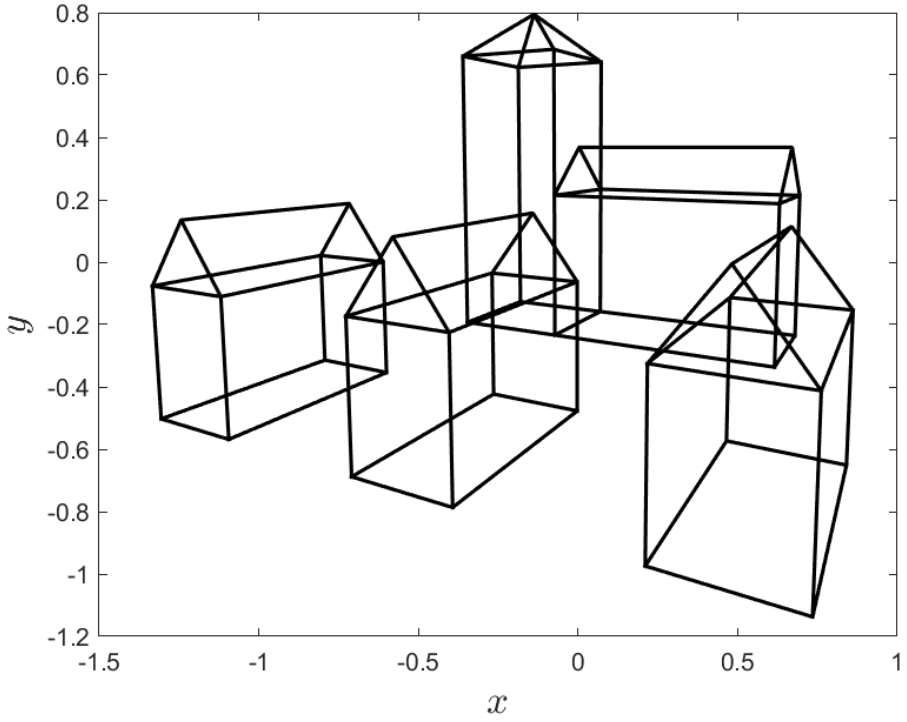
Manchester Metropolitan University

6G5Z3003 Mathematics of Computer Graphics & Virtual Environments

Display Techniques

Dr Jon Shiach

2021 – 2021



Contents

1 Preliminaries **1**
- 1.1 Introduction to the unit . . . 1
- 1.2 Review of linear algebra . . . 2
- 1.3 The graphics pipeline . . . 5
- 1.4 Polygons . . . 7

2 Rasterisation **9**
- 2.1 Rasters . . . 9
- 2.2 Bresenham's algorithm . . . 11
- 2.3 Drawing circles . . . 20
- 2.4 Drawing polygons . . . 25
- 2.5 Texture Mapping . . . 32
- 2.6 Perspective corrected texture mapping . . . 39
- 2.7 Normal mapping . . . 42
- 2.8 Exercises . . . 44

3 Image Processing **47**
- 3.1 Antialiasing . . . 47
- 3.2 Convolution . . . 48

4 Bézier Curves **57**
- 4.1 Bézier curves . . . 58
- 4.2 Bézier surfaces . . . 68
- 4.3 Exercises . . . 70

5 Hidden Surface Removal **73**
- 5.1 Defining objects . . . 73
- 5.2 Back-face culling . . . 75
- 5.3 Painter's algorithm . . . 78
- 5.4 Binary Space Partitioning . . . 81
- 5.5 Exercises . . . 89

6 Clipping **91**
- 6.1 The viewing frustum . . . 91
- 6.2 Line clipping . . . 100
- 6.3 The Cyrus-Beck algorithm . . . 100
- 6.4 The Sutherland-Hodgman algorithm . . . 104
- 6.5 The Cohen-Sutherland algorithm . . . 108
- 6.6 Exercises . . . 112

7 Lighting **115**
- 7.1 The Phong reflection model . . . 116
- 7.2 Shading methods . . . 127

A Exercise solutions **135**
A.1 Rasterisation 135
A.2 Bézier curves 138
A.3 Hidden surface removal 139
A.4 Clipping 140

Chapter 1

Preliminaries

1.1 Introduction to the unit

Welcome to the Mathematics of Computer Graphics and Virtual Environments unit. In this unit you will learn the mathematics and techniques that make the images and animations we see in modern computer games and movies possible. The unit is split into two halves, Dr Killian O'Brien will be covering vector geometry, linear transformations and projections that are used to define a virtual environments and enable us to navigate around a virtual world and view it from different positions and directions. In this half of the unit I will be covering the techniques required to display a virtual world on-screen in addition to image processing techniques and drawing of smooth curves.

1.1.1 Assessment

This unit is assessed through a coursework assignment and an examination. The coursework assignment is worth 30% of the unit and you be given a number of tasks to complete whilst studying the unit. These tasks will include pen and paper calculations as well as writing programs in MATLAB. The examination is worth 70% of the unit and will take place after this teaching block. You will be given a set of questions to complete over a 27 hour period.

1.1.2 Teaching schedule

The teaching schedule for this half of the unit is shown in table 1.1. I will try to stick to this schedule as closely as possible but students should be aware that there may be slight changes to this.

Table 1.1: Teaching schedule

Week	Data (w/c)	Content
1	01/03/2021	Preliminaries (chapter 1) and the rasterisation of lines and circles (chapter 2)
2	08/03/2021	Drawing polygons and texture mapping (chapter 2)
3	15/03/2021	Bézier curves (chapter 4) and image processing (chapter 3)
4	22/03/2021	Hidden surface removal (chapter 5)
5	29/03/2021	Clipping (chapter 6) and lighting (chapter 7)
6	26/04/2021	Consolidation and exam preparation

1.2 Review of linear algebra

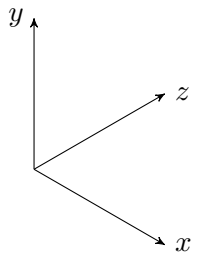
Computer graphics uses concepts and techniques from linear algebra. This section serves as a review of the fundamental concepts you would have studied in the level 4 unit Linear Algebra and Programming Skills.

1.2.1 Co-ordinate systems

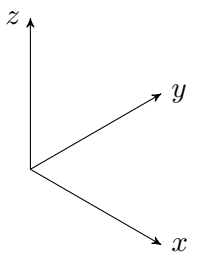
The **Cartesian** co-ordinate system uses a set of orthogonal (perpendicular) number lines known as **axes** which, for three-dimensional space, are typically labelled x, y and z. The position of a point in a space is given by a set of **co-ordinates** which are the signed distances along each axis expressed as an ordered set in the form of a 3-tuple (x, y, z). Since $x, y, z \in \mathbb{R}$ then a Cartesian space is represented by $\mathbb{R}^n$ where n is the number of spatial dimensions.

The order of the axes in $\mathbb{R}^3$ can follow one of two configurations commonly known as the **left-handed** and **right-handed** configurations. If the thumb on our right hand represents the x-axis, the index finger represents the y axis and the middle finger represents the z axis, then with your palm face up move your thumb so that it is at a right-angle to your index finger which is pointing forwards and extend your middle finger upwards. This is the right-handed co-ordinate system and is the one most commonly used in mathematics (figure 1.1b).

Using the same fingers to represent the axes as before but this time on the left hand, if have your left palm facing away from you with the fingers pointing upwards, extend the thumb so that it is at right-angle to your index finger and point your middle finger forwards away from you. This is the left-handed co-ordinate system (figure 1.1a) which is commonly used in computer graphics, therefore these notes will assume a left-hand co-ordinate system.



(a) Left-handed co-ordinate system



(b) Right-handed co-ordinate system

Figure 1.1: The left and right-handed co-ordinate systems.

1.2.2 Homogeneous co-ordinates

In computer graphics applications, co-ordinates that define the position of a point are expressed using **homogeneous co-ordinates**. Let (x, y, z, w) be the homogeneous co-ordinates corresponding to the Cartesian co-ordinates in $\mathbb{R}^3$ (x', y', z') where the following relationships apply

$$x' = \frac{x}{w}, \qquad y' = \frac{y}{w}, \qquad z' = \frac{z}{w},$$

i.e., the Cartesian co-ordinates are calculated by dividing the x, y and z homogeneous co-ordinates by the fourth element w. Note that when $w = 1$ the homogeneous co-ordinates $(x, y, z, 1)$ correspond to the Cartesian co-ordinates (x, y, z), so for many graphics applications we use $w = 1$ for simplicity. The reason why homogeneous co-ordinates are used in computer graphics is because it allows us to apply translation and projection operations using matrix multiplication.

1.2.3 Vectors

A **vector** is an object that has **magnitude** (length) and direction. In mathematical notation vectors are denoted in one of three ways: by a boldface character, e.g., $\vec{a}$ (usually in print) or as an underlined character when writing by hand, e.g., $\underline{a}$. These notes will use will use boldface characters to represent vectors.

An individual vector is defined by the signed distance along each of the axes by a tuple. For example, let $\mathbf{a}$ be a vector in three-dimensions defined by the 3-tuple $\mathbf{a} = (a_x, a_y, a_z)$ then $\mathbf{a}$ can be represented geometrically as the arrow shown in figure 1.2.

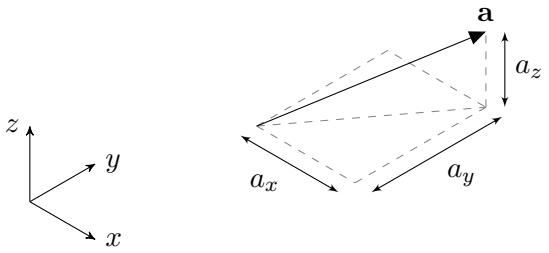


Figure 1.2: The vector $\mathbf{a} = (a_x, a_y, a_z)$.

Here follows some key definitions for vectors.

Definition 1. The **magnitude** of a vector $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ is denoted by $\|\mathbf{a}\|$ and is the length of the vector and is calculated using

$$\|\mathbf{a}\| = \sqrt{\sum_{i=1}^{n} a_i^2}. \tag{1}$$

For example, given the vector $\mathbf{a} = (3, 4, 0)$ then the magnitude is

$$\|\mathbf{a}\| = \sqrt{3^2 + 4^2 + 0^2} = \sqrt{25} = 5.$$

Definition 2. A **unit vector** is a vector with a magnitude of 1. A unit vector can be found for any vector by dividing the vector by its magnitude. The unit vector that points in the same direction as the vector $\mathbf{a}$ is denoted by $\hat{\mathbf{a}}$ (referred to as "$\mathbf{a}$ hat") and can be calculated using

$$\hat{\mathbf{a}} = \frac{\mathbf{a}}{\|\mathbf{a}\|}. \tag{2}$$

For example, given the vector $\mathbf{a} = (3, 4, 0)$ the unit vector pointing in the same direction as $\mathbf{a}$ is

$$\hat{\mathbf{a}} = \frac{(3, 4, 0)}{5} = \left(\frac{3}{5}, \frac{4}{5}, 0\right).$$

Definition 3. The **dot product** of two vectors $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ and $\mathbf{b} = (b_1, b_2, \ldots, b_n)$ is defined by

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n. \tag{3}$$

For example, the dot product of the two vectors $\mathbf{a} = (3, 4, 0)$ and $\mathbf{b} = (5, 12, 0)$ is

$$\mathbf{a} \cdot \mathbf{b} = 3 \times 5 + 4 \times 12 + 0 \times 0 = 15 + 48 = 63.$$

Definition 4. The **geometric definition of the dot product** is

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos(\theta), \tag{4}$$

where θ is the angle between the two vectors (fig 1.3).

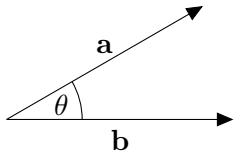


Figure 1.3: The two vectors $\mathbf{a}$, $\mathbf{b}$ and the angle between them θ is related by the dot product.

Definition 5. The **cross product** of two vectors $\mathbf{a} = (a_1, a_2, a_3)$ and $\mathbf{b} = (b_1, b_2, b_3)$ in $\mathbb{R}^3$ is computed using

$$\mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ a_1 & a_2 & a_3 \\ b_1 & b_2 & b_3 \end{vmatrix} = (a_2 b_3 - b_2 a_3, a_3 b_1 - a_1 b_3, a_1 b_2 - a_2 b_1), \tag{5}$$

and returns a vector that is perpendicular to both $\mathbf{a}$ and $\mathbf{b}$ (figure 1.4).

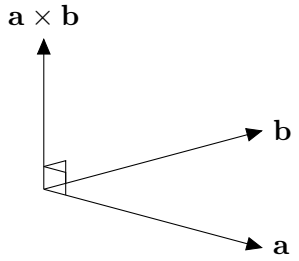


Figure 1.4: The cross product of the two vectors $\mathbf{a}$ and $\mathbf{b}$ produces a vector that is perpendicular to the plan that $\mathbf{a}$ and $\mathbf{b}$ lie on.

For example, the cross product of the two vectors $\mathbf{a} = (3, 4, 0)$ and $\mathbf{b} = (1, 2, 3)$ is

$$\begin{aligned} \mathbf{a} \times \mathbf{b} &= \begin{vmatrix} \mathbf{i} & \mathbf{j} & \mathbf{k} \\ 3 & 4 & 0 \\ 1 & 2 & 3 \end{vmatrix} \\ &= (4 \times 3 - 0 \times 2)\mathbf{i} - (3 \times 3 - 0 \times 1)\mathbf{j} + (3 \times 2 - 4 \times 1)\mathbf{k} \\ &= (12, -9, 2). \end{aligned}$$

We can check that this vector is perpendicular to $\mathbf{a}$ and $\mathbf{b}$ by calculating the dot product, e.g.,

$$\begin{aligned} (12, -9, 2) \cdot (3, 4, 0) &= 36 - 36 + 0 = 0, \\ (12, -9, 2) \cdot (1, 2, 3) &= 12 - 18 + 6 = 0. \end{aligned}$$

1.3 The graphics pipeline

The process of constructing a virtual world and rendering it on a computer display can be summarised by the flow diagram called the **graphics pipeline** shown in figure 1.5.

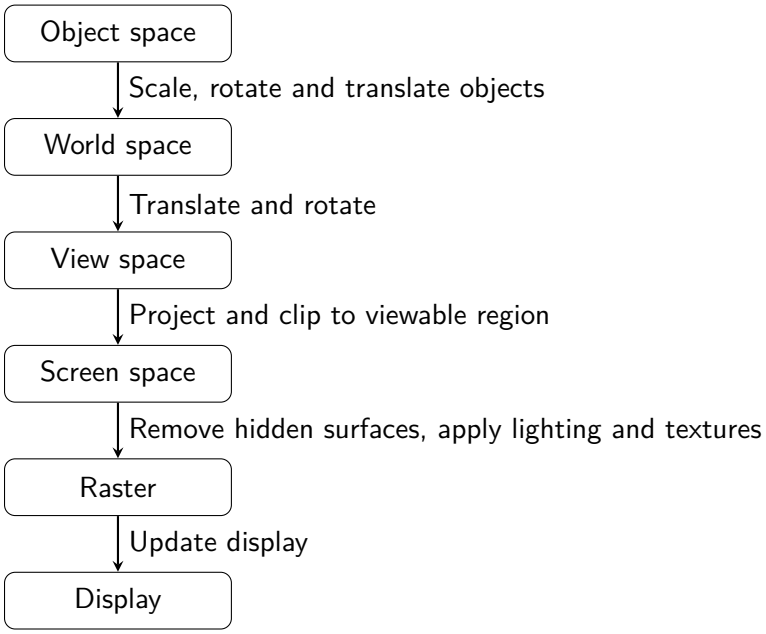
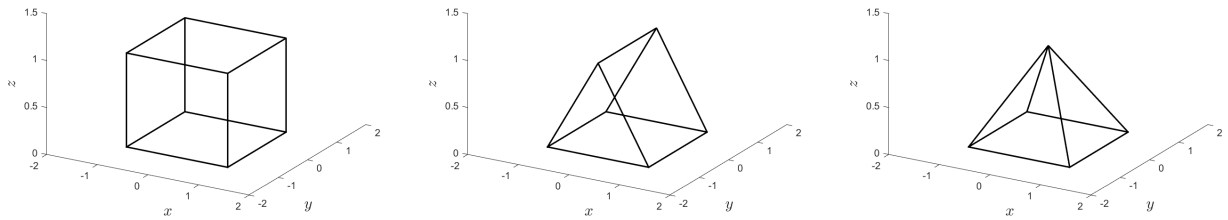


Figure 1.5: The graphics pipeline.

Object space The three dimensional objects that are used to build the virtual world are each defined in their own space (figure 1.6).



(a) Cube (b) Prism (c) Pyramid

Figure 1.6: Primitive objects are defined in their own object space.

World space Objects are scaled, rotated and translated into the world space to construct the virtual world (figure 1.7).

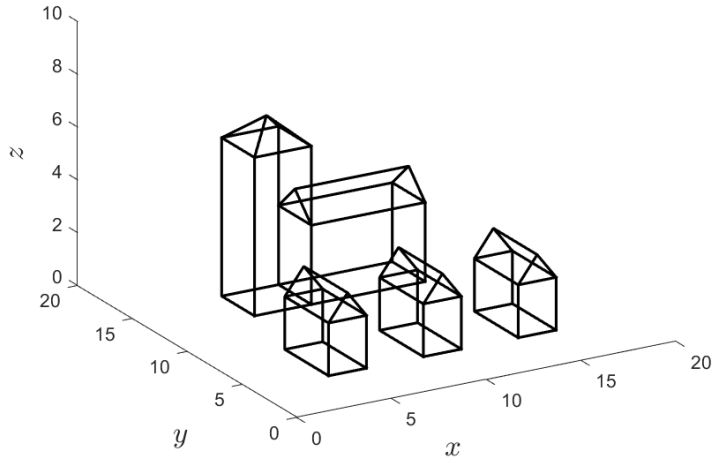
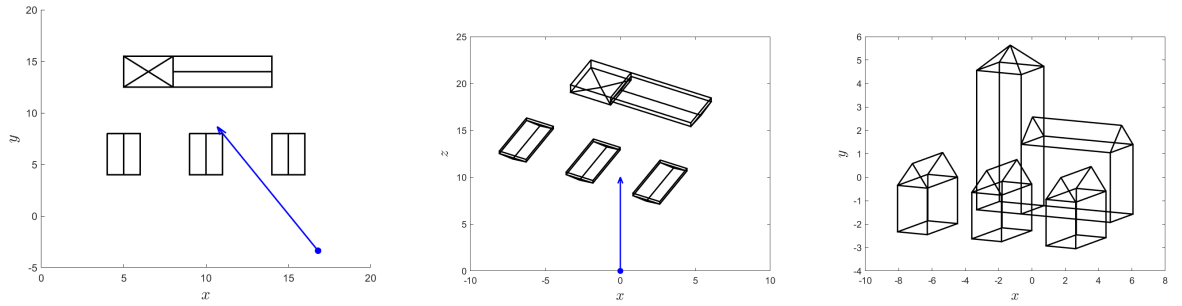


Figure 1.7: Objects are scaled, rotated and translated to build the virtual world in the World space

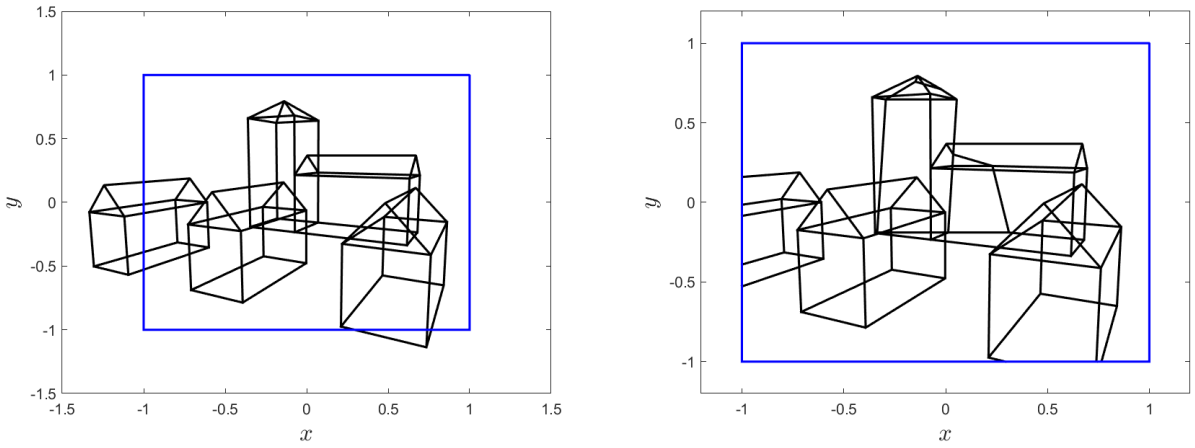
View space The world space is viewed from a given position and direction. The world space is translated so that the viewing position is at the origin and the direction of view is along the z-axis (figure 1.8). A left-handed co-ordinate system is used from this step onwards such that the x and z-axes point along the horizontal and the y-axis points vertically upwards.



(a) World space (b) View space (c) View space viewed along z axis

Figure 1.8: The world space is translated and rotated so that the viewing position is at the origin and the direction of view is along the z-axis.

Screen space The view space is projected onto the screen space using **perspective projection** which provides some indication of the distance of objects from the viewer (figure 1.9a). The screen space objects are clipped to the viewable region so that any objects that the view should not be able to see are removed (figure 1.9b). Clipping is covered in chapter 6.



(a) Screen space

(b) Clipped screen space

Figure 1.9: The view space is projected onto the screen space using perspective projection and clipped to the viewable region.

Raster The screen space is converted to a raster array which is a pixelised representation of the polygons in the screen space (see chapter 2). This step can involve applying texture maps and lighting models. The completed raster is then sent to the display.

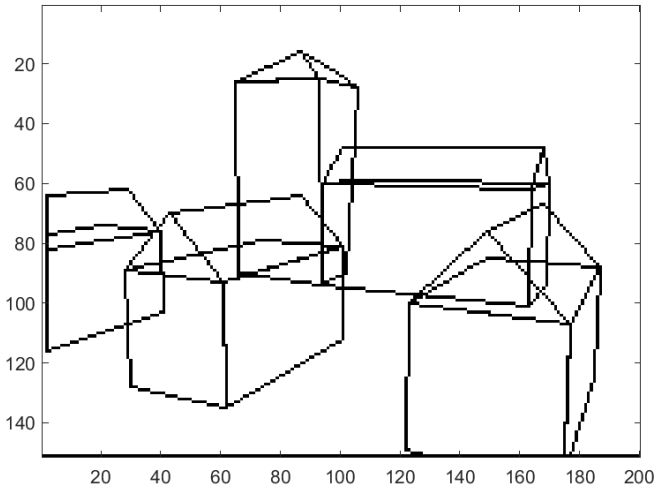


Figure 1.10: The raster representation of the screen space.

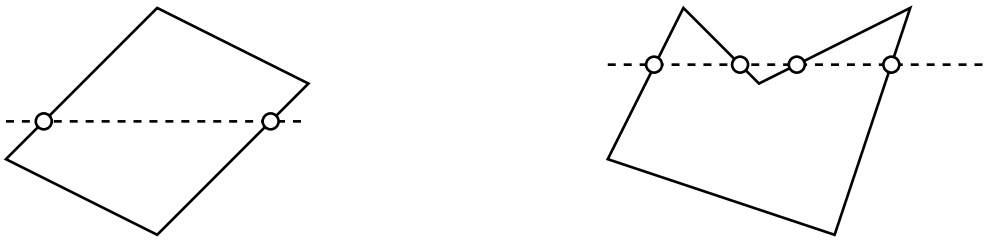
1.4 Polygons

Polygons are the fundamental building blocks of a three-dimensional virtual environment so it is important that they are well understood. To simplify the various techniques used in computer graphics and to help keep computational costs to a minimum in practice only convex polygons are considered. If a concave polygon is encountered we split it up into convex polygons. Usually we only deal with triangular polygons since these are the simplest polygons but many of the techniques and examples used in these notes are applicable to polygons with more than three edges.

Definition 6. A **polygon** is a plane figure that is defined by a closed loop of straight line segments. The straight line segments are called **edges** and the position of the endpoints of an edge are called **vertices** (singular: vertex).

Definition 7. A **convex polygon** is a polygon where any straight line drawn through the polygon will intersect the polygon edges at most twice.

Definition 8. A **concave polygon** is a polygon where it is possible to draw a straight line through the polygon intersecting the polygon edges more than twice. A polygon is concave if it is not convex. Any concave polygon can be split up into multiple convex polygons.



(a) convex polygon (b) concave polygon

Figure 1.11: Convex and concave polygons.

It is convenient to define polygons in a standard way so that techniques used to render polygons and to determine their normal vectors for use in lighting and hidden surface removal (see chapter 7 and chapter 5) are consistent. It is customary to list the vertices that define a polygon in an anti-clockwise direction traversing around the circumference of a polygon. For example, the polygon shown in figure 1.12 is defined by the vertices (x_i, y_i) where $i = 0, 1, 2, 3$ (it is customary to label the first vertex using an index of 0 although this is not strictly necessary).

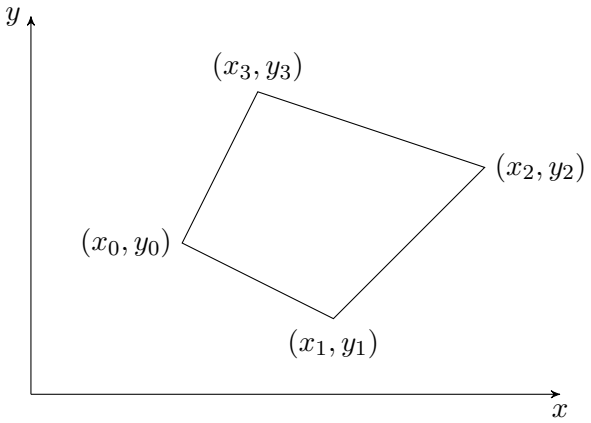


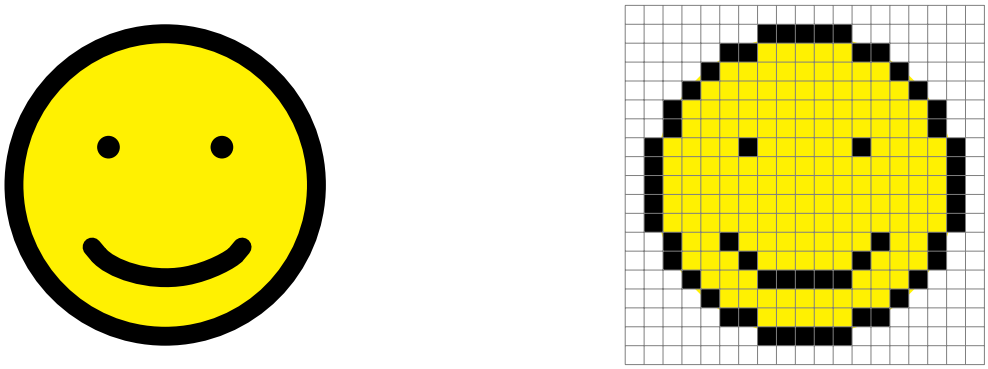
Figure 1.12: Polygon vertices are indexed in an anti-clockwise order.

Chapter 2

Rasterisation

2.1 Rasters

Computer displays use an array of small squares called pixels which are illuminated using different colours. When the array of pixels is viewed as a whole the brain interprets it as a single image (figure 2.1b). The pixel array is called a raster array and the process of determining the illumination of the pixels is called **rasterisation**. The image that is approximated as a raster array is known as the idealised image (figure 2.1a).



(a) idealised image

(b) rasterised image

Definition 9. A **raster** is a rectangular array of pixels that can be displayed on a screen.

Definition 10. A **pixel** is a small square that can be illuminated using different colours.

Definition 11. An **idealised image** is an analogue image that we want to approximate on a raster array.

2.1.1 The RGB colour model

The **RGB colour model** uses the three primary colours Red, Green and Blue (RGB) that are added to produce colours in the visible spectrum figure 2.2. Using a single bit for each primary colour (i.e., adding all of that colour or none of that colour) means that it is possible to produce eight colours: red, yellow, green, cyan, blue, magenta, black and white (table 2.1). Adding proportions of each primary colours means that many more colours can be produced. Using 8 bits for each primary colour means that are a possible $2^8 = 256$ different quantities of that colour. Combining the three primary colours means that the number of colours that can be produced is $2^8 \times 2^8 \times 2^8 = 2^{24} = 16,772,216$. It is estimated that the most number of colours that the human eye can distinguish is approximately 10 million so 24 bit colour (known as **true color**) is considered sufficient.

Figure 2.2: The RGB colour wheel

Table 2.1: The RGB codes for the colours produced using 3 bit colour.

Colour		Red	Green	Blue
Red		1	0	0
Yellow		1	1	0
Green		0	1	0
Cyan		0	1	1
Blue		0	0	1
Magenta		1	0	1
Black		0	0	0
White		1	1	1

2.1.2 Raster arrays

The information that defines a raster can be stored in a **raster array**. If a raster is n_x pixels wide by n_y pixels high then it can be defined either as an $n_y \times n_x$ array where each element contains a number that is linked to a colour corresponding to that pixel or as an $n_y \times n_x \times 3$ array where each pixel is defined using the three primary colours in the RGB colour model.

The MATLAB command `imread` can be used to read in the raster information for an image file and store it in an array. For example the following commands reads in the raster information for the image file `cavendish.jpg` into the array `img` and uses the `whos` commands to output its size.

```
img = imread('cavendish.jpg');
X = im2double(X);
whos img
```

This produces the output

```
  Name          Size                 Bytes  Class     Attributes

  img         240x320x3              230400  uint8
```

Here the raster array `img` is a 240 pixels high by 320 pixels wide.

2.1.3 Plotting raster arrays in MATLAB

An array containing raster information can be plotted in MATLAB using the `image` command. For example,

```
image(img)
```

produces the following plot shown in figure 2.3.

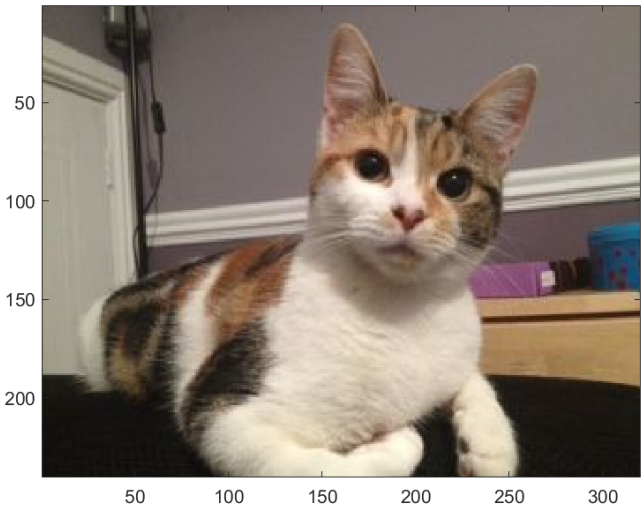


Figure 2.3: Raster array plotted using the `image` command.

2.1.4 Pixel co-ordinates

The co-ordinates of individual pixels in a raster are in the range $[0, n_x]$ and $[0, n_y]$ for the horizontal and vertical directions respectively where the co-ordinate $(0, 0)$ is the pixel in the top left-hand corner and the y co-ordinate increases as we move down the raster (this can be seen in the plot in figure 2.3). The reason for this that digital displays are refreshed using horizontal lines of pixels from top to bottom.

If $x, y \in [0, 1]$ are raster space co-ordinates then the corresponding pixel co-ordinates are

$$x_{pixel} = \lfloor x\, n_x \rfloor, \tag{6a}$$
$$y_{pixel} = \lfloor (1 - y) n_y \rfloor, \tag{6b}$$

where $\lfloor x \rfloor$ rounds x to the integer below (also known as the floor operator). Note that the y co-ordinate is subtracted from 1 to ensure that the pixel co-ordinates $(0, 0)$ correspond to the top-left hand element in the raster array.

2.2 Bresenham's algorithm

One of the fundamental tasks in computer graphics is the rendering of a straight line on a display. Consider the diagram showing the rasterisation of the straight line joining the two points at (x_0, y_0) and (x_1, y_1) shown in figure 2.4. The pixels that are closest to the line are shaded to create a rasterised approximation of the idealise image. This is a simple task for a human since we are able to view the idealised image and determine which pixels need to be shaded, however this is tricky for a computer. Therefore algorithms are required to determine which pixels to illuminate in order to rasterise straight lines.

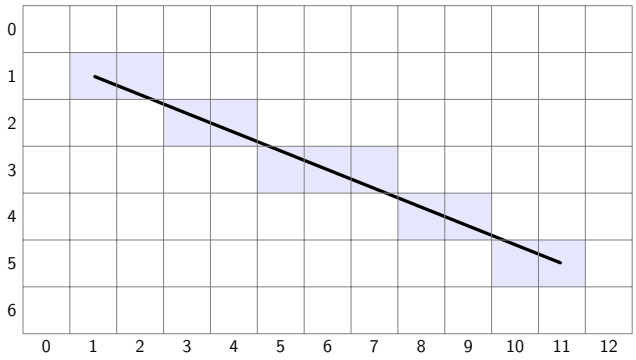


Figure 2.4: Rasterising a straight line.

Bresenham's algorithm (Bresenham 1965) is a line drawing algorithm that uses integer only arithmetic therefore offers a vast improvement over other methods in terms of computational efficiency.

Consider the straight line joining points (x_0, y_0) and (x_1, y_1)

$$y = \frac{\Delta y}{\Delta x}x + c,$$

where $\Delta x = x_1 - x_0$ and $\Delta y = y_1 - y_0$. Rearranging gives

$$0 = (\Delta y)x - (\Delta x)y + (\Delta x)c,$$

and defining the function $f(x, y)$ as

$$f(x, y) = (\Delta y)x - (\Delta x)y + (\Delta x)c,$$

then if $f(x, y) = 0$ the point (x, y) is on the line.

The premise behind Bresenham's algorithm is that we use the sign of the value of $f(x, y)$ at the midpoint between the two pixels whose centres lie either side of the idealised line to determine which of these pixels is plotted. Consider figure 2.5, we know that our idealised line will start at the pixel with co-ordinates (x_0, y_0) so we plot this pixel. Assuming that $x_0 < x_1$ and $y_0 < y_1$ so our line is drawn from left-to-right and top-to-bottom (we will consider the other cases later) we have a choice between the two pixels at $(x_0 + 1, y_0)$ and $(x_0 + 1, y_0 + 1)$ which to plot next. To decide which of these pixels is plotted we can use the value of $f(x_0 + 1, y_0 + \frac{1}{2})$ to determine which of these two pixels is closer to the idealised line. If $f(x_0 + 1, y_0 + \frac{1}{2}) < 0$ then the pixel at $(x_0 + 1, y_0)$ is closer to the idealised line and is plotted else the pixel at $(x_0 + 1, y_0 + 1)$ is closer and plotted.

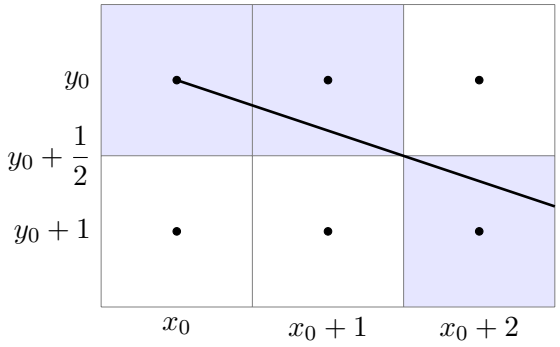


Figure 2.5: When rasterising a straight line we plot the pixel whose centre is closest to the line.

Example 1 Consider the line joining the two co-ordinates $(1, 1)$ and $(6, 4)$. The Cartesian equation of this line is

$$y = \frac{3}{5}x + \frac{2}{5},$$

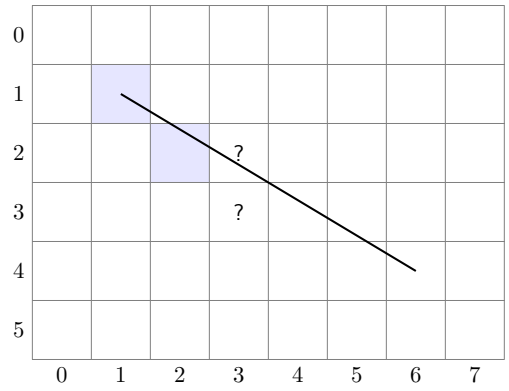
so

$$f(x, y) = 3x - 5y + 2.$$

We plot the first pixel at $(1, 1)$ and then have a choice between the pixels at $(2, 1)$ and $(2, 2)$. Calculating $f(2, 1.5)$ for these we have

$$f(2, 1.5) = 3(2) - 5(1.5) + 2 = 0.5,$$

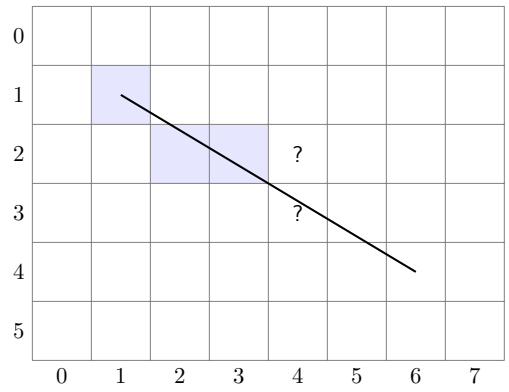
and since $f(2, 1.5) > 0$ the pixel $(2, 2)$ is closer to the idealised line and is plotted.



Continuing to the the next two pixels we have a choice between $(3, 3)$ and $(3, 4)$. Calculating $f(3, 2.5)$ for these we have

$$f(3, 2.5) = 3(3) - 5(2.5) + 2 = -1.5,$$

and since $f(3, 1.5) < 0$ the pixel at $(3, 2)$ is closer and is plotted.



We can continue in this way until we reach the last pixel at $(6, 4)$.

Whilst this method does perform as intended it has one major disadvantage that it requires floating point calculations (calculations involving non-integer quantities). Floating point operations are relatively expensive for a computer to calculate.

2.2.1 Derivation of Bresenham's algorithm

To derive an algorithm that uses integer only values we can define a difference D between the midpoint between two candidate pixels which is updated as we move along the line. Initially the pixel at (x_0, y_0) is plotted and we define D as

$$\begin{aligned} D &= f\left(x_0 + 1, y_0 + \frac{1}{2}\right) - f(x_0, y_0) \\ &= (\Delta y)(x_0 + 1) - (\Delta x)\left(y_0 + \frac{1}{2}\right) + (\Delta x)c - (\Delta y)x_0 + (\Delta x)y_0 - (\Delta x)c \\ &= (\Delta y)x_0 + \Delta y - (\Delta x)y_0 - \frac{1}{2}\Delta x - (\Delta y)x_0 + (\Delta x)y_0 \\ &= \Delta y - \frac{1}{2}\Delta x. \end{aligned} \tag{7}$$

If the value of $D \leq 0$ then the pixel at $(x_0 + 1, y_0)$ is closer to the idealised line is plotted, else if $D > 0$ we plot $(x_0 + 1, y_0 + 1)$ (figure 2.6)

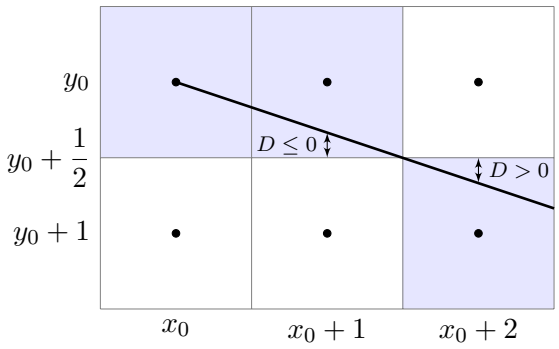


Figure 2.6: When rasterising a straight line we plot the pixel whose centre is closest to the line.

This value of D is updated by ΔD the value of which depends upon which of the two candidate pixels were plotted. If $D \leq 0$ then ΔD is

$$\begin{aligned} \Delta D &= f\left(x_0 + 2, y_0 + \frac{1}{2}\right) - f\left(x_0 + 1, y_0 + \frac{1}{2}\right) \\ &= (\Delta y)(x_0 + 2) - (\Delta x)\left(y_0 + \frac{1}{2}\right) + (\Delta x)c - (\Delta y)(x_0 + 1) + (\Delta x)\left(y_0 + \frac{1}{2}\right) - (\Delta x)c \\ &= (\Delta y)x_0 + 2\Delta y - (\Delta x)y_0 - \frac{1}{2}\Delta x_0 - (\Delta y)x_0 - \Delta y + (\Delta x)y_0 + \frac{1}{2}\Delta x \\ &= \Delta y. \end{aligned} \tag{8}$$

Else if $D > 0$ then ΔD is

$$\begin{aligned} \Delta D &= f\left(x_0 + 2, y_0 + \frac{3}{2}\right) - f\left(x_0 + 1, y_0 + \frac{1}{2}\right) \\ &= (\Delta y)(x_0 + 2) - (\Delta x)\left(y_0 + \frac{3}{2}\right) + (\Delta x)c - (\Delta y)(x_0 + 1) + (\Delta x)\left(y_0 + \frac{1}{2}\right) - (\Delta x)c \\ &= (\Delta y)x_0 + 2\Delta y - (\Delta x)y_0 - \frac{3}{2}\Delta x - (\Delta y)x_0 - \Delta y + (\Delta x)y_0 + \frac{1}{2}\Delta x \\ &= \Delta y - \Delta x. \end{aligned} \tag{9}$$

Equation (7) includes a floating point number in $\frac{1}{2}$ and since we are only interested in the sign of D and not its value, we can multiply equations (7) to (9) by 2 to give

$$D = 2\Delta y - \Delta x, \tag{10}$$

$$\Delta D = \begin{cases} 2\Delta y, & D \leq 0, \\ 2\Delta y - 2\Delta x, & D > 0, \end{cases} \tag{11}$$

which are all integer only expressions.

Bresenham's algorithm is presented in formal algorithmic notation in algorithm 1.

Algorithm 1 Bresenham's algorithm

function DRAWLINE(R, x_0, y_0, x_1, y_1, *colour*)
  Initialise $\Delta x \leftarrow x_1 - x_0$, $\Delta y \leftarrow y_1 - y_0$, $D \leftarrow 2\Delta y - \Delta x$ and $y \leftarrow y_0$
  for $x := x_0 \dots x_1$ **do**
    $R(y, x) \leftarrow colour$ ▷ Note that (y, x) uses matrix indexing
    if $D > 0$ **then**
      $y \leftarrow y + 1$
      $D \leftarrow D - 2\Delta x$
    end if
    $D \leftarrow D + 2\Delta y$
  end for
  return R
end function

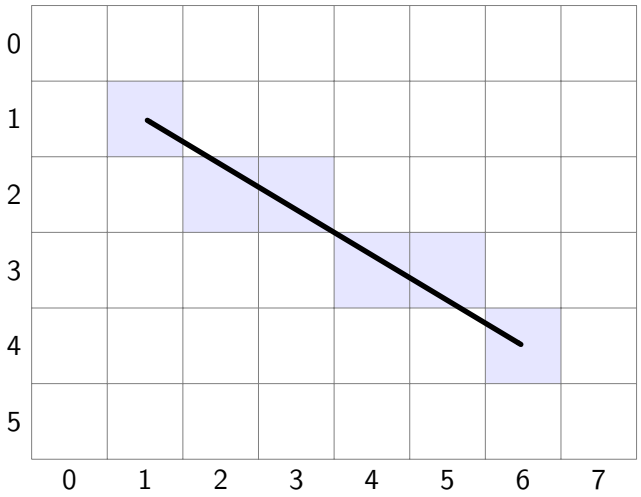
Example 2 Use Bresenham's algorithm to determine the co-ordinates of the pixels on the line joining the two pixels with co-ordinates $(1, 1)$ and $(6, 4)$.

$\Delta x = 6 - 1 = 5$, $\Delta y = 4 - 1 = 3$, $D = 2(3) - 5 = 1$.

Looping through x values from 1 to 6

$$
\begin{array}{llll}
x = 1, & y = 1, & D = 1 > 0 & \therefore D = 1 + 2(3) - 2(5) = -3, \\
x = 2, & y = 2, & D = -3 \leq 0 & \therefore D = -3 + 2(3) = 3, \\
x = 3, & y = 2, & D = 3 > 0 & \therefore D = 3 + 2(3) - 2(5) = -1, \\
x = 4, & y = 3, & D = -1 \leq 0 & \therefore D = -1 + 2(3) = 5, \\
x = 5, & y = 3, & D = 5 > 0 & \therefore D = 5 + 2(3) - 2(4) = 1, \\
x = 6, & y = 4. & &
\end{array}
$$

So the pixels at $(1, 1)$, $(2, 2)$, $(3, 2)$, $(4, 3)$, $(5, 3)$ and $(6, 4)$ are plotted.



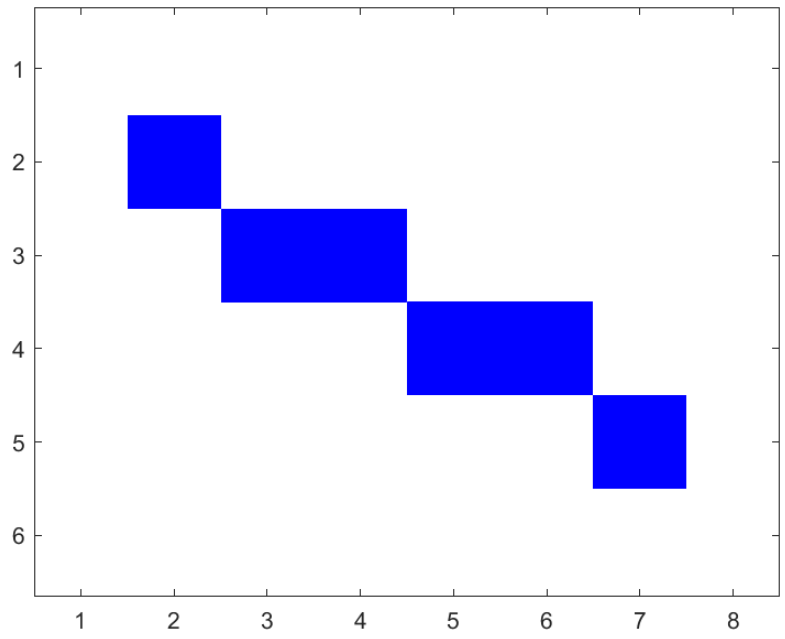
2.2.2 MATLAB code

The MATLAB code in listing 2.1 uses Bresenham's algorithm to draw the straight line from example 2. The array `colour` is a 3×1 array that defines the line colour which is blue in this case since $R = 0$, $G = 0$ and $B = 1$. The raster array `img` is a $6 \times 8 \times 3$ array initialised so that all of its values are ones so the background colour will be white. The function `drawline` contains Bresenham's algorithm so that it can

be used to draw a line given the background raster array, the start and end co-ordinates and the colour of the line. Note that we need to add 1 to the x and y co-ordinates when plotting each pixel since MATLAB arrays start indexing from 1 and not 0.

Listing 2.1: MATLAB program that uses Bresenham's algorithm to draw a straight line.

```
% Initialise raster array
nx = 8;
ny = 6;
img = ones(ny, nx, 3);

% Define start and end co-ordinates and line colour
x0 = 1;
y0 = 1;
x1 = 6;
y1 = 4;
colour = [ 0 ; 0 ; 1 ];

% Draw line
img = drawline(img, x0, y0, x1, y1, colour);

% Plot raster
image(img)
axis equal


function raster = drawline(raster, x0, y0, x1, y1, colour)

dx = x1 - x0;
dy = y1 - y0;
D = 2 * dy - dx;
y = y0;

for x = x0 : x1
   raster(y+1, x+1, :) = colour;
   if D > 0
       D = D - 2 * dx;
       y = y + 1;
   end
    D = D + 2 * dy;
end

end
```



2.2.3 Drawing lines in all directions

The algorithm presented in algorithm 1 only works for lines that are drawn going down and to the right with a gradient less than 1 (i.e., $x_0 < x_1$, $y_0 < y_1$ and $\Delta x > \Delta y$). The basic algorithm can be extended by considering steps in the x and y directions separately which results in the algorithm shown in algorithm 2. Note that in the modified algorithm we introduce a new variable E which is used tests to determine whether the x and y co-ordinates are incremented.

