Outcomes

- Understand the basic rasterization algorithms including
  - DDA Line drawing and Circle drawing
  - Bresenham Line and Circle drawing
  - Sutherland-Hodgeman Polygon Clipper
  - Cohen-Sutherland Line Clipper.

Raster Algorithms

• Goals for Line Drawing Algorithms

- We assume that we want our line drawing algorithm to be able to draw a line between two user-specified pixels on the screen, (x_0, y_0) and (x_1, y_1)

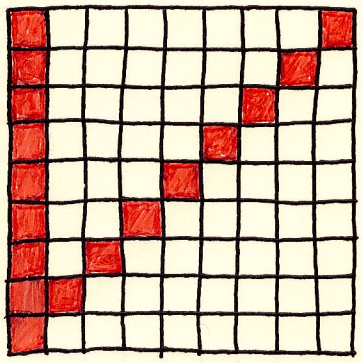
- Note that we aren't specifying the line between two 3D points in the image, but are assuming that any line has already been projected into screen coordinates and clipped to the screen, so that (x_0, y_0) and (x_1, y_1) are integer pairs that are pixel addresses that are presumably projections from 3D floating point points.

- There are a number of goals that we would like for any line drawing algorithm, not all of which can be satisfied at the same time. Some of these goals are:

1. The line displayed should include the end pixels (x_0, y_0) and (x_1, y_1)
2. The line intensity should be constant, and shouldn't depend on the angle of the line
3. The algorithm should be very fast, and easy to implement in a graphics accelerator
4. The line generated should look straight, without jaggies
5. The line shouldn't have any gaps which would let color escape in a flood fill operation, it should be 8-connected.

- Unfortunately these goals contradict each other. The only way, for example, to fully satisfy 2 and 4 is to use anti-aliased lines, and that process is computationally very expensive.

- For example, consider Goal 2, and look at the figure below, where I've drawn two lines, one from (1,1) to (1,9) and the other from (1,1) to (9,9), with both drawn in the obvious way.



- It should be obvious that the line at 45° is much less intense than the vertical line, because its length is $\sqrt{2}$ longer, but it contains the same number of pixels.

- However, any fast line drawing algorithm will have this problem, and graphics systems just have to cope.

- Back to the goals, we can easily meet goals 1 and 3, and goal 2 we can approximate by saying the intensity between lines will never exceed a factor of $\sqrt{2}$

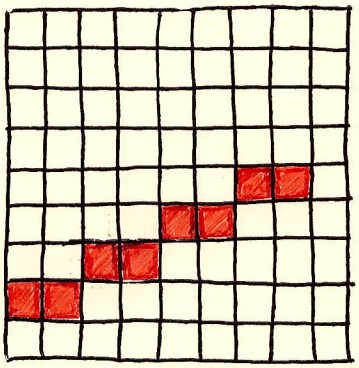
- Jaggies will exist in most lines, but as screen resolutions continue to improve they will continue to become less obvious.
- Before getting into Bresenham's algorithm we'll first look at the more obvious algorithm called DDA, which adds in the slope and rounds to the nearest y for each new pixel displayed. Bresenham will produce the exact same output as DDA, but dramatically faster since it only uses integer addition without the float and round that occurs in DDA.
- For both the DDA and Bresenham algorithms we will assume that the line is being drawn from left to right (i.e., $x_0 < x_1$), and that the line points increase gently, and so its slope satisfies $0 < \text{slope} < 1$. Then in a subsequent section we will show how to generalize the results to any line.
- We will ignore lines with slopes of 0 or ∞, since they can easily and efficiently be handled as special cases.

DDA and Bresenham's Line Rasterizers

DDA Line Drawing Algorithm

- The DDA (Digital Differential Analyzer) algorithm is the obvious way to draw a line. Since we are looking at lines whose increase in y is less than the increase in x, the simplest solution is to start with a pixel at (x_0, y_0) and then loop increasing x by one each time and y by the slope, and then rounding y to find the pixel to be displayed.

- For example, say we want to draw the line from $(1,2)$ to $(8,5)$, then we draw the line as shown below:



- In the DDA line drawing algorithm there is a pixel drawn for each x between the starting and ending values, and the y value closest to the line is selected for the second coordinate.

- The calculations that will be performed are shown in the table below:

x	y	round(y)
1	2	2
2	$2\frac{3}{7}$	2
3	$2\frac{6}{7}$	3
4	$3\frac{2}{7}$	3
5	$3\frac{5}{7}$	4
6	$4\frac{1}{7}$	4
7	$4\frac{4}{7}$	5
8	5	5

- The algorithm being performed here is:

```
procedure DDALine(in out x0,y0,x1,y1, in colorval color)
assert: (x0 < x1) and ((y1-y0) < (x1-x0))
   declare: int x; float y; slope;

   slope = (y1-y0)/(x1-x0);
   x ← x0; y ← y0;
   printPixel(x, y, color);

   while x < x1 do
      x++;
      y += slope;
      printPixel(x, round(y), color);
   end while
end procedure
```

- Where printPixel will display the pixel in the desired color.
- The expensive parts of the algorithm are y+=slope which is a float add and round(y) which converts from float to integer
- The Bresenham algorithm instead only uses integer additions and comparisons.

