ADDIS ABABA SCIENCE AND TECHNOLOGY UNIVERSITY

Department of software engineering

Course: Computer Graphics

Course Code: SWEG4109

Name: Andinet Dereje

ID NO: ETS0198/14

Section: A

Submitted to: inst. Lilise Daniel

Submission date: - Dec 05, 2024

Contents

1.0 Geometry

Geometry is the mathematical representation and manipulation of objects and their spatial relationships. It forms the foundation for rendering scenes and visualizing models in 2D and 3D.

Geometric Primitives

- 2D Primitives: Points, lines, circles, polygons, and curves.
- 3D Primitives: Cubes, spheres, cylinders, cones, and other polyhedral shapes.

1.1 2D primitives

Points

A point is the most basic entity in geometry, representing a position without any dimensions—it has zero length, width, or height. Points lying on the same straight line are called collinear points, while those that do not are known as non-collinear points. A line is a straight path consisting of infinitely many points and extends endlessly in both directions. It has infinite length but no width or height, and its attributes, such as color, width, and style (e.g., solid or dashed), can be customized. A plane is a two-dimensional surface with length and width but no height, stretching infinitely in all directions. It is made up of an infinite number of lines. The midpoint of a segment is the point that divides the segment into two equal parts, resulting in two congruent segments.

1.2 3D primitives

Polygon

A polygon is a two-dimensional geometric figure that has a finite number of sides. The sides of a polygon are made of straight line segments connected to each other end to end. Thus, the line segments of a polygon are called sides or edges.

2.0 Line Generation

Line generation is a fundamental operation in raster graphics. It involves converting a line's mathematical representation into pixels on a screen. The Line drawing algorithm is a graphical algorithm which is used to represent the line segment on discrete graphical media, i.e., printer and pixel-based media. The point is an important element of a line.

2.1 Line Drawing Algorithm

Properties of a good line drawing algorithm include:

- An algorithm should be precise: Each step of the algorithm must be adequately defined.

- Finiteness: An algorithm must contain finiteness. It means the algorithm stops after the execution of all steps.
- Easy to understand: An algorithm must help learners to understand the solution in a more natural way.
- Correctness: An algorithm must be in the correct manner.
- Effectiveness: The steps of an algorithm must be valid and efficient.
- Uniqueness: All steps of an algorithm should be clearly and uniquely defined, and the result should be based on the given input.
- Input: A good algorithm must accept at least one or more input.
- Output: An algorithm must generate at least one output.

Some Line Drawing Algorithms Include:

- DDA (Digital Differential Analyzer) Line Drawing Algorithm
- Brenham's Line Drawing Algorithm
- Mid-Point Line Drawing Algorithm

2.1.0 DDA Line Drawing Algorithms

Digital Differential Analyzer (DDA) algorithm is the simple line generation algorithm which is explained step by step here.

Step 1 − Get the input of two end points

Step 2 − Calculate the difference between two end points.

$$dx = X_1 - X_0 \quad dy = Y_1 - Y_0$$

Step 3 − Based on the calculated difference in step-2, we need to identify the number of steps to put pixel. If $dx > dy$, then you need more steps in x coordinate; otherwise in y coordinate.

Step 4 − Calculate the increment in x coordinate and y coordinate.

Step 5 − Put the pixel by successfully incrementing x and y coordinates accordingly and complete the drawing of the line.

2.1.1 Bresenham's Line Generation

The Bresenham algorithm is another incremental scan conversion algorithm. The big advantage of this algorithm is that, it uses only integer calculations. Moving across the x axis in unit intervals and at each step choose between two different y coordinates.

Given the coordinate of two points A(x1, y1) and B(x2, y2). The task is to find all the intermediate points required for drawing line AB on the computer screen of pixels. Note that every pixel has integer coordinates.

- Step 1: Calculate ΔX and ΔY from the given input. These parameters are calculated as

  $\Delta X = X_n - X_0$

  $\Delta Y = Y_n - Y_0$

- Step 2: Calculate the decision parameter Pk. $P_k = 2\Delta Y - \Delta X$
- Step 3: Suppose the current point is (Xk, Yk) and the next point is (Xk+1, Yk+1). Find the next point depending on the value of decision parameter Pk.

