15-462 Computer Graphics I
Lecture 14

Rasterization

Scan Conversion
Antialiasing
Compositing
[Angel, Ch. 7.9-7.11, 8.9-8.12]

March 13, 2003
Frank Pfenning
Carnegie Mellon University

http://www.cs.cmu.edu/~fp/courses/graphics/

Rasterization

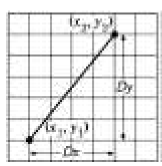
- Final step in pipeline: rasterization (scan conv.)
- From screen coordinates (float) to pixels (int)
- Writing pixels into frame buffer
- Separate z-buffer, display, shading, blending
- Concentrate on primitives:
  - Lines
  - Polygons

DDA Algorithm

- DDA ("Digital Differential Analyzer")
- Represent

$$y = mx + h \quad \text{where} \quad m = \frac{y_2 - y_1}{x_2 - x_1} = \frac{\Delta y}{\Delta x}$$

- Assume $0 \leq m \leq 1$
- Exploit symmetry
- Distinguish special cases

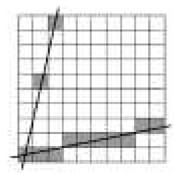


DDA Loop

- Assume write_pixel(int *x*, int *y*, int *value*)

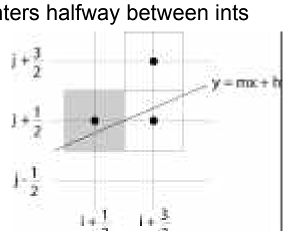
```
For (ix = x1; ix <= x2; ix++)
{
  y += m;
  write_pixel(ix, round(y), color);
}
```

- Slope restriction needed
- Easy to interpolate colors



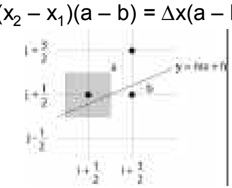
Bresenham's Algorithm I

- Eliminate floating point addition from DDA
- Assume again $0 \leq m \leq 1$
- Assume pixel centers halfway between ints



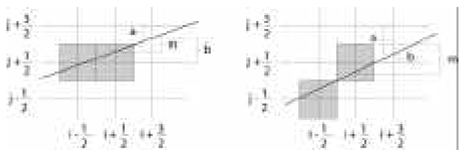
Bresenham's Algorithm II

- Decision variable a – b
  - If a – b > 0 choose lower pixel
  - If a – b $\leq$ 0 choose higher pixel
- Goal: avoid explicit computation of a – b
- Step 1: re-scale $d = (x_2 - x_1)(a - b) = \Delta x(a - b)$
- d is always integer



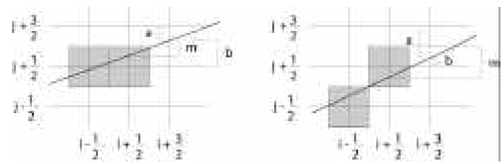
Bresenham's Algorithm III

- Compute d at step k +1 from d at step k!
- Case: j did not change ($d_k > 0$)
  - a decreases by m, b increases by m
  - (a – b) decreases by 2m = $2(\Delta y/\Delta x)$
  - Δx(a-b) decreases by $2\Delta y$



Bresenham's Algorithm IV

- Case: j did change ($d_k \leq 0$)
  - a decreases by m-1, b increases by m-1
  - (a – b) decreases by 2m – 2 = $2(\Delta y/\Delta x - 1)$
  - Δx(a-b) decreases by $2(\Delta y - \Delta x)$



Bresenham's Algorithm V

- So $d_{k+1} = d_k - 2\Delta y$ if $d_k > 0$
- And $d_{k+1} = d_k - 2(\Delta y - \Delta x)$ if $d_k \leq 0$
- Final (efficient) implementation:

```
void draw_line(int x1, int y1, int x2, int y2) {
    int x, y = y0;
    int dx = 2*(x2-x1), dy = 2*(y2-y1);
    int dydx = dy-dx, D = (dy-dx)/2;

    for (x = x1 ; x <= x2 ; x++) {
        write_pixel(x, y, color);
        if (D > 0) D -= dy;
        else {y++; D -= dydx;}
    }
}
```


Bresenham's Algorithm VI

- Need different cases to handle other m
- Highly efficient
- Easy to implement in hardware and software
- Widely used