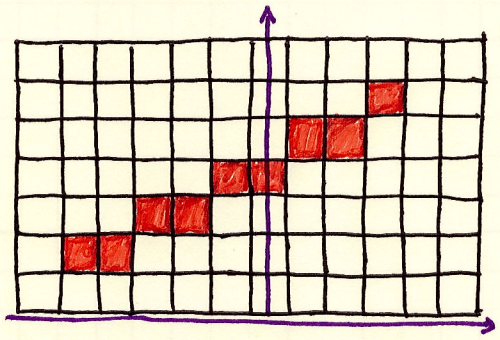
Bresenham Line Algorithm for $x_0 < x_1$ and $0 < slope < 1$

- Bresenham's algorithm displays exactly the same pixels as the DDA algorithm, given the same end points. The difference is that it does it more efficiently.
- The DDA algorithm has to maintain y as a float, and in each loop does a float add and a float to integer round.
- We also have a single float divide to compute the slope. By comparison, Bresenham's algorithm only uses integer variables, and only uses integer compares and additions
- This not only makes it much faster, but also makes it much easier to put into hardware.
- The following is the Bresenham Algorithm

```
procedure BresenhamLine(in int x0,y0,x1,y1, colorval color)
assert: (x0 < x1) and ((y1 - y0) < (x1 - x0))

  declare: int x, y, dx, dy, d, incrE, incrNE

  dx = x1 - x0; dy = y1 - y0;
  incrE = 2 * dy;
  incrNE = 2 * (dy - dx);
  d = 2 * dy - dx;
  x <- x0; y <- y0
  printPixel(x, y, color)

  while x < x1 do
     x++;
     if d < 0 then
        d += incrE; //go east
     else
        y++;
        d += incrNE; //go northeast
     end if
     printPixel(x, y, color)
  end while
end procedure
```

- The most important thing to note about this algorithm is that everything is an integer, so it avoids floating point calculations.
- Also the only operations are integer additions and comparisons, since multiplication by 2 will be implemented as a single addition.
- This is why Bresenham is much faster than DDA, and also why it is easy to implement in graphics hardware.
- Example: Use this algorithm to generate the line from (-5,2) to (4,6)
  - First, we need the initial values of the variables Δx, Δy, d, incrE and incrNE, which will be 9, 4, -1, 8, and -10, respectively.
  - We then initialize x and y to the starting point, -5 and 2, display this pixel in the required color, and enter the while loop.
  - This increments x by 1 each time through the loop, and so it will execute nine times, with x starting at -4 and ending at 4.
  - Inside the loop one of two things will happen based on the value of d.
    - If d is negative, it draws the pixel to the right (east) of the previous pixel, since it doesn't change the value of y, and adds incrE to the current value of d.
    - If d isn't negative it moves northeast by also adding 1 to y, and adds incrNE to d.
    - In either case the new pixel is printed at (x, y)
  - A table and graph of the values is shown below:

x	y	d
-5	2	-1
-4	2	7
-3	3	-3
-2	3	5
-1	4	-5
0	4	3
1	5	-7
2	5	1
3	6	-4
4	6	4



- If you compare this to the DDA algorithm you will see that both algs will generate the exact same pixels.

- The difference is that Bresenham has only used integer arithmetic because of its use of the decision variable, d, which is used to determine whether the next pixel should be east or northeast of the previous pixel.

- Note that because of the constraints on the slope these are the only possible positions for the next pixel.

◦ Bresenham Algorithm for General Lines

- We have assumed that $(x_0 < x_1)$ and $((y_1 - y_0) < (x_1 - x_0))$ i.e., that the line is going left to right and that the slope satisfies $0 < slope < 1$

- Clearly it is trivial to draw vertical and horizontal lines without calling on Bresenham, which means that we can assume that $x_0 \neq x_1$ and $y_0 \neq y_1$

- We need to handle three other cases:

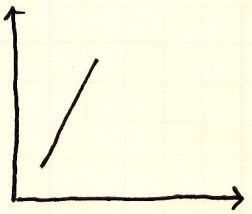
1. $x_0 > x_1$
2. $slope > 1$
3. $slope < 1$

- We'll discuss each of these cases independently

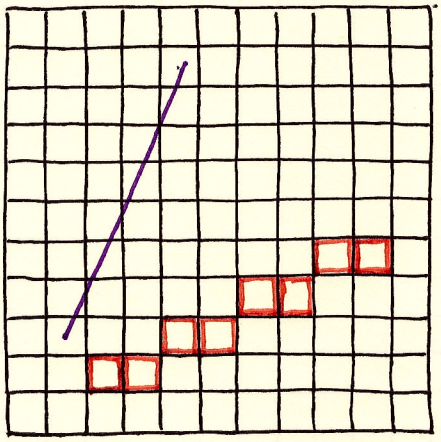
1. $x_0 > x_1$: This is trivial to handle. If the line should be drawn from right to left instead of left to right, we just draw it backwards and everything will be all right. I.e., we just exchange (x_0, y_0) and (x_1, y_1)

   E.g., if the algorithm is asked to draw a line from (3,5) to (1,4), we instead draw the line from (1,4) to (3,5)

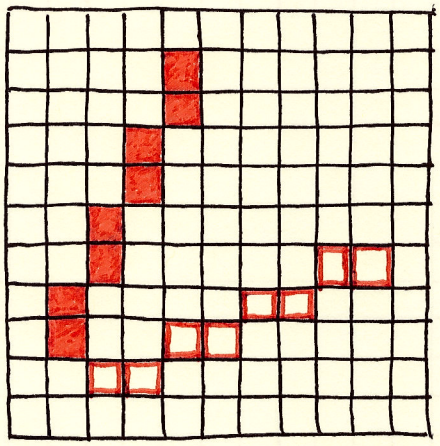
2. $slope > 1$: Say that we are drawing a line from (2,3) to (5,10) as shown in the following figure



We can exchange x and y values and use Bresenham to calculate the dashed line from (3,2) to (10,5), which will give the pixel locations shown below