Algorithm 2 Bresenham's algorithm for drawing lines in all directions

function DRAWLINE(R, x_0, x_1, y_0, y_1, $colour$)
  Initialise $\Delta x \leftarrow |x_1 - x_0|$, $\Delta y \leftarrow |y_1 - y_0|$ and $D \leftarrow \Delta x - \Delta y$
  Calculate $s_x \leftarrow \begin{cases} 1, & x_0 < x_1, \\ -1, & x_0 > x_1, \end{cases}$ and $s_y \leftarrow \begin{cases} 1, & y_0 < y_1, \\ -1, & y_0 > y_1. \end{cases}$
  while true **do**
    $R(y_0, x_0) \leftarrow colour$
    if $x_0 = x_1$ and $y_0 = y_1$ **then**
      return R ▷ Exit function when last pixel has been plotted
    end if
    $E \leftarrow 2D$
    if $E \geq -\Delta y$ **then**
      $x_0 \leftarrow x_0 + s_x$
      $D \leftarrow D - \Delta y$
    end if
    if $E \leq \Delta x$ **then**
      $y_0 \leftarrow y_0 + s_y$
      $D \leftarrow D + \Delta x$
    end if
  end while
end function

Example 3 Use Bresenham's algorithm for drawing lines in any direction to determine the co-ordinates of the pixels on the lines joining the following points:

(i) (0,0) and (4,6);

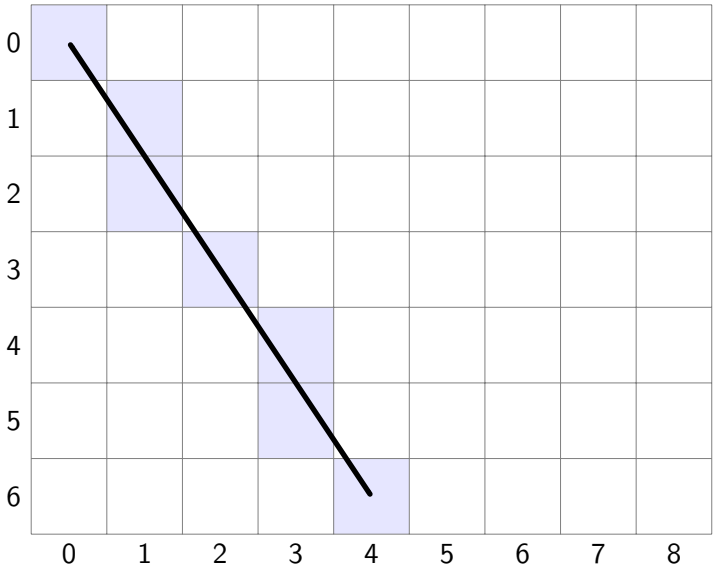
(ii) (6,1) and (2,4);

Solution:

(i) $\Delta x = 4 - 0 = 4$, $\Delta y = 6 - 0 = 6$, $s_x = 1$, $s_y = 1$, $D = 4 - 6 = -2$.

Stepping through the algorithm:

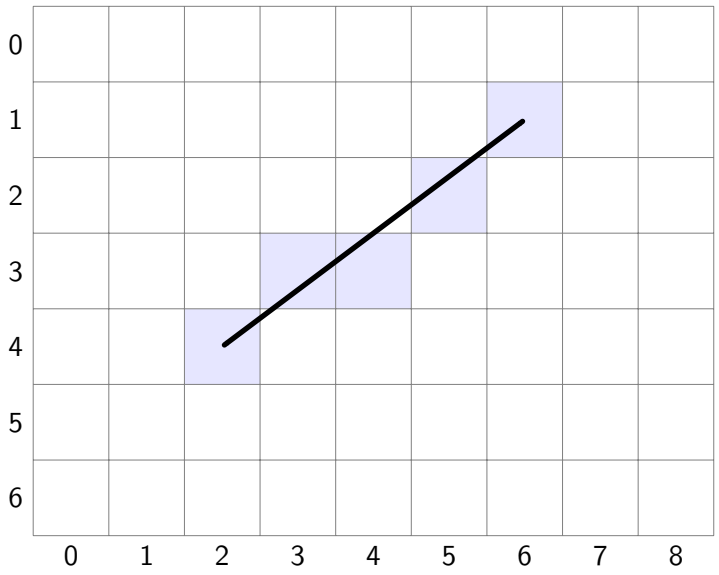
$$
\begin{array}{llllll}
x_0 = 0, & y_0 = 0, & E = 2(-2) = -4 \geq -\Delta y & \therefore & x_0 = 0 + 1 = 1, & D = -2 - 6 = -8, \\
 & & E = -4 \leq \Delta x & \therefore & y_0 = 0 + 1 = 1, & D = -8 + 4 = -4, \\
x_0 = 1, & y_0 = 1, & E = 2(-4) = -8 < -\Delta y, & & & \\
 & & E = -8 \leq \Delta x & \therefore & y_0 = 1 + 1 = 2, & D = -4 + 4 = 0, \\
x_0 = 1, & y_0 = 2, & E = 2(0) = 0 \geq -\Delta y & \therefore & x_0 = 1 + 1 = 2, & D = 0 - 6 = -6, \\
 & & E = 0 \leq \Delta x & \therefore & y_0 = 2 + 1 = 3, & D = -6 + 4 = -2, \\
x_0 = 2, & y_0 = 3, & E = 2(-2) = -4 \geq -\Delta y & \therefore & x_0 = 2 + 1 = 3, & D = -2 - 6 = -8, \\
 & & E = -4 \leq \Delta x & \therefore & y_0 = 3 + 1 = 4, & D = -8 + 4 = -4, \\
x_0 = 3, & y_0 = 4, & E = 2(-4) = -8 < -\Delta y, & & & \\
 & & E = -8 \leq \Delta x & \therefore & y_0 = 4 + 1 = 5, & D = -4 + 4 = 0, \\
x_0 = 3, & y_0 = 5, & E = 2(0) = 0 \geq -\Delta y & \therefore & x_0 = 3 + 1 = 4, & D = 0 - 6 = -6, \\
 & & E = 0 \leq \Delta x & \therefore & y_0 = 5 + 1 = 6, & D = -6 + 4 = -2, \\
x_0 = 4, & y_0 = 6. & & & &
\end{array}
$$



(ii) $\Delta x = |2-6| = 4$, $\Delta y = |4-1| = 3$, $s_x = -1$, $s_y = 1$, $D = 4-3 = 1$.

Stepping through the algorithm:

$$
\begin{array}{llllll}
x_0 = 6, & y_0 = 1, & E = 2(1) = 2, & E \geq -\Delta y & \therefore & x_0 = 6-1 = 5, \quad D = 1-3 = -2, \\
 & & & E \leq \Delta x & \therefore & y_0 = 1+1 = 2, \quad D = -2+4 = 2, \\
x_0 = 5, & y_0 = 2, & E = 2(2) = 4, & E \geq -\Delta y & \therefore & x_0 = 5-1 = 4, \quad D = 2-3 = -1, \\
 & & & E \leq \Delta x & \therefore & y_0 = 2+1 = 3, \quad D = -1+4 = 3, \\
x_0 = 4, & y_0 = 3, & E = 2(3) = 6, & E \geq -\Delta y & \therefore & x_0 = 4-1 = 3, \quad D = 3-3 = 0, \\
 & & & E > \Delta x & & \\
x_0 = 3, & y_0 = 3, & E = 2(0) = 0, & E \geq -\Delta y & \therefore & x_0 = 3-1 = 2, \\
 & & & E \leq \Delta x & \therefore & y_0 = 3+1 = 4, \\
x_0 = 2, & y_0 = 4. & & & &
\end{array}
$$



2.3 Drawing circles

Now that we have the ability to rasterise straight lines the next fundamental problem is the rasterisation of circles. This can be done by deriving a Bresenham-type algorithm that uses the Cartesian equation of a circle to determine which of two candidate pixels are plotted.

2.3.1 Circle symmetry

The circle line drawing algorithms can use the concept of **circle symmetry** to reduce the number of computations required to rasterise a circle. Consider figure 2.7 where a circle is centred at the origin, if the coordinate of a point on the circle in the shaded octant where $x > y$ is known (x, y) then the corresponding points on the circle in the other seven octants can be found via circle symmetry through combinations of $(\pm x, \pm y)$ and $(\pm y, \pm x)$.

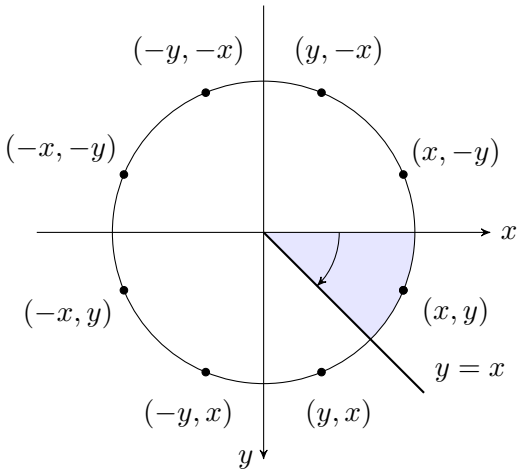


Figure 2.7: Given the co-ordinates of the point (x, y), the corresponding points in the seven other octants can be found using circle symmetry.

For circles not centred at the original, which will be the case in the vast majority of applications, we calculate the co-ordinates of a point on the circle in the first octant (x, y) and then add the co-ordinates of the circle centre (c_x, c_y) giving the following eight pixel co-ordinates

$$\begin{array}{llll} (c_x + x, c_y + y), & (c_x + x, c_y - y), & (c_x - x, c_y + y), & (c_x - x, c_y - y), \\ (c_x + y, c_y + x), & (c_x + y, c_y - x), & (c_x - y, c_y + x), & (c_x - y, c_y - x). \end{array}$$

2.3.2 The midpoint algorithm

The **midpoint algorithm** (Pitteway 1967) is a form of Bresenham's algorithm that is used to draw circles. Consider the Cartesian equation of a circle centred at $(0, 0)$ with radius r

$$x^2 + y^2 = r^2,$$

which an be rearranged to give

$$0 = x^2 + y^2 - r^2,$$

and defining the function $f(x, y)$ as

$$f(x, y) = x^2 + y^2 - r^2,$$

then if $f(x, y) = 0$ we know that the point (x, y) is on the circle.

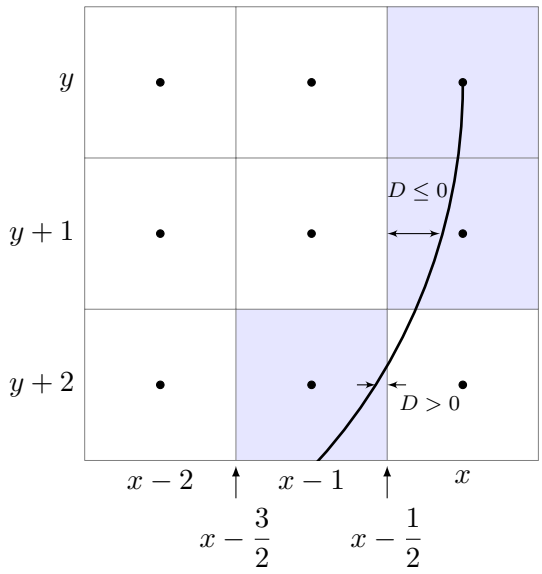


Figure 2.8: The midpoint algorithm uses the distance between the midpoint between two pixels and the ideal line to determine which pixel to illuminate.

To derive an algorithm to rasterise a circle we assume that the circle is centred at $(0, 0)$ and we start at the pixel at the 3 o'clock position with pixel co-ordinates $(r, 0)$. We then move clockwise around the circle calculating the pixels that lie closest to the idealised circle until $x = y$ which is the end of the first octant. For each pixel co-ordinates we calculate, the pixel co-ordinates of the corresponding pixels in the 7 other octants are calculated using circle symmetry.

Once the first pixel $(r, 0)$ is plotted we move down by one pixel so that $y = 1$ and we have a choice between the two pixels at $(r, 1)$ and $(r-1, 1)$ to plot next. Similar to Bresenham's line drawing algorithm, we define a value D as the difference between the value of $f(x, y)$ between the first pixel $(r, 0)$ and the midpoint between the two pixels $(r, 1)$ and $(r - 1, 1)$.

$$\begin{aligned} D &= f\left(r - \frac{1}{2}, 1\right) - f(r, 0) \\ &= \left(r - \frac{1}{2}\right)^2 + 1^2 - r^2 - r^2 - 0^2 + r^2 \\ &= r^2 - r + \frac{1}{4} + 1 - r^2 \\ &= \frac{5}{4} - r. \end{aligned} \tag{12}$$

If $D \leq 0$ then we plot the pixel at $(x, y + 1)$ (figure 2.8) and the change in the value of D is

$$\begin{aligned} \Delta D &= f\left(x - \frac{1}{2}, y + 2\right) - f\left(x - \frac{1}{2}, y + 1\right) \\ &= \left(x - \frac{1}{2}\right)^2 + (y + 2)^2 - r^2 - \left(x - \frac{1}{2}\right)^2 - (y + 1)^2 + r^2 \\ &= y^2 + 4y + 4 - y^2 - y - 1 \\ &= 2y + 3. \end{aligned} \tag{13}$$

Else if $D > 0$ then we plot the pixel at $(x-1, y+1)$ and the change in the value of D is

$$\begin{aligned}\Delta D &= f\left(x - \frac{3}{2}, y + 2\right) - f\left(x - \frac{1}{2}, y + 1\right) \\ &= \left(x - \frac{3}{2}\right)^2 + (y+2)^2 - r^2 - \left(x - \frac{1}{2}\right)^2 - (y+1)^2 + r^2 \\ &= x^2 - 3x + \frac{9}{4} + y^2 + 4y + 4 - x^2 + x - \frac{1}{4} - y^2 - 2y - 1 \\ &= 2y - 2x + 5. \end{aligned} \tag{14}$$

Equation (12) contains a floating point number in $\frac{5}{4}$, similar to Bresenham's algorithm we can multiply equations (12) to (14) by 4 so that we have integer only expressions for D and ΔD

$$D = 5 - 4r, \tag{15}$$

$$\Delta D = \begin{cases} 8y + 12, & D > 0, \\ 8y - 8x + 20, & D \leq 0. \end{cases} \tag{16}$$

The midpoint algorithm for rasterising circles is shown in algorithm 3.

Algorithm 3 The midpoint circle drawing algorithm

function DRAWCIRCLE(R, c_x, c_y, r, $colour$)
  Initialise $D \leftarrow 5 - 4r$, $x \leftarrow r$ and $y \leftarrow 0$
  while $y \leq x$ **do**
    $R(c_x \pm x, c_y \pm y) \leftarrow colour$
    $R(c_x \pm y, c_y \pm x) \leftarrow colour$
    if $D > 0$ **then**
      $D \leftarrow D - 8x + 8$
      $x \leftarrow x - 1$
    end if
    $D \leftarrow D + 8y + 12$
    $y \leftarrow y + 1$
  end while
  return R
end function

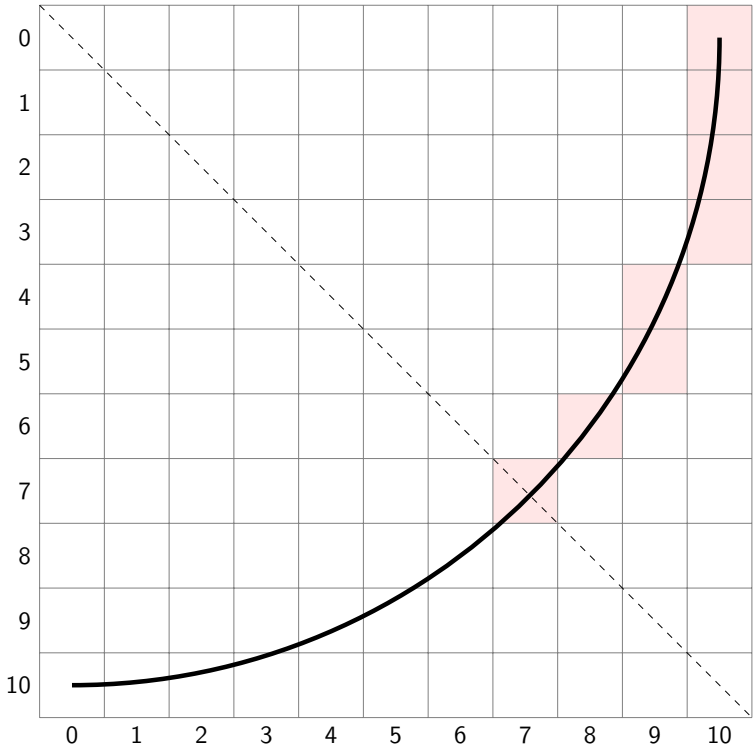
Example 4 Use the midpoint algorithm to rasterise a circle centred at $(11, 11)$ with radius 10.

$x = 10$, $y = 0$, $D = 5 - 4(10) = -35$

Stepping through the algorithm:

$$
\begin{array}{llll}
x = 10, & y = 0, & D \leq 0, & \therefore D = -35 + 8(0) + 12 = -23, \\
x = 10, & y = 1, & D \leq 0, & \therefore D = -23 + 8(1) + 12 = -3, \\
x = 10, & y = 2, & D \leq 0, & \therefore D = -3 + 8(2) + 12 = 25, \\
x = 10, & y = 3, & D > 0, & \therefore D = 25 + 8(3) - 8(10) + 20 = -11, \\
x = 9, & y = 4, & D \leq 0, & \therefore D = -11 + 8(4) + 12 = 33, \\
x = 9, & y = 5, & D > 0, & \therefore D = 33 + 8(5) - 8(9) + 20 = 21, \\
x = 8, & y = 6, & D > 0, & \therefore D = 21 + 8(6) - 8(8) + 20 = 25, \\
x = 7, & y = 7. & &
\end{array}
$$

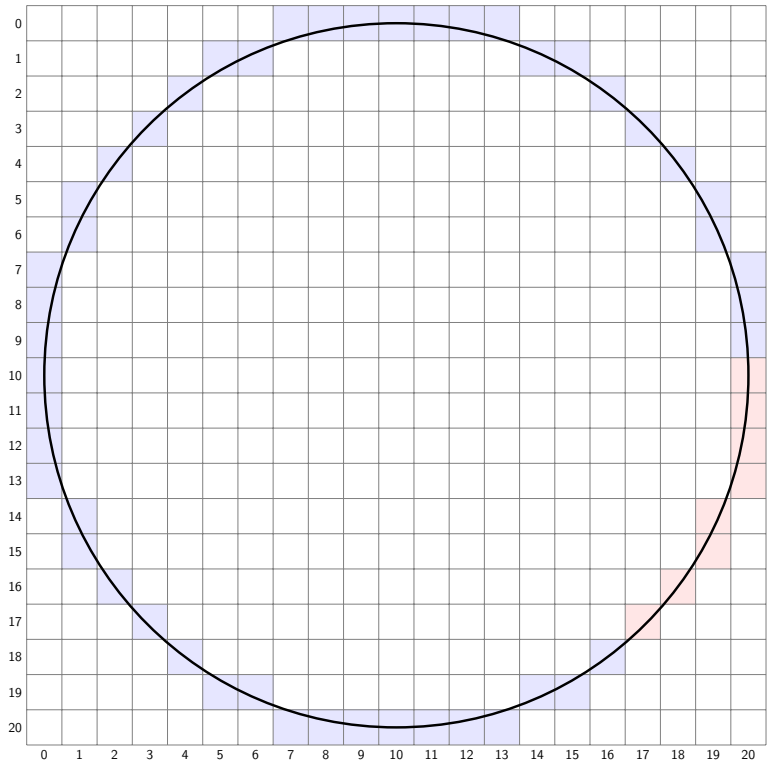
Plotting the pixels in the first octant:



Adding the centre co-ordinates gives

$$(21, 11), \quad (21, 12), \quad (21, 13), \quad (21, 14), \quad (20, 15), \quad (20, 16), \quad (19, 17), \quad (18, 18).$$

Plotting the complete circle:


0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

2.4 Drawing polygons

As well as being able to draw lines on a raster display we also need to be able to draw polygons. Drawing polygons can be achieved using one of two approaches, we can either draw the outline of the polygon using line drawing algorithms and then fill in all of the pixels inside the polygon or just draw the polygon, including the pixels in the interior, all at once. A successful method should produce a filled polygon with no holes in the interior of the polygon, i.e., all pixels that are contained within a polygon should be illuminated using the desired colour. These notes discuss two algorithms that use each of these approaches: the flood fill algorithm and the scanline fill algorithm.

2.4.1 The flood fill algorithm

The **flood fill algorithm** is used to fill a polygon that has been rendered on a raster array using the line drawing algorithms seen in chapter 2. Given the position of a starting pixel (x_0, y_0) that is known to be in the interior of the polygon, the flood fill algorithm illuminates that pixel and moves to a neighbouring pixel either to the east, west, north or south direction. At each pixel, a check is performed to see whether the pixel requires illuminating. A pixel is only illuminated if its current colour is the same as a $target$ colour. If the pixel is to be illuminated then its colour is set to that of the $replacement$ colour and the process is restarted using an adjacent pixel.

To keep track of the pixels that need to be considered a LIFO (Last In First Out) queue is used. The queue is initialised so that it only contains the starting pixel (x_0, y_0). At each iteration of the method the last pixel in the queue is removed and is checked to see if the current colour of the pixel is the same as the target colour. If it is then this the pixel is plotted using the replacement colour and the four neighbouring pixels to the right, left, bottom and top are added onto the end of the queue. This continues until the queue is empty.

Algorithm 4 The flood fill algorithm

```
function FloodFill(R, x_0, y_0, target colour, replacement colour)
    Q ← {(x_0, y_0)}
    while Q ≠ ∅ do
        (x, y) ← last pixel in Q
        Remove last pixel from Q
        if R(x, y) = target colour then
            R(x, y) ← replacement colour
            Append pixels (x + 1, y), (x − 1, y), (x, y + 1) and (x, y − 1) to Q
        end if
    end while
    return R
end function
```

Example 5 Starting with the pixel at $(3, 4)$, use the flood fill algorithm to fill in the polygon on the raster in figure 2.9a where the target colour is white and the replacement colour is red.

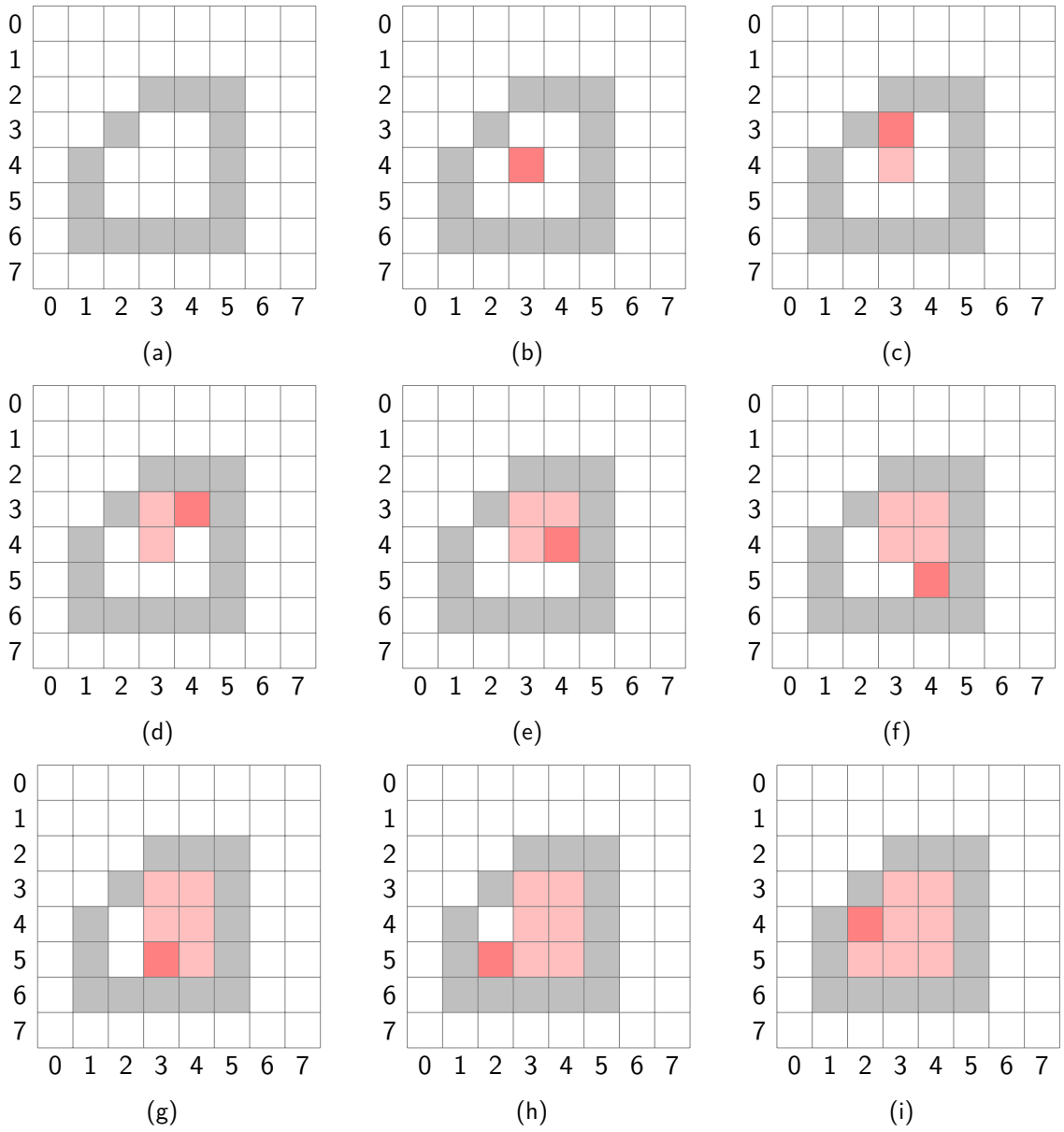


Figure 2.9: Flood fill example

We begin by initialising the queue to contain the starting pixel

$$Q = \{(3, 4)\}.$$

We remove this pixel from the queue and check the current colour is the same as the target colour, which it is so we fill this pixel red (figure 2.9b) and append the pixels to the right, left, bottom and top of pixel $(3, 4)$ to the queue.

$$Q = \{\underbrace{(4, 4), (2, 4), (3, 5), (3, 3)}_{\text{neighbouring pixels to } (3,4)}\}.$$

We now remove pixel $(3, 3)$ from Q. Its colour is the same as the target colour so we fill this pixel red (figure 2.9c) and append the neighbouring pixels to the end of Q.

$$Q = \{(4, 4), (2, 4), (3, 5), \underbrace{(4, 3), (2, 3), (3, 4), (3, 2)}_{\text{neighbouring pxiels to } (3,3)}\}.$$

We now remove pixel $(3, 2)$ from Q. Its colour is not the same as the target colour so we reject this and remove the next last pixel in Q, (3, 4). This is also not the same as the target colour so is rejected and we remove pixel $(2, 3)$. This is also not the same as the target colour so is rejected and we remove pixel $(4, 3)$. This is the same as the target colour so we fill this pixel red (figure 2.9d) and append the neighbouring pixels to the end of Q.

We proceed in the same way that results in the the following queue. The underlined pixel denotes the pixel that is plotted in each step.

$$Q = \{(4,4),(2,4),(3,5),(5,3),(3,3),\underline{(4,4)},(4,3)\},$$
$$Q = \{(4,4),(2,4),(3,5),(5,3),(3,3),(5,4),(3,4),\underline{(4,5)},(4,3)\},$$
$$Q = \{(4,4),(2,4),(3,5),(5,3),(3,3),(5,4),(3,4),(5,5),\underline{(3,5)},(4,6),(4,4)\},$$
$$Q = \{(4,4),(2,4),(3,5),(5,3),(3,3),(5,4),(3,4),(5,5),(4,5),\underline{(2,5)},(3,6),(3,4)\},$$
$$Q = \{(4,4),(2,4),(3,5),(5,3),(3,3),(5,4),(3,4),(5,5),(4,5),(3,5),(1,5),(2,6),\underline{(2,4)}\},$$
$$Q = \emptyset.$$

2.4.2 Use of the flood fill algorithm

Since the flood fill algorithm uses adjacent pixels to the four compass directions to spread the fill colour across a polygon is cannot spread across tight corners (figure 2.10). This can be desirable since if the width of the outline is a single pixel then the flood fill will not leak outside of the polygon. In practice the flood fill algorithm is too computationally expensive to be used for virtual worlds and is only really used in drawing applications to provide a filling tool.

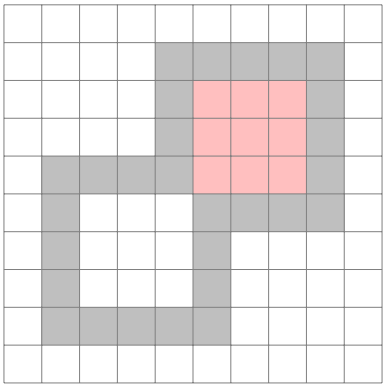


Figure 2.10: The flood fill algorithm is blocked by tight corners.

2.4.3 Scanline filling algorithm

The **scanline filling algorithm** is a method of rendering a polygon without drawing the edges beforehand unlike that flood fill algorithm that requires an outline. Instead of testing pixels one by one, the scanline filling algorithm loops through horizontal rows of pixels (known as a **scanline**) starting from the vertex with the smallest y co-ordinate and going down to the vertex with the largest y co-ordinate. Every edge of a polygon is tested to see whether it intersects with the current scanline. If an edge intersects with the scanline, the co-ordinates of the intersection points, known as **scan extrema**, are calculated using linear interpolation and all of the pixels between pairs of intersection points are filled.

Definition 12. A **scanline** is a row of pixels on a display raster.

Definition 13. The **scan extrema** are the intersection points between a scanline and an edge of a polygon.

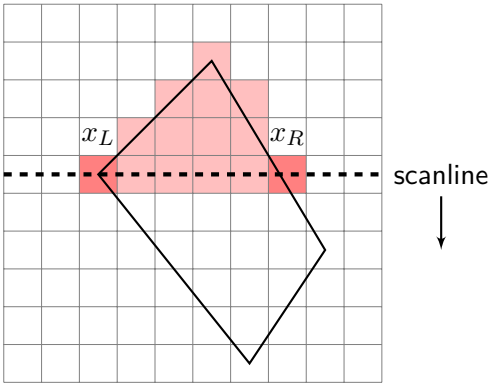


Figure 2.11: The row of pixels between the scan extrema points x_L and x_R inclusive are illuminated for each scanline.

The co-ordinates of the scan extrema are calculated using **linear interpolation** between the two vertices that define an edge. Consider figure 2.12 where a polygon is defined by the vertices (x_0, y_0), (x_1, y_1) and (x_2, y_2) listed in anti-clockwise order. For each scanline there are two scan extrema with co-ordinates (x_L, y) and (x_R, y) for the left and right-hand side of the polygon respectively. Using the scanline algorithm we begin at the co-ordinate with the smallest y value which is (x_2, y_2) in this case so we initialise $x_L = x_R = x_2$ and $y = y_2$. Moving down the scanlines the y co-ordinate is incremented by 1 and the x_L and x_R co-ordinates are calculated using the previous values.

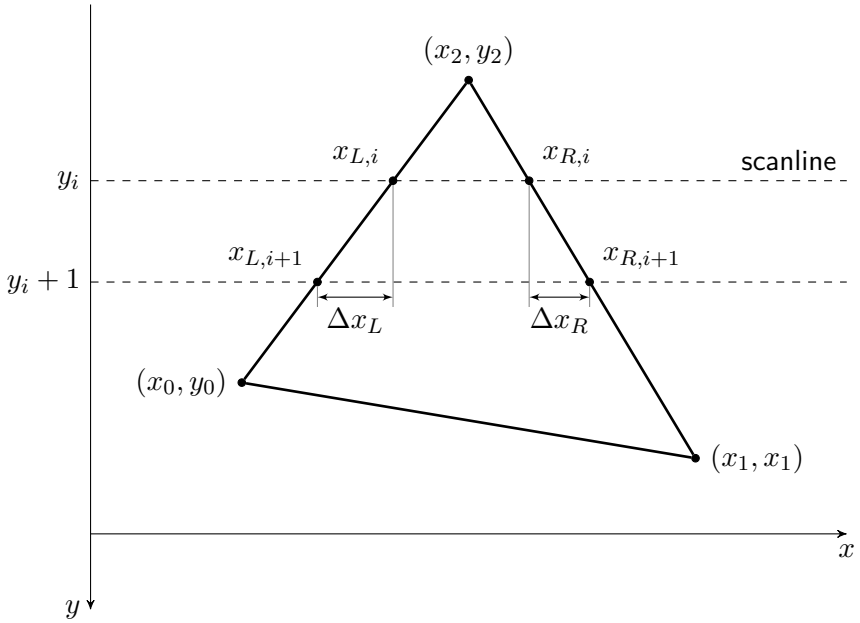


Figure 2.12: Calculating the scan extrema co-ordinates.

To calculate the x co-ordinate of the scan extrema we can use the Cartesian equation of a straight line

$$y = \frac{\Delta y}{\Delta x}x + c,$$

where $\Delta y = y_0 - y_2$ and $\Delta x = x_0 - x_2$ for the left edge and $\Delta y = y_1 - y_2$ and $\Delta x = x_1 - x_2$ for the right edge. Assuming that $(x_{L,i}, y_i)$ is a known scan extrema co-ordinate on the left edge we can determine the value of the constant c using

$$c = y_i - \frac{\Delta y}{\Delta x}x_{L,i},$$

Substituting this expression into the equation of a straight line and rearranging to make x the subject gives:

$$x = x_{L,i} + \frac{\Delta x}{\Delta y}(y - y_i).$$

Let $x = x_{L,i+1}$ and $y = y_i + 1$ then

$$x_{L,i} = x_{L,i} + \frac{\Delta x}{\Delta y}(y_i + 1 - y_i) = x_{L,i} + \frac{\Delta x}{\Delta y}$$

The value of $\dfrac{\Delta x}{\Delta y}$ is constant for all points along that edge so can be pre-calculated prior to looping through the scanlines and is updated when the polygon edge that intersects the scanline changes. The interpolating equations are

$$x_L = x_L + \Delta x_L, \tag{17a}$$
$$x_R = x_R + \Delta x_R, \tag{17b}$$
$$\Delta x_L = \frac{x_p - x_q}{y_p - y_q}, \tag{17c}$$
$$\Delta x_R = \frac{x_r - x_s}{y_r - y_s}, \tag{17d}$$

where p, q and r, s are the indices of the upper and lower vertices for the left and right-hand edges respectively. Recall from section 1.4 on page 7 that it is customary to label vertices in an anti-clockwise direction so for figure 2.12, $p = 2$, $q = 0$, $r = 2$ and $s = 1$. This means for an n-sided polygon, for the left edges we have $q = p + 1$ unless $p = n$ where we use $q = 0$. Similarly, for the right edges, $s = r - 1$ unless $r = 0$ where we use $s = n$. This can be represented using the following case statements

$$q = \begin{cases} p + 1, & p < n, \\ 0, & \text{otherwise,} \end{cases} \tag{18a}$$
$$s = \begin{cases} r - 1, & r > 1, \\ n, & \text{otherwise.} \end{cases} \tag{18b}$$

Algorithm 5 Scanline filling algorithm

function DRAWPOLYGON(R, $(x_0, \ldots, x_n)$, $(y_0, \ldots, y_n)$, $fill\ colour$)
  Initialise p and r to the index of the smallest y polygon vertex
  Initialise q and s using equations (18a) and (18b)
  Initialise $x_L \leftarrow x_p$ and $x_R \leftarrow x_r$
  Calculate Δx_L and Δx_R using equations (17c) and (17d)
  for $y = \min(y) \ldots \max(y)$ **do** ▷ Loop through scanlines
    for $x = \text{round}(x_L) \ldots \text{round}(x_R)$ **do** ▷ Loop across pixels in the current scanline
      $R(y, x) \leftarrow fill\ colour$
    end for
    if $y = y_q$ **then** ▷ left edge changes
      Update $p \leftarrow q$
      Update q using equation (18a)
      Recalculate Δx_L for new left edge using equation (17c)
    end if
    if $y = y_s$ **then** ▷ right edge changes
      Update $r \leftarrow s$
      Update s using equation (18b)
      Recalculate Δx_R for new right edge using equation (17d)
    end if
    Recalculate $x_L \leftarrow x_L + \Delta x_L$ and $x_R \leftarrow x_R + \Delta x_R$ using equations (17a) and (17b)
  end for
  return R
end function

The result of using the scanline algorithm to draw a polygon can be seen in figure 2.13.

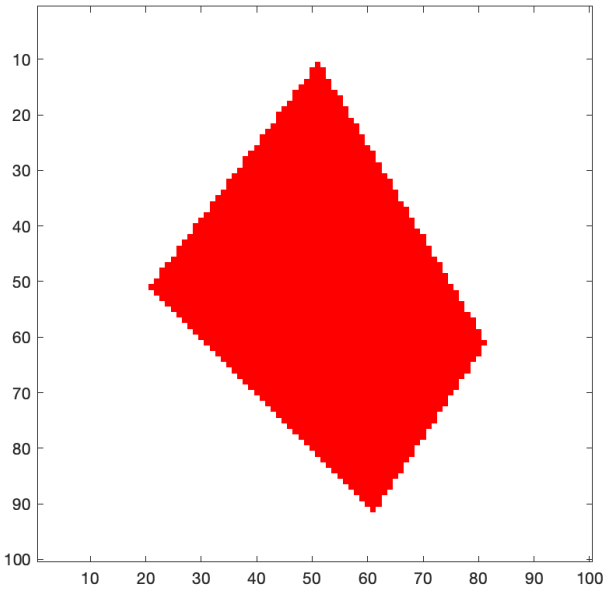
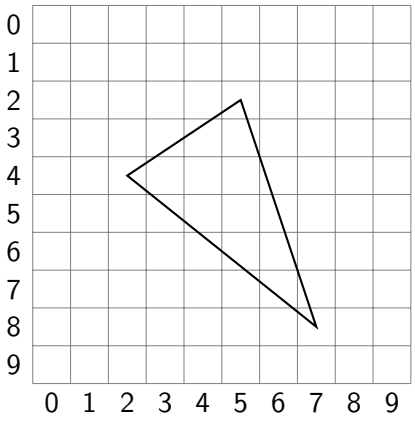


Figure 2.13: Polygon drawn using the scanline filling algorithm.

Example 6 Use the scanline filling algorithm to draw the polygon shown on the raster below.

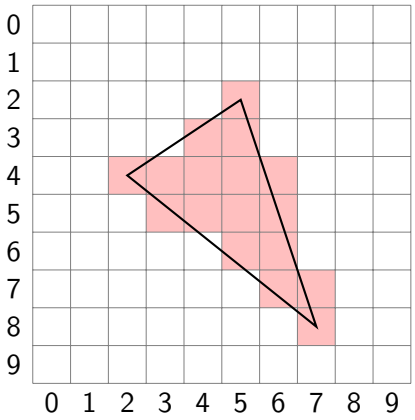


The vertex with the smallest y co-ordinate is at $(5, 2)$ so we initialise $x_L = x_R = 5$ and $y = 2$. The left-hand edge joins with the vertex at $(2, 4)$ so $\Delta x_L = \dfrac{2-5}{4-2} = -1.5$ (remember that we are assuming vertices are listed in the anti-clockwise direction). The right-hand edge joins with the vertex at $(7, 8)$ so $\Delta x_R = \dfrac{5-7}{2-8} = 0.33$.

$y = 2,$	$x_L = 5,$	$x_R = 5,$	$\therefore$ illuminate (5, 2)
$y = 3,$	$x_L = 5 - 1.5 = 3.5,$	$x_R = 5 + 0.33 = 5.33,$	$\therefore$ illuminate (4, 3) to (5, 3)
$y = 4,$	$x_L = 3.5 - 1.5 = 2,$	$x_R = 5.33 + 0.33 = 5.67,$	$\therefore$ illuminate (2, 4) to (6, 4).

Now we have reached the end vertex of the left-hand edge so we move to the next edge which has endpoints at $(2, 4)$ and $(7, 8)$ so $\Delta x_L = \dfrac{7-2}{8-4} = 1.25$.

$y = 5,$	$x_L = 2 + 1.25 = 3.25,$	$x_R = 5.67 + 0.33 = 6,$	$\therefore$ illuminate (3, 5) to (6, 5),
$y = 6,$	$x_L = 3.25 + 1.25 = 4.5,$	$x_R = 6 + 0.33 = 6.33,$	$\therefore$ illuminate (5, 6) to (6, 6),
$y = 7,$	$x_L = 4.5 + 1.25 = 5.75,$	$x_R = 6.33 + 0.33 = 6.67,$	$\therefore$ illuminate (6, 7) to (7, 7),
$y = 8,$	$x_L = 5.75 + 1.25 = 7,$	$x_R = 6.67 + 0.33 = 7,$	$\therefore$ illuminate (7, 8).



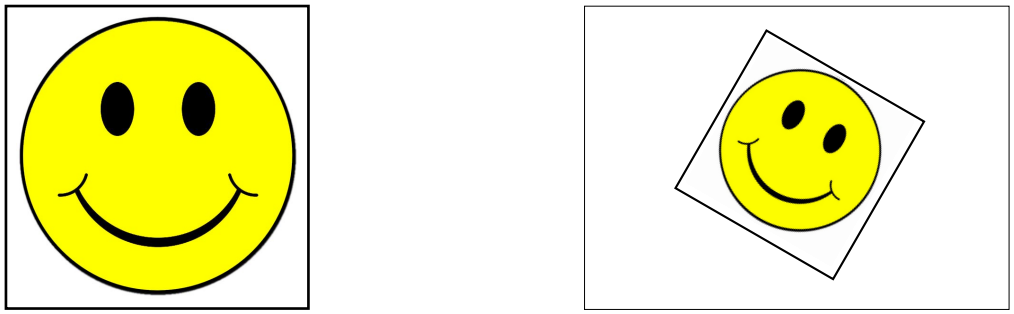
2.5 Texture Mapping

Texture mapping is the process of mapping a two-dimensional image onto a three-dimensional surface by transforming colour data so it conforms to the surface plot. It allows us to apply a texture, such as bumps or wood grain to a surface without performing the geometric modelling necessary to create a surface with these textures. The image that is mapped to a surface is called a **texture** and is a raster containing an array of pixels called **textels**. The colour of a textel will determine the colour of a pixel in the raster array.

Definition 14. A texture map is a raster image that is to be mapped onto a polygon in the display raster.

Definition 15. A textel is a pixel in a texture map.

Texture mapping allows the dimensions of the colour data array to be different from the data defining the surface plot. You can apply an image of arbitrary size to any surface. The texture colour data is interpolated so that it is mapped to the entire surface. Texture mapping is an incredibly powerful addition to a computer graphics library, as we can render simple shapes, flat polygons, and make them appear as realistic objects in a scene.



(a) texture (b) display

Figure 2.14: Texture mapping applies an image to a polygon in the raster.

2.5.1 Texture space

The **texture space** is the space in which the texture exists. It is a two-dimensional space with horizontal and vertical axes are denoted by u and v respectively where $u, v \in [0, 1]$ (figure 2.15). The texture map will fill the texture space so that the bottom left-hand corner of the texture is at $(0, 0)$ and the top right-hand corner of the texture is at $(1, 1)$. The co-ordinates of a textel (U, V) corresponding to the texture space co-ordinates (u, v) for a $t_y \times t_x$ texture map is calculated in a similar way to pixel co-ordinates in equations (6a) and (6b)

$$U = \lfloor u\, t_x \rfloor, \tag{19a}$$
$$V = \lfloor (1 - v) t_y \rfloor, \tag{19b}$$

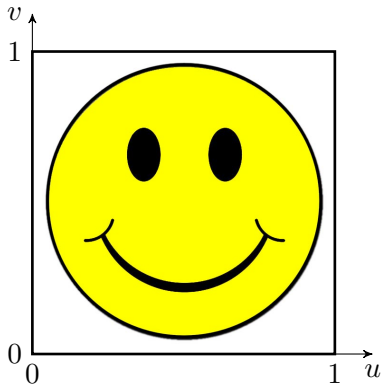
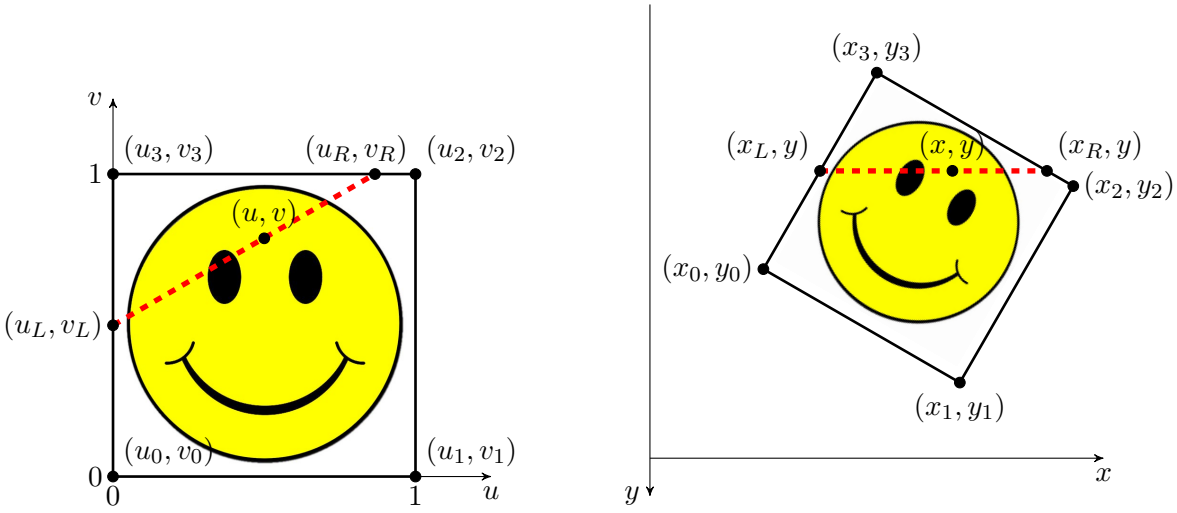


Figure 2.15: The texture space.

2.5.2 Applying a texture map

Consider figure 2.16 where the texture on the left is applied a polygon in the display raster on the right defined by the vertices (x_0, y_0), (x_1, y_1), (x_2, y_2) and (x_3, y_3) which correspond to the vertices of the texture space (u_0, v_0), (u_1, v_1), (u_2, v_2) and (u_3, v_3) respectively. Any transformation that is applied to the polygon should be accounted for by the texture mapping.



(a) texture space (b) display raster

Figure 2.16: Scanlines in the texture and display rasters.

Using a scanline algorithm, we loop through each horizontal row of pixels in the polygon and calculate the co-ordinates of the scan extrema pixels (x_L, y) and (x_R, y). This is done by interpolating between the vertices of the appropriate edge, for example in figure 2.16, x_L is interpolated between x_3 and x_0 and x_R is interpolated between x_3 and x_2. The points in the texture space corresponding to the scan extrema, (u_L, v_L) and (u_R, v_R), are calculated by interpolating between the vertices of the texture corresponding to the vertices in the display raster. The colour of the pixel (x, y) along the scanline is determined by interpolating between (u_L, v_L) and (u_R, v_R) in the texture.