Two cases

Case 1: $P_k < 0$

$P_{k+1} = P_k + 2\Delta y$

$X_{k+1} = X_K + 1$

$Y_{K+1} = Y_K$

Case 2: $P_K >= 0$

$P_{K+1} = P_k + 2\Delta Y - 2\Delta X$

$X_{k+1} = X_k + 1$

$Y_{k+1} = Y_{k+1}$

- Step 4: Keep repeating Step-03 until the end point is reached or number of iterations equals to (ΔX-1) times.

Example:

- Starting coordinates = (X_0, Y_0) = (9, 18)
- Ending coordinates = (X_n, Y_n) = (14, 22)

$\Delta X = Xn - X0 = 14 - 9 = 5$

$\Delta Y = Yn - Y0 = 22 - 18 = 4$

$P_k = 2\Delta Y - \Delta X$

$= 2 \times 4 - 5$

$= 3$

So, decision parameter $P_k = 3$

As $P_k >= 0$, so case-02 is satisfied.

Thus,

- $P_{k+1} = P_k + 2\Delta Y - 2\Delta X = 3 + (2 \times 4) - (2 \times 5) = 1$
- $X_{k+1} = X_k + 1 = 9 + 1 = 10$
- $Y_{k+1} = Y_k + 1 = 18 + 1 = 19$

(Number of iterations = $\Delta X - 1 = 5 - 1 = 4$)

P_k	P_{k+1}	X_{k+1}	Y_{k+1}
3	1	10	19
1	-1	11	20
-1	7	12	20
7	5	13	21
5	3	14	22

2.1.2 Mid-Point Line Drawing Algorithm

The Mid-Point Line Drawing Algorithm is a powerful technique used to draw a line between two points on a pixelated computer screen. It calculates the intermediate points required to represent the line with integer pixel coordinates. This algorithm offers high accuracy and efficiency while relying on simple arithmetic operations.

- Step 1: Calculate ΔX and ΔY from the given input. These parameters are calculated as:

  $\Delta X = X_n - X_0$

  $\Delta Y = Y_n - Y_0$

- Step 2: Calculate the value of initial decision parameter and ΔD. These parameters are calculated as

  $Dinitial = 2\Delta Y - \Delta X$

  $\Delta D = 2(\Delta Y - \Delta X)$

- Step 3: The decision whether to increment X or Y coordinate depends upon the flowing values of $D_{initial}$.

  Case 1: $D_{initial} < 0$

  $X_{k+1} = X_K + 1$

$Y_{K+1} = Y_K$

Dnew= Dinitial + 2ΔY

Case 2: $D_{initial} >= 0$

$X_{k+1} = X_K + 1$

$Y_{K+1} = Y_K + 1$

Dnew = Dinitial + ΔD

- Step 4: Keep repeating Step 3 until the end point is reached.

Example: Given

Starting coordinates = $(X_0, Y_0) = (20, 10)$

Ending coordinates = $(X_n, Y_n) = (30, 18)$

- $\Delta X = X_n – X_0 = 30 – 20 = 10$
- $\Delta Y = Y_n – Y_0 = 18 – 10 = 8$

Calculate $D_{initial}$ and ΔD as-

- $D_{initial} = 2\Delta Y – \Delta X = 2 \times 8 – 10 = 6$
- $\Delta D = 2(\Delta Y – \Delta X) = 2 \times (8 – 10) = -4$

As $D_{initial} >= 0$, so case-02 is satisfied.

- $X_{k+1} = X_k + 1 = 20 + 1 = 21$
- $Y_{k+1} = Y_k + 1 = 10 + 1 = 11$
- $D_{new} = D_{initial} + \Delta D = 6 + (-4) = 2$

Dinitial	Dnew	Xk+1	Yk+1
6	2	21	11
2	-2	22	12
-2	14	23	12
14	10	24	13
10	6	25	14
6	2	26	15
2	-2	27	16
-2	14	28	16
14	10	29	17
10		30	18

3.0 Fill Area Primitives

Filled Area primitives are used to filling solid colors to an area or image or polygon. Filling the polygon means highlighting its pixel with different solid colors. Any fill-area shape is possible, graphics libraries generally do not support specifications for arbitrary fill shapes. Graphics routines can more efficiently process polygons than other kinds of fill shapes because polygon boundaries are described with linear equations. When lighting effects and surface-shading procedures are applied, an approximated curved surface can be displayed quite realistically. Approximating a curved surface with polygon facets is sometimes referred to as surface tessellation, or fitting the surface with a polygon mesh. Objects described with a set of polygon surface patches are usually referred to as standard graphics objects, or just graphics objects.

