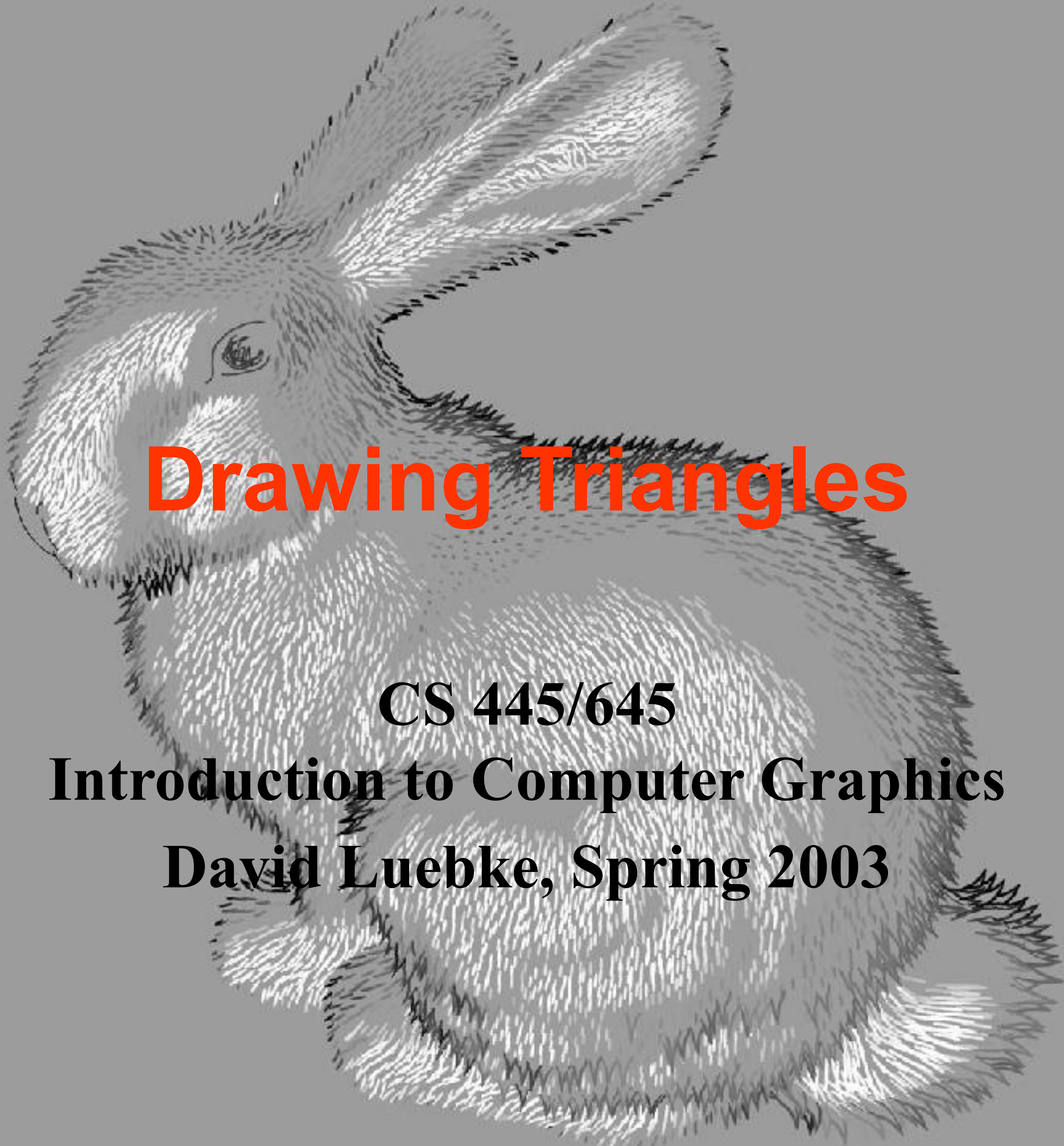
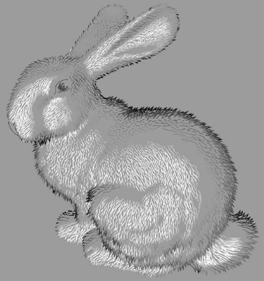


Drawing Triangles

CS 445/645
Introduction to Computer Graphics
David Luebke, Spring 2003

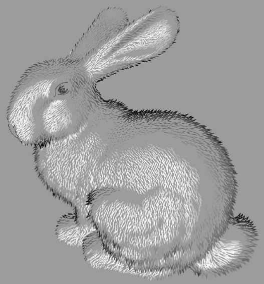
Admin

- Homework 1 graded, will post this afternoon



Rasterizing Polygons

- In interactive graphics, polygons rule the world
- Two main reasons:
  - Lowest common denominator for surfaces
    - Can represent any surface *with arbitrary accuracy*
    - Splines, mathematical functions, volumetric isosurfaces…
  - Mathematical simplicity lends itself to simple, regular rendering algorithms
    - Like those we're about to discuss…
    - Such algorithms embed well in hardware

