Dire Dawa Institute of Technology
Department of Computer Science
Chapter 3 Handout – Graphics Primitives (part-II)
Attributes of graphics primitives

Compiled by Gaddisa Olani,2016

1. Introduction

In this handout we will consider the subject of *attributes* of graphics primitives. We can define an attribute of a primitive as a parameter that affects the way the primitive is displayed. For example, the drawing colour and line style are both attributes of the line primitive. We will examine what attributes there are for different primitives, and look at some algorithms for implementing them.

Recall from Chapter 1 that OpenGL is a *state system*: it maintains a list of current state variables that are used to modify the appearance of each primitive as it is displayed. Therefore these state variables represent the attributes of the primitives. The values of state variables remain in effect until new values are specified, and changes to state variables affects primitives drawn after the change.

First we will look at colour attributes, which affect almost all primitives. The rest of this chapter is structured according to the type of primitive that the attribute affects. We will look at attributes of point primitives, line primitives, fill-area primitives and character primitives. For each we will describe the underlying algorithms for implementing the change, and introduce how the attribute can be modified in OpenGL. Finally we will consider the related topic of *antialiasing*.

2. Colour Attribute

In our first OpenGL program in Chapter 1 we defined values for two colour state variables: the drawing colour and the background colour. We also saw that we can define colours in two different ways:

- *RGB*: the colour is defined as the combination of its red, green and blue components.
- *RGBA*: the colour is defined as the combination of its red, green and blue components together with an alpha value, which indicates the degree of opacity/transparency of the colour.

In the simple program in Chapter 1, the background colour was defined with an alpha value, whereas the drawing colour did not. If no alpha value is defined it is assumed to be equal to 1, which is completely opaque.

It can often be useful to combine colours of overlapping objects in the scene or of an object with the background colour. For example, this is necessary to achieve transparency effects and also for antialiasing (see Section 7). This process is known as *colour blending*, and the *RGBA* colour representation is necessary to achieve colour blending.

2.1. Direct Storage vs. Colour Look-Up Tables

Recall that the frame buffer is used by raster graphics systems to store the image ready for display on the monitor. Therefore, for colour displays, the frame buffer must have some way of storing colour values. We have just seen that colours can be stored as either *RGB* or *RGBA* values. Whichever representation is used, there are two ways in which graphics packages store colour values in a frame buffer:

- *Direct storage*
- *Look-up table*

Using direct storage, the colour values of each pixel are stored directly in the frame buffer (see Figure 1). Therefore, each pixel in the frame buffer must have 3 (*RGB*) or 4 (*RGBA*) values stored. This means that the frame buffer can take up a lot of memory. For example, suppose we have a screen resolution of 1024x768, and we use 8 bits for each of the red, green and blue components of the colours stored. The total storage required will be 1024x768x24 bits = 2.4MB.

Colour look-up tables are also illustrated in Figure 1. They try to reduce the amount of storage required by just storing an *index* for each pixel in the frame buffer. The actual colours are stored in a separate look-up table, and the index *looks up* a colour in this table. Therefore, using colour look-up tables we can only have a limited number of different colours in the scene (the number of rows in the look-up table), but these colours can be chosen from a palette containing the complete range of colours. For example, suppose we are using 8 bits for each colour (=24 bits in all), and the look-up table has 256 rows. In this case the total range of colours we can choose from is 2^{24}, but we can only use 256 of these in any one image.

Although colour look-up tables do save storage, they slow down the rasterisation process as an extra look-up operation is required. They were commonly used in the early days of computer graphics when memory was expensive, but these days memory is cheaper so most systems use direct storage. OpenGL uses direct storage by default, but the programmer can choose to use a colour look-up table if they wish.

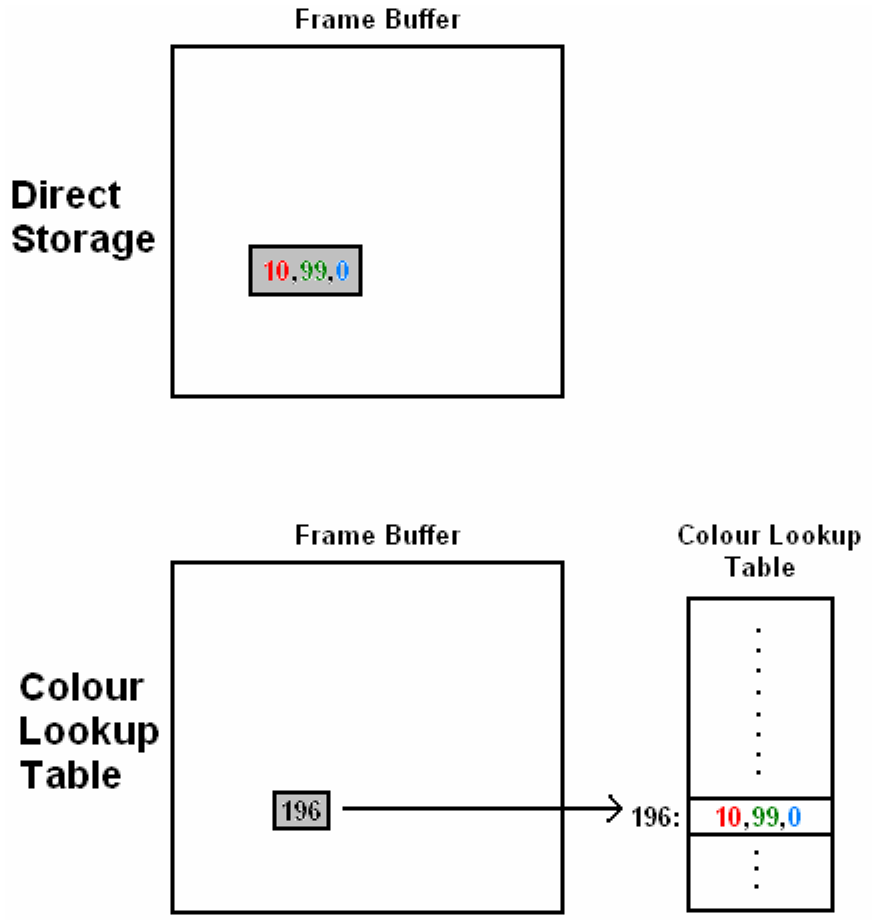


Figure 1 - Direct Storage vs. Colour Look-Up Tables

2.2. OpenGL Colour Attributes

We have already seen how to modify the drawing and background colours in the simple OpenGL program in Chapter 1. To recap, we change the current drawing colour using the *glColor** function and the current background colour using the *glClearColor* function. The background colour is always set using an *RGBA* colour representation (although, as we will see below, the alpha value is sometimes ignored). The drawing colour can be set using either *RGB* or *RGBA* representations. The following are all valid examples of setting the drawing and background colours:

```
glColor3f(1.0,0.0,0.0);            // set drawing colour to red
glColor4f(0.0,1.0,0.0,0.05);       // set drawing colour to semi-transparent green
glClearColor(1.0,1.0,1.0,1.0);     // set background colour to opaque white
```

Before we do any drawing in OpenGL, we must define a frame buffer. If we want to do colour drawing then we must specify that we want a colour frame buffer. Again, we saw how to do this in Chapter 1, but to recap the following line defines a single direct storage colour frame buffer with red, green and blue components:

glutInitDisplayMode(GLUT_SINGLE,GLUT_RGB)

In order to store alpha values in the frame buffer we must change the second argument to *GLUT_RGBA*. To specify that we want to use a colour look-up table we set the second argument equal to *GLUT_INDEX*.

