Basic 2D Raster Algorithms

CSC 292:
Computer Graphics

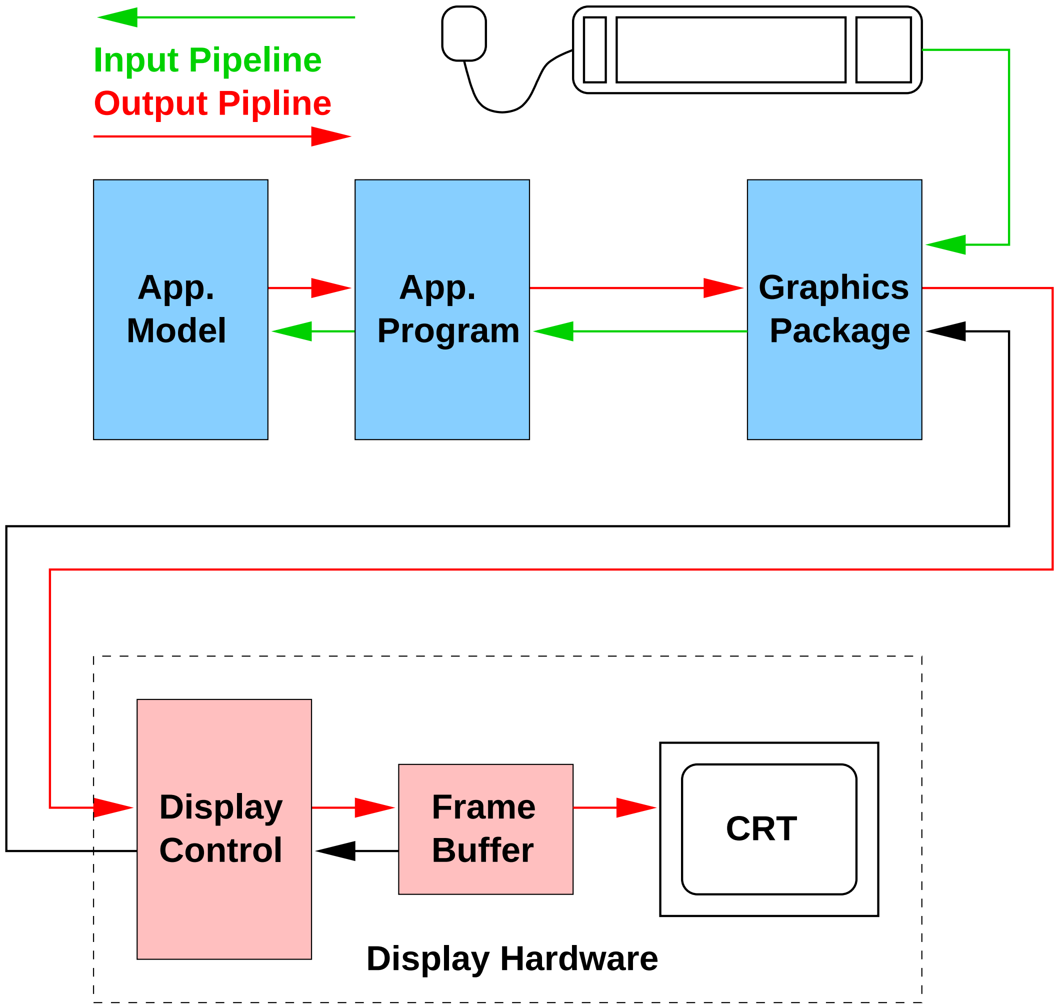
Randal C. Nelson

Topics

- Scan Conversion of lines
- Scan converting circles (and ellipses)
- Filling primitives
- Thick Primitives
- Clipping Primitives
- Generating characters
- Antialiasing

Pipelines

The *output pipeline*: General term for the linear sequence of stages from the application model, through the applications program, and the graphics graphics to the display hardware. The *input pipeline* runs the other way.