The co-ordinates of the scan extrema on the polygon are calculated using the scanline algorithm where we loop through each horizontal row of pixels in the polygon and determine the co-ordinates of the pixels on the far left and right of the row.

2.5.3 Calculating the scan extrema in the texture space

As the co-ordinates of the scan extrema in the display raster changes, so should the co-ordinates of the scan extrema in the texture space. Since the texture fills the uv space where $u, v \in [0, 1]$ then the scan extrema for the texture will always trace along the boundary of the texture space. Initially, the scan extrema in the texture space will be the vertex corresponding to the vertex in the display raster with the smallest y co-ordinate. The texture space co-ordinates of the scan extrema are then calculated by interpolating along the edges of the texture corresponding the edges of the polygon.

Consider the texture and polygon in figure 2.16. The polygon vertex at (x_3, y_3) corresponds to the texture vertex (u_3, v_3) so the initial values of the scan extrema in the texture space are $u_L = u_R = 0$ and $v_L = v_R = 1$. Since the left-hand scan extrema will have a y co-ordinate between y_3 and y_0 then the texture space co-ordinates (u_L, v_L) range between (u_3, v_3) and (u_0, v_0) in the same span, therefore the

change in the u_L and v_L for moving down to the next scanline are

$$\Delta u_L = \frac{u_3 - u_0}{y_3 - y_0} = 0,$$
$$\Delta v_L = \frac{v_3 - v_0}{y_3 - y_0} = \frac{1}{y_3 - y_0}.$$

Doing similar for the right-hand scan extrema

$$\Delta u_R = \frac{u_3 - u_2}{y_3 - y_2} = \frac{1}{y_3 - y_2},$$
$$\Delta v_R = \frac{v_3 - v_2}{y_3 - y_2} = 0.$$

Note that one of Δu_L or Δv_L will be 0 since the edge of the texture is either horizontal or vertical (and similar for Δu_R and Δv_R). When the polygon edges that intersect the scanline changes, the edge of the texture that the extrema is on will also change. For example, for the polygon in figure 2.16 when $y_0 < y < y_2$ the point (u_R, v_R) is on the right-hand texture edge and the values of Δu_R and Δv_R change to

$$\Delta u_R = 0,$$
$$\Delta v_R = \frac{1}{y_2 - y_1}.$$

The general interpolating equations for the scan extrema points are

$$a_L = a_L + \Delta a_L, \tag{20a}$$
$$a_R = a_R + \Delta a_R, \tag{20b}$$
$$\Delta a_L = \frac{a_p - a_q}{y_p - y_q}, \tag{20c}$$
$$\Delta a_R = \frac{a_r - a_s}{y_r - y_s}, \tag{20d}$$

where a denotes either x, u or v and p, q and r, s are the indices of the upper and lower vertices for the left and right-hand edges respectively.

2.5.4 Interpolating along the scanline

Once the scan extrema in the screen and texture space have been calculated, we initialise u and v to u_L and v_L respectively before looping across the pixels in the scanline between x_L and x_R and the corresponding pixels in the texture space between (u_L, v_L) and (u_R, v_R). The textel co-ordinates are calculated using equations (19a) and (19b) the the colour of the textel is assigned to the pixel in the display raster. Since the x co-ordinate will go between x_L and x_R, the u and v co-ordinates in the texture space must go between (u_L, v_L) and (u_R, v_R) in the same span, the interpolating equations are

$$a = a + \Delta a, \tag{21a}$$
$$\Delta a = \frac{a_R - a_L}{x_R - x_L}, \tag{21b}$$

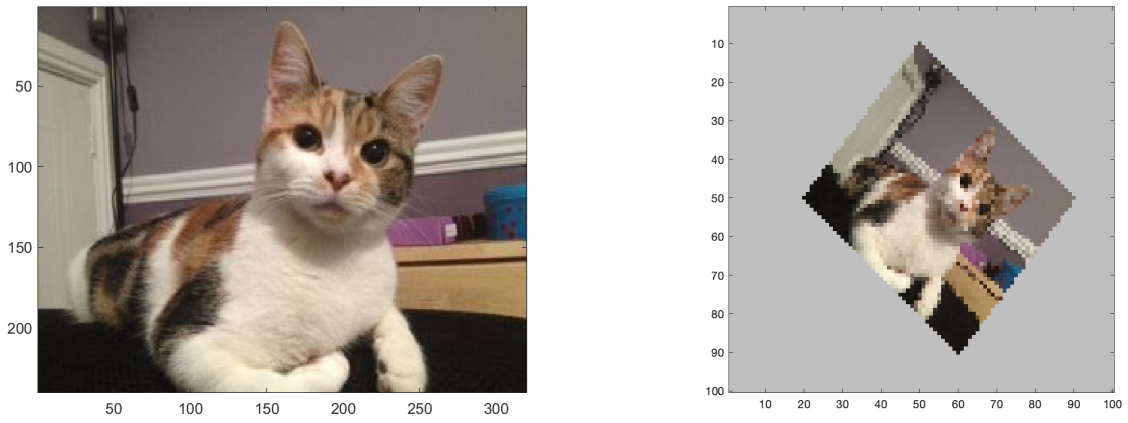
where a denotes either u or v.

2.5.5 Algorithm

The algorithm used to apply a texture map is given in algorithm 6. The function uses inputs of the raster array R, texture map T and the pixel co-ordinates of the polygon $(x_0, \ldots x_n)$ and $(y_0, \ldots, y_n)$. The results of a MATLAB function for this algorithm is shown in figure 2.17.

Algorithm 6 Texture mapping algorithm

function TEXTUREMAPPER(R, T, $(x_0, \ldots, x_n)$, $(y_0, \ldots, y_n)$)
  Initialise $(u_0, v_0) \leftarrow (0, 0)$, $(u_1, v_1) \leftarrow (1, 0)$, $(u_2, v_2) \leftarrow (1, 1)$ and $(u_3, v_3) \leftarrow (0, 1)$
  Initialise p and r to the index of the smallest y polygon vertex
  Initialise q and s using equations (18a) and (18b)
  Initialise $x_L \leftarrow x_p$, $u_L \leftarrow u_p$, $v_L \leftarrow v_p$, $x_R \leftarrow x_r$, $u_R \leftarrow u_r$ and $v_R \leftarrow v_r$
  Calculate Δx_L, Δu_L, Δv_L , Δx_R, Δu_R and Δv_R using equations (20c) and (20d)
  for $y := y_{\min} \ldots y_{\max}$ **do** ▷ Loop through scanlines
    Initialise $u \leftarrow u_L$, $v \leftarrow v_L$
    Calculate Δu and Δv using equation (21b)
    for $x := x_L \ldots x_R$ **do** ▷ Loop across pixels in the current scanline
      Calculate U and V using equations (19a) and (19b)
      $R(y, x) \leftarrow T(V, U)$
      Calculate u and v using equation (21a)
    end for
    if $y = y_p$ **then** ▷ left edge changes
      $p \leftarrow q$
      Update q using equation (18a)
      Recalculate Δx_L, Δu_L and Δv_L for new left edge using equation (20c)
    end if
    if $y = y_r$ **then** ▷ right edge changes
      $r \leftarrow s$
      Update s using equation (18b)
      Recalculate Δx_R, Δu_R and Δv_R for new right edge using equation (20d)
    end if
    Recalculate x_L, u_L, v_L, x_R, u_R and v_R using equations (20a) and (20b)
  end for
  return R
end function



(a) texture (b) raster

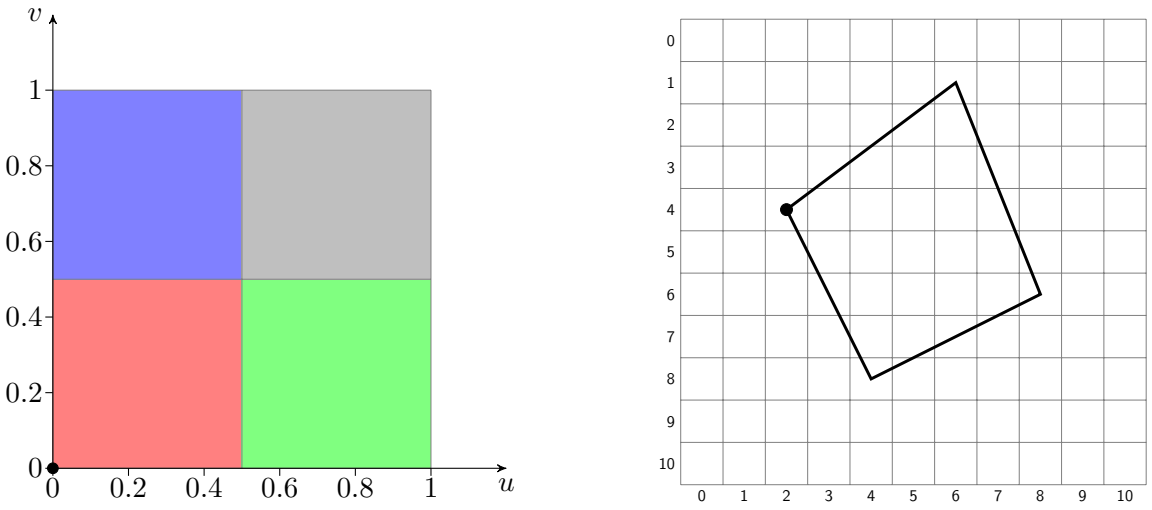
Figure 2.17: Texture mapping onto a polygon in the display raster.

Example 7 A 2×2 textel texture is to be mapped to the display raster shown in figure 2.18. The vertex with pixel co-ordinates $(2, 4)$ corresponds to the point $(0, 0)$ on the texture map.

(i) For each scanline in the polygon, calculate the scan extrema co-ordinates for the display raster and the texture map.

(ii) For each pixel across the scanline, calculate the texture space co-ordinates.

(iii) Copy the display raster and shade in the pixels with the corresponding colour from the texture map.



(a) Texture (b) Display raster

Figure 2.18: Texture mapping example

(i) The vertex with the smallest y co-ordinate is at $(6, 1)$ with corresponds to the texture map co-ordinates $(0, 1)$. We initialise $x_L = x_R = 6$, $y = 1$, $u_L = u_R = 0$ and $v_L = v_R = 1$ and calculate

$$\Delta x_L = \frac{6-2}{1-4} = -1.33, \qquad \Delta u_L = \frac{0-0}{1-4} = 0, \qquad \Delta v_L = \frac{1-0}{1-4} = -0.33,$$
$$\Delta x_R = \frac{6-8}{1-6} = 0.4, \qquad \Delta u_R = \frac{0-1}{1-6} = 0.2, \qquad \Delta v_R = \frac{0-0}{1-6} = 0.$$

The scan extrema values for the next few scanlines are

y	x_L	u_L	v_L	x_R	u_R	v_R
1	6	0	1	6	0	1
2	4.67	0	0.67	6.4	0.2	1
3	3.33	0	0.33	6.8	0.4	1
4	2	0	0	7.2	0.6	1

We have reached the end of the left-hand edge so we need to recalculate Δx_L, Δu_L and Δv_L. The upper and lower vertices for the new left edge have co-ordinates $(2, 4)$ and $(4, 8)$ in the display raster and $(0, 0)$ and $(1, 0)$ in the texture map, so we have

$$\Delta x_L = \frac{2-4}{4-8} = 0.5, \qquad \Delta u_L = \frac{0-1}{4-8} = 0.25, \qquad \Delta v_L = \frac{0-0}{4-8} = 0.$$

The scan extrema values for the next few scanlines are

y	x_L	u_L	v_L	x_R	u_R	v_R
5	2.5	0.25	0	7.6	0.8	0
6	3	0.5	0	8	1	1

We have reached the end of the right-hand edge so we need to recalculate Δx_R, Δu_R and Δv_R. The upper and lower vertices for the new right edge have co-ordinates $(8, 6)$ and $(4, 8)$ in the display raster and $(1, 1)$ and $(1, 0)$ in the texture map, so we have

$$\Delta x_R = \frac{8-4}{6-8} = -2, \qquad \Delta u_R = \frac{1-1}{6-8} = 0, \qquad \Delta v_R = \frac{1-0}{6-8} = -0.5.$$

The scan extrema values for the next few scanlines are

y	x_L	u_L	v_L	x_R	u_R	v_R
7	3.5	0.75	0	6	1	0.5
8	4	1	0	4	1	0

(ii) For $y = 1$: $x_L = x_R = 6$, $u_L = u_R = 0$, $v_L = v_R = 1$ so we have a single texture space point at $(0, 1)$.

For $y = 2$: $x_L = 5$, $x_R = 6$, $u_L = 0$, $v_L = 1$, $u_R = 0$ and $v_R = 1$ so

$$\Delta u = \frac{0.2-0}{6-5} = 0.2, \qquad \Delta v = \frac{1-0.67}{6-5} = 0.33,$$

and the texture space co-ordinates are $(0, 0.67)$, $(0.2, 1)$

For $y = 3$: $x_L = 3$, $x_R = 7$, $u_L = 0$, $v_L = 0.33$, $u_R = 0.4$ and $v_R = 1$ so

$$\Delta u = \frac{0.4-0}{7-3} = 0.1, \qquad \Delta v = \frac{1-0.33}{7-3} = 0.17,$$

and the texture space co-ordinates are $(0, 0.33)$, $(0.1, 0.5)$, $(0.2, 0.67)$, $(0.3, 0.84)$ and $(0.4, 1)$.

For $y = 4$: $x_L = 2$, $x_R = 7$, $u_L = 0$, $v_L = 0$, $u_R = 0.6$ and $v_R = 1$ so

$$\Delta u = \frac{0.6-0}{7-2} = 0.12, \qquad \Delta v = \frac{1-0}{7-2} = 0.2,$$

and the texture space co-ordinates are $(0, 0)$, $(0.12, 0.2)$, $(0.24, 0.4)$, $(0.36, 0.6)$, $(0.48, 0.8)$ and $(0.6, 1)$.

For $y = 5$: $x_L = 3$, $x_R = 8$, $u_L = 0.25$, $v_L = 0$, $u_R = 0.8$ and $v_R = 0$ so

$$\Delta u = \frac{0.8-0.25}{8-3} = 0.11, \qquad \Delta v = \frac{1-0}{8-3} = 0.2,$$

and the texture space co-ordinates are $(0.25, 0)$, $(0.36, 0.2)$, $(0.47, 0.4)$, $(0.58, 0.6)$, $(0.69, 0.8)$ and $(0.8, 1)$.

For $y = 6$: $x_L = 3$, $x_R = 8$, $u_L = 0.5$, $v_L = 0$, $u_R = 1$ and $v_R = 1$ so

$$\Delta u = \frac{1-0.5}{8-3} = 0.1, \qquad \Delta v = \frac{1-0}{8-3} = 0.2,$$

and the texture space co-ordinates are $(0.5, 0)$, $(0.6, 0.2)$, $(0.7, 0.4)$, $(0.8, 0.6)$, $(0.9, 0.8)$ and $(1, 1)$.

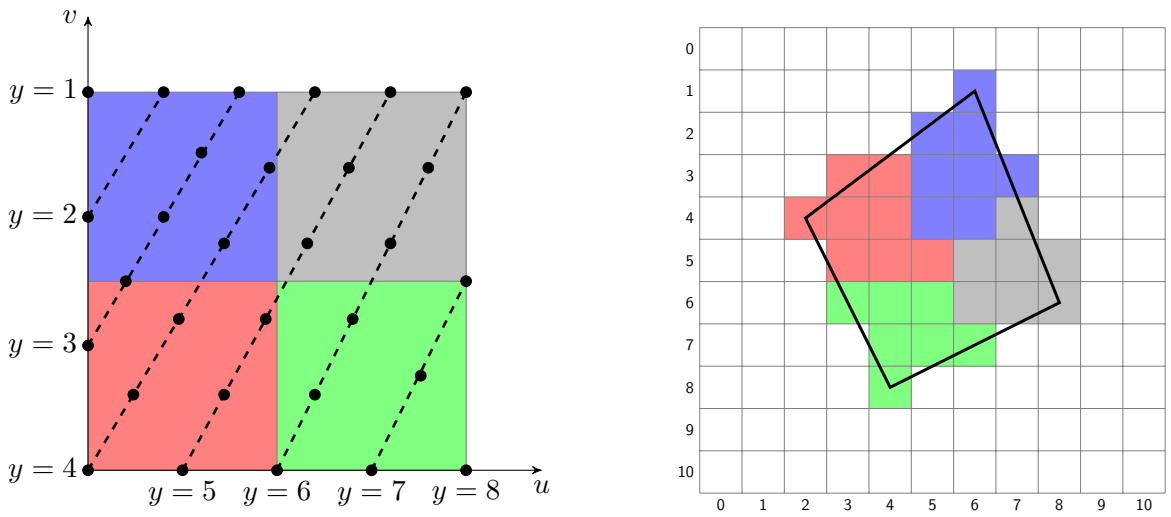
For $y = 7$: $x_L = 4$, $x_R = 6$, $u_L = 0.75$, $v_L = 0$, $u_R = 1$ and $v_R = 0.5$ so

$$\Delta u = \frac{1-0.75}{6-4} = 0.13, \qquad \Delta v = \frac{0.5-0}{6-4} = 0.25,$$

and the texture space co-ordinates are $(0.75, 0)$, $(0.88, 0.25)$ and $(1, 0.5)$.

For $y = 7$: $x_L = x_R = 4$, $u_L = u_R = 1$ and $v_L = v_R = 0$ so we have a single texture space point at $(1, 0)$.

(iii)



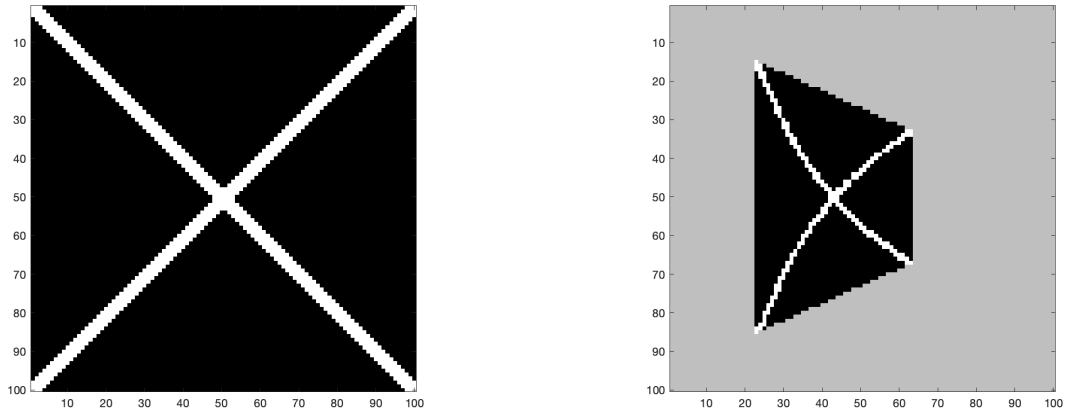
(a) Texture

(b) Display raster

Figure 2.19: Texture mapping example: interpolating in the texture space.

2.6 Perspective corrected texture mapping

Consider figure 2.20 where a texture map of an X texture has been mapped to a polygon that have been rotated about the y-axis and projected onto the screen space using perspective projection so that the right-hand edge that is further away appears smaller. The white diagonal lines should appear to be straight lines on the texture mapped polygon, but in this case they appeared curved. The reason for this is that the depth of the pixels in the polygon have not been taken into account.



(a) texture (b) linear texture map

Figure 2.20: Texture mapping of a simple '$\times$' texture demonstrating the 'bulging' that is experienced when using linear texture mapping.

To correct our texture mapper we need to consider the relationship between the view space co-ordinates and the display raster co-ordinates when perspective projection has been applied. The diagram in figure 2.21 shows a polygon defined by the view space co-ordinates (x_0, y_0, z_0) and (x_1, y_1, z_1) when viewed from above (down the y-axis). Applying perspective projection will give the screen space co-ordinates (x'_0, y'_0, f) and (x'_1, y'_1, f) that defines the polygon in the screen space. Note that the depth co-ordinate or all values along the projected polygon is $z = f$ so using the screen space co-ordinates alone does not provide us with any information about the depth of the polygon.

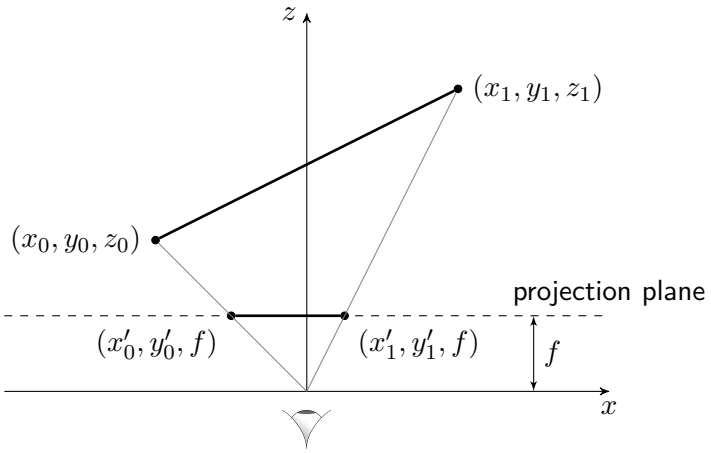


Figure 2.21: Perspective projection of a polygon onto the viewing plane.

To take the depth of the polygon into account we need to find a relationship between the x and y screen space co-ordinates and the z co-ordinate in the view space. Using the polygon in figure 2.21, the x co-ordinate of points on the line joining the vertices (x_0, y_0, z_0) and (x_1, y_1, z_1) in the view space can be

calculated using

$$x = mz + c. \tag{22}$$

for some scalar values m and c. The perspective projection of the x view space co-ordinate is given in equation (38)) which is

$$x' = \frac{f}{z}x,$$

which can be transposed to give

$$x = \frac{z}{f}x'.$$

Substituting this expression into equation (22) results in

$$\frac{z}{f}x = mz + c$$

Making z the subject provides the relationship between the x co-ordinate in the screen space and the z co-ordinate in the world space.

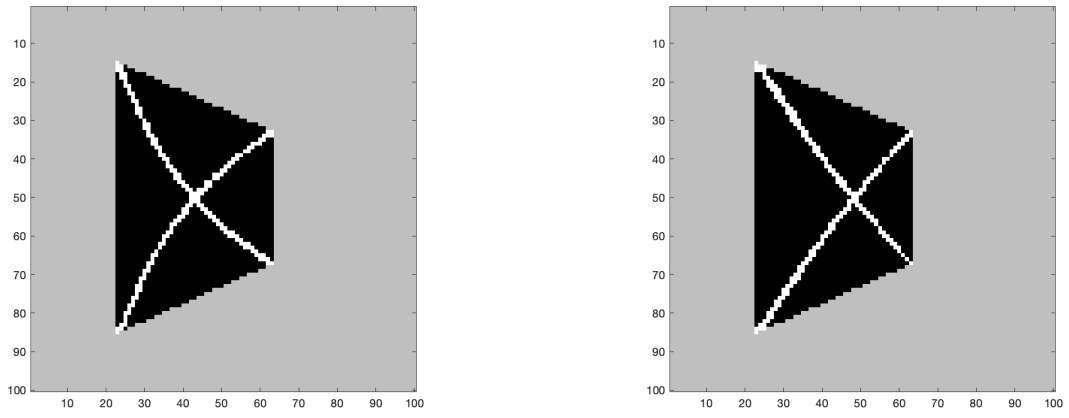
$$z = \frac{cf}{x - fm} \tag{23}$$

Since x is in the denominator the z co-ordinate in the view space does not change in a linear fashion as the x' co-ordinate in the projected screen space changes. This causes the bulging of the texture mapped polygon seen in figure 2.20. However, what if we consider the reciprocal of equation (23)

$$\frac{1}{z} = \frac{x - fm}{cf},$$

then we have a linear function of the variable x. Therefore we can divide the vertices of the texture space by the corresponding z co-ordinates of the vertices in the screen space and interpolate the texture space in the same way as before. When we need to determine the textel colour we divide the texture co-ordinates by z and find the textel at $\left(\frac{u}{z}, \frac{v}{z}\right)$. The modified texture mapping algorithm with perspective correction is shown in algorithm 7.

The results of using the perspective correction can be seen in figures 2.22 and 2.23. Figure 2.22 shows a simple '$\times$' texture mapped a polygon that is rotated $\pi/6$ radians about the y axis. The bulge that was clearly visible when using the linear texture mapper is not present when using the perspective correction and the lines appear to be straighter.



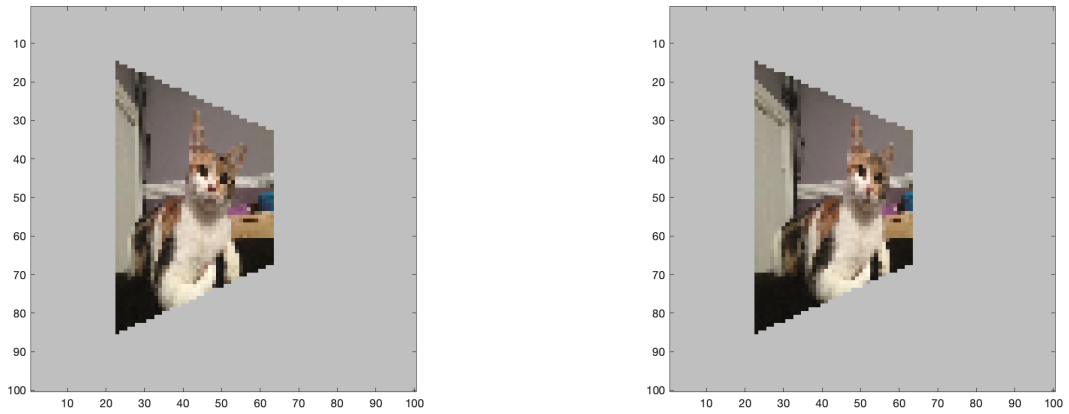
(a) linear texture map (b) perspective correction

Figure 2.22: Comparison of linear and perspective texture mapping.

Algorithm 7 Perspective corrected texture mapping algorithm

function TEXTUREMAPPER(R, T, $(x_0, \ldots, x_n)$, $(y_0, \ldots, y_n)$, $(z_0, \ldots, z_n)$)
  Initialise $(u_0, v_0) \leftarrow (0, 0)$, $(u_1, v_1) \leftarrow \left(\frac{1}{z_1}, 0\right)$, $(u_2, v_2) \leftarrow \left(\frac{1}{z_2}, \frac{1}{z_2}\right)$ and $(u_3, v_3) \leftarrow \left(0, \frac{1}{z_3}\right)$
  Initialise p and r to the index of the smallest y polygon vertex
  Initialise q and s using equations (18a) and (18b)
  Initialise $x_L \leftarrow x_p$, $u_L \leftarrow u_p$, $v_L \leftarrow v_p$, $z_L \leftarrow z_p$, $x_R \leftarrow x_r$, $u_R \leftarrow u_r$, $v_R \leftarrow v_r$ and $z_R \leftarrow z_r$
  Calculate Δx_L, Δu_L, Δv_L, Δz_L, Δx_R, Δu_R, Δv_R and Δz_R using equations (20c) and (20d)
  for $y := y_{\min} \ldots y_{\max}$ **do** ▷ Loop through scanlines
    Initialise $u \leftarrow u_L$, $v \leftarrow v_L$, $z \leftarrow z_L$
    Calculate Δu, Δv and Δz using equation (21b)
    for $x := x_L \ldots x_R$ **do** ▷ Loop across pixels in the scanline
      Calculate U and V using equations (19a) and (19b) with $\frac{u}{z}$ and $\frac{v}{z}$
      $R(y, x) \leftarrow T(V, U)$
      Calculate u, v and z using equation (21a)
    end for
    if $y = y_p$ **then** ▷ left edge changes
      Update $p \leftarrow q$
      Update q using equation (18a)
      Recalculate Δx_L, Δu_L, Δv_L and Δz_L for new left edge using equation (20c)
    end if
    if $y = y_r$ **then** ▷ right edge changes
      Update $r \leftarrow s$
      Update s using equation (18b)
      Recalculate Δx_R, Δu_R, Δv_R and Δz_R for new right edge using equation (20d)
    end if
    Recalculate x_L, u_L, v_L, z_L, x_R, u_R, v_R and z_R using equations (20a) and (20b)
  end for
  return R
end function

The improvement from using perspective texture mapping over linear mapping is less pronounced in figure 2.23 where the photograph has been mapped to the same polygon. However, the bulging that is seen using the linear texture mapper is more evident when viewing moving polygons.



(a) linear texture map (b) perspective correction

Figure 2.23: Comparison of linear and perspective texture mapping.

2.7 Normal mapping

Another application of texture mapping is the use of **normal mapping** (also known as **bump mapping**). Normal mapping is an attempt to make a flat polygon appear to have three-dimensional artefacts such as bumps or indentations by applying a texture that is based on normal vectors instead of colours. Lighting models use the normal vectors to determine the amount of light and shadow that should be seen by the viewer (we will cover lighting models elsewhere in the unit), so an application of a normal map can enhance the appearance of a flat polygon.

Consider figure 2.24 where it is necessary to model an uneven surface. We could do this using a number of different polygons each with their own normal vector. However, using normal mapping we can map the various normal vectors on to a single polygon which when combined with a lighting model will give the appearance of the uneven surface.

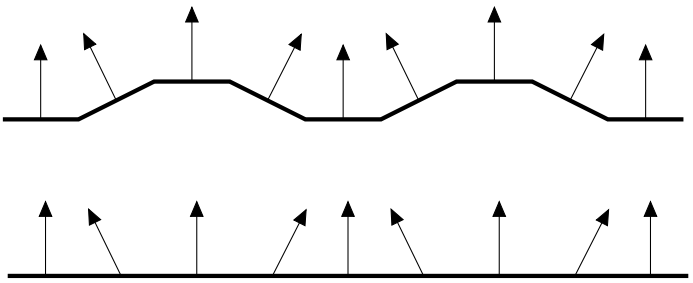


Figure 2.24: The bumpy surface modelled using 9 polygons (top) can be approximated using a normal map with just 1 polygon (bottom).

An example of a three-dimensional normal map can be seen in figure 2.26. The normal map shown here is represented by an RGB colour map where the colour that represents each normal vector depends on the direction it is pointing. The x direction is represented by red, the y direction green and the z direction blue figure 2.25. This is why normal maps often appear purple because most normal vectors have a large z component (these notes are best viewed in colour).

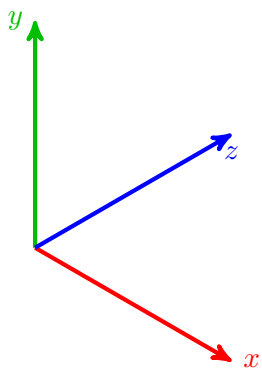


Figure 2.25: Normal maps are represented by a RGB colour map depending on the direction of the normal vector is pointing.

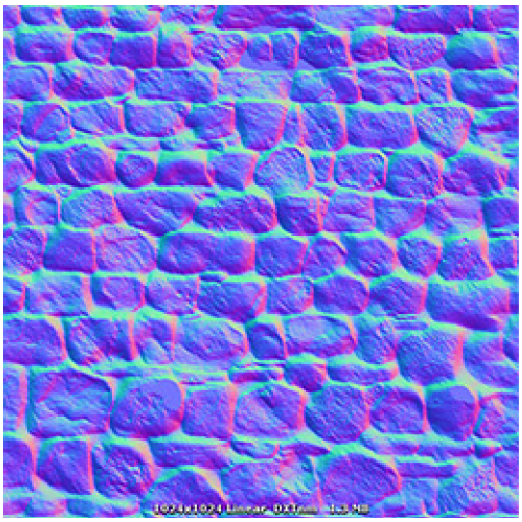


Figure 2.26: A normal map for a brick wall (Unity Technologies 2017).

The affects of applying a normal map can be seen in figure 2.27 where the brick wall texture map is applied to a polygon with and without the normal map shown in figure 2.26. The rough surface of the bricks can be seen by the shadows that appear in the bottom image.

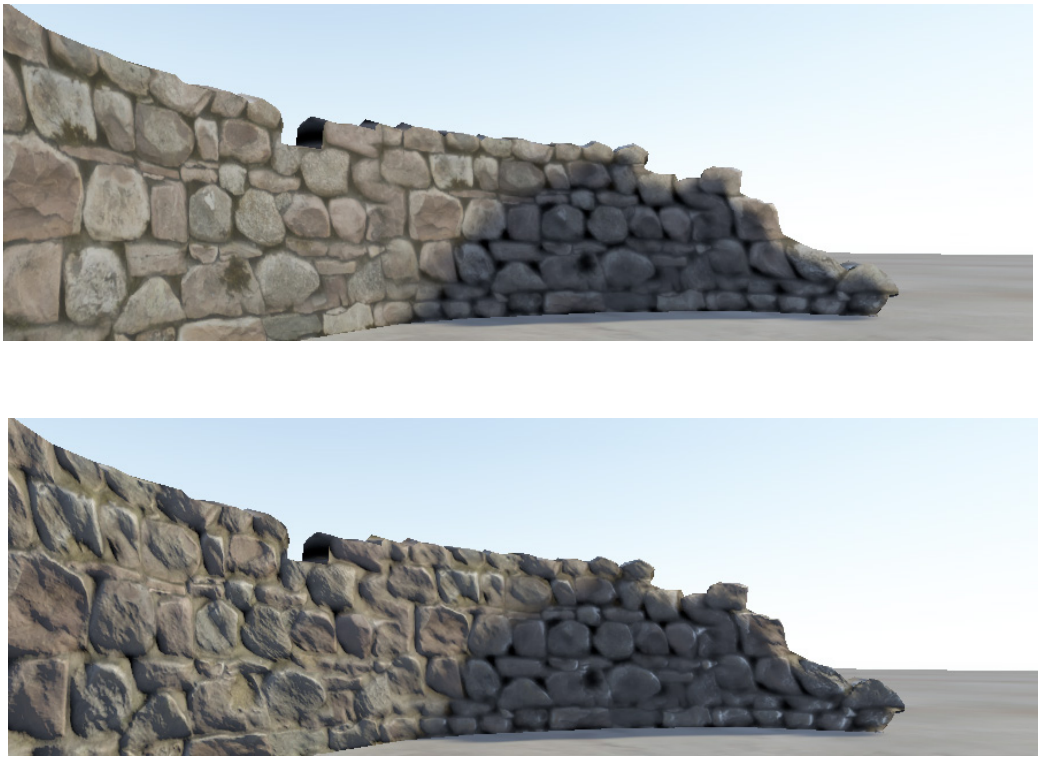


Figure 2.27: A stone wall rendered with just a texture map (top) and with a texture map and a normal map (bottom) (Unity Technologies 2017)

2.8 Exercises

1. Use Bresenham's algorithm (algorithm 2 on page 17) to determine the co-ordinates of the pixels on the lines joining the following points:
   (a) $(2, 1)$ and $(8, 5)$;
   (b) $(3, 0)$ and $(7, 6)$;
   (c) $(10, 4)$ and $(3, 7)$;
   (d) $(9, 8)$ and $(4, 4)$.
2. Use the midpoint algorithm (algorithm 3 on page 22) to determine the co-ordinates of pixels in the first-octant (where $y \leq x$) of circles centred at $(0, 0)$ with radius:
   (a) $r = 7$;
   (b) $r = 9$;
   (c) $r = 15$.
3. The flood fill algorithm (algorithm 4 on page 25) is used to fill in the region in figure 2.28 starting at pixel $(5, 5)$. Determine the ordering of the pixels that are filled by the algorithm.

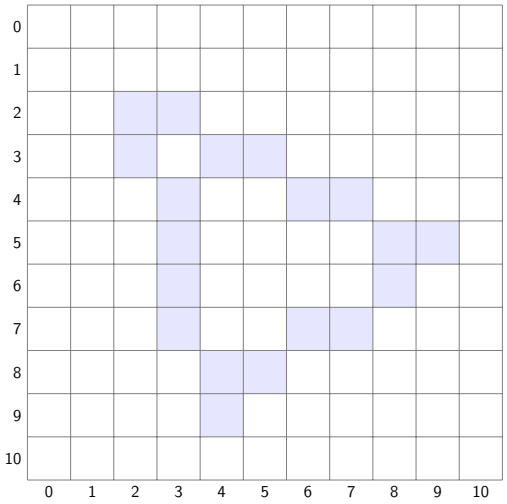


Figure 2.28: Flood fill exercise.

4. The scanline algorithm (algorithm 5 on page 30) is used to draw a polygon with the vertices at pixels $(2, 8)$, $(9, 4)$ and $(4, 1)$. Calculate the co-ordinates of the scan extrema pixels for each scanline in the polygon.

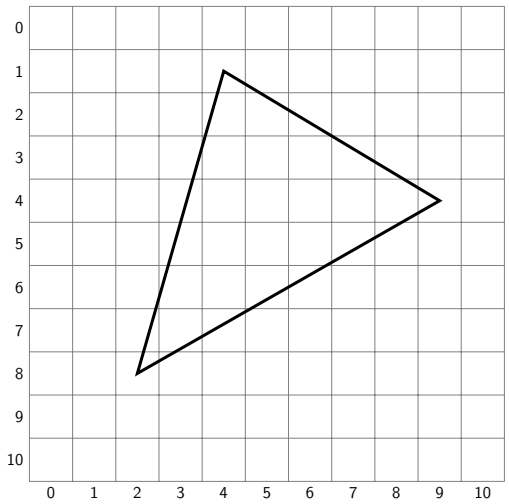


Figure 2.29: Scanline exercise.

The solutions to these exercises can be found on page 135.

Chapter 3

Image Processing

3.1 Antialiasing

We have seen in chapter 2 that when an idealised image is represented using a raster array, lines and edges of curved and sloping surfaces appear jagged because pixels are square. We can attempt to reduce this effect by using **antialising** which is the process of smoothing lines and edges so that they appear less jagged. Antialiasing is achieved by illuminating the pixels adjacent to the idealised image using a colour intensity less than that of the colour intensity used for the rasterised image. There are several approaches used in practice and the most common of these is super sampling antialiasing.

3.1.1 Super sampling antialiasing

Super sampling antialiasing (SSAA) uses a raster which has twice or four times the number of pixels in the horizontal and vertical directions as the display which is known as a **super sample**. The scene that is to be rendered is calculated using the super sample and the colours of each pixel on the display raster is then calculated by averaging the pixels of the super-sample raster that correspond to the pixel on the display raster. For example, consider a super-sample raster that is $2\times$ that of the display raster.

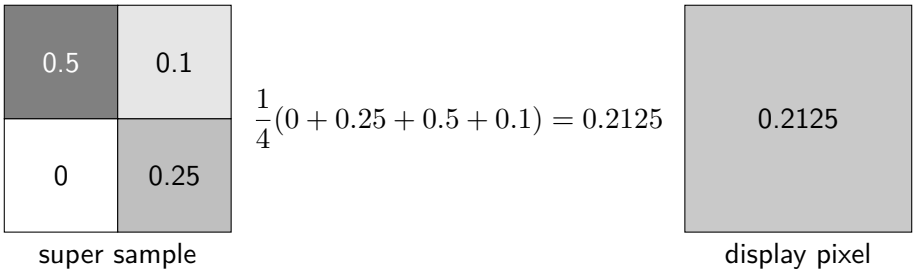
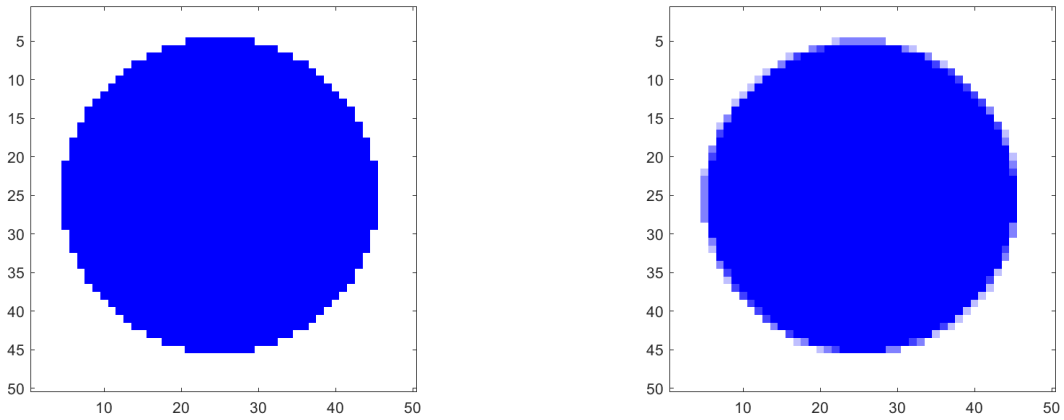


Figure 3.1: The pixels of the $2\times$ super sample are averaged to give the intensity of the display pixel.

The effects of super sampling antialiasing can be seen in figure 3.2 where a filled circle drawn on a 100×100 display raster has had $2\times$ super sampling antialiasing applied. The antialiased circle appears less blocky than the no antialiased circle, however, since we are using 4 times the number of pixels in the super sample our rasterisation routines such as filling, texture mapping and lighting, require 4 times the operations as a non-antialiased scene.



(a) No antialiasing (b) Super sample antialiasing

Figure 3.2: The effects of super sample antialiasing on a filled circle.

3.2 Convolution

Convolution is a common method used in image processing to apply blurring, sharpening, embossing and edge detection to an image. Given an input image we can apply convolution using a **kernel** (or filter) on the input to produce an output image that has the kernel applied.

The convolution of an image array X and an $n \times n$ kernel array K where n is odd is denoted using $X * K$ and defined for the element in row i and column j is

$$[X * Y]_{ij} = \sum_{p=-r}^{r} \sum_{q=-r}^{r} [X]_{i+p,j+q}[K]_{1+r+p,1+r+q}, \tag{24}$$

where $r = \dfrac{n-1}{2}$. In other words the kernel array is aligned with elements from the image array centred at pixel ij. The values of the corresponding elements are multiplied and the sum of the result is the value of $[X * K]_{ij}$.

Example 8 Calculate the convolution $X * K$ for the following image and kernel arrays

$$X = \begin{pmatrix} 4 & 6 & 4 & 3 & 6 \\ 6 & 5 & 5 & 2 & 4 \\ 5 & 2 & 3 & 4 & 1 \\ 4 & 6 & 5 & 3 & 6 \\ 6 & 3 & 2 & 4 & 6 \end{pmatrix}, \qquad K = \begin{pmatrix} 0 & \frac{1}{4} & 0 \\ \frac{1}{4} & 1 & \frac{1}{4} \\ 0 & \frac{1}{4} & 0 \end{pmatrix}.$$

Computing the value of $[X * K]_{22}$ gives

$$\begin{aligned} [X * K]_{2,2} &= \sum_{p=-1}^{1} \sum_{q=-1}^{1} [X]_{2+p,2+q}[K]_{2+p,2+q} \\ &= [X]_{11}[K]_{11} + [X]_{12}[K]_{12} + [X]_{13}[K]_{13} + [X]_{21}[K]_{21} + [X]_{22}[K]_{22} \\ &\quad + [X]_{23}[K]_{23} + [X]_{31}[K]_{31} + [X]_{32}[K]_{32} + [X]_{33}[K]_{33} \\ &= 6 \times 0 + 4 \times \frac{1}{4} + 4 \times 0 + 6 \times \frac{1}{4} + 5 \times 1 + 3 \times \frac{1}{4} + 5 \times 0 + 2 \times \frac{1}{4} + 3 \times 0 \\ &= 9.75 \end{aligned}$$

4	6	4	3	6
6	5	5	2	4
5	2	3	4	1
4	6	5	3	6
6	3	2	4	6

X

$\longrightarrow$

$4(0)$	$6(\frac{1}{4})$	$4(0)$	3	6
$6(\frac{1}{4})$	$5(1)$	$3(\frac{1}{4})$	2	4
$5(0)$	$2(\frac{1}{4})$	$3(0)$	4	1
4	6	5	3	6
6	3	2	4	6

X with K centred on $[X]_{22}$

$=$

	9.75			

$X * K$

Repeating the calculations for the other interior elements of Y gives

$$X * K = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 9.75 & 8.5 & 6 & 0 \\ 0 & 6.75 & 7 & 6.25 & 0 \\ 0 & 9.5 & 8.5 & 7.75 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

The pseudo-code for convolution is given in algorithm 8.

Algorithm 8 Convolution

```
function CONVOLUTION(X, K)
    Y ← n_y × n_x × 3 array of zeros        ▷ n_y and n_x are the number of rows and columns in X
    r ← (n − 1)/2                           ▷ n is the number of rows and columns in K
    for i := 1 + r, ..., n_y − r do         ▷ loop through the image pixels
        for j := 1 + r, ..., n_x − r do
            for k := 1, ..., 3 do           ▷ loop through each primary RGB colour
                for p := −r, ..., r do      ▷ loop through the kernel elements
                    for q := −r, ..., r do
                        [Y]_{i,j,k} = [Y]_{i,j,k} + [X]_{i+p,j+q,k}[K]_{1+r+p,1+r+q}
                    end for
                end for
            end for
        end for
    end for
    return Y
end function
```

3.2.1 Handling the edge pixels

We can see from example 8 that it isn't possible to calculate the convolution for the edge pixels of the image. We can either ignore these pixels and crop the convoluted image or we can apply on of the following methods

- Extend – the image array is extended as far as necessary to provide values for the convolution of the edge pixels. The values of the new extended pixels assume that of the edge pixels.
- Wrap – the image is conceptually wrapped so that the pixels adjacent to the edge are taken from the opposite side.
- Mirror – the edge of the image is conceptionally mirrored so that attempting to access a pixel outside of the image will be replaced by a pixel inside the image at the same distance from the edge pixel.

3.2.2 Box blur

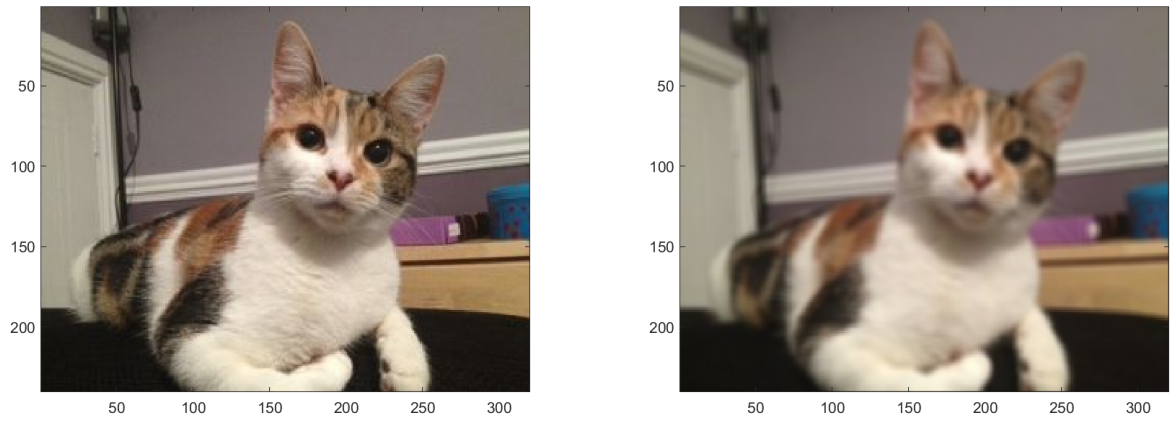
Different kernel filters can be used manipulate images in different ways. One common application is the blurring of an image or part of an image. The simplest method used to blur an image is called the **box blur** where all elements of a kernel have the same value, e.g.,

$$K = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

If we were to apply this kernel using convolution it would increase the brightness of each pixel by a factor of 25. Therefore, when designing kernels we need to ensure the elements sum to 1, i.e.,

$$K_{blur} = \frac{1}{25} \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

The results from applying a box blur kernel filter can be seen in figure 3.3.



(a) original image (b) box blur

Figure 3.3: The affects of applying a 5×5 box blur filter to an image.