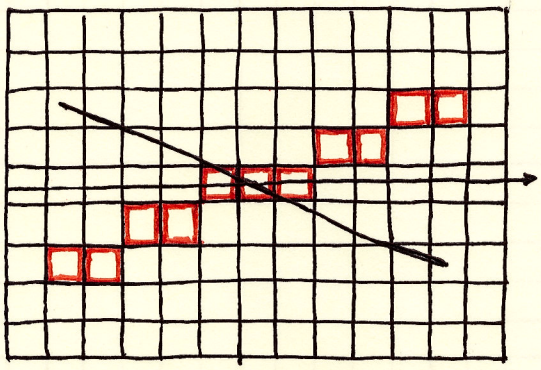


Now we can exchange their x and y values to get the pixels displayed on the line

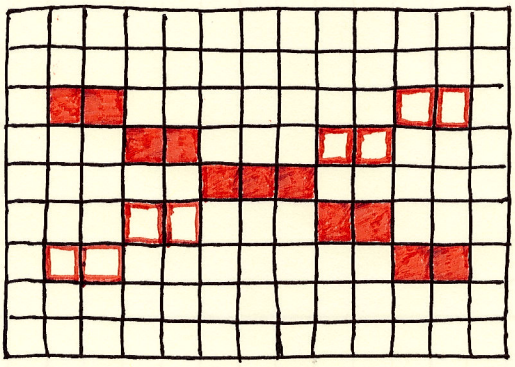


So to summarize, we wanted to draw a line from (2,3) to (5,10). Since the slope was greater than 1 we exchanged the x and y values and calculate the pixels from (3,2) to (10,5). These are (3,2), (4,2), (5,3), (6,3), (7,4), (8,4), (9,5) and (10,5). Then we exchanged their x and y values to get the pixels that we display: (2,3), (2,4), (3,5), (3,6), (4,7), (4,8), (5,9), (5,10). In effect, we have reflected the line that we want around the diagonal $y=x$, which turned a slope of greater than 1 into a slope of less than 1

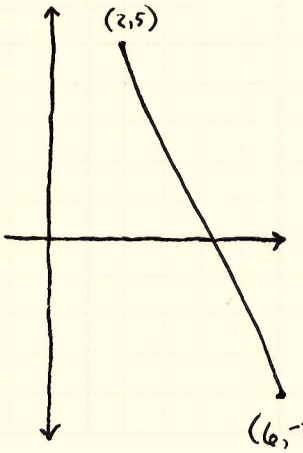
3. slope < 0: If the slope is less than zero we can reflect the line around the x-axis to get a line whose slope is greater than zero, generate the points there, and then reflect back around the x-axis to get the displayed pixels. What this means is that instead of using Bresenham from (x_0, y_0) to (x_1, y_1), we will use the algorithm on the line from $(x_0, -y_0)$ to $(x_1, -y_1)$. Using the same notation used before and working the line from (2,2) to (12,-2), we compute the pixels for (2,-2) to (12,2) using Bresenham, as shown.



given the values (2,-2), (3,-2), (4,-1), (5,-1), (6,0), (7,0), (8,0), (9,1), (10,1), (11,2) and (12,2). Changing the signs of the y-values gives the displayed line with pixels (2,2), (3,2), (4,2), (5,1), (6,0), (7,0), (8,0), (9,-1), (10,-1), (11,-2) and (12,-2)

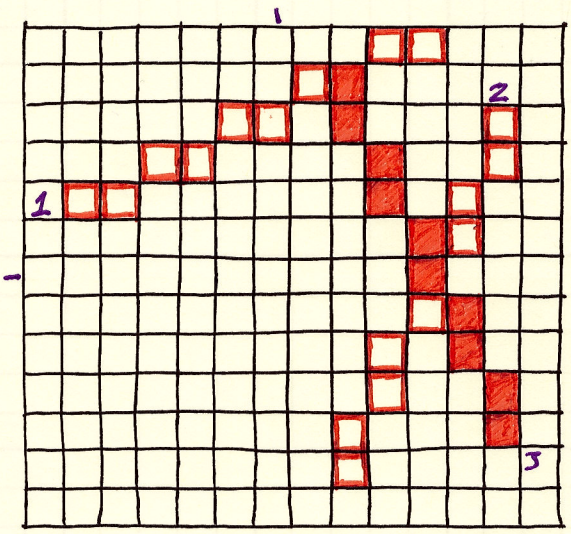


- Example: Use Bresenham to generate the line from (6,-4) to (2,5)



Everything is wrong here. The line goes right to left, but we can fix that by using the line (2,5) to (6,-4). The slope is negative, and so we mirror around the x-axis and use the line from (2,-5) to (6,4). The slope of this line is greater than 1, so we mirror around $y=x$, and we use Bresenham for the line from (-5,2) to (4,6). This satisfies everything, and so we can finally run the algorithm.

The three lines are shown in the next figure. First we will use Bresenham to give us the pixel coordinates on line 1, then exchange x and y to get the pixel coordinates on the line labeled 2, and then switch the signs of the y values to get the line labeled 3, which is the line that we wanted. In effect the first operation is a mirror back around the line $y=x$, and the second is a mirror back around the x-axis.



