

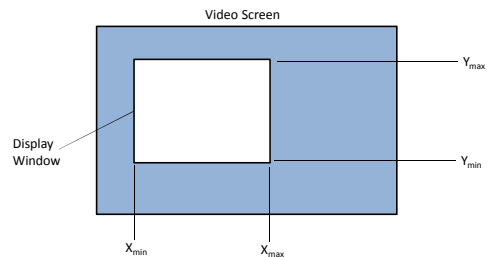
Chapter 3: Graphics Output Primitives

- ❑ Primitives: functions in graphics package that we use to describe picture element
- ❑ Points and straight lines are the simplest primitives
- ❑ Some packages include circles, conic sections, quadric surfaces, spline curves and surfaces, and polygon color area

1

Two-Dimensional World Coordinate Reference Frame in OpenGL

```
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluOrtho2D (xmin, xmax, ymin, ymax);
```



2

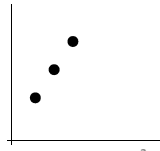
OpenGL Point Functions

- ❑ OpenGL function to state coordinates values for point
 - ❑ `glVertex* ();` where (*) suffix code required
 - ❑ A `glVertex` must be placed between `glBegin` and `glEnd`
- ```
glBegin (GL_POINTS);
glVertex* ();
glEnd ();
```

```
glBegin (GL_POINTS);
glVertex2i (50, 100);
glVertex2i (75, 150);
glVertex2i (100, 200);
glEnd ();

int pnt1 [] = {50, 100};
int pnt2 [] = {75, 150};
int pnt3 [] = {100, 200};

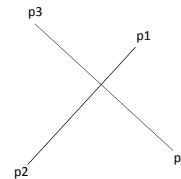
glBegin (GL_POINTS);
glVertex2iv (pnt1);
glVertex2iv (pnt2);
glVertex2iv (pnt3);
glEnd ();
```



3

## OpenGL Line Functions

- ❑ Straight line segments between each successive endpoints
- ```
glBegin (GL_LINES);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glEnd ();
```

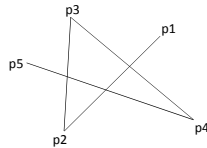


4

OpenGL Line Functions

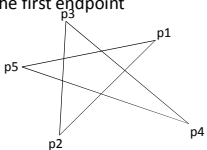
- ❑ A sequence of connected line segments between first and last endpoints

```
glBegin (GL_LINE_STRIP);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glEnd ();
```



- ❑ Last coordinate is connected to the first endpoint

```
glBegin (GL_LINE_LOOP);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glEnd ();
```

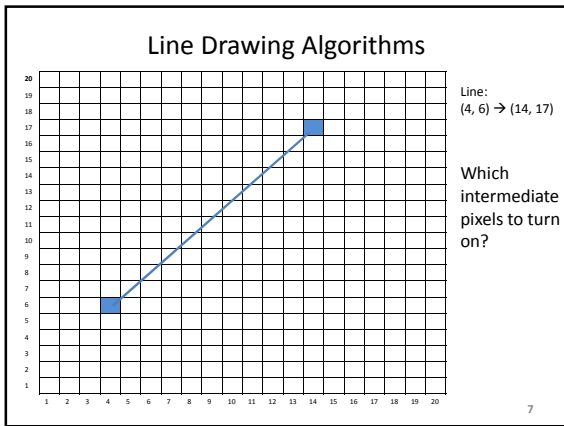


5

Line Drawing Algorithms

- ❑ Programmer specifies (x, y) values of end pixels
- ❑ Need algorithm to figure out which intermediate pixels are on line path
- ❑ Pixel (x, y) values constrained to integer values
- ❑ Actual computed intermediate line values may be floats
- ❑ Rounding may be required. E.g. computed point (10.48, 20.51) rounded to (10, 21)
- ❑ Rounded pixel value is off actual line path (jaggy!!)
- ❑ Sloped lines end up having jaggies
- ❑ Vertical, horizontal lines, no jaggies

6



Line Drawing Algorithms

- Slope-intercept line equation
 - $y = m \cdot x + b$
 - m : slope of the line, b : y intercept
 - Given two end points (x_0, y_0) and (x_{end}, y_{end}) , how to compute m and b ?
- $m = (y_{end} - y_0) / (x_{end} - x_0)$
- $b = y_0 - m \cdot x_0$

Example:
Find the slope of (23, 41) and (125, 96)

$$m = (96 - 41) / (125 - 23) = 55 / 102 = 0.5392$$

8

DDA (Digital Differential Analyzer)

