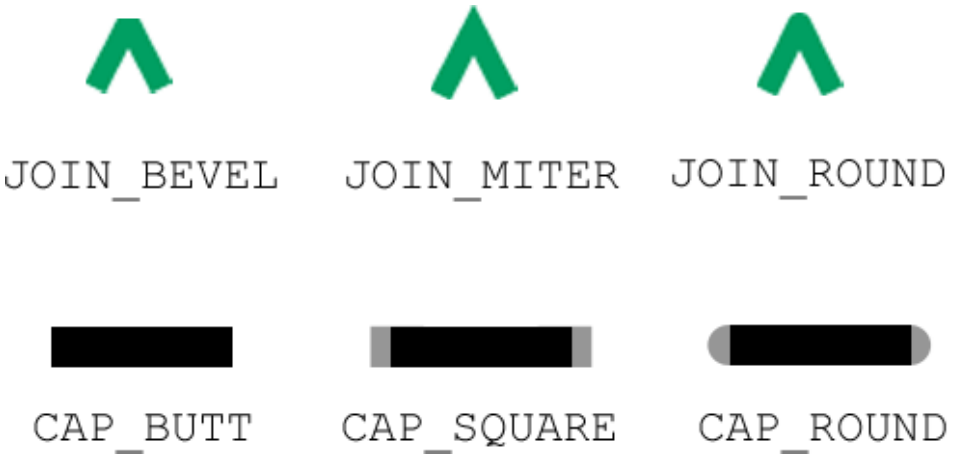
Real Solution: Draw 2D area



- Allows for anti-aliasing and fractional width
- Main approach these days!

End points handling (not available in current OpenGL)

- Joining: handling of joints between lines
  - **Bevel:** connect outer edges by straight line
  - **Miter**: join by extending outer edges to intersection
  - **Round**: join with radius of half the line width

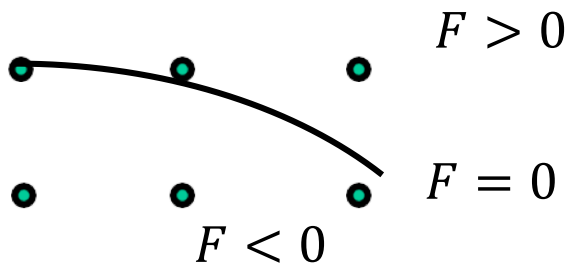


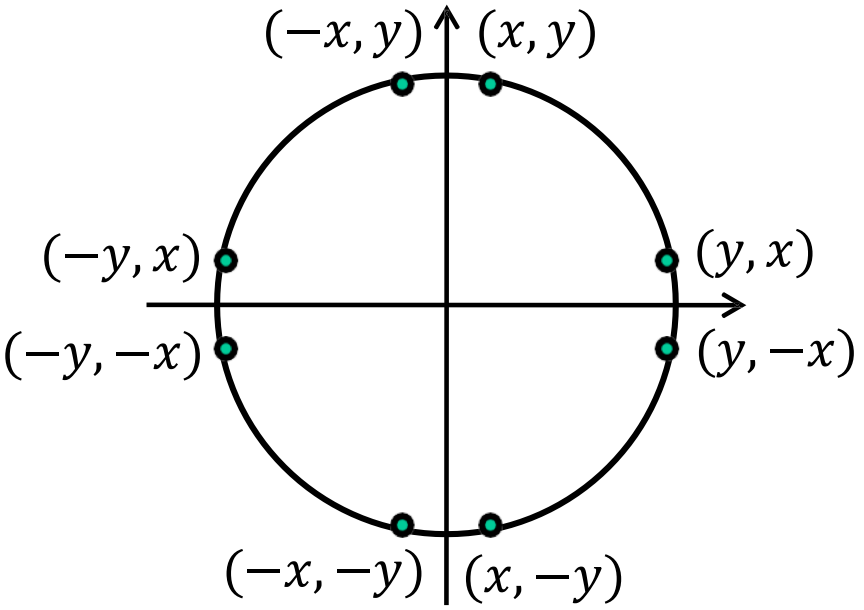
- Capping: handling of end point
  - **Butt**: end line orthogonally at end point
  - **Square**: end line with oriented square
  - **Round**: end line with radius of half the line width