In our first Bresenham algorithm example we showed that for (-5,2) to (4,6) Bresenham generates 10 pixels (-5,2), (-4,2), (-3,3), (-2,3), (-1,4), (0,4), (1,5), (2,5), (3,6), and (4,6)

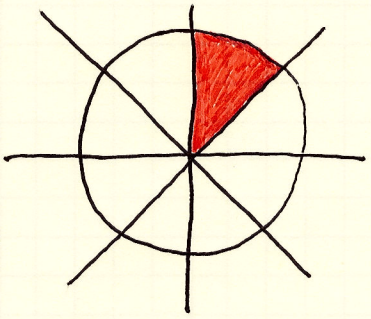
Mirroring back around the line $X=Y$, by exchanging x and y values, gives the pixels on the intermediate line (2,-5), (2,-4), (3,-3), (4,-1), (4,0), (5,1), (5,2), (6,3), (6,4).

Switching the signs of the y values completes the whole process and gives us the pixels (2,5), (2,4), (3,3), (3,2), (4,1), (4,0), (5,-1), (5,-2), (6,-3) and (6,-4)

° Generating Circles

- We will assume that the equation of the circle is $x^2+y^2=R^2$. I.e., we have a circle with its center at the origin and radius R. Later we will generalize this to circles with equation $(x-cx)^2+(y-cy)^2=R^2$, which is a circle with radius R and center (cx, cy), by generating the circle centered at the origin and then translating it to the correct center.

- An important efficiency improvement for both DDA and Bresenham is to only calculate pixel coordinates in the primary octant of the circle and then use symmetry to calculate the points in the other seven octants. Since the circle is centered at the origin, if we generate a point (x,y), then the other seven points are combinations of positive and negative x and y values can be drawn without further computations



- So it is sufficient to just generate about $\frac{1}{8}$ of the points on the circle, and then output seven additional points for each one.

- DDA Algorithm for Circles Centered at the Origin

  - The advantages of using Bresenham are even greater for drawing circles (and other conics) than they are for drawing lines.

  - The DDA algorithm for circles, below, has to use the square roots to compute y values for different x's, whereas Bresenham once again only uses integer addition

  - First, however, we'll look at the DDA algorithm to ensure that it generates the same points as Bresenham does

  - The points in the primary octant start at $(0, R)$ above the origin, and then increase x by 1 as long as $x \leq y$ (the diagonal at the end of the primary octant)

  - The algorithm is:

```
procedure DDACircle (in int R, in colorval color)
  declare: int x; float y;
  x ← 0; y ← R;
  print8Pixels(x, y, color);
  while x < y do
    x++;
    y = √(R*R - x*x);
    print8Pixels(x, round(y), color);
  end while
end procedure
```

  - I.e., each time we are solving the circle equation $y^2 = R^2 - x^2$, taking the square root and printing the x and the closest y, plus seven symmetric points. The table below gives the (x,y) values

x	$y = \sqrt{100 - x \cdot x}$	round(y)
0	$\sqrt{100} = 10.00$	10
1	$\sqrt{99} = 9.95$	10
2	$\sqrt{96} = 9.80$	10
3	$\sqrt{91} = 9.54$	10
4	$\sqrt{84} = 9.17$	9
5	$\sqrt{75} = 8.66$	9
6	$\sqrt{64} = 8.00$	8
7	$\sqrt{51} = 7.14$	7

- Bresenham Algorithm for Circles Centered at the Origin

- While it is suprising that it is possible to draw lines with only integer additions using a decision variable, it is even more suprising that it is possible to do the same for circles
- However, Bresenham's algorithm accomplishes this, through the use of another decision variable, d.
- Obviously if it is possible to draw a circle without the square root and round operators in the main loop then we can significantly improve the speed of the algorithm.

```
procedure BresenhamCircle (in int R, in colorval color)
   declare: int x, y, d

   d = 1-R;
   x ← 0; y ← R;
   print8Pixels (x, y, color)

   while x < y do
      x++;
      if d < 0 then
         d += 2 * x + 1; // go east
      else
         y--;
         d += 2 * (x-y) + 1; // go southeast
      end if
      print8Pixels (x, y, color) if x ≤ y;
   end while
end procedure
```

- E.g., for R=10 the table below shows that it generates the same points as DDA

x	y	d
0	10	-9
1	10	-6
2	10	-1
3	10	6
4	9	-3
5	9	8
6	8	5
7	7	6

- Note that, as promised, Bresenham's circle algorithm uses no floats, and only integer additions and comparisons

• Bresenham Algorithm for General Circles

- For both DDA and Bresenham we assumed that the circle was centered at the origin. In general, a circle will have equation

$$(x-c_x)^2+(y-c_y)^2=R^2$$

which is a circle with radius R centered at point (c_x, c_y)

- To generate all of the pixels for this circle we first generate all of the points (including symmetries) for the circle, radius R, centered at the origin, and then add (c_x, c_y) to each point to get displayed pixels. E.g., for the circle

$$(x-3)^2+(y+1)^2=100$$

- We first generate the primary octant points we generated in the last section to get the 64 points in the first table, then add (3,-1) to each point to get the pixel shown in the second table.