Some Scan Conversion Issues

- Varying abilities of display devices
  - Frame buffer only
  - Scan conversion hardware
  - 3D hardware
- Computation must be *FAST*
  - Minimize operations - especially multiplication and division.
  - Use integer arithmetic if possible.
- Scan conversion and clipping
  - Pre-clipping
  - Brute force *scissoring* (dynamic bounds checking)
  - Write and window (wasteful but easy)

Scan Converting Lines

Ideal 1-pixel-wide lines

- One pixel per column (slope -1 to +1), otherwise, one pixel per row
- Want constant brightness
- Want fast algorithm

Other issues

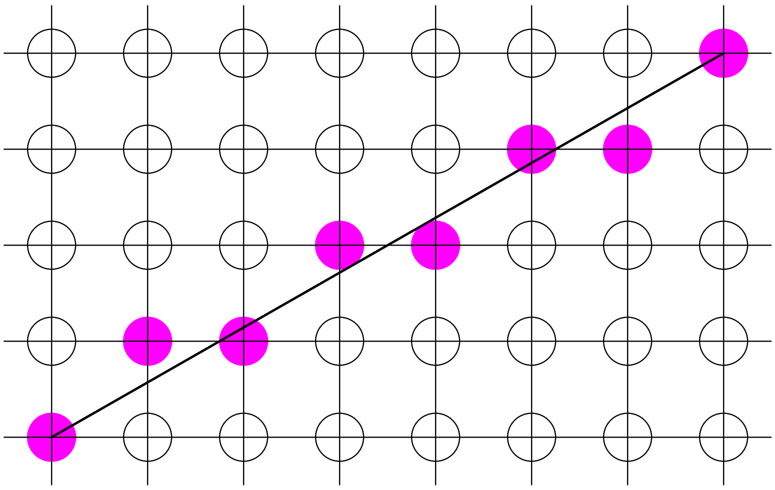
- Wider lines
- Dashed effects
- Beveled, rounded, mitered corners
- Minimizing "jaggies"

Screen geometry issues

- Spot shape (circular)
- Overlapping or spaced spots
- Varying *aspect ratio*
- The other pixel rep (macs)

Basic Incremental Algorithm

Basic idea: For slope between -1 and 1, draw line between endpoints, and turn on the pixel in each column closest to the line.



In detail, for each column x_i from x_{min} to x_{max}, light up pixel $y_i = Round(mx_i + B)$ where the slope m is given by $\Delta y/\Delta x$, and B is the base y value.

Flip axes to handle other slopes.

Problems

- Inefficient: Takes floating point multiply, add, and round (floor) operation per iteration.
- Can get rid of multiply by using *incremental* technique. $y_{i+1} = y_i + m$.
- This method is sometimes called a *digital differential analyzer (DDA)*.
- Still uses floating point.
- Could suffer roundoff error (not for normal images though).

The *Midpoint* Line Algorithm

Start by noting that: If slope of line is between 0 and 1, and we know the digital value $Round(y_i)$, then $Round(y_{i+1})$ is either $Round(y_i)$ or $Round(y_i) + 1$, and we just need to choose between them.