3. Point Attributes

Points are the simplest type of primitive so the only attributes we can modify are the colour and size of the point. The simple program given in Chapter 1 showed how to modify each of these state variables in OpenGL. To change the size of points in pixels we use:

glPointSize(size)

Points are drawn in the current drawing colour. Section 2.2 above showed how to change the drawing colour. All points in OpenGL are drawn as squares with a side length equal to the point size. The default point size is 1 pixel.

4. Line Attributes

The attributes of line primitives that we can modify include the following:

- Line colour
- Line width
- Line style (solid/dashed etc)

We have already seen how to change the drawing colour. In the following sections we examine the line width and line style attributes.

4.1. Line Width

The simplest and most common technique for increasing the width of a line is to plot a line of width 1 pixel, and then add extra pixels in either the horizontal or vertical directions. This concept is illustrated in Figure 2. We can see that for some lines (e.g. Figure 2(a)) we should plot extra pixels vertically, and for others (e.g. Figure 2(b)) we should plot them horizontally. Which of these two approaches we use depends on the gradient m of the line. We identify the following cases:

- If $|m| \leq 1$ plot extra pixels vertically.
- If $|m| > 1$, plot extra pixels horizontally.

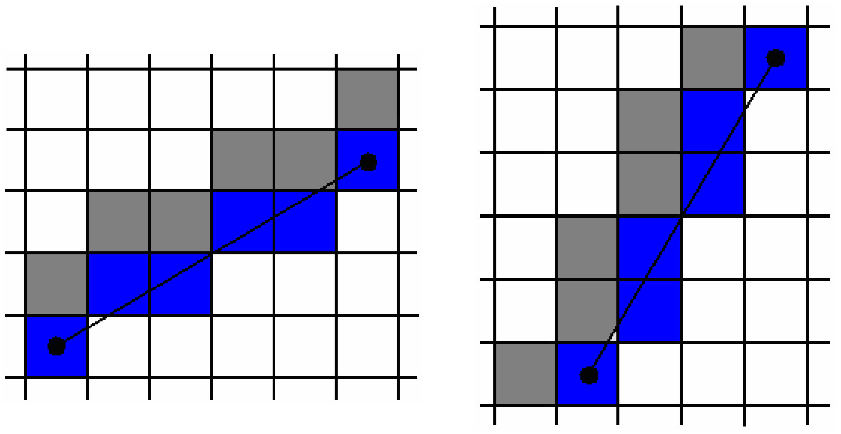


Figure 2 - Increasing Line Width by Plotting Extra Pixels in the Vertical (a) and Horizontal (b) Directions

Although this approach is simple and effective, there are two slight problems:

- The actual thickness of a plotted line depends on its slope. This is illustrated in Figure 3. Although this is a small weakness of the technique most graphics packages do not attempt to address this problem. You can notice this effect in, for example, the Paint accessory in Microsoft Windows.
- The ends of lines are either vertical or horizontal. This is illustrated in Figure 4. Depending on whether we are plotting extra pixels in the horizontal or vertical directions, the line ends will be horizontal or vertical.

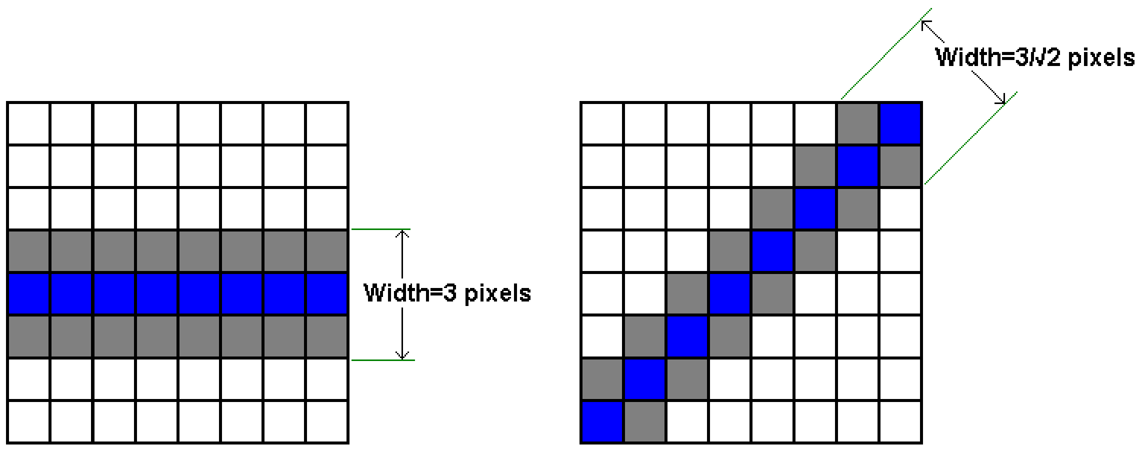


Figure 3 - Line Thickness Depends on Its Gradient

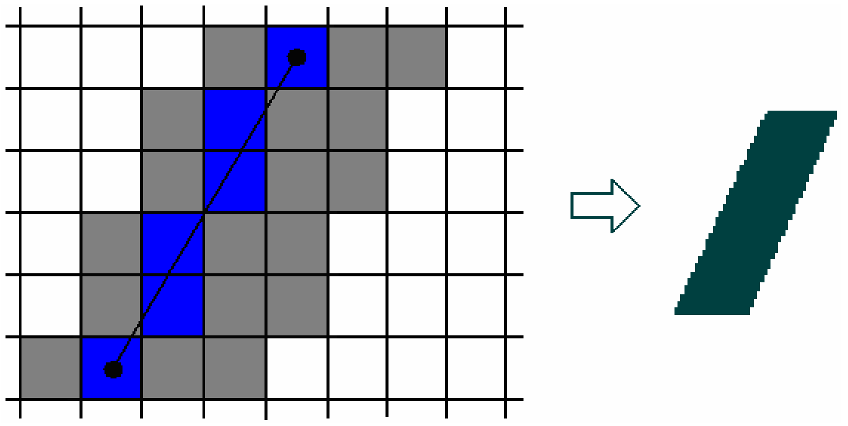


Figure 4 – Horizontal/Vertical Line Ends

The answer to the second problem (horizontal/vertical line ends) is to use *line caps*. A line cap is a shape that is applied to the end of the line only. Three types of line cap are common in computer graphics:

- *Butt cap*
- *Round cap*
- *Projecting square cap*

These are illustrated in Figure 5. The butt cap is formed by drawing a line through each end-point at an orientation of 90^o to the direction of the line. The round cap is formed by drawing a semi-circle at each end-point with radius equal to half the line width. The projecting square cap is similar to the butt cap but the position of the line is extended by a distance of half the line width.

