Anti-Aliasing
Outline

• What is anti-aliasing?
  • Sampling theory
• Anti-aliasing for object boundaries
  • Accumulator buffer algorithm
  • Super-sampling & post filtering
• Anti-aliasing for textures
  • MIP mapping
  • Summed-area
What is Anti-aliasing

- **Aliasing:**
  - Jagged edges and random noises appeared in the computer synthesized images.
- **Anti-aliasing:**
  - The technique of minimizing aliasing effects.
Sampling Theory

- In order to reconstruct a signal from a set of samples, the sampling frequency must be at least twice of the maximum signal frequency.
  - \( f_s > 2 \times f_{\text{max}} \)
- Aliasing is caused by lack of samples.
**Aliasing and Line Drawing**

- We draw lines by sampling at intervals of one pixel and drawing the closest pixels.

- Results in stair-stepping.
Anti-aliasing Lines

- Idea:
  - Make line thicker
  - Fade line out (removes high frequencies)
  - Now sample the line
Anti-aliasing Lines

Solution 1 - Unweighted Area Sampling:

- Treat line as a single-pixel wide rectangle
- Color pixels according to the percentage of each pixel covered by the rectangle.
Solution 1: Unweighted Area Sampling

- Pixel area is unit square
- Constant weighting function
- Pixel color is determined by computing the amount of the pixel covered by the line, then shading accordingly
- Easy to compute, gives reasonable results
Solution 2: Weighted Area Sampling

- Treat pixel area as a circle with a radius of one pixel
- Use a radially symmetric weighting function (e.g., cone):
  - Areas closer to the pixel center are weighted more heavily
- Better results than unweighted, slightly higher cost
Gupta-Sproull Algorithm

- Calculate pixel intensity by computing distance from pixel center to line using the midpoint line algorithm.

\begin{align*}
\text{Line to draw} & \quad \text{NE} \\
D & \quad \text{E} \\
\theta & \quad \text{v} \\
\end{align*}
Gupta-Sproull Algorithm (cont)

- d is the perpendicular distance from E to the line
- How do we compute it?

\[
\cos \theta = \frac{\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}}
\]

\[
\cos \theta = \frac{D}{v}
\]

\[
D = \frac{v\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}}
\]
Gupta-Sproull Algorithm

Recall from the midpoint algorithm:

\[ f(x, y) = 2(ax + by + c) = 0 \]

So

\[ y = -\frac{ax + c}{b} \]

and

\[ y_v = -\frac{a(x_p + 1) + c}{b} \]

Therefore

\[ v = y_v - y_p = -\frac{a(x_p + 1) + c}{b} - y_p \]
**Gupta-Sproull Algorithm**

From previous slide:

\[ v = -\frac{a(x_p + 1) + c}{b} - y_p \]

So

\[ -bv = a(x_p + 1) + c + by_p \]

From the midpoint computation,

\[ b = -\Delta x \]

So:

\[ v\Delta x = a(x_p + 1) + by_p + c = \frac{1}{2} f(x_p + 1, y_p) \]
Gupta-Sproull Algorithm

From the midpoint algorithm, we had the decision variable

\[ d = f(m) = f(x_{p+1}, y_p + \frac{1}{2}) \]

Going back to our previous equation:

\[ 2v \Delta x = f(x_p + 1, y_p) \]

\[ = 2a(x_p + 1) + 2by_p + 2c \]

\[ = 2a(x_p + 1) + 2b(y_p + \frac{1}{2}) - 2b \frac{1}{2} + 2c \]

\[ = f(x_p + 1, y_p + \frac{1}{2}) - b \]

\[ = f(m) - b \]

\[ = d - b \]

\[ = d + \Delta x \]
Gupta-Sproull Algorithm

So,

\[
D = \frac{v \Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} = \frac{d + \Delta x}{2\sqrt{\Delta x^2 + \Delta y^2}}
\]

The denominator is constant.
Since we are blurring the line, we also need to compute the color at the pixels above and below the E pixel

\[
D_{up} = \frac{(1 - v) \Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} \quad D_{lower} = \frac{(1 + v) \Delta x}{\sqrt{\Delta x^2 + \Delta y^2}}
\]
Gupta-Sproull Algorithm

If the NE pixel had been chosen:

\[ 2v\Delta x = f(x_p + 1, y_p + 1) \]
\[ = 2a(x_p + 1) + 2b(y_p + 1) + 2c \]
\[ = 2a(x_p + 1) + 2b(y_p + \frac{1}{2}) + 2b\frac{1}{2} + 2c \]
\[ = f(x_p + 1, y_p + 1/2) + b \]
\[ = f(m) + b \]
\[ = d + b \]
\[ = d - \Delta x \]

\[ D = \frac{v\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} = \frac{d - \Delta x}{2\sqrt{\Delta x^2 + \Delta y^2}} \]
\[ D_{up} = \frac{(1 - v)\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} \]
\[ D_{lower} = \frac{(1 + v)\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} \]
Gupta-Sproull Algorithm Summary