Eight different cases, here $+X, y --$

```
Initialization: x = 0, y = R
F(x,y) = x² + y² - R²
d = F(x+1, y-1/2)
if d < 0
    d = F(x+2, y-1/2)
else if d > 0
    d = F(x+2, y-3/2)
    y = y-1
x = x+1
```

- Works because slope is smaller than 1

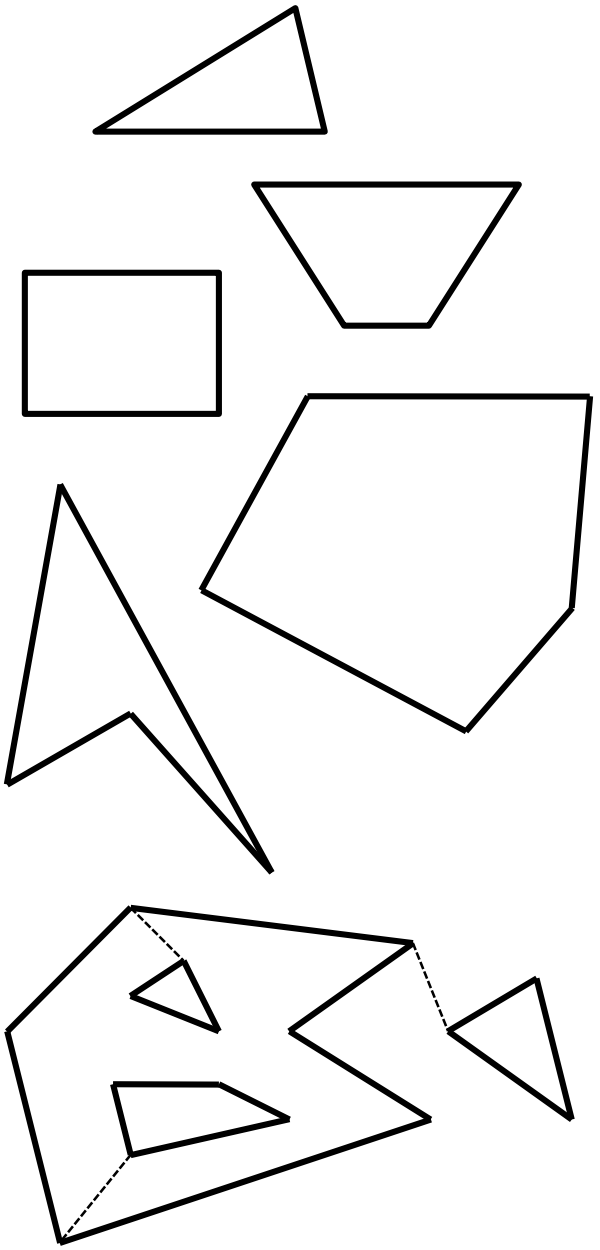




Eight-way symmetry: only one 45° segment is needed to determine all pixels in a full circle

Types

- Triangles
- Trapezoids
- Rectangles
- Convex polygons
- Concave polygons
- Arbitrary polygons
  - Holes
  - Non-coherent

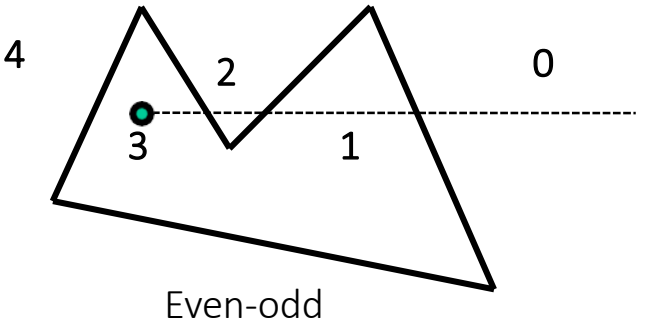


Two approaches

- Polygon tessellation into triangles
  - Only option for OpenGL
  - Needs edge-flags for not drawing internal edges
  - Or separate drawing of the edge
- Direct scan-conversion
  - Mostly in early SW algorithms

What is the interior of a polygon?