Outline

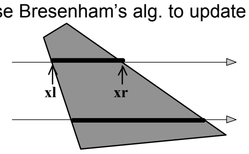
- Scan Conversion for Lines
- Scan Conversion for Polygons
- Antialiasing
- Compositing

Scan Conversion of Polygons

- Multiple tasks for scan conversion
  - Filling polygon (inside/outside)
  - Pixel shading (color interpolation)
  - Blending (accumulation, not just writing)
  - Depth values (z-buffer hidden-surface removal)
  - Texture coordinate interpolation (texture mapping)
- Hardware efficiency critical
- Many algorithms for filling (inside/outside)
- Much fewer that handle all tasks well

Filling Convex Polygons

- Find top and bottom vertices
- List edges along left and right sides
- For each scan line from top to bottom
  - Find left and right endpoints of span, xl and xr
  - Fill pixels between xl and xr
  - Can use Bresenham's alg. to update xl and xr

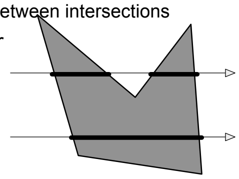


Other Operations

- Pixel shading (Gouraud)
  - Bilinear interpolation of vertex colors
- Depth values (z-Buffer)
  - Bilinear interpolation of vertex depth
  - Read, and write only if visible
  - Preserve depth (final orthographic projection)
- Texture coordinates u and v
  - Rational linear interpolation to avoid distortion
  - u(x,y) = (Ax+By+C)/(Dx+Ey+F) similarly for v(x,y)
  - Two divisions per pixel for texture mapping
  - Due to perspective transformation

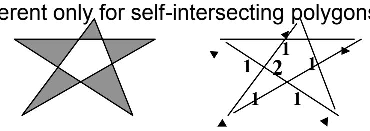
Concave Polygons: Odd-Even Test

- Approach 1: odd-even test
- For each scan line
  - Find all scan line/polygon intersections
  - Sort them left to right
  - Fill the interior spans between intersections
- Parity rule: inside after an odd number of crossings



Concave Polygons: Winding Rule

- Approach 2: winding rule
- Orient the lines in polygon
- For each scan line
  - Winding number = right-hdd – left-hdd crossings
  - Interior if winding number non-zero
- Different only for self-intersecting polygons



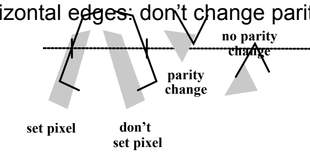
Even-odd rule **Winding rule**

Concave Polygons: Tessellation

- Approach 3: divide non-convex, non-flat, or non-simple polygons into triangles
- OpenGL specification
  - Need accept only simple, flat, convex polygons
  - Tessellate explicitly with tessellator objects
  - Implicitly if you are lucky
- GeForce3 scan converts only triangles

Boundary Cases

- Boundaries and special cases require care
  - Cracks between polygons
  - Parity bugs: fill to infinity
- Intersections on pixel: set at beginning, not end
- Shared vertices: count y_{min} for parity, not y_{max}
- Horizontal edges: don't change parity



Edge/Scan Line Intersections

- Brute force: calculate intersections explicitly
- Incremental method (Bresenham's algorithm)
- Caching intersection information
  - Edge table with edges sorted by y_{min}
  - Active edges, sorted by x-intersection, left to right
- Process image from smallest y_{min} up

Flood Fill

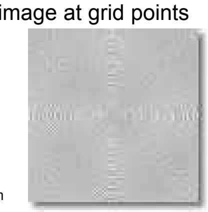
- Draw outline of polygon
- Color seed
- Color surrounding pixels and recurse
- Must be able to test boundary and duplication
- More appropriate for drawing than rendering

Outline

- Scan Conversion for Lines
- Scan Conversion for Polygons
- Antialiasing
- Compositing

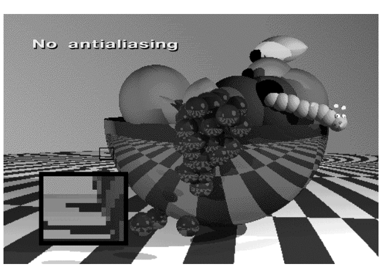
Aliasing

- Artefacts created during scan conversion
- Inevitable (going from continuous to discrete)
- Aliasing (name from digital signal processing): we sample a continues image at grid points
- Effect
  - Jagged edges
  - Moire patterns