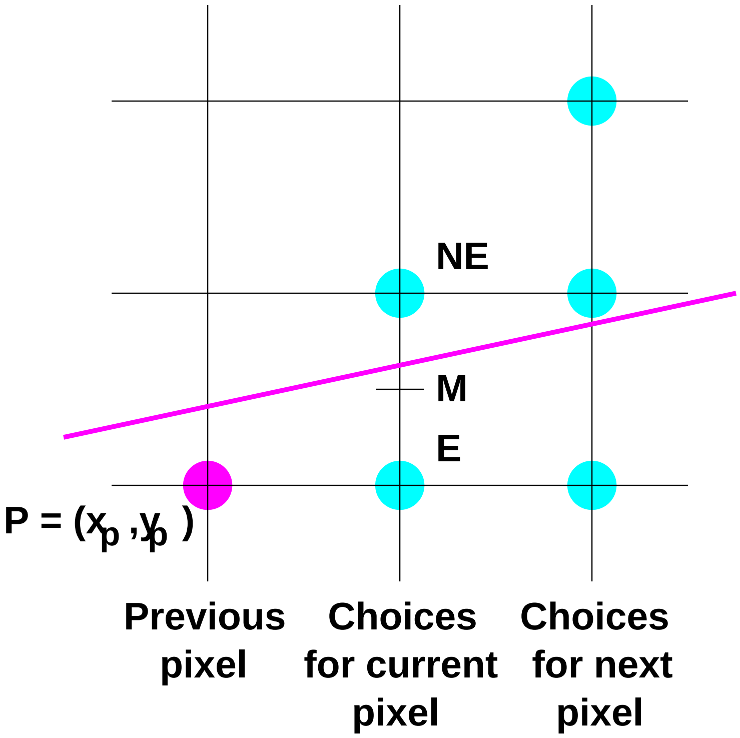
Recall: we get the floating value of at each new column y_{i+1} just by adding the slope increment dy/dx to the floating value y_i at the previous column.

We then pick either the old digital value $Round(y_i)$, or $Round(y_i) + 1$, depending on which is closer.

Insight: We can multiply everything by the denominator dx; thus we add dy at each step, and compare with $Round(y_i)dx$ and $Round(y_i)dx + dx$.

- Same result, no floating point (dx and dy are integer)
- Slight modification: Compare midpoint instead of upper and lower possibilities and branch on sign.

Elegant Mathematical Derivation



- Represent line by parametric equation $F(x, y) = dy \cdot x - dx \cdot y + B \cdot dx = 0$
- $F(x, y)$ is 0 on line, positive above, and negative below.
- Define *decision variable* $d = F(M) = F(x_p + 1, y_p + 1/2)$

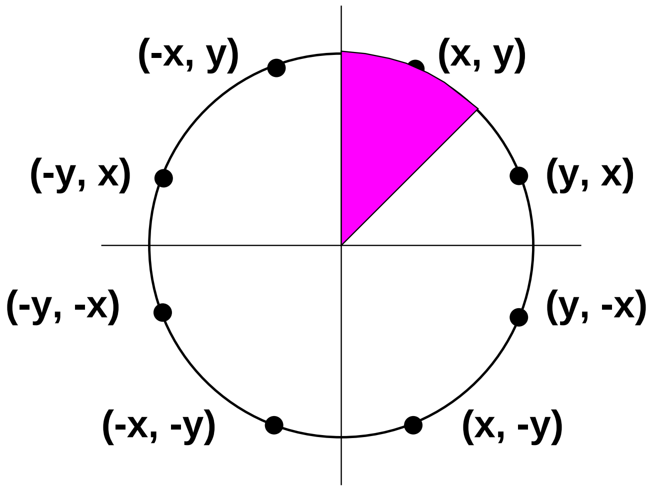
- If $d > 0$ pick NE; if $d <= 0$ pick E
- Once decision is made, we can determine the next value of d by substituting into F and equating terms.
- If E, $d_{new} = d_{old} + dy$
- If NE, $d_{new} = d_{old} + dy - dx$
- Note this is just what we did previously.
- Finally, to get rid of 1/2, multiply everything by 2.

Various Issues

- Endpoint order issue (invert condition for $d = 0$ when going backwards)
- Clipping issues: Must correctly initialize d at clip point.
- Variation of intensity with slope: Must correct pixel brightness for lines at different angles.
- Primitives composed of lines (e.g. polygons): Don't set pixels twice (general principle). Can cause problems with XOR writes and film recorders.

