Dire Dawa Institute of Technology
Department of Computer Science
Chapter 2 Handout – Graphics Primitives
Compiled by Gaddisa Olani, 2016

1. Introduction

All graphics packages construct pictures from basic building blocks known as *graphics primitives*. Primitives that describe the *geometry*, or shape, of these building blocks are known as *geometric primitives*. They can be anything from 2-D primitives such as points, lines and polygons to more complex 3-D primitives such as spheres and polyhedra (a *polyhedron* is a 3-D surface made from a mesh of 2-D polygons).

In the following sections we will examine some algorithms for drawing different primitives, and where appropriate we will introduce the routines for displaying these primitives in OpenGL.

2. OpenGL Point Drawing Primitives

The most basic type of primitive is the point. Many graphics packages, including OpenGL, provide routines for displaying points. We have already seen the OpenGL routines for point drawing in the simple OpenGL program introduced in Chapter 1. To recap, we use the pair of functions *glBegin ... glEnd*, using the symbolic constant *GL_POINTS* to specify how the vertices in between should be interpreted. In addition, the function *glPointSize* can be used to set the size of the points in pixels. The default point size is 1 pixel. Points that have a size greater than one pixel are drawn as squares with a side length equal to their size. For example, the following code draws three 2-D points with a size of 2 pixels.

```
glPointSize(2.0);
glBegin(GL_POINTS);
  glVertex2f(100.0, 200.0);
  glVertex2f(150.0, 200.0);
  glVertex2f(150.0, 250.0);
glEnd();
```

Note that when we specify 2-D points, OpenGL will actually create 3-D points with the third coordinate (the *z*-coordinate) equal to zero. Therefore, there is not really any such thing as 2-D graphics in OpenGL – but we can simulate 2-D graphics by using a constant *z*-coordinate.

3. Line Drawing Algorithms

Lines are a very common primitive and will be supported by almost all graphics packages. In addition, lines form the basis of more complex primitives such as *polylines* (a connected sequence of straight-line segments) or *polygons* (2-D objects formed by straight-line edges).

Lines are normally represented by the two end-points of the line, and points *(x,y)* along the line must satisfy the following *slope-intercept* equation:

$$y = mx + c \quad \text{.......} \quad (1)$$

Where *m* is the *slope* or *gradient* of the line, and *c* is the coordinate at which the line *intercepts* the *y*-axis. Given two end-points (x_0,y_0) and (x_{end},y_{end}), we can calculate values for *m* and *c* as follows:

$$m = \frac{(y_{end} - y_0)}{(x_{end} - x_0)} \quad \text{.......} \quad (2)$$

$$c = y_0 - mx_0 \quad \text{.......} \quad (3)$$

Furthermore, for any given *x*-interval δx, we can calculate the corresponding *y*-interval δy:

$$\delta y = m.\delta x \quad \text{.......} \quad (4)$$

$$\delta x = (1/m).\delta y \quad \text{.......} \quad (5)$$

These equations form the basis of the two line-drawing algorithms described below: the *DDA algorithm* and *Bresenham's algorithm*.

3.1. DDA Line-Drawing Algorithm

The *Digital Differential Analyser* (DDA) algorithm operates by starting at one end-point of the line, and then using Eqs. (4) and (5) to generate successive pixels until the second end-point is reached. Therefore, first, we need to assign values for δx and δy.

Before we consider the actual DDA algorithm, let us consider a simple first approach to this problem. Suppose we simply increment the value of *x* at each iteration (i.e. $\delta x = 1$), and then compute the corresponding value for y using Eqs. (2) and (4). This would compute correct line points but, as illustrated by Figure 1, it would leave gaps in the line. The reason for this is that the value of δy is greater than one, so the gap between subsequent points in the line is greater than 1 pixel.

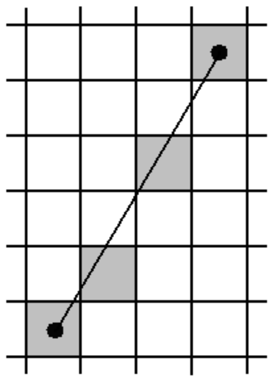


Figure 1 – 'Holes' in a Line Drawn by Incrementing *x* and Computing the Corresponding *y*-Coordinate

The solution to this problem is to make sure that both δx and δy have values less than or equal to one. To ensure this, we must first check the size of the line gradient. The conditions are:

- If $|m| \leq 1$:
    - $\delta x = 1$
    - $\delta y = m$
- If $|m| > 1$:
    - $\delta x = 1/m$
    - $\delta y = 1$

Once we have computed values for δx and δy, the basic DDA algorithm is:

- Start with (x_0,y_0)
- Find successive pixel positions by adding on $(\delta x, \delta y)$ and rounding to the nearest integer, i.e.
    - $x_{k+1} = x_k + \delta x$
    - $y_{k+1} = y_k + \delta y$
- For each position (x_k,y_k) computed, plot a line point at $(round(x_k),round(y_k))$, where the *round* function will round to the nearest integer.

Note that the actual pixel value used will be calculated by rounding to the nearest integer, but we keep the real-valued location for calculating the next pixel position.

Let us consider an example of applying the DDA algorithm for drawing a straight-line segment. Referring to see Figure 2, we first compute a value for the gradient *m*:

$$m = \frac{(y_{end} - y_0)}{(x_{end} - x_0)} = \frac{(13-10)}{(15-10)} = \frac{3}{5} = 0.6$$

Now, because $|m| \leq 1$, we compute δx and δy as follows:

$\delta x = 1$
$\delta y = 0.6$

Using these values of δx and δy we can now start to plot line points:

- Start with (x_0,y_0) = **(10,10)** – colour this pixel
- Next, (x_1,y_1) = (10+1,10+0.6) = (11,10.6) – so we colour pixel **(11,11)**
- Next, (x_2,y_2) = (11+1,10.6+0.6) = (12,11.2) – so we colour pixel **(12,11)**
- Next, (x_3,y_3) = (12+1,11.2+0.6) = (13,11.8) – so we colour pixel **(13,12)**
- Next, (x_4,y_4) = (13+1,11.8+0.6) = (14,12.4) – so we colour pixel **(14,12)**
- Next, (x_5,y_5) = (14+1,12.4+0.6) = (15,13) – so we colour pixel **(15,13)**
- We have now reached the end-point (x_{end}, y_{end}), so the algorithm terminates

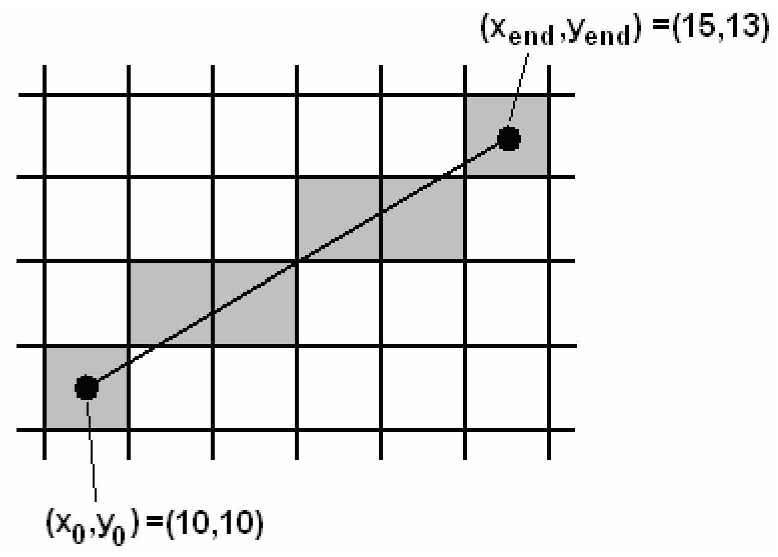


Figure 2 - The Operation of the DDA Line-Drawing Algorithm

The DDA algorithm is simple and easy to implement, but it does involve floating point operations to calculate each new point. Floating point operations are time-consuming when compared to integer operations. Since line-drawing is a very common operation in computer graphics, it would be nice if we could devise a faster algorithm which uses integer operations only. The next section describes such an algorithm.

3.2. Bresenham's Line-Drawing Algorithm

Bresenham's line-drawing algorithm provides significant improvements in efficiency over the DDA algorithm. These improvements arise from the observation that for any given line, if we know the previous pixel location, we only have a choice of 2 locations for the next pixel. This concept is illustrated in Figure 3: given that we know (x_k, y_k) is a point on the line, we know the next line point must be either pixel *A* or pixel *B*. Therefore we do not need to compute the actual floating-point location of the 'true' line point; we need only make a *decision* between pixels *A* and *B*.

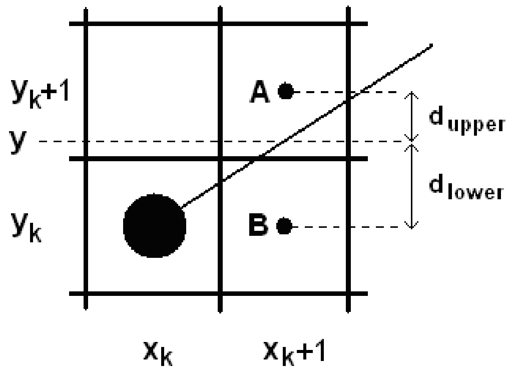


Figure 3 - Bresenham's Line-Drawing Algorithm

Bresenham's algorithm works as follows. First, we denote by d_{upper} and d_{lower} the distances between the centres of pixels *A* and *B* and the 'true' line (see Figure 3). Using Eq. (1) the 'true' *y*-coordinate at x_k+1 can be calculated as:

$$y = m(x_k + 1) + c \quad \text{..........} \quad (6)$$

Therefore we compute d_{lower} and d_{upper} as:

$$d_{lower} = y - y_k = m(x_k + 1) + c - y_k \quad \text{..........} \quad (7)$$
$$d_{upper} = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - c \quad \text{..........} \quad (8)$$

Now, we can decide which of pixels *A* and *B* to choose based on comparing the values of d_{upper} and d_{lower}:

- If $d_{lower} > d_{upper}$, choose pixel A
- Otherwise choose pixel B

We make this decision by first subtracting d_{upper} from d_{lower}:

$$d_{lower} - d_{upper} = 2m(x_k + 1) - 2y_k + 2c - 1 \quad \text{..........} \quad (9)$$

If the value of this expression is positive we choose pixel A; otherwise we choose pixel B. The question now is how we can compute this value efficiently. To do this, we define a *decision variable* p_k for the k^{th} step in the algorithm and try to formulate p_k so that it can be computed using only integer operations. To achieve this, we substitute $m = \Delta y / \Delta x$ (where Δx and Δy are the horizontal and vertical separations of the two line end-points) and define p_k as:

$$p_k = \Delta x(d_{lower} - d_{upper}) = 2\Delta y x_k - 2\Delta x y_k + d \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (10)$$

Where d is a constant that has the value $2\Delta y + 2c\Delta x - \Delta x$. Note that the sign of p_k will be the same as the sign of $(d_{lower} - d_{upper})$, so if p_k is positive we choose pixel A and if it is negative we choose pixel B. In addition, p_k can be computed using only integer calculations, making the algorithm very fast compared to the DDA algorithm.