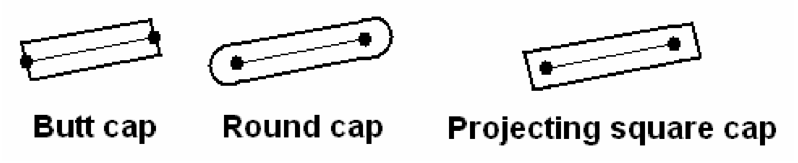


Figure 5 - Types of Line Cap

These line caps effectively solve the problem of vertical/horizontal line endings, but in the process they introduce a new problem. If we are drawing *polylines*, or a connected series of line segments, with these line caps, we can get problems at *line joins*. For example, Figure 6 shows what happens at the line join of two line segments drawn with butt caps. There is a small triangular area at the join that does not get filled.

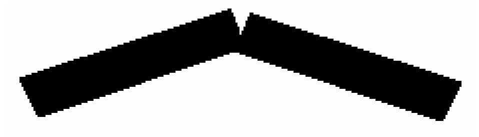


Figure 6 - Problems at Line Joins

This problem can be overcome by using the following *join types*:

- *Miter* join – extend the outer boundaries of the two lines until they meet.
- *Round* join – draw a circular boundary centred on the join point with a diameter equal to the line width.
- *Bevel* join – use butt caps, then fill in the triangle that is left unfilled.

These join types are illustrated in Figure 7. One possible problem occurs when using the miter join when the angle between the two lines is very small. In this case the projecting point of the join can be very long, making the line-join appear unrealistic. Therefore graphics packages will often check the angle between lines before using a miter join.

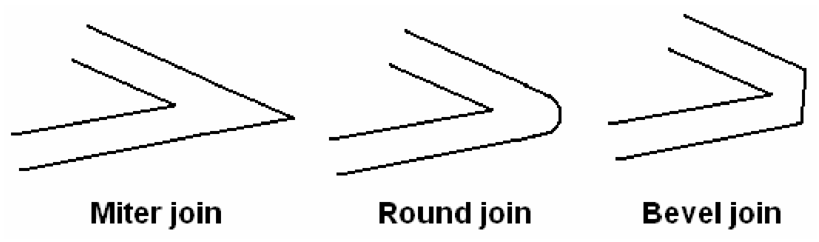


Figure 7 - Line Join Types

4.2. Line Style

The *style* of a line refers to whether it is plotted as a *solid* line, *dotted*, or *dashed*. The normal approach to changing line style is to define a *pixel mask*. A pixel mask specifies a sequence of bit values that determine whether pixels in the plotted line should be *on* or *off*. For example, the pixel mask 11111000 means a dash length of 5 pixels followed by a spacing of 3 pixels. In other words, if the bit in the pixel mask is a 1, we plot a pixel, and if it is a zero, we leave a space.

4.3. OpenGL Line Attribute Functions

OpenGL allows us to change both the line width and the line style. To change the line width we use the *glLineWidth* routine. This takes a single floating point argument (which is rounded to the nearest integer) indicating the thickness of the line in pixels. For example,

glLineWidth(4.3)

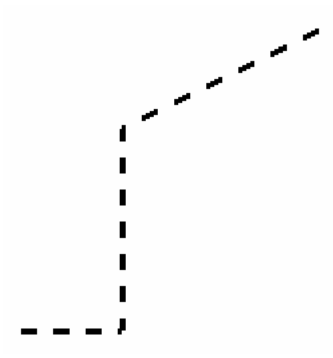
will set the line width state variable to 4 pixels. This value will be used for all subsequent line primitives until the next call to *glLineWidth*. The ‘true’ line thickness will depend on the orientation of the line.

The line style is specified using a pixel mask. First we must enable the *line stipple* feature of OpenGL using the following line:

glEnable(GL_LINE_STIPPLE)

Next, we use the *glLineStipple* function to define the line style. *glLineStipple* takes two arguments: a *repeat factor*, which specifies how many times each bit in the pixel mask should be repeated, and a *pattern*, which is a 16-bit pixel mask. For example, the code shown below draws the dashed line shown to the right. The pixel mask is the hexadecimal number *00FF*, which is 8 zeros followed by 8 ones. The repeat factor is 1 so each one or zero in the pixel mask corresponds to a single pixel in the line.

```
glEnable(GL_LINE_STIPPLE);
glLineWidth(3.0);
glLineStipple(1, 0x00FF);
glBegin(GL_LINE_STRIP);
  glVertex2i(100,100);
  glVertex2i(150,100);
  glVertex2i(150,200);
  glVertex2i(250,250);
glEnd();
glDisable(GL_LINE_STIPPLE);
```



5. Fill-Area Attributes

The most important attribute for fill-area polygons is whether they are filled or not. We can either draw a filled polygon or drawn the outline only. Drawing the outline only is known as *wireframe* rendering. If the polygon is to be filled we can specify a *fill style*.

In this section we first consider some algorithms for filling polygons and other fill-area primitives. Then we describe how to modify fill-area attributes in OpenGL.

5.1. Fill Algorithms

Broadly speaking, we can identify two different approaches to filling enclosed boundaries:

- *Scan-line approach*: We automatically determine which pixels are inside or outside the boundary for each scan line. These approaches are normally used by general-purpose graphics packages.
- *Seed-based approach*: Start from an interactively defined *seed point* and paint outwards until we reach a boundary. These approaches are normally used by special-purpose graphics packages.

5.1.1. Scan-Line Fill Algorithm

leader

Scan-line fill algorithms are automatic (i.e. they require no user intervention) and are commonly used by general-purpose graphics packages such as OpenGL. One scan-line fill algorithm is described below.

The scan-line fill algorithm automatically fills any non-degenerate polygon by considering each scan-line (i.e. row of the frame buffer) in turn. We move across the scan-line, from left-to-right, until we reach a boundary. Then we start plotting pixels in the fill colour and continue to move across the scan-line. When we reach another boundary we stop filling. At the next boundary we start filling again, and so on. We can think of this algorithm as being similar to the *odd-even rule* inside-outside test: the starting point (i.e. the far left of the scan-line) is assumed to be outside the polygon, so points between the first and second boundary crossings are assumed to be inside. This process is illustrated in Figure 8.