Scan Converting Circles

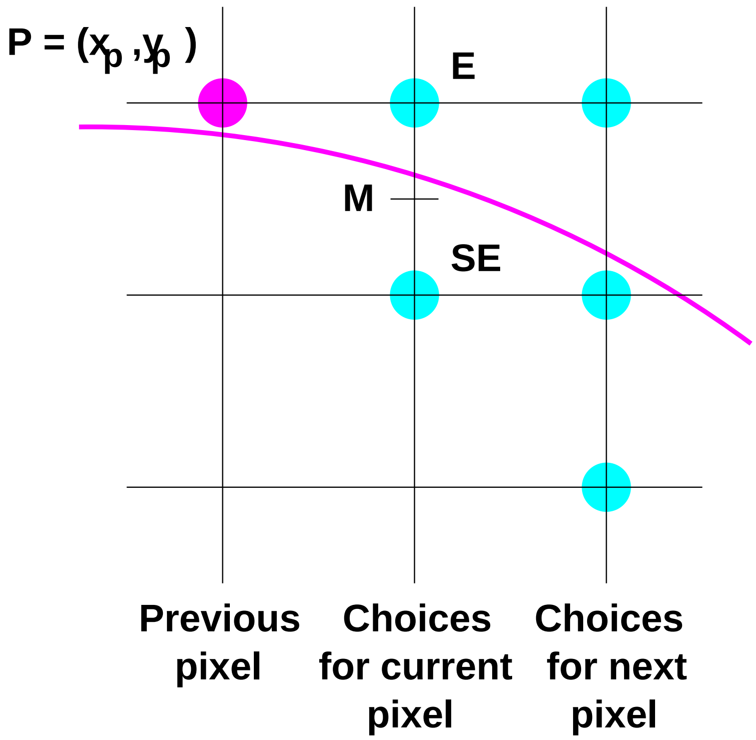
- Consider circles at origin (translate later).
- Simple algorithms using sqrt, or sin and cos are very inefficient
- 8-way symmetry can help. Means we only have to convert a 45 degree arc.



Midpoint Circle Algorithm (patented)

- Like the line algorithm, only a bit more complicated and clever.
- Do only top 45 degrees (second octant), generate the rest by symmetry.
- Same idea of determining which of two candidate pixels is closer to the circle by evaluating a decision function at the midpoint.

Details



- Use parametric function $F(x, y) = x^2 + y^2 - R^2 = 0$
- Then $d_{old} = (x_p + 1)^2 + (y_p - 1/2)^2 - R^2$
- If E then $d_{new} = d_{old} + 2x_p + 3$
- If SE then $d_{new} = d_{old} + 2x_p - 2y_p + 5$
- Note that update is a function of P.
- Also, initialization produces a fractional d; can be eliminated with simple substitution.

Midpoint Ellipse Algorithm

Generalization of circle algorithm, using 4-way instead of 8-way symmetry. Gets pretty tricky eventually, especially if ellipses are tilted, but can be done.

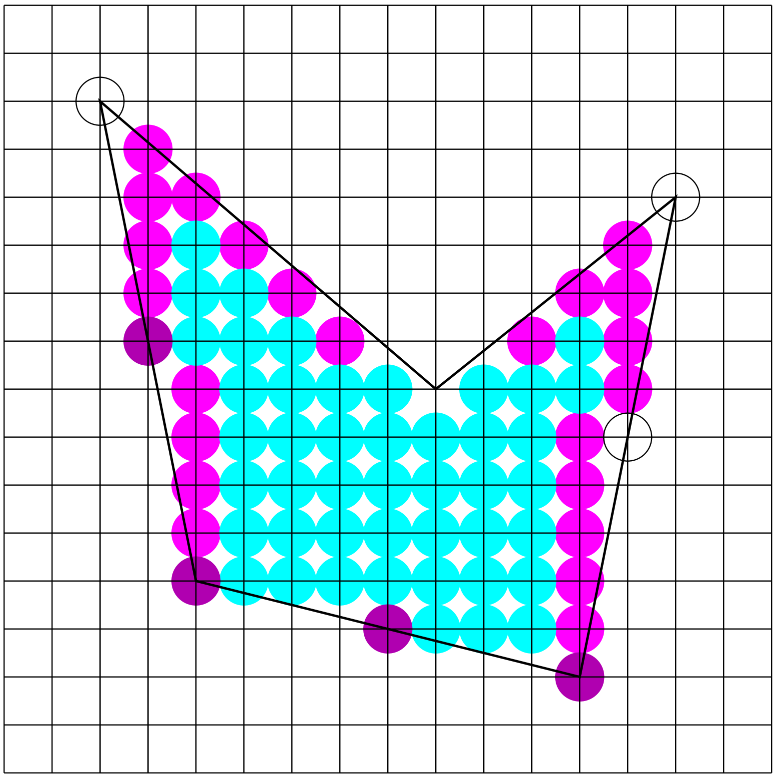
Big point overall: We don't ever have to use sin, cos or sqrt. Savings of factor of 10-100.