An efficient *incremental calculation* makes Bresenham's algorithm even faster. (An *incremental calculation* means we calculate the next value of p_k from the last one.) Given that we know a value for p_k, we can calculate p_{k+1} from Eq. (10) by observing that:

- Always $x_{k+1} = x_k+1$
- If $p_k < 0$, then $y_{k+1} = y_k$, otherwise $y_{k+1} = y_k+1$

Therefore we can define the incremental calculation as:

$$p_{k+1} = p_k + 2\Delta y\text{, if } p_k < 0 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (11)$$

$$p_{k+1} = p_k + 2\Delta y - 2\Delta x\text{, if } p_k \geq 0 \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (12)$$

The initial value for the decision variable, p_0, is calculated by substituting $x_k = x_0$ and $y_k = y_0$ into Eq. (10), which gives the following simple expression:

$$p_0 = 2\Delta y - \Delta x \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (13)$$

So we can see that we never need to compute the value of the constant d in Eq. (10).

Summary

To summarise, we can express Bresenham's algorithm as follows:

- Plot the start-point of the line (x_0,y_0)
- Compute the first decision variable:
  - $p_0 = 2\Delta y - \Delta x$
- For each k, starting with k=0:
  - If $p_k < 0$:
    - Plot (x_k+1,y_k)
    - $p_{k+1} = p_k + 2\Delta y$
  - Otherwise:
    - Plot (x_k+1,y_k+1)
    - $p_{k+1} = p_k + 2\Delta y - 2\Delta x$

The steps given above will work for lines with positive $|m| < 1$. For $|m| > 1$ we simply swap the roles of x and y. For negative slopes one coordinate decreases at each iteration whilst the other increases.

Exercise

Consider the example of plotting the line shown in Figure 2 using Bresenham's algorithm:

- First, compute the following values:
    - $\Delta x = 5$
    - $\Delta y = 3$
    - $2\Delta y = 6$
    - $2\Delta y - 2\Delta x = -4$
    - $p_0 = 2\Delta y - \Delta x = 2 \times 3 - 5 = 1$
- Plot (x_0,y_0) = **(10,10)**
- Iteration 0:
    - $p_0 \geq 0$, so
        - Plot $(x_1,y_1) = (x_0+1,y_0+1)$ = **(11,11)**
        - $p_1 = p_0 + 2\Delta y - 2\Delta x = 1 - 4 = -3$
- Iteration 1:
    - $p_1 < 0$, so
        - Plot $(x_2,y_2) = (x_1+1,y_1)$ = **(12,11)**
        - $p_2 = p_1 + 2\Delta y = -3 + 6 = 3$
- Iteration 2:
    - $p_2 \geq 0$, so
        - Plot $(x_3,y_3) = (x_2+1,y_2+1)$ = **(13,12)**
        - $p_3 = p_2 + 2\Delta y - 2\Delta x = 3 - 4 = -1$
- Iteration 3:
    - $p_3 < 0$, so
        - Plot $(x_4,y_4) = (x_3+1,y_3)$ = **(14,12)**
        - $p_4 = p_3 + 2\Delta y = -1 + 6 = 5$
- Iteration 4:
    - $p_4 \geq 0$, so
        - Plot $(x_5,y_5) = (x_4+1,y_4+1)$ = **(15,13)**
        - We have reached the end-point, so the algorithm terminates

We can see that the algorithm plots exactly the same points as the DDA algorithm but it computes those using only integer operations. For this reason, Bresenham's algorithm is the most popular choice for line-drawing in computer graphics.

3.3. OpenGL Line Functions

We can draw straight-lines in OpenGL using the same *glBegin* … *glEnd* functions that we saw for point-drawing. This time we specify that vertices should be interpreted as line end-points by using the symbolic constant *GL_LINES*. For example, the following code

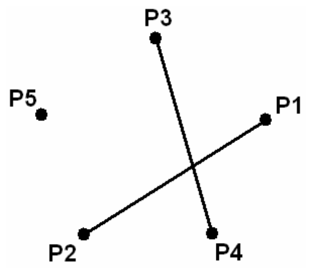
```
glLineWidth(3.0);
glBegin(GL_LINES);
  glVertex2f(100.0, 200.0);
  glVertex2f(150.0, 200.0);
  glVertex2f(150.0, 250.0);
  glVertex2f(200.0, 250.0);
glEnd()
```

Will draw two separate line segments: one from (100,200) to (150,200) and one from (150,250) to (200,250). The line will be drawn in the current drawing colour and with a width defined by the argument of the function *glLineWidth*.

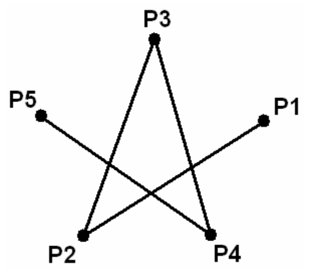
Two other symbolic constants allow us to draw slightly different types of straight-line primitive: *GL_LINE_STRIP* and *GL_LINE_LOOP*. The following example illustrates the difference between the three types of line primitive. First we define 5 points as arrays of 2 *Glint* values. Next, we define exactly the same vertices for each of the three types of line primitive. The images to the right show how the vertices will be interpreted by each primitive.

```
Glint p1[] = {200,100};  Glint p2[] = {50,0}
Glint p3[] = {100,200};  Glint p4[] = {150,0};
Glint p5[] = {0,100};
```

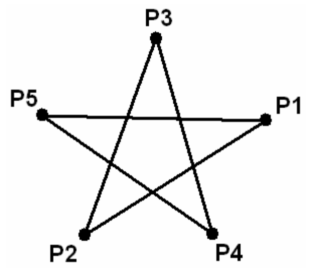
```
glBegin(GL_LINES);
glVertex2iv(p1);
glVertex2iv(p2);
glVertex2iv(p3);
glVertex2iv(p4);
  glVertex2iv(p5);
glEnd();
```



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



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



We can see that *GL_LINES* treats the vertices as pairs of end-points. Lines are drawn separately and any extra vertices (i.e. a start-point with no end-point) are ignored. *GL_LINE_STRIP* will create a connected *polyline*, in which each vertex is joined to the one before it and after it. The first and last vertices are only joined to one other vertex. Finally, *GL_LINE_LOOP* is the same as *GL_LINE_STRIP* except that the last point is joined to the first one to create a loop.

4. Circle-Drawing Algorithms

Some graphics packages allow us to draw circle primitives. Before we examine algorithms for circle-drawing we will consider the mathematical equations of a circle. In Cartesian coordinates we can write:

$$(x-x_c)^2+(y-y_c)^2=r^2 \quad \dots\dots \quad (14)$$

where (x_c,y_c) is the centre of the circle. Alternatively, in polar coordinates we can write:

$$x=x_c+r\cos\theta \quad \dots\dots \quad (15)$$

$$y=y_c+r\sin\theta \quad \dots\dots \quad (16)$$

In the following sections we will examine a number of approaches to plotting points on a circle, culminating in the most efficient algorithm: the *midpoint algorithm*.

4.1. Plotting Points Using Cartesian Coordinates

As a first attempt at a circle-drawing algorithm we can use the Cartesian coordinate representation. Suppose we successively increment the *x*-coordinate and calculate the corresponding *y*-coordinate using Eq. (14):

$$y=y_c\pm\sqrt{r^2-\left(x_c-x\right)^2} \quad \dots\dots \quad (17)$$

This would correctly generate points on the boundary of a circle. However, like the first attempt at a line-drawing algorithm we saw in Section 3.1 (see Figure 1) we would end up with ‘holes’ in the line – see Figure 4. We would have the same problem if we incremented the *y*-coordinate and plotted a calculated *x*-coordinate. As with the DDA line-drawing algorithm we can overcome this problem by calculating and checking the gradient: if $|m| \leq 1$ then increment *x* and calculate *y*, and if $|m| > 1$ then increment *y* and calculate *x*. However, with the DDA algorithm we only needed to compute and check the gradient once for the entire line, but for circles the gradient changes with each point plotted, so we would need to compute and check the gradient at each iteration of the algorithm. This fact, in addition to the square root calculation in Eq. (17), would make the algorithm quite inefficient.

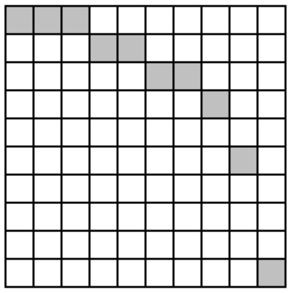


Figure 4 - Circle-Drawing by Plotting Cartesian Coordinates

4.2. Plotting Points Using Polar Coordinates

An alternative technique is to use the polar coordinate equations. Recall that in polar coordinates we express a position in the coordinate system as an angle θ and a distance r. For a circle, the radius r will be constant, but we can increment θ and compute the corresponding x and y values according to Eqs. (15) and (16).

For example, suppose we want to draw a circle with $(x_c,y_c) = (5,5)$ and $r = 10$. We start with $\theta = 0^o$ and compute x and y as:

- $x = 5 + 10 \cos 0^o = 15$
- $y = 5 + 10 \sin 0^o = 5$
- Therefore we plot (15,5)

Next, we increase θ to 5^o:

- $x = 5 + 10 \cos 5^o = 14.96$
- $y = 5 + 10 \sin 5^o = 5.87$
- Therefore we plot (15,6)

Increase θ to 10^o:

- $x = 5 + 10 \cos 10^o = 14.85$
- $y = 5 + 10 \sin 10^o = 6.73$
- Therefore we plot (15,7)

This process would continue until we had plotted the entire circle (i.e. $\theta = 360^o$). Using this polar coordinate technique, we can avoid holes in the boundary if we make small enough increases in the value of θ. In fact, if we use $\theta = 1/r$ (where r is measured in pixels, and θ in radians) we will get points exactly 1 pixel apart and so there is guaranteed to be no holes.

This algorithm is more efficient than the Cartesian plotting algorithm described in Section 4.1. It can be made even more efficient, at a slight cost in quality, by increasing the size of the steps in the value of θ and then joining the computed points by straight-line segments (see Figure 5).