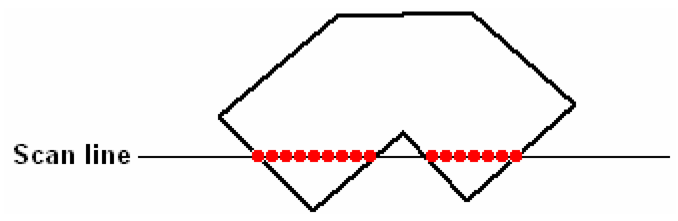


Figure 8 - The Scan-Line Fill Algorithm

The scan-line fill algorithm works well for simple polygons, but there are cases in which it can experience problems. We now consider such a case, illustrated in Figure 9. The scan-line algorithm must always treat vertices as <u>two</u> boundary crossings: if we did not then as we crossed the vertex in scan-line *x* in Figure 9 the algorithm would stop filling when it should continue. However, if we treat vertices as two boundary crossings it can cause problems in other case, as shown by scan-line *y*. Here we treat the vertex as two boundary crossings, but it causes the algorithm to continue filling when it should stop. The difference between scan-lines *x* and *y* is that in scan-line *x* the boundary does not cross the scan-line at the vertex, whereas in scan-line *y* it does.

There are two possible answers to this problem. First, we can perform a preprocessing stage to detect which vertices cross the scan-line and which don't. This would work OK, but it also takes time. Therefore the more common approach is to insist that all polygons are convex. As we can see from Figure 10, in a convex polygon there is always a maximum of two boundary crossings in every scan-line. Therefore if there are two crossing we simply fill between the two crossing points. If there is one we know that it is a vertex crossing so we fill a single pixel. This greatly improves the efficiency of the scan-line fill algorithm, and is the reason why most graphics packages insist that all fill-area primitives are convex polygons.

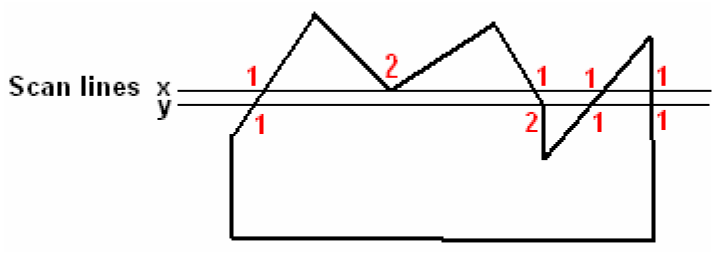


Figure 9 - Problems in the Scan-Line Fill Algorithm: Scan-Line 'x' is Filled Correctly but Scan-Line 'y' is not

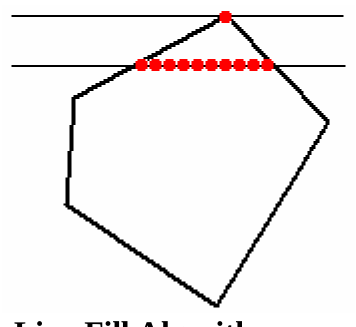


Figure 10 - Scan-Line Fill Algorithm on a Convex Polygon

5.1.2. Seed-Based Fill Algorithms

Seed-based fill algorithms have the advantage that they can fill arbitrarily complex shapes. They are less efficient than scan-line algorithms, and require user interaction in the form of specifying a seed-point (i.e. a starting point for the fill, known to be inside the boundary). Therefore they are typically used by interactive packages such as paint programs.

In this section we will examine two similar techniques: *boundary-fill* and *flood-fill*. Both are recursive in nature and work by starting from the seed point and painting outwards until some boundary condition is met. The difference between the two algorithms is in the nature of the boundary condition.

5.1.2.1. Boundary Fill

To implement the boundary fill algorithm we first define three parameters: the seed point, a fill colour and a boundary colour. The pseudocode for the algorithm is:

- Start with seed point
- If the current pixel is not in the boundary colour and not in the fill colour:
    - Fill current pixel using fill colour
    - Recursively call boundary-fill for all neighbouring pixels

Notice that the termination condition for the algorithm is when we reach pixels in the boundary colour. Therefore the boundary must be in a single (known) colour. We check to see if the current pixel is in the fill colour to ensure we don't fill the same pixel many times.

One question that we must answer before implementing boundary-fill is what exactly we mean by a *neighbouring point*. Figure 11 shows that there are two possible interpretations of a neighbour: 4-connected and 8-connected neighbours. Both interpretations can be useful, although using 8-connected neighbours can lead to the algorithm 'escaping' from thin diagonal boundaries, so 4-connected neighbours are more commonly used.

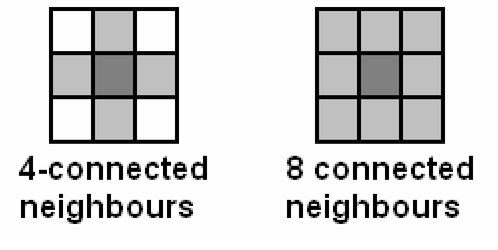


Figure 11 - 4-Connected and 8-Connected Neighbours

Another issue that we must consider is what to do if the region to be filled is already partly filled. For example, in Figure 12 the boundary colour is grey, the fill colour is red and the seed-point is the pixel marked by the letter ‘S’. Since boundary-fill will not process any pixels that are already in the fill colour, in this example only half of the fill-area will be filled: the algorithm will stop when it reaches the already-filled pixels. One solution to this problem is to pre-process the fill-area to remove any partly-filled areas.

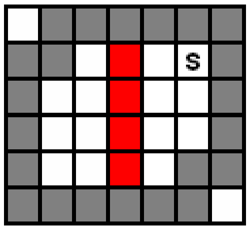


Figure 12 - A Partly Filled Region

A C++ implementation of boundary fill is given below.

```
void boundaryFill (int x, int y,
                   int fillColour,
                   int borderColour) {
 int interiorColour = getpixel (x, y);
 if ((interiorColour != borderColour) &&
     (interiorColour != fillColour) {
  setPixel(x,y,fillColour);
  boundaryFill(x,y-1,fillColour,borderColour);
  boundaryFill(x-1,y,fillColour,borderColour);
  boundaryFill(x+1,y,fillColour,borderColour);
  boundaryFill(x,y+1,fillColour,borderColour);
 }
}
```

5.1.2.2. Flood Fill

Flood-fill is very similar to boundary-fill, but instead of filling until it reaches a particular boundary colour, it continues to fill while the pixels are in specific *interior colour*. Therefore, first we must define a seed point, a fill colour and an interior colour. Pseudocode for flood-fill is given below:

- Start with seed point
- If current pixel in interior colour:
    - Fill pixel using fill colour
    - Recursively call flood-fill for all neighbouring pixels

Again, for flood-fill we have the same issues regarding partly-filled areas and what we mean by a neighbouring pixel. Actually the two algorithms are very similar, and in many cases their operation will be identical. But flood fill can be more useful for cases where the boundary is not in a single colour, such as that shown in Figure 13.