3.1 Polygon Filling

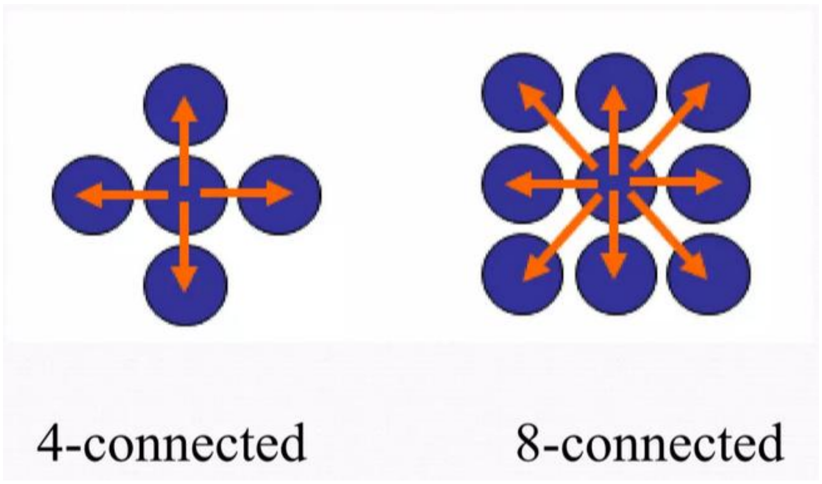
Filling the polygon means highlighting all the pixels which lie inside the polygon with any color other than background colour. Polygons are easier to fill since they have linear boundaries. There are two basic approaches used to fill the polygon. One way to fill a polygon is to start from a given "seed", point known to be inside the polygon and highlight outward from this point i.e. neighboring pixels until we encounter the boundary pixels. This approach l, called seed fill because color flows from the seed pixel until reaching the polygcn boundary, like water flooding on the surface of the container Another approach to fill the polygon is to apply the inside test i.e. to check whether the pixel is inside the polygon or outside the polygon and then highlight pixels which lie inside the polygon". This approach is known as scan-line algorithm. It avoids the need for a seed pixel but it requires some computation.

3.1.0 Seed Fill

The seed fill algorithm is further classified as flood fill algorithm and boundary fill algorithm. Algorithms that fill interior-defined regions are called flood-fill algorithms; those that fill boundary-defined regions are called boundary-fill algorithms edge-fill algorithms.

1 Boundary Fill Algorithm / Edge Fill Algorithm

In this method, edges of the polygons are drawn. Then starting with some any point inside the polygon we examine the neighboring pixels to check whether the boundary pixel is reached. If boundary pixels are not reached, pixels highlighted and the process is continued until boundary pixels are reached. Boundary defined regions may be either 4-cormected or 8-connected.



In some cases, an 8-connected algorithm is more accurate than the 4-connected algorithm. This is illustrated in bellow Figure. Here, a 4-connected algorithm produces the partial fill.