Filling Stuff

- Upright rectangles with solid color = easy
- Just fill all the *spans* (rows from left to right) with the appropriate color, scan line by scan line.
- Small problem with pixel duplication in adjacent rectangles.
- Solution: Use *half open* intervals, i.e., don't write the last (highest indexed) pixel in each span, and don't write the last (highest indexed) span.

Filling Polygons

- Basic algorithm: Compute extrema of spans by intersecting scan lines with polygon edges.
- Fill in the interior points.
- Handle integral intersections with same half-open strategy used for rectangles.



Details

- Computing intersections efficiently is tricky.
- Use *spatial coherence* of edges and spans to improve efficiency.
- Modified line scan conversion algorithm tracks edges from scan-line to scan line.
- Bucket sorted *edge table* allows efficient access to edges.
- Write / no-write condition is determined by keeping track of *parity* of edge crossings. Works even for self-intersecting polygons.
- Same sort of strategy works for circles, ellipses and arcs as well.

Problems

- Bad aliasing on *Slivers*
- Pixels are missed because of half-open condition (but better than simple alternatives).
- Sometimes hard to avoid double writes when multiple polygons are present.
- Basic *antialiasing* techniques can address some of these problems.

Pattern Filling

- Use repetitive (tiled) pattern, bitmap or pixmap, instead of solid color (e.g. 8x8).
- Anchor on primitive – pattern moves with primitive
- Anchor to screen – primitive moves through pattern. Easy to do and no interference between primitives.
- For primitives used repeatedly (e.g. characters) it sometimes makes sense to scan-convert once and copy a block of pixels.
- Issues with transparency and overly if we do this.

Thick Primitives

Basic Idea is to draw with “thick brush” along thin path. Some issues are:

- What shape is brush
- What is the brush orientation
- How are line ends handled
- How are intersections (polygon vertices) handled
- How does thickness interact with style considerations

Several different methods:

- Pixel replication (vertically or horizontally)
  - Orientation problems
  - End problems
  - Gaps at joins
  - How to do even widths?
- Moving pen (square or round)
  - Looks a little better than pixel replication
  - Some of same problems
  - How to draw without pixel replication (propagate spans as "pen" is moved)

- Area filling (e.g. between parallel lines or concentric circles)
  - Looks good
  - Ellipses are not quite right mathematically
- Approximation by thick polylines
  - Not bad if you can solve the join problem (Chapter 19)

Line and pen style

How to make dashed and dotted lines:

- Easiest method is to use bitmask to determine whether Consecutive pixels (in thin primitive) are drawn.
- Problems in engineering drawings, since all dashes are not the same length.
- For really good quality, scan convert individual lines segments or rectangles.

