CS4620/5620: Lecture 14

Pipeline

Announcements

- HW 2 extension till next Monday

Pipeline overview

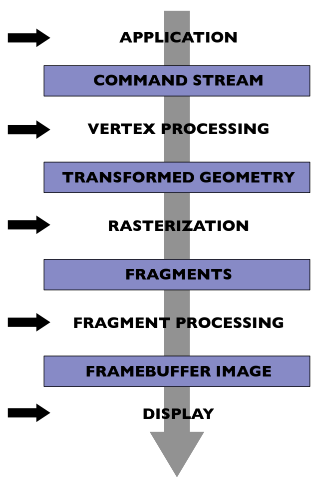
you are here →

3D transformations; shading →

conversion of primitives to pixels →

blending, compositing, shading →

user sees this →



Primitives

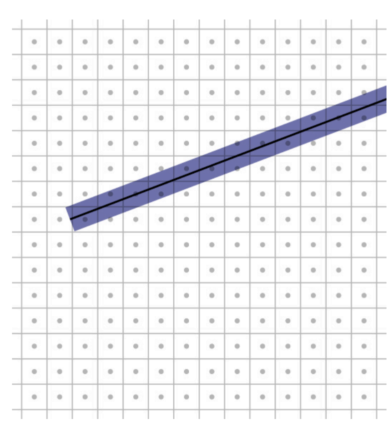
- Points
- Line segments
  - and chains of connected line segments
- Triangles
- And that's all!
  - Curves? Approximate them with chains of line segments
  - Polygons? Break them up into triangles
  - Curved regions? Approximate them with triangles
- Trend has been toward minimal primitives
  - simple, uniform, repetitive: good for parallelism
  - and of course, cyclical; now you can send curves, and the vertex shader will convert to primitives

Rasterization

- First job: enumerate the pixels covered by a primitive
  - simple, aliased definition: pixels whose centers fall inside
- Second job: interpolate values across the primitive
  - e.g. colors computed at vertices
  - e.g. normals at vertices
  - will see applications later on

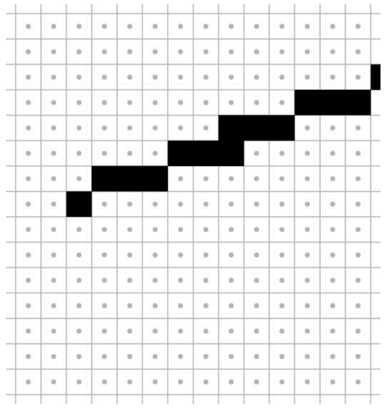
Rasterizing lines

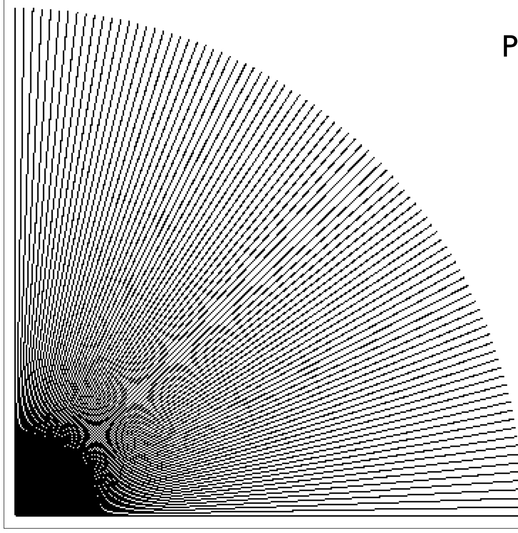
- Define line as a rectangle
- Specify by two endpoints
- Ideal image: black inside, white outside



Point sampling

- Approximate rectangle by drawing all pixels whose centers fall within the line
- Problem: sometimes turns on adjacent pixels

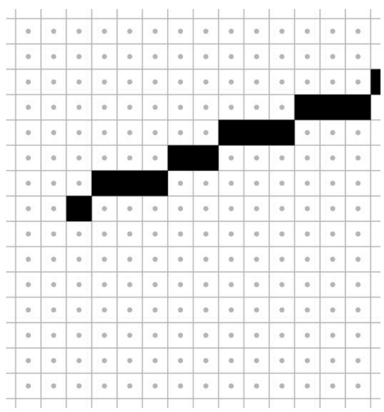


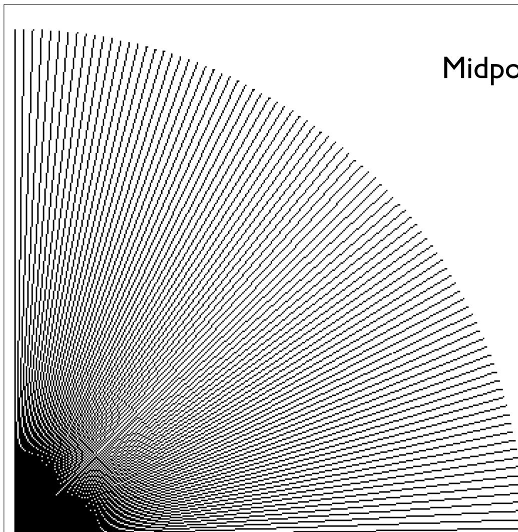


Point sampling
in action

Bresenham lines (midpoint alg.)