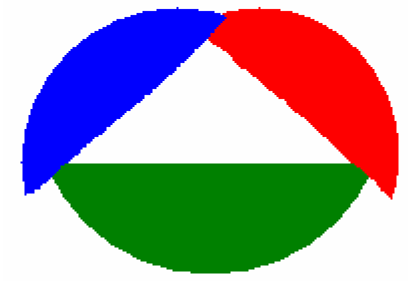


Figure 13 - An Enclosed Region Where the Boundary is not in a Single Colour

C++ code to implement the flood-fill algorithm is shown below.

```
void floodFill (int x, int y,
                int fillColour,
                int interiorColour) {
 int colour = getpixel (x, y);
 if (colour == interiorColour) {
  setPixel(x,y,fillColour);
  floodFill(x+1,y,fillColour,interiorColour);
  floodFill(x-1,y,fillColour,interiorColour);
  floodFill(x,y+1,fillColour,interiorColour);
  floodFill(x,y-1,fillColour,interiorColour);
 }
}
```

5.2. OpenGL Fill-Area Attribute Functions

OpenGL provides a number of features to modify the appearance of fill-area polygons. First, we can choose to fill the polygon or just display an outline (i.e. wireframe rendering). We do this by changing the display mode using the *glPolygonMode* routine. The basic form of this function is:

glPolygonMode(face, displayMode)

Here, the *face* argument can take any of the following values:

- *GL_FRONT*: apply changes to front-faces of polygons only.
- *GL_BACK*: apply changes to back-faces of polygons only.
- *GL_FRONT_AND_BACK*: apply changes to front and back faces of polygons.

The *displayMode* argument can take any of these values:

- *GL_FILL*: fill the polygon faces.
- *GL_LINE*: draw only the edges of the polygons (*wireframe* rendering)
- *GL_POINT*: draw only the vertices of the polygons.

For example, the following code draws a polygon with four vertices using wireframe rendering for the front face.

```
glPolygonMode (GL_FRONT, GL_LINE);
glBegin(GL_POLYGON);
  glVertex2i(350,250);
  glVertex2i(400,300);
  glVertex2i(400,350);
  glVertex2i(300,250);
glEnd();
```

In addition to changing the polygon drawing mode, we can also specify a fill pattern for polygons. We do this using the *polygon stipple* feature of OpenGL. The steps can be summarised as:

- Define a fill pattern
- Enable the polygon stipple feature of OpenGL
- Draw polygons

For example, the following code draws an eight-sided polygon with a fill pattern. The fill pattern accepted by the *glPolygonStipple* function must be an array of 128 8-bit bitmaps. These bitmaps represent a 32x32 pixel mask to be used for filling. The image produced by the code below is shown in Figure 14.

```
GLubyte fillPattern[] = {0x00, 0xFF, 0x00, ... };
glPolygonStipple(fillPattern);
glEnable(GL_POLYGON_STIPPLE);
glBegin(GL_POLYGON);
  glVertex2i(350,250);
  glVertex2i(400,300);
  glVertex2i(400,350);
  glVertex2i(350,400);
  glVertex2i(300,400);
  glVertex2i(250,350);
  glVertex2i(250,300);
  glVertex2i(300,250);
glEnd();
```

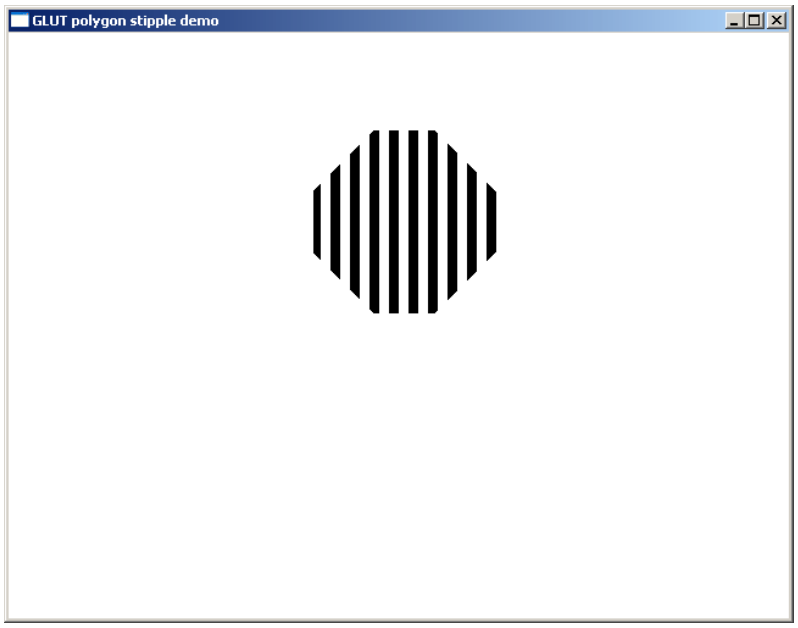


Figure 14 - Example of Using the OpenGL Polygon Stipple Feature

6. Character Attributes

The attributes that change the appearance of character primitives are:

- Size
- Style/font
- Colour

Whether or not it is possible to change these attributes depends upon what type of character primitive is being drawn. Recall from Chapter 2 that character primitives can be either *bitmap* characters or *stroke* characters. The next section describes what character attribute changes can be made for each of these types of primitive.

6.1. OpenGL Character Attribute Functions

Recall that 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.

7. Aliasing

The term *aliasing* refers to the introduction of errors into an image as a consequence of the *discretisation* (quality of being discrete) of the primitives when plotting on a raster graphics system. Whenever a primitive is drawn on a raster display it has to be *scan-converted*. This involves determining which of the discrete locations of the pixels the primitives should be plotted at. By forcing the primitives to be plotted only at these discrete locations we inevitably introduce errors. Another name for the effect of aliasing is the "staircase effect", or the "jaggies". For example, in Figure 15 a curved line is plotted on a limited resolution display. The staircase effect in the plotted line is obvious. The lower the resolution of the display, the more noticeable will be the effects of aliasing.

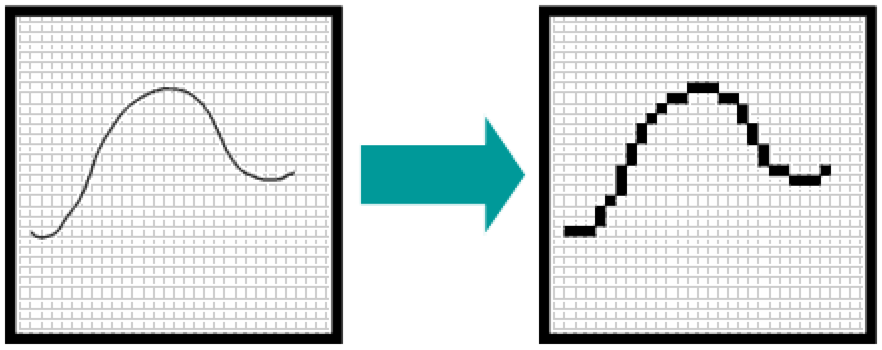


Figure 15 - The Effects of Aliasing in Plotting a Curved Line

In mathematical terms, the actual cause of aliasing is *undersampling*. This is illustrated in Figure 16. Here we have a 1-D signal which is sampled at regular intervals. The resulting undersampling signal contains obvious errors when compared to the original signal. When primitives are scan-converted this same undersampling occurs in 2-D (i.e. the *x* and *y* dimensions of the image) causing errors in the image.