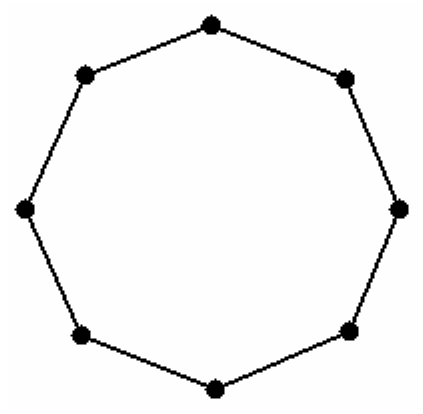


Figure 5 - Circle-Drawing by Plotting Polar Coordinates

4.3. Taking Advantage of the Symmetry of Circles in Plotting

We can improve the efficiency of any circle-drawing algorithm by taking advantage of the *symmetry* of circles. As illustrated in Figure 6, when we compute the Cartesian coordinates *x* and *y* of points on a circle boundary we have 4 axes of symmetry (shown in blue): we can generate a point reflected in the *y*-axis by negating the *x*-coordinate; we can generate a point reflected in the *x*-axis by negating the *y*-coordinate, and so on. In total, for each point computed, we can generate seven more through symmetry. Computing these extra points just by switching or negating the coordinates of a single point is much more efficient than computing each boundary point separately. This means that we only need to compute boundary points for one *octant* (i.e. one eighth) of the circle boundary – shown in red in Figure 6.

In addition to generating extra points, the 4-way symmetry of circles has another advantage if combined with the Cartesian plotting algorithm described in Section 4.1. Recall that this algorithm resulted in holes in some parts of the circle boundary (see Figure 4), which meant that a time-consuming gradient computation had to be performed for each point. In fact, the problem of holes in the boundary only occurs when the gradient is greater than 1 for computing *x*-coordinates, or when the gradient is less than or equal to 1 for computing *y*-coordinates. Now that we know we only need to compute points for one octant of the circle we do not need to perform this check. For example, in the red octant in Figure 6, we know that the gradient will never become greater than one, so we can just increment the *y*-coordinate and compute the corresponding *x*-coordinates.

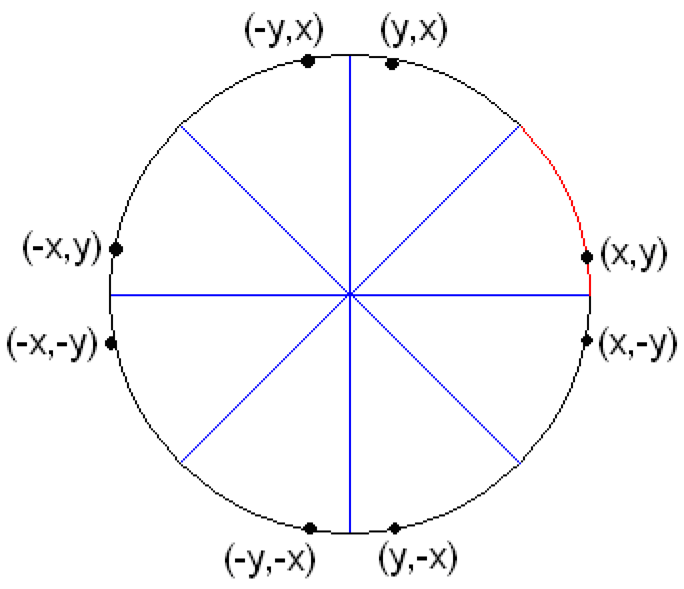


Figure 6 - Four-Way Symmetry of Circles

However, even with these efficiency improvements due to symmetry, we still need to perform a square root calculation (for Cartesian plotting) or a trigonometric calculation (for polar plotting). It would be nice if there were an algorithm for circle-drawing that used integer operations only, in the same way that Bresenham's algorithm does for line-drawing.

4.4. Midpoint Circle-Drawing Algorithm

The *midpoint algorithm* takes advantage of the symmetry property of circles to produce a more efficient algorithm for drawing circles. The algorithm works in a similar way to Bresenham's line-drawing algorithm, in that it formulates a decision variable that can be computed using integer operations only.

The midpoint algorithm is illustrated in Figure 7. Recall that we only need to draw one octant of the circle, as the other seven octants can be generated by symmetry. In Figure 7 we are drawing the right-hand upper octant – the one with coordinate (y,x) in Figure 6, but the midpoint algorithm would work with slight modifications whichever octant we chose. Notice that in this octant, when we move from one pixel to try to draw the next pixel there is only a choice of two pixels: *A* and *B*, or (x_{k+1},y_k) and (x_{k+1},y_{k-1}). Therefore we don't need to calculate a real-valued coordinate and then round to the nearest pixel, we just need to make a *decision* as to which of the two pixels to choose.

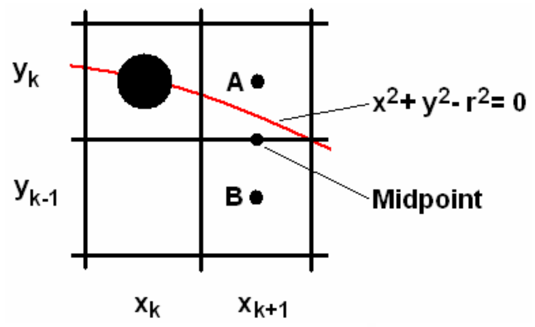


Figure 7 - The Midpoint Circle-Drawing Algorithm

We start by defining a function f_{circ} as follows:

$$f_{circ}(x,y) = x^2 + y^2 - r^2 \quad \ldots\ldots\ldots\ldots \quad (18)$$

This term can be derived directly from Eq. (14). Based on the result of this function, we can determine the position of any point relative to the circle boundary:

- For points *on* circle, $f_{circ} = 0$
- For points *inside* circle, $f_{circ} < 0$
- For points *outside* circle, $f_{circ} > 0$

Now referring again to Figure 7, we note that the position of the *midpoint* of the two pixels *A* and *B* can be written as:

$$mid_k = (x_k+1, y_k-0.5) \quad \ldots\ldots\ldots\ldots \quad (19)$$

Now we can see from Figure 7 that if the midpoint lies inside the circle boundary the next pixel to be plotted should be pixel *A*. Otherwise it should be pixel *B*. Therefore we can use the value of $f_{circ}(mid_k)$ to make the decision between the two candidate pixels:

- If $f_{circ}(mid_k) < 0$, choose *A*
- Otherwise choose *B*

In order to make this decision quickly and efficiently, we define a *decision variable* p_k, by combining Eqs. (18) and (19):

$$p_k = f_{circ}(x_k+1, y_k-0.5) = (x_k+1)^2 + (y_k-0.5)^2 - r^2 \quad \ldots\ldots\ldots\ldots \quad (20)$$

An incremental calculation for p_{k+1} can be derived by subtracting p_k from p_{k+1} and simplifying – the result is (Therefore we can define the incremental calculation as):

$$p_{k+1} = p_k + 2x_{k+1} + 1, \text{ if } p_k < 0 \quad \ldots\ldots\ldots\ldots \quad (21)$$

$$p_{k+1} = p_k + 2x_{k+1} + 1 + 2y_{k+1}, \text{ if } p_k \geq 0 \quad \ldots\ldots\ldots\ldots \quad (22)$$

The initial value of the decision variable, p_0, is calculated by evaluating it at the starting point $(0,r)$:

$$p_0 = f_{circ}(1, r-0.5) = 1 + (r-0.5)^2 - r^2 = \frac{5}{4} - r. \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (23)$$

If r is an integer, then all increments are integers and we can round Eq. (23) to the nearest integer:

$$p_0 = 1 - r \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (24)$$

Summary

To summarise, we can express the midpoint circle-drawing algorithm for a circle centred at the origin as follows:

- Plot the start-point of the circle $(x_0,y_0) = (0,r)$
- Compute the first decision variable:
  - $p_0 = 1 - r$
- For each k, starting with k=0:
  - If $p_k < 0$:
    - Plot (x_k+1, y_k)
    - $p_{k+1} = p_k + 2x_{k+1} + 1$
  - Otherwise:
    - Plot (x_k+1, y_k-1)
    - $p_{k+1} = p_k + 2x_{k+1} + 1 - 2y_{k+1}$

Example

For example, given a circle of radius r=10, centred at the origin, the steps are:

- First, compute the initial decision variable:
  - $p_0 = 1 - r = -9$
- Plot $(x_0,y_0) = (0,r) =$ **(0,10)**
- Iteration 0:
  - $p_0 < 0$, so
    - Plot $(x_1,y_1) = (x_0+1,y_0) =$ **(1,10)**
    - $p_1 = p_0 + 2x_1 + 1 = -9 + 3 = -6$
- Iteration 1:
  - $p_1 < 0$, so
    - Plot $(x_2,y_2) = (x_1+1,y_1) =$ **(2,10)**
    - $p_2 = p_1 + 2x_2 + 1 = -6 + 5 = -1$
- Iteration 2:
  - $p_2 < 0$, so
    - Plot $(x_3,y_3) = (x_2+1,y_2) =$ **(3,10)**
    - $p_3 = p_2 + 2x_3 + 1 = -1 + 7 = 6$

- Iteration 3:
    - $p_3 \geq 0$, so
        - Plot $(x_4,y_4) = (x_3+1,y_3-1) = \mathbf{(4,9)}$
        - $p_4 = p_3 + 2x_4 + 1 - 2y_4 = 6 - 9 = -3$
- Iteration 4:
    - $p_4 < 0$, so
        - Plot $(x_5,y_5) = (x_4+1,y_4) = \mathbf{(5,9)}$
        - $p_5 = p_4 + 2x_5 + 1 = -3 + 11 = 8$
- Iteration 5:
    - $p_5 \geq 0$, so
        - Plot $(x_6,y_6) = (x_5+1,y_5-1) = \mathbf{(6,8)}$
        - $p_6 = p_5 + 2x_6 + 1 - 2y_6 = 8 - 3 = 5$
- Etc.

5. Ellipse-Drawing Algorithms

Some graphics packages may provide routines for drawing ellipses, although as we will see ellipses cannot be drawn as efficiently as circles. The equation for a 2-D ellipse in Cartesian coordinates is:

$$\left(\frac{x - x_c}{r_x}\right)^2 + \left(\frac{y - y_c}{r_y}\right)^2 = 1 \quad \ldots\ldots \quad (25)$$

Alternatively, in polar coordinates, we can write:

$$x = x_c + r_x \cos\theta \quad \ldots\ldots \quad (26)$$
$$y = y_c + r_y \sin\theta \quad \ldots\ldots \quad (27)$$

To understand these equations, refer to Figure 8. Notice that Eqs. (25)-(27) are similar to the circle equations given in Eqs. (14)-(16), except that ellipses have two radius parameters: r_x and r_y. The long axis (in this case, r_x) is known as the *major axis* of the ellipse; the short axis (in this case r_y) is known as the *minor axis*. The ratio of the major to the minor axis lengths is called the *eccentricity* of the ellipse.