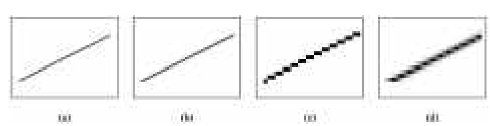
Moire pattern from sandlotscience.com

More Aliasing



Antialiasing for Line Segments

- Use area averaging at boundary

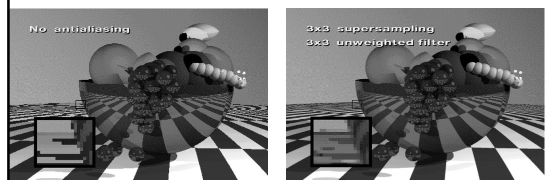


- (c) is aliased, magnified
- (d) is antialiased, magnified
- Warning: these images are sampled on screen!

Antialiasing by Supersampling

- Mostly for off-line rendering (e.g., ray tracing)
- Render, say, 3x3 grid of mini-pixels
- Average results using a filter
- Can be done adaptively
  - Stop if colors are similar
  - Subdivide at discontinuities

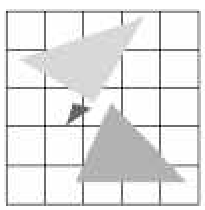
Supersampling Example



- Other improvements
  - Stochastic sampling (avoiding repetition)
  - Jittering (perturb a regular grid)

Pixel-Sharing Polygons

- Another aliasing error
- Assign color based on area-weighted average
- Interaction with depth information
- Use accumulation buffer or α-blending

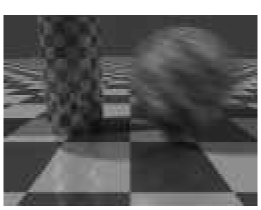


Temporal Aliasing

- Sampling rate is frame rate (30 Hz for video)
- Example: spokes of wagon wheel in movie
- Possible to supersample and average
- Fast-moving objects are blurred
- Happens automatically in video and movies
  - Exposure time (shutter speed)
  - Memory persistence (video camera)
  - Effect is motion blur

Motion Blur

- Achieve by stochastic sampling in time
- Still-frame motion blur, but smooth animation



Motion Blur Example

T. Porter, Pixar, 1984
16 samples/pixel

Outline

- Scan Conversion for Polygons
- Antialiasing
- Compositing

Accumulation Buffer

- OpenGL mechanism for supersampling or jitter
- Accumulation buffer parallel to frame buffer
- Superimpose images from frame buffer
- Copy back into frame buffer for display

```
glClear(GL_ACCUM_BUFFER_BIT);
for (i = 0; i < num_images; i++) {
  glClear(GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT);
  display_image(i);
  glAccum(GL_ACCUM, 1.0/(float)num_images);
}
glAccum(GL_RETURN, 1.0);
```


Filtering and Convolution

- Image transformation at pixel level
- Represent N × M image as matrix **A** = $[a_{ik}]$
- Process each color component separately
- Linear filter produces matrix **B** = $[b_{ik}]$ with

$$b_{ik} = \sum_{j=-m}^{m} \sum_{l=-n}^{n} a_{jl} h_{i-j,k-l}$$

- **B** is the result of convolving **A** with filter **H**
- Represent **H** by n × m convolution matrix

Filters for Antialiasing

- Averaging pixels with neighbors

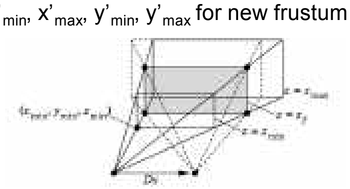
$$\mathbf{H} = \frac{1}{5}\begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

- For antialiasing: weigh center more heavily

$$\mathbf{H} = \frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Filter for Depth-of-Field

- Simulate camera depth-of-field
  - Keep plane $z = z_f$ in focus
  - Keep near and far planes unchanged
- Move viewer by Δx
- Compute x'_{min}, x'_{max}, y'_{min}, y'_{max} for new frustum