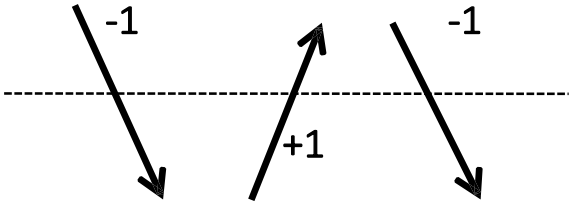
- Jordan Curve Theorem
  - „Any continuous ***simple*** closed curve in the plane, separates the plane into two disjoint regions, the inside and the outside, one of which is bounded.“



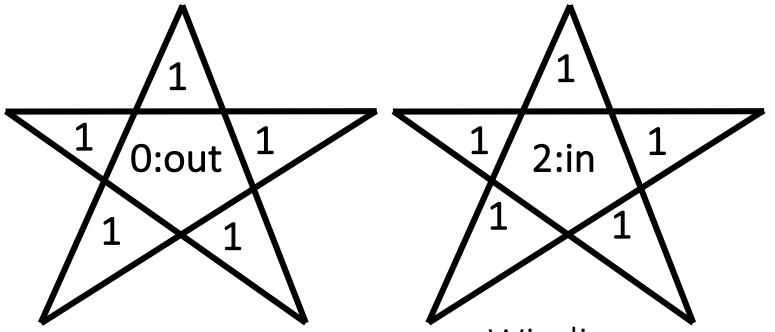
Even-odd

What to do with *non-simple* polygons?

- Even-odd rule (odd parity rule)
  - Counting the number of edge crossings with a ray starting at the queried point **P** till infinity
  - Inside, if the number of crossings is odd
- Non-zero winding number rule
  - Relies on knowing the direction of stroke for each part of the curve
  - Counts # times polygon wraps around **P**
    - Signed intersections with a ray
  - Inside, if the number is not equal to zero
- Differences only in the case of non-simple curves (*e.g.* self-intersection)

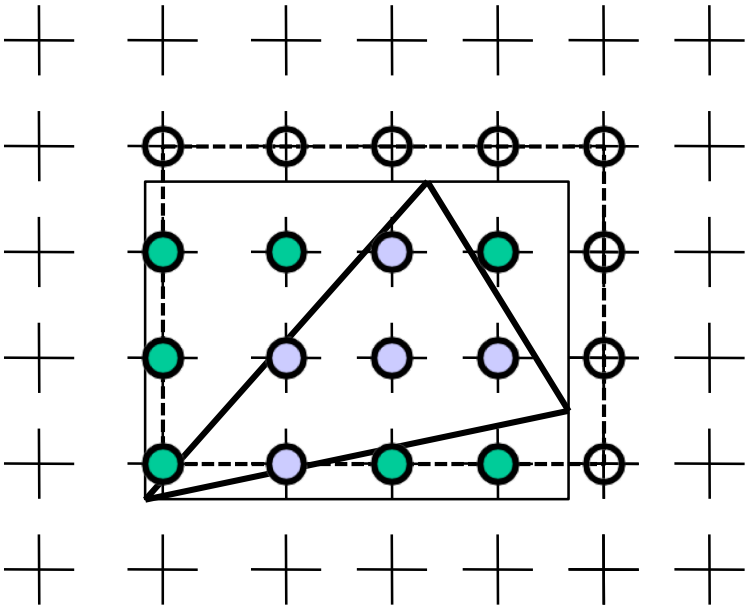


Winding



Even-odd Winding

```
Raster3_box(vertex v[3])
{
    CBoundingBox b;
    bound3(v, &b);
    for (int y = b.ymin; y < b.ymax; y++)
        for (int x = b.xmin; x < b.xmax; x++)
            if (inside(v, x, y)) // upcoming
                fragment(x,y);
}
```