3.2.3 Gaussian blur

The problem with the box blur is that it applies the blurring over all pixels equally and we do lose some of the information in the image. Another blurring technique is the **Gaussian blur** where the elements of the kernel approximate the Gaussian function (figure 3.4) so that the information from the pixels closer to the central pixel has more of an effect than pixels further away. This results in a smoothing of the source image and it useful for cleaning up noisy images.

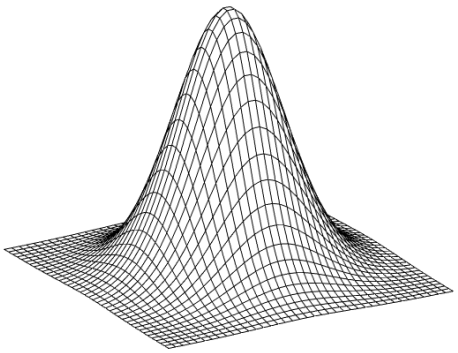


Figure 3.4: Two-dimensional Gaussian curve.

To generate the kernel function consider the Gaussian function for mean $\mu = 0$ and standard deviation σ

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{\left(-\frac{x^2}{2\sigma^2}\right)}.$$

For a 1×5 element kernel we can use

$$x = (-2, 1, 0, 1, 2),$$

therefore is we let $\sigma = 1$

$$G(x) = (0.054, 0.0242, 0.399, 0.242, 0.054).$$

To generate a 5×5 kernel array we calculate

$$K = G(x)^T G(x) = \begin{pmatrix} 1.0000 & 4.4817 & 7.3891 & 4.4817 & 1.0000 \\ 4.4817 & 20.0855 & 33.1155 & 20.0855 & 4.4817 \\ 7.3891 & 33.1155 & 54.5982 & 33.1155 & 7.3891 \\ 4.4817 & 20.0855 & 33.1155 & 20.0855 & 4.4817 \\ 1.0000 & 4.4817 & 7.3891 & 4.4817 & 1.0000 \end{pmatrix},$$

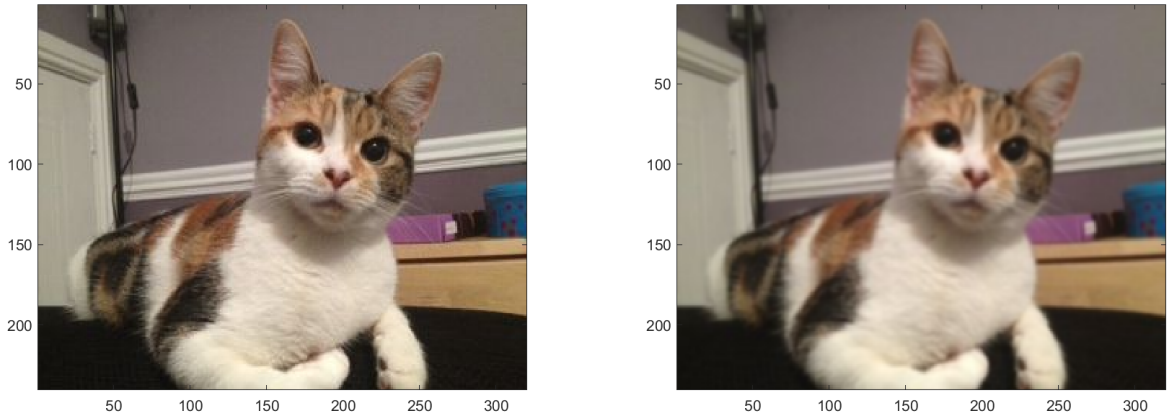
and rounding to the nearest integer to save computational effort

$$K = \begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 20 & 33 & 20 & 4 \\ 7 & 33 & 55 & 33 & 7 \\ 4 & 20 & 33 & 20 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix}.$$

Finally we need to ensure the elements of K sum to 1, therefore

$$K_{gaussian} = \frac{1}{331} \begin{pmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 20 & 33 & 20 & 4 \\ 7 & 33 & 55 & 33 & 7 \\ 4 & 20 & 33 & 20 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{pmatrix}.$$

The effects of applying the Gaussian blur filter can be seen in figure 3.5.



(a) original image (b) Gaussian blur

Figure 3.5: The affects of applying a Gaussian blur filter to an image.

3.2.4 Sharpening Images

A common tool used in image manipulation software is the sharpen tool. Given a photograph that is slightly out of focus the edges around features in the photograph will appear blurred and blend into each other, sharpening is used to accentuate these edges giving the appearance of a sharper image (this is often the default 'enhance' filter).

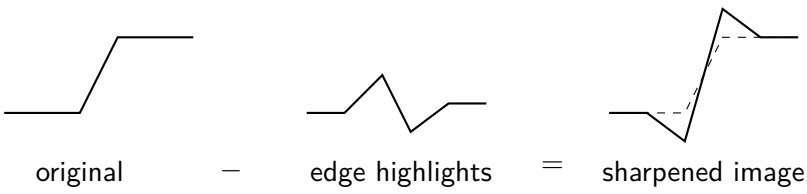


Figure 3.6: Intensity curves over space for sharpening an image.

To sharpen an image we subtract highlights of the changes in the colour of pixels which accentuates the edges of objects in the image from the original image (figure 3.6).

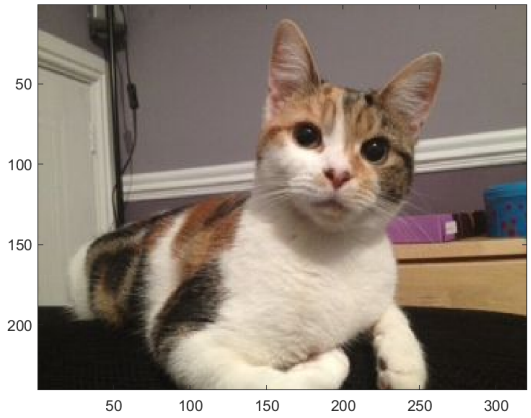
The kernel used to detect edges is

$$K_{edge} = \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix},$$

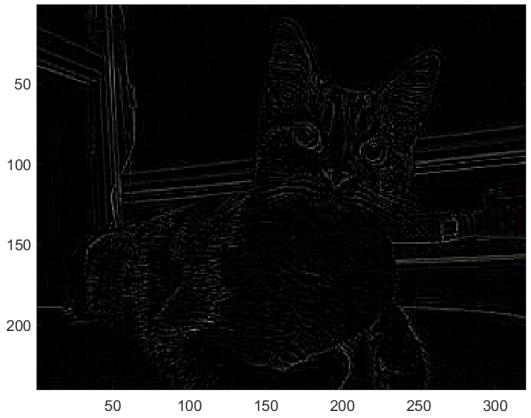
so subtracting this from the identity kernel (a kernel such that $X * K = X$) gives

$$K_{sharpen} = K_{identity} - K_{edge} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} - \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{pmatrix}. \tag{25}$$

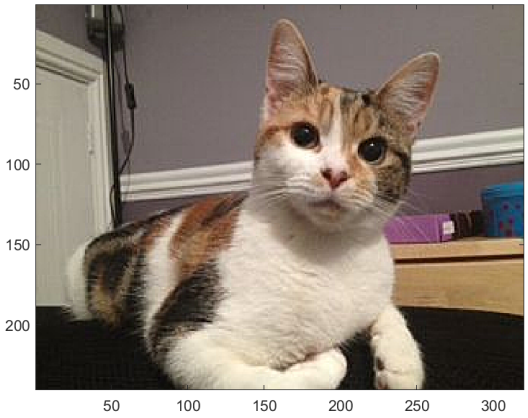
The effects of sharpening an image is shown in figure 3.7.



(a) original



(b) edge highlights



(c) sharpened

Figure 3.7: Sharpening an image.

3.2.5 Embossing

Images can be **embossed** to accentuate changes in pixel colour by applying convolution using the following kernel

$$K_{emboss} = \begin{pmatrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$

The effects of applying this kernel is shown in figure 3.8.

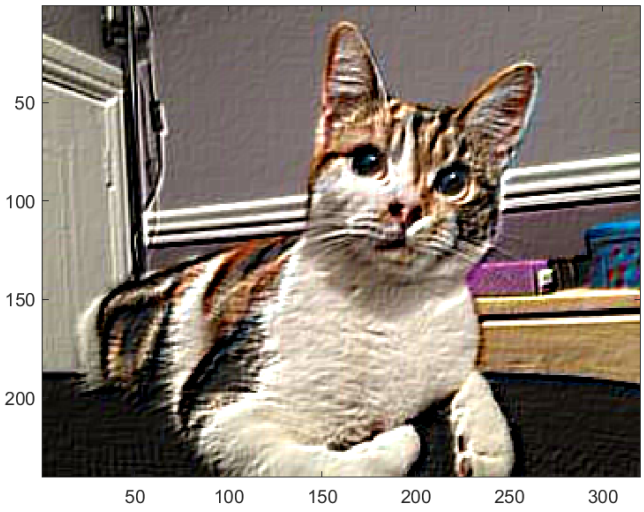


Figure 3.8: Result of convolution using the embossing kernel.

3.2.6 Edge detection

A common application of image processing requires the detection of edges of objects in a raster image. The **Sobel operator** uses convolution to calculate approximations of the colour gradient in the x and y directions using the following convolutions

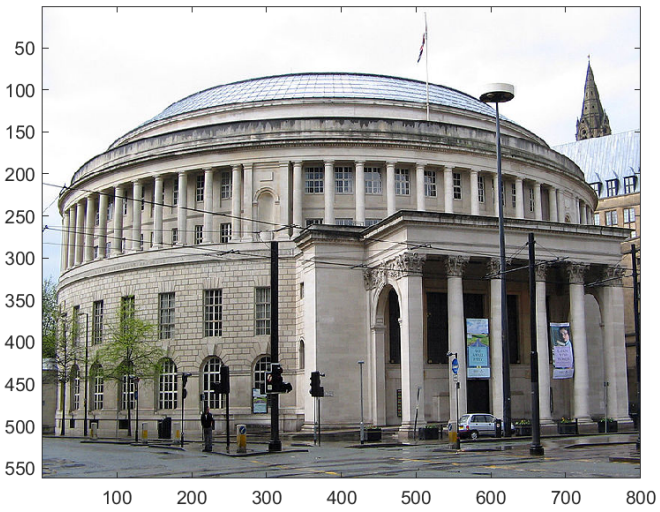
$$G_x = X * \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix},$$
$$G_y = X * \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}.$$

The gradient approximations are then combined using

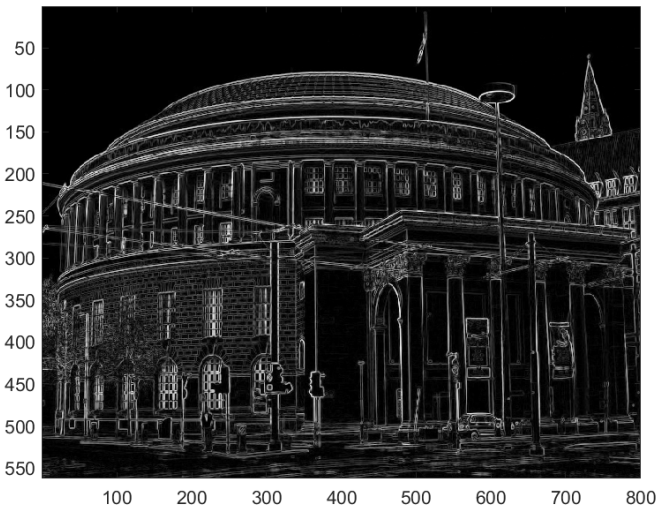
$$G = \sqrt{G_x^2 + G_y^2},$$

for each pixel. The elements of G are then normalised so they are in the range $[0, 255]$.

The result of applying the Sobel operator is shown in figure 3.9.



(a) input image



(b) Sobel operator

Figure 3.9: Edge detection using the Sobel operator on a photograph of Manchester's central library.

Chapter 4

Bézier Curves

There are many applications in computer graphics that require smooth curves and surfaces. Examples include Computer Aided Design (CAD), automobile design, computer generated animation, typefaces, computer games etc. Bézier curves is a common method used to draw smooth curves.

Definition 16. A **polynomial** is an mathematical expression involving a sum of variables raised to powers multiplied by coefficients. The general form of a polynomial is

$$\sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \ldots + a_n x^n,$$

where $a_0, a_1, \ldots, a_n$ are the **coefficients** of the variable x. It is common practice to write a polynomial in descending order by the power of the variable, for example,

$$a_3 x^3 + a_2 x^2 + a_1 x + a_0.$$

Definition 17. The **degree** of a polynomial is the highest power of the variables in the polynomial.

Polynomials of degree 1, 2, 3 and 4 are known as **linear**, **quadratic**, **cubic** and **quartic** polynomials respectively, for example, consider the following polynomial functions:

$$\begin{aligned} f(x) &= x + 1, \\ g(x) &= x^2 + x + 1, \\ h(x) &= x^3 + x^2 + x + 1, \\ i(x) &= x^4 + x^3 + x^2 + x + 1. \end{aligned}$$

Here $f(x)$ has degree 1 and is a linear polynomial function; $g(x)$ has degree 2 and is a quadratic polynomial function; $h(x)$ has degree 3 and is a cubic polynomial function; and $i(x)$ has degree 4 is a quartic polynomial function.

Definition 18. Parametric equations are a set of equations that express a series of quantities in terms of independent variables known as **parameters**.

For example, the straight line joining two points with co-ordinates (x_0, y_0) and (x_1, y_1) can be expressed using the following parametric equations

$$\begin{aligned} x(t) &= (1-t)x_0 + tx_1, \\ y(t) &= (1-t)y_0 + ty_1, \end{aligned}$$

where t is a parameter in the range $t \in [0, 1]$ (i.e., $0 \leq t \leq 1$). The values of $x(t)$ and $y(t)$ give the co-ordinates of a point on the line between the two points. Let $\mathbf{p}_0 = (x_0, y_0)$ and $\mathbf{p}_1 = (x_1, y_1)$ denote the two points and $\mathbf{p} = (x, y)$ denote a point on the line $\mathbf{p}_0 \to \mathbf{p}_1$ then

$$\mathbf{p}(t) = (1-t)\mathbf{p}_0 + t\mathbf{p}_1 \tag{26}$$

which is known as the **parametric equation of a straight line**.

Example 9 Calculate the points on the straight line joining the two points with co-ordinates $(2, 3)$ and $(10, 11)$ using: (i) $t = 0$; (ii) $t = 0.25$; (iii) $t = 0.5$; (iv) $t = 1$.

(i) $\mathbf{p}(0) = (1 - 0)(2, 3) + 0(10, 11) = (2, 3)$;

(ii) $\mathbf{p}(0.25) = (1 - 0.25)(2, 3) + 0.25(10, 11) = (4, 5)$;

(iii) $\mathbf{p}(0.5) = (1 - 0.5)(2, 3) + 0.5(10, 11) = (6, 7)$;

(iv) $\mathbf{p}(1) = (1 - 1)(2, 3) + 1(10, 11) = (10, 11)$.

Note that when $t = 0$ and $t = 1$ the co-ordinates are the endpoints of the line.

4.1 Bézier curves

A **Bézier curve** is a parametric curve that is used in computer graphics to draw smooth lines. Named after Pierre Bézier (1910 – 1999) who used Bézier curves in 1962 whilst working as an engineer at Renault, it was however Paul de Casteljau who derived them three years earlier whilst working for the rival car manufacturer Citroën. In addition to engineering applications, Bézier curves are used extensively in 3D modelling and in particular fonts (figure 4.1).

A Bézier curve is defined by a number of **control points** that determine the start and end points and the shape of the curve. The number of control points used also determines the degree of the curve where a Bézier curve of degree n requires $n+1$ control points to define it. The higher the degree of the curve, the more turns it can take. In practice, Bézier curves tend to have a degree of no larger than four (quartic) but mostly degree 3 (cubic) Bézier curves are used.

A major benefit of using Bézier curves is that alignment transformations can be applied to the control points instead of the points on the curve thereby saving computational effort. For example, the smaller S in figure 4.1 is drawn by applying scaling and translation transformations to the control points (red circles and crosses) defining the bigger S.

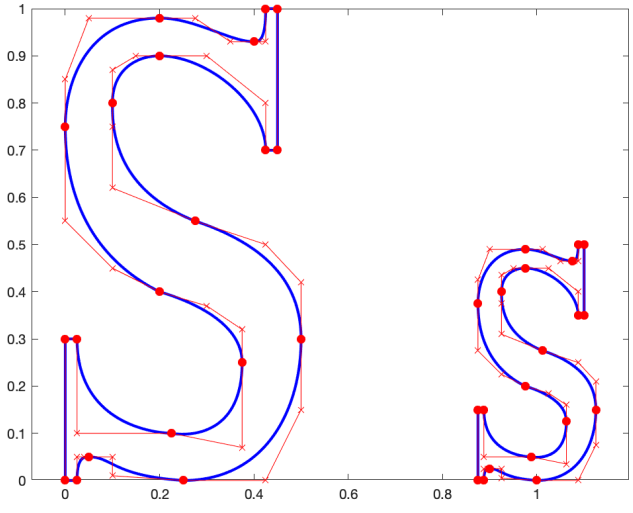


Figure 4.1: Bézier curves used to draw the character S.

4.1.1 Derivation of a Bézier curve

A degree n Bézier curve is defined by $n+1$ control points $(\mathbf{p}_0, \mathbf{p}_1, \ldots, \mathbf{p}_n)$. Consider the quadratic Bézier curve defined by the three control points $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$ in figure 4.2.

For a given value of the parameter t, the points that lie on the Bézier curve will depend upon the control points. When $t = 0$ and $t = 1$, the Bézier curve starts and ends at the two outer control points $\mathbf{p}_0$ and

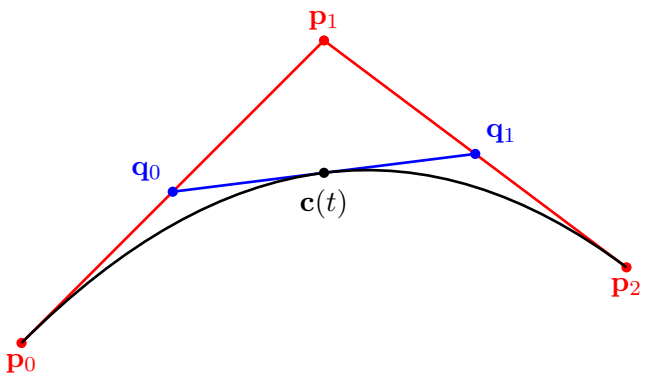


Figure 4.2: A quadratic Bézier curve.

$\mathbf{p}_2$. For all other values of t, a point on the Bézier curve will lie somewhere along the line $\mathbf{q}_0 \to \mathbf{q}_1$ where $\mathbf{q}_0$ and $\mathbf{q}_1$ lie somewhere along the lines $\mathbf{p}_0 \to \mathbf{p}_1$ and $\mathbf{p}_1 \to \mathbf{p}_2$ respectively.

The points $\mathbf{q}_0$ and $\mathbf{q}_1$ are calculated using the parametric equation of a straight line, equation (26).

$$\mathbf{q}_0 = (1-t)\mathbf{p}_0 + t\mathbf{p}_1, \tag{27a}$$
$$\mathbf{q}_1 = (1-t)\mathbf{p}_1 + t\mathbf{p}_2. \tag{27b}$$

Note that when $t = 0$, $\mathbf{q}_0 = \mathbf{p}_0$ and as t increases, $\mathbf{q}_0$ gets further away from $\mathbf{p}_0$ and closer to $\mathbf{p}_1$ until $\mathbf{q}_0 = \mathbf{p}_1$ when $t = 1$ (and similar for $\mathbf{q}_1$ and the control points $\mathbf{p}_1$ and $\mathbf{p}_2$). A point on the quadratic Bézier curve, $\mathbf{c}(t) = (x, y)$ lies on the line $\mathbf{q}_0 \to \mathbf{q}_1$, i.e.,

$$\mathbf{c}(t) = (1-t)\mathbf{q}_0 + t\mathbf{q}_1. \tag{28}$$

Substituting equations (27a) and (27b) into equation (28) gives:

$$\begin{aligned}\mathbf{c}(t) &= (1-t)((1-t)\mathbf{p}_0 + t\mathbf{p}_1) + t((1-t)\mathbf{p}_1 + t\mathbf{p}_2) \\ &= (1-t)^2\mathbf{p}_0 + t(1-t)\mathbf{p}_1 + t(1-t)\mathbf{p}_1 + t\mathbf{p}_2\end{aligned}$$

Collecting like terms gives the polynomial function for a point on a quadratic Bézier curve

$$\mathbf{c}(t) = (1-t)^2\mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2\mathbf{p}_2. \tag{29}$$

Example 10 A quadratic Bézier curve is defined by the control points $\mathbf{p}_0 = (1, 2)$, $\mathbf{p}_1 = (5, 6)$ and $\mathbf{p}_2 = (9, 3)$. The point on the quadratic Bézier curve corresponding to $t = 0.25$ is calculated using:

$$\begin{aligned}\mathbf{c}(0.25) &= (1-0.25)^2(1,2) + 2(0.25)(1-0.25)(5,6) + 0.25^2(9,3) \\ &= (0.5625, 1.125) + (1.875, 2.25) + (0.5625, 0.1875) \\ &= (3, 3.5625).\end{aligned}$$

4.1.2 General form of a Bézier curve

The general form of a degree n Bézier curve defined by the control points $\mathbf{p}_i$ (where $i = 0, 1, \ldots, n$) is

$$\mathbf{c}(t) = \sum_{i=0}^{n} b_{i,n}(t)\mathbf{p}_i, \tag{30}$$

where $b_{i,n}(t)$ are called **Bernstein polynomials** that are defined using

$$b_{i,n}(t) = \binom{n}{i} t^i (1-t)^{n-i}, \tag{31}$$

and $\binom{n}{i}$ is the Binomial coefficient.

Definition 19. The **Binomial coefficient** is written using $\binom{n}{i}$ and is read as "n choose i" since it gives the number of ways of choosing i items from a set of n items (note that $\binom{n}{i}$ is also written as ${}^nC_i$ in some textbooks). The value of the Binomial coefficient is calculating using

$$\binom{n}{i} = \frac{n!}{i!(n-i)!},$$

where $n!$ denotes the factorial of n, e.g., $n! = n \times n-1 \times \ldots \times 2 \times 1$.

An easy way to calculate the Binomial coefficient is to use Pascal's triangle (figure 4.3) where each value in the interior of the triangle is the sum of the two values immediately above. The value of $\binom{n}{i}$ is the number in the nth row and ith value along from the left starting the count at 0.

For example, to determine $\binom{3}{2}$ we look at the number in the 4th row down and 3rd number across.

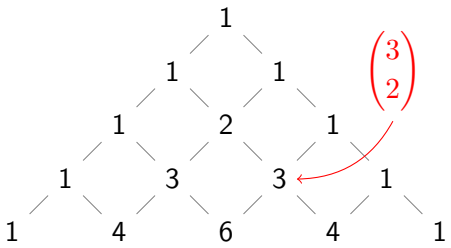


Figure 4.3: Pascal's triangle

Example 11 Use equation (30) to derive the parameteric equation for a quadratic Bézier curve.

Since we require a quadratic curve, $n = 2$ and

$$\mathbf{c}(t) = \sum_{i=0}^{2} b_{i,n}\mathbf{p}_i = b_{0,n}\mathbf{p}_0 + b_{1,n}\mathbf{p}_1 + b_{2,n}\mathbf{p}_2.$$

The Bernstein polynomials are

$$\begin{aligned} b_{0,2}(t) &= \binom{2}{0} t^0(1-t)^{2-0} = (1-t)^2, \\ b_{1,2}(t) &= \binom{2}{1} t^1(1-t)^{2-1} = 2t(1-t), \\ b_{2,2}(t) &= \binom{2}{2} t^2(1-t)^{2-2} = t^2, \end{aligned}$$

so the quadratic Bézier curve is

$$\mathbf{c}(t) = (1-t)^2\mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2\mathbf{p}_2,$$

which is the same as equation (29).

4.1.3 Behaviour of the Bernstein polynomials

The Bernstein polynomials for a quadratic Bézier curve have been plotted in figure 4.4. When $t \to 0$, $b_{0,2}(t) \to 1$, $b_{1,2}(t) \to 0$ and $b_{1,2} \to 0$ so it is only the $b_{0,2}(t)$ term (and therefore the $\mathbf{p}_0$ control point) that has any influence on the Bézier curve. As t increases, the affect that the other Bernstein polynomials have on the shape of the curve increases while $b_{0,2}(t)$ decreases. When $t \to 0.5$ it is the $b_{1,2}(t)$ term that has most influence on the shape of the curve. As $t \to 1$, $b_{0,2}(t)$ and $b_{1,2}(t) \to 0$ whereas $b_{2,2}(t) \to 1$ so it is only the final control point that has an influence. Note that for all values of t, the Bernstein polynomials sum to 1.

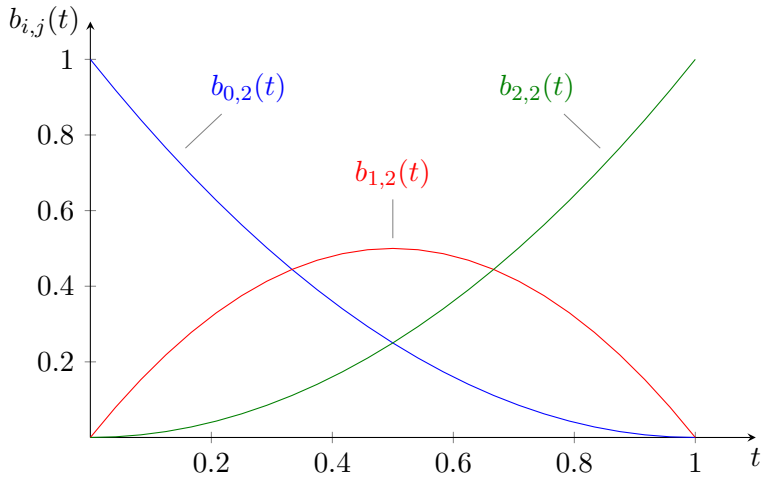


Figure 4.4: The Bernstein polynomials for a quadratic Bézier curve.

Similar behaviour can be seen in figure 4.5 that shows the Bernstein polynomials for a cubic Bézier curve with the exception that there are now four Bernstein polynomials.

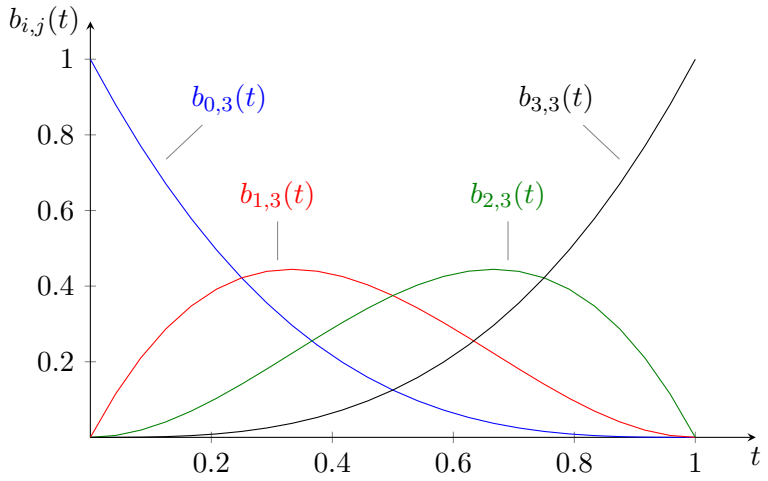


Figure 4.5: The Bernstein polynomials for a cubic Bézier curve.

4.1.4 Expressing Bézier curves in matrix form

For convenience, Bézier curves are written in matrix form

$$\mathbf{c}(t) = \begin{pmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \cdots & \mathbf{p}_{n-1} & \mathbf{p}_n \end{pmatrix} M \begin{pmatrix} t^n \\ t^{n-1} \\ \vdots \\ t \\ 1 \end{pmatrix},$$

where $\mathbf{p}_i$ are the co-ordinates of the control points expressed as column vectors and M is an $(n+1)\times(n+1)$ matrix. For example, consider the quadratic Bézier curve

$$\mathbf{c}(t) = (1-t)^2\mathbf{p}_0 + 2t(1-t)\mathbf{p}_1 + t^2\mathbf{p}_2.$$

Expanding out the brackets gives

$$\mathbf{c}(t) = (t^2 - 2t + 1)\mathbf{p}_0 + (-2t^2 + 2t)\mathbf{p}_1 + t^2\mathbf{p}_2,$$

so the quadratic Bézier curve expressed in matrix form is

$$\mathbf{c}(t) = \begin{pmatrix} \mathbf{p}_0 & \mathbf{p}_1 & \mathbf{p}_2 \end{pmatrix} \begin{pmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} t^2 \\ t \\ 1 \end{pmatrix}. \tag{32}$$

Note that each column of M contains the coefficients of t^2, t and 1 that are multiplied by the control points.

Example 12 Use equation (32) to calculate the point on a quadratic Bézier curve defined by the control points $\mathbf{p}_0 = (1,2)$, $\mathbf{p}_1 = (5,6)$ and $\mathbf{p}_2 = (9,3)$ when $t = 0.25$.

$$\begin{aligned} \mathbf{c}(0.25) &= \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 3 \end{pmatrix} \begin{pmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} 0.25^2 \\ 0.25 \\ 1 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 3 \end{pmatrix} \begin{pmatrix} 0.5625 \\ 0.3750 \\ 0.0625 \end{pmatrix} \\ &= \begin{pmatrix} 3 \\ 3.5625 \end{pmatrix}. \end{aligned}$$

Therefore $\mathbf{c}(0.25) = (3, 3.5625)$.

4.1.5 Properties of Bézier curves

Bézier curves have the following properties:

- a Bézier curve begins at point $\mathbf{p}_0$ and ends at point $\mathbf{p}_n$;
- a Bézier curve is a straight line if and only if it is possible to draw a straight line through all of the control points;
- the start and end of a Bézier curve is tangential to the start and end section of the control polygon;
- a Bézier curve can be split into two Bézier curves;
- a Bézier curve is contained within its control polygon. This is known as the **convex hull property** (figure 4.6).

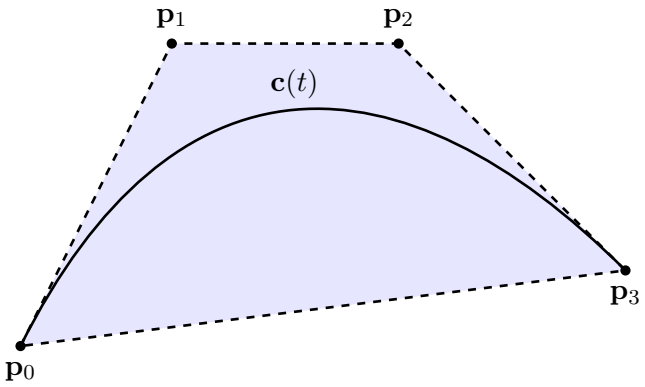


Figure 4.6: The Bézier curve is contained within its control polygon.

4.1.6 MATLAB code

The program in listing 4.1 defines four control points in the array `P`, uses the function `quadratic_bezier` to calculate the points on the Bézier curve and plots the results shown in figure 4.7.

Listing 4.1: MATLAB function to calculate the points on a quadratic Bézier curve.

```
clear

% Define control points
P = [ 0.1, 0.5, 0.9 ;
    0.1, 0.8, 0.4 ];

% Calculate Bezier curve
C = bezier(P);

% Plot Bezier curve and control points
plot(P(1, :), P(2, :), 'ro-', 'markerfacecolor', 'r')
hold on
plot(C(1, :), C(2, :), 'b-')
hold off

axis([0, 1, 0, 1])
xlabel('$x$', 'fontsize', 16, 'interpreter', 'latex')
ylabel('$y$', 'fontsize', 16, 'interpreter', 'latex')
legend('control points', 'quadratic Bezier curve', 'fontsize', 12, 'interpreter', '
    latex')


function C = bezier(P)

t = linspace(0, 1, 100);
M = [ 1, -2, 1 ; -2, 2, 0 ; 1, 0, 0 ];
T = [ t.^2 ; t ; t.^0 ];
C = P * M * T;

end
```

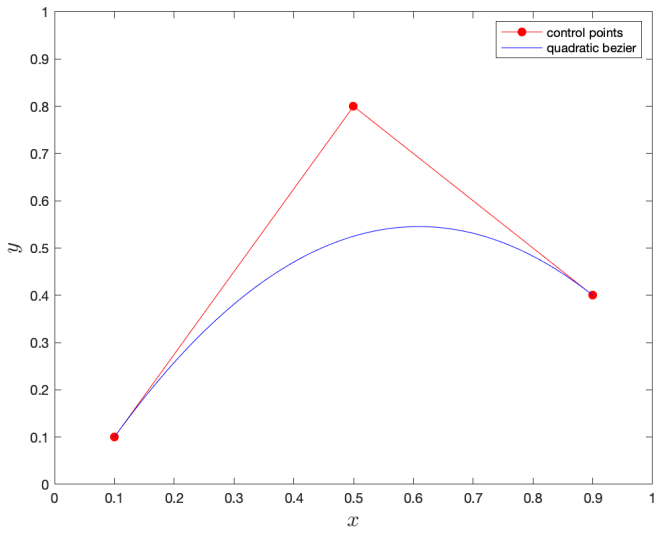


Figure 4.7: A quadratic Bézier curve.

A MATLAB function that calculates a cubic Bézier curve (code not shown) has been used to draw different curves as shown in figure 4.8. Unlike the quadratic curve, a cubic curve can exhibit two changes of direction.

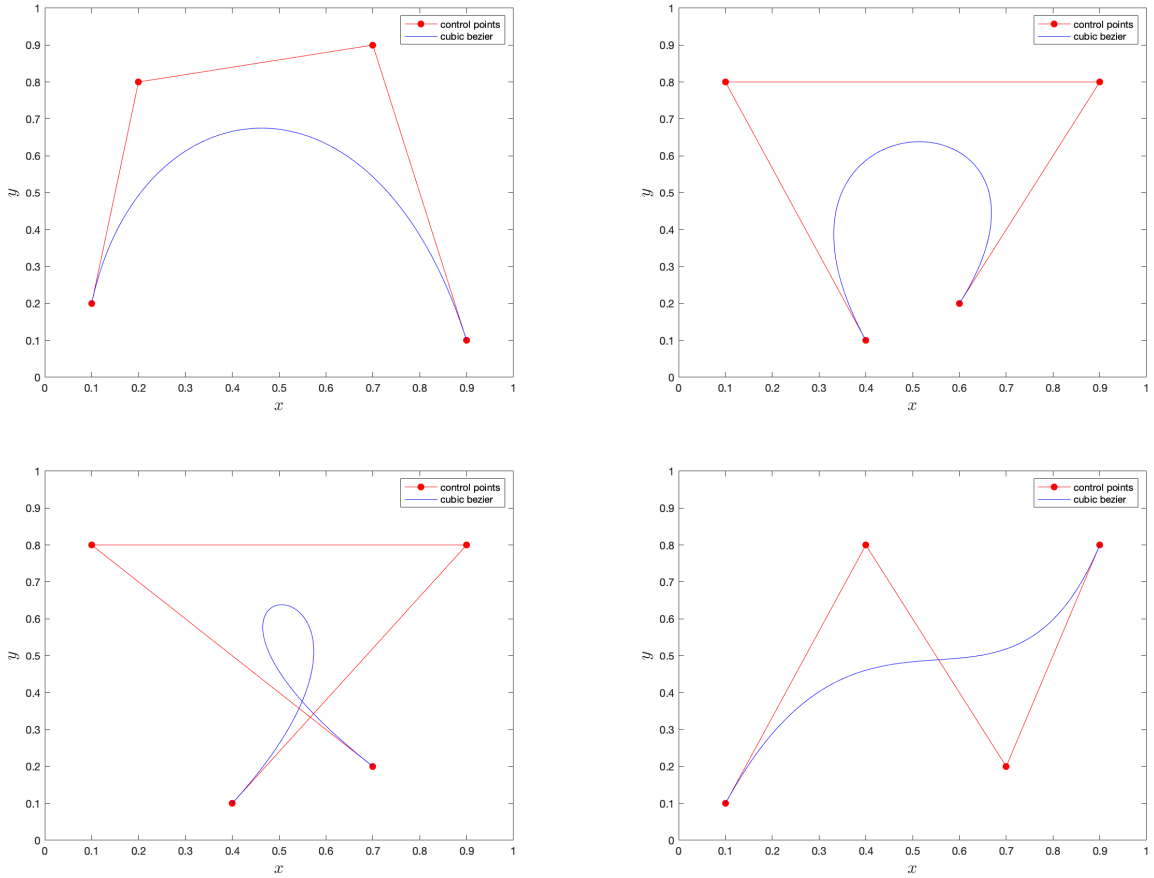


Figure 4.8: Cubic Bézier curves.

4.1.7 Drawing shapes using Bézier curves

Bézier curves can be used to draw shapes by joining the last point of one curve to the first point of the next. For example, consider figure 4.9 where a shape is defined using 4 quadratic Bézier curves.

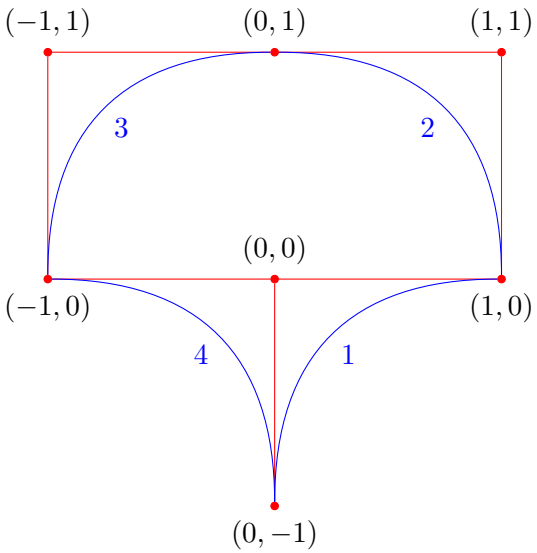


Figure 4.9: A shape defined using 4 quadratic Bézier curves.

The control points that define each curve are given below

$$P_1 = \begin{pmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \end{pmatrix}, \quad P_2 = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix}, \quad P_3 = \begin{pmatrix} 0 & -1 & -1 \\ 1 & 1 & 0 \end{pmatrix}, \quad P_4 = \begin{pmatrix} -1 & 0 & 0 \\ 0 & 0 & -1 \end{pmatrix}.$$

Since we will want to apply transformations to our shape we need to use homogeneous co-ordinates and for convenience we can combined all 4 sets of control points into a single matrix

$$P = (P_1, P_2, P_3, P_4) = \begin{pmatrix} 0 & 0 & 1 & 1 & 1 & 0 & 0 & -1 & -1 & -1 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & -1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

The Bézier curves are calculated using each set of 3 columns from P are the control points.

The MATLAB program in listing 4.2 defines the control points for this shape and calculates each quadratic Bézier curves and stores the co-ordinates in a single array `C1`. The control points are rotated anti-clockwise by $\frac{\pi}{4}$ and translated by $(2, 1)$ resulting giving another set of control points stored in the array `Q` that define another shape. The Bézier curves using the control points in `Q` are calculated and are stored in the array `C2`. The two shapes are plotted and the result shown in figure 4.10.

Listing 4.2: MATLAB program that uses Bézier curves to draw shapes.

```
% Define control points (using homogeneous co-ordinates)
P = [ 0, 0, 1, 1, 1, 0, 0, -1, -1, -1, 0, 0 ;
    -1, 0, 0, 0, 1, 1, 1, 1, 0, 0, 0, -1 ;
    1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 ];

% Calculate Bezier curves
C1 = [ bezier(P(1:2, 1:3)), bezier(P(1:2, 4:6)), ...
    bezier(P(1:2, 7:9)), bezier(P(1:2, 10:12)) ];

% Rotate and translate control points
R = [ cos(pi/4), -sin(pi/4), 0 ;
    sin(pi/4), cos(pi/4), 0 ;
    0, 0, 1 ];
T = [ 1, 0, 2 ;
    0, 1, 1 ;
    0, 0, 1 ];
Q = T * R * P;

% Calculate Bezier curves for the transformed shape
C2 = [ bezier(Q(1:2, 1:3)), bezier(Q(1:2, 4:6)), ...
    bezier(Q(1:2, 7:9)), bezier(Q(1:2, 10:12)) ];

% Plot shapes
hold on
plot(C1(1, :), C1(2, :), 'b')
plot(C2(1, :), C2(2, :), 'b')
hold off
axis equal


function C = bezier(P)

M = [ 1, -2, 1 ;
    -2, 2, 0 ;
    1, 0, 0 ];
t = linspace(0, 1, 100);
T = [ t.^2 ; t ; t.^0 ];
C = P * M * T;

end
```

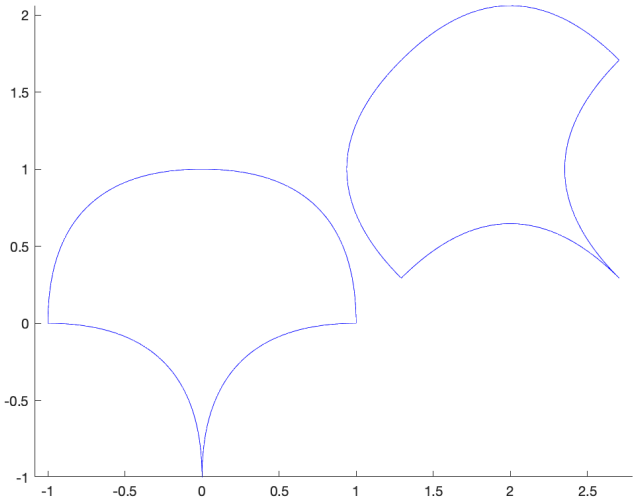


Figure 4.10: Shapes drawn using Bézier curves.

4.2 Bézier surfaces

The concept of Bézier curves can be extended to three dimensions to create a smooth curved surface called a **Bézier surface**. Let P_x, P_y and P_z be $(n+1)\times(n+1)$ matrices containing the x, y and z co-ordinates of the control points, i.e.,

$$P_x = \begin{pmatrix} x_{00} & x_{01} & \cdots & x_{0n} \\ x_{10} & x_{11} & \cdots & x_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n0} & x_{n1} & \cdots & x_{nn} \end{pmatrix}, \quad P_y = \begin{pmatrix} y_{00} & y_{01} & \cdots & y_{0n} \\ y_{10} & y_{11} & \cdots & y_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ y_{n0} & y_{n1} & \cdots & y_{nn} \end{pmatrix}, \quad P_z = \begin{pmatrix} z_{00} & z_{01} & \cdots & z_{0n} \\ z_{10} & z_{11} & \cdots & z_{1n} \\ \vdots & \vdots & \ddots & \vdots \\ z_{n0} & z_{n1} & \cdots & z_{nn} \end{pmatrix}.$$

then the co-ordinates of the points on a Bézier surface are calculated using

$$X(u,v) = \sum_{j=0}^{n} \left(\sum_{i=0}^{n} b_{i,n}(u)[P_x]_{ij} \right) b_{j,n}(v), \tag{33a}$$

$$Y(u,v) = \sum_{j=0}^{n} \left(\sum_{i=0}^{n} b_{i,n}(u)[P_y]_{ij} \right) b_{j,n}(v), \tag{33b}$$

$$Z(u,v) = \sum_{j=0}^{n} \left(\sum_{i=0}^{n} b_{i,n}(u)[P_z]_{ij} \right) b_{j,n}(v). \tag{33c}$$

where X, Y and Z are matrices containing the x, y and z co-ordinates of a point on the Bézier surface and $u, v \in [0,1]$ (i.e., similar to the values of t for a Bézier curve). The surface is only guaranteed to pass through the four control points at the corners. Multiple Bézier surfaces can be tessellated to form complex three-dimensional objects (e.g., the Utah teapot shown in figure 4.12).

Similar to Bézier curves, a Bézier surface can be written in matrix form as:

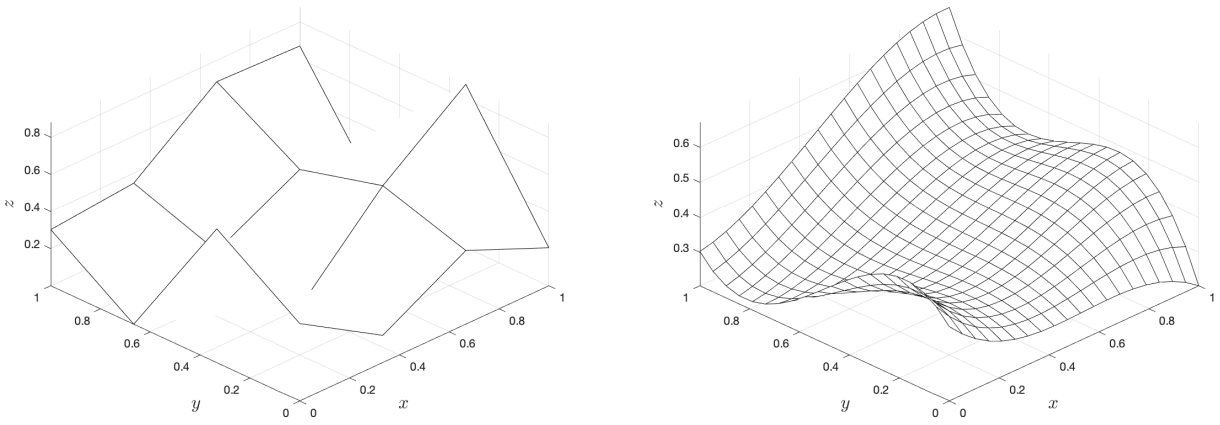
$$X(u,v) = \begin{pmatrix} u^n & u^{n-1} & \cdots & u & 1 \end{pmatrix} \cdot M \cdot P_x \cdot M \cdot \begin{pmatrix} v^n \\ v^{n-1} \\ \vdots \\ v \\ 1 \end{pmatrix}, \tag{34}$$

where M is an $(n+1)\times(n+1)$ matrix for the Bézier curve, e.g., equation (32). For example, the x co-ordinates of the points on a quadratic Bézier surface are calculated using

$$X(u,v) = \begin{pmatrix} u^2 & u & 1 \end{pmatrix} \begin{pmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} x_{00} & x_{01} & x_{02} \\ x_{10} & x_{11} & x_{12} \\ x_{20} & x_{21} & x_{22} \end{pmatrix} \begin{pmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} v^2 \\ v \\ 1 \end{pmatrix},$$

and similar for $Y(u,v)$ and $Z(u,v)$.

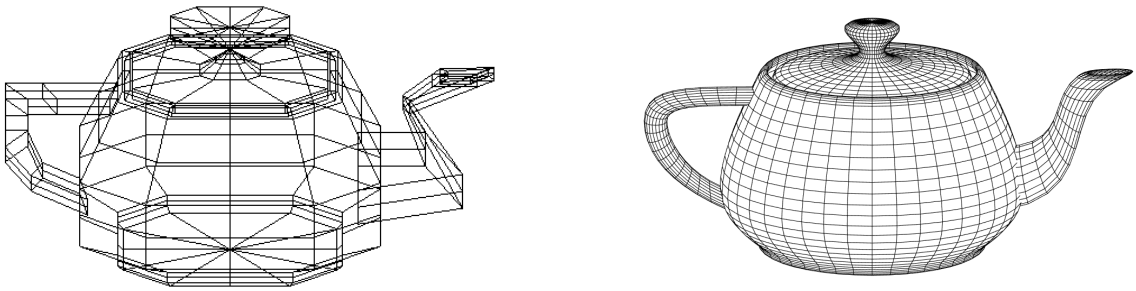
Two plots showing 16 random control points and the corresponding Bézier surface are shown in figure 4.11



(a) control points (b) Bézier surface

Figure 4.11: Bézier surface generated using random control points.