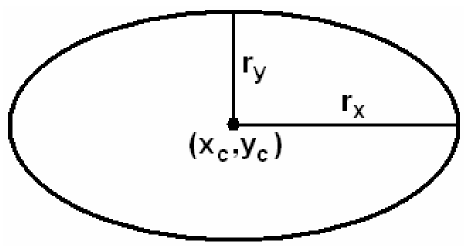


Figure 8 - Ellipse Drawing

5.1. The Midpoint Algorithm for Ellipse Drawing

The most efficient algorithm for drawing ellipses is an adaptation of the midpoint algorithm for circle-drawing. Recall that in the midpoint algorithm for circles we first had to define a term whose sign indicates whether a point is inside or outside the circle boundary, and then we applied it to the midpoint of the two candidate pixels. For ellipses, we define the function $f_{ellipse}$ as:

$$f_{ellipse}(x,y) = r_y^2x^2 + r_x^2y^2 - r_x^2r_y^2 \quad \ldots\ldots\ldots\ldots\ldots\ldots \quad (28)$$

Eq. (28) can be derived from Eq. (25) by assuming that (x_c,y_c) = (0,0), and then multiplying both sides by $r_x^2r_y^2$. Now we can say the same for $f_{ellipse}$ as we did for the circle function:

- For points on ellipse $f_{ellipse}= 0$
- For points inside ellipse $f_{ellipse}< 0$
- For points outside ellipse $f_{ellipse}> 0$

But at this point we have a slight problem. For circles there was 4-way symmetry so we only needed to draw a single octant. This meant we always had a choice of just 2 pixels to plot at each iteration. But ellipses only have 2-way symmetry, which means we have to draw an entire *quadrant* (see Figure 9).

Therefore the midpoint algorithm for ellipses is:

- At each point, compute gradient *m*:
  - If $|m| \leq 1$ we increment *x* and decide between candidate *y* positions
  - If $|m| > 1$ we increment *y* and decide between candidate *x* positions

Again, a fast incremental algorithm exists to compute a decision function, but because of the gradient calculation the midpoint ellipse-drawing algorithm is not as efficient as the midpoint circle drawing algorithm.

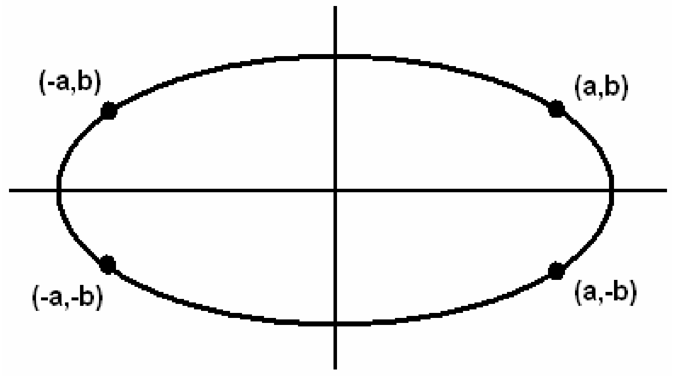


Figure 9 - Two-Way Symmetry of Ellipses

6. Fill-Area Primitives

The most common type of primitive in 3-D computer graphics is the *fill-area primitive*. The term *fill-area primitive* refers to any enclosed boundary that can be filled with a solid colour or pattern. However, fill-area primitives are normally *polygons*, as they can be filled more efficiently by graphics packages. Polygons are 2-D shapes whose boundary is formed by any number of connected straight-line segments. They can be defined by three or more *coplanar vertices* (*coplanar* points are positioned on the same plane). Each pair of adjacent *vertices* is connected in sequence by *edges*. Normally polygons should have no edge crossings: in this case they are known as *simple polygons* or *standard polygons* (see Figure 10).

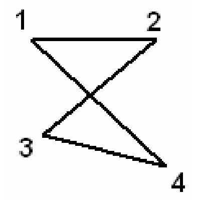


Figure 10 - A Polygon with an Edge Crossing

Polygons are the most common form of graphics primitive because they form the basis of *polygonal meshes*, which is the most common representation for 3-D graphics objects. Polygonal meshes approximate curved surfaces by forming a *mesh* of simple polygons. Some examples of polygonal meshes are shown in Figure 11.

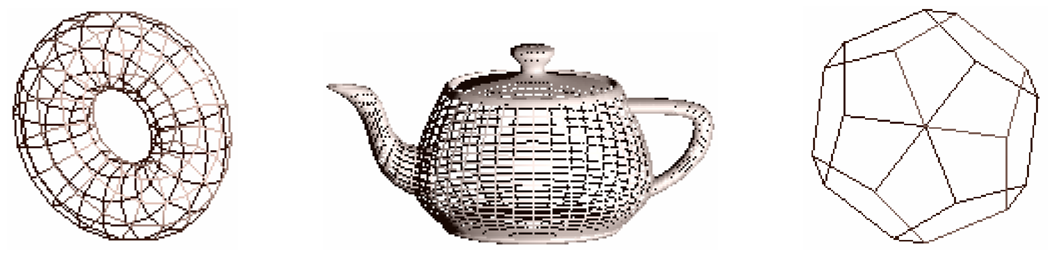


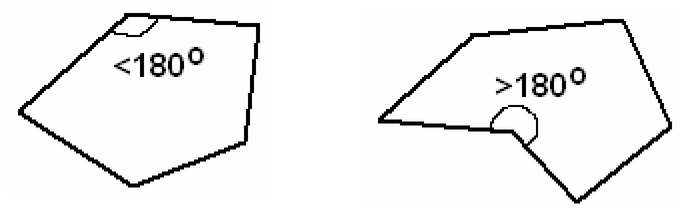
Figure 11 - Examples of Polygonal Mesh Surfaces

6.1. Convex and Concave Polygons

We can differentiate between *convex* and *concave* polygons:

- Convex polygons have all interior angles $\leq 180^o$
- Concave polygons have at least one interior angle $> 180^o$

The difference between convex and concave polygons is shown in Figure 122. Many graphics packages (including OpenGL) require all fill-area polygons to be *convex* polygons. The reason for this, as we will see in Chapter 3, is that it is much more efficient to fill a polygon if we know it is convex. Some packages even require that all fill-area primitives are triangles, as this makes filling even more straightforward.



(a) **(b)**

Figure 12 – Types of Polygon: (a) Convex; (b) Concave

Most graphics packages also insist on some other conditions regarding polygons. Polygons may not be displayed properly if any of the following conditions are met:

- The polygon has less than 3 vertices; or
- The polygon has collinear vertices; or
- The polygon has non-coplanar vertices; or
- The polygon has repeated vertices.

Polygons that meet one of these conditions are often referred to as *degenerate polygons*. Degenerate polygons may not be displayed properly by graphics packages, but many packages (including OpenGL) will not check for degenerate polygons as this takes extra processing time which would slow down the rendering process. Therefore it is up to the programmer to make sure that no degenerate polygons are specified.

How can we identify if a given polygon is convex? A number of approaches exist:

- Compute and check all interior angles – if one is greater than 180^o then the polygon is concave; otherwise it is convex.
- Infinitely extend each edge, and check for an intersection of the infinitely extended edge with other edges in the polygon. If any intersections exist for any edge, the polygon is concave; otherwise it is convex.
- Compute the *cross-product* of each pair of adjacent edges. If all cross-products have the same sign for their *z*-coordinate (i.e. they point in the same direction away from the plane of the polygon), then the polygon is convex; otherwise it is concave.

The last technique used the *cross-product* of vectors. The result of the cross-product of two vectors is a vector perpendicular to both vectors, whose magnitude is the product of the two vector magnitudes multiplied by the *sin* of the angle between them:

$$\vec{N} = \vec{u}|E_1||E_2|\sin\theta \text{, where } 0 \le \theta \le 180^o \qquad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \quad (29)$$

The direction of the cross-product is determined by the *right-hand rule*:

- *For V1xV2, with your right-hand grasp an axis that is perpendicular to V1 and V2, so that the fingers curl from V1 to V2 forming an angle <180°. The direction of the cross-product is the direction of your right thumb.*

This is illustrated in Figure 13. If you imagine *V1* pointing along the *x*-axis and *V2* pointing along the *y*-axis, then the *z*-axis is the direction of the cross-product. This is true so long as the angle between *V1* and *V2* is less than 180°. If it becomes greater than 180°, then we would need to switch *V1* and *V2* (i.e. *V1* is the *y*-axis and *V2* the *x*-axis), so that the angle between them became less than 180° again. In this case the cross-product would point in the opposite direction.

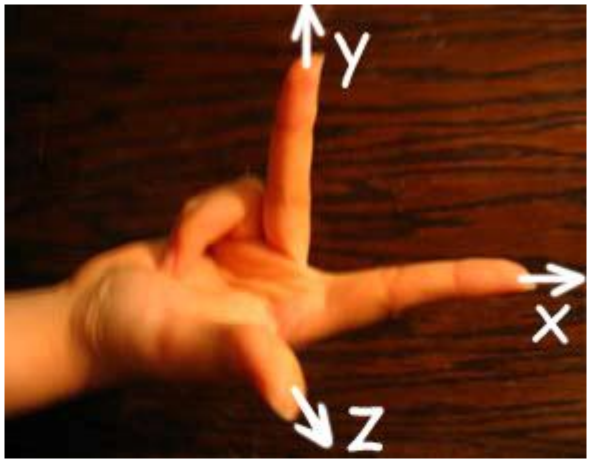


Figure 13 - The Right-Hand Rule for Computing Cross-Products

6.1.1. Splitting Concave Polygons

Because most graphics packages insist on polygons being convex, once we have identified concave polygons we need to split them up to create a number of convex polygons.

One technique for splitting concave polygons is known as the *vector technique*. This is illustrated in Figure 14, and can be summarised as follows:

- Compute the cross-product of each pair of adjacent edge vectors.
- When the cross-product of one pair has a different sign for its *z*-coordinate compared to the others (i.e. it points in the opposite direction):
    - Extend the first edge of the pair until it intersects with another edge.
    - Form a new vertex at the intersection point and split the polygon into two.

Recall from Section 6.1 that the cross-product switches direction when the angle between the vectors becomes greater than 180°. Therefore the vector technique is detecting this condition by using a cross-product.