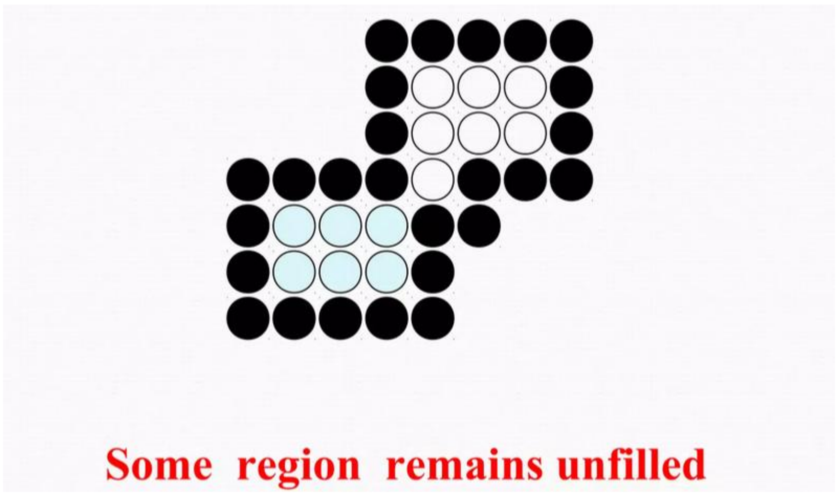


Figure: partial filing resulted using 4-connected algorithm

Algorithm

```
Function boundaryfill(x, y, color, color1)
int c;
c = getpixel(x ,y);
if (c! =color) & (c!=color1)
{
    setpixel (x, y, color);
   boundaryfill( x+1, y, color, color1);
   boundaryfill( x-1, y, color, color1);
   boundaryfill( x, y+1, color, color1);
```

```
  boundaryfill( x, y-1, color, color1);
}
```

Advantages**:**

- The boundary fill algorithm is used to create attractive paintings.

Disadvantages:

- In the 4-connected approach, it does not color corners.

3.1.1 Flood Fill Algorithm

Sometimes it is required to fill in an area that is not defined within a single colour boundary. In such cases we can fill areas by replacing a specified interior colour instead of searching for a boundary colour. This approach is called a flood-fill algorithm. The flood fill algorithm is used when the polygon has multiple color boundaries. Like boundary fill algorithm, here we start with some seed and examine the neighboring pixels. However, here pixels are checked for a specified interior colour instead of boundary colour and they are replaced by new colour. Using either a 4-connected or 8-connected approach, we can step through pixel positions until all interior point have been filled.

3.1.1.0 Algorithm

```
Function floodfill(x, y, fillcolor, previouscolor)
if (getpixel (x, y) = previouscolor)
{
   setpixel (x, y, fillcolor);
   floodfill( x+1, y, fillcolor,previouscolor);
   floodfill( x-1, y, fillcolor,previouscolor);
   floodfill( x, y+1, fillcolor,previouscolor);
   floodfill( x, y-1, fillcolor,previouscolor);
}
```

getpixel() - The color of specified pixel.

setpixel() - Sets the pixel to specified color.

Advantages:

- This method is easy to fill colors in computer graphics.
- It fills the same color inside the boundary.

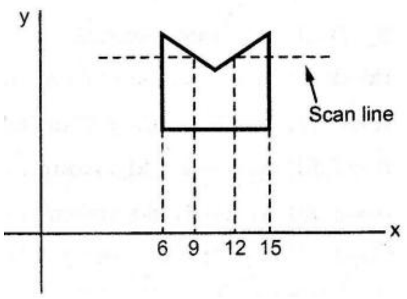
Disadvantages:

- Fails with large area polygons.
- It is a slow method to fill the area.

3.1.2 Scan fill Algorithm

Scan fill algorithm is an area-filling algorithm that fill colors by scanning horizontal lines. Recursive algorithm for seed fill methods have got two difficulties: The first difficulty is that if some inside pixels are already displayed in fill colour then recursive branch terminates, leaving further internal pixels unfilled. To avoid this difficulty, we have to first change the colour of any internal pixels that are initially set to the fill colour before applying the seed fill procedures. Another difficulty with recursive seed fill methods is that it cannot be used for large polygons. This is because recursive seed fill procedures require stacking of neighboring points and in case of large polygons stack space may be insufficient for stacking of neighboring points. To avoid this problem more efficient method can be used. Such method fills horizontal pixel spans across scan lines, instead of proceeding to 4-connected or 8-connected neighboring points. This is achieved by identifying the rightmost and leftmost pixels of the seed pixel and then drawing a horizontal line between these two boundary pixels. This procedure is repeated with changing the seed pixel above and below the line just drawn until complete polygon is filled. With this efficient method we have to stack only a beginning position for each horizontal pixel span, instead of stacking all unprocessed neighboring positions around the current position.

The Figure bellow illustrates the scan line algorithm for filling of polygon. For each scan line crossing a polygon, this algorithm locates the intersection points of the scan line with the polygon edges. These intersection points are then sorted from left to right, and the corresponding positions between each intersection pair are set to the specified fill colour.