The control points for the Utah teapot and the cubic Bézier surfaces that they define are shown in figure 4.12. Note that the plot of the control points does not look realistic, when Bézier surfaces are applied the image looks like an actual teapot.

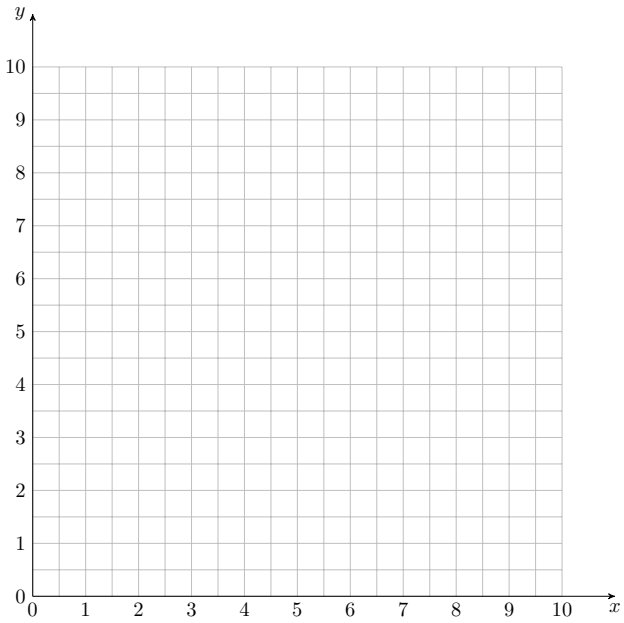


(a) control points (b) Bézier surfaces

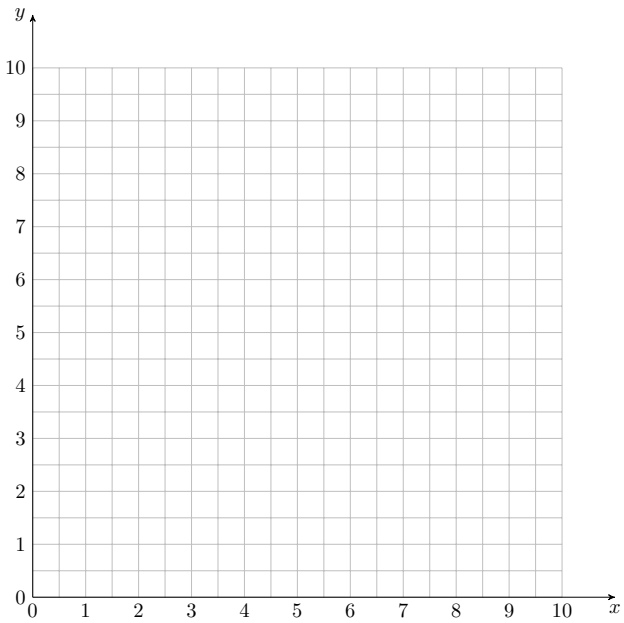
Figure 4.12: The control points for the Utah teapot and with Bézier surfaces applied.

4.3 Exercises

1. A quadratic Bézier curve is defined by the control points $\mathbf{p}_0 = (1, 1)$, $\mathbf{p}_1 = (6, 8)$ and $\mathbf{p}_2 = (9, 2)$. Calculate the co-ordinates of the points on the Bézier curve using the following values. Copy out the axes below and plot the Bézier curve and control polygon.
   (a) $t = 0.2$;
   (b) $t = 0.4$;
   (c) $t = 0.6$;
   (d) $t = 0.8$.



2. (a) Derive the parametric polynomial for a cubic Bézier curve using equation (30).
   (b) Hence find the matrix M when the cubic Bézier curve is written in matrix form.
3. Using your result from 2(b) or otherwise, sketch the cubic Bézier curve defined by the control points $\mathbf{p}_0 = (1, 1)$, $\mathbf{p}_1 = (3, 9)$, $\mathbf{p}_2 = (9, 1)$ and $\mathbf{p}_3 = (6, 9)$.



The solutions to these exercises can be found on page 138.

Chapter 5

Hidden Surface Removal

When rendering a scene in three-dimensions the overall impression of realism is much improved if we remove the parts of the object that cannot normally be seen by the viewer. This removes the wireframe impression of the object and makes it look more solid. Consider the drawing of the Utah teapot in figure 5.1. The image on the left shows the object drawn using all of the polygons that define the object whereas the image on the right is the same object with those polygons that should not be seen by the viewer removed. This eliminates the wireframe appearance of the object and the object appears to the viewer to be solid.

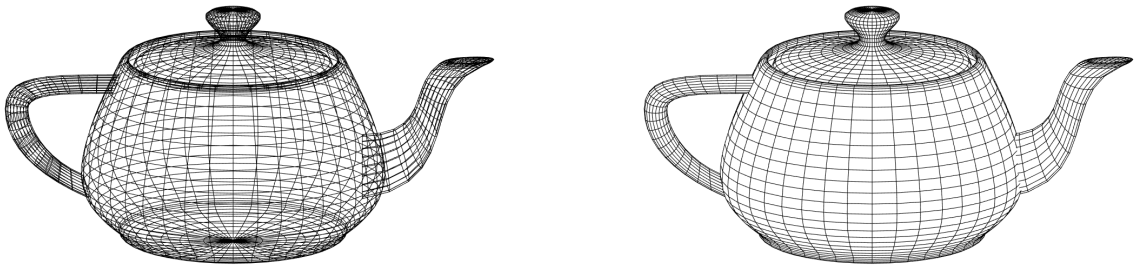


Figure 5.1: The Utah teapot drawn with and without hidden surfaces.

In this chapter we will be concentrating on hidden surface removal techniques in order to make the drawing of three-dimensional objects appear more realistic. Much of the advances in rendering of three-dimensional objects have been spearheaded by the Computer Aided Design (CAD) and computer games industries where accurate and realistic images important. Due to the complexity of three-dimensional scenes in modern computer games, movies and virtual environments, a method for removing hidden surfaces must be computationally efficient in order to be of practical use. We will consider three methods of hidden surface removal: back-face culling, the painter's algorithm and Binary Space Partitioning (BSP).

5.1 Defining objects

Three-dimensional objects are constructed using a set of polygons called **faces** that are used to form the surface of the object. Each face is defined by a set of **vertices**, the co-ordinates of which are expressed using **homogeneous co-ordinates** (homogeneous co-ordinates will be covered in Dr O'Brien's part of the unit).

The vertices and faces of an object are representing in two arrays V and F. V is a $4 \times n$ array where n is

the total number of vertices of an object. The x, y and z object space co-ordinates are contained in the first three rows and the fourth row contains 1's (note that some people write V as a $n \times 4$ matrix where the x, y and z co-ordinates are contained in the first three columns of V). Each row of the F array contains the column number (or row number if V is a $n \times 4$ array) of the vertices that make up each face.

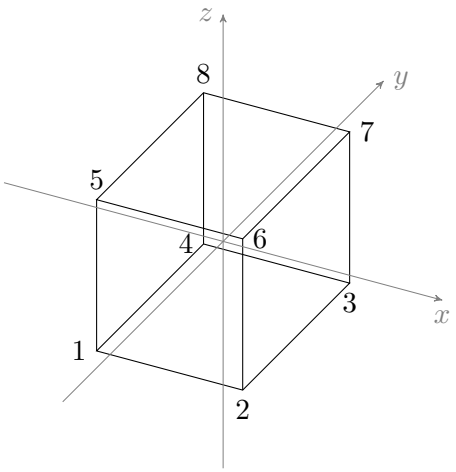


Figure 5.2: Cube object defined in the object space.

5.1.1 The vertex array

Consider the cube object shown in figure 5.2 which is defined in the object space using a right-handed co-ordinate system. The eight vertices that define the object are labelled 1 through 8. We want the centre of the object to have co-ordinates $(0, 0, 0)$ so if we consider a cube with side lengths of 2 the V array is (a cube like this is known as a **unit cube** because all of the vertex co-ordinates are 1 or -1)

$$V = \begin{pmatrix} -1 & 1 & 1 & -1 & -1 & 1 & 1 & -1 \\ -1 & -1 & 1 & 1 & -1 & -1 & 1 & 1 \\ -1 & -1 & -1 & -1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

Here the first column of V contains the co-ordinates of vertex 1, the second column contains the co-ordinates of vertex 2 and so on. Note that in this cube object each vertex is shared by three different faces of the object.

5.1.2 The face array

Looking at the cube object in figure 5.2 along the y axis the face of the object we can see is defined by vertices 1, 2, 5 and 6. Objects are defined such that the normal vectors to each face is pointing away from the centre of the object.

Definition 20. The **normal vector** of a plane is a vector that points in a perpendicular direction to the plane. In $\mathbb{R}^3$ the normal vector can be computed using the cross produce, i.e., if $\mathbf{a}$ and $\mathbf{b}$ are any two vectors that lie on a plane then the normal vector $\mathbf{n}$ is

$$\mathbf{n} = \mathbf{a} \times \mathbf{b}.$$

A plane has two normal vectors pointing in opposite directions so we list the vertices of a face in **anti-clockwise** order when looking towards the centre of the object. This ensures that when using a right-hand co-ordinate system the following method returns an outward pointing normal vector

$$\mathbf{n} = (\mathbf{v}_{i+1} - \mathbf{v}_i) \times (\mathbf{v}_{i+2} - \mathbf{v}_{i+1}). \tag{35}$$

Therefore the row of the face array that defines the face we can see when looking along the y-axis is $(1, 2, 6, 5)$.

The face array for the cube object in figure 5.2 will have 6 rows, one for each of the faces, and 4 columns since each face is a square, i.e.,

$$F = \begin{pmatrix} 1 & 2 & 6 & 5 \\ 2 & 3 & 7 & 6 \\ 3 & 4 & 8 & 7 \\ 4 & 1 & 5 & 8 \\ 4 & 3 & 2 & 1 \\ 5 & 6 & 7 & 8 \end{pmatrix}.$$

5.1.3 Calculating the normal vector for a face

Given an object defined using a vertex array V and a face array F then the normal vector for face i is calculated using

$$\mathbf{n}_i = (\mathbf{v}_{[F]_{i,j+1}} - \mathbf{v}_{[F]_{i,j}}) \times (\mathbf{v}_{[F]_{i,j+2}} - \mathbf{v}_{[F]_{i,j+1}}), \tag{36}$$

where $\mathbf{v}_k$ is the kth column of V.

Example 13 Calculate the normal vector for the first face in the cube object defined in sections 5.1.1 and 5.1.2.

The first row of the face array is $(1, 2, 6, 5)$ so

$$\mathbf{v}_{[F]_{1,1}} = \mathbf{v}_1 = (-1, -1, -1), \qquad \mathbf{v}_{[F]_{1,2}} = \mathbf{v}_2 = (1, -1, -1), \qquad \mathbf{v}_{[F]_{1,3}} = \mathbf{v}_6 = (1, -1, 1).$$

Therefore the normal vector for this face is

$$\mathbf{n} = ((1, -1, -1) - (-1, -1, -1)) \times ((1, -1, 1) - (1, -1, -1)) = (2, 0, 0) \times (0, 0, 2) = (0, -4, 0).$$

Note that $\mathbf{n}$ points in the negative y direction.

5.2 Back-face culling

The **back-face culling** method is the simplest of the hidden surface removal methods. It is used to remove the hidden surfaces in closed objects where the outward facing polygons are used. As the name suggests, back-face culling only renders the polygons that are front facing and ignores (culls) the back facing polygons.

Consider figure 5.3 that shows a closed object defined by six polygons A through F which are all outward facing (the normal vectors point away from the centre of the object). When the camera is located at point $\mathbf{p}$, only be able the front facing polygons A, B and C should be visible whereas the back facing polygons D, E and F should be obscured.

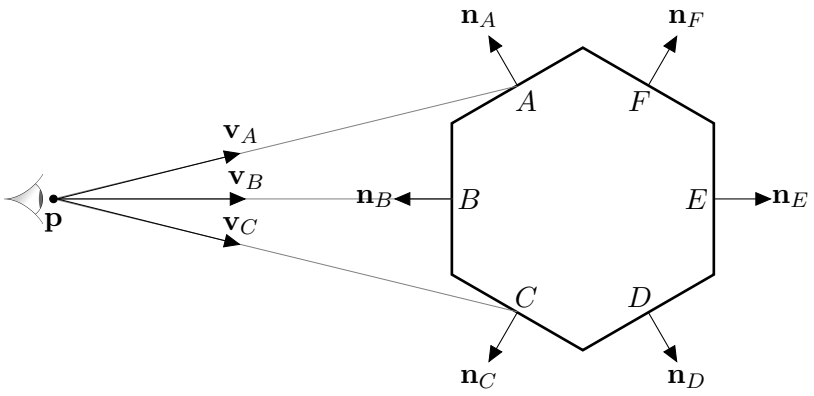


Figure 5.3: Back-face culling only renders the polygons that are facing the camera.

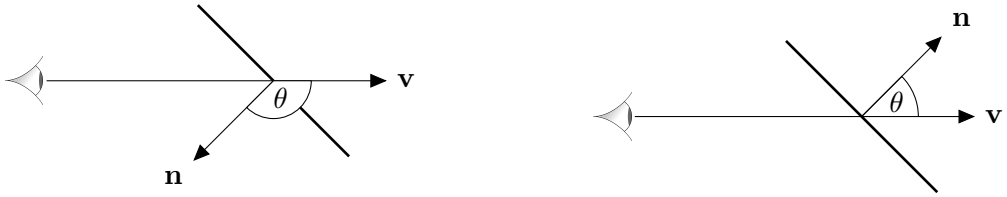
Polygons are determined to be front facing if the angle between the viewing vector pointing from the camera position to the polygon and the surface normal is greater than $\pi/2$ (figure 5.4). Using the definition of the dot product

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\| \cos(\theta),$$

and the fact that $\cos(\theta) < 0$ when $\theta > \pi/2$, then a polygon is front facing if

$$\mathbf{v} \cdot \mathbf{n} < 0,$$

else the polygon is back facing.



(a) front facing polygon: $\theta > \pi/2$ (b) back facing polygon: $\theta < \pi/2$

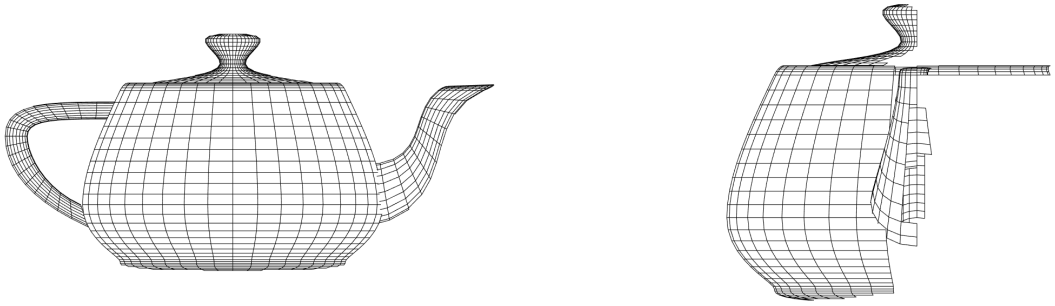
Figure 5.4: The angle between the viewing vector $\mathbf{v}$ and the surface normal $\mathbf{n}$ can be used to determine whether a polygon is front or back facing.

The back-face culling method is presented in algorithm 9. The function uses inputs of the face array for all the faces in the world space F and the position of which the world space is to be viewed $\mathbf{p}$ (note that $\mathbf{p}$ is usually $(0, 0, 0)$ when back-face culling is applied to the view space). The function returns F_{front} which is the face array for all front facing polygons.

Algorithm 9 The back-face culling algorithm

```
function BackFaceCulling(V, F, p)
    for each face in F do
        Calculate the view vector v
        Calculate the outward pointing normal vector for face, n
        if v · n < 0 then
            add face to F_front
        end if
    end for
    return F_front
end function
```

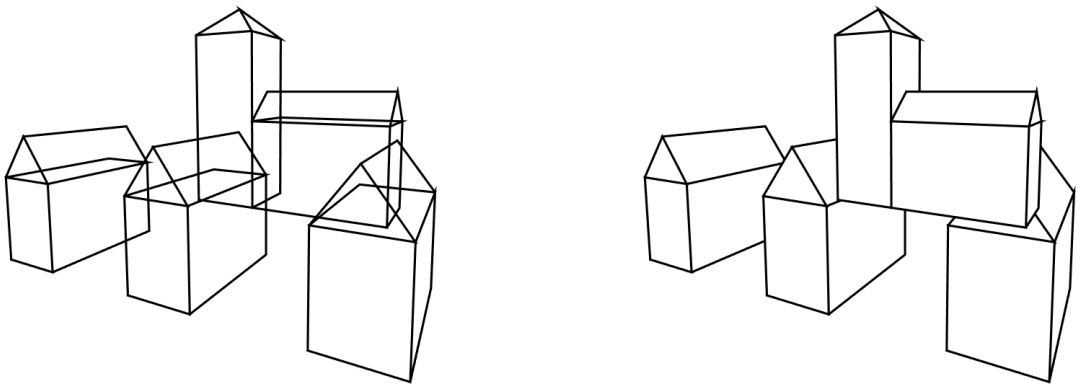
Figure 5.5 shows the Utah teapot drawn using back-face culling to remove the hidden surfaces. The view from the front shows that the teapot appears solid and realistic. Rotating the teapot so that the camera position is to the left shows that the polygons that construct the back of the teapot (from the point of view of the camera) are not rendered.



(a) frontview (b) side view

Figure 5.5: The Utah teapot drawn using back-face culling to remove the hidden surfaces.

Back-face culling has been applied to the screen space depiction of a virtual world seen and the before and after result is shown in figure 5.6. Note that back-face culling does not take into account whether surfaces are hidden by other objects in a scene.



(a) before back-face culling (b) after back-face culling (opaque polygons)

Figure 5.6: Before and after affects applying back-face culling to the screen space.

Example 14 A cube object in the view space is defined by the following vertex and face arrays. Given that the view space is viewed from position $(0, 0, 0)$, determined which faces are front facing

$$V = \begin{pmatrix} 5 & 6 & 7 & 6 & 5 & 6 & 7 & 6 \\ 3 & 2 & 3 & 4 & 3 & 2 & 3 & 4 \\ 2 & 2 & 2 & 2 & 4 & 4 & 4 & 4 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}, \qquad F = \begin{pmatrix} 1 & 2 & 6 & 5 \\ 2 & 3 & 7 & 6 \\ 3 & 4 & 8 & 7 \\ 4 & 1 & 5 & 8 \\ 4 & 3 & 2 & 1 \\ 5 & 6 & 7 & 8 \end{pmatrix}.$$

Face 1:

$$\mathbf{n} = ((6,2,2) - (5,3,2)) \times ((6,2,4) - (6,2,2)) = (1,-1,0) \times (0,0,2) = (-2,-2,0),$$
$$\mathbf{v} = (5,3,2),$$
$$\therefore \mathbf{v} \cdot \mathbf{n} = (5,3,2) \cdot (-2,-2,0) = -16 \implies \text{front facing.}$$

Face 2:

$$\begin{aligned}\mathbf{n} &= ((7,3,2)-(6,2,2)) \times ((7,3,4)-(7,3,2)) = (1,1,0)\times(0,2,0) = (2,-2,0),\\ \mathbf{v} &= (6,2,2),\\ \therefore \mathbf{v}\cdot\mathbf{n} &= (6,2,2)\cdot(2,-2,0) = 8 \implies \text{back facing.}\end{aligned}$$

Face 3:

$$\begin{aligned}\mathbf{n} &= ((6,4,2)-(7,3,2)) \times ((6,4,4)-(6,4,2)) = (-1,1,0)\times(0,0,2) = (2,2,0),\\ \mathbf{v} &= (7,3,2),\\ \therefore \mathbf{v}\cdot\mathbf{n} &= (7,3,2)\cdot(2,2,0) = 20 \implies \text{back facing.}\end{aligned}$$

Face 4:

$$\begin{aligned}\mathbf{n} &= ((5,3,2)-(6,4,2)) \times ((5,3,4)-(5,3,2)) = (-1,-1,0)\times(0,0,1) = (-2,2,0),\\ \mathbf{v} &= (6,4,2),\\ \therefore \mathbf{v}\cdot\mathbf{n} &= (6,4,2)\cdot(-2,2,0) = -4 \implies \text{front facing.}\end{aligned}$$

Face 5:

$$\begin{aligned}\mathbf{n} &= ((7,3,2)-(6,4,2)) \times ((6,2,2)-(7,3,2)) = (1,-1,0)\times(-1,-1,0) = (0,0,-2),\\ \mathbf{v} &= (6,4,2),\\ \therefore \mathbf{v}\cdot\mathbf{n} &= (6,4,2)\cdot(0,0,-2) = -4 \implies \text{front facing.}\end{aligned}$$

Face 6:

$$\begin{aligned}\mathbf{n} &= ((6,2,4)-(5,3,4)) \times ((7,3,4)-(6,2,4)) = (1,-1,0)\times(1,1,0) = (0,0,2),\\ \mathbf{v} &= (5,3,4),\\ \therefore \mathbf{v}\cdot\mathbf{n} &= (5,3,4)\cdot(0,0,2) = 8 \implies \text{back facing.}\end{aligned}$$

5.3 Painter's algorithm

The painter's algorithm is another solution to the hidden surface removal problem and is used to remove hidden surfaces in static polygons. It is used more commonly for the rendering of walls that construct a level of a computer game as opposed to dynamic objects that may move due to interactions with the player or objects in the environment. The name derives from the steps used by an oil painter in painting a scene where due to the opacity of the oil paints, the background is painted first, followed by the middle ground with the elements in the foreground painted last figure 5.7.

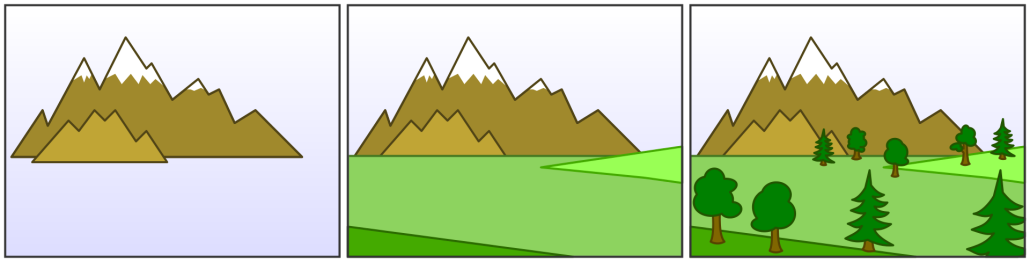


Figure 5.7: The steps that an oil painter uses to paint a scene (Zapyon 2011).

The same principle can be used in computer graphics to remove hidden surfaces. Consider the plan view of the three walls in figure 5.8. Wall A is the furthest from the camera position, wall B is the next nearest

and wall C is the closest to the camera position. Therefore, using the painter's algorithm the walls should be rendered in the order, A, B then C (figure 5.9).

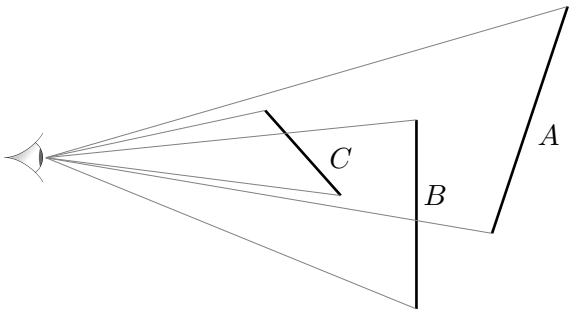


Figure 5.8: Wall A is partially obscured by walls B and C.

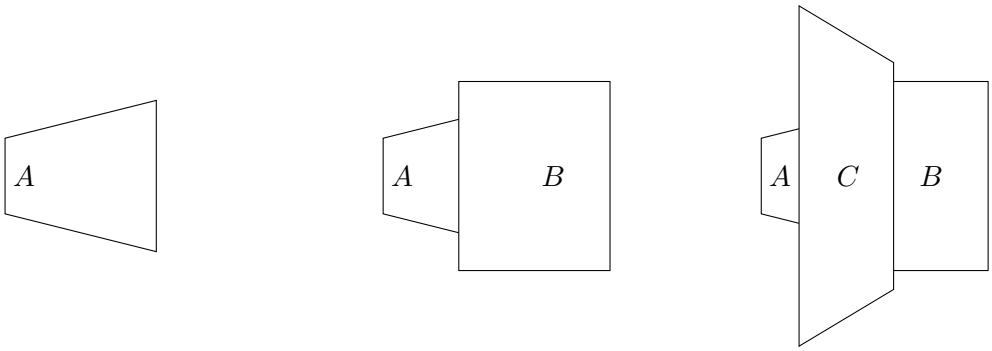


Figure 5.9: The view from the camera position after drawing polygons A (left), B (centre) and C (right).

Painter's algorithm is written in algorithm 10 where the inputs to the function are F which is the face array for the faces in the world space (usually after back-face culling has been applied) and V are the screen space vertex co-ordinates.

Algorithm 10 Painter's algorithm

```
function PaintersAlgorithm(F, V)
    Sort F by in descending order by the z co-ordinate of the closest vertex
    for each face in sorted F do
        Draw face
    end for
end function
```

The results of applying painters algorithm to the front facing polygons seen in the right-hand plot in figure 5.6 is shown in figure 5.10. Note that the objects closer to the view now obscure the objects further away and hidden surfaces have been removed from the scene.

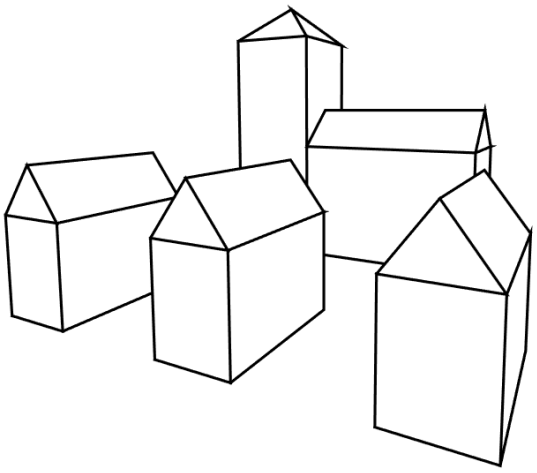


Figure 5.10: The affects of the painter's algorithm applied to the front facing polygons from figure 5.6.

5.3.1 Overlapping polygons

The painters algorithm is a simple and effective method for removing hidden surfaces. However, the method can fail when attempting to render overlapping polygons. Consider the three polygons in figure 5.11. Since each polygon is both in front and behind of another polygon, painter's algorithm would fill in this case. The solution is to split up polygons that overlap in this way.

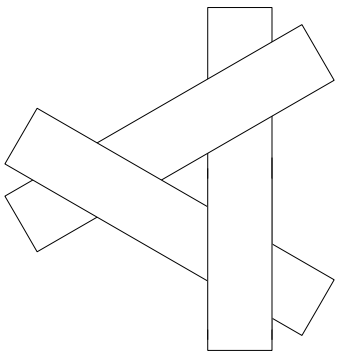


Figure 5.11: Overlapping polygons will cause the painter's algorithm to fail.

5.4 Binary Space Partitioning

Binary Space Partitioning (BSP) is a method that is used to determine the rendering order of polygons that make up static objects of a virtual world (walls, buildings etc.). BSP was first described in the 1960s by Schumaker et al. (1969) to improve the rendering of three-dimensional scenes using computer graphics. However, it wasn't until the early 1990s that BSP became widely used in the computer game industry to improve the performance of rendering three-dimensional scenes. id Software's Doom (Carmack and Romero 1993) and Quake (Abrash and Carmack 1996) are perhaps the most famous games to use BSP but most 3D games since have used the method

Definition 21. A **convex set** is a set of polygons where every polygon is facing every other polygon. A polygon A is said to be facing another polygon B if the surface normal vector is pointing towards B.

Consider the two spaces in figure 5.12. The set of polygons on the left is a convex set since every polygon in the set has its normal vector pointing to every other polygon in the set. The set of polygons on the right is not a convex set since one of the polygons has its normal vector pointing away from the other polygons in the set.

(a) convex set (b) non-convex set

Figure 5.12: The difference between a convex and non-convex set.

Definition 22. A **hyperplane** in n-dimensional space is an $(n-1)$-dimensional object that is used to bisect the space to form two new subspaces.

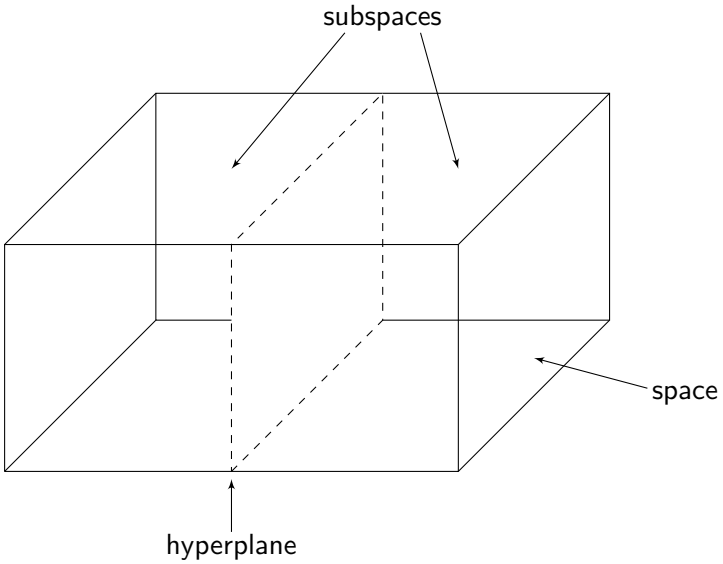


Figure 5.13: A hyperplane divides a space into two subspaces.

Definition 23. An **atomic subspace** is a subspace where all polygons contained within the subspace form a convex set.

The aim of binary space partitioning is to divide the world space into atomic subspaces by inserting hyperplanes along polygons. By definition, the polygons that form each atomic subspace do not obscure each other so can be rendered at the same time. The order that each atomic subspace is rendered is calculated to ensure that hidden surfaces are removed in a similar way to the painters algorithm.

Binary space partitioning is primarily used in the rendering of static polygons in the world space. For example, consider a computer game that requires the player to navigate a three-dimensional virtual environment. The polygons that construct the walls, floors, buildings etc. are known prior to the player navigating the map and remain fixed in place. So we can perform binary space partitioning before the scene needs to be rendered in order to save computational effort whilst the game is being played.

5.4.1 BSP trees

When using binary space partitioning we need to record each subdivision and the polygons that are contained within each subspace. To this a data structure known as a **binary tree** is used. A binary tree consists of **nodes** that are joined by **edges** where each node has a single input edge and at most two output edges (figure 5.14). The node attached to the other end of the input edge is called the **parent node** and the node attached to the other end of the output edges are called **child nodes**. For example in figure 5.14, A is the root node with two child nodes B and C. B is the parent of nodes D and E and C is the parent of nodes F and G. Nodes D, E, F and G have no child nodes so are therefore leaf nodes.

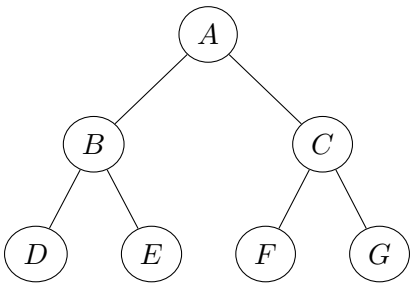


Figure 5.14: A simple binary tree

Consider the space containing the three polygons A, B and C in figure 5.15. The set $\{A, B, C\}$ is not an atomic subspace since none of the three polygons face each other. If we subdivide the space by inserting a polygon on the same plane that polygon A lies on then, since the normal vector for polygon A points to the right, polygon B is in the front subspace and polygon C is in the back subspace. We represent this subdivision as a binary tree with polygon A contained in the root node, polygon B contained in the left child node and polygon C contained within the right child node figure 5.15. The choice of which node to insert the hyperplane along is arbitrary and we can select any polygon in the set, however, there may be optimality considerations (see section 5.4.2).

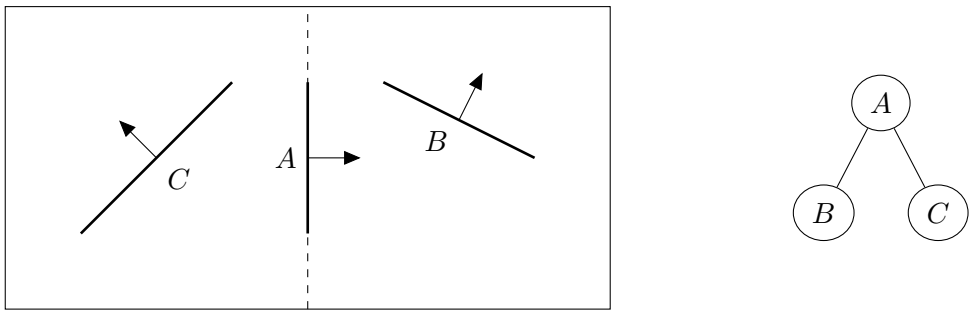


Figure 5.15: Polygons in the front subspace are listed in the left child node and polygons in the back subspace are listed in the right child node in a BSP tree.

If two or more polygons exist on the same plane and they are facing in the same direction then they are

said to be **coincident** and can be treated as one polygon in the BSP process. It is advantageous to use coincident polygons as the ones which a hyperplane is inserted since it reduces the number of polygons we have to deal with. For example, consider figure 5.16 where polygons A and B are coincident. Polygon C is in the front subspace and polygon D is in the back subspace.

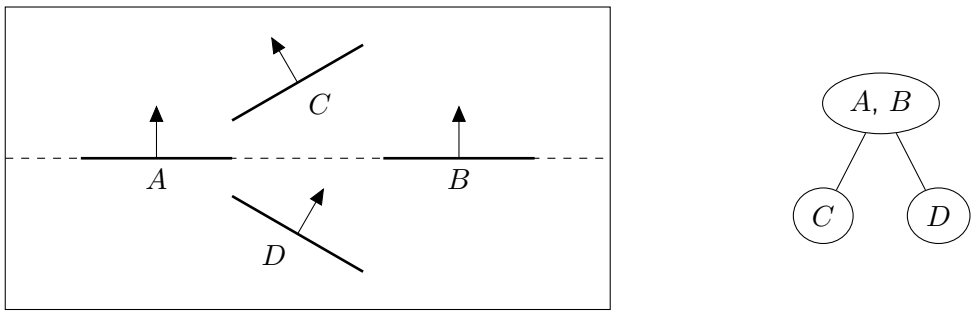


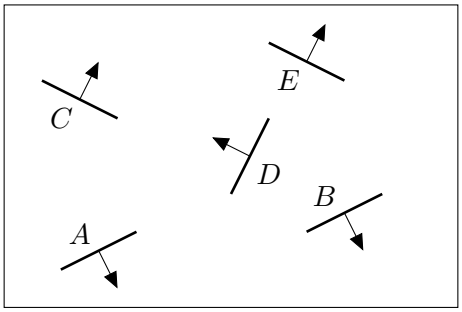
Figure 5.16: Coincident polygons are group into one node in the BSP tree.

This procedure continues until all subspaces contain convex sets, i.e., all subspaces are atomic subspaces. The algorithm for the generation of a BSP tree is given below:

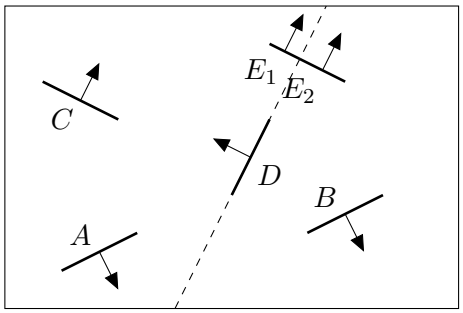
Algorithm 11 BSP-tree generating algorithm

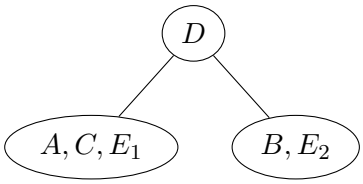
function BSPTREE($polyList$)
  if $polyList$ is a convex set **then**
    Exit function
  else
    Choose a polygon (or set of coincident polygons) from $polyList$ and set as the current node
    List all of the polygons wholly in front of the current node in the left child node.
    List all of the polygons wholly behind the current node in the right child node.
    Split any polygons that are intersected by the hyperplane that the current node lies on and place the two nodes in the appropriate child node.
    BSPTREE($left\ child\ node$)
    BSPTREE($right\ child\ node$)
  end if
end function

Example 15 Construct a BSP tree for the space shown below.

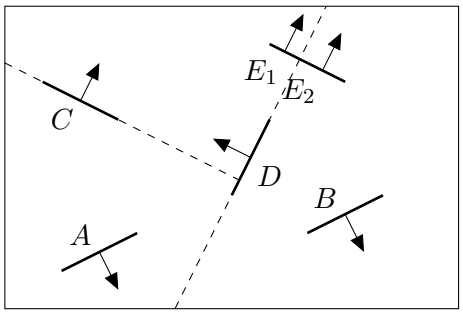


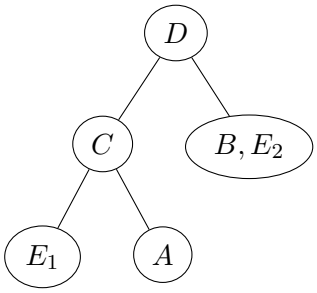
We start with the set of polygons $\{A, B, C, D, E\}$ which is clearly not a convext set. From this set we can select polygon D, place it the root node and insert a hyperplane along it. Polygons A and C are wholly in front of the hyperplane so are listed in the left child node and polygon B is wholly behind the hyperplane so is listed in the right child node. Polygon E intersects the hyperplane and is split into polygons E_1 and E_2. Polygon E_1 is wholly in front of the hyperplane so is listed in the left child node and E_2 is wholly behind the polygon so it is listed in the right child node.



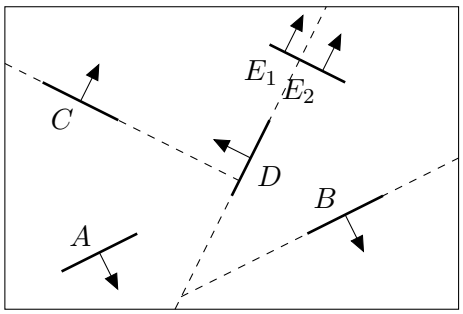


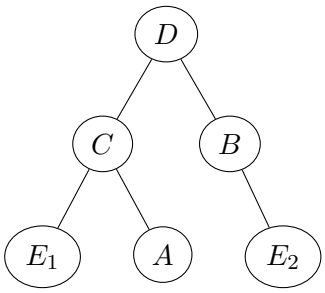
We now look at the left child node and repeat the algorithm for the set $\{A, C, E_1\}$. This is not a convex set so we choose polygon C and insert a hyperplane along it. Polygon A is wholly behind the hyperplane so is listed in the left child node and polygon E_1 is wholly in front of the hyperplane so is listed in the right child node. Since both child nodes only contain one polygon they must both be convex sets.





The next step is to look the right child node of D which contains the set $\{B, E_2\}$. This is not a convex set so we choose polygon B and insert a hyperplane along it. The only other polygon is E_2 which is wholly behind the hyperplane so is listed in the right child node. Since this child node only contains one polygon is it a convex set.



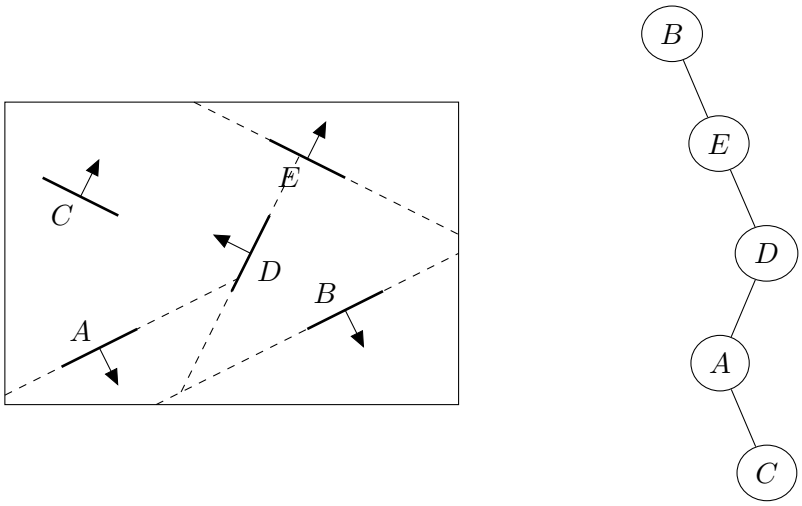


All nodes of the BSP tree contain convex sets so the tree generation algorithm terminates. The final tree now contains the information needed to determine the visibility ordering of the polygons using painter's algorithm (section 5.4.4).

5.4.2 Optimising BSP trees

During the BSP process, the splitting of polygons into two separate polygons causes more work in the long run since more polygons will need to be processes when it comes to clipping, rendering, lighting and textures etc. However, in practical cases splitting is a requirement of the hidden surface removal process. Consider a virtual world where selecting a particular root node will cause each polygon to be split into two. If a different root node was chosen that causes no splitting of polygons, then this BSP tree would contain half the number polygons than the case where each polygon was split. Compound this over a number of steps and the number of polygons in the virtual world increases by a factor of 2^n.

Consider the BSP tree from example 15. In the first step a hyperplane was inserted along polygon D causing polygon E to be split into two. However, if we inserted hyperplanes along polygons B, E, D and A then we can generate a BSP tree where no polygons were split, i.e.,



Clearly an efficient BSP tree will result in the fewest number of splittings possible but how is this done? One method is to check all of the polygons in a space to see how many polygons the hyperplane intersects and then select the polygon that results in the fewest splittings. If there are n polygons in a virtual world then testing each one at every stage of the BSP process will mean checking $n!$ polygons. So consider a very simple virtual world constructed using 20 polygons . Checking every polygon at every stage would mean checking 2.43×10^{18} polygons (2.43 billion billion polygons) (Abrash 2001). If it takes a computer 1 microsecond to check each one, then the BSP tree will take 77,146 years to generate. Obviously it is impractical to check every polygon.

In practice, a random sample is taken of the polygons in a subspace. Each polygon from the sample is checked to see how many polygons intersect with its plane and the polygon with the lowest number of intersections is chosen as the next hyperplane. A study to examine the increase in the number of polygons produced by splitting depending on the number of polygons sampled from the subspace produced the graph shown in figure 5.17.

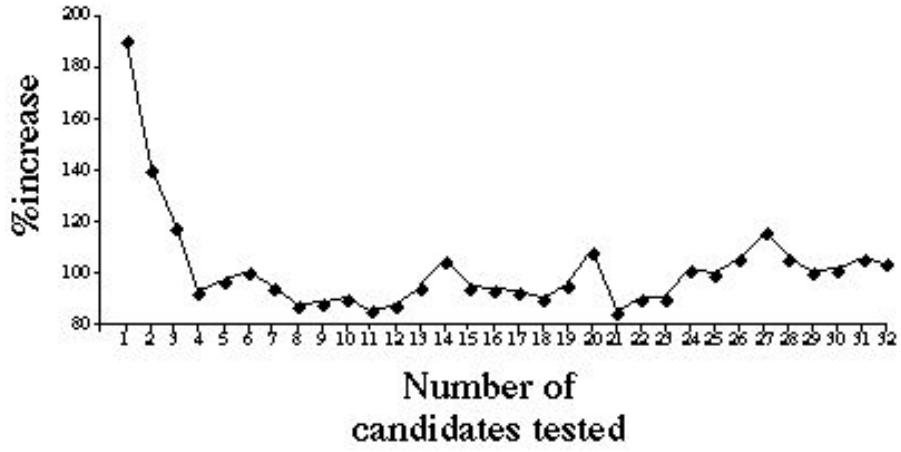


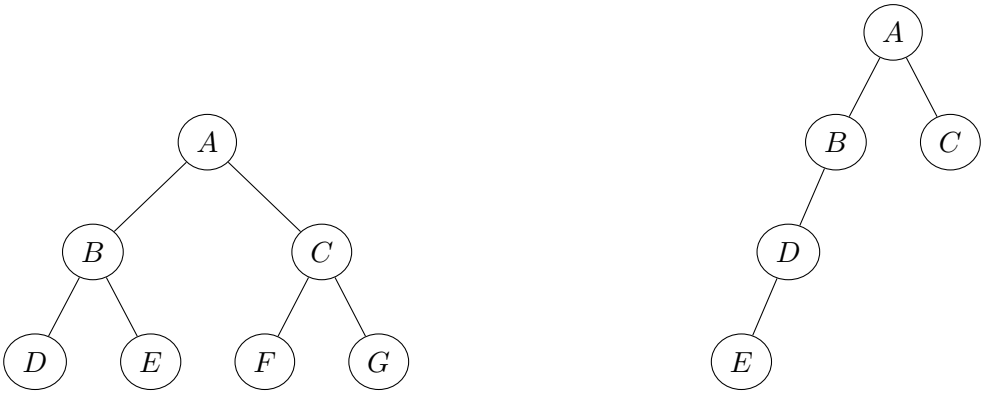
Figure 5.17: The % increase in polygons for different sample sizes.

When a sample of only one polygon was chosen, on average the number of polygons present in the final BSP tree was 190% of the number of original polygons due to splitting. This number drops as the number in the sample is increased. Note that after a sample size of five, the number of polygons produced by splitting does not decrease and therefore we only need to test five polygons in each subspace.

5.4.3 Balanced trees

Another consideration when performing BSP is whether the BSP tree is balanced or unbalanced. A binary tree is said to be **balanced** if it has the same number of nodes down each branch and a tree is said to be

unbalanced if this is not the case.



(a) balanced tree (b) unbalanced tree

Figure 5.18: Balanced and an unbalanced tree.

When it comes to determining the rendering order of polygons, it is quicker to use a balanced tree than an unbalanced tree. To create a balanced tree, the polygon selected for the hyperplane must have a similar number of polygons in front and behind. In creating a balanced tree, the number of splittings that occur may increase. There is usually a tradeoff between the number of splittings that are allowed to occur and how balanced the BSP is.

5.4.4 Visibility ordering using BSP trees

So far we can construct a BSP tree by performing subdivision of a space and splitting the polygons where necessary and optimise it using various criteria. You might be forgiven in thinking how does BSP help when rendering three-dimensional scenes? If you consider what is actually happening with a BSP tree, we are determining which polygons are in front of the others. Therefore BSP trees lend themselves well to a technique similar to the painter's algorithm where we render the furthest polygons from the camera before the nearer ones. The problem with the painter's algorithm is that every time the camera position changes, the distances of each polygon have to be recalculated. Therefore this is not a suitable method to use in computer games where the camera position is commonly controlled by the player and changing often. BSP trees provide the visibility ordering for a scene depending on the camera position relative to the root node of the tree.

Consider the scene shown in figure 5.19 which is viewed from the viewpoint at $\mathbf{p}$. $\mathbf{p}$ is in the back space to the root node D so all of the polygons that are in the front space of D (A, C and E_1) are further away from $\mathbf{p}$ than the polygons in the back space of D (B and E_2). Notice that polygons A, C and E_1 are all contained in the left sub-tree of D and polygons B and E_2 are contained in the right sub-tree. So the rendering order of polygons using the painter's algorithm can be determined using a BSP tree.

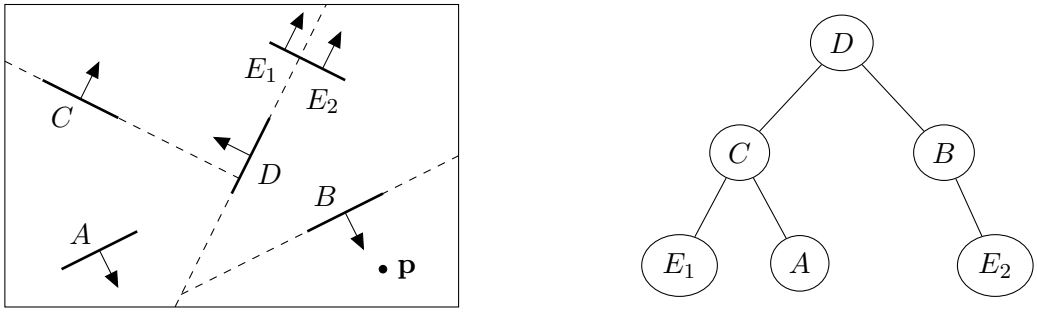


Figure 5.19: The polygons are viewed from the viewpoint at $\mathbf{p}$.