Clipping

Recall Methods

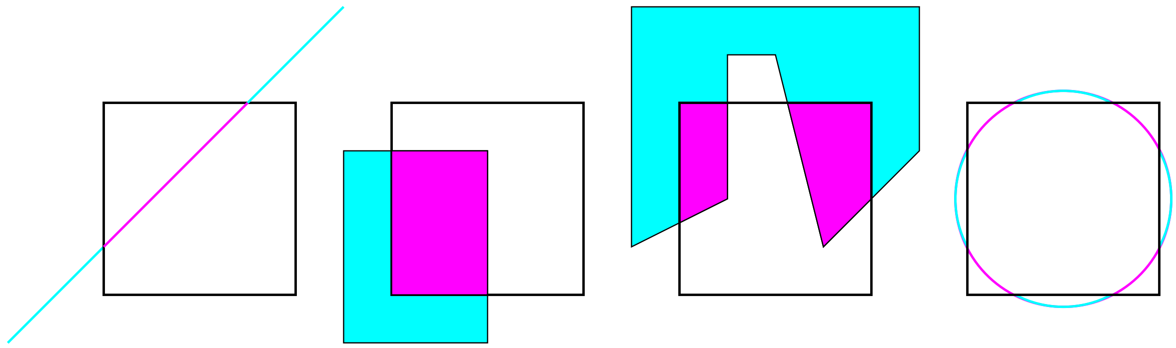
- Analytic bound computation
- Scissoring (dynamic bounds checking)
- Copying from premade image

Analytic should be good, but computing intersection and initialization is often expensive.

Scissoring can be made more efficient by testing (say) every 8 pixels to home in on intersection site.

Clipping Against a Rectangle.

What can happen to various primitives



- Single lines $\Rightarrow$ single lines
- Rectangles $\Rightarrow$ rectangles
- Convex polygons $\Rightarrow$ convex polygons (up to 4 new sides)
- Concave polygons $\Rightarrow$ multiple polygons
- Circles/ellipses $\Rightarrow$ up to 4 arcs

Clipping points – four inequalities to look at.

Clipping Lines

Simultaneous equation (brute force) method

- Check for both endpoints in (*trivial acceptance*)
- Check for one endpoint in; if true, compute intersection.
- Otherwise, consider all four edge-line intersections: parallel special case, 2 simultaneous equations, and 2 bounds checks for each edge.
- One form: Solve $a_1x + b_1y = c_1$ and $a_2x + b_2y = c_2$ simultaneously for x and y.

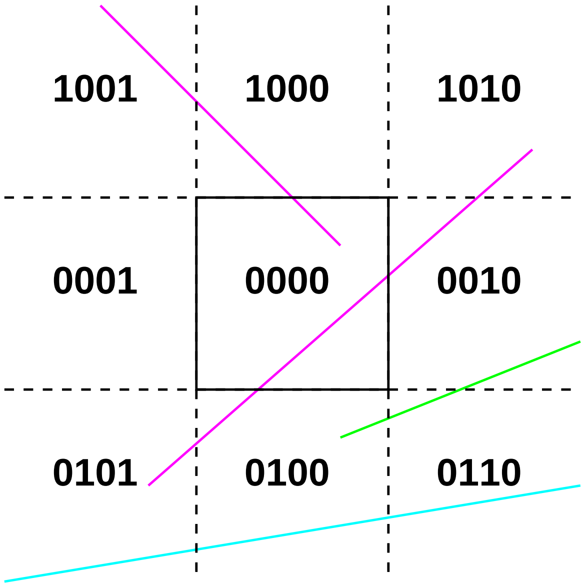
Lots of computation and basically inefficient

Cohen-Southerland Algorithm

More efficient algorithm.

- Check for trivial accept (both endpoints in)
- Check for *trivial reject* using *region checks*, (e.g. if both endpoints have $x < x_{min}$)
- If line can be neither trivially accepted or trivially rejected, then split in two at a clip edge and iterate.
- The extra tests reduce the number of intersection calculations that must be performed.

