Today’s class

- Drawing lines
- Bresenham’s algorithm
- Compositing
- Polygon filling
Drawing lines

- Plot line from \((x_1, y_1)\) to \((x_2, y_2)\)
- \(y = mx + b\)
  - \(m = (y_2 - y_1) / (x_2 - x_1)\)
  - \(b = y_1 - mx_1\)
- Straight-forward approach:
  - for \(x = x_1\) to \(x_2\) do
    - \(y = m \times x + b\)
    - WritePixel \((x, \text{round}(y))\)
Digital differential analyzer

- Straight-forward approach involves one multiplication and one addition per iteration of the loop
- Recognize $\Delta y = m \Delta x$
- Choose $\Delta x$ to be 1; $\Delta y = m$; $y_{i+1} = y_i + m$
- Digital differential analyzer (DDA):
  - $y = y_1$
  - for $x = x_1$ to $x_2$ do
    - WritePixel ($x$, round($y$))
    - $y = y + m$
- Have removed the multiplication from the loop
Effect of slope

- The DDA algorithm shown assumes left-to-right processing (i.e., $x_1 < x_2$) and $|\text{slope}| \leq 1$
- For $|\text{slope}| > 1$ reverse $x$ and $y$ roles
  - step along $y$
  - change $x$ by $1 / m$
Remaining problems

- There are still two problems that remain to be removed:
  - one real addition per iteration of the loop
  - one rounding per iteration of the loop
- Would like to get the algorithm down to only integer operations
- Bresenham’s algorithm will do this
Bresenham’s algorithm

- An algorithm for drawing lines with only integer operations
- Consider the case where $0 \leq \text{slope} \leq 1$
- Suppose $(x_i, y_i)$ is plotted
- Which $y$ value do you plot at $x_i+1$? Is it $y_i$ or $y_i+1$?
- The true value is $y$
Error terms

- Let $e_1 = y_{i+1} - y = y_{i+1} - (m(x_{i+1}) + b)$
- Let $e_2 = y - y_i = m(x_{i+1}) + b - y_i$
- $e_1 - e_2 = 2y_i - 2m(x_{i+1}) - 2b + 1 = 2y_i - 2(\Delta y/\Delta x)(x_{i+1}) - 2b + 1$
- If this difference is negative, $e_2 > e_1$ so $y_{i+1}$ is closer to true value than $y_i$
Definition of $p_i$

- Multiply through by $\Delta x$ to get rid of fraction
- $p_i = \Delta x(e_1-e_2) =$
  
  $2y_i\Delta x - 2\Delta y(x_i+1) - 2b\Delta x + \Delta x =$
  
  $2\Delta xy_i - 2\Delta yx_i - 2\Delta y - \Delta x(2b-1)$

- $p_{i+1} = 2\Delta xy_{i+1} - 2\Delta yx_{i+1} - 2\Delta y - \Delta x(2b-1)$
Updating equation for $p_i$

- $p_{i+1} - p_i =$
  
  $2\Delta xy_{i+1} - 2\Delta xy_i - 2\Delta yx_{i+1} + 2\Delta yx_i =$
  
  $2\Delta x(y_{i+1} - y_i) - 2\Delta y(x_{i+1} - x_i) =$
  
  $2\Delta x(y_{i+1} - y_i) - 2\Delta y(x_i + 1 - x_i) =$
  
  $2\Delta x(y_{i+1} - y_i) - 2\Delta y$

- $p_{i+1} = p_i + 2\Delta x(y_{i+1} - y_i) - 2\Delta y$
Handling the $y_{i+1} - y_i$ term

- $y_{i+1} - y_i = 0$ or $1$, depending on whether we move up a scan line or not

- If $p_i$ is positive (i.e., we don’t move up, so $y_{i+1} = y_i$ and $y_{i+1} - y_i = 0$), then $p_{i+1} = p_i - 2\Delta y$

- If $p_i$ is negative (i.e., we move up, so $y_{i+1} = y_i + 1$ and $y_{i+1} - y_i = 1$), then $p_{i+1} = p_i + 2\Delta x - 2\Delta y$
Only integer operations

- Note that $\Delta x$ and $\Delta y$ are both integers, are constant for a line segment, and can be computed before entering the loop.
- The drawing loop now only contains integer addition operations.
$p_1$

Need a value for $p_1$ to get started with

Use $(x_1, y_1)$ in the equation for $p_i$

\[
p_1 = 2\Delta xy_1 - 2\Delta yx_1 - 2\Delta y - \Delta x(2b-1)
= 2\Delta xy_1 - 2\Delta yx_1 - 2\Delta y - \Delta x(2y_1 - 2(\Delta y/\Delta x)x_1 - 1)
= 2\Delta xy_1 - 2\Delta yx_1 - 2\Delta y - 2\Delta xy_1 + 2\Delta yx_1 + \Delta x
= \Delta x - 2\Delta y
\]
Other cases