The visibility ordering of polygons in a BSP tree are determined using an **in-order** tree walk presented in

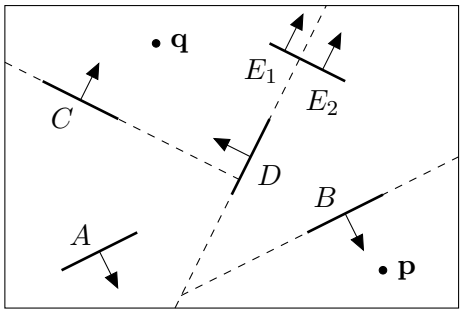
algorithm 12.

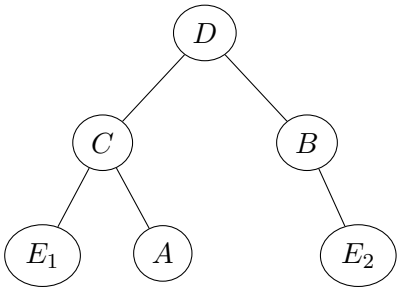
Algorithm 12 BSP-tree traversal

```
function BSPTreeTraversial(node)
    if node is a leaf node then
        Draw the polygons in node
    else if viewing position is in the front subspace of node then
        BSPtreetraversial(right child node)
        Draw the polygons in node
        BSPtreetraversial(left child node)
    else if viewing position is in the back subspace of node then
        BSPtreetraversial(left child node)
        Draw the polygons in node
        BSPtreetraversial(right child node)
    end if
end function
```

The advantage of using BSP trees is that once a tree has been generated for a virtual world it can be used to determine the rendering order for any position of the viewpoint.

Example 16 Using the BSP tree for the polygons given below, determine the order that the polygons are drawn when the viewed from (i) $\mathbf{p}$ and (ii) $\mathbf{q}$





(i) Starting at the root node D.

- D is not a leaf node so we check whether $\mathbf{p}$ is in the front or back subspace of D.
- $\mathbf{p}$ is in the back subspace of D so we move to the left child node C and restart.
- C is not a leaf node and $\mathbf{p}$ is in the back subspace (just) of C so we move to the left child node E_1 and restart.
- E_1 is a leaf node so we render it and return its parent node C
- We render C and move to the right child node A.
- A is a leaf node so we render it and return to the parent node C.
- C has already been rendered so we return to the parent node D.
- We render D and move to the right child node B (because we have already considered the left child node).
- B is not a leaf node and $\mathbf{p}$ is in the front subspace of B so we move to the right child node E_2 and restart.
- E_2 is a leaf node so we render it and return to its parent node B.
- We render B.

- All nodes have been rendered so the tree walk terminates. The rendering order of the polygons which results in all hidden surfaces being removed is

$$\{E_1\}, \{C\}, \{A\}, \{D\}, \{E_2\}, \{B\}.$$

(ii) Starting at the root node D

- D is not a leaf node and $\mathbf{q}$ is in the front subspace of D so we move to the right child node B and restart.
- B is not a leaf node and $\mathbf{q}$ is in the back subspace of B so we attempt to move to the left child node.
- B does not have a left child node so we return to B, render it and then move to the right child node E_2.
- E_2 is a leaf node so we render it and return to the parent node B.
- B has already been rendered so we return to the parent node D.
- We render D and move to the left child node C.
- C is not a leaf node and $\mathbf{q}$ is in the front subspace of C so we move to the right child node A and restart.
- A is a leaf node so we render it and return to the parent node C.
- We render C and move to the left child node E_1.
- E_1 is a leaf node so we redner it.
- All nodes have been rendered so the tree walk terminates. The rendering order of the polygons which results in all hidden surfaces being removed is

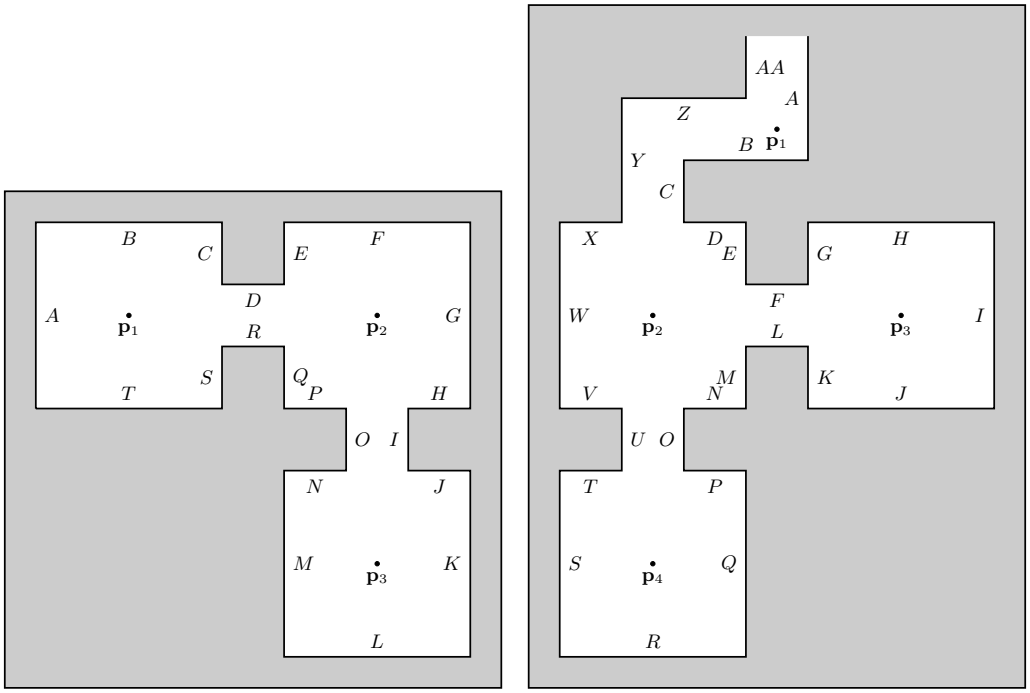
$$\{B\}, \{E_2\}, \{D\}, \{A\}, \{C\}, \{E_1\}.$$

5.5 Exercises

1. A trapezoid object is defined in the view space by the following vertex and face arrays. Determine which faces are front facing and which faces are back facing when viewed from the origin at $(0, 0, 0)$.

$$V = \begin{pmatrix} 10 & 11 & 9 & 10 \\ 4 & 5 & 6 & 5 \\ 2 & 2 & 2 & 3 \\ 1 & 1 & 1 & 1 \end{pmatrix}, \qquad F = \begin{pmatrix} 3 & 2 & 1 \\ 1 & 2 & 4 \\ 2 & 3 & 4 \\ 1 & 4 & 3 \end{pmatrix}$$

2. Two levels of a computer game are to be rendered using BSP trees to help in hidden surface removal. Construct a BSP tree for each level using hyperplanes to subdivide the level until only atomic subspaces remain. You may assume that all polygons are inward facing.



(a) (b)

3. Using your BSP trees from question 2, determine the rendering order of the polygons for each level when viewed from the positions denoted by $\mathbf{p}_i$.

The solutions to these exercises can be found on page 139.

Chapter 6

Clipping

Consider the graphics pipeline shown in figure 6.1. After the world space has been aligned to the viewer position and direction of view the next step is to remove polygons that should not be visible to the viewer, i.e., polygons that are behind the viewer or outside their peripheral vision. Some polygons will lie partially inside and partially outside this visible region and need to split where they intersect with the boundary of the visible region. This process is known as **clipping**.

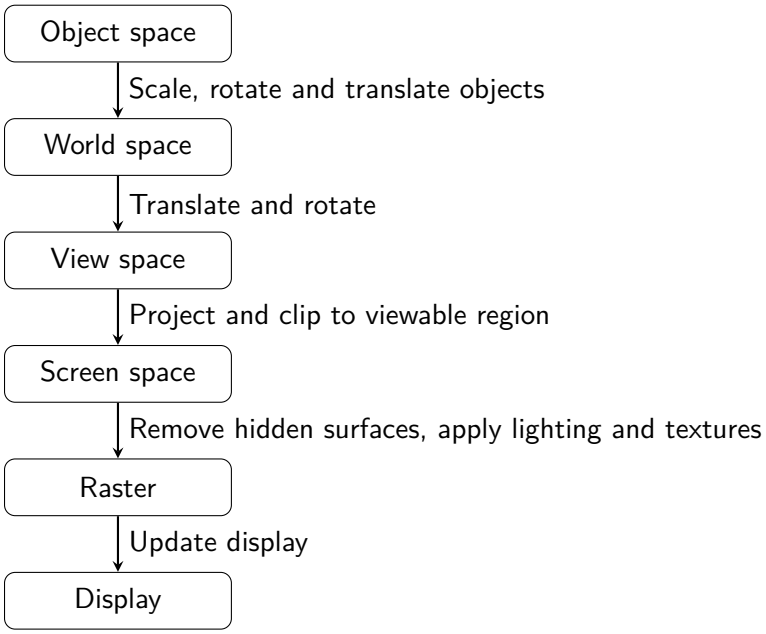


Figure 6.1: The graphics pipeline.

6.1 The viewing frustum

The **viewing frustum** is the region of the view space that should be visible to the viewer. Any polygon that is outside of the viewing frustum is ignored from this stage on-wards in the graphics pipeline. Polygons that lie partially outside of the viewing frustum need to be clipped to the edges of the frustum. If the screen is defined by a rectangular region on the **near viewing plane** (i.e., the projection plane) then the viewing frustum is bounded by the screen, the projection of the screen onto the **far viewing plane** and the four sides of the screen projected between the near and far viewing planes (figure 6.2).

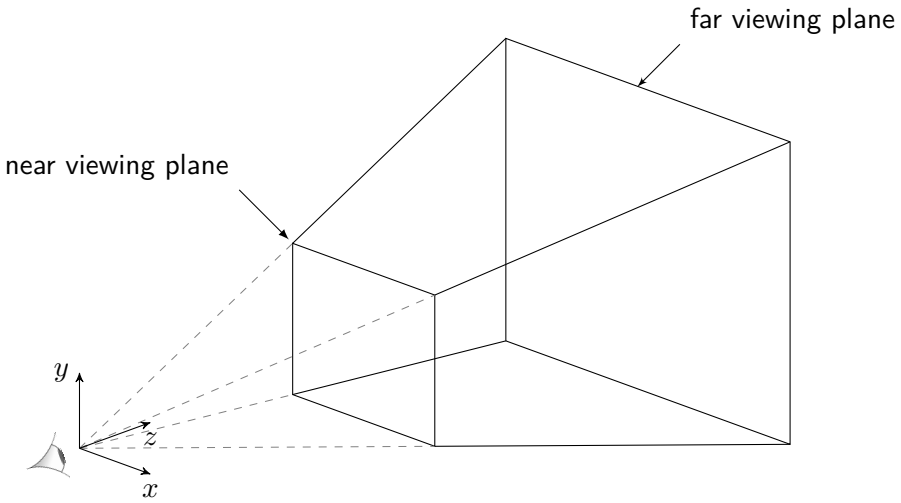
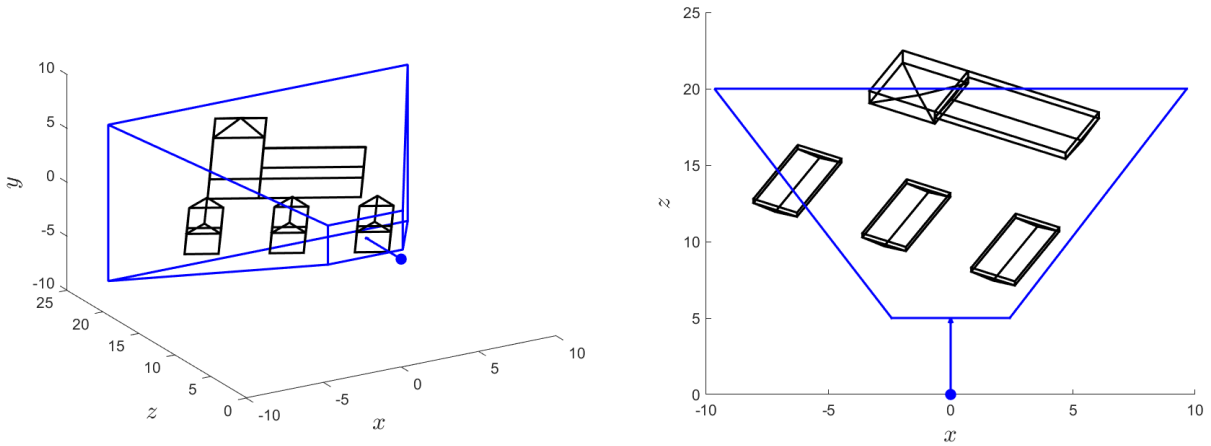


Figure 6.2: The viewing frustum.

To determine the polygons that are behind the viewer we can simply check whether the view space z co-ordinate is negative. For the remaining polygons, we determine which are contained within the viewing frustum, which polygons need to be clipped to the edges and which polygons can be ignored entirely, each polygon is checked against the six planes bounding the viewing frustum. If a polygon lies entirely in front of each of the six planes then that polygon is included in all subsequent calculations. If a polygon lies entirely behind one of the bounding planes then the polygon lies completely outside of the viewing frustum and removed from all subsequent calculations. If a polygon is split by one or more of the bounding planes then the polygon needs to be clipped to those planes.

For example, the view space representation of a virtual world shown in is plotted along with a viewing frustum in figure 6.3. Here the house object on the far left of the viewers field of view needs to be clipped to the left frustum edge and the church object needs to be clipped to the far edge.



(a) view space (b) plan view

Figure 6.3: Plots of the camera space with the viewing frustum shown.

6.1.1 Perspective projection

The aim of perspective projection is to project a three-dimensional space onto a two-dimensional plane such that objects that are close to the plane appear larger than objects further away. Consider figure 6.4 where a point in the view space at (x, y, z) is projected onto the viewing plane parallel to the x and y axes located at distance $z = f$ from the origin. The co-ordinates of the projected point $\mathbf{q} = (x_s, y_s, f)$ are

determined by calculating the intersection between the viewing plane and the projector line that goes from the origin to the point in the view space.

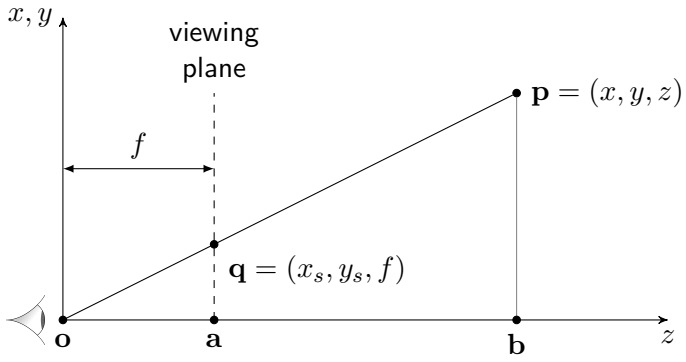


Figure 6.4: A point in the view space P is projected onto the viewing plane at P_s.

The two triangles $\mathbf{oaq}$ and $\mathbf{obp}$ are similar therefore

$$\frac{x_s}{f} = \frac{x}{z},$$
$$\frac{y_s}{f} = \frac{y}{z},$$

so the screen space co-ordinates are

$$x_s = \frac{f}{z}x, \tag{37a}$$
$$y_s = \frac{f}{z}y. \tag{37b}$$

6.1.2 Field of view

The **field of view** (fov) is an angle that determines how much of the view space can be seen on the screen. Consider figure 6.5 that shows a plan view of the view space viewed looking down the y-axis. The x co-ordinates of the left and right hand edges of the screen are denoted by ℓ and r respectively which depend upon the angle fov and the distance of the viewing plane from the viewer (f_{near}). The larger the value of fov the wider the field of view and more of the world space can be seen on screen.

Simple trigonometry is used to calculate the x co-ordinates of the left and right-hand edge of the screen

$$r = f_{near} \tan\left(\frac{fov}{2}\right),$$
$$\ell = -f_{near} \tan\left(\frac{fov}{2}\right) = -r.$$

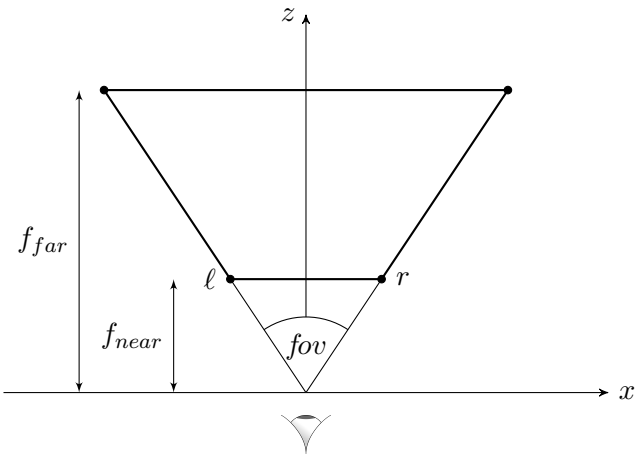


Figure 6.5: The field of view angle determines the width of the viewable area.

6.1.3 Aspect ratio

The fov angle only applies to the horizontal edges of the screen. Consider figure 6.6 that shows the edges of the screen on the screen space viewed down the z-axis. The y co-ordinates of the top and bottom of the screen, denoted here by t and b respectively, are determined using ℓ and r and the width-to-height ratio of the screen known as the **aspect ratio**

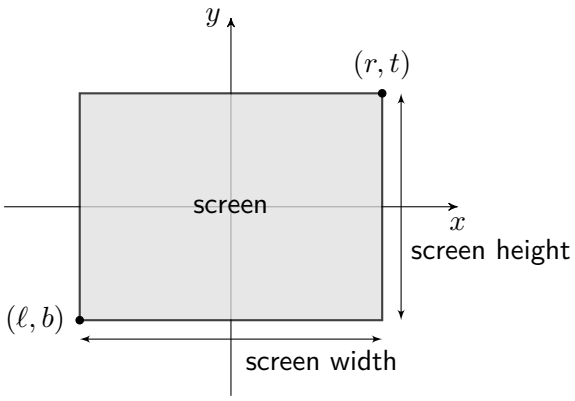


Figure 6.6: The aspect ratio is the ratio of width-to-height of the screen.

$$aspect = \frac{\text{screen width}}{\text{screen height}} = \frac{r - \ell}{t - b}.$$

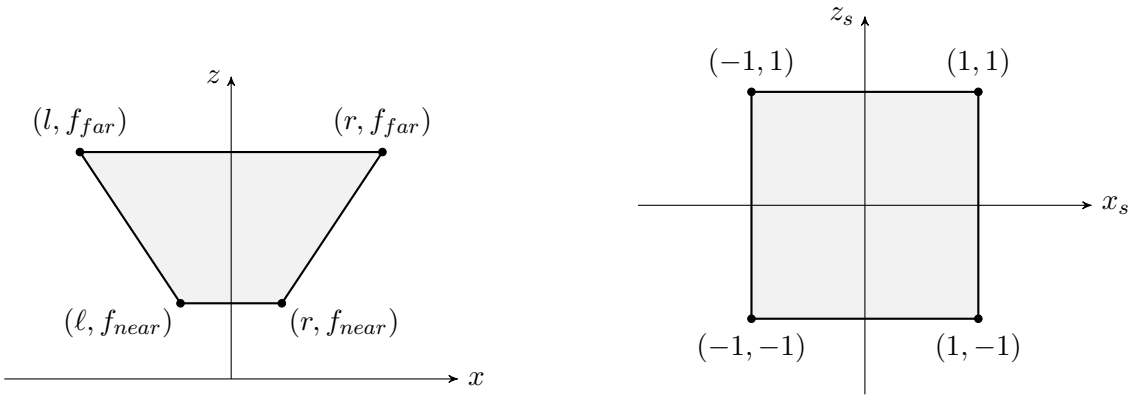
Since $b = -t$ and $\ell = -r$ then

$$t = \frac{r}{aspect}.$$

6.1.4 Transforming the viewing frustum

Determining the intersection between the edges of a polygon and the near and far viewing planes is relatively simple since they are parallel to the xy plane and the normal vectors to these planes will contain only one non-zero element (i.e., for the near viewing plane the normal is $\mathbf{n}_{far} = (0, 0, 1)$ and for the far viewing plane the normal is $\mathbf{n}_{near} = (0, 0, -1)$). For the four remaining bounding planes this will not be the case since these are not parallel to any of the x, y or z axes. In order to simplify the clipping process, a co-ordinate transformation is applied so that the sides of the viewing frustum are parallel to the x, y and z axes.

If $(x_s, y_s, z_s, 1)$ are the screen space co-ordinates corresponding to a point in the view space with co-ordinates $(x, y, z, 1)$. The viewing frustum is transformed so that it is a unit square in the screen space, therefore $x_s, y_s, z_s \in [-1, 1]$ (figure 6.7).



(a) viewing frustum (b) transformed viewing frustum

Figure 6.7: A co-ordinate transformation is applied to the viewing frustum to simplify clipping.

To derive the transformation that maps $x \to x_s$, consider a point within the viewing frustum where $\ell \leq x_s \leq r$. A transformation is needed that maps $\ell \to -1$ and $r \to 1$ and also takes into account perspective projection. Since $\ell = -r$ then $-r \leq x \leq r$ then dividing throughout by r gives

$$-1 \leq \frac{x}{r} \leq 1.$$

Using equation (37a)

$$-1 \leq \frac{f_{near}}{r}\frac{x}{z} \leq 1,$$

therefore if x_s is the perspective projection of x scaled to the screen with edges bounded by ℓ and r then

$$x_s = \frac{f_{near}}{r}\frac{x}{z}, \tag{38}$$

and doing similar for y_s

$$y_s = \frac{f_{near}}{t}\frac{y}{z}. \tag{39}$$

As with other transformations such as translation, scaling and rotation, it is convenient to calculate perspective projection using matrix multiplication. Using equations equations (38) and (39) the matrix that performs perspective projection can be expressed using

$$P = \begin{pmatrix} \frac{f_{near}}{r} & 0 & 0 & 0 \\ 0 & \frac{f_{near}}{t} & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{40}$$

where a and b are constants use to map the z co-ordinate and are determined later. Note that the Cartesian screen space co-ordinates are computed by dividing the projected co-ordinates by the fourth element, i.e.,

$$P\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{f_{near}}{r} & 0 & 0 & 0 \\ 0 & \frac{f_{near}}{t} & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{pmatrix}\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} \frac{f_{near}}{r}x \\ \frac{f_{near}}{t}y \\ az + b \\ z \end{pmatrix}$$

and dividing by the fourth element z gives

$$\begin{pmatrix} \frac{f_{near}}{r}\frac{x}{z} \\ \frac{f_{near}}{t}\frac{y}{z} \\ \frac{az+b}{z} \\ 1 \end{pmatrix}$$

which the first two elements are x_s and y_s from equations equations (38) and (39).

The transformation of the z_s co-ordinate is given in terms of two undetermined values a and b which when written as screen co-ordinates is

$$z_s = \frac{az+b}{z}.$$

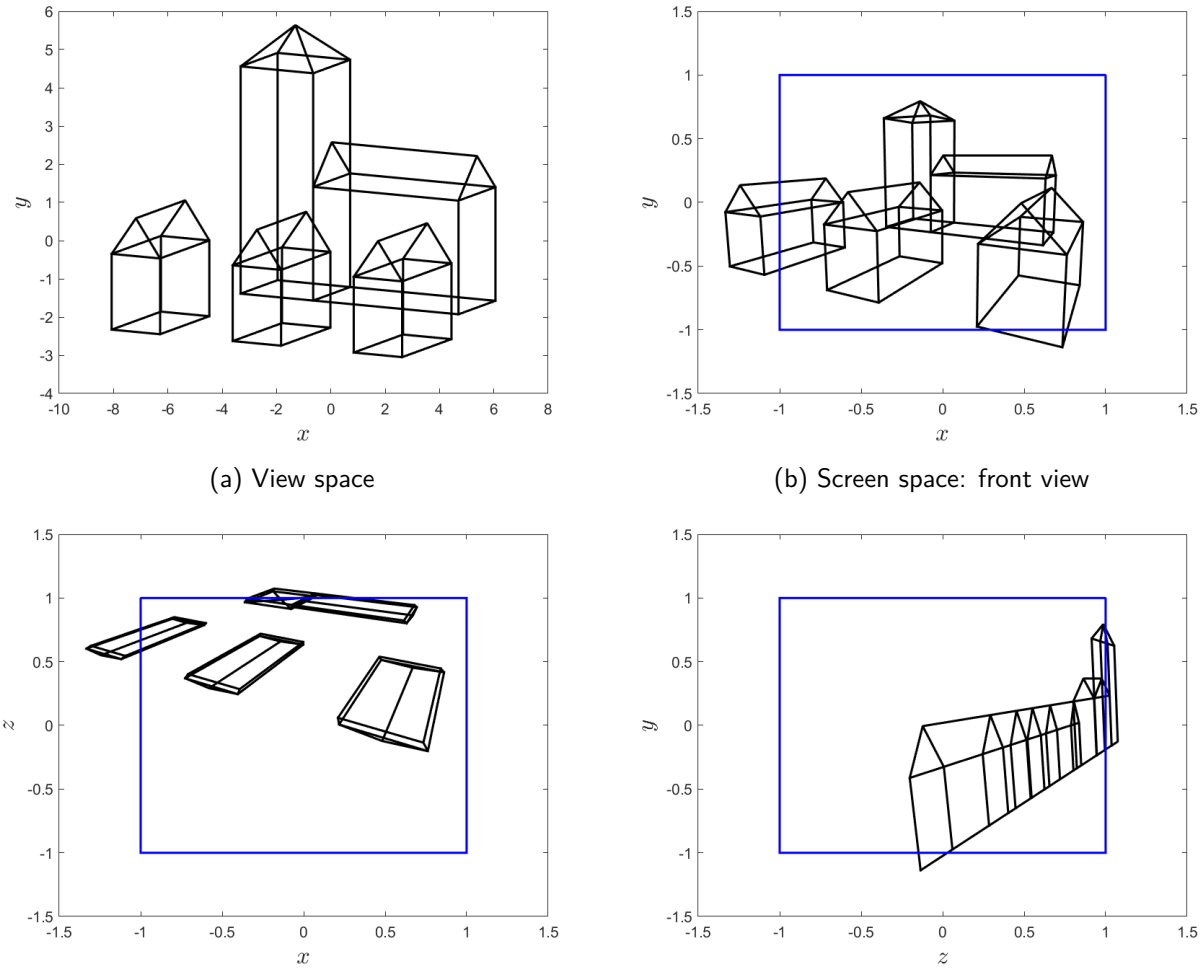
To determine the values of a and b we wish to map $z \mapsto z_s$ so that $z = f_{near} \mapsto -1$ and $z = f_{far} \mapsto 1$. This gives the following system

$$\begin{aligned} \frac{af_{near}+b}{f_{near}} &= -1 \\ \frac{af_{far}+b}{f_{far}} &= 1. \end{aligned}$$

The solution of this system is $a = \frac{f_{far}+f_{near}}{f_{far}-f_{near}}$ and $b = -\frac{2f_{near}f_{far}}{f_{far}-f_{near}}$ therefore equation (40) becomes

$$P = \begin{pmatrix} \frac{f_{near}}{r} & 0 & 0 & 0 \\ 0 & \frac{f_{near}}{t} & 0 & 0 \\ 0 & 0 & \frac{f_{far}+f_{near}}{f_{far}-f_{near}} & -\frac{2f_{near}f_{far}}{f_{far}-f_{near}} \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{41}$$

The matrix P from equation (41) is the complete transformation that combines perspective projection of the world space co-ordinates onto the viewing plane and the transformation of the viewing frustum to the unit square to give the screen space co-ordinates. The affects of applying this transformation can be seen in figure 6.8. Note how that some of the objects lie partially outside of the unit cube that represents the visible region, these objects will need to be clipped .



(a) View space

(b) Screen space: front view

(c) Screen space: plan view

(d) Screen space: side view

Figure 6.8: Plots of the screen space with the visible region represented by a unit cube.

Example 17 A cube object is defined by the following view space co-ordinate array V_{view} and a face array F

$$V_{view} = \begin{pmatrix} -2 & 2 & 2 & -2 & -2 & 2 & 2 & -2 \\ -2 & -2 & -2 & -2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 8 & 8 & 4 & 4 & 8 & 8 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}, \qquad F = \begin{pmatrix} 1 & 2 & 6 & 5 \\ 2 & 3 & 7 & 6 \\ 3 & 4 & 8 & 7 \\ 4 & 1 & 5 & 8 \\ 4 & 3 & 2 & 1 \\ 5 & 6 & 7 & 8 \end{pmatrix}.$$

The view space is to be projected onto the screen space with a field of view angle $fov = 1.5$, a width-to-height screen aspect ratio of $4:3$ and near and far viewing planes located at $z = 2$ and $z = 10$ respectively. Calculate the screen space co-ordinates of the object such that the viewing frustum is bounded by a unit cube centred at the origin.

The co-ordinates for the top-right hand corner of the screen on the near projection plane are

$$r = f_{near} \tan\left(\frac{fov}{2}\right) = 2\tan\left(\frac{1.5}{2}\right) = 1.8632,$$
$$t = \frac{r}{aspect} = \frac{1.8632}{4/3} = 1.3974,$$

so the projection matrix is

$$P = \begin{pmatrix} \frac{f_{near}}{r} & 0 & 0 & 0 \\ 0 & \frac{f_{near}}{t} & 0 & 0 \\ 0 & 0 & \frac{f_{far}+f_{near}}{f_{far}-f_{near}} & -\frac{2f_{near}f_{far}}{f_{far}-f_{near}} \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} \frac{2}{1.8632} & 0 & 0 & 0 \\ 0 & \frac{2}{1.3974} & 0 & 0 \\ 0 & 0 & \frac{10+2}{10-2} & -\frac{2(2)(10)}{10-2} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$
$$= \begin{pmatrix} 1.0734 & 0 & 0 & 0 \\ 0 & 1.4312 & 0 & 0 \\ 0 & 0 & 1.5 & -5 \\ 0 & 0 & 1 & 0 \end{pmatrix}.$$

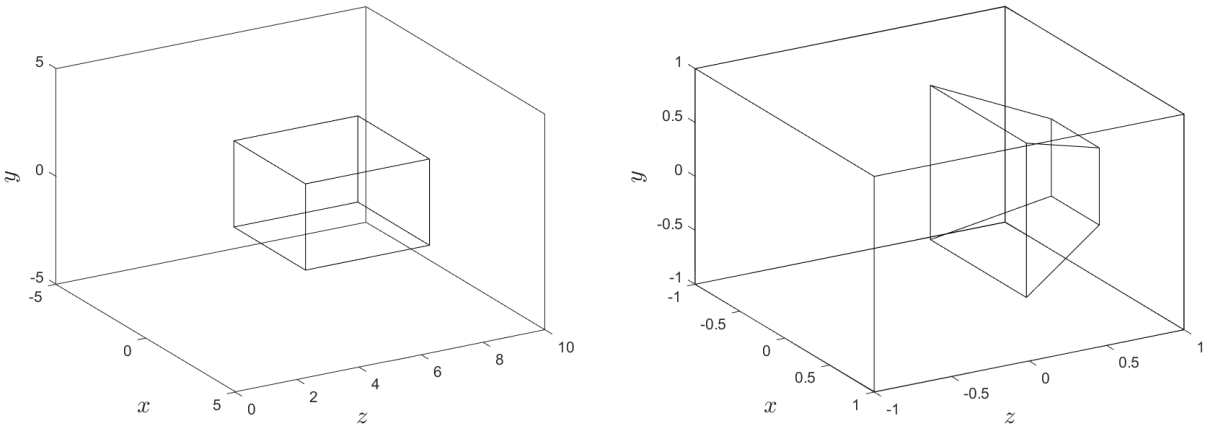
Applying the projection matrix to V_{view} gives

$$P \cdot V_{view} = \begin{pmatrix} 1.0734 & 0 & 0 & 0 \\ 0 & 1.4312 & 0 & 0 \\ 0 & 0 & 1.5 & -5 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} -2 & 2 & 2 & -2 & -2 & 2 & 2 & -2 \\ -2 & -2 & -2 & -2 & 2 & 2 & 2 & 2 \\ 4 & 4 & 8 & 8 & 4 & 4 & 8 & 8 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$
$$= \begin{pmatrix} -2.1469 & 2.1469 & 2.1469 & -2.1469 & -2.1469 & 2.1469 & 2.1469 & -2.1469 \\ -2.8625 & -2.8625 & -2.8625 & -2.8625 & 2.8625 & 2.8625 & 2.8625 & 2.8625 \\ 1 & 1 & 7 & 7 & 1 & 1 & 7 & 7 \\ 4 & 4 & 8 & 8 & 4 & 4 & 8 & 8 \end{pmatrix}.$$

Divide by the fourth element in each co-ordinate to get the Cartesian screen space co-ordinates

$$V_{screen} = \begin{pmatrix} -0.5367 & 0.5367 & 0.2684 & -0.2684 & -0.5367 & 0.5367 & 0.2684 & -0.2684 \\ -0.7156 & -0.7156 & -0.3578 & -0.3578 & 0.7156 & 0.7156 & 0.3578 & 0.3578 \\ 0.2500 & 0.2500 & 0.8750 & 0.8750 & 0.2500 & 0.2500 & 0.8750 & 0.8750 \\ 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

The affectos of the projection of the view space onto the screen space can be seen in figure 6.9. Note how the face of the cube furthest from the viewer is smaller than the face that is closest.



(a) view space

(b) screen space

Figure 6.9

6.2 Line clipping

For a given area, known as the **clip region**, defined as the region that should be visible to the computer user, we wish to draw all lines that lie within this region, clip any lines that lie partially outside of this clip region to the edges of the clip region and ignore any lines that lie completely outside of the clip region.

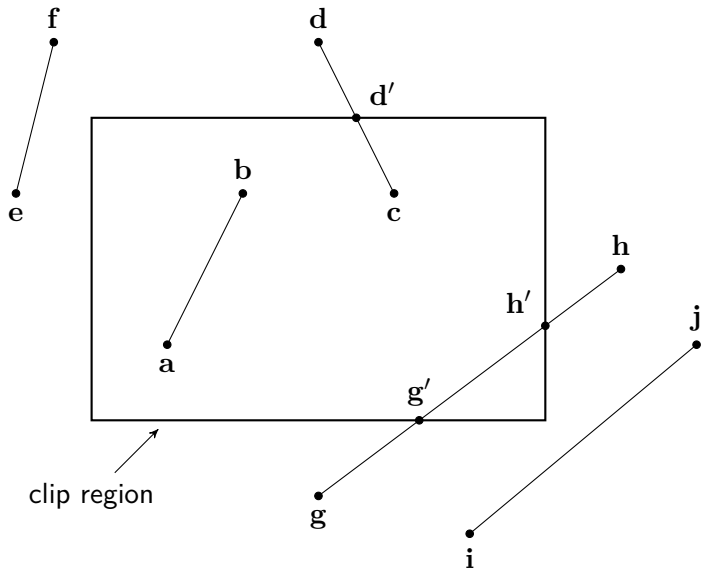


Figure 6.10: Lines are clipped to the clip region.

Consider the diagram shown in figure 6.10. Here the clip region is represented by a rectangle. A suitable line clipping algorithm should identify the following:

- line $\mathbf{a} \rightarrow \mathbf{b}$ should be drawn as it lies completely within the clip region;
- line $\mathbf{c} \rightarrow \mathbf{d}$ lies partially outside of the clip region so $\mathbf{d}$, the point outside of the clip region, should be clipped to the top edge at $\mathbf{d}'$ and line $\mathbf{c} \rightarrow \mathbf{d}'$ is drawn;
- line $\mathbf{e} \rightarrow \mathbf{f}$ lies completely outside of the clip region so is ignored;
- line $\mathbf{g} \rightarrow \mathbf{h}$ lies partially outside of the clip region with both endpoints outside, so $\mathbf{g}$ is clipped to $\mathbf{g}'$ and $\mathbf{h}$ is clipped to $\mathbf{h}'$ and line $\mathbf{g}' \rightarrow \mathbf{h}'$ is drawn;
- line $\mathbf{i} \rightarrow \mathbf{j}$ lies completely outside of the clip region and is ignored.

6.3 The Cyrus-Beck algorithm

The **Cyrus-Beck algorithm** (Cyrus and Beck 1978) uses the intersection between a line and a plane to determine whether a line requires clipping and for calculating the co-ordinates of the point on where the line should be clipped to. The perpendicular distance of the start and end points of a line are calculated and the sign of these distances determine whether the line is in front, crossing or behind an edge of the clip region.

6.3.1 Calculating the perpendicular distance between a point and a plane

A plane is defined by its normal vector $\mathbf{n}$ and a point on the plane $\mathbf{p}$. Let $\mathbf{q}$ be an arbitrary point in space and d be the perpendicular distance from $\mathbf{q}$ to the plane (figure 6.11).

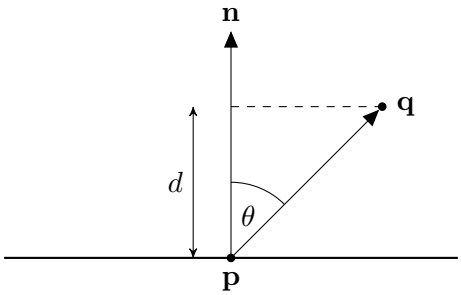


Figure 6.11: The distance between a point and a plane.

The geometric definition of a dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$ is:

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\| \cos(\theta)$$

where θ is the angle between $\mathbf{a}$ and $\mathbf{b}$. So the dot product between the normal vector $\mathbf{n}$ and the vector joining $\mathbf{p}$ to $\mathbf{q}$ is

$$(\mathbf{q} - \mathbf{p}) \cdot \mathbf{n} = \|\mathbf{q} - \mathbf{p}\|\|\mathbf{n}\| \cos(\theta). \tag{42}$$

Forming a right-angled triangle where $\|\mathbf{q} - \mathbf{p}\|$ is the length of the hypotenuse and d is the length of the adjacent side then

$$\cos(\theta) = \frac{\text{adjacent}}{\text{hypotenuse}} = \frac{d}{\|\mathbf{q} - \mathbf{p}\|}.$$

Substituting this expression into equation (42) gives

$$(\mathbf{q} - \mathbf{p}) \cdot \mathbf{n} = \|\mathbf{q} - \mathbf{p}\|\|\mathbf{n}\| \frac{d}{\|\mathbf{q} - \mathbf{p}\|},$$

therefore

$$d = (\mathbf{q} - \mathbf{p}) \cdot \frac{\mathbf{n}}{\|\mathbf{n}\|}. \tag{43}$$

This expression is simplified further if $\hat{\mathbf{n}}$ is a unit vector then

$$d = (\mathbf{q} - \mathbf{p}) \cdot \hat{\mathbf{n}}.$$

6.3.2 Determining when clipping is required

The sign of the perpendicular distance, d, from an end point of a line $\mathbf{a}$ to an edge of the clip region can be used to determine whether a line requires clipping or not:

$$d_{\mathbf{a}} \begin{cases} < 0 & \mathbf{a} \text{ is behind the edge,} \\ = 0 & \mathbf{a} \text{ is on the edge,} \\ > 0 & \mathbf{a} \text{ is in front of the edge.} \end{cases} \tag{44}$$

Let $\mathbf{a}$ and $\mathbf{b}$ be the start and end points of a line. There are four possible cases that need to be considered depending on the sign of $d_{\mathbf{a}}$ and $d_{\mathbf{b}}$ that are summarised in table 6.1.

Table 6.1: The signs of $d_{\mathbf{a}}$ and $d_{\mathbf{b}}$ determine whether clipping is required.

$d_{\mathbf{a}}$	$d_{\mathbf{b}}$	$\mathbf{a}$	$\mathbf{b}$	Action
< 0	< 0	behind	behind	do not draw $\mathbf{a} \rightarrow \mathbf{b}$
< 0	≥ 0	behind	in front	clip $\mathbf{a}$ to $\mathbf{a}'$
≥ 0	< 0	in front	behind	clip $\mathbf{b}$ to $\mathbf{b}'$
≥ 0	≥ 0	in front	in front	do nothing

6.3.3 Calculating the intersection between a line and an edge of the clip region

Consider the diagram in figure 6.12 that shows a line $\mathbf{a} \to \mathbf{b}$ that is clipped to a plane defined by the point $\mathbf{p}$ and normal vector $\mathbf{n}$.

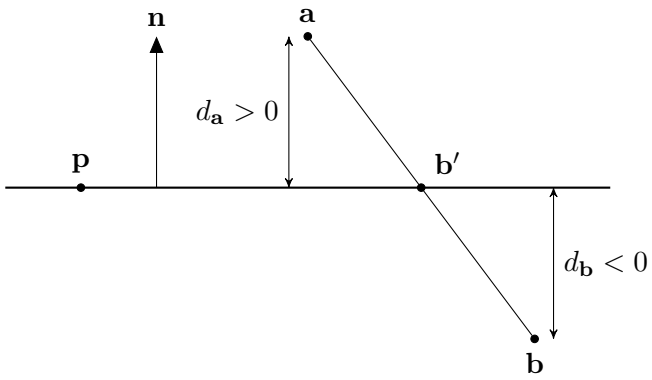


Figure 6.12: The line $\mathbf{a} \to \mathbf{b}$ is clipped to the edge at $\mathbf{b}'$.

The clipped point $\mathbf{b}'$ that lies on the edge has a distance $d_{\mathbf{b}'} = 0$, therefore using equation (43)

$$0 = (\mathbf{b}' - \mathbf{p}) \cdot \hat{\mathbf{n}}. \tag{45}$$

The equation of the line joining $\mathbf{a}$ and $\mathbf{b}$ is

$$\mathbf{b}' = \mathbf{a} + t(\mathbf{b} - \mathbf{a}) \tag{46}$$

where $t \in [0, 1]$. Substituting equation (46) into equation (45) and rearranging gives

$$\begin{aligned} 0 &= (\mathbf{a} + t(\mathbf{b} - \mathbf{a}) - \mathbf{p}] \cdot \hat{\mathbf{n}} \\ &= (\mathbf{a} - \mathbf{p}) \cdot \hat{\mathbf{n}} + t(\mathbf{b} - \mathbf{a}) \cdot \hat{\mathbf{n}} \\ t(\mathbf{a} - \mathbf{b}) \cdot \hat{\mathbf{n}} &= (\mathbf{a} - \mathbf{p}) \cdot \hat{\mathbf{n}} \\ t &= \frac{(\mathbf{a} - \mathbf{p}) \cdot \hat{\mathbf{n}}}{(\mathbf{a} - \mathbf{b}) \cdot \hat{\mathbf{n}}}, \end{aligned}$$

which can be written in terms of $d_\mathbf{a}$ and $d_\mathbf{b}$ as

$$t = \frac{d_\mathbf{a}}{d_\mathbf{a} - d_\mathbf{b}}. \tag{47}$$

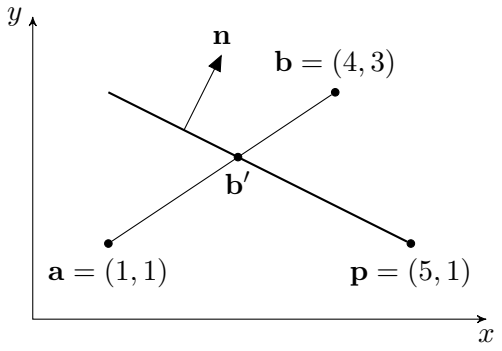
Equation (47) gives the value of t that can be substituted into equation (46) to give the intersection point $\mathbf{c}$.

The Cyrus-Beck algorithm for clipping a line to a window edge is given in algorithm 13. The function uses inputs of the co-ordinates of the endpoints of the line $\mathbf{a}$ and $\mathbf{b}$, the normal vector to the window edge $\mathbf{n}$ and the co-ordinates of a point on the window edge $\mathbf{p}$. This function is applied to each widow edge.

Algorithm 13 The Cyrus-Beck algorithm

function LINECLIPPING($\mathbf{a}$, $\mathbf{b}$, $\mathbf{n}$, $\mathbf{p}$)
  Initialise $d_\mathbf{a} \leftarrow (\mathbf{a} - \mathbf{p}) \cdot \mathbf{n}$, $d_\mathbf{b} \leftarrow (\mathbf{b} - \mathbf{p}) \cdot \mathbf{n}$ and $t \leftarrow \dfrac{d_\mathbf{a}}{d_\mathbf{a} - d_\mathbf{b}}$
  if $d_\mathbf{a} < 0 \wedge d_\mathbf{b} < 0$ **then** ▷ Both $\mathbf{a}$ and $\mathbf{b}$ are behind edge
    $\mathbf{b} \leftarrow \mathbf{a}$
  else if $d_\mathbf{a} > 0 \wedge d_\mathbf{b} < 0$ **then** ▷ $\mathbf{b}$ needs clipping to edge
    $\mathbf{b} \leftarrow \mathbf{a} + t(\mathbf{b} - \mathbf{a})$
  else if $d_\mathbf{a} < 0 \wedge d_\mathbf{b} > 0$ **then** ▷ $\mathbf{a}$ needs clipping to edge
    $\mathbf{a} \leftarrow \mathbf{a} + t(\mathbf{b} - \mathbf{a})$
  end if
  return $\mathbf{a}$ and $\mathbf{b}$
end function

Example 18 A line joining the points $\mathbf{a} = (1, 1)$ and $\mathbf{b} = (4, 3)$ is to be clipped to an edge of a clip region defined by the normal vector $\mathbf{n} = (2, 4)$ and the point $\mathbf{p}_0 = (5, 1)$.



The distances of $\mathbf{a}$ and $\mathbf{b}$ from the edge are

$$d_{\mathbf{a}} = (\mathbf{a} - \mathbf{p}) \cdot \mathbf{n} = ((1, 1) - (5, 1)) \cdot (2, 4) = -8,$$
$$d_{\mathbf{b}} = (\mathbf{b} - \mathbf{p}) \cdot \mathbf{n} = ((4, 3) - (5, 1)) \cdot (2, 4) = 6.$$

So $\mathbf{a}$ is behind the edge and $\mathbf{b}$ is in front. Clipping $\mathbf{a}$ to the edge at $\mathbf{b}'$

$$t = \frac{d_{\mathbf{a}}}{d_{\mathbf{a}} - d_{\mathbf{b}}} = \frac{-8}{-8 - 6} = \frac{4}{7},$$
$$\therefore \mathbf{b}' = \mathbf{a} + t(\mathbf{b} - \mathbf{a}) = (1, 1) + \frac{4}{7}((4, 3) - (1, 1))$$
$$= (1, 1) + \frac{4}{7}(3, 2)$$
$$= \frac{1}{7}(19, 15) = (2.714, 2.143).$$

6.4 The Sutherland-Hodgman algorithm

The **Sutherland-Hodgman algorithm** (Sutherland and Hodgman 1974) is used to clip a polygon to a clip region as opposed to line clipping seen previously with the Cyrus-Beck algorithm. The polygon to be clipped is known as the **subject polygon** and the polygon that represents the clip region is known as the **clip polygon**. The clip polygon can be any simple polygon so the Sutherland-Hodgman algorithm is useful for clipping against complicated shapes.