x,y	y,x	y,-x	x,-y	-x,-y	-y,-x	-y,x	-x,y
0,10	10,0	10,0	0,-10	0,-10	-10,0	-10,0	0,10
1,10	10,1	10,-1	1,-10	-1,-10	-10,-1	-10,1	-1,10
2,10	10,2	10,-2	2,-10	-2,-10	-10,-2	-10,2	-2,10
3,10	10,3	10,-3	3,-10	-3,-10	-10,-3	-10,3	-3,10
4,9	9,4	9,-4	4,-9	-4,-9	-9,-4	-9,4	-4,9
5,9	9,5	9,-5	5,-9	-5,-9	-9,-5	-9,5	-5,9
6,8	8,6	8,-6	6,-8	-6,-8	-8,-6	-8,6	-6,8
7,7	7,7	7,-7	7,-7	-7,-7	-7,-7	-7,7	-7,7

x,y	y,x	y,-x	x,-y	-x,-y	-y,-x	-y,x	-x,y
3,9	9,3	9,-3	3,-9	-3,-9	-9,-3	-9,3	-3,9
4,9	9,4	9,-4	4,-9	-4,-9	-9,-4	-9,4	-4,9
5,9	9,5	9,-5	5,-9	-5,-9	-9,-5	-9,5	-5,9
6,9	9,6	9,-6	6,-9	-6,-9	-9,-6	-9,6	-6,9
7,8	8,7	8,-7	7,-8	-7,-8	-8,-7	-8,7	-7,8
8,8	8,8	8,-8	8,-8	-8,-8	-8,-8	-8,8	-8,8
9,7	7,9	7,-9	9,-7	-9,-7	-7,-9	-7,9	-9,7
10,6	6,10	6,-10	10,-6	-10,-6	-6,-10	-6,10	-10,6

- Note, that the order matters. We must first generate all of the symmetries from the primary octant and then shift the points to the correct center.

- If we first shift the primary octant points to the correct center and then generate the symmetries we will not get the correct points.

- Also note that some points (on the axes and the diagonals) are generated twice. It is easy to define the function plot8Pixels so that this is avoided (which would be important if we were in xor write mode, where the second pixel displayed, would delete the first).

- Other Conics

  - Bresenham can be modified to work with other conic curves.

  - One main difference, however, is that we cannot no longer use the 8-way symmetry of the circle, and so, for example, with ellipses we must generate a quadrant in two parts, working from the axes towards each other, and then use 4-way symmetry to get the ellipse

- Anti-Aliasing Lines

  - If it weren't for its computational cost, which is prohibitive in most applications anti-aliasing would solve all line drawing problems.

  - Line drawing is discussed here, but the techniques to circles and any other edge-based figures

  - There are a number of anti-aliasing techniques. Conceptually the easiest is to consider the effect of a rectangle in the line color going from the first pixel to the last.

  - If a pixel is, say, 60% under the line and 40% not under the line, it will be assigned the color that is 60% the color that is 60% line color and 40% the normal color for that pixel

  - E.g., say that a pixel is the color $.85\,cp + .15\,cl$, where cp is the normal pixel color and cl is the line color, would have, without the line drawn, the normalized RGB color $(.4, .2, .6)$, and that the line color is a medium dark gray of $(.4, .4, .4)$, then the color selected for this pixel will be $(.4, .23, .57)$ because

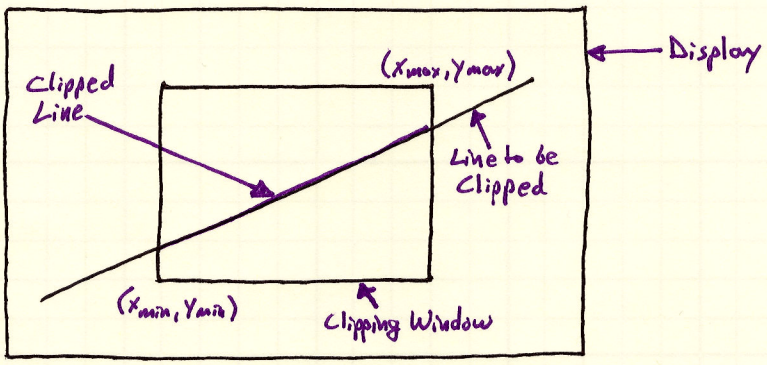
$$.4 \times .85 + .4 \times .15 = .4$$

$$.2 \times .85 + .4 \times .15 = .23$$

$$.6 \times .85 + .4 \times .15 = .57$$

Clipping - Points, Lines, and Polygons in Aligned Rectangular Windows

- Things that are/aren't covered here
  - Covered
    - Clipping relative to aligned, rectangular windows.
      - The window can be a window within a computer display, or it can be the entire display.
      - Aligned means that the edges of the window will have constant x (vertical) or y (horizontal) screen coordinates
      - This means that we can fully define the window with two screen coordinates, (x_{min}, y_{min}) and (x_{max}, y_{max})
      - The typical situation for a line clip is:



    - Three Type of Clipping
      - Point Clipping
      - Line Clipping
      - Polygon Clipping
  - Not Covered
    - Character Clipping
  - Historically
    - Clipping was done by the user and not the graphics system
    - If you didn't clip you'd get overflow wrapping

◦ Point Clipping

- To clip a point (x, y) against a window is trivial, since there isn't a better way to do it than just performing the test:

$$x_{min} \leq x \leq x_{max} \text{ and } y_{min} \leq y \leq y_{max}$$

- Only displaying the point if the test is true

◦ Some Basic Line Segment Math

- Although most people are trained to use the slope-intercept form of a line of a line when dealing with lines, which is

$$y = mx + b$$

- This is rarely the best approach in graphics
  - ◦ The slope-intercept form describes an infinite line
  - ◦ In graphics we have a finite line segment between two points (x_0, y_0) and (x_1, y_1)