Brute-force algorithm

- Iterate over all pixels within bounding box

Possible approaches for dealing with scissoring

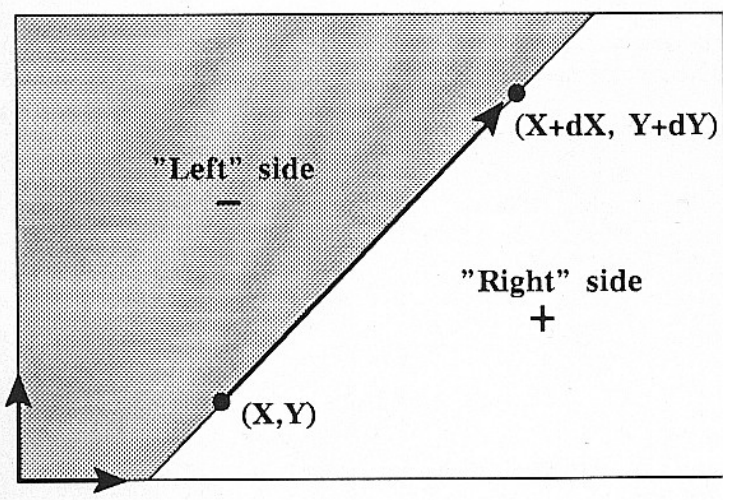
- Scissoring: Only draw on AA-Box of the screen (region of interest)
  - Test triangle for overlap with scissor box, otherwise discard
  - Use intersection of scissor and bounding box, otherwise as above

Approach (Pineda, `88)

- Implicit edge functions for every edge

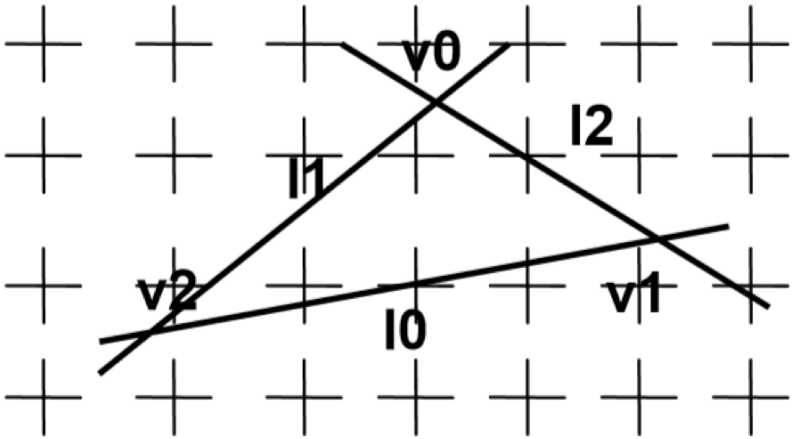
$$F_i(x, y) = ax + by + c$$

- Point is *inside* triangle, if every $F_i(x, y)$ has the same sign
- Perfect for parallel evaluation at many points
  - Particularly with wide SIMD machines (GPUs, SIMD CPU instructions)
- Requires “triangle setup”: Computation of edge function
- Evaluation can also be done in homogeneous coordinates