Consider a subject polygon with n edges defined by the **vertex list**:

$$List = \{\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n\}.$$

where $\mathbf{v}_i$ are the co-ordinates of vertex i. The edges of the subject polygon are checked to see whether they intersect an edge of the clip polygon by determining whether the endpoints are in front or behind the edge (using equation (43)). If the polygon edge does intersect then the point of intersection is calculated and replaces the endpoint that is behind the edge of the clip polygon in the vertex list. If both endpoints are behind the edge of the clip polygon then these are removed from the vertex list. Once this has been done for all edges of the clip polygon the final vertex list will define the clipped polygon and can be drawn.

The Sutherland-Hodgman algorithm is shown in algorithm 14.

Algorithm 14 The Sutherland-Hodgman algorithm

```
function PolygonClipping(subjectPolygon, clipPolygon)
    newList ← subjectPolygon
    for each edge in clipPolygon do
        List ← newList
        clear newList
        for each vertex v_i in List do
            if v_i is in front of edge then
                add v_i to newList
                if v_{i+1} is behind edge then
                    i ← LineClipping(v_i, v_{i+1}, n_edge, p_edge)
                    add i to newList
                end if
            else
                if v_{i+1} is in front of edge then
                    i ← LineClipping(v_i, v_{i+1}, n_edge, p_edge)
                    add i to newList
                end if
            end if
        end for
    end for
    return newList
end function
```

Example 19 Consider the clipping of the polygon to the clip region shown in figure 6.13.

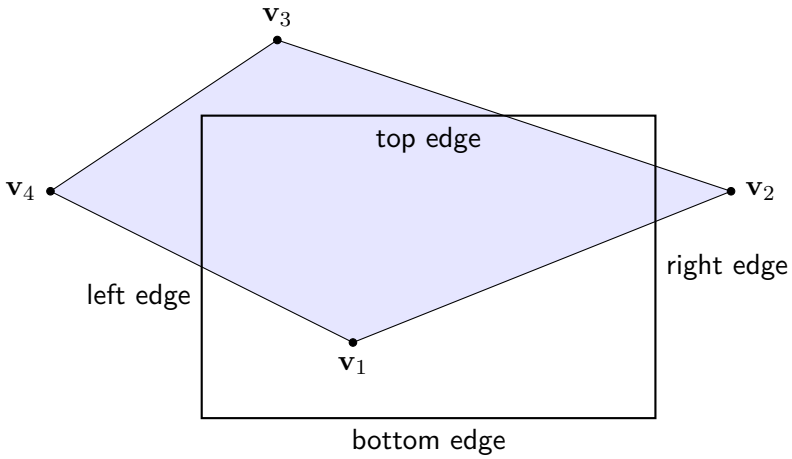


Figure 6.13: Clipping a polygon to a clip region using the Sutherland-Hodgman algorithm.

The vertex list of the subject polygon is

$$List = \{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4\}.$$

Clip to bottom edge:

- All vertices are in front so $newList = \{\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4\}$

Clip to right edge:

- Set $List = newList$ and clear $newList$
- $\mathbf{v}_1$ is in front so add to $newList$
- $\mathbf{v}_2$ is behind so clip $\mathbf{v}_1 \to \mathbf{v}_2$ to $\mathbf{v}_1 \to \mathbf{i}_1$ and add $\mathbf{i}_1$ to $newList$
- $\mathbf{v}_2$ is behind and $\mathbf{v}_3$ is in front so clip $\mathbf{v}_1 \to \mathbf{v}_3$ to $\mathbf{i}_2 \to \mathbf{v}_3$ and add $\mathbf{i}_2$ to $newList$
- $\mathbf{v}_3$ and $\mathbf{v}_4$ are in front so add to $newList$

$$newList = \{\mathbf{v}_1, \mathbf{i}_1, \mathbf{i}_2, \mathbf{v}_3, \mathbf{v}_4\}$$

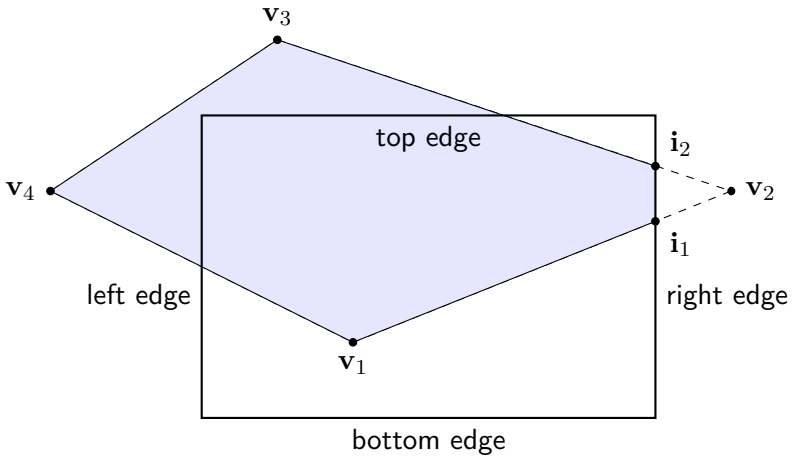


Figure 6.14: Clipping to right edge.

Clip to top edge:

- Set $List = newList$ and clear $newList$
- $\mathbf{v}_1$, $\mathbf{i}_1$ and $\mathbf{i}_2$ are in front so add to $newList$
- $\mathbf{v}_3$ is behind so clip $\mathbf{v}_2 \rightarrow \mathbf{v}_3$ to $\mathbf{v}_2 \rightarrow \mathbf{i}_3$ and add $\mathbf{i}_3$ to $newList$
- $\mathbf{v}_3$ is behind and $\mathbf{v}_4$ is in front so clip $\mathbf{v}_3 \rightarrow \mathbf{v}_4$ to $\mathbf{i}_4 \rightarrow \mathbf{v}_4$ and add to $\mathbf{i}_4$ to $newList$
- $\mathbf{v}_4$ is in front so add to $newList$

$$newList = \{\mathbf{v}_1, \mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_3, \mathbf{i}_4, \mathbf{v}_4\}$$

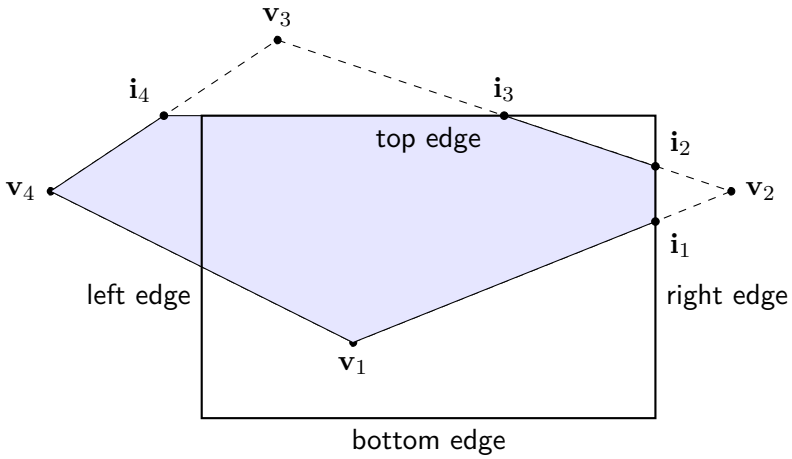


Figure 6.15: Clipping to top edge.

Clip to left edge:

- Set $List = newList$ and clear $newList$
- $\mathbf{v}_1$, $\mathbf{i}_1$, $\mathbf{i}_2$ and $\mathbf{i}_3$ are in front so add to $newList$
- $\mathbf{i}_4$ is behind so clip $\mathbf{i}_3 \rightarrow \mathbf{i}_4$ to $\mathbf{i}_3 \rightarrow \mathbf{i}_5$ and add $\mathbf{i}_5$ to $newList$
- $\mathbf{v}_1$ is in front so clip $\mathbf{v}_4 \rightarrow \mathbf{v}_1$ to $\mathbf{i}_6 \rightarrow \mathbf{v}_1$ and add $\mathbf{i}_6$ to $newList$

$$newList = \{\mathbf{v}_1, \mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_3, \mathbf{i}_5, \mathbf{i}_6\}$$

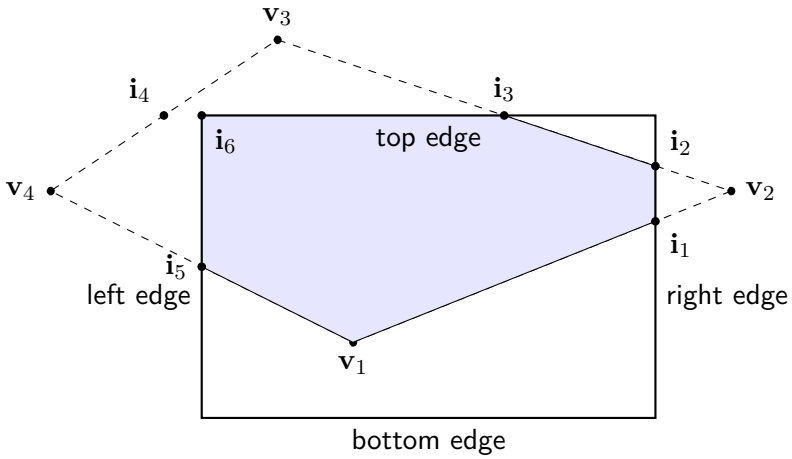
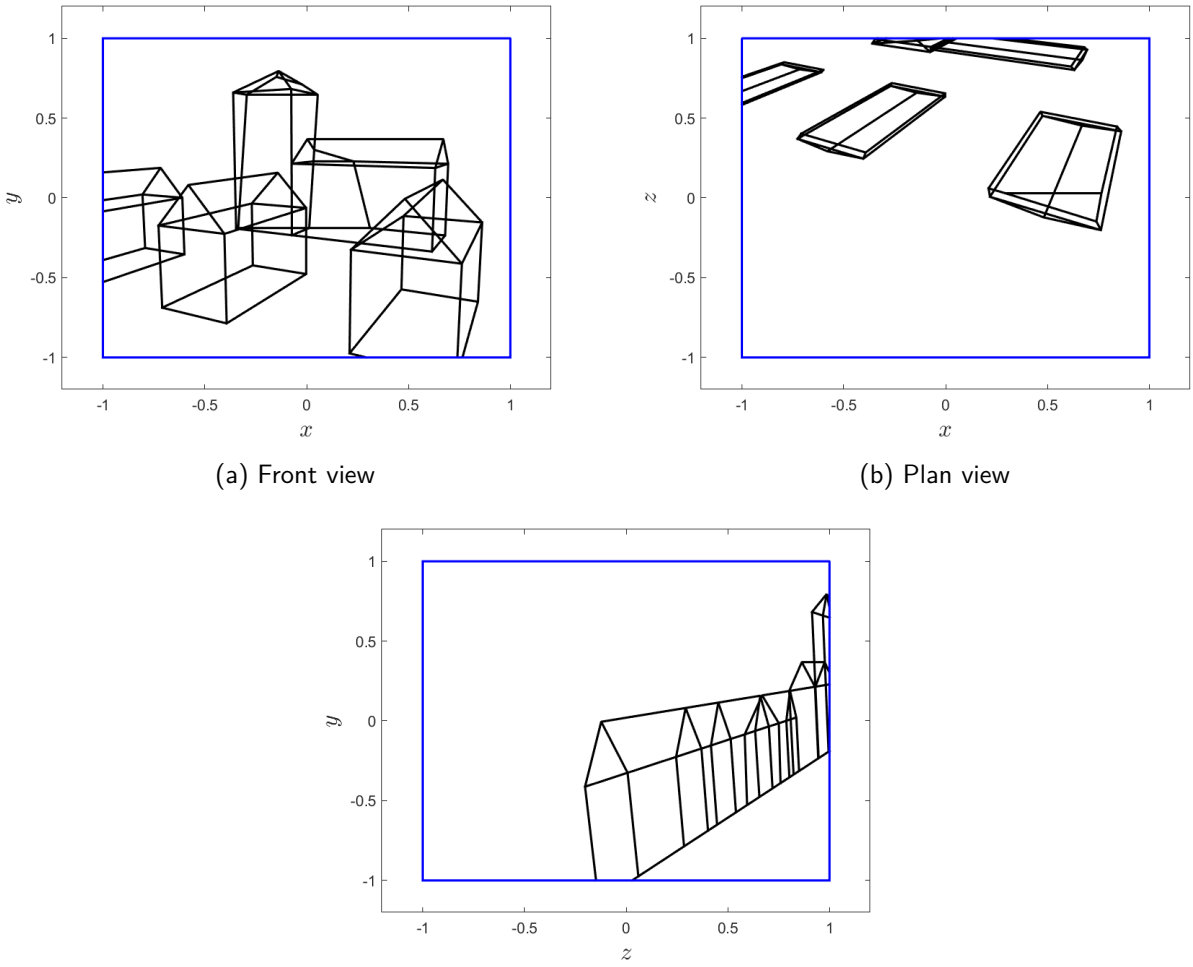


Figure 6.16: Final clipped polygon.

The affect of apply the Sutherland-Hodgman algorithm to clip the screen space can be seen in figure 6.17



(a) Front view

(b) Plan view

(c) Side view

Figure 6.17: Plots of the screen space clipped to the viewable region using the Sutherland-Hodgman algorithm.

6.5 The Cohen-Sutherland algorithm

The **Cohen-Sutherland algorithm** uses logical expressions to determine which endpoints of a line need to be clipped to which edge of the clip region. The Cohen-Sutherland algorithm can only be applied to clip regions where the edges are parallel to the horizontal and vertical axes and is often used for windowed Graphical User Interfaces (GUIs). However, it has many advantages over other clipping methods: trivial cases that make up the majority of line segments in a scene are accepted or rejected with minimal calculation and is simple to code and computationally efficient.

Consider the diagram in figure 6.18 that shows a window and the eight surrounding regions. Each of the four digits are given a value of 1 if the point is above (1st digit), below (2nd digit), right (3rd digit) and left (4th digit) of the clip window or 0 otherwise

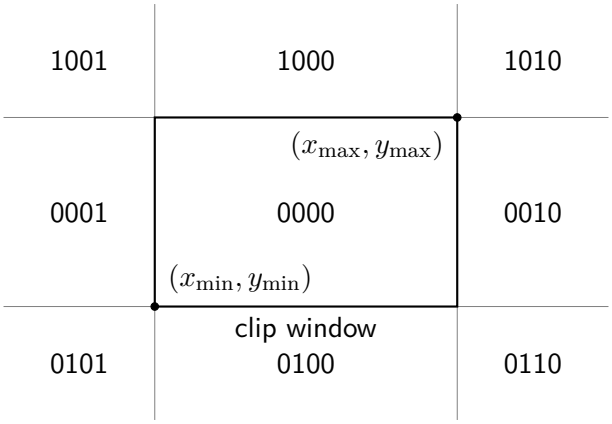


Figure 6.18: Bitcodes used in the Cohen-Sutherland algorithm.

The bitcode for an arbitrary point with co-ordinates (x, y) is generated using the function presented in algorithm 15.

Algorithm 15 Function for determining a bitcode for the Cohen-Sutherland algorithm

function BitCode(x, y, $x_{\min}$, $y_{\min}$, $x_{\max}$, $y_{\max}$)
  $X \leftarrow 0000$
  if $y > y_{\max}$ **then**
    $X(1) \leftarrow 1$ ▷ $X(n)$ denotes the n^{th} digit of X
  else if $y < y_{\min}$ **then**
    $X(2) \leftarrow 1$
  end if
  if $x > x_{\max}$ **then**
    $X(3) \leftarrow 1$
  else if $x < x_{\min}$ **then**
    $X(4) \leftarrow 1$
  end if
  return X
end function

The Cohen-Sutherland algorithm uses the bitcodes of the two endpoints of a line to determine whether clipping is required. To do this it makes use of the **bitwise operations** AND and OR.

Definition 24. The logical **OR** operator is denoted by $p \vee q$ and returns a 1 if p or q are true, else it returns 0.

Definition 25. The logical **AND** operator is denoted by $p \wedge q$ and returns a 1 only both p and q are true, else it returns a 0.

The values of $p \vee q$ and $p \wedge q$ for all possible combinations of the binary values p and q are listed in the truth table in table 6.2.

Table 6.2: The truth table for logical OR and AND operators

p	q	$p \vee p$	$p \wedge q$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	1	1

Bitwise operations between two bitcodes apply the logical operators to each corresponding pair of digits to return another bitcode, e.g., $0110 \vee 1100 = 1110$ and $0110 \wedge 100 = 0100$.

The Cohen-Sutherland algorithm is presented in algorithm 16.

Algorithm 16 The Cohen-Sutherland algorithm

function CohenSutherland($\mathbf{a}$, $\mathbf{b}$, $x_{\min}$, $y_{\min}$, $x_{\max}$, $y_{\max}$)
  while true **do**
    $A \leftarrow$ BitCode(a_x, a_y, $x_{\min}$, $y_{\min}$, $x_{\max}$, $y_{\max}$)
    $B \leftarrow$ BitCode(b_x, b_y, $x_{\min}$, $y_{\min}$, $x_{\max}$, $y_{\max}$)
    if $A \vee B = 0000$ **then**
      $\mathbf{a} \rightarrow \mathbf{b}$ does not need to be clipped
      return $\mathbf{a}$ and $\mathbf{b}$
    else if $A \wedge B \neq 0000$ **then**
      $\mathbf{a} \rightarrow \mathbf{b}$ does not enter the clip region so reject line and exit algorithm
    else
      if $A \neq 0000$ **then**
        clip $\mathbf{a}$ to the edge indicated by the first non-zero digit in A
      else
        clip $\mathbf{b}$ to the edge indicated by the first non-zero digit in B
      end if
    end if
  end while
end function

Example 20 The Cohen-Sutherland algorithm is to be applied to clip the lines to the clip region in figure 6.19.

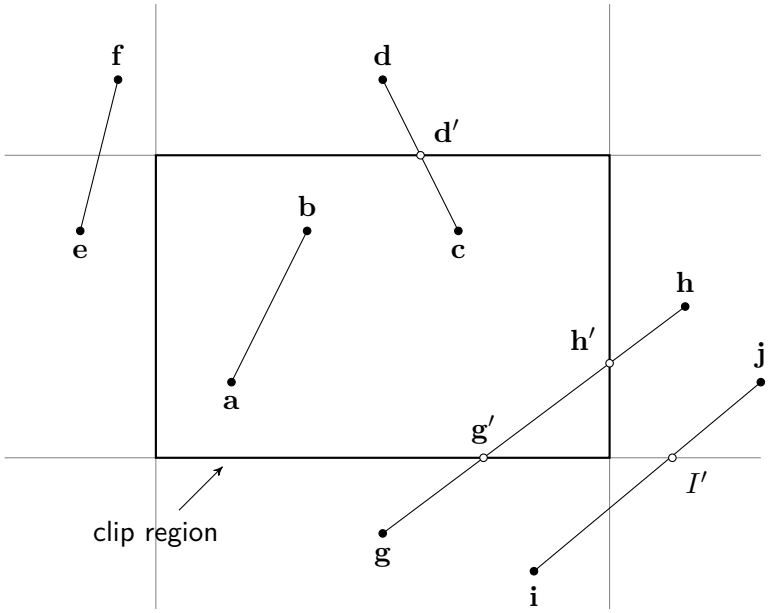


Figure 6.19: Clipping the lines to the clip region using the Cohen-Sutherland algorithm.

- $\mathbf{a} \to \mathbf{b}$:

$$A = 0000,$$
$$B = 0000.$$

$A \vee B = 0000$ therefore line $\mathbf{a} \to B$ is drawn without clipping.

- $\mathbf{c} \to \mathbf{d}$:

$$C = 0000,$$
$$D = 1000.$$

$C \vee D = 1000$ and $C \wedge D = 0000$ therefore line $\mathbf{c} \to \mathbf{d}$ needs to be clipped. The first digit of D is 1 so $\mathbf{d}$ is clipped to the top edge at $\mathbf{d}'$.

$$D' = 0000,$$

$C \vee D' = 0000$ therefore line $\mathbf{c} \to \mathbf{d}'$ is drawn without further clipping.

- $\mathbf{e} \to \mathbf{f}$:

$$E = 0001,$$
$$F = 1001.$$

$E \vee F = 1001$ and $E \wedge F = 0001$ therefore line $\mathbf{e} \to \mathbf{f}$ does not enter the clip region and is rejected.

- $\mathbf{g} \to \mathbf{h}$:

$$G = 0100,$$
$$H = 0010.$$

$G \vee H = 0110$ and $G \wedge H = 0000$ therefore line $\mathbf{g} \to \mathbf{h}$ needs to be clipped. The second digit in G is 1 so $\mathbf{g}$ is clipped to the bottom edge at $\mathbf{g}'$

$$G' = 0000.$$

$G' \vee H' = 0010$ and $G' \wedge H = 0000$ therefore line $\mathbf{g}' \to \mathbf{h}$ needs to be clipped. The third digit of H is 1 so $\mathbf{h}$ is clipped to the right edge at $\mathbf{h}'$.

$$H' = 0000.$$

$G' \vee H' = 0000$ therefore line $\mathbf{g}' \to \mathbf{h}'$ is drawn without further clipping.

- Line $\mathbf{i} \to \mathbf{j}$

$$I = 0100,$$
$$J = 0010.$$

$I \vee J = 0110$ and $I \wedge J = 0000$ therefore line $\mathbf{i} \to \mathbf{j}$ needs to be clipped. The second digit of I is 2 so $\mathbf{i}$ is clipped to the bottom edge at $\mathbf{i}'$.

$$I' = 0010.$$

$I' \vee J = 0010$ and $I' \wedge J = 0010$ so line $\mathbf{i}' \to \mathbf{j}$ is rejected (not drawn).

6.6 Exercises

1. A polygon with the homogeneous view space co-ordinates given below is to be projected onto the screen space defined by a projection plane located at $f_{near} = 4$, a far viewing plane at $f_{far} = 20$, a field of view angle of $fov = 1.5$ and a screen width-to-height aspect ratio of 16 : 9

$$V = \begin{pmatrix} -2 & 3 & 1 \\ -3 & -1 & 2 \\ 6 & 10 & 8 \\ 1 & 1 & 1 \end{pmatrix}.$$

   (a) Determine the projection matrix that is used to project the view space onto to the screen space.

   (b) Hence, calculate the screen space co-ordinates for this polygon.

2. Use the Cyrus-Back algorithm to clip the lines joining the following the following points to the clip region shown in figure 6.20.

   (a) $(2, 1)$ and $(5, 3)$;

   (b) $(1, 4)$ and $(6, 5)$;

   (c) $(4, 1)$ and $(4, 6)$.

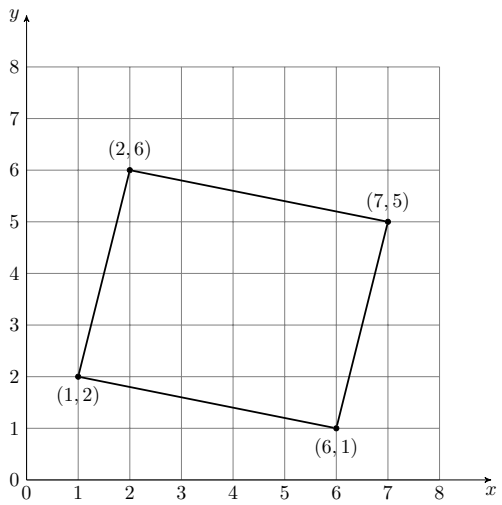


Figure 6.20: Clip region

3. Use the Sutherland-Hodgman algorithm to clip the polygon to the clip region shown in figure 6.21.

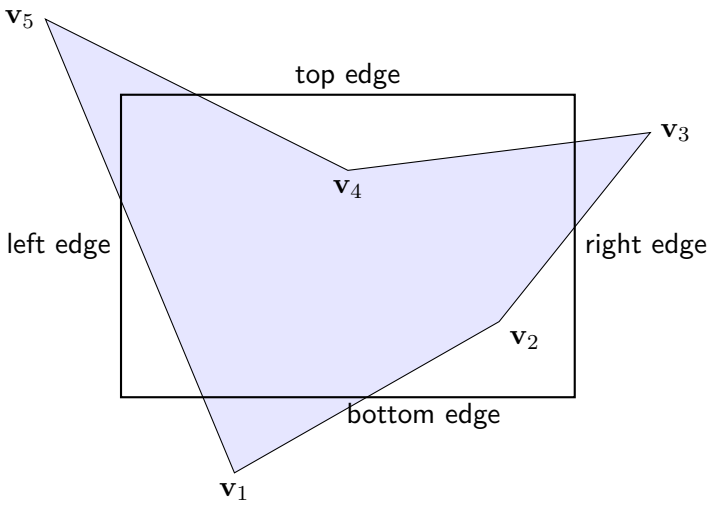


Figure 6.21: The polygon is clipped to the clip region using the Sutherland-Hodgman algorithm.

4. A clip region with four sides parallel to the horizontal and vertical axes is defined by the bottom-left and top-right co-ordinates $(5, 5)$ and $(20, 15)$ respectively. Use the Cohen-Sutherland algorithm to clip the lines $\mathbf{a} \to \mathbf{b}$, $\mathbf{c} \to \mathbf{d}$, $\mathbf{e} \to \mathbf{f}$ and $\mathbf{g} \to \mathbf{h}$ to the clip region

$$\begin{array}{llll} \mathbf{a} = (1, 6), & \mathbf{b} = (8, 10), & \mathbf{c} = (2, 2), & \mathbf{d} = (21, 4), \\ \mathbf{e} = (15, 2), & \mathbf{f} = (22, 9), & \mathbf{g} = (18, 20), & \mathbf{h} = (23, 14). \end{array}$$

The solutions to these exercises can be found on page 140.

Chapter 7

Lighting

The addition of lighting and texture gives visual cues to the geometry of objects, the smoothness of objects, the position of light sources in relation to objects and the orientation of objects in a scene. Consider figure 7.1 that shows two renderings of the Utah teapot. The teapot on the left is the three-dimensional teapot without any kind of lighting model applied. Note that we have no clues to the shape of the teapot. The teapot on the right is the same object but this time a lighting model has been applied. Note that with lighting applied we can see the general shape of the teapot and we can determine depth perception.

(a) without lighting (b) with lighting

Figure 7.1: The Utah teapot rendered with and without lighting.

Lighting models fall into two categories: **direct** illumination models and **global** illumination models. Direct illumination models only take into account light coming directly from a light source. Global illumination models take into account light coming directly from a light source and light reflected off of other objects (figure 7.2). Global illumination models are much more computationally expensive than direct illumination models.

Definition 26. Direct illumination is the illumination of an object from light that is only emitted directly from a light source.

Definition 27. Global illumination is the illumination of an object from light that is emitted from light sources and reflected off other objects in the scene.

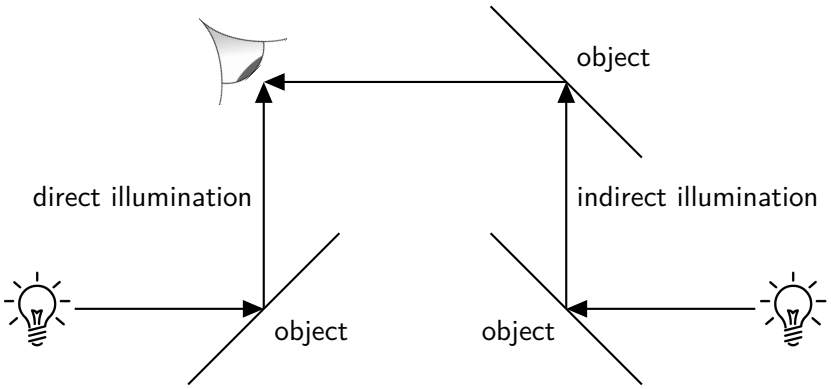


Figure 7.2: Direct and indirect illumination.

7.1 The Phong reflection model

The **Phong reflection model** (Phong 1975) is a direct illumination model and is the most common model used to calculate lighting in virtual worlds and computer games. The Phong reflection model uses a combination of three different types of reflection: ambient reflection, diffuse reflection and specular reflection.

7.1.1 Ambient reflection

Ambient reflection is the reflection of light that does not come directly from a light source, rather is it the sum of all light that is reflected off of other surfaces. Consider a dark room with no sources of light and the curtains drawn. Faint levels of light that are reflected off of other objects illuminate the scene. A simple model for ambient reflection is to assume that all objects in a scene are illuminated equally. This may seem an over simplistic assumption, however, if we were to resolve all light reflected off of all objects (as in global illumination models, e.g., ray tracing) this would be far too computationally expensive to be of practical use.

In the Phong reflection mode, ambient reflection is modelled using

$$A = I_a k_a, \tag{48}$$

where I_a is the intensity of the ambient illumination and k_a is the **ambient reflection coefficient**. The value of k_a is in the range $[0, 1]$ and is set to give the appropriate amount of ambient light in the scene. For example, a day time scene may have a value of k_a close to 1, whereas a dark underground scene may have a value of k_a close to 0.

The effect of changing the value of the ambient reflection coefficient can be seen in figure 7.3 where the Utah teapot is rendered rendered using different values of k_a.

(a) $k_a = 0.25$ (b) $k_a = 0.5$

(c) $k_a = 0.75$ (d) $k_a = 1$

Figure 7.3: The Utah teapot rendered using different values of the ambient reflection coefficient.

7.1.2 Diffuse reflection

Diffuse reflection is the reflection of light falling on a rough or uneven surface where light is scattered in all directions (figure 7.4). To the viewer, a rough surface will look dull when a light is shined on it, this is because of the scattering of the reflected rays.

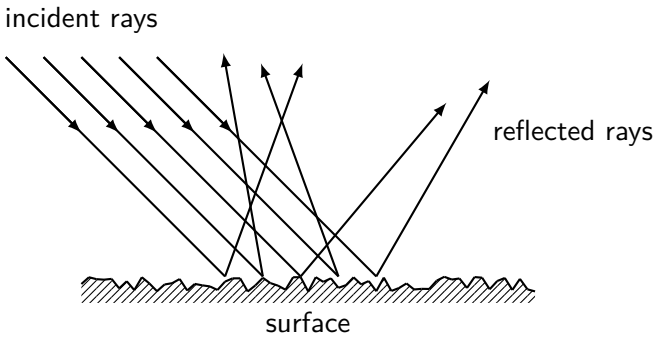


Figure 7.4: Light rays falling on a rough surface are scattered in all directions.

To model diffuse reflection, Phongs reflection model makes the assumption that all light falling on a surface will be scattered equally in all directions (figure 7.5). Whilst this may not be strictly true for all surfaces, it is a fair assumption and adequate for our purposes. Therefore diffuse reflection only depends upon the angle between the surface normal vector and the light source vector.

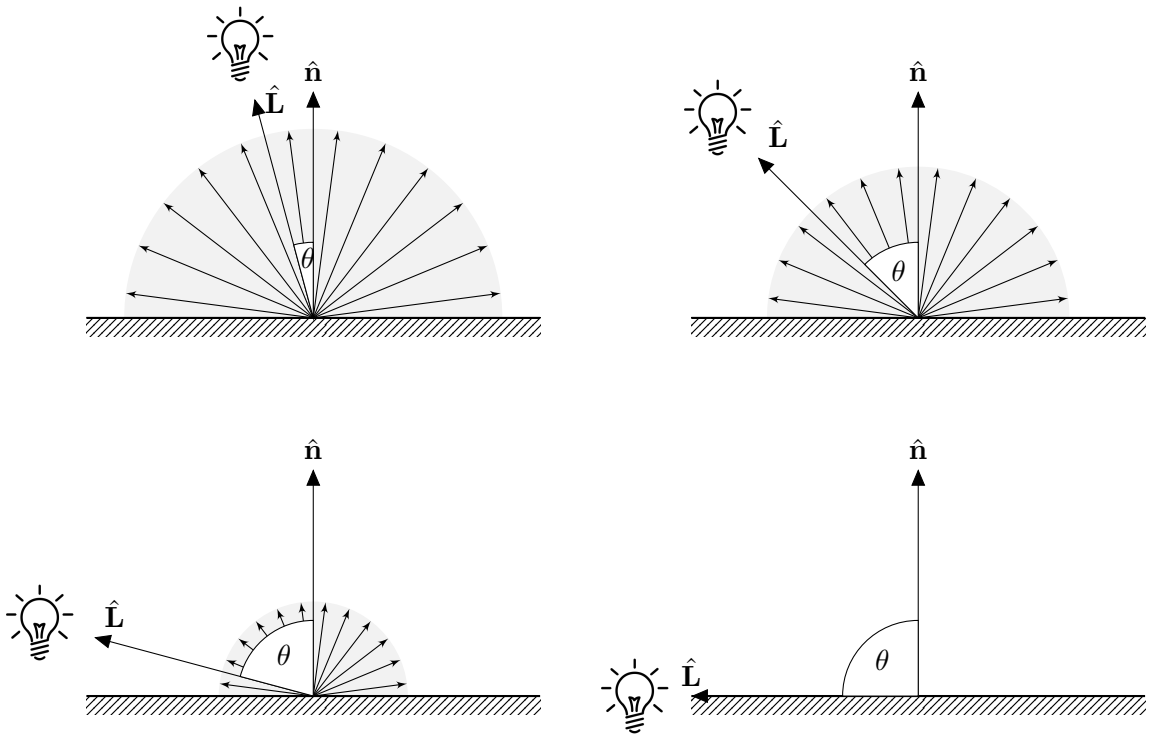


Figure 7.5: The diffuse reflection model scatters light equally in all directions and is dependent upon the angle between light source vector $\hat{\mathbf{L}}$ and the surface normal $\mathbf{n}$.

Consider figure 7.5 where θ denotes the angle between the light source vector $\hat{\mathbf{L}}$ and the surface normal vector $\hat{\mathbf{n}}$. The light will only fall on the surface if $\theta < \pi/2$. When θ is small the light source vector is close to the surface normal vector therefore lots of light will be reflected off of the surface. As θ increases, the amount of light reflected decreases until $\theta = \pi/2$ where the light source vector is parallel to the surface and no light is reflected. When $\theta > \pi/2$ the light source is behind the surface therefore no light is reflected. To model diffuse reflection, the Phong reflection model uses a cosine function, i.e.,

$$D = I_p k_d \max(\cos(\theta), 0),$$

where I_p is the intensity of the point light source and k_d is the **diffuse reflection coefficient**. Similar to k_a, the value of k_d is in the range $[0, 1]$ and is set according to the diffuse properties of the object being modelled. The $\max$ function is used so that the value of D can not be negative.

To simplify the calculations we can replace the $cos(\theta)$ term with a dot product. Recall that the definition of the dot product between two vectors $\mathbf{a}$ and $\mathbf{b}$ is

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\| \cos(\theta). \tag{49}$$

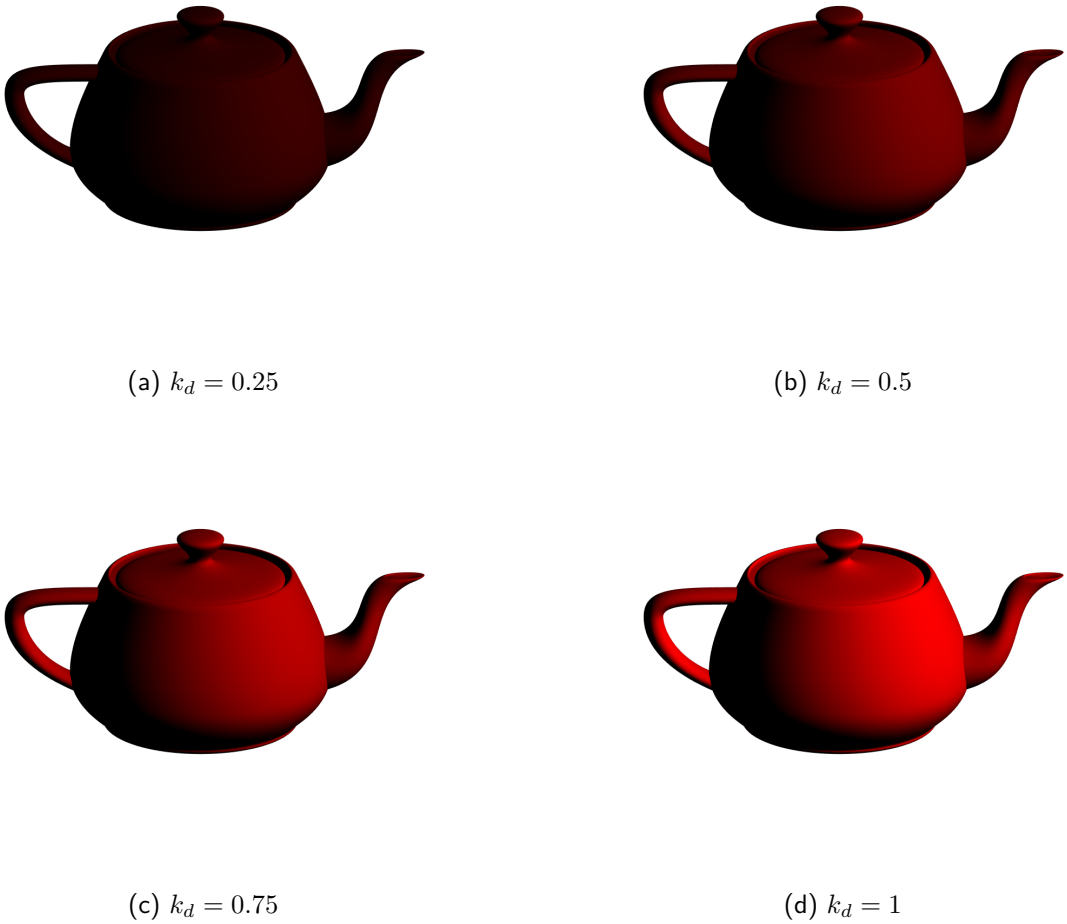
Since $\hat{\mathbf{L}}$ and $\hat{\mathbf{n}}$ are unit vectors then we can write

$$\hat{\mathbf{L}} \cdot \hat{\mathbf{n}} = \cos(\theta),$$

Therefore diffuse reflection is modelled using

$$D = I_p k_d \max(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}, 0). \tag{50}$$

The effect of altering the diffuse reflection coefficient can be seen in figure 7.6 that shows the Utah teapot rendered using different values of k_d.



(a) $k_d = 0.25$ (b) $k_d = 0.5$

(c) $k_d = 0.75$ (d) $k_d = 1$

Figure 7.6: The Utah teapot rendered using different values of the diffuse reflection coefficient ($k_a = 0$).

Example 21 A triangular polygon defined by the V matrix below where each column contains the coordinates of the vertex is lit from a point light source located at $\mathbf{p} = (0, 4, 1)$ with a lighting intensity $I_p = 1$

$$V = \begin{pmatrix} -2 & 1 & 0 \\ -1 & 1 & 3 \\ 4 & 5 & 4 \end{pmatrix}.$$

Given that the surface has a diffuse reflection coefficient of $k_d = 0.8$, calculate the intensity of the diffuse light reflected off the centre of the polygon.

Calculate the surface normal vector

$$\begin{aligned} \mathbf{n} &= ((1,1,5) - (-2,-1,4)) \times ((0,3,4) - (1,1,5)) = (3,2,1) \times (-1,2,-1) = (-4,2,8) \\ \|\mathbf{n}\| &= \sqrt{(-4)^2 + 2^2 + 8^2} = 9.1652 \\ \hat{\mathbf{n}} &= \frac{1}{9.1652}(-4,2,8) = (-0.4364, 0.2182, 0.8729). \end{aligned}$$

Calculate the light source vector

$$\begin{aligned}
\mathbf{c} &= \frac{1}{3}\left((-2,-1,4)+(1,1,5)+(0,3,4)\right) = (-0.3333, 1, 4.3333) \\
\mathbf{L} &= \mathbf{p} - \mathbf{c} = (0,4,1) - (-0.3333, 1, 4.3333) = (-0.3333, -3, 3.3333) \\
\|\mathbf{L}\| &= \sqrt{(-0.33)^2 + (-3)^2 + (3.33)^2} = 4.4969 \\
\hat{\mathbf{L}} &= \frac{1}{4.4969}(-0.3333, -3, 3.3333) = (-0.0741, -0.6671, 0.7412).
\end{aligned}$$

Calculate the diffuse intensity

$$\begin{aligned}
D &= I_p k_d \max(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}, 0) \\
&= 1(0.8)\max\left((-0.0741, -0.6671, 0.7412) \cdot (-0.4364, 0.2182, 0.8729), 0\right) \\
&= 0.8\max(0.5338, 0) \\
&= 0.4270.
\end{aligned}$$

7.1.3 Specular reflection

Specular reflection is the reflection of light off a smooth surface. Light rays falling on a smooth surface will be reflected predominantly in one direction (figure 7.7). To the viewer, a smooth surface will appear shiny when a light is shined on it e.g., a mirror.

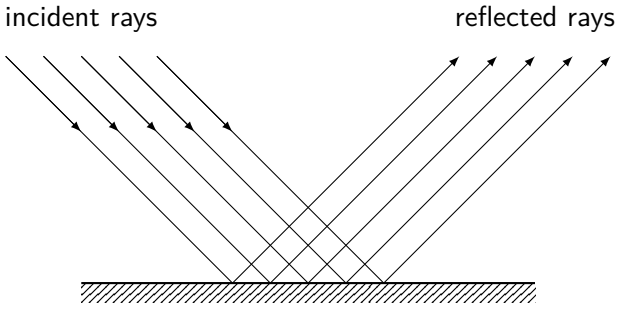


Figure 7.7: Light rays falling on a smooth surface are scattered in one direction.

Consider figure 7.8 where $\hat{\mathbf{R}}$ is the **reflection vector** that represents the direction of a reflected ray of light and $\hat{\mathbf{V}}$ is the **viewing vector** pointing to the viewer. The angle between the reflection vector and the surface normal is the same as the angle between the light source vector and the surface normal. The angle between the viewing vector and the reflecting vector is denoted by α. The amount of reflected light that can be seen by the viewer depends upon the size of α. When α is close to zero, the viewing vector is close to the reflection vectors and the viewer should be able to see a lot of the reflected light. When α is larger, the viewing vector is further away from the reflection vector and the amount of reflected light that the viewer can see decreases.

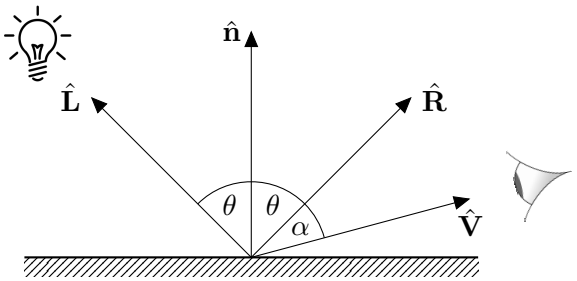


Figure 7.8: Modelling specular reflection.

To model specular reflection, the Phong reflection model uses a cosine function raised to a power, i.e.,

$$S = I_p k_s \cos^n(\alpha),$$

where I_p is the intensity of the point light source as used in the diffuse reflection model, k_s is the **specular coefficient** and n is the **specular exponent**. k_s is in the range $[0, 1]$ which controls the shininess of the surface where a k_s value close to zero means the surface is dull and when k_s is close to 1 the surface is shiny. The value of n is any positive real number and controls the spread of the specular reflection (figure 7.9).

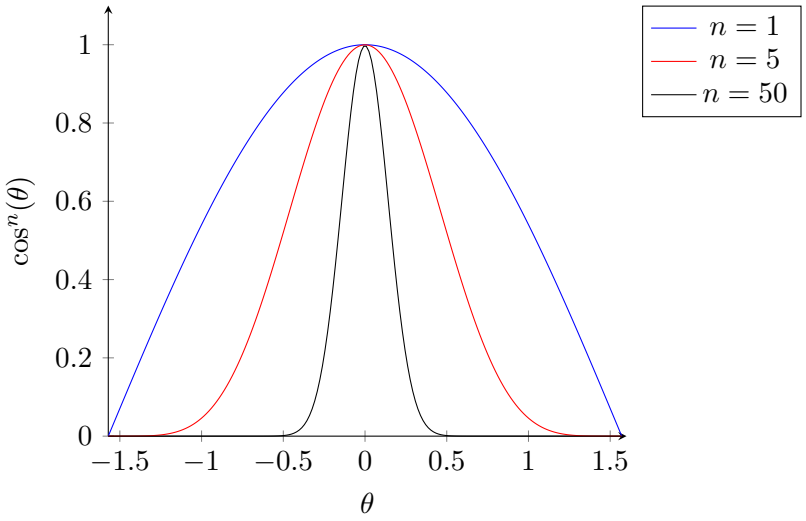
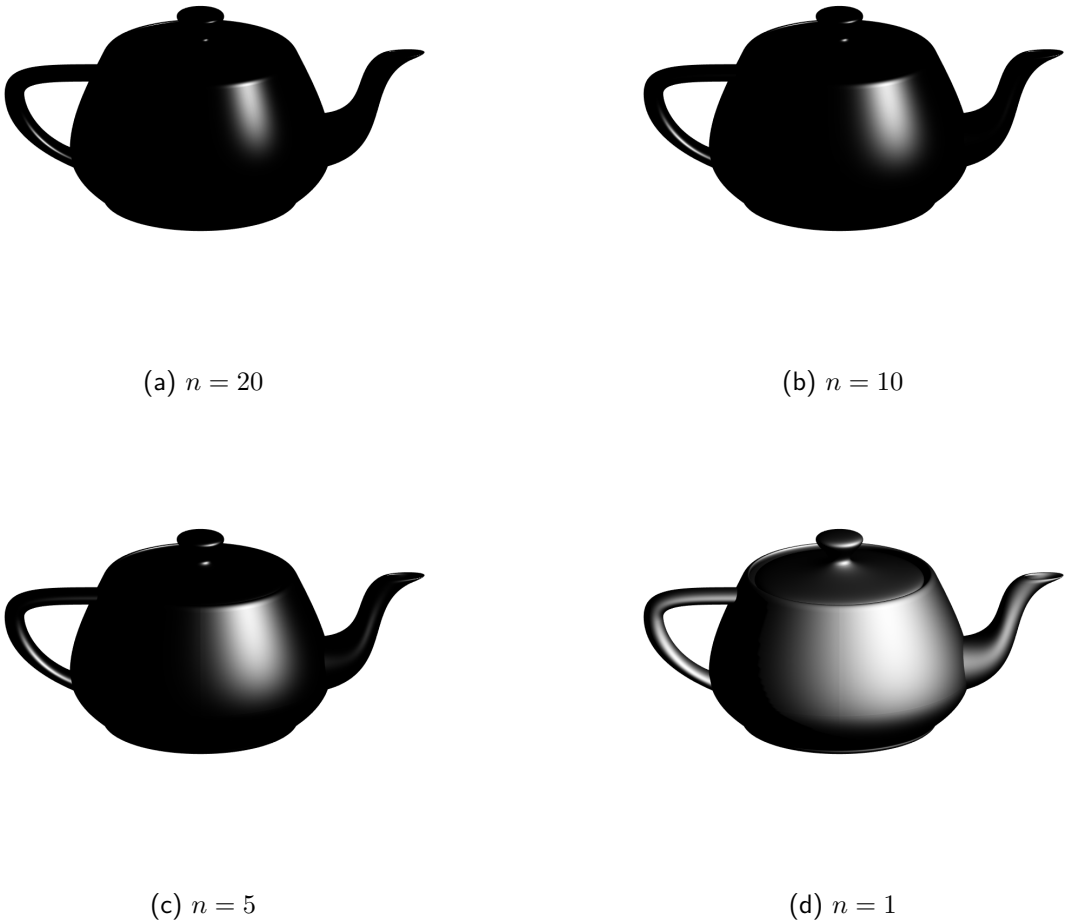


Figure 7.9: Plots of $\cos^n(\alpha)$ for different values of the specular exponent n.

Similar to the diffuse reflection model, we can substitute a dot product in place of the cosine function to improve computational efficiency. If $\hat{\mathbf{R}}$ and $\hat{\mathbf{V}}$ are unit vectors then specular reflection model is

$$S = I_p k_s (\hat{\mathbf{V}} \cdot \hat{\mathbf{R}})^n. \tag{51}$$

The effect of altering the specular exponent can be seen in figure 7.10 where the Utah teapot has been rendered using different values for n.