- A line is sampled at unit intervals in one coordinate and the corresponding integer values nearest the line path are determined for the other coordinate
- Case ($m \leq 1$) : sample at unit x intervals ($\Delta x = 1$), computer successive y values as $y_{k+1} = y_k + m$
- Case ($m > 1$) : sample at unit y intervals ($\Delta y = 1$), computer successive x values as $x_{k+1} = x_k + 1 / m$

9

DDA (Digital Differential Analyzer)

$m < 1, y_{k+1} = y_k + m$ $x_1 = x_0 + 1 \quad y_1 = y_0 + m$ $x_2 = x_1 + 1 \quad y_2 = y_1 + m$...	$m > 1, x_{k+1} = x_k + 1 / m$ $y_1 = y_0 + 1 \quad x_1 = x_0 + 1 / m$ $y_2 = y_1 + 1 \quad x_2 = x_1 + 1 / m$...
---	---

10

DDA code

```

inline int round (const float a)
{
    return int (a + 0.5);
}

void linedDA (int x0, int y0, int xEnd, int yEnd)
{
    int dx = xEnd - x0, dy = yEnd - y0, steps;
    float xIncrement, yIncrement, x = x0, y = y0;

    if (fabs (dx) > fabs (dy))
        steps = fabs (dx);
    else
        steps = fabs (dy);

    xIncrement = float (dx) / float (steps);
    yIncrement = float (dy) / float (steps);

    setPixel (round (x), round (y));

    for (int i = 0; i < steps; i++) {
        x += xIncrement;
        y += yIncrement;
        setPixel (round (x), round (y));
    }
}

```

11

Bresenham's Line Algorithm

- Accurate, efficient, and uses only incremental integer calculations
- Sampling at unit intervals, decide which of two possible pixel positions is closer to the line path

12

Bresenham's Line Algorithm

13

Bresenham's Line Algorithm

- Consider positive slope less than 1.0
- Sampling at position $x_{k+1} = x_k + 1$, decide $(x_k + 1, y_k)$ or $(x_k + 1, y_k + 1)$

$$y = m(x_k + 1) + b$$

$$d_{lower} = y - y_k = m(x_k + 1) + b - y_k$$

$$d_{upper} = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$$

Determine which pixel is closer based on the difference

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2b - 1$$

Substitute $m = \Delta y / \Delta x$ so it involves only integer calculations

$$d_{lower} - d_{upper} = 2(\Delta y / \Delta x)(x_k + 1) - 2y_k + 2b - 1$$

$$\Delta x (d_{lower} - d_{upper}) = 2\Delta y (x_k + 1) - \Delta x 2y_k + \Delta x (2b - 1)$$

14

Bresenham's Line Algorithm

P_k is a decision parameter for the k th step

$$P_k = \Delta x (d_{lower} - d_{upper})$$

$$P_k = 2\Delta y \cdot x_k - 2\Delta x \cdot y_k + \boxed{2\Delta y + \Delta x(2b - 1)} \rightarrow C$$

if P_k is negative then pixel y_k is closer that $(y_k + 1)$

At step $k + 1$

$$P_{k+1} = 2\Delta y \cdot x_{k+1} - 2\Delta x \cdot y_{k+1} + c$$

$$P_{k+1} - P_k = 2\Delta y (x_{k+1} - x_k) - 2\Delta x (y_{k+1} - y_k)$$

But $x_{k+1} = x_k + 1$

$$P_{k+1} = P_k + 2\Delta y - 2\Delta x (y_{k+1} - y_k)$$

$y_{k+1} - y_k$ is either 0 or 1, depending on the sign of P_k

At the starting pixel position (x_0, y_0) $P_0 = 2\Delta y - \Delta x$

15

Bresenham's Line Algorithm for $|m| < 1.0$

- Calculate the constants Δx , Δy , $2\Delta y$, and $2\Delta y - 2\Delta x$, and obtain the starting value for decision parameter as

$$P_0 = 2\Delta y - \Delta x$$
- At each x_k along the line, starting at $k = 0$, perform the following test. If $P_k < 0$, the next point to plot is $(x_k + 1, y_k)$ and

$$P_{k+1} = P_k + 2\Delta y$$
 Otherwise, the next point is $(x_k + 1, y_k + 1)$ and

$$P_{k+1} = P_k + 2\Delta y - 2\Delta x$$
- Perform step 2 $\Delta x - 1$ times

16

Bresenham's Example

- Line with endpoints $(20, 10)$ and $(30, 18)$
 - $m = 0.8$, $\Delta x = 10$, $\Delta y = 8$, $P_0 = 2\Delta y - \Delta x = 6$, $2\Delta y = 16$, $2\Delta y - 2\Delta x = -4$
- Plot initial point $(x_0, y_0) = (20, 10)$, and determine successive pixel positions

k	P_k	(x_{k+1}, y_{k+1})
0	6	(21, 11)
1	2	(22, 12)
2	-2	(23, 12)
3	14	(24, 13)
4	10	(25, 14)
5	6	(26, 15)
6	2	(27, 16)
7	-2	(28, 16)
8	14	(29, 17)
9	10	(30, 18)