- Most of the time it is better to use the parametric form of the line, where both x and y are defined in terms of a parameter t, e.g., the line above is described using two equations:

$$x(t) = x_0 + (x_1 - x_0) * t$$

$$y(t) = y_0 + (y_1 - y_0) * t$$

where

$$0 \leq t \leq 1$$

- Alternatively we can just say

$$P(t) = P_0 + (P_1 - P_0) * t \quad \text{where} \quad 0 \leq t \leq 1$$

- Note that when $t=0$, $P = P_0$, and when $t=1$, $P = P_1$. Also, if we look at the slope of the line

$$\frac{dy}{dx} = \frac{dy/dt}{dx/dt} = \frac{y_1 - y_0}{x_1 - x_0}$$

it is constant, so this is a straight line segment between P_0 and P_1

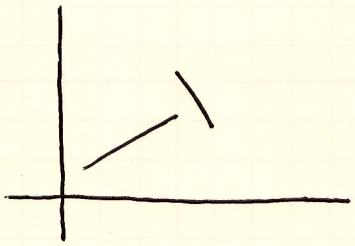
- To use this form, say we want to know whether the line segment from (1,1) to (3,2) intersects the line segment from (3,3) to (4,2). The line equations (being careful to use different parameters names for each line) are:

$$x(t) = 1 + 2t \qquad x(s) = 3 + s$$
$$y(t) = 1 + t \qquad y(s) = 3 - s$$

- The segments intersect when they have the same x and y values, which gives:

$$1 + 2t = 3 + s$$
$$1 + t = 3 - s$$

- These simultaneous equations have the solution $t = \frac{4}{3}$ and $s = \frac{2}{3}$, and so the line segments don't intersect since both parameters aren't between 0 and 1. Looking at a graph of this, these values make sense

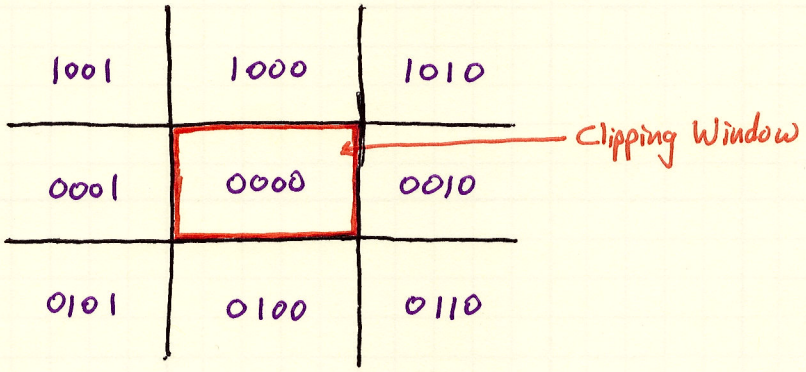


- The major advantage of this approach over the traditional slope-intercept approach is that it includes the fact that these are line segments, not lines.

- With slope-intercept we would have found that the lines intersect since they aren't parallel, but would have had to check to see where the intersection was relative to the segments

Cohen-Sutherland Line Clipper

- Most popular line clipping algorithm
- The underlying idea is to divide 2D space up into nine regions, shaped like an infinite tic-tac-toe board, and assign four-bit codes to each region, where the center is the clipping window, as shown below:



- If you look carefully at the pattern of the four bits $b_3 b_2 b_1 b_0$, you will see a pattern
- If point (x, y) satisfies $x < x_{min}$, then $b_0 = 1$ (where b_0 is the rightmost bit) otherwise $b_0 = 0$, and the same edge rules apply to the other bits, i.e.,

$$b_0 = x < x_{min}$$
$$b_1 = x > x_{max}$$
$$b_2 = y < y_{min}$$
$$b_3 = y > y_{max}$$

- There are times when including the edges in the clip makes sense, and then the comparison operators change with $\leq$ replacing $<$ and $\geq$ replacing $>$
- Given any line segment from $P_0 = (x_0, y_0)$ to $P_1 = (x_1, y_1)$, the algorithm first assigns these four-bit codes to P_0 and P_1 based on their locations in this grid. There are 3 main steps to the algorithm:

1.) If code (P_0) = code (P_1) = 0000, then both endpoints are in the window, so display the line without clipping and return

2.) If (code (P_0) and code $(P_1)) \neq 0000$, where and is the bitwise and, then this line is completely outside the window, so reject it and return

3.) Otherwise we have two choices on how to handle the current line, which might or might not cross the window. The two choices, are to either divide it into equal pieces and run the algorithm again, or to edge intersections with the appropriate edges (they can be found through (code(P_0) or code(P_1))) which gives the clipped line.

- So the goal of this algorithm is to rapidly draw lines that are fully in the window or can easily be detected as being fully outside the window and then handle the other cases with more work in step 3.

- Step 1 is obvious

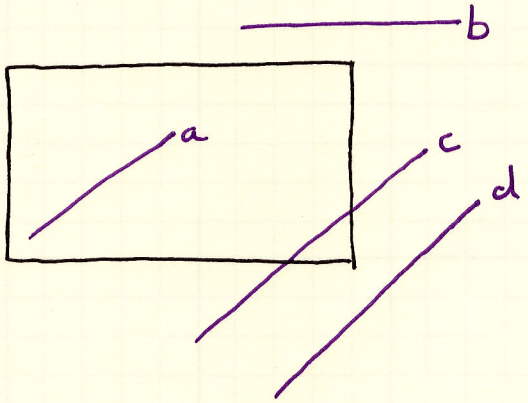
- Step 2 uses the fact that if the and of the two 4-bit codes is not 0000, then somewhere there must be a 1 in the same bit locations in both codes.

  - Say, for example, that it is bit b_0 that is one in both. This means that both points are to the right side, outside the window, so it cannot intersect the window.

  - Obviously similar arguments apply to the other three bits

- Consider the diagram below, the codes on the endpoints of line a are both 0000 and so the line will be drawn in step 1. The endpoint codes of line b are 1000 and 1010, so since their and is 1000, the line will be rejected in step 2.

- The only difficulty occurs with lines like c and d, which share the same endpoint codes (0100 and 0010), but one needs clipping and the other can be rejected.