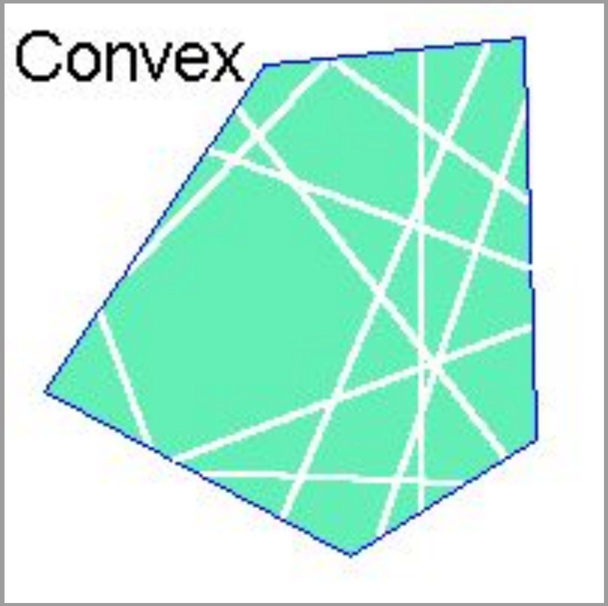


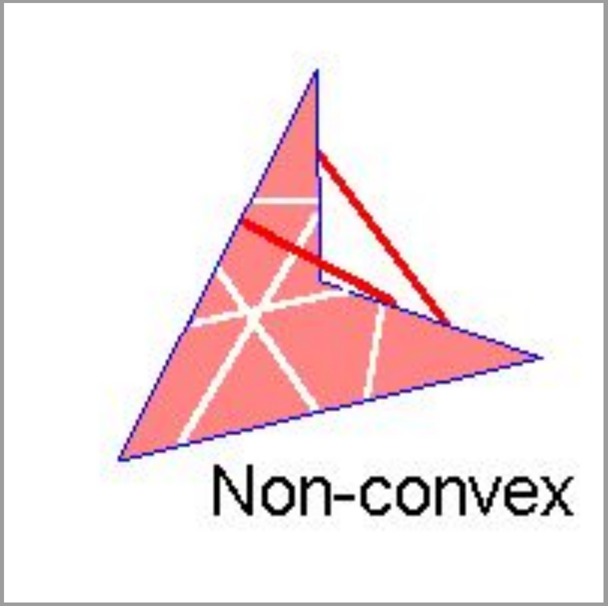
Rasterizing Polygons

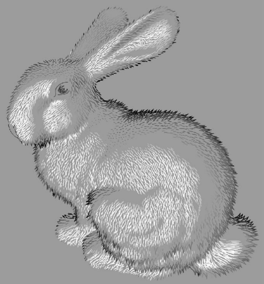
- Triangle is the *minimal unit* of a polygon
  - All polygons can be broken up into triangles
    - Convex, concave, complex
  - Triangles are guaranteed to be:
    - Planar
    - Convex
  - *What exactly does it mean to be convex?*

Convex Shapes

- A two-dimensional shape is *convex* if and only if every line segment connecting two points on the boundary is entirely contained.





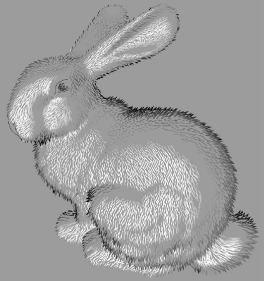


Convex Shapes

- *Why do we want convex shapes for rasterization?*
- One good answer: because any scan line is guaranteed to contain at most one segment or *span* of a triangle
  - Another answer coming up later
  - Note: Can also use an algorithm which handles concave polygons. It is more complex than what we'll present here!

Decomposing Polys Into Tris

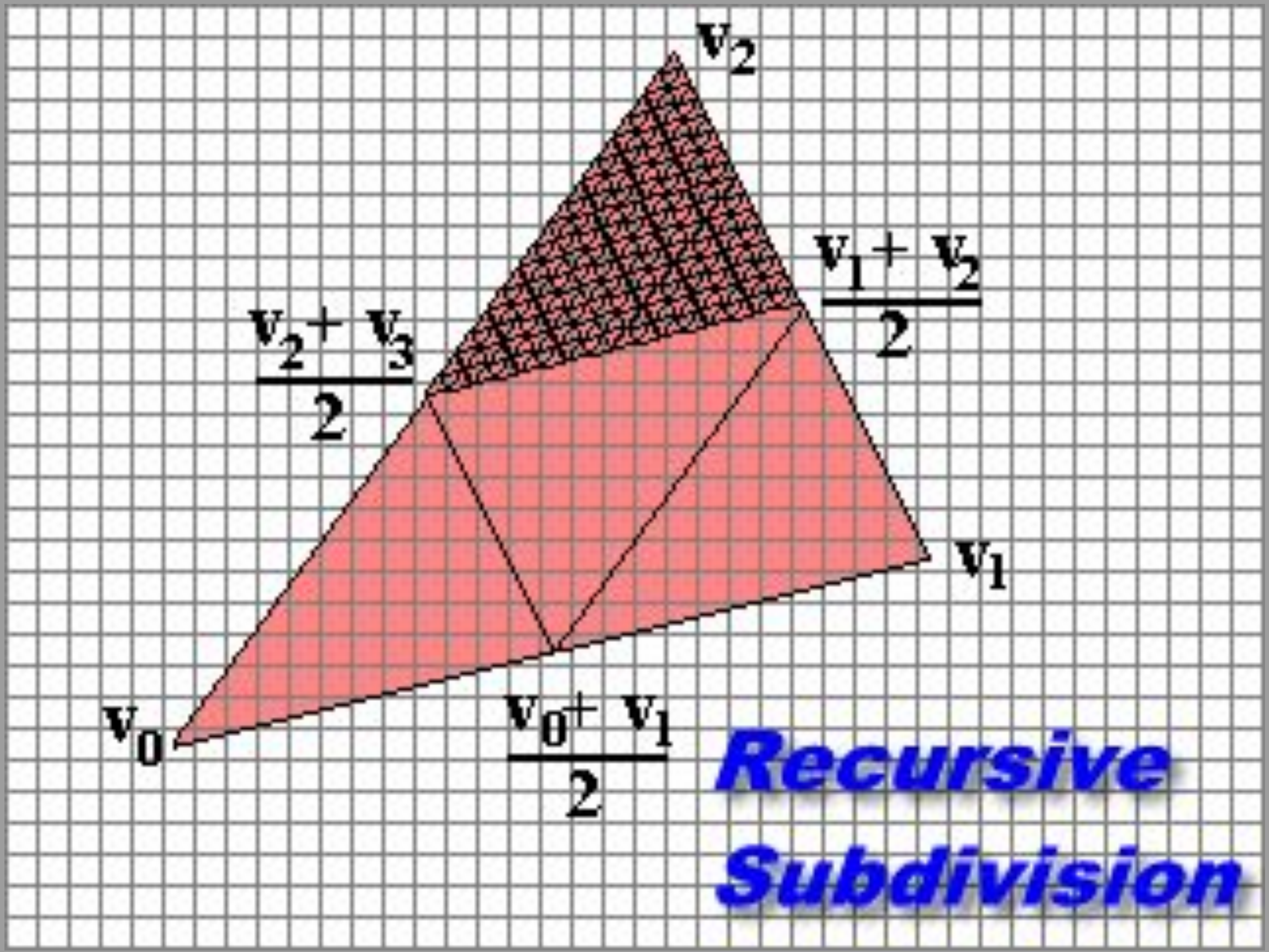
- Any convex polygon can be trivially decomposed into triangles
  - Draw it
- Any concave or complex polygon can be decomposed into triangles, too
  - Non-trivial!