- Point sampling unit width rectangle leads to uneven line width
- Define line width parallel to pixel grid
- That is, turn on the single nearest pixel in each column
- Note that 45° lines are now thinner



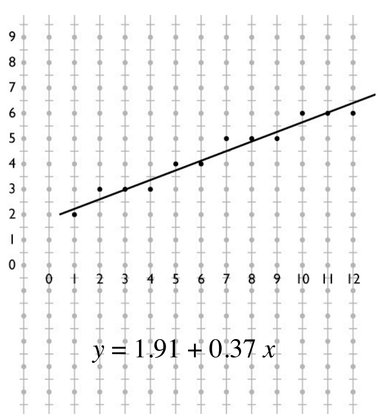


Midpoint algorithm
in action

Algorithms for drawing lines

- line equation:
  $y = b + m\,x$
  $d = m\,x + b - y$
- Simple algorithm: evaluate line equation per column
- W.l.o.g. $x_0 < x_1$;
  $0 \le m \le 1$

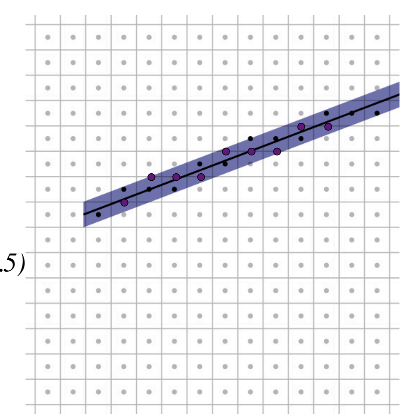
```
for x = ceil(x0) to floor(x1)
    y = b + m*x
    output(x, round(y))
```



Bresenham lines (midpoint alg.)

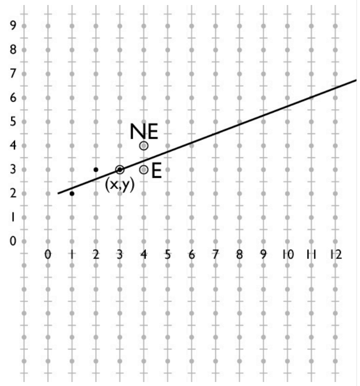
- round (y)?
  - cutoff at midpt

  $y = m\,x + b$
  $d = m\,x + b - y$
- d(x+1,y+0.5)
  $= m(x + 1) + b - (y+0.5)$
- $d > 0$? E : NE
- – what does d > 0 mean?



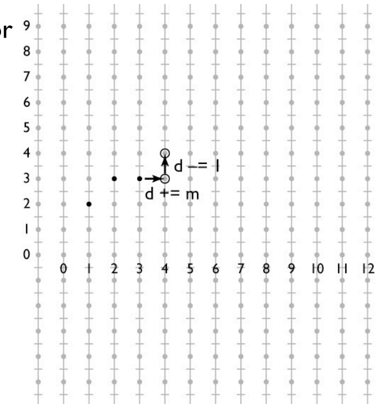
Optimizing line drawing

- Multiplying and rounding: slow
- At each pixel
  - only options are E and NE



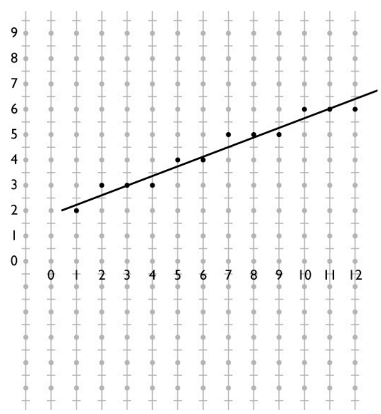
Optimizing line drawing

- Only need to update d for integer steps in x and y
- Do that with addition
- Known as "DDA" (digital differential analyzer)



Midpoint line algorithm

```
x = ceil(x0)
y = y0 = round(m*x + b)
d = m*(x + 1) + b – y
while x < floor(x1)
   if d > 0.5
      y += 1
      d –= 1
   x += 1
   d += m
   output(x, y)
```