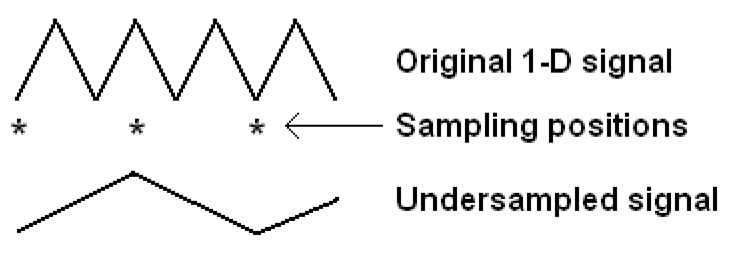


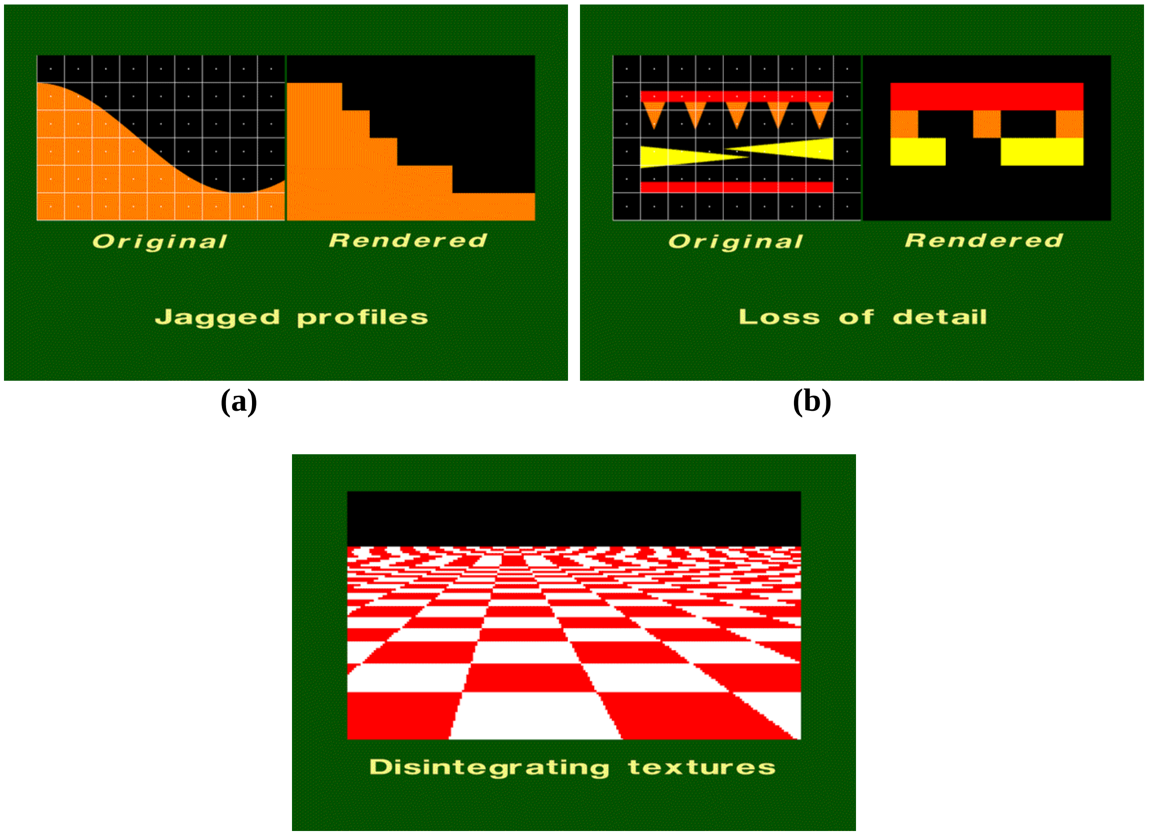
Figure 16 - Undersampling

The errors caused by aliasing are known as *aliasing artefacts*, and they can be broadly classified into four different types (see Figure 17):

- *Jagged profiles*: This is the most common form of artefact, and occurs when a smooth boundary is discretised into a 'jaggy' one. It is especially noticeable when there is a high contrast between the two colours.
- *Loss-of-detail*: When there is fine detail whose size is less than the size of a pixel, the detail can be rendered incorrectly, or even disappear completely. In Figure 17(b) we can see that one red rectangle disappears, one changes size, two of the orange triangles disappear, and the two yellow triangles are different in size after scan-conversion.
- *Disintegrating textures*: This is a related phenomenon to the loss-of-detail. One important type of detail is texture (consistency). In Figure 17(c) we can see that the square texture pattern becomes distorted close to the horizon, where the size of the squares approaches the size of a pixel. Some rectangles appear to get bigger instead of smaller, and others become distorted.
- *Creeping*: This is an effect of aliasing that is noticeable in animated scenes. We saw in the loss-of-detail example that the position of aliased features can change as well as be distorted. For example, in Figure 17(b) the ends of the two yellow triangles are shifted from their 'actual' locations. If this effect occurs in animated sequences, the position of features may change <u>between</u> frames, causing objects to 'creep' even though they shouldn't be noticeably moving.

The term *antialiasing* refers to any technique (hardware or software) that compensates for the effects of aliasing. For example, Figure 18 shows that the jagged profile aliasing artefact can be made less noticeable by processing the aliased image in some way. In the following sections we will look at three possible antialiasing algorithms:
We will look at 3 techniques:

- *Supersampling*
- *Area sampling*
- *Pixel phasing*



(a) (b)

(c)

Figure 17 - Types of Aliasing Artefact

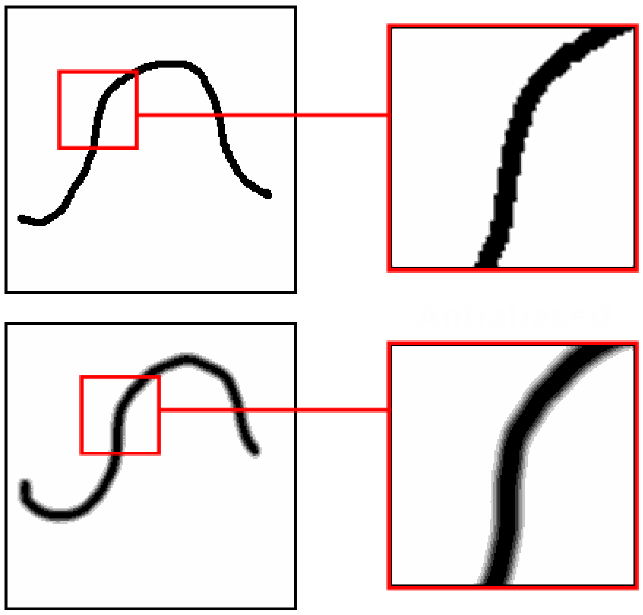


Figure 18 - Antialiasing Techniques Can Reduce Aliasing Artefacts

7.1. Supersampling

The *supersampling* technique is also known as *postfiltering*. It attempts to compensate for the effects of undersampling (i.e. reduced resolution) by supersampling (increasing the resolution) before plotting. To plot an antialiased line, the basic idea of supersampling is as follows:

- 'Supersample' the image (i.e. increase its resolution).
- Plot the line in the supersampled image.
- Count the number of plotted points within each corresponding 'real' pixel.
- Plot the 'real' pixel with an intensity that is proportional to the count of supersampled pixels. Incredible

For example, Figure 19 shows a supersampled image (to the left) with a straight-line plotted in it. 'Real' pixels in the original image correspond to a 3x3 block of pixels in the supersampled image. To find the intensity for each original pixel, we count the number of supersampled pixels that are 'on'. For the bottom left 'real' pixel there are three plotted supersampled pixels – this is the maximum number possible so we plot this pixels with the maximum intensity (in this case, black). For the bottom-centre pixel, there is only one supersampled pixel that is 'on', so we use an intensity that is 1/3 of the maximum. For the centre pixel there are two supersampled pixels that are 'on', so we plot this pixel with 2/3 the maximum intensity, and so on. The overall effect of this approach is to 'blur' the edges of the line slightly, reducing the staircase effect.

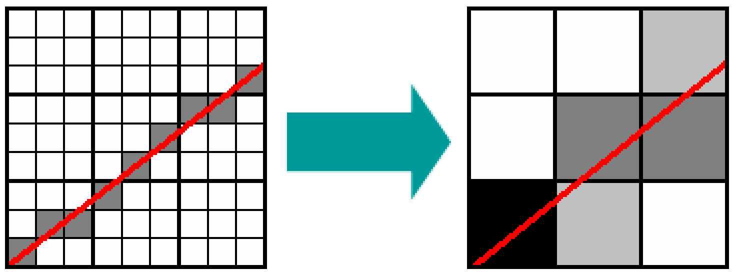


Figure 19 - The Supersampling Antialiasing Technique

Supesampling is often implemented using a *pixel mask*. When counting plotted supersampled pixels, pixel masks allow us to give a higher weighting to pixels closer to the centre of the 'real' pixel. Figure 20 shows the use of one particular pixel mask. Using this mask, if the centre pixel of the 3x3 supersampled block is on, we add 4 to the count. If a corner pixel is on, we only add 1. In this case, the bottom-left 'real' pixel has a supersampled pixel count of 7 (=4+2+1), the centre-bottom pixel has a count of 1 and the centre pixel has a count of 4 (=2+2). As before, these counts are used to determine the intensity of the pixel. Supersampling using a pixel mask is slightly more time-consuming, but can give better results.

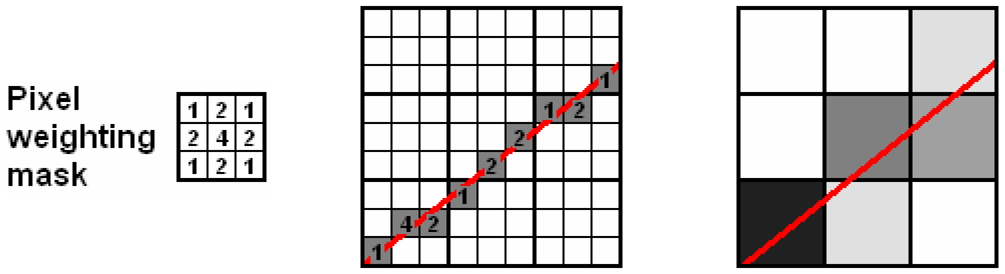


Figure 20 - Supersampling Using a Pixel Mask

7.2. Area Sampling

An alternative antialiasing technique is known as *area sampling*, or *prefiltering*. The basic idea of area sampling is illustrated in Figure 21(a). We determine the location of the primitive (e.g. a straight line) and then compute the *area of overlap* of each pixel with the primitive. The plotted intensity of each pixel is proportional to this area of overlap. So if the primitive completely covers the pixel, it will be plotted with maximum intensity. If it covers 50% of the pixel area it will be plotted with 50% of the maximum intensity, and so on.

In practise, computing the exact area of overlap can be time-consuming. Therefore a simplified implementation of area sampling approximates the area by counting supersampled pixels that are inside the primitive. This is illustrated in Figure 21(b). So if there are 6 overlapping supersampled pixels (as in the bottom-left 'real' pixel), we say that the area of overlap is approximately 6/9 = 67%. In fact, this approximation of area sampling is very similar to the supersampling technique described above.

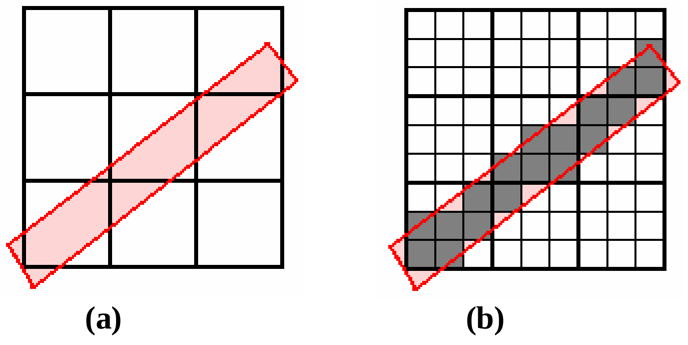
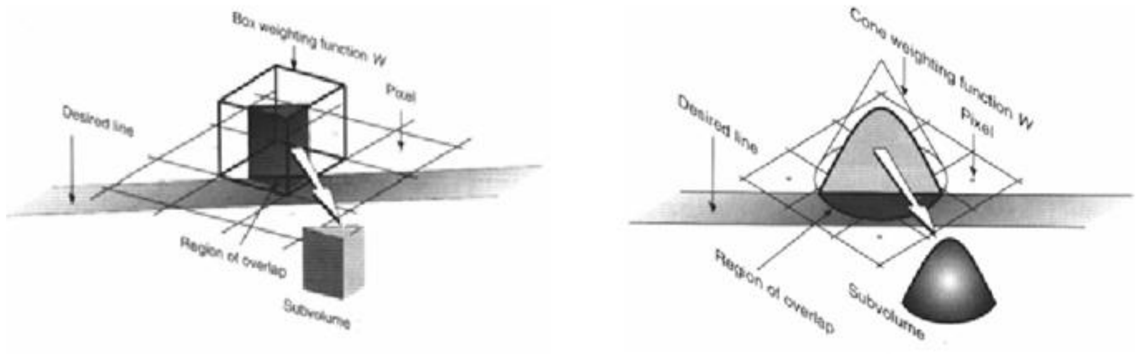


Figure 21 - The Area Sampling Antialiasing Technique

A better (but more computationally expensive) variation of area sampling is known as *weighted area sampling*. This technique is similar to the use of a pixel mask in supersampling, in that it tries to give more weight to the centre of the 'real' pixel. But instead of a discrete pixel mask we use a continuous function. Figure 22 illustrates this idea. We consider the pixel as having a continuous surface superimposed on it. Then when we find the portion of the pixel that overlaps with the primitive we ‘slice’ off a part of this surface. The volume contained under this slice is used as the weight area of overlap. In Figure 22(a) the surface used is flat (a box function), in which case no weighting is applied and the technique is the same as unweighted area sampling. In Figure 22(b) we use a weighting function that is higher at the centre of the pixel, so if the primitive passes through the centre of the pixel it will have a higher weighted area and therefore a higher intensity.