Rasterizing Triangles

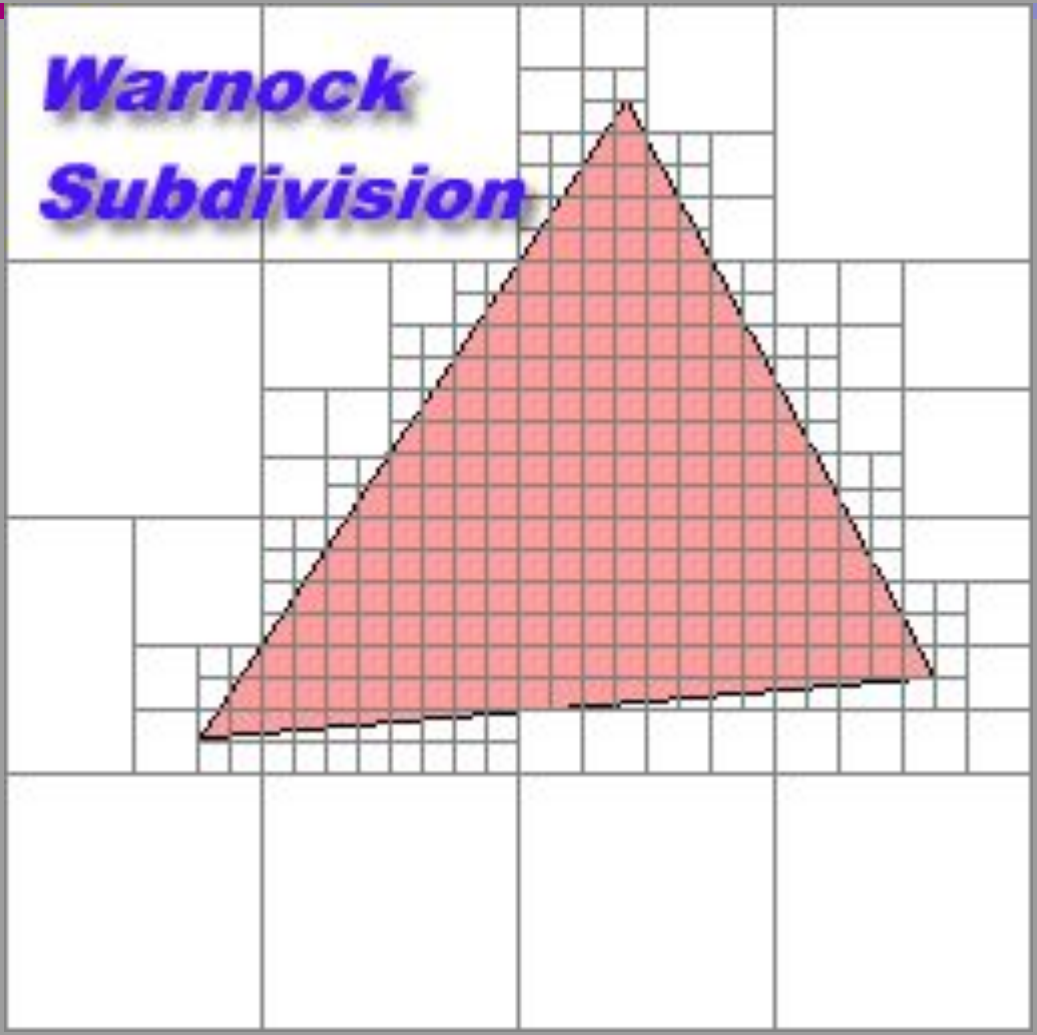
- Interactive graphics hardware commonly uses *edge walking* or *edge equation* techniques for rasterizing triangles
- Two techniques we won't talk about much:
  - Recursive subdivision of primitive into micropolygons (REYES, Renderman)
  - Recursive subdivision of screen (Warnock)