Details



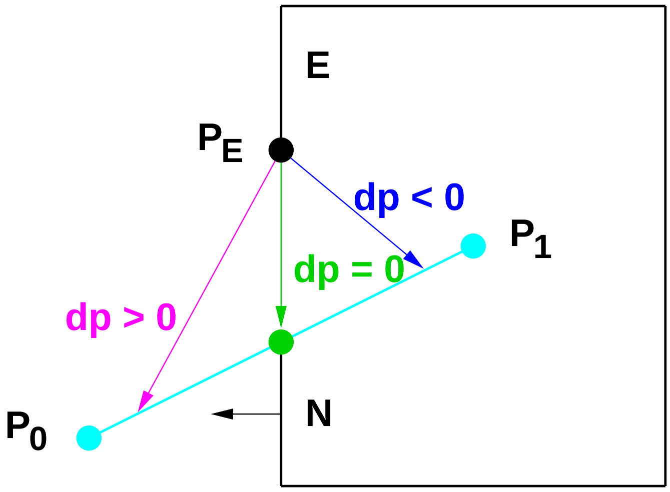
- 4-bit *outcodes* indicate which edges region is outside
- If AND of outcodes of endpoints is non-zero, line can be trivially rejected.
- We can also use outcodes to determine which edges a line may intersect (the ones whose bits change).
- May end up clipping line multiple times.

Cyrus-Beck Method

An even more efficient (and complicated) method

- Formulated using parametric line representation and the fact that the dot product of two perpendicular vectors is 0.
- Works for clipping against arbitrary polygons
- Works against polyhedra in 3D.

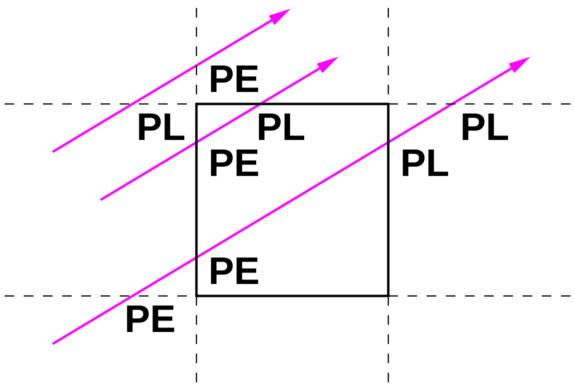
Details



- Parametric equation for line is $P(t) = P_0 + (P_1 - P_0)t$
- Find value of t at intersection by solving $dp = N \cdot [P(t) - P_E] = 0.$
- This yields $t = N \cdot [P_0 - P_E]/(-N \cdot D)$ where D is the vector from P_0 to P_1

How do we find which t values are actually on the clip region?

- First, we don't have to consider any values $t < 0$ or $t > 1$. These are beyond our endpoints.
- Next, we can classify the remaining intersections as *potentially entering* (PE) or *potentially leaving* (PL) on the basis of the sign of the dot product $N \cdot d$.
- Order by increasing t, and the clipped segment, if any, is bounded by the PE with the largest t and the PL with the smallest (or the endpoints).



Clipping Other Primitives

Clipping Circles and ellipses

- Either Scissor, or compute analytic intersection and initialize scan conversion algorithm.

Clipping polygons: Southerland-Hodgman Algorithm

- For each line of polygon, check for clip edge crossing, and compute intersection if it crosses.
- Output intersection points as new polygon vertices as they are found.
- Can be pipelined
- Can introduce spurious edges
- Generalizes to 3D