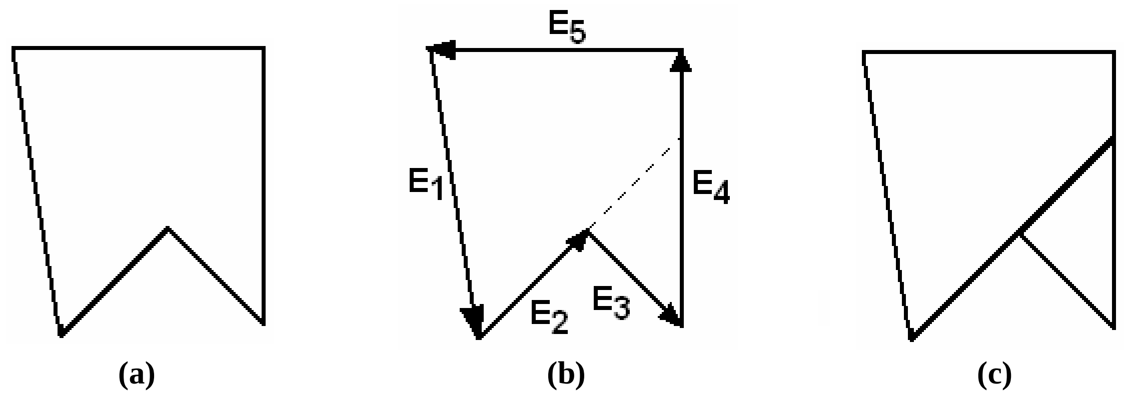


Figure 14 - Splitting a Concave Polygon into Convex Polygons

6.2. Polygon Inside-Outside Tests

In order to fill polygons we need some way of telling if a given point is inside or outside the polygon boundary: we call this an *inside-outside test*. We will see in Chapter 7 that such tests are also useful for the *ray-tracing* rendering algorithm.

We will examine two different inside-outside tests: the *odd-even rule* and the *nonzero winding number rule*. Both techniques give good results, and in fact usually their results are the same, apart from for some more complex polygons.

6.2.1. Odd-Even Rule

The odd-even rule is illustrated in Figure 15(a). Using this technique, we determine if a point *P* is inside or outside the polygon boundary by the following steps:

- Draw a line from *P* to some distant point (that is known to be outside the polygon boundary).
- Count the number of crossings of this line with the polygon boundary:
    - If the number of crossings is *odd*, then *P* is *inside* the polygon boundary.
    - If the number of crossings is *even*, then *P* is *outside* the polygon boundary.

We can see from Figure 15(a) that the two white regions are considered to be outside the polygon since they have two line crossings to any distant points.

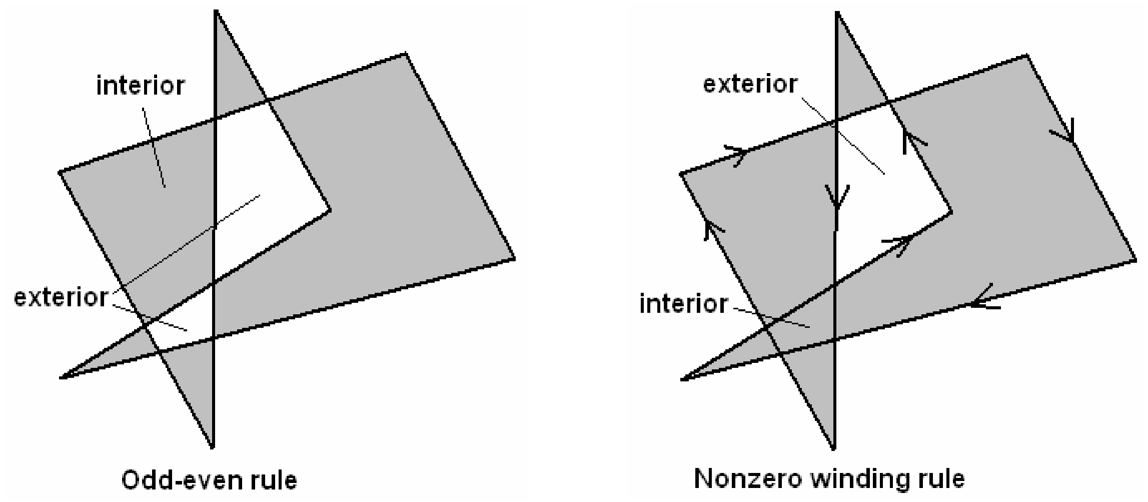
6.2.2. Nonzero Winding Number Rule

The nonzero winding number rule is similar to the odd-even rule, and is illustrated in Figure 15(b). This time we consider each edge of the polygon to be a vector, i.e. they have a direction as well as a position. These vectors are directed in a particular order around the boundary of the polygon (the programmer defines which direction the vectors go). Now we decide if a point *P* is inside or outside the boundary as follows:

- Draw a line from *P* to some distant point (that is known to be outside the polygon boundary).

- At each edge crossing, add 1 to the *winding number* if the edge goes from right to left, and subtract 1 if it goes from left to right.
    - If the total *winding number* is nonzero, *P* is inside the polygon boundary.
    - If the total *winding number* is zero, *P* is outside the polygon boundary.

We can see from Figure 15(b) that the nonzero winding number rule gives a slightly different result from the odd-even rule for the example polygon given. In fact, for most polygons (including all convex polygons) the two algorithms give the same result. But for more complex polygons such as that shown in Figure 15 the nonzero winding number rule allows the programmer a bit more flexibility, since they can control the order of the edge vectors to get the results they want.



(a) **(b)**

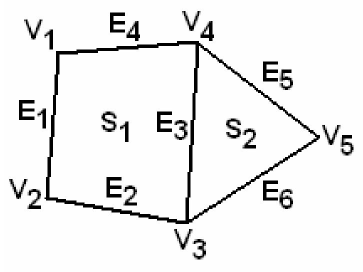
Figure 15 - Inside-Outside Tests for Polygons

6.3. Representing Polygons

Polygons, and in particular convex polygons, are the most common type of primitive in 3-D graphics because they are used to represent polygonal meshes such as those shown in Figure 11. But how can polygonal meshes be represented? A common technique is to use *tables* of data. These tables can be of two different types:

- *Geometric tables*: These store information about the *geometry* of the polygonal mesh, i.e. what are the shapes/positions of the polygons?
- *Attribute tables*: These store information about the appearance of the polygonal mesh, i.e. what colour is it, is it opaque or transparent, etc. This information can be specified for each polygon individually or for the mesh as a whole.

Figure 16 shows a simple example of a geometric table. We can see that there are three tables: a *vertex table*, an *edge table* and a *surface-facet table*. The edge table has pointers into the vertex table to indicate which vertices comprise the edge. Similarly the surface-facet table has pointers into the edge table. This is a compact representation for a polygonal mesh, because each vertex's coordinates are only stored once, in the vertex table, and also information about each edge is only stored once.



Vertex Table	
V_1:	x1,y1,z1
V_2:	x2,y2,z2
V_3:	x3,y3,z3
V_4:	x4,y4,z4
V_5:	x5,y5,z5

Edge Table	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_4
E_4:	V_1, V_4
E_5:	V_4, V_5
E_6:	V_3, V_5

Surface-Facet Table	
S_1:	E_1, E_2, E_3, E_4
S_2:	E_3, E_5, E_6

Figure 16 - A Geometric Table for Representing Polygons

6.4. Polygon Front and Back Faces

Most polygons are part of a 3-D polygonal mesh, which are often enclosed (solid) objects. Therefore when storing polygon information we need to know which *face* of the polygon is facing outward from the object. Every polygon has two faces: the *front-face* and the *back-face*. The front-face of a polygon is defined as the one that points outward from the object, whereas the back-face points towards the object interior.

Often in graphics we need to decide if a given point is on one side of a polygon or the other. For example, if we know the position of the virtual camera we may want to decide if the camera is looking at the front-face or the back-face of the polygon. (It can be more efficient for graphics packages not to render back-faces.)

We can make this calculation using the standard equation of a plane:

$$Ax + By + Cz + D = 0 \quad \ldots\ldots \quad (30)$$

Given at least three coplanar points (e.g. polygon vertices) we can always calculate the values of the coefficients A, B, C, and D for a plane. Now, for any given point (x,y,z):

- If $Ax + By + Cz + D = 0$, the point is *on* the plane.
- If $Ax + By + Cz + D < 0$, the point is *behind* the plane.
- If $Ax + By + Cz + D > 0$, the point is *in front of* the plane.

6.5. Polygon Normal Vectors

Polygon front-faces are usually identified using *normal vectors*. A normal vector is a vector that is perpendicular to the plane of the polygon and which points away from front face (see Figure 17).

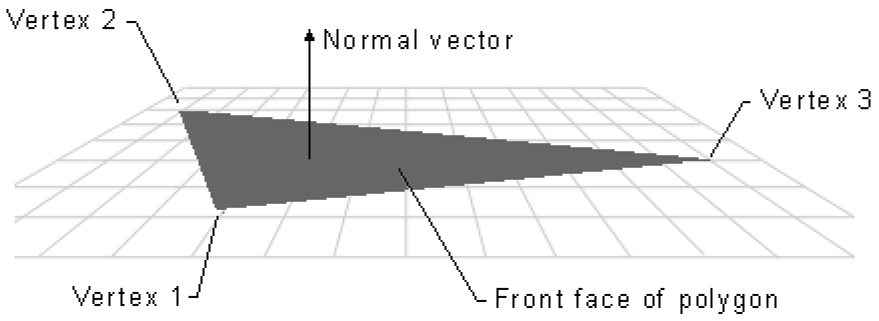


Figure 17 - The Normal Vector of a Polygon

6.5.1. Computing Normal Vectors

In graphics packages, we can either specify the normal vectors ourselves, or get the package to compute them automatically. How will a graphics package compute normal vectors automatically?

The simplest approach is to use the *cross-product* of adjacent edge vectors in the polygon boundary. Recall that the result of a cross-product is a vector that is perpendicular to the two vectors. Consider Figure 18. Here we can see three vertices of a polygon: V_1, V_2 and V_3. These form the two adjacent edge vectors E_1 and E_2:

$E_1 = (1,0,0)$
$E_2 = (0,1,0)$

Now, using Eq. (29) we can compute the surface normal $\vec{N} = \vec{u}|E_1||E_2|\sin\theta$, where $0 \leq \theta \leq 180^o$, and the direction of $\vec{u}$ is determined by the *right-hand rule*. This will give a vector that is perpendicular to the plane of the polygon. In this example, $\vec{N} = (0,0,1)$.

Notice that if we consider the edge vectors to go the other way round the boundary (i.e. clockwise instead of anti-clockwise) then the normal vector would point in the other direction. This is the reason why in most graphics packages it is important which order we specify our polygon vertices in: specifying them in an anti-clockwise direction will make the normal vector point towards us, whereas specifying them in a clockwise direction will make it point away from us.