Recursive Triangle Subdivision



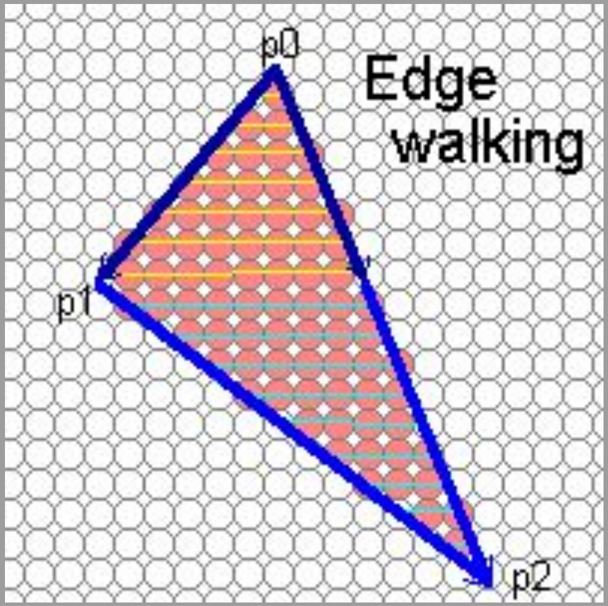
David Luebke

Recursive Screen Subdivision



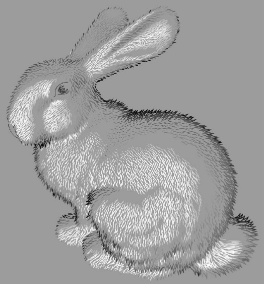
Edge Walking

- Basic idea:
  - Draw edges vertically
  - Fill in horizontal spans for each scanline
  - Interpolate colors down edges
  - At each scanline, interpolate edge colors across span



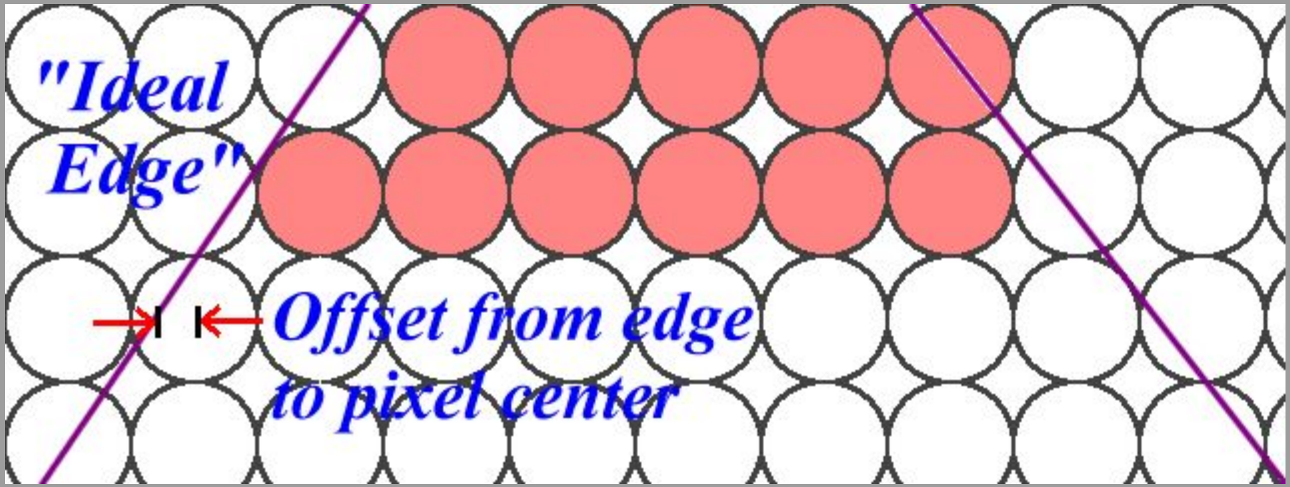
Edge Walking: Notes

- Order vertices in x and y
  - 3 cases: break left, break right, no break
- Walk down left and right edges
  - Fill each span
  - Until breakpoint or bottom vertex is reached
- Advantage: can be made very fast
- Disadvantages:
  - Lots of finicky special cases
  - Tough to get right
  - Need to pay attention to *fractional offsets*

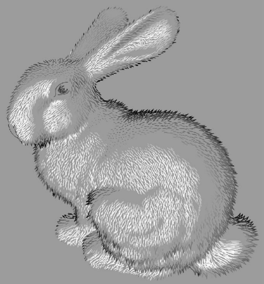


Edge Walking: Notes

- Fractional offsets:



- Be careful when interpolating color values!
- Also: beware gaps between adjacent edges