- Step 3 has to handle cases like this. There are two approaches:

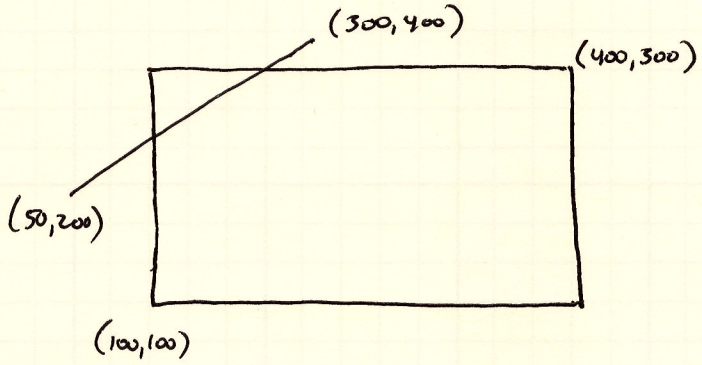
  - The simplest in terms of hardware accelerators is to just divide the line in two, and run the algorithm again on both halves. In some cases this will immediately terminate, but in others it will lead to repeated calls. The usual rule is to terminate this after 11 calls, since by then you will be at pixel resolution of a screen with approximately 1K x 1K resolution and a longest possible diagonal line

  - The second approach is to compute intersections with the edges using the parametric form of the line segment, which will immediately determine if the line misses the window or, if it crosses the window, will give the piece to display

- Using bitwise or on the codes will determine which edge intersections need to be considered.

Ex: for c and d, the endpoint codes were 0100 and 0010. So we need to check intersections against the edges related to bits 1 and 2 of the code

Ex: Consider the line below:



The line from (50,200) to (300,400) has parametric equations

$$x(t) = 50 + 250t$$

$$y(t) = 200 + 200t$$

$$0 \le t \le 1$$

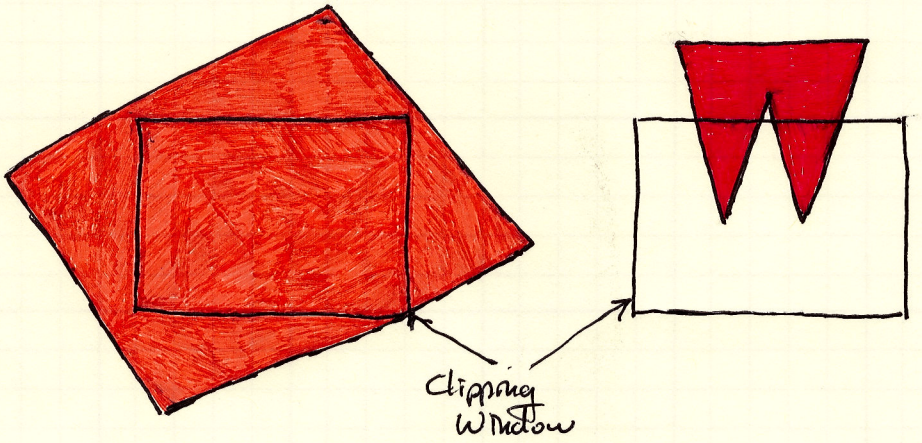
The endpoint codes, 0001 and 1000, tell us that we need to compute intersections with the left and top edges, which have equations $x=100$ and $y=100$

Substituting $x=100$ into the first equation gives $t=0.2$, and using this in the second equation gives $y=240$. So the first intersection point is at (100, 240)

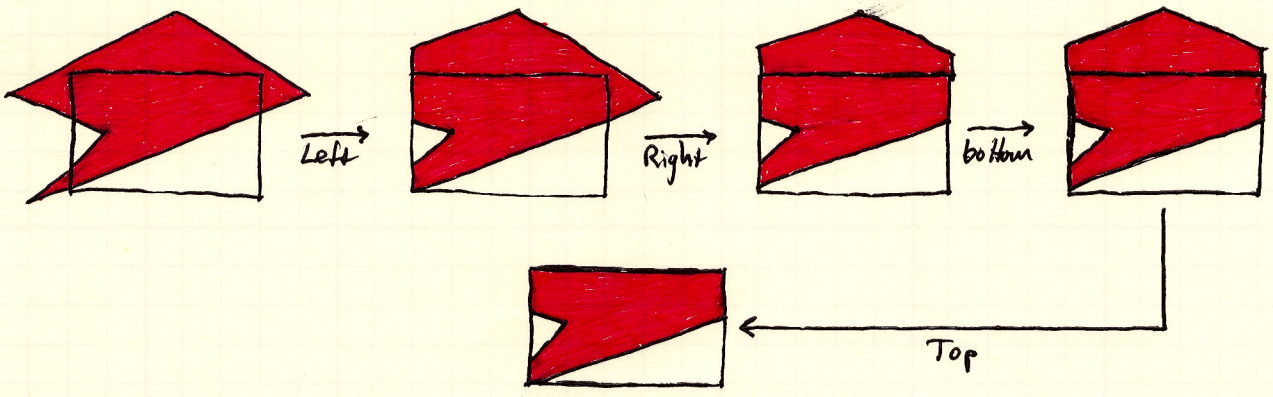
Similarly, intersecting with $y=300$ gives $t=0.5$ and so the intersection point is at (175, 300). The clipped line is between (100, 240) and (175, 300)

Sutherland-Hodgman Polygon Clipper

- It might appear that we can solve polygon clipping by just using line clipping on the edges, but unfortunately this doesn't work.
- For example, consider the case where a red polygon completely covers the viewing window. If we clip its edges they will disappear, and the polygon will vanish. There is also a problem if the clipped polygon fragments into multiple small pieces.
- For example, both of the two cases below cause problems:

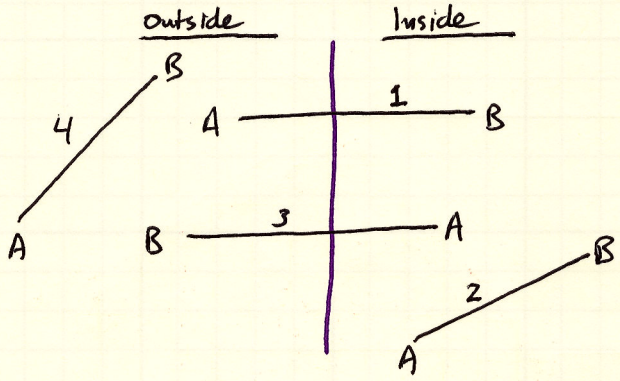


- Sutherland-Hodgeman works by taking the polygon description as a list of vertices, and then building a new polygon description where the part outside one edge is discarded, then repeats with a second edge, then a third, and finally the remaining edge, so that the final polygon is strictly inside the clipping window
- This process is illustrated in the following diagram. Here we have a colored polygon and a clipping window. First we clip against the left window edge, then against the right edge, then against against the bottom edge, and finally against the top edge, leaving the clipped polygon shown.

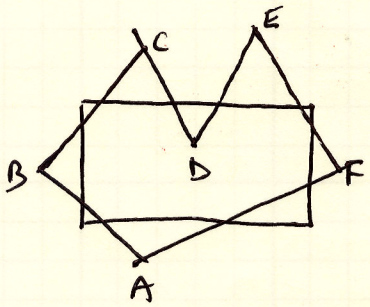


- The goal is to do these edge clipping operations as efficiently as possible. The Sutherland-Hodgeman algorithm does this by looking at the relationship between individual polygon edges, as we go around the polygon, and the current edge is first the left, then the right, then the bottom, and finally the top.

- Obviously any edge order will work, but we will stick with this one for consistency
- So say we have a polygon edge from vertex A to vertex B and we are seeing how it relates to the window side (the in side) and the non-window side (the out side) of the current window edge.
- There are four possibilities, below:

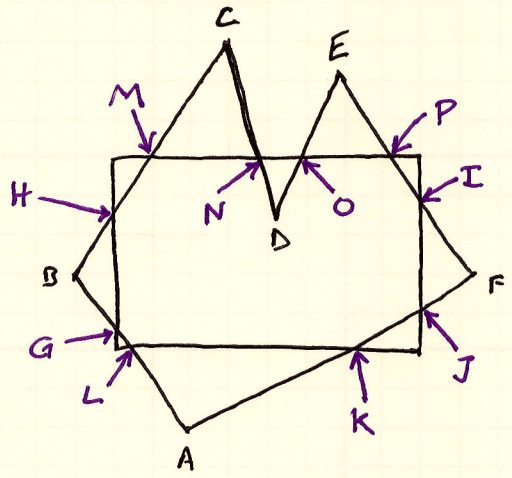


- I.e., case 1 is when the polygon edge is going from outside to inside, case 2 is from inside to inside, case 3 is from inside to outside, and case 4 is from outside to outside.
- The algorithm takes a polygon description as an ordered list of vertices that go around the polygon, and outputs a new polygon list clipped to this list. The output rule that it uses to do this are:
  - Case 1: output I, B, where I is the intersection of the line and edge
  - Case 2: output B
  - Case 3: output I, where I is the intersection of the line and edge
  - Case 4: output nothing
- To see how this works, see the polygon and clipping window below:



- The polygon is described by the vertex list ABCDEF, where there is an assumption that it wraps back from F to A

- Clipping against the left edge first, where the inside is to the right and the outside is to the left, we run the algorithm from A to B, then B to C, ..., and finally F to A
  - A → B is inside to outside, case 3, so output the intersection G shown in the next figure
  - B → C is outside to inside, case 1, so output H (intersection) and C
  - C → D is inside to inside, case 2, so output D
  - Similarly, D → E, E → F, and F → A are all inside relative to the left edge, so will output E, F, and A
- The output list from GHCDEFA, which is our first new polygon, which will be used for the next pass, which is clipped against the right edge.



- Since we are clipping against the right edge, the inside is now to its left and the outside to its right. So G → H, H → C, C → D, and D → E are all inside to inside, and will output H, C, D, and E, respectively
- E → F is in to out, so the output is the intersection, I
- F → A is out to in and outputs J and A, and A → G outputs G. So the new polygon is HCDEIJAG.
- Clipping against the bottom, A is the only point on the outside, and so the output will be CDEIJKLGH
- Finally, clipping against the top, C and E are on the outside, so we get the polygon NDOPIJKLGHC, which is clipped against the window.

- Other Line and Polygon Clipping Systems

  - The Liang-Barsky method is based on the same approach as Cohen-Sutherland, but takes advantage of some efficiencies that can be made using parametric equations

  - The Nicholl-Lee-Nicholl method is quite a bit more complicated than the Cohen-Sutherland based systems and is based on increasing the complexity of the region structure.

  - The Weiler-Atherton is the main competition for Sutherland-Hodgeman for clipping polygons. The algorithm traces around the polygon looking for intersections with the clipping polygon edges and ultimately jumping to reentry points after it leaves the clipping area.

CS 4458 | Lecture 16

For Next Time

- Read Chapter 13
- Review lectures
- Come to class!

Additional Notes