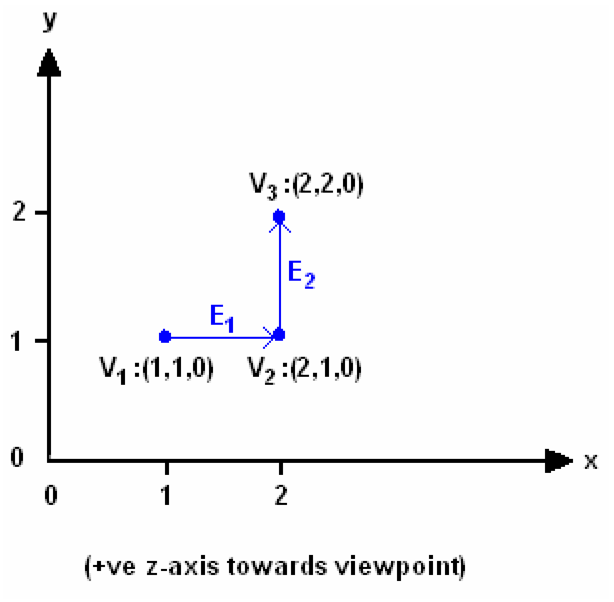


Figure 18 - Computing Surface Normals Using a Cross-Product of Adjacent Edges

6.6. OpenGL Fill-Area Routines

OpenGL provides a variety of routines to draw fill-area polygons. In all cases these polygons must be convex. In most cases the vertices should be specified in an *anti-clockwise* direction when viewing the polygon from outside the object, i.e. if you want the front-face to point towards you. The default fill-style is solid, in a colour determined by the current colour settings.

In all, there are seven different ways to draw polygons in OpenGL:

- *glRect**
- Six different symbolic constants for use with *glBegin … glEnd*

We will now consider each of these techniques in turn.

<u>*glRect**</u>

Two-dimensional rectangles can also be drawn using some of the other techniques described below, but because drawing rectangles in 2-D is a common task OpenGL provides the *glRect** routine especially for this purpose (*glRect** is more efficient for 2-D graphics than the other alternatives). The basic format of the routine is:

glRect (x1, y1, x2, y2)*

where *(x1,y1)* and *(x2,y2)* define *opposite* corners of the rectangle. Actually when we call the *glRect** routine, OpenGL will construct a polygon with vertices defined in the following order:

(x1,y1), (x2,y1), (x2,y2), (x1,y2)

For example, Figure 19 shows an example of executing the following call to *glRecti*:

glRecti(200,100,50,250);

(The black crosses are only shown for the purpose of illustrating where the opposing corners of the rectangle are.)

In 2-D graphics we don't need to worry about front and back faces – both faces will be displayed. But if we use *glRect** in 3-D graphics we must be careful. For example, in the above example we actually specified the vertices in a *clockwise* order. This would mean that the *back-face* would be facing *toward* the camera. To get an anti-clockwise order (and the *front-face* pointing towards the camera), we must specify the bottom-left and top-right corners in the call to *glRect**.

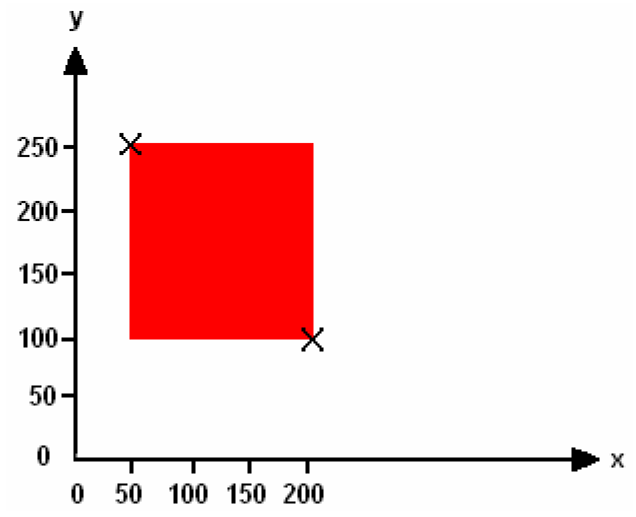
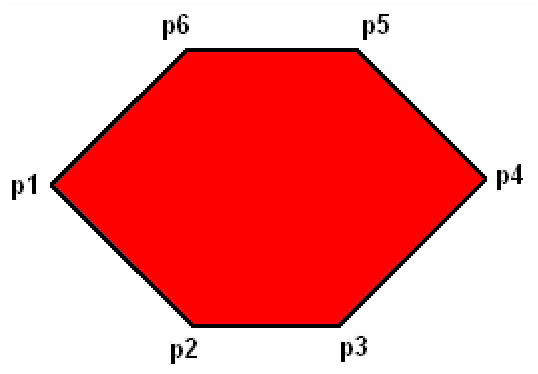


Figure 19 - Drawing a 2-D Rectangle with the OpenGL routine *glRect**

<u>*GL_POLYGON*</u>

The *GL_POLYGON* symbolic constant defines a single convex polygon. Like all of the following techniques for drawing fill-area primitives it should be used as the argument to the *glBegin* routine. For example, the code shown below will draw the shape shown in Figure 20. Notice that the vertices of the polygon are specified in anti-clockwise order.

glBegin(GL_POLYGON);



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

Figure 20 - A Polygon Drawn Using the GL_POLYGON OpenGL Primitive

GL_TRIANGLES

The *GL_TRIANGLES* symbolic constant causes the *glBegin ... glEnd* pair to treat the vertex list as groups of three 3 vertices, each of which defines a triangle. The vertices of each triangle must be specified in anti-clockwise order. Figure 21 illustrates the use of the *GL_TRIANGLES* primitive.

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

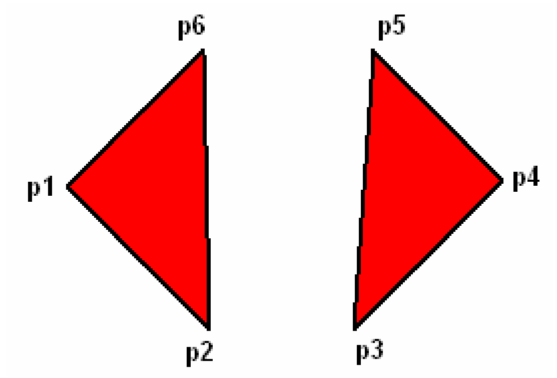
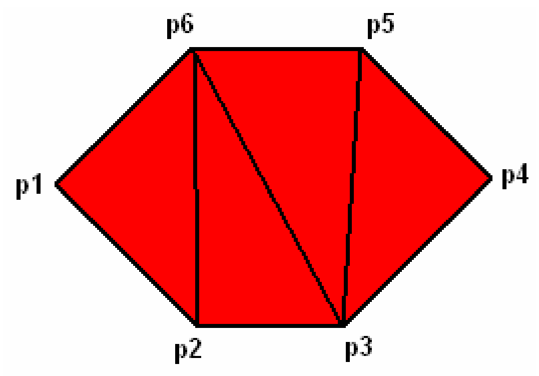


Figure 21 - Triangles Drawn Using the GL_TRIANGLES OpenGL Primitive

GL_TRIANGLE_STRIP

To form polygonal meshes it is often convenient to define a number of triangles using a single *glBegin ... glEnd* pair. The *GL_TRIANGLE_STRIP* primitive enables us to define a strip of connected triangles. The vertices of the *first* triangle only must be specified in anti-clockwise order. Figure 22 illustrates the use of GL_TRIANGLE_STRIP.

```
glBegin(GL_TRIANGLE_STRIP);
```



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

Figure 22 - Triangles Drawn Using the GL_TRIANGLE_STRIP OpenGL Primitive

GL_TRIANGLE_FAN

GL_TRIANGLE_FAN is similar to *GL_TRIANGLE_STRIP*, in that it allows us to define a number of triangles using a single *glBegin … glEnd* command. In this case, we can define a 'fan' of triangles emanating from a single point. The first point of the first triangle is the source of the fan, and the vertices of this triangle must be specified in an anti-clockwise direction. In addition, the fan of triangles must also be defined in an anti-clockwise direction. See Figure 23 for an example of the use of *GL_TRIANGLE_FAN*.

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

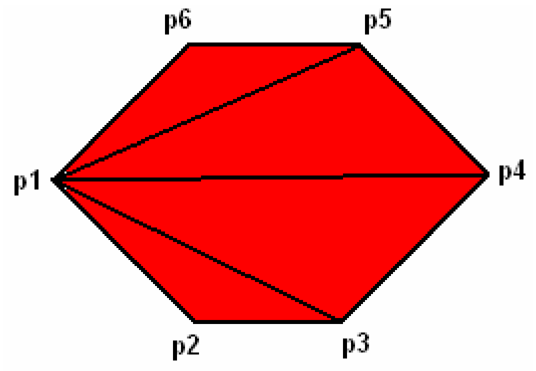


Figure 23 - Triangles Drawn Using the GL_TRIANGLE_FAN OpenGL Primitive

GL_QUADS

Using the *GL_QUADS* primitive, the vertex list is treated as groups of four vertices, each of which forms a *quadrilateral*. If the number of vertices specified is not a multiple of four, then the extra vertices are ignored. The vertices for each quadrilateral must be defined in an anti-clockwise direction. See Figure 24 for an example of *GL_QUADS*.

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

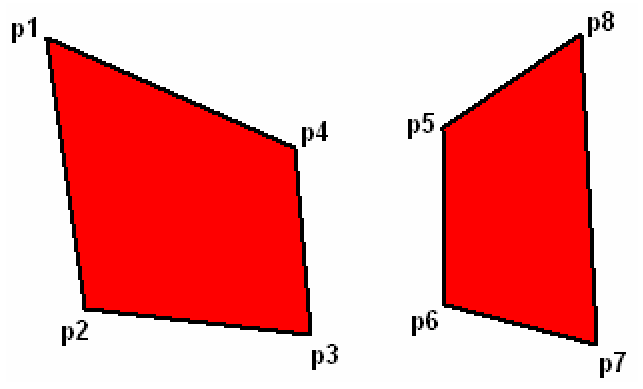


Figure 24 - Quadrilaterals Drawn Using the GL_QUADS OpenGL Primitive

GL_QUAD_STRIP

In the same way that *GL_TRIANGLE_STRIP* allowed us to define a strip of connected triangles, *GL_QUAD_STRIP* allows us to define a strip of quadrilaterals. The first four vertices form the first quadrilateral, and each subsequent pair of vertices is combined with the two before them to form another quadrilateral. The vertices of the first quadrilateral must be specified in an anti-clockwise direction. Figure 25 illustrates the use of *GL_QUAD_STRIP*.