Hierarchical approach

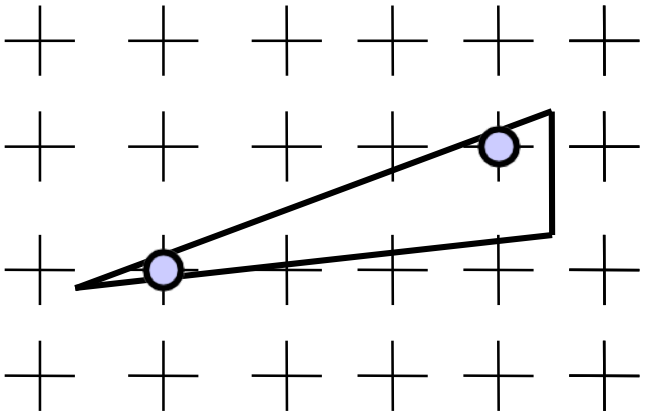
- Can be used to efficiently check large rectangular blocks of pixels
  - Divide screen into tiles / bins (possibly at several levels)
  - Evaluate F at tile corners
  - Recurse only where necessary, possibly until subpixel level



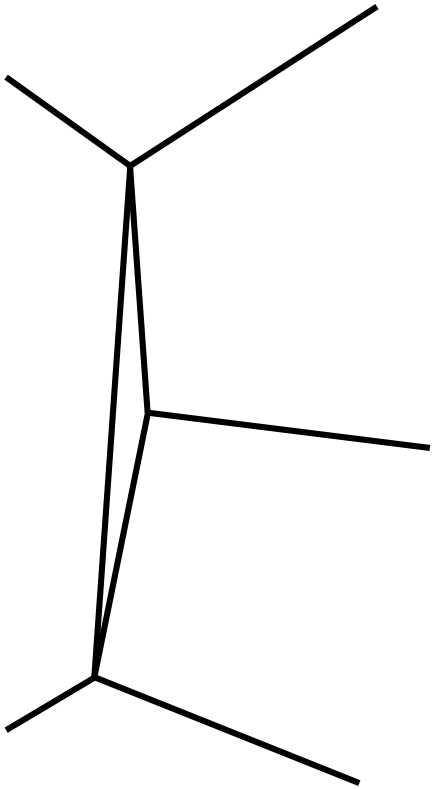

```
Raster3_incr(vertex v[3])
{
    edge l0, l1, l2;
    value d0, d1, d2;
    CBoundingBox b;
    bound3(v, &b);

    mkedge(v[0],v[1],&l2);
    mkedge(v[1],v[2],&l0);
    mkedge(v[2],v[0],&l1);

    d0 = l0.a * b.xmin + l0.b * b.ymin + l0.c;
    d1 = l1.a * b.xmin + l1.b * b.ymin + l1.c;
    d2 = l2.a * b.xmin + l2.b * b.ymin + l2.c;

    for (int y = b.ymin; y < b.ymax, y++) {
        for (int x = b.xmin; x < b.xmax, x++) {
            if (d0 <= 0 && d1 <= 0 && d2 <= 0)
                fragment(x,y);
            d0 += l0.a; d1 += l1.a; d2 += l2.a;
        }
        d0 += l0.a * (b.xmin - b.xmax) + l0.b;
        …
    }
```

