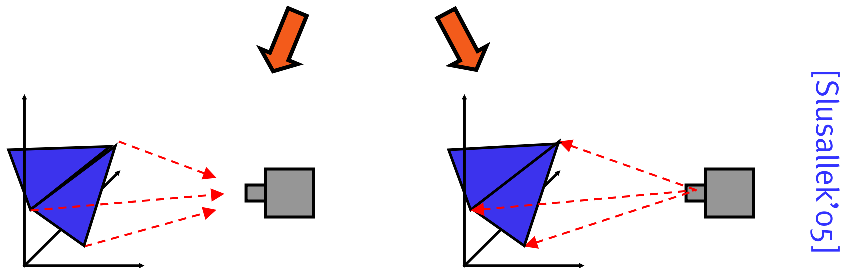
Rasterization

Basic Rendering Model

Models for objects and cameras?



Rasterization:
Project geometry forward

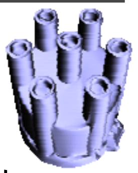
Ray Tracing:
Project image samples backwards

Background

You will write a renderer in several steps

- Step1: frame buffer management
- Step2: rasterization
- Step3: transforms
- Etc…

Meshes



Representation:

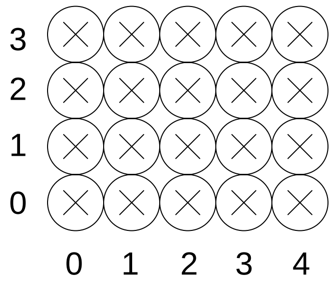
- VRML: list of vertices and triangles (connectivity and geometry).
- Compressed format: more complicated.

Display:

- Ray tracing
- Rasterization…

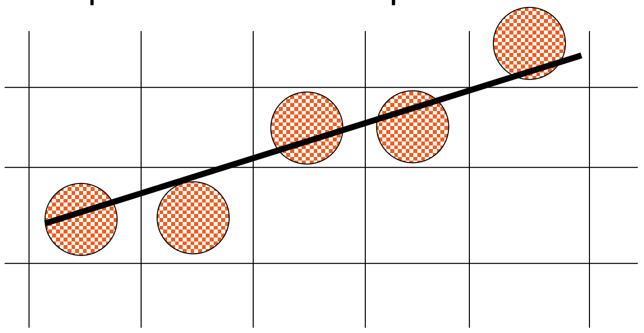
Rasterization

Array of pixels



Rasterizing Lines

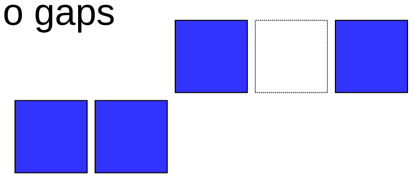
Given two endpoints, $(x_0, y_0), (x_1, y_1)$
find the pixels that make up the line.



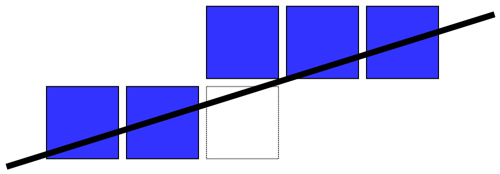
Rasterizing Lines

Requirements

1. No gaps



2. Minimize error (distance to line)



Rasterizing Lines

Equation of a Line:

$$y = mx + b = f(x)$$

Taylor Expansion:

$$y(x + \Delta x) = y + f'(x)\,\Delta x$$

So if we have an x,y on the line,
we can find the next point incrementally.

Rasterizing Lines

Assume –1 < m < 1, x0 < x1

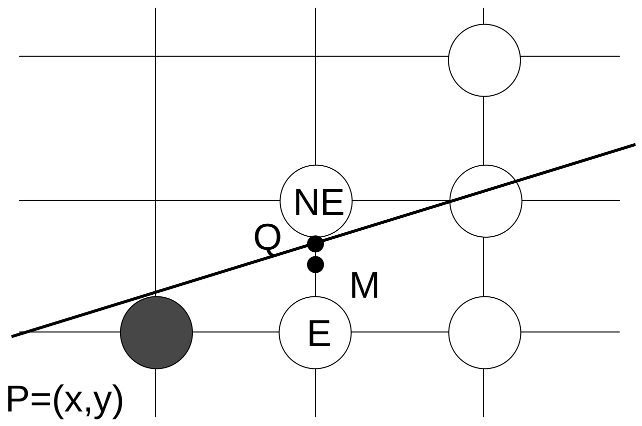
```
Line(int x0, int y0, int x1, int y1)
  float dx = x1 – x0;
  float dy = y1 – y0;
  float m  = dy/dx;
  float x = x0, y= y0;

  for(x = x0; x <= x1; x++)
     setPixel(x,round(y));
     y = y+m;
```