- Compute midpoint line algorithm, with the following alterations at each iteration:
  - At each iteration of the algorithm:
    - If the E pixel is chosen
      \[ D = \frac{d + \Delta x}{2\sqrt{\Delta x^2 + \Delta y^2}} \]
    - If the NE pixel is chosen
      \[ D = \frac{d - \Delta x}{2\sqrt{\Delta x^2 + \Delta y^2}} \]
    - Update d as in the regular algorithm
    - Color the current pixel according to D
    - Compute
      \[ D_{up} = \frac{(1 - v)\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} \quad D_{lower} = \frac{(1 + v)\Delta x}{\sqrt{\Delta x^2 + \Delta y^2}} \]
    - Color upper and lower pixels accordingly
Solution 3: Super-sampling

- Divide pixel up into “sub-pixels”: 2×2, 3×3, 4×4, etc.
- Sub-pixel is colored if inside line
- Pixel color is the average of its sub-pixel colors
- Easy to implement (in software and hardware)

No anti-aliasing

Anti-aliasing (2×2 super-sampling)
Many Types of Supersampling

- **Grid**
- **Random**
- **Poisson Disc**
- **Jittered**
Foreground and Background

- Compute percent of pixel covered by line, $p$
- Line color is $c_l$
- Background color is $c_b$
- Pixel color is computed as

$$color = p \cdot c_l + (1-p) \cdot c_b$$
Polygon Anti-aliasing

- To anti-alias a line, we treat it as a rectangle
- Anti-aliasing a polygon is similar.
- Some concerns:
  - Micro-polygons: smaller than a pixel
  - Super-sampling: There may still be polygons that “slip between the cracks”
Aliasing Caused by Object Boundaries

- Pixels along the object boundaries are not fully covered by a single object.
  - Using one object to calculate pixel color causes jagged edges.
- For anti-aliasing:
  - Model each pixel as a square window.
  - Calculate pixel color based on the percentage covered by different objects.
A-Buffer Algorithm

- A straightforward anti-aliasing approach is proposed by Catmull in 1978
  - Based on scanline approach.
  - Involves clipping polygons against the square window and compute the area of overlapping region.
- Accumulator buffer (A-buffer) algorithm is proposed by Carpenter in 1984.
  - Based on the Z-buffer algorithm.
  - Use bitwise logical operators between the masks that representing polygon fragments.
    - *Floating point geometry calculations are avoided.*
  - More efficient than Catmull’s approach.
A-Buffer Algorithm (Cont’d)

• The area covered by a given polygon is represented using a 32-bit integer.
  • Each bit indication whether the corresponding position is covered.
  • Clipping one polygon against another becomes a simple bitwise Boolean operation.
Super-Sampling & Post Filtering

- Basic idea:
  - Generate a set of samples at higher frequency than the image resolution required.
  - The color of a given pixel is calculated using the average of samples within the window.
Uniform Sampling

- Increases the sampling density uniformly across the image.
- For an 1024×768 image:
  - First render at 4096×3072 resolution.
  - Then calculate the average of 16×16 samples.
Adaptive Sampling

- Uniform sampling introduces high computational costs.
  - For pixels that are covered by a single polygon, high sampling rate is unnecessary.
- Adaptive sampling only increases sampling rate in locations that is needed.
Stochastic Sampling

- Both uniform & adaptive sampling samples at predefined locations.
  - Aliasing may still appear for objects with patterns.
- Stochastic sampling samples a pixel at random locations.
  - For each pixel 10 locations randomly determined and are used for sampling.
Aliasing Caused by Textures

- Textures provide details (high frequency signal) and are prone to aliasing effects.
- The 3D surface covered by a single pixel may map to a large area in texture space.
  - Using color at a single location causes aliasing.
- For anti-aliasing, the average color within the area should be used.
Averaging in Texture Space

- The area covered by a square pixel in texture space can be an arbitrary shape.
- The size and shape of the area depends on surface orientation and distance to the viewpoint.
  - The further away the object, the larger the area is.
MIP Mapping

- MIP mapping works by creating lower resolution, pre-filtered versions of the original texture.
  - MIP is acronym of Latin phrase "multum in parvo" - "much in a small space".
- When rendering, the appropriate resolution MIP map is chosen.
  - A single pixel in lower resolution keeps the average of a square area.
MIP Map
Summed-Area Approach

- A pixel in a MIP map covers a square region:
  - The averaging is always isotropic.
  - Cannot support anisotropic filtering.
- The summed-area approach uses a rectangle of arbitrary aspect ratio.
  - The average within the rectangle region can be calculated efficiently using summed-area table.
Summed-Area Table

• A summed-area table pre-calculates the sum of different rectangles starting from the top-left corner:

\[ S_{m,n} = \sum_{0 \leq i \leq m, 0 \leq j \leq n} t_{i,j} \]

• The average within an arbitrary rectangle can be calculated using:

\[ \frac{S_{r,b} - S_{l,b} - S_{r,t} + S_{l,t}}{(r - l) \times (b - t)} \]