17

Circle Drawing Algorithms

- A circle is a set of points that are a given distance r from center point (x_c, y_c)

$$(x - x_c)^2 + (y - y_c)^2 = r^2$$

$$y = y_c \pm \sqrt{r^2 - (x - x_c)^2}$$
- The calculations are not very efficient.
 - The square (multiply) operations
- To make our circle drawing algorithm more efficient is that circles centred at $(0, 0)$ have eight-way symmetry

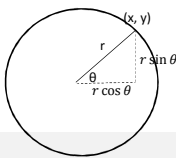
18

Circle Drawing Algorithms

- Another way: use polar coordinates r and θ to express circle in parameter polar form

$$x = x_c + r \cos \theta$$

$$y = y_c + r \sin \theta$$



```

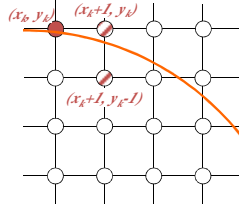
const float DEG2RAD = 3.14159/180;
void drawCircle (float radius)
{
  glBegin (GL_LINE_LOOP);
  for (int i=0; i < 360; i++) {
    float degInRad = i * DEG2RAD;
    glVertex2f (cos (degInRad) * radius, sin (degInRad) * radius);
  }
  glEnd();
}

```

19

Midpoint Circle Algorithm

- Assume that we have just plotted point (x_k, y_k)
- The next point is a choice between (x_k+1, y_k) and (x_k+1, y_k-1)
- We would like to choose the point that is nearest to the actual circle
- So how do we make this choice?



20

Midpoint Circle Algorithm

- Let's rearrange the equation of the circle slightly to give us:

$$f_{circ}(x, y) = x^2 + y^2 - r^2$$
- The equation evaluates as follows:

$$f_{circ}(x, y) \begin{cases} < 0, & \text{if } (x, y) \text{ is inside the circle boundary} \\ = 0, & \text{if } (x, y) \text{ is on the circle boundary} \\ > 0, & \text{if } (x, y) \text{ is outside the circle boundary} \end{cases}$$
- By evaluating this function at the midpoint between the candidate pixels we can make our decision

21

Midpoint Circle Algorithm

- Assuming we have just plotted the pixel at (x_k, y_k) so we need to choose between (x_k+1, y_k) and (x_k+1, y_k-1)
- Our decision variable can be defined as:

$$p_k = f_{circ}(x_k+1, y_k - \frac{1}{2})$$

$$= (x_k+1)^2 + (y_k - \frac{1}{2})^2 - r^2$$
- If $p_k < 0$ the midpoint is inside the circle and the pixel at y_k is closer to the circle
- Otherwise the midpoint is outside and $y_k - 1$ is closer

22

Midpoint Circle Algorithm

- To ensure things are as efficient as possible we can do all of our calculations incrementally
- First consider:

$$p_{k+1} = f_{circ}(x_{k+1}+1, y_{k+1} - \frac{1}{2})$$

$$= [(x_k+1)+1]^2 + (y_{k+1} - \frac{1}{2})^2 - r^2$$
- or:

$$p_{k+1} = p_k + 2(x_k+1) + (y_{k+1}^2 - y_k^2) - (y_{k+1} - y_k) + 1$$
- where y_{k+1} is either y_k or $y_k - 1$ depending on the sign of p_k

23

Midpoint Circle Algorithm

- The first decision variable at position $(x_0, y_0) = (0, r)$ is given as:

$$p_0 = f_{circ}(1, r - \frac{1}{2})$$

$$= 1 + (r - \frac{1}{2})^2 - r^2$$

$$= \frac{5}{4} - r$$
- Then if $p_k < 0$ then the next decision variable is given as:

$$p_{k+1} = p_k + 2x_{k+1} + 1$$
- If $p_k > 0$ then the decision variable is:

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

24

The Midpoint Circle Algorithm

1. Input radius r and circle centre (x_c, y_c) , then set the coordinates for the first point on the circumference of a circle centred on the origin as:

$$(x_0, y_0) = (0, r)$$

2. Calculate the initial value of the decision parameter as:

$$p_0 = \frac{5}{4}r^2 - r$$

3. Starting with $k=0$ at each position x_k , perform the following test. If $p_k < 0$, the next point along the circle centred on $(0, 0)$ is (x_{k+1}, y_k) and:

$$p_{k+1} = p_k + 2x_{k+1} + 1$$

25

The Midpoint Circle Algorithm

Otherwise the next point along the circle is (x_k+1, y_{k+1}) and:

$$p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$$

4. Determine symmetry points in the other seven octants
5. Move each calculated pixel position (x, y) onto the circular path centred at (x_c, y_c) to plot the coordinate values:

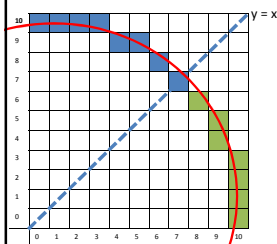
$$x = x + x_c \quad y = y + y_c$$

6. Repeat steps 3 to 5 until $x \geq y$

26

Midpoint Circle Example

- Given radius $r = 10$
- Determine position along the circle octant in the first quadrant from $x = 0$ to $x = y$
 - $p_0 = 1 - r = -9, \quad 2x_0 = 0, \quad 2y_0 = 20$

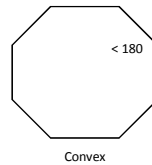


k	P_k	(x_{k+1}, y_{k+1})	$2x_{k+1}$	$2y_{k+1}$
0	-9	(1, 10)	2	20
1	-6	(2, 10)	4	20
2	-1	(3, 10)	6	20
3	6	(4, 9)	8	18
4	-3	(5, 9)	10	18
5	8	(6, 8)	12	16
6	7	(7, 7)	14	14

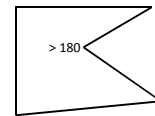
27

Polygon

- Polygon is a plane figure specified by set of three or more coordinate positions, called vertices
- If all interior angles of a polygon are less than or equal to 180 degrees, the polygon is **convex**
- A polygon that is not convex is called a **concave** polygon



Convex

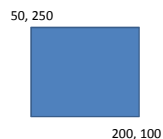


Concave

28

OpenGL Polygon Fill Area Functions

- `glRect* (x1, y1, x2, y2);` one corner is at coordinate position $(x1, y1)$ and the opposite corner is at position $(x2, y2)$
- `glRect (200, 100, 50, 250);`

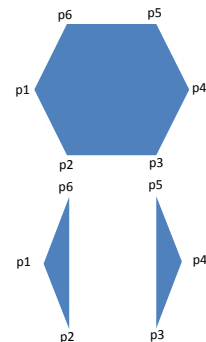


29

OpenGL Polygon Fill Area Functions

```
glBegin (GL_POLYGON);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glVertex2iv (p6);
glEnd ();

glBegin (GL_TRIANGLES);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p6);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glEnd ();
```

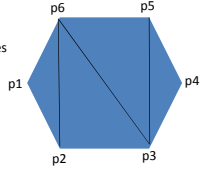


30

OpenGL Polygon Fill Area Functions

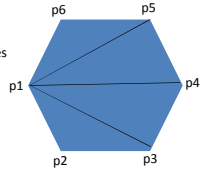
```
glBegin (GL_TRIANGLE_STRIP);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p6);
glVertex2iv (p3);
glVertex2iv (p5);
glVertex2iv (p4);
glEnd ();
```

N - 2 triangles



```
glBegin (GL_TRIANGLE_FAN);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glVertex2iv (p6);
glEnd ();
```

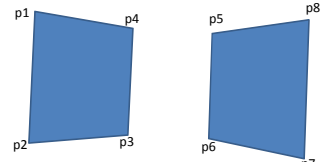
N - 2 triangles



31

OpenGL Polygon Fill Area Functions

```
glBegin (GL_QUADS);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glVertex2iv (p6);
glVertex2iv (p7);
glVertex2iv (p8);
glEnd ();
```



```
glBegin (GL_QUAD_STRIP);
glVertex2iv (p1);
glVertex2iv (p2);
glVertex2iv (p3);
glVertex2iv (p4);
glVertex2iv (p5);
glVertex2iv (p6);
glVertex2iv (p7);
glVertex2iv (p8);
glEnd ();
```