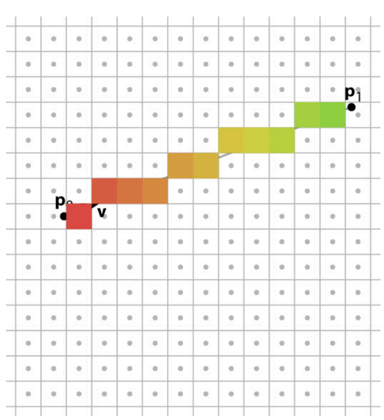


Linear interpolation

- We often attach attributes to vertices
  - e.g. computed diffuse color of a hair being drawn using lines
  - want color to vary smoothly along a chain of line segments
- Recall basic definition
  - 1D: $f(x) = (1 - \alpha)\, y_0 + \alpha\, y_1$
  - where $\alpha = (x - x_0) / (x_1 - x_0)$
- In the 2D case of a line segment, alpha is just the fraction of the distance from (x_0, y_0) to (x_1, y_1)

Linear interpolation

- Pixels are not exactly on the line
- Define 2D function by projection on line
  - this is linear in 2D
  - therefore can use DDA to interpolate

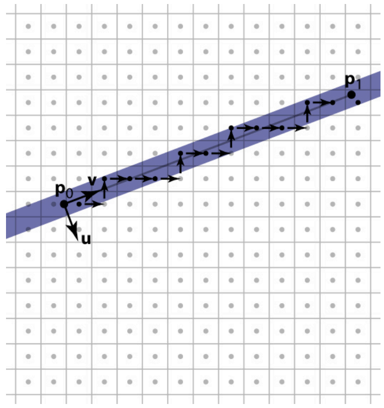


Alternate interpretation

- We are updating d and α as we step from pixel to pixel
  - d tells us how far from the line we are
    α tells us how far along the line we are
- So d and α are coordinates in a coordinate system oriented to the line

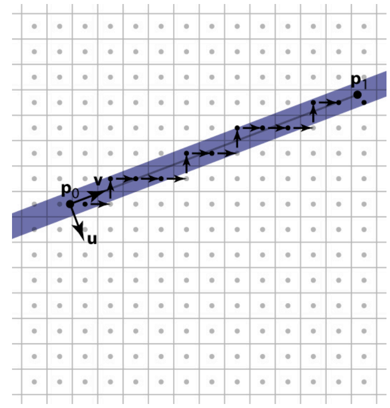
Alternate interpretation

- View loop as visiting all pixels the line passes through
  - Interpolate d and α for each pixel
  - Only output frag. if pixel is in band
- This makes linear interpolation the primary operation



Pixel-walk line rasterization

```
x = ceil(x0)
y = round(m*x + b)
d = m*x + b – y
while x < floor(x1)
    if d > 0.5
        y += 1; d –= 1;
    else
        x += 1; d += m;
    if –0.5 < d ≤ 0.5
        output(x, y)
```



Rasterizing triangles

- The most common case in most applications
  - with good antialiasing can be the only case
  - some systems render a line as two skinny triangles
- Triangle represented by three vertices
- Simple way to think of algorithm follows the pixel-walk interpretation of line rasterization
  - walk from pixel to pixel over (at least) the polygon's area
  - evaluate linear functions as you go
  - use those functions to decide which pixels are inside

Rasterizing triangles

- Input:
  - three 2D points (the triangle's vertices in pixel space)
    - (x_0, y_0); (x_1, y_1); (x_2, y_2)
  - parameter values at each vertex
    - $q_{00}, \ldots, q_{0n}$; $q_{10}, \ldots, q_{1n}$; $q_{20}, \ldots, q_{2n}$
- Output: a list of fragments, each with
  - the integer pixel coordinates (x, y)
  - interpolated parameter values $q_0, \ldots, q_n$

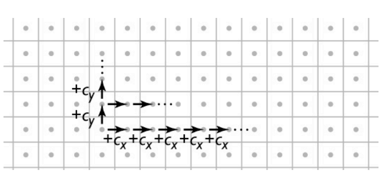
Incremental linear evaluation

- A linear (affine, really) function on the plane is:

$$q(x, y) = c_x x + c_y y + c_k$$

- Linear functions are efficient to evaluate on a grid:

$$q(x+1, y) = c_x(x+1) + c_y y + c_k = q(x, y) + c_x$$
$$q(x, y+1) = c_x x + c_y(y+1) + c_k = q(x, y) + c_y$$

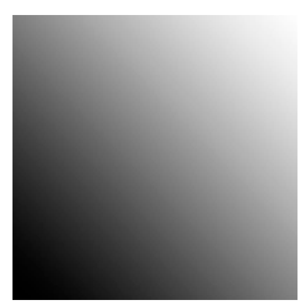


Incremental linear evaluation

```
linEval(xl, xh, yl, yh, cx, cy, ck) {

  // setup
  qRow = cx*xl + cy*yl + ck;

  // traversal
  for y = yl to yh {
    qPix = qRow;
    for x = xl to xh {
      output(x, y, qPix);
      qPix += cx;
    }
    qRow += cy;
  }
}
```



$c_x = .005; c_y = .005; c_k = 0$

(image size 100x100)