Observations

- Pixels set can be non-connected
- May have overlap and gaps at T-edges



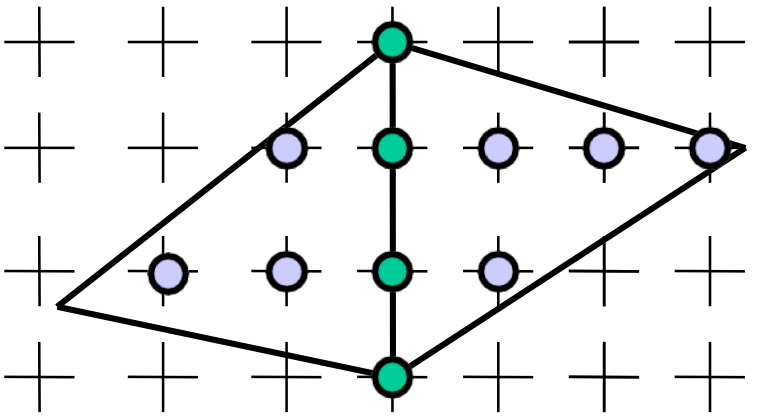
Non-connected pixels: OK



Not OK: Model must be changed

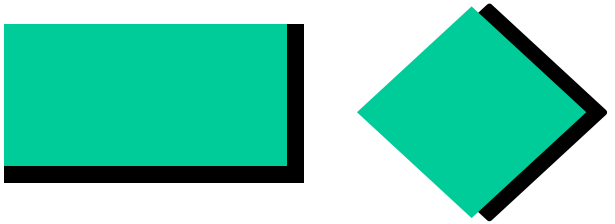
Consistency: edge singularity (shared by 2 triangles)

- What if term $d = ax + by + c = 0$ (pixel centers lies exactly on the line)
- For $d \leq 0$: pixels would get set twice
  - Problem with some algorithms
  - Transparency, XOR, CSG, ...
- Missing pixels for $d < 0$ (set by no triangle)



Solution: "shadow" test

- Pixels are not drawn on the right and bottom edges
- Pixels are drawn on the left and upper edges
  - Evaluated via derivatives a and b
- Test for all edges also solves problem at vertices



```
bool inside(value d, value a, value b)
{       // ax + by + c = 0
    return (d < 0) || (d == 0 && !shadow(a, b));
}
bool shadow(value a, value b)
{
    return (a > 0) || (a == 0 && b > 0);
}
```

In-Triangle test (for common origin)

- Rasterization:
  - Project to 2D, clip
  - Set up 2D edge functions, evaluate for each sample (using 2D point)
- Ray tracing:
  - Set up 3D edge functions, evaluate for each sample (using direction)
- The ray tracing test can also be used for rasterization in 3D
  - Avoids projection & clipping

Enumerating scene primitives

- Rasterization (simple):
  - Linearly test them all in random order
- Rasterization (advanced):
  - Build (coarse) spatial index (typically on application side)
  - Traverse with (large) view frustum
    - Every one separately when using tiled rendering
- Ray Tracing:
  - Build (detailed) spatial index
  - Traverse with (infinitely thin) ray or with some (small) frustum
- Both approaches can benefit greatly from spatial index

Binning

- Test to (hierarchically) find pixels likely to be covered by a primitive
- Rasterization:
  - Great speedup due to very large view frustum (many pixels)
- Ray tracing (frustum tracing)
  - Can speed up, depending on frustum size [Benthin'09]
- Ray Tracing (single / few rays)
  - Not needed