Rasterizing Lines

Problems with previous algorithm

1. round takes time
2. uses floating point arithmetic

Midpoint Algorithm



If Q <= M, choose E. If Q > M, choose NE

Implicit Form of a Line

Implicit form

$$ax + by + c = 0$$

Explicit form

$$y = \frac{dy}{dx} x + B$$

$$dy\ x - dx\ y + B\ dx = 0$$

$$a = dy \quad b = -dx \quad c = B\ dx$$

Positive below the line
Negative above the line
Zero on the line

Decision Function

$$d = F(x, y) = a\,x + b\,y + c$$

$$d = F(x+1, y+\frac{1}{2}) = a\,(x+1) + b\,(y+\frac{1}{2}) + c$$

Choose NE if d > 0
Choose E   if d <= 0

Incrementing d

If choosing E:

$$d_{new} = F(x+2, y+\tfrac{1}{2}) = a\,(x+2) + b\,(y+\tfrac{1}{2}) + c$$

But:

$$d_{old} = F(x+1, y+\tfrac{1}{2}) = a\,(x+1) + b\,(y+\tfrac{1}{2}) + c$$

So:

$$d_{inc} = d_{new} - d_{old} = a = \Delta E$$

Incrementing d

If choosing NE:

$$d_{new} = F(x+2, y+\tfrac{3}{2}) = a(x+2) + b(y+\tfrac{3}{2}) + c$$

But:

$$d_{old} = F(x+1, y+\tfrac{1}{2}) = a(x+1) + b(y+\tfrac{1}{2}) + c$$

So:

$$d_{inc} = d_{new} - d_{old} = a + b = \Delta NE$$

Initializing d

$$d = F(x_0+1, y_0+\frac{1}{2}) = a(x_0+1) + b(y_0+\frac{1}{2}) + c$$

$$= a\,x_0 + b\,y_0 + c + a + b\frac{1}{2}$$

$$= a + b\frac{1}{2}$$

Multiply everything by 2 to remove fractions (doesn’t change the sign)

Midpoint Algorithm

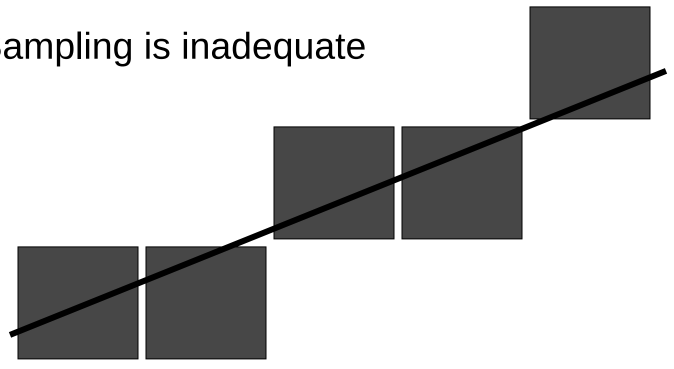
Assume 0 < m < 1, x0 < x1

```
Line(int x0, int y0, int x1, int y1)
  int dx = x1 – x0, dy = y1 – y0;
  int d = 2*dy-dx;
  int delE = 2*dy, delNE = 2*(dy-dx);
  int x = x0, y = y0;
  setPixel(x,y);

  while(x < x1)
     if(d<=0)
        d += delE; x = x+1;
     else
        d += delNE; x = x+1; y = y+1;
     setPixel(x,y);
```