(a) $n = 20$ (b) $n = 10$

(c) $n = 5$ (d) $n = 1$

Figure 7.10: The Utah teapot rendered using different values of the specular exponent ($k_a = 0$, $k_d = 0$, $k_s = 1$).

7.1.4 Calculating the reflection vector

To calculate the reflection vector $\hat{\mathbf{R}}$ we can use the **projection** of one vector onto another. Consider figure 7.11 where the vector $\mathbf{a}$ is projected onto vector $\mathbf{b}$ resulting in vector $\mathbf{c}$.

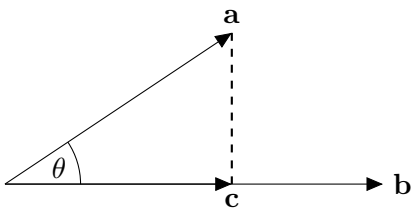


Figure 7.11: The vector $\mathbf{a}$ projected onto vector $\mathbf{b}$ results in vector $\mathbf{c}$.

To determine $\mathbf{c}$ we use the definition of a dot product

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\| \cos(\theta).$$

Since $\cos(\theta) = \dfrac{\|\mathbf{c}\|}{\|\mathbf{a}\|}$ then

$$\mathbf{a} \cdot \mathbf{b} = \|\mathbf{a}\|\|\mathbf{b}\|\frac{\|\mathbf{c}\|}{\|\mathbf{a}\|} = \|\mathbf{b}\|\|\mathbf{c}\|,$$
$$\therefore \|\mathbf{c}\| = \frac{\mathbf{a} \cdot \mathbf{b}}{\|\mathbf{b}\|}$$

If $\hat{\mathbf{b}}$ is a unit vector then

$$\|\mathbf{c}\| = \mathbf{a} \cdot \hat{\mathbf{b}}$$

and

$$\mathbf{c} = (\mathbf{a} \cdot \hat{\mathbf{b}})\hat{\mathbf{b}}.$$

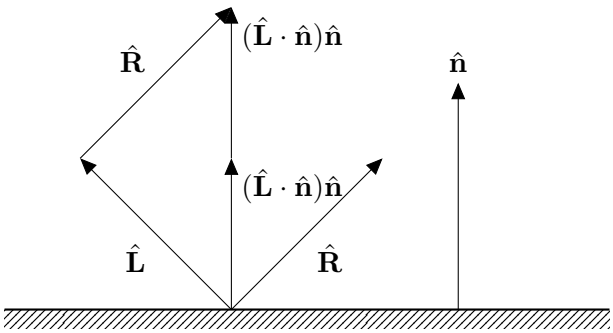


Figure 7.12: Calculating the reflection vector $\hat{\mathbf{R}}$.

Consider figure 7.12 where the relationship between the light source vector $\hat{\mathbf{L}}$, the reflection vector $\hat{\mathbf{R}}$ and the surface unit normal vector $\hat{\mathbf{n}}$ is shown. The projection of $\hat{\mathbf{L}}$ onto $\hat{\mathbf{n}}$ is $(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$ and the sum of $\hat{\mathbf{L}}$ and $\hat{\mathbf{R}}$ is equal to two times $(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}}$, i.e.,

$$\hat{\mathbf{L}} + \hat{\mathbf{R}} = 2(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}},$$

rearranging to make $\hat{\mathbf{R}}$ the subject gives

$$\hat{\mathbf{R}} = 2(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} - \hat{\mathbf{L}}. \tag{52}$$

Example 22 The triangular polygon from example 21 is viewed from the origin $(0, 0, 0)$. Given that the surface has a specular reflection coefficient of $k_s = 1$ and a specular exponent of $n = 5$, calculate the intensity of the specular light seen by the viewer.

We saw in example 21 that the surface normal vector and light source vector were

$$\hat{\mathbf{n}} = (-0.4364, 0.2182, 0.8729),$$
$$\hat{\mathbf{L}} = (-0.0741, -0.6671, 0.7412).$$

Calculating the reflection vector

$$\begin{aligned}\hat{\mathbf{R}} &= 2(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}})\hat{\mathbf{n}} - \hat{\mathbf{L}} \\ &= 2\left((-0.0741, -0.6671, 0.7412) \cdot (-0.4364, 0.2182, 0.8729)\right)(-0.4364, 0.2182, 0.8729) \\ &\quad - (-0.0741, -0.6671, 0.7412) \\ &= (-0.3918, 0.9001, 0.1906).\end{aligned}$$

Calculate the viewing vector

$$\begin{aligned}\mathbf{V} &= \mathbf{c} - (0,0,0) = (-0.3333, 1, 4.3333)\\ \|\mathbf{V}\| &= \sqrt{0.3333^2 + 1^2 + 4.3333^2} = 4.4597\\ \hat{\mathbf{V}} &= \frac{1}{4.4597}(-0.3333, 1, 4.3333) = (-0.0747, 0.2242, 0.9717).\end{aligned}$$

Calculating the specular light reflected to the viewer

$$\begin{aligned}S &= I_p k_s (\hat{\mathbf{V}} \cdot \hat{\mathbf{R}})^n\\ &= 1(1)\left((-0.0747, 0.2242, 0.9717) \cdot (-0.3918, 0.9001, 0.1906)\right)^5\\ &= 0.4163^5 = 0.0625.\end{aligned}$$

7.1.5 Attenuation

Attenuation is the loss of light energy through space. The Phong reflection model uses an approximation of the attenuation that multiplies both the diffuse and specular terms. This approximation is

$$f_{att} = 1 - \left(\frac{d}{r}\right)^2, \tag{53}$$

where r is the radius of the light source's sphere of influence.

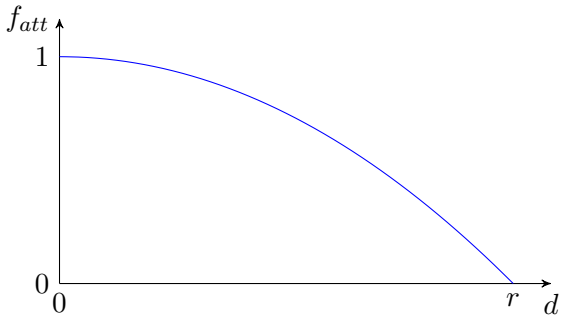


Figure 7.13: The attenuation reduces the intensity of diffuse and specular reflection components based on the distance d of the object from the light source.

7.1.6 Phong's reflection model

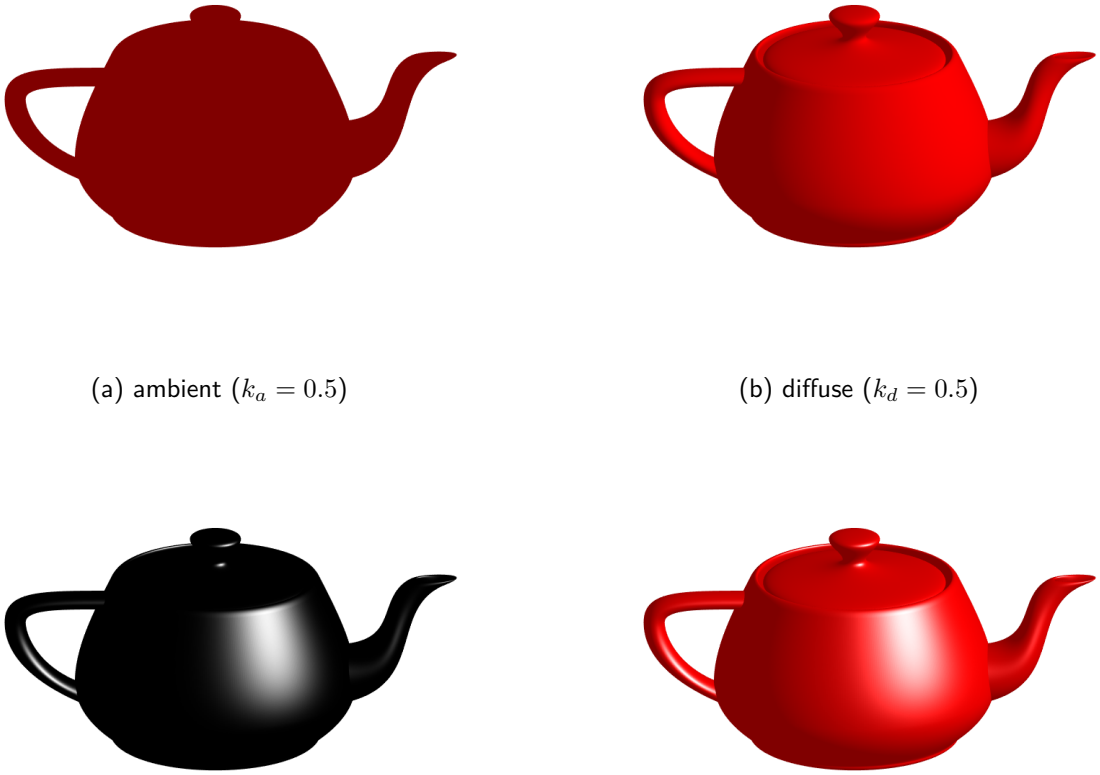
Combining models of ambient reflection equation (48)), diffuse reflection equation (50)), specular reflection equation (51)) and the attenuation factor gives Phong's reflection model

$$I = \underbrace{I_a k_a}_{\text{ambient}} + \underbrace{f_{att} I_p k_d \max(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}, 0)}_{\text{diffuse}} + \underbrace{f_{att} I_p k_s (\hat{\mathbf{V}} \cdot \hat{\mathbf{R}})^n}_{\text{specular}}.$$

Factorising the diffuse and specular terms gives

$$I = I_a k_a + f_{att} I_p \left(k_d \max(\hat{\mathbf{L}} \cdot \hat{\mathbf{n}}, 0) + k_s (\hat{\mathbf{V}} \cdot \hat{\mathbf{R}})^n\right). \tag{54}$$

figure 7.14 demonstrates how the separate components of the Phong reflection model combine to produce a realistic rendering of the Utah teapot. Note that the colour of the specular reflections are not the same colour as the object. The colour of specular reflections depends upon the colour of the point light source.



(a) ambient ($k_a = 0.5$)

(b) diffuse ($k_d = 0.5$)

(c) specular ($k_s = 1, n = 5$)

(d) ambient + diffuse + specular

Figure 7.14: Components of the Phong reflection model applied to the Utah teapot.

7.1.7 Object and light source colour

The colour of objects and light sources is modelled using the RGB colour model where levels of Red, Green and Blue are combined to produce colours in the visible spectrum. To account for the colour of objects and light sources, the Phong reflection model is applied to each colour component in the RGB colour model. Let O_R, O_G and O_B be the red, green and blue colour components for the object, I_{aR}, I_{aG} and I_{aB} be the colour of the ambient light source and I_{pR}, I_{pG} and I_{pB} be the colour of the point light source, then the Phong reflection model taking into account object and light source colour is:

$$
\begin{aligned}
I_R &= I_{aR}k_a + f_{att}I_{pR}\left(O_R k_d \max(\hat{\mathbf{L}}\cdot\hat{\mathbf{n}}, 0) + k_s(\hat{\mathbf{V}}\cdot\hat{\mathbf{R}})^n\right), \\
I_G &= I_{aG}k_a + f_{att}I_{pG}\left(O_G k_d \max(\hat{\mathbf{L}}\cdot\hat{\mathbf{n}}, 0) + k_s(\hat{\mathbf{V}}\cdot\hat{\mathbf{R}})^n\right), \\
I_B &= I_{aB}k_a + f_{att}I_{pB}\left(O_B k_d \max(\hat{\mathbf{L}}\cdot\hat{\mathbf{n}}, 0) + k_s(\hat{\mathbf{V}}\cdot\hat{\mathbf{R}})^n\right).
\end{aligned}
$$

For simplicity the colour components are combined so that $\lambda = (Red, Green, Blue)$ giving

$$
I_\lambda = I_{a\lambda}k_a + f_{att}I_{p\lambda}\left(O_\lambda k_d \max(\hat{\mathbf{L}}\cdot\hat{\mathbf{n}}, 0) + k_s(\hat{\mathbf{V}}\cdot\hat{\mathbf{R}})^n\right). \tag{55}
$$

7.1.8 Phong's reflection model for multiple light sources

The attenuation factor, intensity of the point light source, the lighting vector and reflection vector all depend on the light source and its position. If there are multiple light sources in a scene, the diffuse and

specular reflection models need to be calculated for each light source. To extend Phong's reflection model to take into account multiple light sources we simply the sum the diffuse and specular terms for each light source, i.e.,

$$I_\lambda = I_{a\lambda}k_a + \sum_{i=1}^{m} f_{att,i}I_{p\lambda,i}\left(O_\lambda k_d \max(\hat{\mathbf{L}}_i \cdot \hat{\mathbf{n}}, 0) + k_s(\hat{\mathbf{V}} \cdot \hat{\mathbf{R}}_i)^n\right) \tag{56}$$

where m is the number of light sources.

The variables used in the Phong reflection model are summarised in table 7.1.

Table 7.1: Variables used in the Phong reflection model

Variable	Description
$I_{a\lambda}$	Ambient light intensity ($\lambda = (Red, Green, Blue)$)
$I_{p\lambda}$	Point light source intensity
O_λ	Object colour
k_a	Ambient reflection coefficient
k_d	Diffuse reflection coefficient
k_s	Specular reflection coefficien
n	Specular exponent
f_{att}	Attenuation factor
$\hat{\mathbf{n}}$	Surface normal vector
$\hat{\mathbf{L}}$	Light source vector
$\hat{\mathbf{R}}$	Reflection vector
$\hat{\mathbf{V}}$	Viewing vector

7.2 Shading methods

The Phong reflection model presented in the previous section allows us to calculate the lighting for a single pixel given the position of the point light source and the viewer. To apply the Phong reflection model to a scene we need to consider how all pixels in a polygon are to be shaded. In this section we will look at three shading methods: flat shading, Gouraud shading and Phong shading.

7.2.1 Flat shading

Flat shading (also known as **Lambertian shading**) assumes that all pixels in a polygon are illuminated equally. The Phong reflection model is calculated using the normal vector for the polygon and all pixels assume the same illumination intensity.

The MATLAB `peaks` function has been shaded using flat shading in figure 7.15.

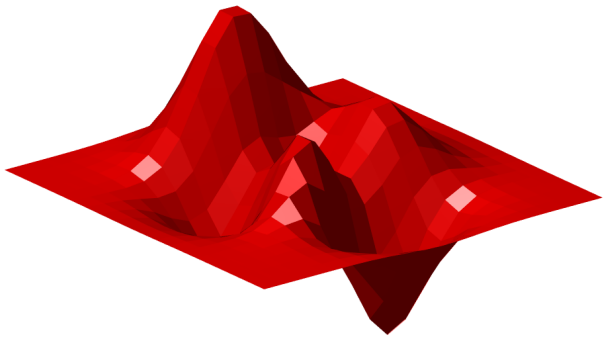


Figure 7.15: MATLAB's `peaks` function shaded using flat shading (25×25 polygons).

7.2.2 Mach banding

The rendering of the `peaks` function in figure 7.15 demonstrates an effect called **mach banding**. It is easy for us to be able to detect the edges of the polygons and the facets in the image. This is because the human brain has evolved to be very good at detecting changes in colour (useful for hunting and avoiding predators).

One method that we could use to avoid the mach banding effect is to increase the number of polygons that are used to represent a surface. Figure 7.17 shows the peaks function with 400×400 polygons shaded using flat shading. Increasing the number of polygons that represent a surface is obviously not a practical option so we need better shading methods.

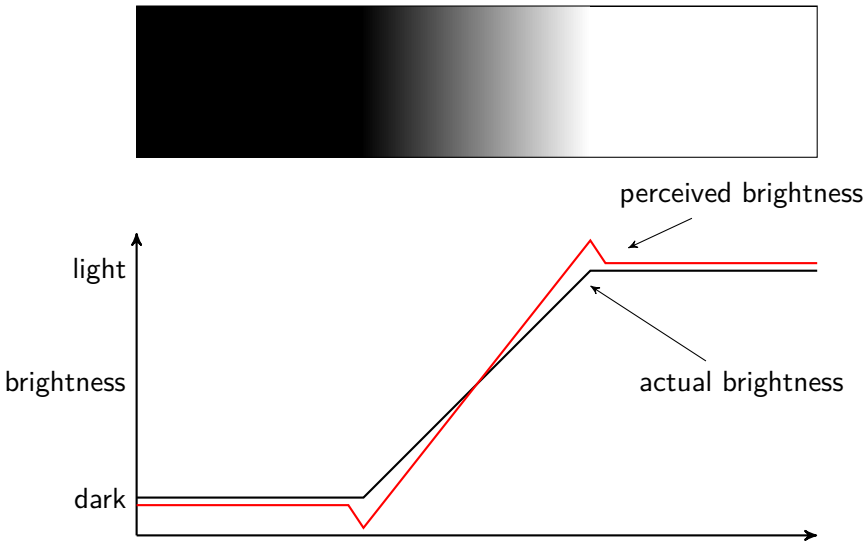


Figure 7.16: Mach banding means that changes in colour are exaggerated.

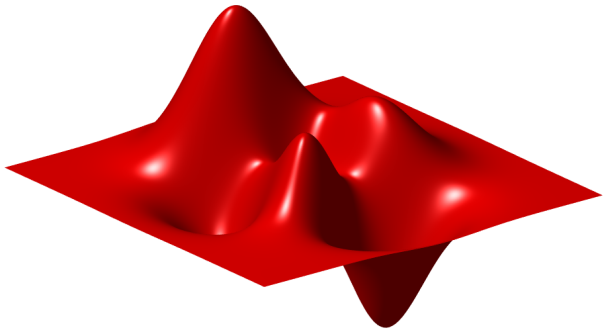


Figure 7.17: MATLAB's `peaks` function shaded using flat shading (400×400 polygons).

7.2.3 Gouraud shading

Gouraud shading was first developed by Henri Gouraud (1973) and it attempts to improve the appearance of three-dimensional objects when lit using the Phong reflection model. We have seen that shading a polygon using a constant colour results in a poor image, Gouraud shading linearly interpolates the intensity of the shading of the pixels that make up the interior of the polygons.

Gouraud shading is applied by applying the Phong reflection model to calculate the intensity of the vertex pixels. The normal vectors used in the Phong reflection model are calculated by averaging the the normal vectors of the polygons that share that vertex. For all other pixels in a polygon, the illumination intensity of the vertex pixels is interpolated across the polygon.

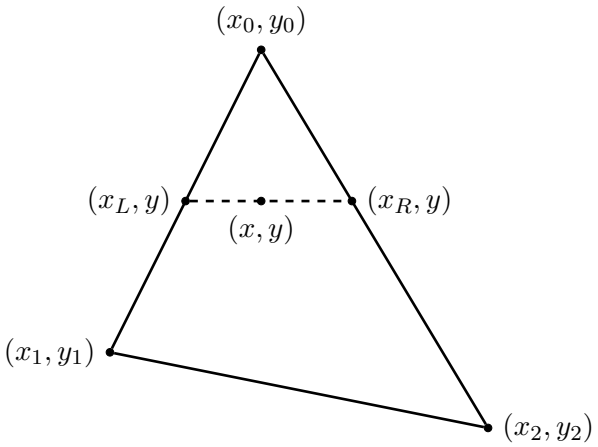


Figure 7.18: The illumination intensities of the interior pixels (x, y) are calculated by interpolating across the scanline between the scan extrema (x_L, y) and (x_R, y).

To calculate the illumination intensities of the pixels in a polygon Gouraud shading uses the scanline algorithm (see section 2.4.3 in chapter 2). Consider figure 7.18 that shows a polygon with vertices (x_0, y_0), (x_1, y_1) and (x_2, y_2). (x_L, y) and (x_R, y) are the left and right scan extrema respectively and (x, y) is pixel along the scanline. Let I_0 and I_1 be the illumination intensities of the vertex pixels (x_0, y_0) and (x_1, y_1) respectively (calculated using the Phong reflection model).

The scanline algorithm loops through each horizontal row of pixels starting at the pixel with the smallest y co-ordinate (remember that the origin of a raster is in the top left-hand corner). The interpolating equations for the intensities of the scan extrema pixels are

$$\begin{aligned} I_L &= I_0 + \Delta I_L, \\ I_R &= I_0 + \Delta I_R, \\ \Delta I_L &= \frac{I_p - I_q}{y_p - y_q}, \\ \Delta I_R &= \frac{I_r - I_s}{y_r - y_s}, \end{aligned}$$

where p, q and r, s are the indices of the upper and lower vertices for the left and right-hand polygon edges respectively. Note that like with scanline filling, ΔI_L and ΔI_R is constant for all points along the current edge so can be pre-calculated to save computational effort.

The interpolating equations for the intensity of the pixels on the scanline are

$$\begin{aligned} I &= I + \Delta I, \\ \Delta I &= \frac{I_R - I_L}{x_R - x_L}. \end{aligned}$$

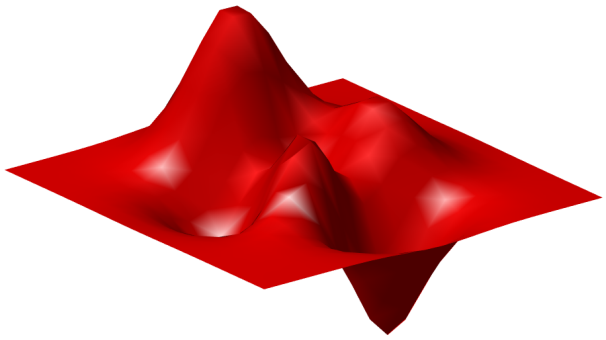


Figure 7.19: MATLAB's peaks function shaded using Gouraud shading (25×25 polygons).

The peaks function has been shaded using Gouraud shading in figure 7.19. It is clearly noticeable that Gouraud shading produces a significantly better image than when lit using flat shading. However because the Phong lighting model is only applied at the vertices, the specular reflection components do not appear to be as vivid as they should be. Also, it is possible to determine the edges of the polygons when the specular reflections are predominant at the vertices. Gouraud shading does present an improvement over the flat shading method but one major disadvantage is that when a light source is close to the centre of a large polygon, the Phong lighting model applied at the vertices of the polygon will not be lit as brightly has the source been close to the vertices. The interpolation across the interior of the polygon means that although there is a light source close to the interior, the lighting of the polygon does not reflect this. It is for this reason that when using Gouraud shading smaller polygons are preferable.

7.2.4 Phong shading

Phong shading (Phong 1975) (not to be confused with the Phong reflection model) is a shading method that improves on Gouraud shading. Instead of interpolating the illumination intensities of the vertex pixels across the polygon, Phong shading interpolates the normal vectors of the vertex pixels. Since every pixel in a polygon has a normal vector, Phong's reflection model can then be calculated for each pixel.

Consider figure 7.20 that shows the same three sided polygon used to describe Gouraud shading except the normal vectors at each vertex are shown. $\hat{\mathbf{n}}_0$, $\hat{\mathbf{n}}_1$ and $\hat{\mathbf{n}}_2$ are the normal vectors for the vertices (x_0, y_0), (x_1, y_1) and (x_2, y_2) respectively, using a similar interpolation method shown in section 7.2.3 the normal vectors at pixels (x_L, y), (x_R, y) and (x, y) are calculated using

$$\begin{aligned}
\mathbf{n}_L &= \mathbf{n}_L + \Delta\mathbf{n}_L, \\
\mathbf{n}_R &= \mathbf{n}_R + \Delta\mathbf{n}_R, \\
\mathbf{n} &= \mathbf{n} + \Delta\mathbf{n}, \\
\Delta\mathbf{n}_L &= \frac{\mathbf{n}_p - \mathbf{n}_q}{y_p - y_q}, \\
\Delta\mathbf{n}_R &= \frac{\mathbf{n}_r - \mathbf{n}_s}{y_r - y_s}, \\
\Delta\mathbf{n} &= \frac{\mathbf{n}_R - \mathbf{n}_L}{x_R - x_L},
\end{aligned}$$

where p, q and r, s are the indices of the upper and lower vertices for the left and right-hand edges respectively.

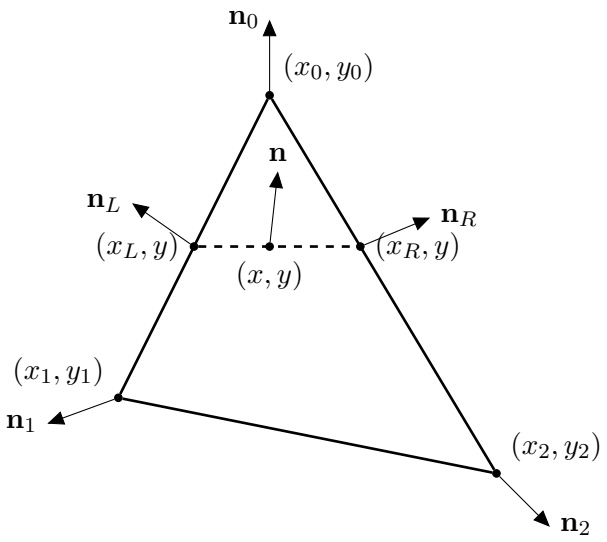


Figure 7.20: Phong shading interpolates the normals vectors of the vertices across all pixels in the polygon.

The `peaks` function has been shaded using Phong shading in figure 7.21. It is clear that Phong shading produces a much improved image than flat or Gouraud shading. The specular reflections are more vivid that in the Gouraud plot (figure 7.19) and the plot closely resembles that done using flat shading with 400×400 polygons (figure 7.17).

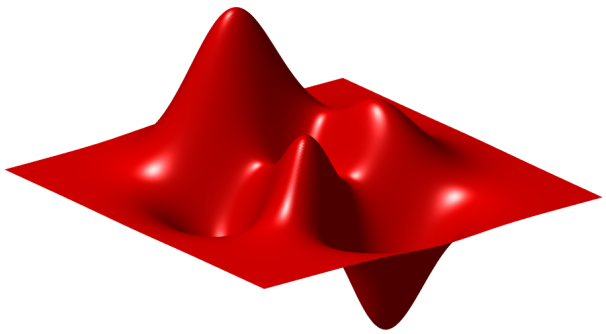


Figure 7.21: MATLAB's `peaks` function shaded using Phong shading (25×25 polygons).

References

Abrash, M. (2001). **Graphics Programming Black Book**. URL: http://www.jagregory.com/abrash-black-book/.

Abrash, M. and Carmack, J. (1996). "Quake". In: **id software**.

Bresenham, J.E. (1965). "Algorithm for computer control of a digital plotter". In: **IBM Systems Journal** 4.1, pp. 25–30.

Carmack, J. and Romero, J. (1993). "Doom". In: **id software**.

Cyrus, M. and Beck, J. (1978). "Generalized two- and three-dimensional clipping". In: **Computers & Graphics**, pp. 23–28.

Gouraud, H. (1973). "Continuous shading of curved surfaces". In: **IEEE Transactions on Computers** 20.6, pp. 623–628.

Phong, B.T. (1975). "Illumination of Computer Generated Images". In: **Communications of ACM** 18.6, pp. 311–317.

Pitteway, M.L.V. (1967). "Algorithm for drawing ellipses of hyperbolae with a digital plotter". In: **Computer Journal** 10.3, pp. 282–289.

Schumaker, R., Brand, B., Gilliland, M., and Sharp, W. (1969). **Study for applying computer generated images to visual simulation**. Tech. rep. General Electric Co.

Sutherland, I. and Hodgman, G.W. (1974). "Reentrant polygon clipping". In: **Communications of the ACM** 17, pp. 32–42.

Unity Technologies (2017). **Unity - Manual: Normal map (Bump mapping)**. URL: https://docs.unity3d.com/Manual/StandardShaderMaterialParameterNormalMap.html (visited on 01/15/2018).

Zapyon (2011). **Painter's algorithm**. URL: https://en.wikipedia.org/wiki/Painter%27s_algorithm#/media/File:Painter%27s_algorithm.svg (visited on 05/2017).

Appendix A

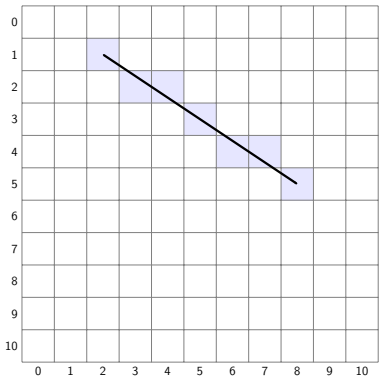
Exercise solutions

A.1 Rasterisation

Solutions to the exercises on page 44.

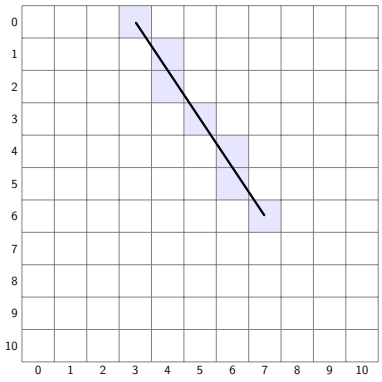
1. (a)

x	y
2	1
3	2
4	2
5	3
6	4
7	4
8	5



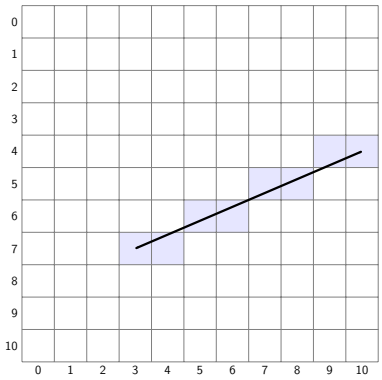
(b)

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



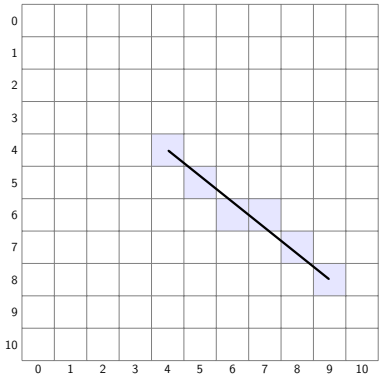
(c)

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



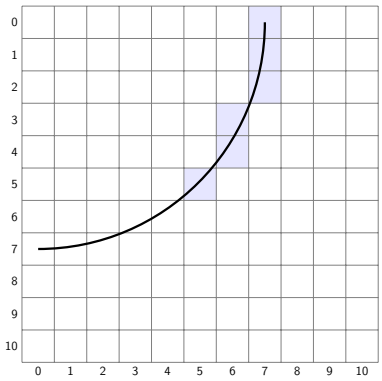
(d)

x	y
9	8
8	7
7	6
6	6
5	5
4	4



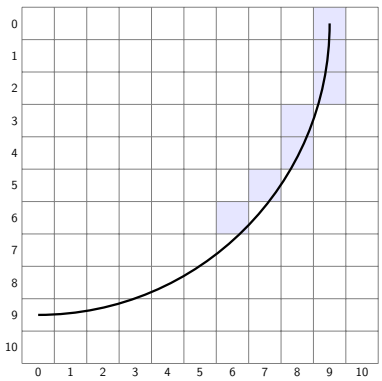
2. (a)

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



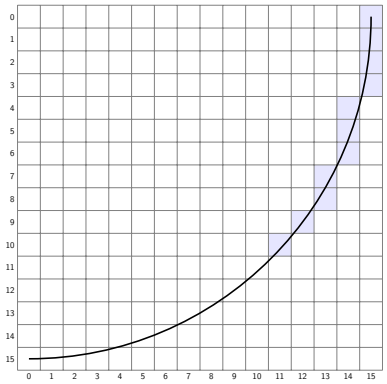
(b) $(9,0)$, $(9,1)$, $(9,2)$, $(8,3)$, $(8,4)$, $(7,5)$, $(6,6)$

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



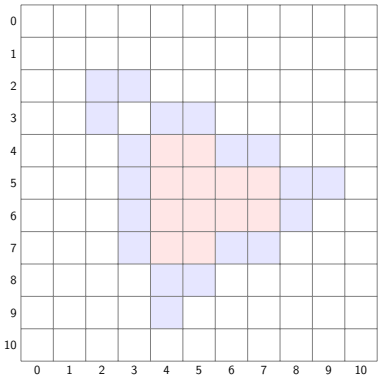
(c) $(15,0)$, $(15,1)$, $(15,2)$, $(15,3)$, $(14,4)$, $(14,5)$, $(14,6)$, $(13,7)$, $(13,8)$, $(12,9)$, $(11,10)$

x	y
15	0
15	1
15	2
15	3
14	4
14	5
14	6
13	7
13	8
12	9
11	10



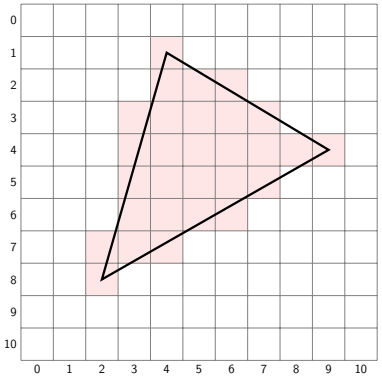
3.

Order	x	y
1	4	4
2	4	5
3	4	6
4	4	7
5	5	7
6	5	6
7	5	5
8	5	4
9	6	5
10	6	6
11	7	6
12	7	5



4.

y	x_L	x_R
1	4	4
2	4	6
3	3	7
4	3	9
5	3	7
6	3	6
7	2	4
8	2	2



A.2 Bézier curves

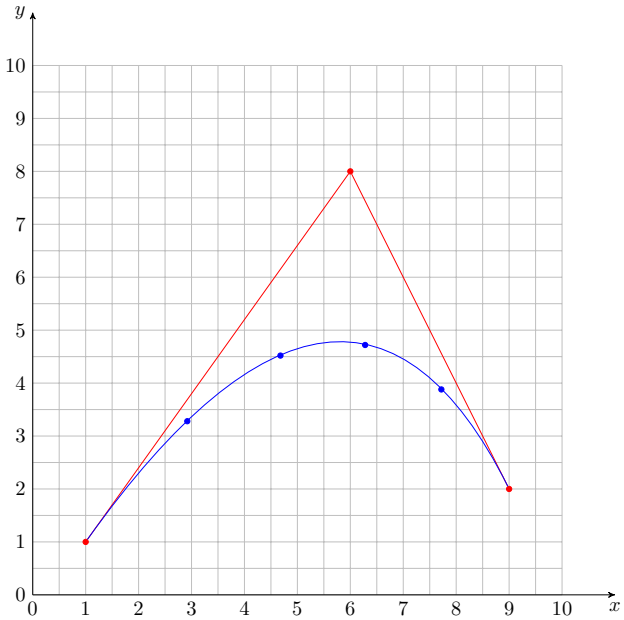
Solutions to the exercises on page 70.

1. (a) $(2.92, 3.28)$

   (b) $(4.68, 4.52)$

   (c) $(6.28, 4.72)$

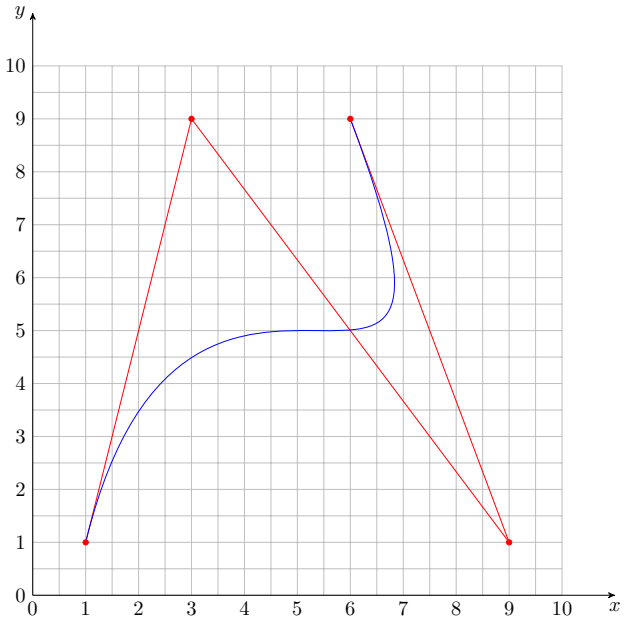
   (d) $(7.72, 3.88)$



2. (a) $\mathbf{c}(t) = (1-t)^3\mathbf{p}_0 + 3t(1-t)^2\mathbf{p}_1 + 3t^2(1-t)\mathbf{p}_2 + t^3\mathbf{p}_3$

   (b) $M = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$

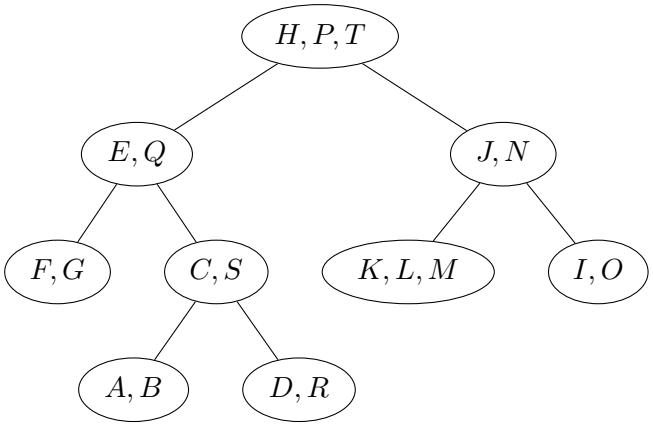
3.



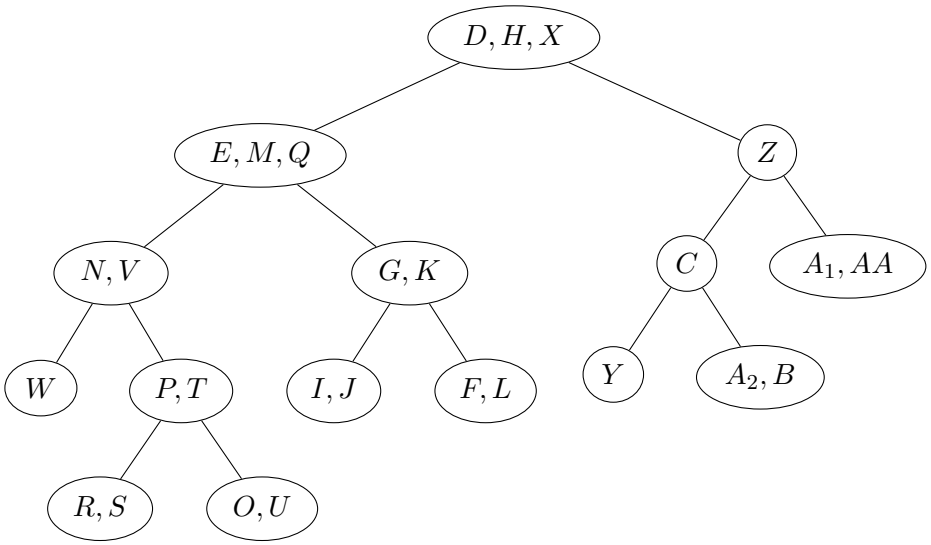
A.3 Hidden surface removal

Solutions to the exercises on page 89.

1. Faces 1 and 4 are front facing.
2. (a) A valid BSP tree is



(b) A valid BSP tree is



3. (a) $\mathbf{p}_1$: $\{K, L, M\}, \{J, N\}, \{I, O\}, \{H, P, T\}, \{F, G\}, \{E, Q\}, \{D, R\}, \{C, S\}, \{A, B\}$;
   $\mathbf{p}_2$: $\{A, B\}, \{C, S\}, \{D, R\}, \{E, Q\}, \{F, G\}, \{H, P, T\}, \{K, L, M\}, \{J, N\}, \{I, O\}$;
   $\mathbf{p}_3$: $\{A, B\}, \{C, S\}, \{D, R\}, \{E, Q\}, \{F, G\}, \{H, P, T\}, \{I, O\}, \{J, N\}, \{K, L, M\}$.

   (b) $\mathbf{p}_3$: $\{R, S\}, \{P, T\}, \{O, U\}, \{N, V\}, \{W\}, \{E, M, Q\}, \{I, J\}, \{G, K\}, \{F, L\}, \{D, H, X\}$, $\{A_1, AA\}, \{Z\}, \{Y\}, \{C\}, \{A_2, B\}$;
   $\mathbf{p}_2$: $\{A_1, AA\}, \{Z\}, \{A_2, B\}, \{C\}, \{Y\}, \{D, H, X\}, \{I, J\}, \{G, K\}, \{F, L\}, \{E, M, Q\}$, $\{R, S\}, \{P, T\}, \{O, U\}, \{N, V\}, \{W\}$;
   $\mathbf{p}_3$: $\{A_1, AA\}, \{Z\}, \{Y\}, \{C\}, \{A_2, B\}, \{D, H, X\}, \{R, S\}, \{P, T\}, \{O, U\}, \{N, V\}, \{W\}$, $\{E, M, Q\}, \{F, L\}, \{G, K\}, \{I, J\}$;
   $\mathbf{p}_4$: $\{A_1, AA\}, \{Z\}, \{A_2, B\}, \{C\}, \{Y\}, \{D, H, X\}, \{I, J\}, \{G, K\}, \{F, L\}, \{E, M, Q\}, \{W\}$, $\{N, V\}, \{O, U\}, \{P, T\}, \{R, S\}$.

A.4 Clipping

Solutions to the exercises on page 112.

1. (a) $P = \begin{pmatrix} 1.0734 & 0 & 0 & 0 \\ 0 & 1.9083 & 0 & 0 \\ 0 & 0 & 1.5 & -10 \\ 0 & 0 & 1 & 0 \end{pmatrix}$

   (b) $V_{screen} = \begin{pmatrix} -0.3578 & 0.3220 & 0.1342 \\ -0.9542 & -0.1908 & 0.4771 \\ -0.1667 & 0.5000 & 0.2500 \\ 1 & 1 & 1 \end{pmatrix}$

2. (a) $(2.9231, 1.6154) \rightarrow (5, 3)$;

   (b) $(1.5263, 4.1053) \rightarrow (6, 5)$;

   (c) $(4, 1.4) \rightarrow (4, 5.6)$.

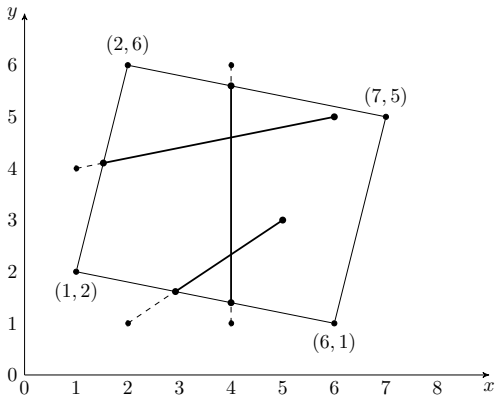
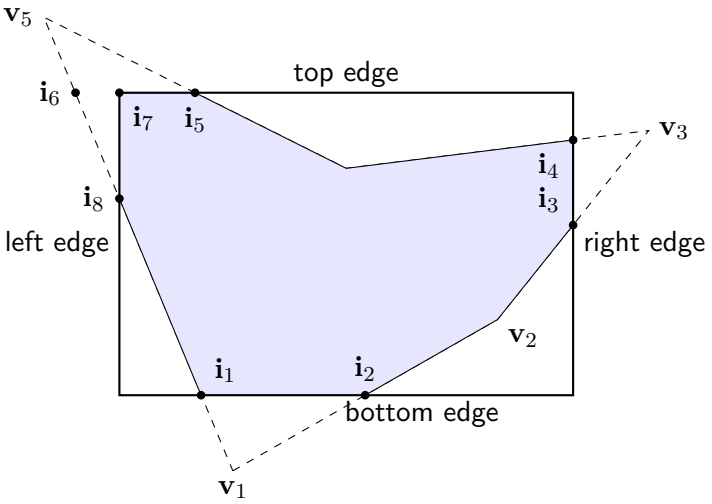


Figure A.1: Clip region

3. $List = \{\mathbf{i}_7, \mathbf{i}_8, \mathbf{i}_1, \mathbf{i}_2, \mathbf{v}_2, \mathbf{i}_3, \mathbf{i}_4, V_4, \mathbf{i}_6\}$



4. (a) $(5, 8.2857) \rightarrow (8, 10)$;

   (b) $(20, 3.8947) \rightarrow (16.25, 3.5)$;

   (c) $(18, 5) \rightarrow (20, 7)$;

   (d) reject line;

Index

Bézier curves
    Bézier surface, 68
    Bernstein polynomials, 59
    general form, 59
    MATLAB code, 64
    matrix form, 62

antialiasing, 47

back-space culling, 75
    algorithm, 76
    normal vector calculation, 75
binary space partitioning, 81
    atomic subspace, 81
    balanced trees, 85
    BSP tree generation, 83
    BSP tree traversial, 86
    BSP trees, 82
    convex set, 81
    hyperplane, 81
    optimising BSP trees, 84
    visibility ordering, 86
Binomial coefficient, 60
Bresenham's algorithm, 11
    algorithm, 15
    derivation, 14
    drawing lines in any direction, 17
    MATLAB code, 15

Cartesian co-ordinate systems, 2
clipping, 91
    aspect ratio, 94
    bitcode, 108
    calculating intersection point, 102
    Cohen-Sutherland algorithm, 108, 109
    Cyrus-Beck algorithm, 100, 102
    field of view, 93
    line clipping, 100
    perspective projection, 92
    point and plane distance, 100
    polygon clipping, 104
    projection matrix, 96
    Sutherland-Hodgman algorithm, 104
    viewing frustum, 91
convolution, 48
    box blur, 50
    Gaussian blur, 50
    sharpening, 52
cross product, 4

dot product, 3

face array, 74
flood fill algorithm, 25

graphics pipeline, 5
    object space, 5
    screen space, 6
    view space, 6
    world space, 5

hidden surface removal
    back-space culling, 75
    binary space partitioning, 81
    painter's algorithm, 78
homogeneous co-ordinates, 2

image processing
    antialiasing, 47
    edge detection, 54
    embossing, 53
    kernel, 48
    Sobel operator, 54
    super sampling antialiasing, 47

kernel, 48

lighting
    ambient reflection, 116
    attenuation, 124
    diffuse reflection, 117
    direct illumination, 115
    flat shading, 127
    global illumination, 115
    Gouraud shading, 128
    Lambertian shading, 127
    mach banding, 127
    Phong reflection model, 116
    Phong shading, 130
    Phong's reflection model for multiple light sources, 125
    reflection vector, 122
    specular reflection, 120
logical and, 108
logical or, 108

magnitude, 3
MATLAB
    image, 11
    imread, 10
    whos, 10
midpoint algorithm, 20
    circle symmetry, 20

normal vector, 74

painter's algorithm, 78
parametric equation of a straight line, 57
parametric equations, 57
perspective projection, 92
perspective projection matrix, 96
point and plane distance, 100
polygon, 7
    concave polygon, 8
    convex polygon, 8
    definition, 7
    edges, 7
    vertices, 7
polynomial, 57
polynomial degree, 57

rasterisation
    Bresenham's algorithm, 11
    drawing circles, 20
    drawing lines in any direction, 17
    drawing polygons, 25
    flood fill algorithm, 25
    idealised image, 9
    line drawing, 11
    midpoint algorithm, 20
    pixel, 9
    pixel co-ordinates, 11
    raster, 9
    raster arrays, 10
    RGB colour model, 9
    scanline, 28
    scanline filling algorithm, 28
    texture mapping, 32
RGB colour model, 9

scan extrema, 28
scanline filling
    algorithm, 29
    scan extrema, 28
    scanline, 28
super sampling antialising, 47

texture mapping, 32
    algorithm, 34
    bump mapping, 42
    calculating scan extrema, 33
    interpolating along scanline, 34
    normal mapping, 42
    perspective corrected algorithm, 40
    perspective correction, 39
    textel, 32
    texture map, 32
    texture space, 32

unit vector, 3
Utah teapot, 73

vector magnitude, 3
vectors, 2
vertex array, 74