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

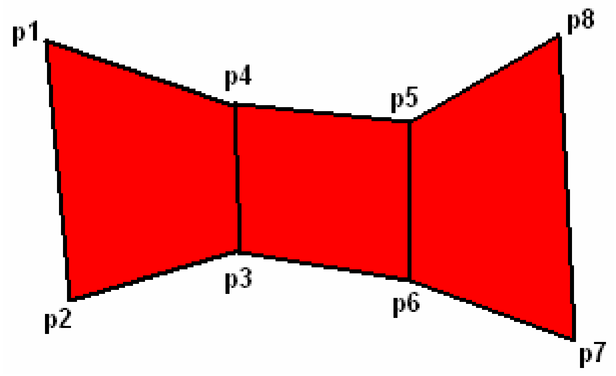


Figure 25 - Quadrilaterals Drawn Using the GL_QUAD_STRIP OpenGL Primitive

7. Character Primitives

The final type of graphics primitive we will consider is the *character primitive*. Character primitives can be used to display text characters. Before we examine how to display characters in OpenGL, let us consider some basic concepts about text characters.

We can identify two different types of representation for characters: *bitmap* and *stroke* (or *outline*) representations. Using a bitmap representation (or *font*), characters are stored as a grid of pixel values (see Figure 26(a)). This is a simple representation that allows fast rendering of the character. However, such representations are not easily *scalable*: if we want to draw a larger version of a character defined using a bitmap we will get an *aliasing* effect (the edges of the characters will appear jagged due to the low resolution). Similarly, we cannot easily generate bold or italic versions of the character. For this reason, when using bitmap fonts we normally need to store multiple fonts to represent the characters in different sizes/styles etc.

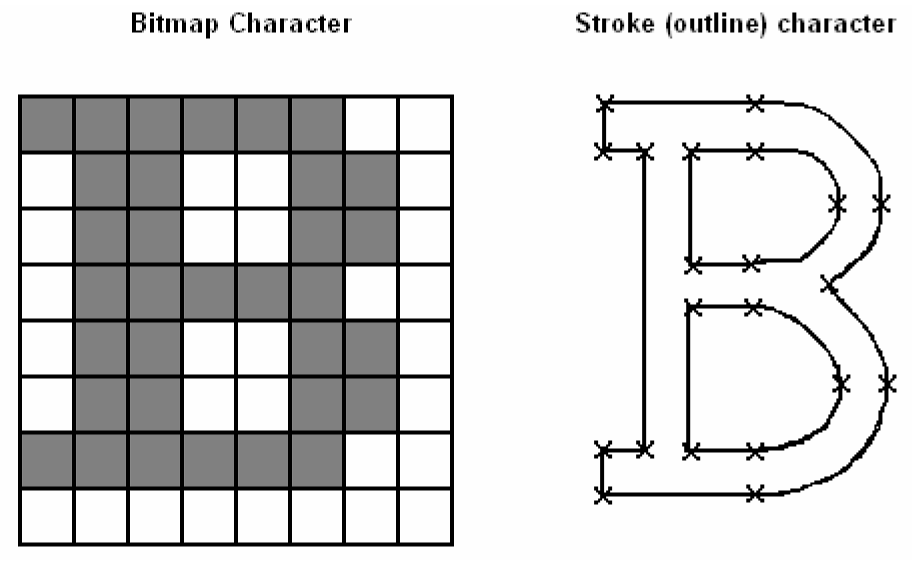
An alternative representation to bitmaps is the *stroke*, or *outline*, representation (see Figure 26(b)). Using stroke representations characters are stored using line or curve primitives. To draw the character we must convert these primitives into pixel values on the display. As such, they are much more easily scalable: to generate a larger version of the character we just multiply the coordinates of the line/curve primitives by some scaling factor. Bold and italic characters can be generated using a similar approach. The disadvantage of stroke fonts is that they take longer to draw than bitmap fonts.

We can also divide character primitives into *serif* and *sans-serif* fonts. Sans-serif fonts have no *accents* on the characters, whereas serif fonts do have accents. For example, as shown below, Arial and Verdana fonts are examples of sans-serif fonts whereas Times-Roman and Garamond are serif fonts:

Arial is a sans-serif font
Verdana is a sans-serif font
Times Roman is a serif font
Garamond is a serif font

Finally, we can categorise fonts as either *monospace* or *proportional* fonts. Characters drawn using a monospace font will always take up the same width on the display, regardless of which character is being drawn. With proportional fonts, the width used on the display will be proportional to the actual width of the character, e.g. the letter ‘i’ will take up less width than the letter ‘w’. As shown below, Courier is an example of a monospace font whereas Times-Roman and Arial are examples of proportional fonts:

Courier is a monospace font
Times-Roman is a proportional font
Arial is a proportional font



(a) **(b)**

Figure 26 - Bitmap and Stroke Character Primitives

8. OpenGL Character Primitive Routines

OpenGL on its own does not contain any routines dedicated to drawing text characters. However, the *glut* library does contain two different routines for drawing individual characters (not strings). Before drawing any character, we must first set the *raster position*, i.e. where will the character be drawn. We need to do this only once for each sequence of characters. After each character is drawn the raster position will be automatically updated ready for drawing the next character. To set the raster position we use the *glRasterPos2i* routine. For example,

glRasterPos2i(x, y)

positions the raster at coordinate location *(x,y)*.

Next, we can display our characters. The routine we use to do this will depend on whether we want to draw a bitmap or stroke character. For bitmap characters we can write, for example,

glutBitmapCharacter(GLUT_BITMAP_9_BY_15, 'a');

This will draw the character 'a' in a monospace bitmap font with width 9 and height 15 pixels. There are a number of alternative symbolic constants that we can use in place of *GLUT_BITMAP_9_BY_15* to specify different types of bitmap font. For example,

- *GLUT_BITMAP_8_BY_13*
- *GLUT_BITMAP_9_BY_15*

We can specify proportional bitmap fonts using the following symbolic constants:

- *GLUT_BITMAP_TIMES_ROMAN_10*

- *GLUT_BITMAP_HELVETICA_10*

Here, the number represents the height of the font.

Alternatively, we can use stroke fonts using the *glutStrokeCharacter* routine. For example,

glutStrokeCharacter(GLUT_STROKE_ROMAN, ‘a’);

This will draw the letter ‘a’ using the Roman stroke font, which is a proportional font. We can specify a monospace stroke font using the following symbolic constant:

- GLUT_STROKE_MONO_ROMAN

Summary

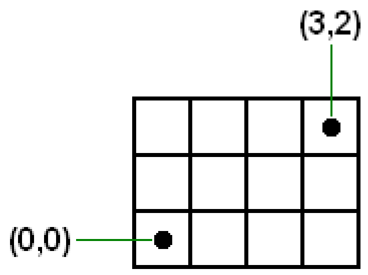
The following points summarise the key concepts of this chapter:

- *Graphics primitives* are the basic building blocks of images.
- *Geometric primitives* describe the *geometry*, or shape, of these building blocks.
- The *GL_POINTS* primitive can be used to draw points in OpenGL.
- The *DDA (Digital Differential Analyser)* line-drawing algorithm computes successive points by iteratively adding the x and y offsets δx and δy onto the previously computed location.
- Bresenham's line-drawing algorithm is the most efficient line-drawing algorithm, and determines computes successive points by computing a *decision function*, which can be calculated using integer calculations only.
- The *GL_LINES* primitive can be used to draw separate line segments in OpenGL.
- The *GL_LINE_STRIP* and *GL_LINE_LOOP* primitives can be used to draw *polylines* (connected line segments) in OpenGL.
- Circles can be drawn by plotting Cartesian or polar coordinates.
- The efficiency of any circle-drawing algorithm can be improved by taking advantage of the four-way symmetry of circles: for each point we compute, we can generate seven more through symmetry.
- The *midpoint algorithm* is the most efficient circle-drawing algorithm, and works in a similar way to Bresenham's line-drawing algorithm: successive points are determined by computing a *decision function*, which can be calculated using integer calculations only.
- The midpoint algorithm can also be adapted to work with *ellipses*, but ellipses have only two-way symmetry, so drawing them is less efficient than drawing circles.
- A *fill-area primitive* is any boundary that can be filled. Most fill-area primitives are polygons.
- Polygon primitives are typically used to construct 3-D objects in the form of a *polygonal mesh*.
- Polygonal meshes are normally represented using a *geometric table*, which consists of a *vertex table*, an *edge table* and a *surface-facet table*.
- A *simple polygon*, or *standard polygon*, is one with no edge crossings.
- A *degenerate polygon* is one that has less than three vertices, collinear vertices, non-coplanar vertices or repeated vertices. Degenerate polygons may not be displayed correctly by graphics packages.
- A *convex* polygon is one in which all interior angles are less or equal to 180^o. If a polygon has at least one interior angle greater than 180^o it is a *concave* polygon.
- Concave/convex polygons can be identified by taking the *cross-product* of adjacent pairs of edge vectors. If one cross-product has a different sign for its z-coordinate, the polygon is concave.
- The *vector technique* can be used to split concave polygons into a number of convex polygons.
- An *inside-outside* test determines if a given point is inside or outside a polygon boundary.
- Common inside-outside tests are the odd-even rule and the nonzero winding number rule.
- The *front-face* of a polygon is normally the face that is pointing out from the surface of the polygonal mesh.

- The *back-face* of a polygon points in to the interior of the object.
- The front-face of a polygon is normally identified by a *surface normal vector*.
- Surface normal vectors can be computed by taking the cross-product of any two adjacent edge vectors.
- OpenGL requires that all polygons are convex polygons.
- The *glRect** routine can be used to draw 2-D rectangles in OpenGL.
- The *GL_POLYGON*, *GL_TRIANGLES* and *GL_QUADS* and primitives can be used to draw fill-area polygons in OpenGL
- The *GL_TRIANGLE_STRIP*, *GL_TRIANGLE_FAN* and *GL_QUAD_STRIP* primitives can be used to draw connected sets of fill-area polygons in OpenGL.
- Character primitives can be drawn using either *bitmap* or *stroke* fonts:
    - Bitmap fonts represent characters using a grid of pixel values. They are efficient to draw but not easily scalable.
    - Stroke (or *outline*) fonts represent characters using line or curve primitives. They are less efficient to draw but are easily scalable.
- Fonts can be either *serif* or *sans-serif*: serif fonts feature accents at the ends of the lines whereas sans-serif fonts do not.
- Fonts can be either *monospace* or *proportional*: monospace fonts use the same width for each character whereas proportional fonts use a width proportional to the width of the character.
- The *glutBitmapCharacter* routine can be used to draw a bitmap character primitive in OpenGL.
- The *glutStrokeCharacter* routine can be used to draw a stroke character primitive in OpenGL.