Anti-aliasing Lines

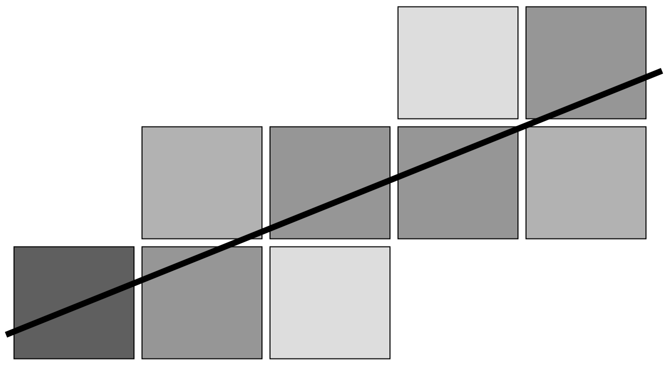
Lines appear jaggy

Sampling is inadequate



Anti-aliasing Lines

Trade intensity resolution for spatial resolution



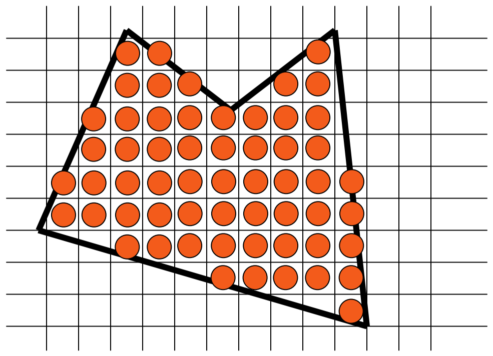
Anti-aliasing Lines

Assume $0 < m < 1$, $x0 < x1$

```
Line(int x0, int y0, int x1, int y1)
  float dx = x1 – x0;
  float dy = y1 – y0;
  float m  = dy/dx;
  float x = x0, y= y0;

  for(x = x0; x <= x1; x++)
     int yi = floor(y); float f = y – yi;
     setPixel(x,yi,  1-f);
     setPixel(x,yi+1, f);
     y = y+m;
```

Rasterizing Polygons

Given a set of vertices and edges,
find the pixels that fill the polygon.



Rasterizing Polygons

vList is an ordered list of the polygon’s vertices

```
fillPoly(vertex vList[ ])
  boundingBox b = getBounds(vList);
  int xmin = b.minX;
  int xmax = b.maxX;
  int ymin = b.minY;
  int ymax = b.maxY;

  for(int y = ymin; y <= ymax; y++)
    for(int x = xmin; x <= xmax; x++)
      if(insidePoly(x,y,vList))
        setPixel(x,y);
```

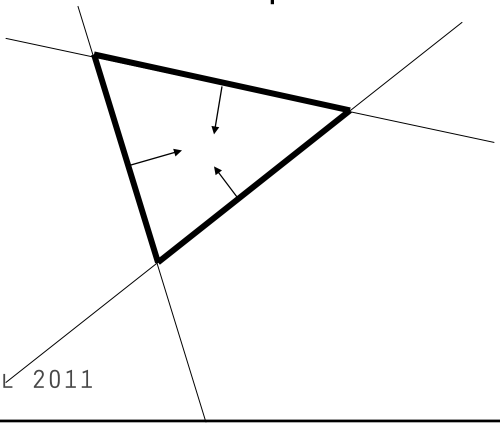
What is Inside?

How to test if a point is inside a polygon
- Half-space tests
- Jordan Curve Theorem

What about shared polygonal edges?
- Shadow Edges algorithm

Half Space Tests

Given the edges of a triangle, the inside is the intersection of half-spaces defined by the edges

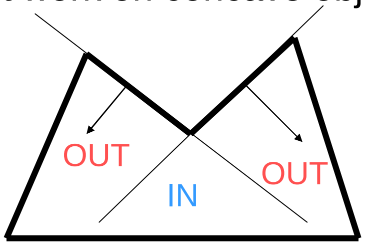


Half Space Tests

Easily computable:

$l(x,y) = ax + by + c < 0$ Iff (x,y) is inside

Doesn't work on concave objects!! (triangulate)



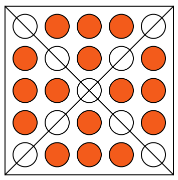
Half Space Tests

lineEq computes the implicit line value for 2 vertices & a point

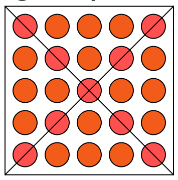
```
fillTriangle(vertex vList[3])
  //-- get the bounding box as before --//
  float e1 = lineEq(vList[0],vList[1],xmin,ymin);
  float e2 = lineEq(vList[1],vList[2],xmin,ymin);
  float e3 = lineEq(vList[2],vList[0],xmin,ymin);
  int xDim = xMax - xMin;

  for(int y = ymin; y <= ymax; y++)
    for(int x = xmin; x <= xmax; x++)
      if(e1<0 && e2<0 && e3<0)
        setPixel(x,y);
    e1 += a1; e2 += a2; e3+= a3;
  e1 += -xDim*a1+b1; e2 = -xDim*a2+b2; e3 = -xDim*a3+b3
```

What about shared Edges?