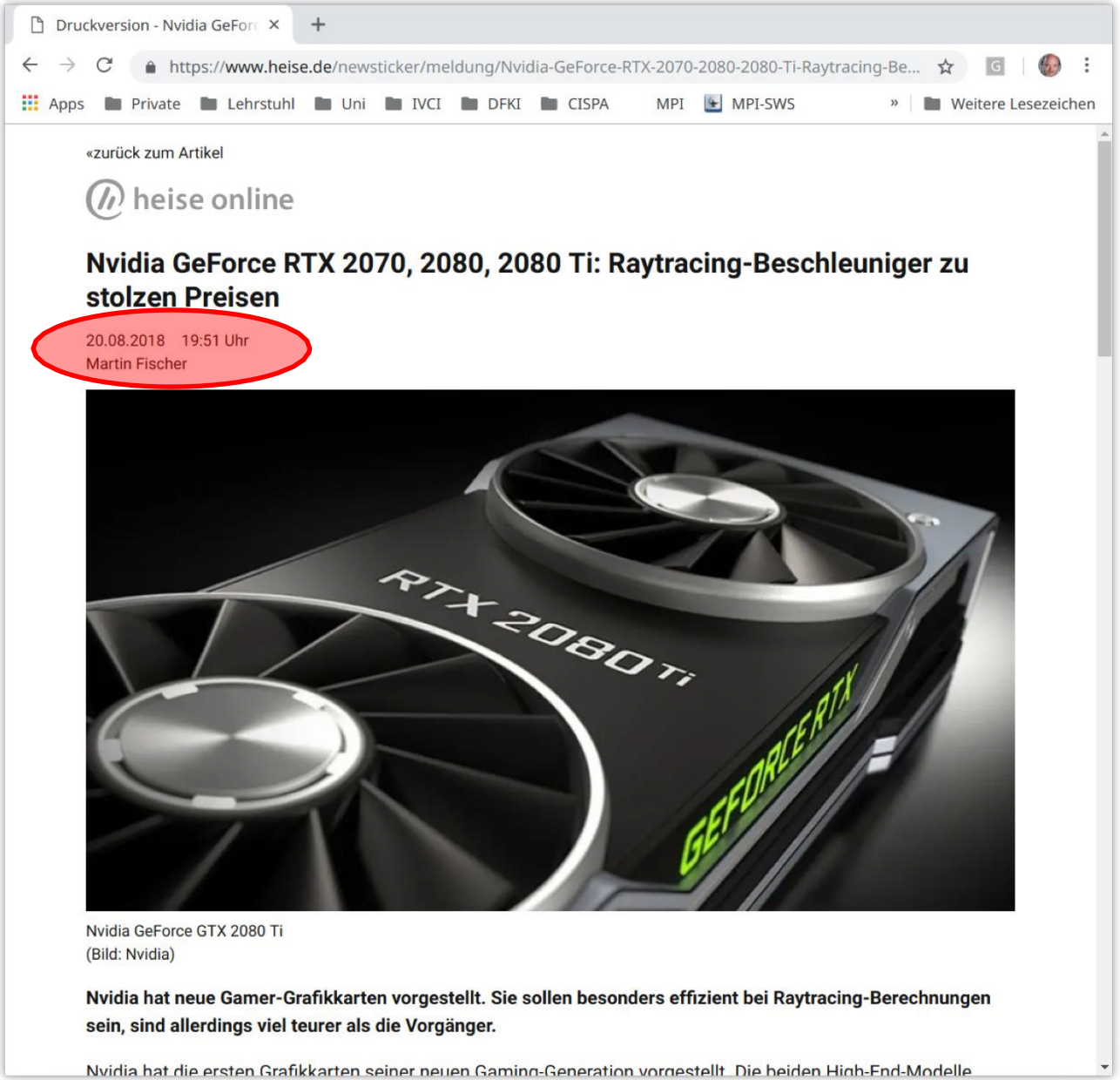
Conclusion

- Both algorithms can use the same in-triangle test
  - In 3D, requires floating point, but boils down to 2D computation
- Both algorithms can benefit from spatial index
  - Benefit depends on relative cost of in-triangle test (HW vs. SW)
- Both algorithms can benefit from 2D binning to find relevant samples
  - Benefit depends on ratio of covered/uncovered samples per frustum

Both approaches are essentially the same

- Different organization (size of frustum, binning)
- There is no reason RT needs to be slower for primary rays (exc. FP)

HW-Supported Ray Tracing



ACMSIGGRAPH | Blog

Ray Tracing Will be Everywhere in 2020

8
OCT 2019

by Jon Peddie | posted in: Augmented Reality, Gaming, Graphics, Hardware, Production, Real-Time | 1

Not just PCs, but consoles, and mobile devices will show the rays

Photo by Davide Cantelli on Unsplash

Ray tracing, the long-held goal for computer graphics, has been in the movies, CAD design, and visualization for decades. The physically accurate, beautiful images have historically made realistic scenes and impressions; however, those images came at a price,

Search

About

Since 1974, ACM SIGGRAPH has been fostering and celebrating innovation in Computer Graphics and Interactive Techniques, building communities that invent, educate, inspire, and redefine the computer graphics landscape. For more news and headlines, visit the ACM SIGGRAPH news feed.

Categories

- ACM SIGGRAPH
- Adaptive Technology
- Animation
- Art
- Augmented Reality
- Awards
- Business