Exercises

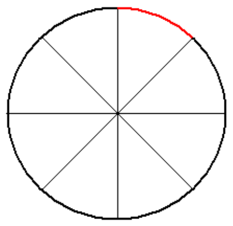
1) The figure below shows the start and end points of a straight line.



   a. Show how the DDA algorithm would draw the line between the two points.

   b. Show how Bresenham's algorithm would draw the line between the two points.

2) Show how the following circle-drawing algorithms would draw a circle of radius 5 centred on the origin. You need only consider the upper-right octant of the circle, i.e. the arc shown in red in the figure below.

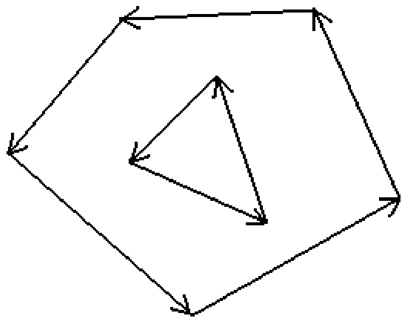


   a. Plotting points using Cartesian coordinates.

   b. Plotting points using polar coordinates.

   c. The midpoint algorithm.

3) Show which parts of the fill-area primitive shown below would be classified as *inside* or *outside* using the following inside-outside tests:

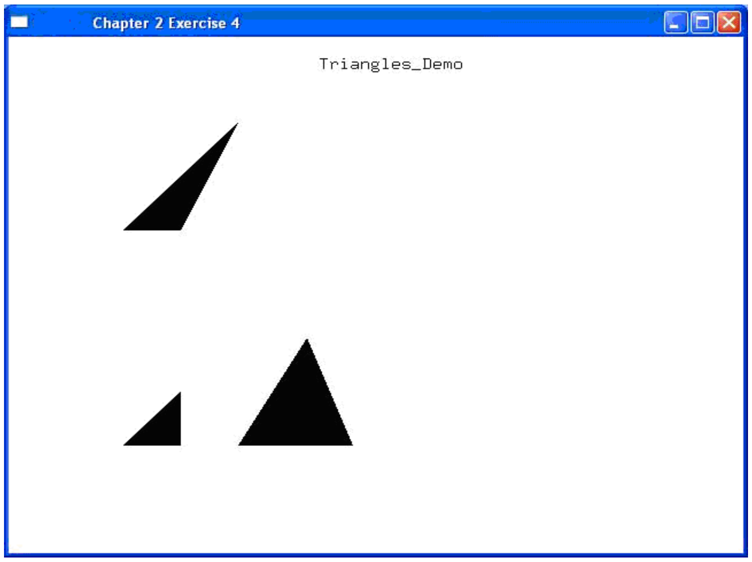


a. Odd-even rule.

b. Nonzero winding number rule.

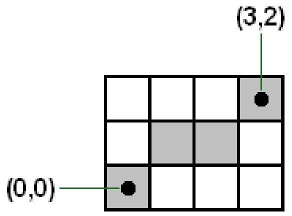
4) Write a C++/OpenGL program to read in a string and a sequence of points from a file, and display the string as a title and the points as triangles using the GL_TRIANGLES primitive. For example, the file listed below would lead to the following image being displayed.

```
Triangle_Demo
150 100
150 150
100 100
200 100
300 100
260 200
150 300
200 400
100 300
```



Exercise Solutions

1) The final line would be as shown below:



a. First we calculate the gradient:

$$m = \frac{(y_{end} - y_0)}{(x_{end} - x_0)} = \frac{(2-0)}{(3-0)} = \frac{2}{3} = 0.67$$

Now, because $|m| \leq 1$, we compute δx and δy as follows:

$\delta x = 1$

$\delta y = 0.67$

Using these values of δx and δy we can now start to plot line points:

- Start with (x_0,y_0) = (0,0) – colour this pixel
- Next, (x_1,y_1) = (0+1,0+0.67) = (1,0.67) – so we colour pixel (1,1)
- Next, (x_2,y_2) = (1+1,1 +0.67) = (2,1.33) – so we colour pixel (2,1)
- Next, (x_3,y_3) = (2+1,1 +0.67) = (3,2) – so we colour pixel (3,2)
- We have now reached the end-point (x_{end},y_{end}), so the algorithm terminates

b. First, compute the following values:

- $\Delta x = 3$
- $\Delta y = 2$
- $2\Delta y = 4$
- $2\Delta y - 2\Delta x = -2$
- $p_0 = 2\Delta y - \Delta x = 2 \times 2 - 3 = 1$

Plot (x_0,y_0) = **(0,0)**

Iteration 0:

- $p_0 \geq 0$, so
  - Plot $(x_1,y_1) = (x_0+1,y_0+1)$ = **(1,1)**
  - $p_1 = p_0 + 2\Delta y - 2\Delta x = 1 - 2 = -1$

Iteration 1:

- $p_1 < 0$, so
  - Plot $(x_2,y_2) = (x_1+1,y_1)$ = **(2,1)**
  - $p_2 = p_1 + 2\Delta y = -1 + 4 = 3$

Iteration 2:

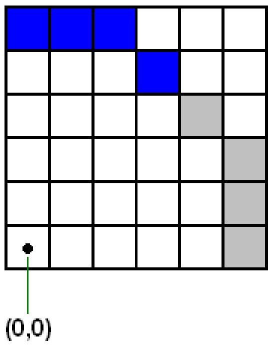
- $p_2 \geq 0$, so
  - Plot $(x_3,y_3) = (x_2+1,y_2+1)$ = **(3,2)**
  - $p_3 = p_2 + 2\Delta y - 2\Delta x = 3 - 2 = 1$

2) The answers are:

a. We start from $x = 0$, and then successively increment $x$and calculate the corresponding y using Eq. (17): $y = y_c \pm \sqrt{r^2 - (x_c - x)^2}$. We can tell that we have left the first octant when $x > y$.

- $x = 0$, $y = 0 + \sqrt{5^2 - (0-0)^2} = 5$, so we plot **(0,5)**.
- $x = 1$, $y = 0 + \sqrt{5^2 - (0-1)^2} = 4.9$, so we plot (4.9,1) = **(1,5)**.
- $x = 2$, $y = 0 + \sqrt{5^2 - (0-2)^2} = 4.58$, so we plot (4.58,2) = **(2,5)**.
- $x = 3$, $y = 0 + \sqrt{5^2 - (0-3)^2} = 4$, so we plot **(3,4)**.
- $x = 4$, $y = 0 + \sqrt{5^2 - (0-4)^2} = 3$, so $x > y$ and we stop.

The figure below shows the first quadrant of the circle, with the plotted points shown in blue, and points generated by symmetry in grey.



b. First we calculate the angular increment using θ = 1/r radians. This is equal to (1/5)*(180/π) = 11.46°. Then we start with θ = 90°, and compute x and y according to Eqs. (15) and (16) for successive value of θ, subtracting 11.46° at each iteration. We stop when θ becomes less than 45°.

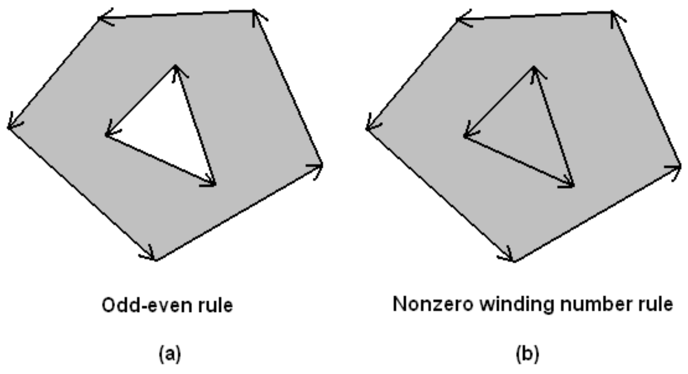
- θ = 90°, $x = 0 + 5\cos 90^o = 0$, $y = 0 + 5\sin 90^o = 5$, so we plot **(0,5)**
- θ = 78.54°, $x = 0 + 5\cos 78.54^o = 0.99$, $y = 0 + 5\sin 78.54^o = 4.9$, so we plot **(1,5)**
- θ = 67.08°, $x = 0 + 5\cos 67.08^o = 1.95$, $y = 0 + 5\sin 67.08^o = 4.6$, so we plot **(2,5)**
- θ = 55.62°, $x = 0 + 5\cos 55.62^o = 2.82$, $y = 0 + 5\sin 55.62^o = 4.13$, so we plot **(3,4)**
- θ = 44.16°, which is less than 45°, so we stop.

The points plotted are the same as for the Cartesian plotting algorithm.

c. The steps of the algorithm are:

- Plot the first circle point $(0,r)$ = **(0,5)**
- Compute the first value of the decision variable $p_0 = 1 - r = -4$
- Iteration 0:
  - $p_0 < 0$, so we plot (x_0+1, y_0) = **(1,5)**
  - $p_1 = p_0 + 2x_1 + 1 = -4 + 2\times 1 + 1 = -1$
- Iteration 1:
  - $p_1 < 0$, so we plot (x_1+1, y_1) = **(2,5)**
  - $p_2 = p_1 + 2x_1 + 1 = -1 + 2\times 2 + 1 = 4$
- Iteration 2:
  - $p_2 \geq 0$, so we plot (x_2+1, y_2-1) = **(3,4)**
  - $p_3 = p_2 + 2x_3 + 1 - 2y_3 = 4 + 2\times 3 + 1 - 2\times 4 = 3$
- Iteration 3:
  - $p_3 \geq 0$, so we plot (x_3+1, y_3-1) = **(4,3)**
  - At this point, $x > y$ so we have left the octant and the algorithm stops.

3) The figure below shows the classifications for each of the algorithms: the grey shaded areas are classified as *inside*, and the white areas as *outside*. In this case the two algorithms produce different results. The odd-even rule classifies the inner polygon as outside because points inside it have two (an even number) line crossings to reach any distant point. For the nonzero winding number rule, both of the line crossings go from right to left, so the winding number is incremented in both cases. Therefore the total winding number for points inside the inner polygon is 2, which in nonzero and so the points are classified as inside. By reversing the direction of the edge vectors of the inner (or outer) polygon we could get the same result as the odd-even rule.



4) See the code listing included in the zip file for this handout for the solution to this question.

```
Source: Computer Graphics with OpenGL,
        Donald Hearn and M. Pauline Baker,
        Prentice-Hall, 2004 (Third Edition)
```