- How would you handle each of the following cases with Bresenham’s algorithm?
  - $-1 \leq \text{slope} \leq 0$
  - $\text{slope} > 1$
  - $\text{slope} < -1$
Circles

Cartesian coordinates

- \((x - x_c)^2 + (y - y_c)^2 = r^2\)
- \(y = y_c \pm \sqrt{r^2 - (x_c - x)^2}\)

Polar coordinates

- \(x = x_c + r \cos \theta\)
- \(y = y_c + r \sin \theta\)
A Bresenham algorithm for circles

- Assume center of circle is (0, 0)
- Recognize symmetry of circle (only need to consider 1/8 of circle)
- Choose region to consider as (0, r) to (r√2, r√2) along circumference
  - slope of circle is between 0 and -1
  - can do work stepping along x-axis
The error terms for circles

- Suppose \((x_i, y_i)\) is plotted
- Next point is either \((x_i+1, y_i)\) or \((x_i+1, y_i-1)\)
- Compute two error terms, using \(y^2\) instead of \(y\)
  - let \(e_1\) be error for point above circle
  - let \(e_2\) be error for point below circle
**p_i for circles**

- Define $p_i$ as $e_1 - e_2$
- If $p_i < 0$ then $(x_{i+1}, y_i)$ is plotted, otherwise $(x_{i+1}, y_{i-1})$ is plotted
- $p_i = 2(x_{i+1})^2 + y_i^2 + (y_i-1)^2 - 2r^2$
- $p_{i+1} = p_i + 4x_i + 6 + 2(y_{i+1}^2 - y_i^2) - 2(y_{i+1} - y_i)$
- If $p_i < 0$ then $p_{i+1} = p_i + 4x_i + 6$ else $p_{i+1} = p_i + 4(x_i - y_i) + 10$
- $p_1 = 3 - 2r$ (at $(0, r)$)
Compositing

- Blending together of values from different objects at the same pixel
- This is the use of the alpha channel (the A in RGBA)
- Like R, G, and B, A is defined over [0.0, 1.0]
- A = 1.0 implies a completely opaque surface, while A = 0.0 implies a completely transparent surface
How to do compositing

- When an object is rendered into the frame buffer, both the object’s pixel and the current pixel value in the frame buffer are scaled by appropriate constants and the products added together.

- There are 4 scale factors for the object’s pixel, 1 each for RGBA, and 4 for the buffer’s pixel.
Blending in OpenGL

- `glBlendFunc (sfactor, dfactor)`
- The `sfactor` tells how to scale the object being processed
- The `dfactor` tells how to scale what’s currently in the buffer
- The table in the OpenGL Reference Manual for `glBlendFunc` gives the symbolic constants you can use for `sfactor` and `dfactor`
- Note that you need to `glEnable (GL_BLEND);`
Example program

- alpha.c is an example using blending
- Can you derive the colors that should be displayed in each quadrant?
Scan line approach to filling a polygon

- For each scan line in range of polygon:
  - find intersections of scan line with all edges of polygon
  - sort intersections in increasing x
  - fill in all pixels between pairs of intersections that lie interior to polygon, using the odd-parity rule to determine that a point is inside:
    - parity is initially even
    - each intersection encountered inverts the parity
    - draw when parity is odd
Edge coherence

- To find intersections of each scan line with the edges of the polygon we take advantage of **edge coherence** - many of the edges that intersect scan line $i$ will also intersect scan line $i+1$

- Will also take advantage of the incremental changes in $x$ and $y$ so we don’t have to solve equations to find intersections
Global edge table

- Contains each edge of the polygon, sorted by their smaller y coordinate
- Use buckets for each scan line, and store edges in increasing x coordinate order
- Each edge contains $y_{max}$, $x_{min}$, dy, dx, and x-increment in going from one scan line to the next
Active edge table

- For each scan line we construct the active edge table, where the set of edges a scan line intersects is kept sorted on the $x$ intersection values.
- When we move to the next scan line, the active edge table is updated.
The complete algorithm

- Set y to the smallest y coordinate that has an entry in the ET
- Initialize the AET to be empty
- Repeat until AET and ET are empty:
  - Move from ET bucket y to the AET those edges whose $y_{\text{min}} = y$ (entering edges), then sort the AET on x (made easier because the ET is presorted)
  - Fill in desired pixel values on scan line $y$ by using pairs of $x$ coordinates from AET
  - Remove from AET those entries for which $y = y_{\text{max}}$ (edges not involved in next scan line)
  - Increment $y$ by 1 (go to next scan line)
  - For each edge remaining in AET, update $x$ for the new y
Patterns

- Define a pattern in a small matrix
- Fix a reference point (usually lower left corner)
- As you fill an area, pick the corresponding pixel from the pattern matrix
- The pattern usually repeats, so use a mod operation to get the repeating effect