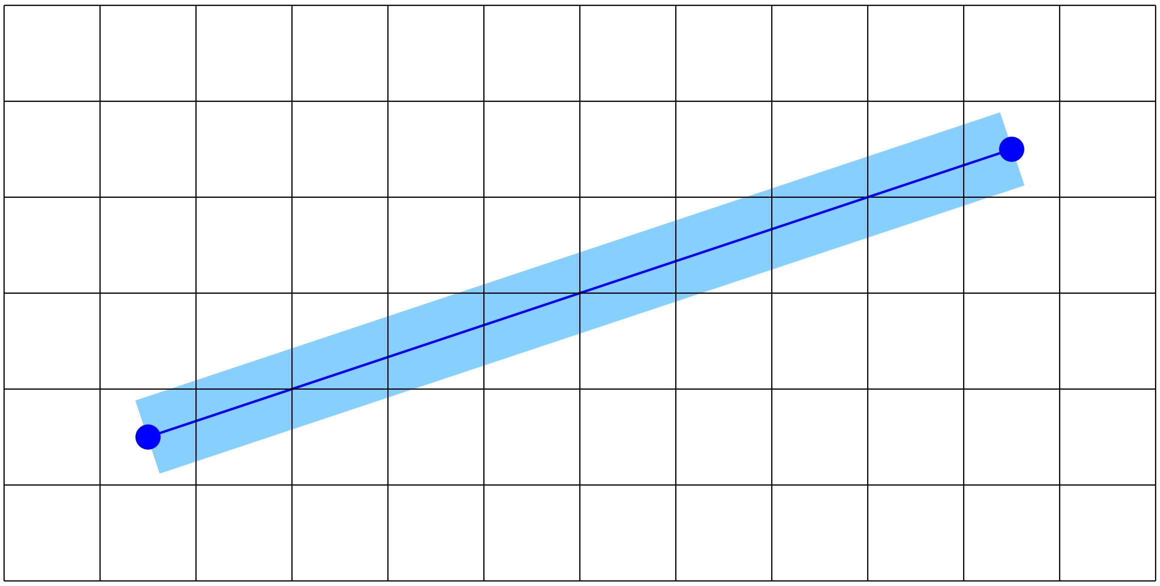
Characters

- Scan convert each time from linear definition (slow).
- Store bitmaps in *font caches* (often in fast vram). May need several – different sizes, different fonts.
- Splined representations can be (to some extent) size independent.
- Build font caches on the fly from *splined representation* – best of both worlds.

Simple *Antialiasing*

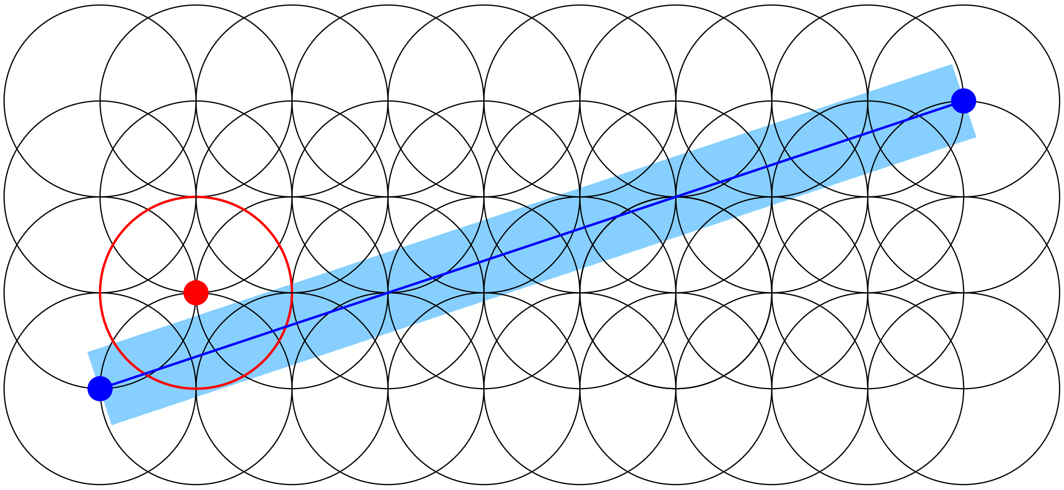
- Digitized representation produces staircasing, "jaggies" etc.
- Increasing resolution is an expensive solution that only diminishes problem.
- Basic idea is to dump all-or-nothing pixel concept, and use graded values to achieve a better visual effect.

Simple idea for lines : Intersect thin (e.g. 1 pixel wide) rectangle with grid of pixels. Referred to as *unweighted area sampling*.



- Somewhat better results than on-off.
- Looks slightly blurred close up.
- Does not take into account distance from center.
- Could be implemented by simulating smaller grid.

A better technique, *weighted area sampling* involves *convolving* with a *weighting function* centered on each pixel. (e.g. a cone or bell shape).



- Generally Support region is bigger than pixel. (e.g. 1 pixel radius).
- Implement efficiently by precomputing overlap table parameterized by distance and angle (Gupta-Sproull algorithm).