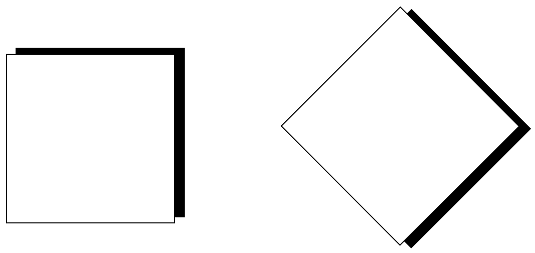
Don't use edges (e==o) – missing pixels



Always use edges (e==0) – waste & flicker



Use Shadow Edges

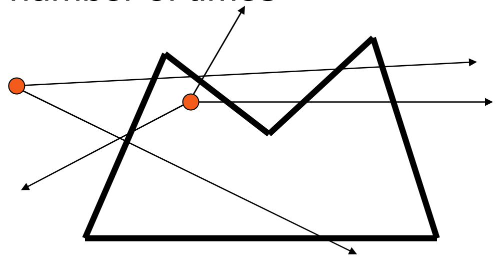


int shadow(a,b) return ((a > 0) || (a == 0 && b > 0))

int inside(e,a,b) return ((e == 0) ? !shadow(a,b) : (e < 0))

Jordan Curve Theorem

Any ray from a point inside a polygon will intersect the polygon's edges an odd number of times

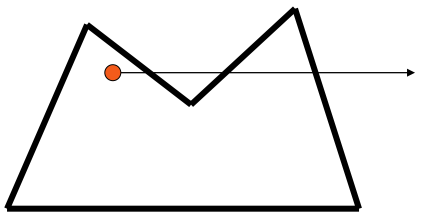


What if it goes through a vertex?

Jordan Curve Theorem

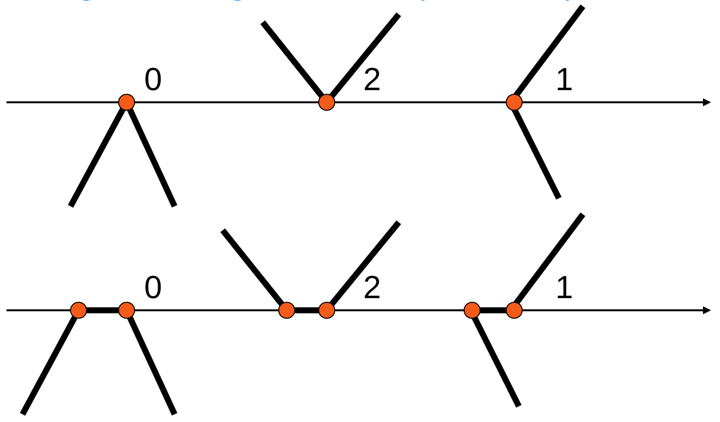
vList is an ordered list of the n polygon vertices

```
int jordanInside(vertex vList[ ], int n, float x, float y)
  int cross = 0;
  float x0, y0, x1, y1;

  x0 = vList[n-1].x – x;     y0 = vList[n-1].y – y;
  for(int i = 0; i < n; i++)
    x1 = vList[i].x – x;      y1 = vList[i].y – y;
    if(y0 > 0)
      if(y1 <= 0)
        if( x1*y0 > y1*x0)
          cross++
    else
      if(y1 > 0)
        if( x0*y1 > y0*x1)
          cross++
    x0 = x1; y0 = y1;

  return cross & 1;
```



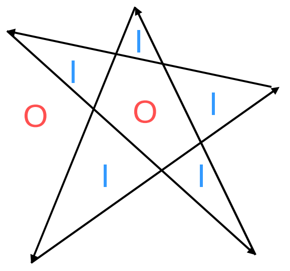
Jordan Curve Theorem

What if it goes through a vertex? (use half open intervals)

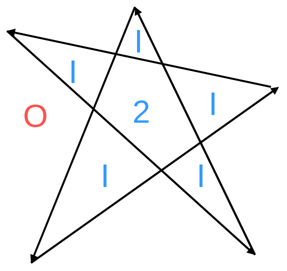


Jordan Curve Theorem

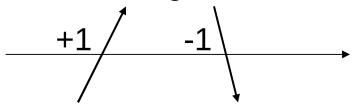
What if the polygon is self-intersecting?