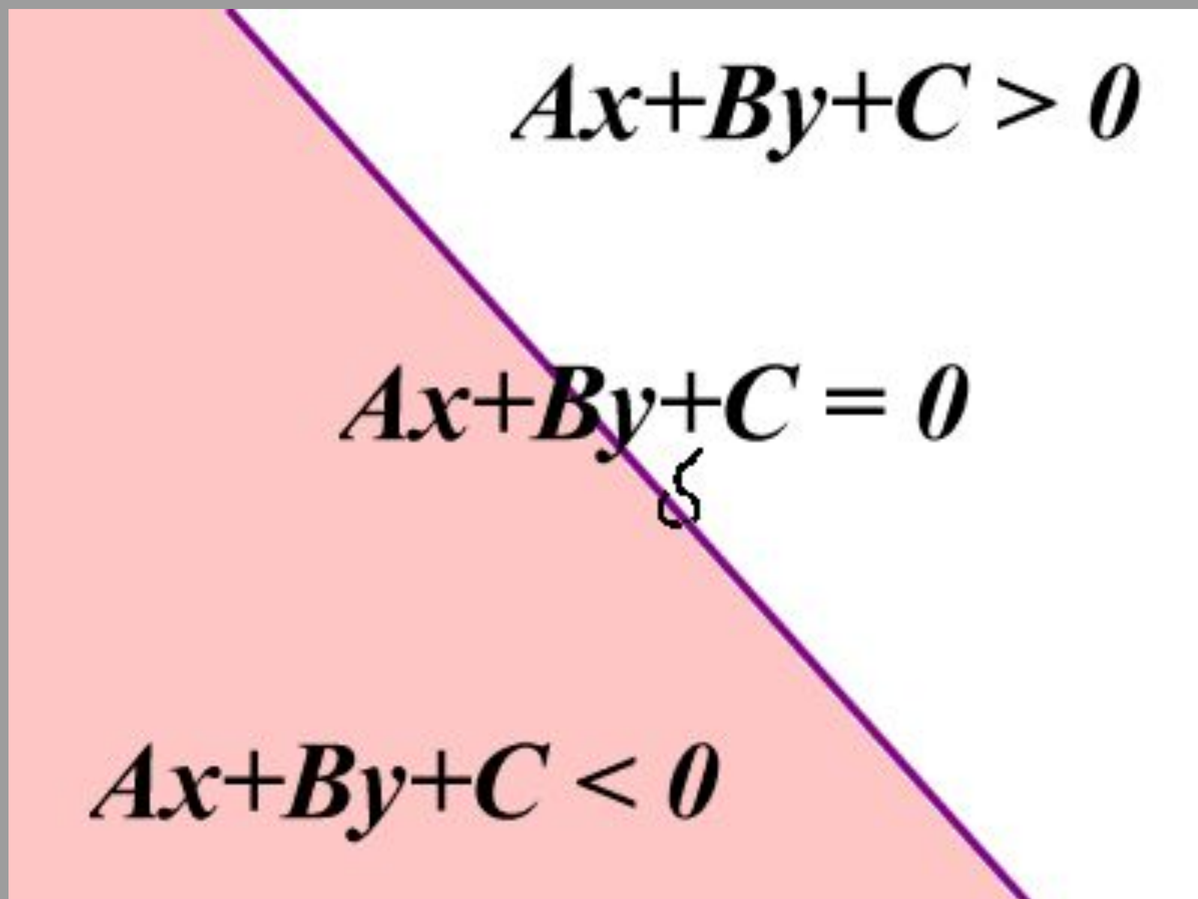


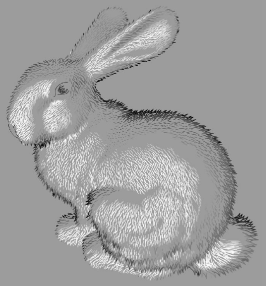
Edge Equations

- An edge equation is simply the equation of the line containing that edge
  - Q: *What is the equation of a 2D line?*
  - A: $Ax + By + C = 0$
  - Q: *Given a point (**x**,**y**), what does plugging **x** & **y** into this equation tell us?*
  - A: Whether the point is:
    - On the line: $Ax + By + C = 0$
    - "Above" the line: $Ax + By + C > 0$
    - "Below" the line: $Ax + By + C < 0$

Edge Equations

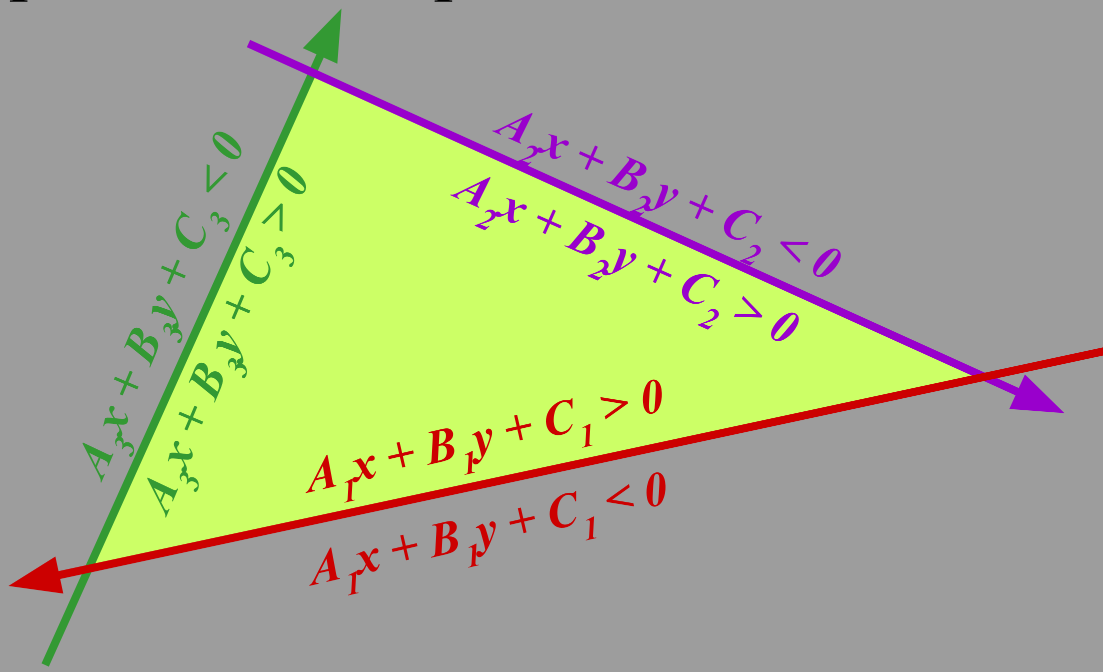
- Edge equations thus define two *half-spaces*:





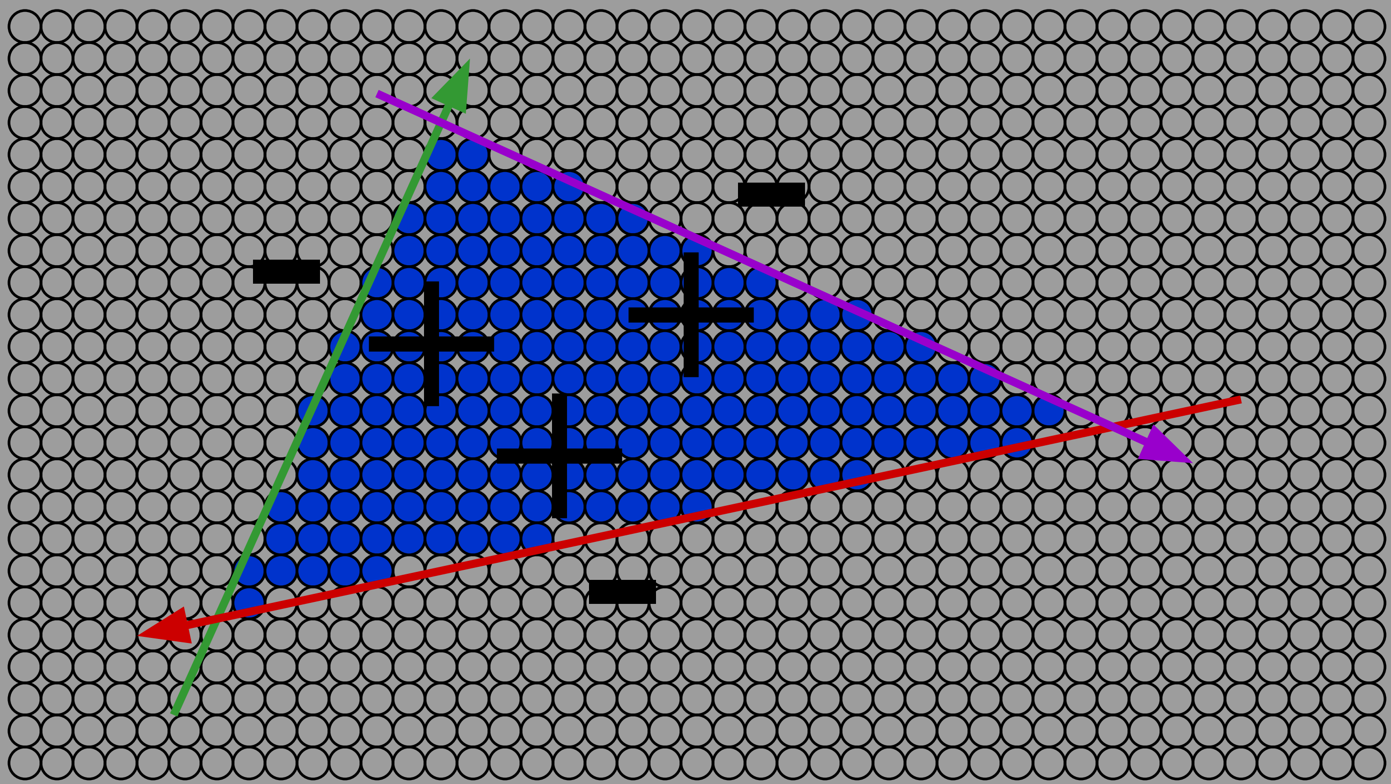
Edge Equations

- And a triangle can be defined as the intersection of three positive half-spaces:

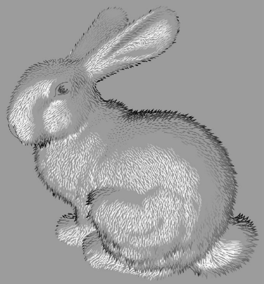


Edge Equations

- So…simply turn on those pixels for which all edge equations evaluate to > 0:



David Luebke

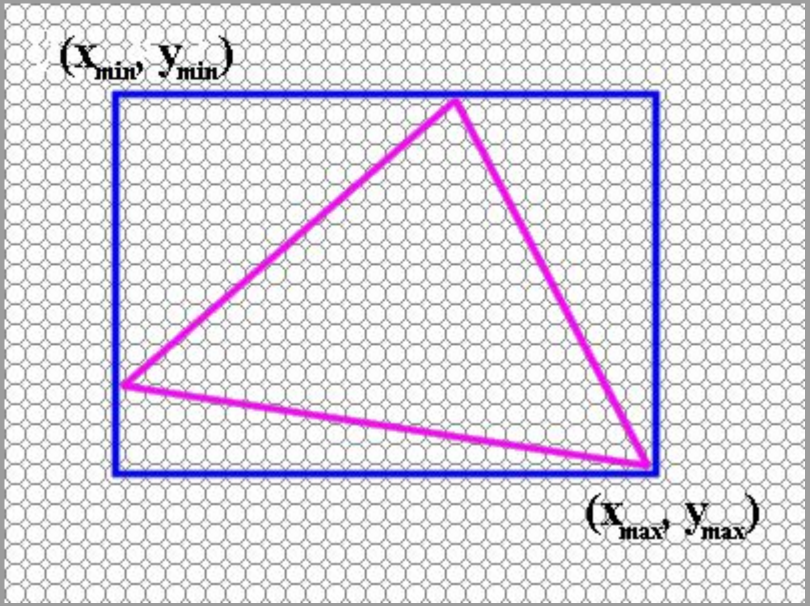


Using Edge Equations

- An aside: *How do you suppose edge equations are implemented in hardware?*
- How would you implement an edge-equation rasterizer in software?
  - *Which pixels do you consider?*
  - *How do you compute the edge equations?*
  - *How do you orient them correctly?*

Using Edge Equations

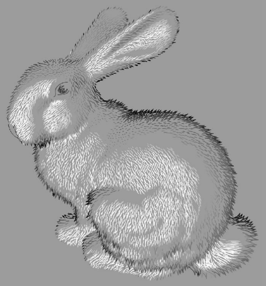
- Which pixels: compute min,max bounding box



- Edge equations: compute from vertices
- Orientation: ensure area is positive (*why?*)

Computing a Bounding Box

- Easy to do
- Surprising number of speed hacks possible
  - See McMillan's Java code for an example



Computing Edge Equations

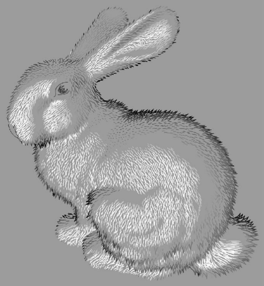
- Want to calculate A, B, C for each edge from ($\boldsymbol{x_i}$, $\boldsymbol{y_i}$) and ($\boldsymbol{x_j}$, $\boldsymbol{y_j}$)
- Treat it as a linear system:

$$Ax_1 + By_1 + C = 0$$
$$Ax_2 + By_2 + C = 0$$

- Notice: two equations, three unknowns
- *Does this make sense? What can we solve?*
- Goal: solve for A & B in terms of C

Computing Edge Equations

- Set up the linear system: $$\begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \end{bmatrix} \begin{bmatrix} A \\ B \end{bmatrix} = -C \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$
- Multiply both sides by matrix inverse: $$\begin{bmatrix} A \\ B \end{bmatrix} = \frac{-C}{x_0 y_1 - x_1 y_0} \begin{bmatrix} y_1 - y_0 \\ x_1 - x_0 \end{bmatrix}$$
- Let $\boldsymbol{C = x_0 y_1 - x_1 y_0}$ for convenience
  - Then $\boldsymbol{A = y_0 - y_1}$ and $\boldsymbol{B = x_1 - x_0}$



Computing Edge Equations: Numerical Issues

- Calculating $\boldsymbol{C = x_0 y_1 - x_1 y_0}$ involves some numerical precision issues
  - *When is it bad to subtract two floating-point numbers?*
  - A: When they are of similar magnitude
    - Example: $1.234 \text{x} 10^4 - 1.233 \text{x} 10^4 = 1.000 \text{x} 10^1$
    - We lose most of the significant digits in result
  - In general, (x_0, y_0) and (x_1, y_1) (corner vertices of a triangle) are fairly close, so we have a problem

Computing Edge Equations: Numerical Issues

- We can avoid the problem in this case by using our definitions of *A* and *B*:

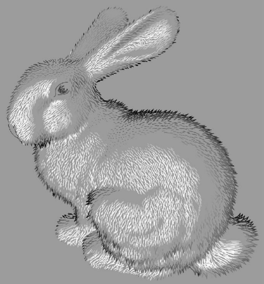
$$A = y_0 - y_1 \qquad B = x_1 - x_0 \qquad C = x_0 y_1 - x_1 y_0$$

Thus:

$$C = -Ax_0 - By_0 \qquad \text{or} \qquad C = -Ax_1 - By_1$$

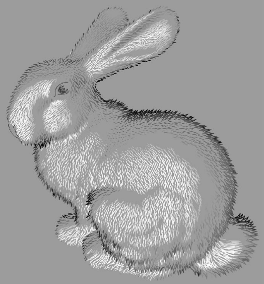
- *Why is this better?*
- *Which should we choose?*
  - Trick question! Average the two to avoid bias:

$$C = -[A(x_0 + x_1) + B(y_0 + y_1)] / 2$$



Edge Equations

- So…we can find edge equation from two verts.
- Given three corners $\mathbf{C_0}$, $\mathbf{C_1}$, $\mathbf{C_0}$ of a triangle, what are our three edges?
- *How do we make sure the half-spaces defined by the edge equations all share the same sign on the interior of the triangle?*
- A: Be consistent (Ex: $[\mathbf{C_0}\,\mathbf{C_1}]$, $[\mathbf{C_1}\,\mathbf{C_2}]$, $[\mathbf{C_2}\,\mathbf{C_0}]$)
- *How do we make sure that sign is positive?*
- A: Test, and flip if needed ($\boldsymbol{A = -A}$, $\boldsymbol{B = -B}$, $\boldsymbol{C = -C}$)