(a) **(b)**

Figure 22 - Weighted Area Sampling

Which weighting function should we use for weighted area sampling? Figure 23 shows three alternatives. As was stated above, the box function does not apply any weighting and so is equivalent to unweighted area sampling. However, it is very fast and efficient. The ideal function to use is the Gaussian function, but this is computationally quite expensive to apply. Therefore a commonly used compromise between the box and Gaussian functions is the cone function. Its performance is good but it is not too expensive computationally.

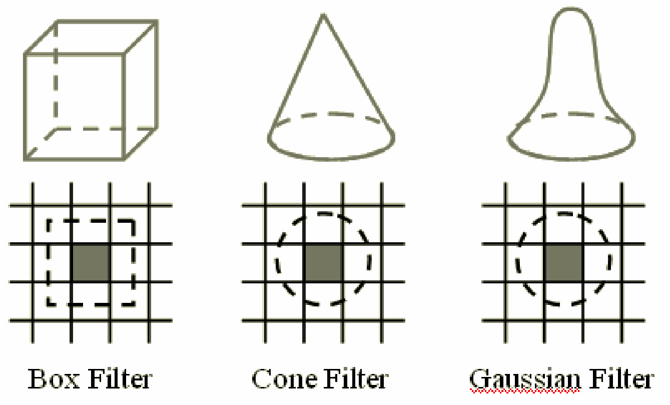


Figure 23 - Weighting Functions for Weighted Area Sampling

7.3. Pixel Phasing

Supersampling and area sampling were both software techniques, i.e. they were algorithms that processed image data to produce antialiased image data. The final technique, *pixel phasing*, is a hardware technique. Therefore, it is not possible to use pixel phasing to perform antialiasing unless you are using a display monitor that uses it.

The basic idea is that the CRT beam is shifted by a fraction of a pixel to bring the plotted pixels closer to the true mathematical line. The hardware normally enables the pixel position to be shifted by 1/2, 1/3, or 1/4 of a pixel. Some systems allow the pixel size to be adjusted too.

Although this technique can produce impressive results, most monitors do not support it so specialised hardware is necessary.

7.4. OpenGL Anti-Aliasing Functions

The OpenGL antialiasing feature can be applied to points, lines or polygons. We enable the appropriate feature using the *glEnable* routine. For example:

glEnable(GL_POINT_SMOOTH)
glEnable(GL_LINE_SMOOTH)
glEnable(GL_POLYGON_SMOOTH)

OpenGL performs antialiasing by colour blending at the edges of primitives. Therefore we also need to enable the colour blending feature, as we have already seen in Section 2.2:

glEnable(GL_BLEND)

Finally we specify the colour blending functions as follows:

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)

Summary

The following points summarise the key concepts of this chapter:

- *Attributes* of graphics primitives are parameters that affect the way the primitive is displayed.
- Most colours are stored as RGB (red, green blue) or RGBA (red, green, blue, alpha) values.
- The *alpha* value of a colour indicates the degree of transparency of the colour.
- Colours can be stored by raster graphics systems using either *direct storage* or a *colour look-up table*.
- Using direct storage the colour is stored directly in the frame buffer.
- Using a colour look-up table the colour is stored in a separate table and the frame buffer stores an index into this table.
- The *glColor** and *glClearColor* functions are used in OpenGL to set the drawing and background colours respectively.
- *Colour blending* allows us to mix one colour with another, often to achieve transparency effects.
- The *glBlendFunc* OpenGL routine allows us to define how colour blending occurs.
- The *glPointSize* function allows us to change the size of point primitives in OpenGL.
- Attributes of line primitives include the line width, line colour and line style.
- *Line caps* are commonly used to improve the appearance of the ends of plotted lines.
- Three common types of line cap are the *butt cap*, the *round cap* and the *projecting square cap*.
- *Line joins* are commonly used to improve the appearance of polylines.
- Three common line joins are the *miter join*, the *bevel join* and the *round join*.
- The *glLineWidth* OpenGL function allows us to change the width of subsequently drawn line primitives.
- The *glLineStipple* function allows us to change the style of lines by specifying a *pixel mask*.
- *Scan-line* fill algorithms fill enclosed areas automatically without user intervention. They are commonly used by general purpose graphics packages such as OpenGL.
- *Seed-based* fill algorithms fill enclosed areas by starting from a user-specified *seed-point*, and painting outwards until they reach a boundary.
- *Boundary-fill* and *flood-fill* are two common seed-based fill algorithms.
- The *glPolygonMode* OpenGL routine defines whether polygons should be filled, displayed in wireframe, or displayed as just vertices.
- The *glPolygonStipple* function allows us to specify fill patterns for fill-area polygons in OpenGL.
- The attributes of character primitives include the size, font/style and colour of the character.
- For *stroke* characters, we can change the thickness of characters using the *glLineWidth* function in OpenGL. We can change the style of the lines in the characters using *glLineStipple*.
- Aliasing is a result of undersampling and causes errors in images.

- Four common types of aliasing artefact are *jagged profiles*, *loss-of-detail*, *disintegrating textures* and *creeping*.
- *Antialiasing* techniques try to compensate for the effects of aliasing.
- Three common antialiasing techniques are *supersampling* (or *postfiltering*), *area sampling* (or *prefiltering*) and *pixel phasing*.
- Supersampling works by increasing the resolution of the image, plotting the primitive and then counting plotted supersampled pixels for each corresponding 'real' pixel.
- Supersampling can make use of a *pixel mask*, which gives more weight to supersampled pixels near the centre of the 'real' pixel.
- *Area sampling* works by computing the area of overlap of the primitive with each pixel.
- A variation of area sampling is known as *weighted area sampling*, in which a weighting function is used to give more weight to the centre of the pixel.
- Pixel phasing is a hardware technique that can shift the location of a pixel by 1/2, 1/3 or 1/4 of a pixel width to bring it closer to the true mathematical position.
- The *GL_POINT_SMOOTH*, *GL_LINE_SMOOTH* and *GL_POLYGON_SMOOTH* features in OpenGL can be used to perform antialiasing.

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