Even-Odd

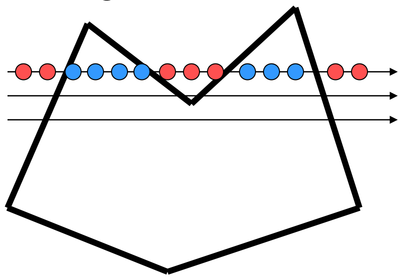


Winding



Scan Line Algorithms

Take advantage of coherence in “insided-ness”

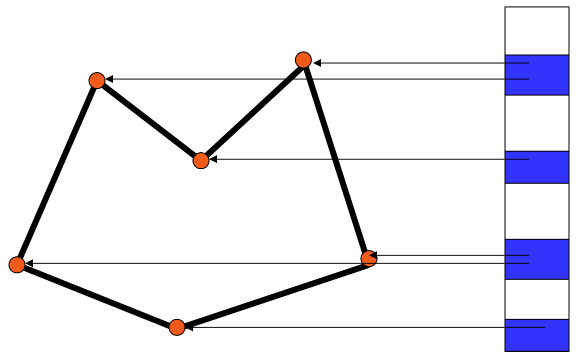


Inside/outside can only change at edge events

Current edges can only change at vertex events

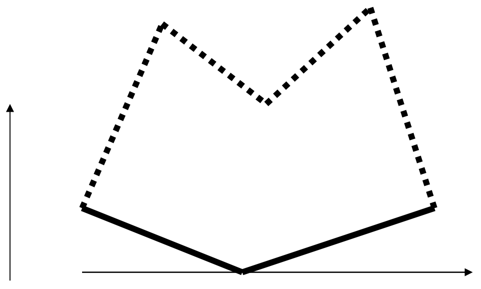
Scan Line Algorithms

Create a list of vertex events (bucket sorted by y)



Scan Line Algorithms

Create a list of the edges intersecting the first scanline



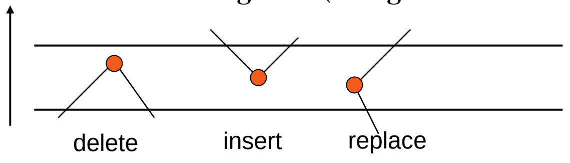
Sort this list by the edge's x value on the first scanline

Call this the active edge list

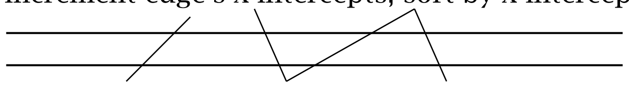
Scan Line Algorithms

For each scanline:

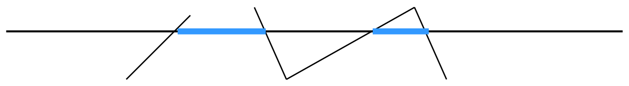
1. Maintain active edge list (using vertex events)



2. Increment edge's x-intercepts, sort by x-intercepts

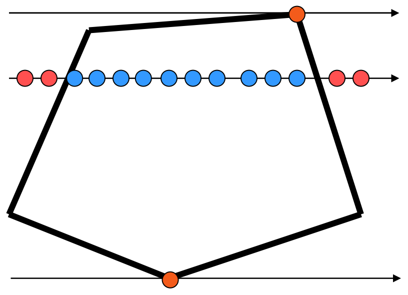


3. Output spans between left and right edges



Convex Polygons

Convex polygons only have 1 span



Insertion and deletion events happen only once

Crow's Algorithm

Find the vertex with the smallest y value to start

```
crow(vertex vList[], int n)
   int imin = 0;

   for(int i = 0; i < n; i++)
      if(vList[i].y < vList[imin].y)
         imin = i;
   scanY(vList,n,imin);
```

Crow's Algorithm

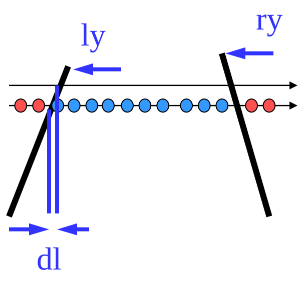
Scan upward maintaining the active edge list

```
scanY(vertex vList[], int n, int i)
   int li, ri;         // left & right upper endpoint indices
   int ly, ry;         // left & right upper endpoint y values
   vertex l, dl;       // current left edge and delta
   vertex r, dr;       // current right edge and delta
   int rem;            // number of remaining vertices
   int y;              // current scanline

   li = ri = i;
   ly = ry = y = ceil(vList[i].y);

(1) for( rem = n; rem > 0)
  (3) // find appropriate left edge
      // find appropriate right edge
  (2) // while l & r span y (the current scanline)
         // draw the span
```