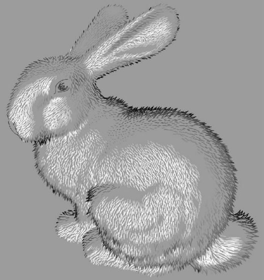
Edge Equations: Code

- Basic structure of code:
  - Setup: compute edge equations, bounding box
  - (Outer loop) For each scanline in bounding box...
  - (Inner loop) …check each pixel on scanline, evaluating edge equations and drawing the pixel if all three are positive

Optimize This!

```
findBoundingBox(&xmin, &xmax, &ymin, &ymax);
setupEdges (&a0,&b0,&c0,&a1,&b1,&c1,&a2,&b2,&c2);

/* Optimize this: */
for (int y = yMin; y <= yMax; y++) {
   for (int x = xMin; x <= xMax; x++) {
      float e0 = a0*x + b0*y + c0;
      float e1 = a1*x + b1*y + c1;
      float e2 = a2*x + b2*y + c2;
      if (e0 > 0 && e1 > 0 && e2 > 0)
  setPixel(x,y);
   }
}
```



Edge Equations: Speed Hacks

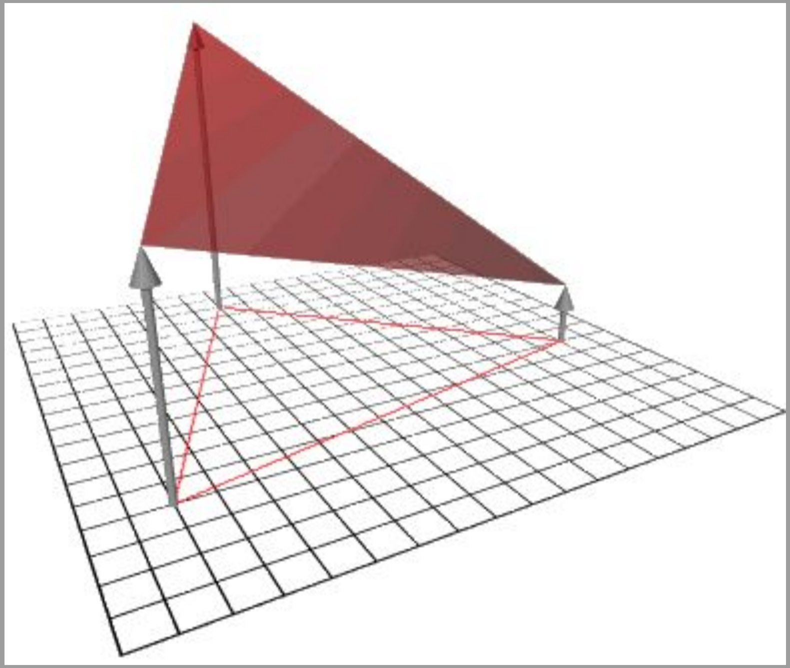
- Some speed hacks for the inner loop:

```
int xflag = 0;
for (int x = xMin; x <= xMax; x++) {
    if (e0|e1|e2 > 0) {
        setPixel(x,y);
        xflag++;
    } else if (xflag != 0) break;
    e0 += a0; e1 += a1; e2 += a2;
}
```

  - *Incremental update of edge equation values* (think DDA)
  - Early termination (*why does this work?*)
  - Faster test of equation values

Edge Equations: Interpolating Color

- Given colors (and later, other parameters) at the vertices, how to interpolate across?
- Idea: triangles are planar in any space:
  - This is the “redness” parameter space
  - Note:plane follows form $z = Ax + By + C$
  - Look familiar?



Edge Equations: Interpolating Color

- Given redness at the 3 vertices, set up the linear system of equations:

$$\begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \end{bmatrix} \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

- The solution works out to:

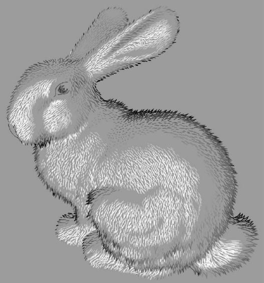
$$\frac{1}{2area} \begin{bmatrix} y_1 - y_2 & y_2 - y_0 & y_0 - y_1 \\ x_2 - x_1 & x_0 - x_2 & x_1 - x_0 \\ x_1 y_2 - x_2 y_1 & x_2 y_0 - x_0 y_2 & x_0 y_1 - x_1 y_0 \end{bmatrix} \begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

Edge Equations: Interpolating Color

- Notice that the columns in the matrix are exactly the coefficients of the edge equations!

$$\frac{1}{2area}\begin{bmatrix} A_2 & A_3 & A_1 \\ B_2 & B_3 & B_1 \\ C_2 & C_3 & C_1 \end{bmatrix}\begin{bmatrix} r_0 \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} A_r \\ B_r \\ C_r \end{bmatrix}$$

- So the setup cost per parameter is basically a matrix multiply
- Per-pixel cost (the inner loop) cost equates to tracking another edge equation value



Triangle Rasterization Issues

- *Exactly which pixels should be lit?*
- A: Those pixels inside the triangle edges
- *What about pixels exactly on the edge?* (Ex.)
  - Draw them: order of triangles matters (it shouldn't)
  - Don't draw them: gaps possible between triangles
- We need a consistent (if arbitrary) rule
  - Example: draw pixels on left or top edge, but not on right or bottom edge