Depth-of-Field Jitter

- Compute

$$x'_{min} = x_{min} + \frac{\Delta x}{z_f}(z_f - z_{min})$$

- Blend the two images in accumulation buffer

Blending

- Frame buffer
  - Simple color model: R, G, B; 8 bits each
  - α-channel A, another 8 bits
- Alpha determines opacity, pixel-by-pixel
  - α = 1: opaque
  - α = 0: transparent
- Blend translucent objects during rendering
- Achieve other effects (e.g., shadows)

Image Compositing

- Compositing operation
  - Source: **s** = $[s_r\ s_g\ s_b\ s_a]$
  - Destination: **d** = $[d_r\ d_g\ d_b\ d_a]$
  - **b** = $[b_r\ b_g\ b_b\ b_a]$ source blending factors
  - **c** = $[c_r\ c_g\ c_b\ c_a]$ destination blending factors
  - **d'** = $[b_r s_r + c_r d_r\ \ b_g s_g + c_g d_g\ \ b_b s_b + c_b d_b\ \ b_a s_a + c_a d_a]$
- Overlay n images with equal weight
  - Set α-value for each pixel in each image to 1/n
  - Source blending factor is "α"
  - Destination blending factor is "1"

Blending in OpenGL

- Enable blending

  glEnable(GL_BLEND);
- Set up source and destination factors

  glBlendFund(source_factor, dest_factor);
- Source and destination choices
  - `GL_ONE, GL_ZERO`
  - `GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA`
  - `GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA`

Blending Errors

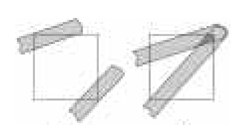
- Operations are not commutative
- Operations are not idempotent
- Interaction with hidden-surface removal
  - Polygon behind opaque one should be culled
  - Translucent in front of others should be composited
  - Solution: make z-buffer read-only for translucent polygons with `glDepthMask(GL_FALSE);`

Antialiasing Revisited

- Single-polygon case first
- Set α-value of each pixel to covered fraction
- Use destination factor of "1 – α"
- Use source factor of "α"
- This will blend background with foreground
- Overlaps can lead to blending errors

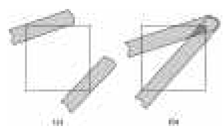
Antialiasing with Multiple Polygons

- Initially, background color $\mathbf{C}_0$, $\alpha_0 = 0$
- Render first polygon; color C_1fraction α_1
  - $\mathbf{C}_d = (1 - \alpha_1)\mathbf{C}_0 + \alpha_1\mathbf{C}_1$
  - $\alpha_d = \alpha_1$
- Render second polygon; assume fraction α_2
- If no overlap (a), then
  - $\mathbf{C'}_d = (1 - \alpha_2)\mathbf{C}_d + \alpha_2\mathbf{C}_2$
  - $\alpha'_d = \alpha_1 + \alpha_2$



Antialiasing with Overlap

- Now assume overlap (b)
- Average overlap is $\alpha_1\alpha_2$
- So $\alpha_d = \alpha_1 + \alpha_2 - \alpha_1\alpha_2$
- Make front/back decision for color as usual



Antialiasing in OpenGL

- Avoid explicit α-calculation in program
- Enable both smoothing and blending

```
glEnable(GL_POINT_SMOOTH);
glEnable(GL_LINE_SMOOTH);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```


Depth Cueing and Fog

- Another application of blending
- Use distance-dependent (z) blending
  - Linear dependence: depth cueing effect
  - Exponential dependence: fog effect
  - This is not a physically-based model

```
GLfloat fcolor[4] = {...};
glEnable(GL_FOG);
glFogf(GL_FOG_MODE; GL_EXP);
glFogf(GL_FOG_DENSITY, 0.5);
glFogfv(GL_FOG_COLOR, fcolor);
```

[Example: Fog Tutor]

Summary

- Scan Conversion for Polygons
  - Basic scan line algorithm
  - Convex vs concave
  - Odd-even and winding rules, tessellation
- Antialiasing (spatial and temporal)
  - Area averaging
  - Supersampling
  - Stochastic sampling
- Compositing
  - Accumulation buffer
  - Blending and α-values

Preview

- Assignment 5 due in one week
- Assignment 6 out in one week
- Next topics:
  - More on image processing and pixel operations
  - Ray tracing