Crow's Algorithm

Draw the spans

```
(2) for( ; y < ly && y < ry; y++)
       // scan and interpolate edges
       scanX(&l, &r, y);
       increment(&l,&dl);
       increment(&r,&dr);
```



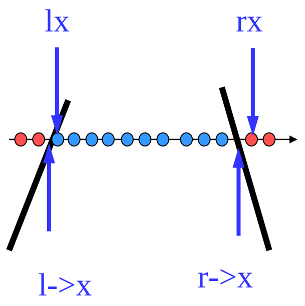
Increment the x value

```
increment(vertex *edge, vertex *delta)
   edge->x += delta->x;
```

Crow's Algorithm

Draw the spans

```
scanX(vertex *l, vertex *r, int y)
  int x, lx, rx;
  vertex s, ds;

  lx = ceil(l->x);
  rx = ceil(r->x);
  if(lx < rx)
    differenceX(l, r, &s, &ds, lx);
    for(x = lx, x < rx; x++)
      setPixel(x,y);
      increment(&s,&ds);
```

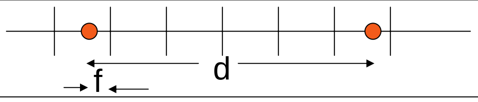


Crow's Algorithm

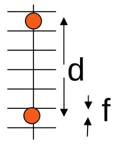
Calculate delta and starting values

```
difference(vertex *v1, vertex *v2, vertex *e, vertex *de, float d, float f)
  de->x = (v2->x – v1->x) / d;
  e->x   = v1->x + f * de->x;
```

```
differenceX(vertex *v1, vertex *v2, vertex *e, vertex *de, int x)
  difference(v1, v2, e, de, (v2->x – v1->x), x – v1->x);
```



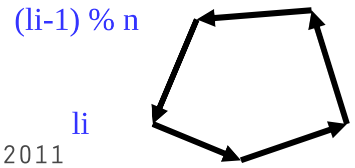
```
differenceY(vertex *v1, vertex *v2, vertex *e, vertex *de, int y)
  difference(v1, v2, e, de, (v2->y – v1->y), y – v1->y);
```



Crow's Algorithm

Find the appropriate next left edge

```
(3) while( ly < = y && rem > 0)
      rem--;
      i = li – 1; if(i < 0) i = n-1;  // go clockwise
      ly = ceil( v[i].y );
      if( ly > y ) // replace left edge
         differenceY( &vList[li], &vList[i], &l, &dl, y);
      li = i;
```



Crow's Algorithm

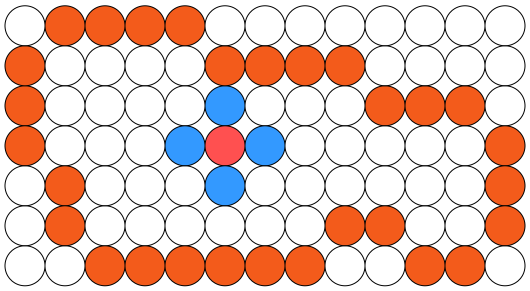
Interpolating other values

```
difference(vertex *v1, vertex *v2, veretx *e, vertex *de, float d, float f)
   de->x = (v2->x – v1->x) / d;
   e->x   = v1->x + f * de->x;
   de->r = (v2->r – v1->r) / d;
   e->r   = v1->r + f * de->r;
   de->g = (v2->g – v1->g) / d;
   e->g   = v1->g + f * de->g;
   de->b = (v2->b – v1->b) / d;
   e->b  = v1->b + f * de->b;
```

```
increment( vertex *v, vertex *dv)
   v->x += dv->x;
   v->r += dv->r;
   v->g += dv->g;
   v->b += dv->b;
```

Flood Fill

How to fill polygons whose edges are already drawn?



Choose a point inside, and fill outwards

Flood Fill

Fill a point and recurse to all of its neighbors

```
floodFill(int x, int y, color c)
   if(stop(x,y,c))
      return;

   setPixel(x,y,c);
   floodFill(x-1,y,c);
   floodFill(x+1,y,c);
   floodFill(x,y-1,c);
   floodFill(x,y+1,c);

int stop(int x, int y, color c)
   return colorBuffer[x][y] == c;
```