In Figure above, we can see that there are two stretches of interior pixels from x = 6 to x = 9 and x = 72 to x = 15. The scan line algorithm first finds the largest and smallest y values of the polygon. It then starts with the largest y value and works its way down, scanning from left to right, in the manner of a raster display. The important task in the scan line algorithm is to find the intersection points of the scan line with the polygon boundary. When intersection points are even, they are sorted from left to right, paired and pixels between paired points are set to the fill colour. But in some cases intersection point is a vertex. When scan line intersects polygon vertex a special handling is required to find the exact intersection points. To handle such cases, we must look at the other endpoints of the two line segments of the polygon which meet at this vertex. If these points lie on the same (up or down) side of the scan line, then the point in question counts as an even number of intersections. If they lie on opposite sides of the scan line, then the point is counted as single intersection.

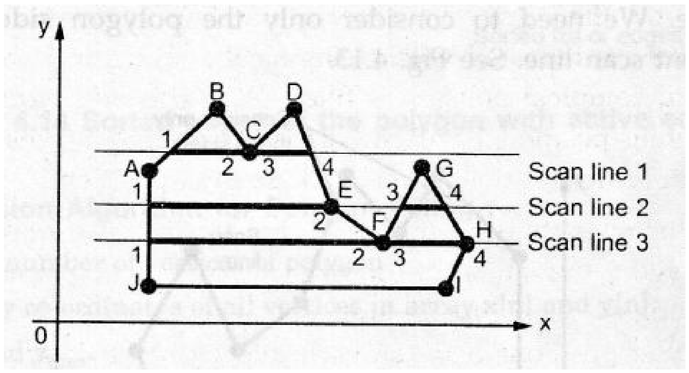


Figure: Intersection points along the scan line that intersect polygon vertices

As shown in the Figure above, each scan line intersects the vertex or vertices of the polygon. For scan line 1, the other end points (B and D) of the two line segments of the polygon lie on the same side of the scan line, hence there are two intersections resulting two pairs: 1 - 2 and 3 - 4. Intersections points 2 and 3 are actually same Points. For scan line 2 the other endpoints (D and F) of the two line segments of the Polygon lie on the opposite sides of the scan line, hence there is a single intersection resulting two pairs: 'l' - 2 and 3 - 4. For scan line 3, two vertices are the intersection points" For vertex F the other end points E and G of the two line segments of the polygon lie on the same side of the scan line whereas for vertex H, the other endpoints G and I of the two line segments of the polygon lie on the opposite side of the scan line. Therefore, at vertex F there are two intersections and at vertex H there is only one intersection. This results two pairs: 1 - 2 and 3 - 4 and points 2 and 3 are actually same points.

It is necessary to calculate x intersection points for scan line with every polygon side. We can simplify these calculations by using coherence properties. A coherence property of a scene is a proper\ of a scene by which we can relate one part of a scene with the other parts of a scene. Here, we can use a slope of an edge as a coherence property, by using this property we can determine the x intersection value on the lower scan line if the x intersection value. For current scan line is known. This is given as

$$X_{i+1} = X_i - 1/m$$

Where m is the slope of the edge

As we scan from top to bottom value of y coordinates between the two scan line changes by 1.

$$Y_{i+1} = Y_i - 1$$

Many times it is not necessary to compute the x intersections for scan line with every polygon side. We need to consider only the polygon sides with endpoints straddling the current scan line.

Scan Line Conversion Algorithm for Polygon Filling

1. **Read** n, the number of vertices of the polygon.
2. **Read** x and y-coordinates of all vertices into arrays x[n] and y[n]
3. **Find** ymin and ymax
4. **Store** the initial x-value (x1), y-values (y1 and y2) for the two endpoints, and the x-increment (dx) from scan line to scan line for each edge in the array edges[n][4].
    - While doing this, check that y1>y2 . If not, interchange y1 and y2 and the corresponding x1 and x2 , so that for each edge, y1 represents its maximum y-coordinate and y2 represents its minimum y-coordinate.
5. **Sort** the rows of the array edges[n][4] in descending order of y1 , descending order of y2 , and ascending order of x1.
6. **Set** y=ymax
7. **Find** the active edges and update the active edge list:

8. **Compute** the x-intersects for all active edges for the current y-value. Initially, the x-intersect is x1 . For successive y-values, the x-intersects can be calculated as:

   Xintersectnew=Xintersectold +Δx

   where Δx=x2−x1 / y2−y1

9. **Apply the vertex test** if the xxx-intersect is a vertex (i.e., xintersect=x1 and y=y1), to check whether to consider one intersect or two intersects. Store all x-intersects in the array x_intersect[].
10. **Sort** the x_intersect[] array in ascending order.
11. **Extract** pairs of intersects from the sorted x_intersect[] array.
12. **Pass** pairs of x-values to the line-drawing routine to draw the corresponding line segments.
13. **Set** y=y−1.
14. **Repeat** steps 7 through 13 until y<ymin.
15. **Stop**.

4.0 Curve Representation

In computer graphics, we often need to draw different types of objects onto the screen. Objects are not flat all the time and we need to draw curves many times to draw an object. A curve is an infinitely large set of points. Each point has two neighbors' except endpoints. Curves can be broadly classified into three categories − explicit, implicit, and parametric curves

4.1 Types of Curves

Implicit Curves: Implicit curve representations define the set of points on a curve by employing a procedure that can test to see if a point in on the curve. Usually, an implicit curve is defined by an implicit function of the form f x,y = 0

It can represent multivalued curves. A common example is the circle, whose implicit representation is

$X^2 + y^2 – R^2 = 0$

Explicit Curves : A mathematical function y = f can be plotted as a curve. Such a function is the explicit representation of the curve. The explicit representation is not general, since it cannot represent vertical lines and is also single-valued. For each value of x, only a single value of y is normally computed by the function

Parametric Curves: Curves having parametric form are called parametric curves. The explicit and implicit curve representations can be used only when the function is known. In practice the parametric curves are used. A two-dimensional parametric curve has the following form

P t = f t, g t or P t = x t, y t

The functions f and g become the coordinates of any point on the curve, and the points are obtained when the parameter t is varied over a certain interval [a, b], normally [0, 1].

Bezier Curves

Bezier curve is discovered by the French engineer Pierre Bézier. These curves can be generated under the control of other points. Approximate tangents by using control points are used to generate curve. The Bezier curve can be represented mathematically as

$$\sum_{k=0}^{n} P_i B_i^n(t)$$

Where is the set of points and

$$B_i^n(t) = \binom{n}{i}(1-t)^{n-i}t^i$$

Where n is the polynomial degree, i is the index, and t is the variable. The simplest Bézier curve is the straight line from the point P0 to P1. A quadratic Bezier curve is determined by three control points. A cubic Bezier curve is determined by four control points.

<u>5.0 Character Attributes, Primitives and representation</u>

Character Primitives

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. 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 16(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 categorize fonts as either moonscape or proportional fonts. Characters drawn using a moonscape 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.

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

OpenGL Character Attribute Functions

In OpenGL bitmap characters are drawn using the glutBitmapCharacter routine. For bitmap characters we can change the font by changing the first argument to this function, e.g. glutBitmapCharacter(GLUT_BITMAP_TIMES_ROMAN_10, ‘a’); will write the letter a in 10-point Times-Roman font, whereas glutBitmapCharacter(GLUT_BITMAP_HELVETICA_10, ‘a’); will draw the same letter in 10-point Helvetica font.

In addition, we can change the colour of the character by using the glColor routine, as we have seen before. To make other attribute changes, we must be using stroke character primitives. Stroke character primitives are drawn using the glutStrokeCharacter routine. Using this type of character primitive, we can change the font and colour as described above, and also the line width and the line style. Remember that stroke characters are stored as set of line primitives. Therefore we can change the width of these lines using the glLineWidth routine we introduced in Section 4.3. Similarly, we can change the style of the lines using the glLineStipple routine.

References

- https://www.geeksforgeeks.org/filled-area-primitives-computer-graphics/
- https://dspmuranchi.ac.in/pdf/Blog/Computer%20Graphics%20Curves.pdf
- https://www.studocu.com/row/document/jimma-university/computer-networking/chapter-3-attributes-of-graphics-primitives/64465688