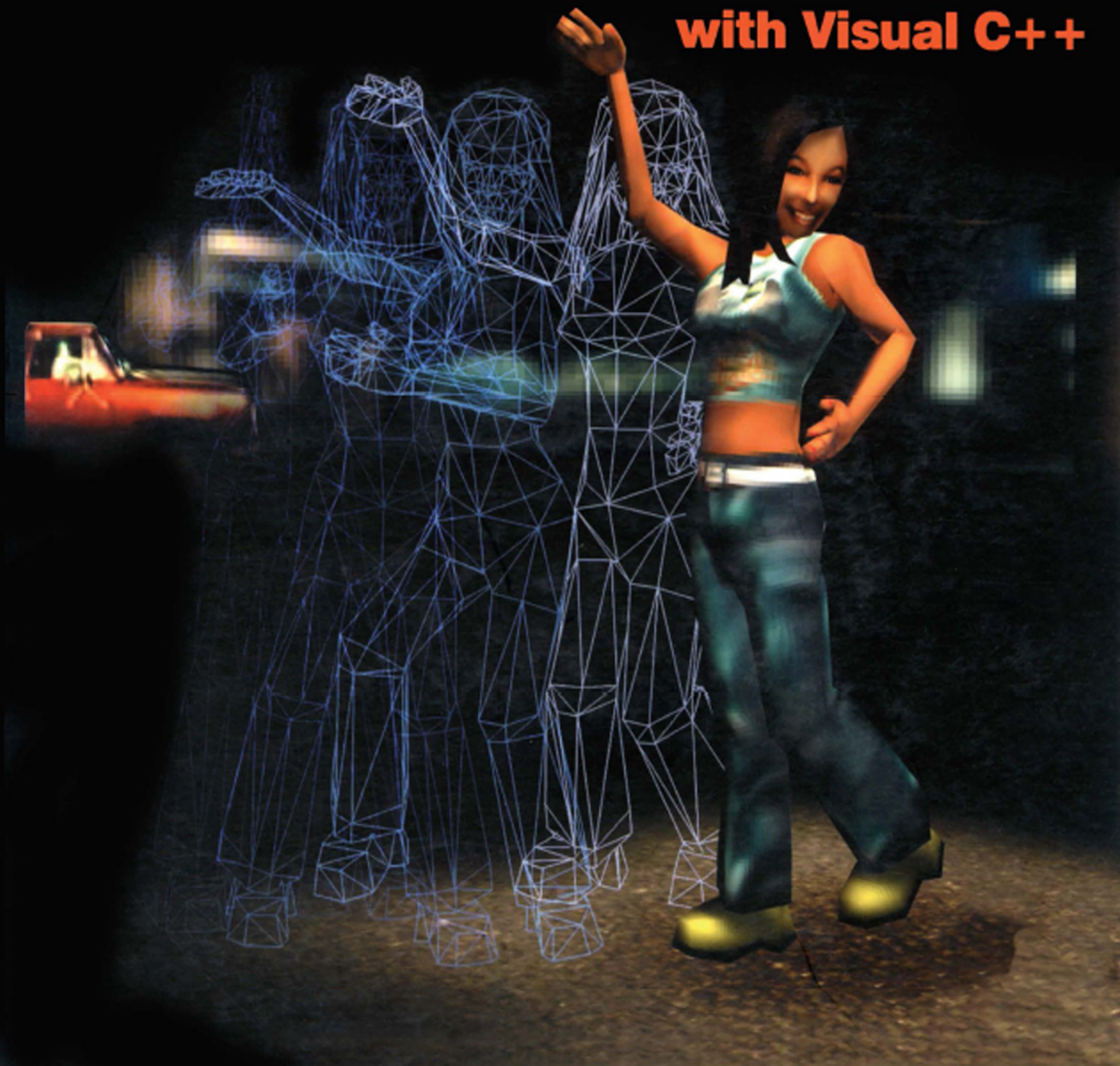
ROUTLEDGE

Nik Lever

Real-time 3D Character Animation

with Visual C++



Real-time 3D Character Animation with Visual C++

This book is dedicated to David Lever (1927–2001).

My Dad, who is greatly missed.

Real-time 3D Character Animation with Visual C++

Nik Lever

Focal Press
Taylor & Francis Group
NEW YORK AND LONDON

First published 2002
This edition published 2012 by Focal Press
70 Blanchard Road, Suite 402, Burlington, MA 01803

Simultaneously published in the UK by Focal Press
2 Park Square, Milton Park, Abingdon, Oxon OX14 4RN

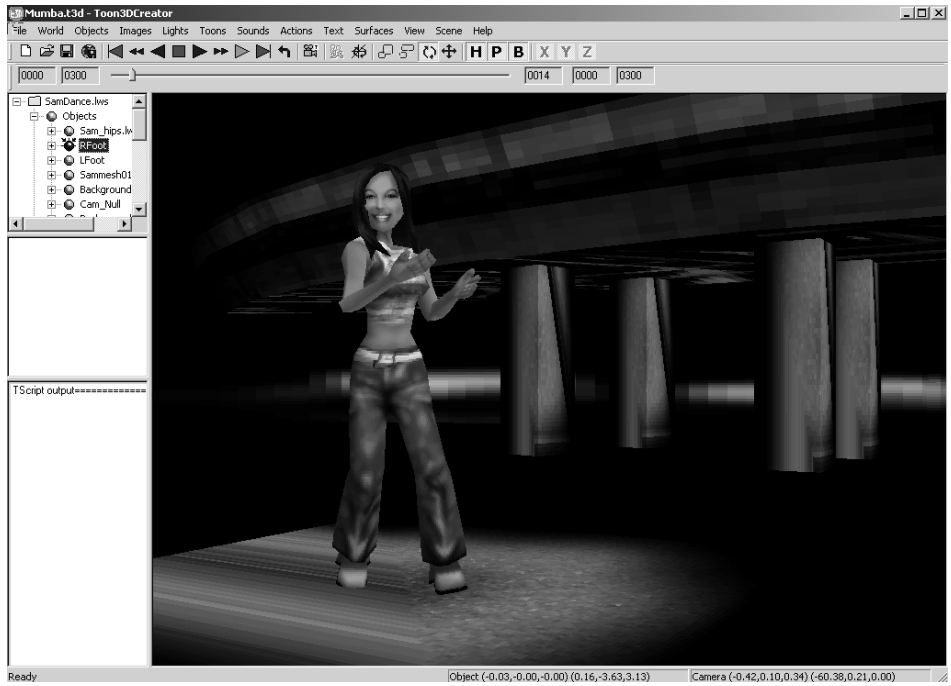
Focal Press is an imprint of the Taylor & Francis Group, an informa business

Notices
Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information, methods, compounds, or experiments described herein.

Library of Congress Cataloguing in Publication Data
A catalogue record for this book is available from the Library of Congress

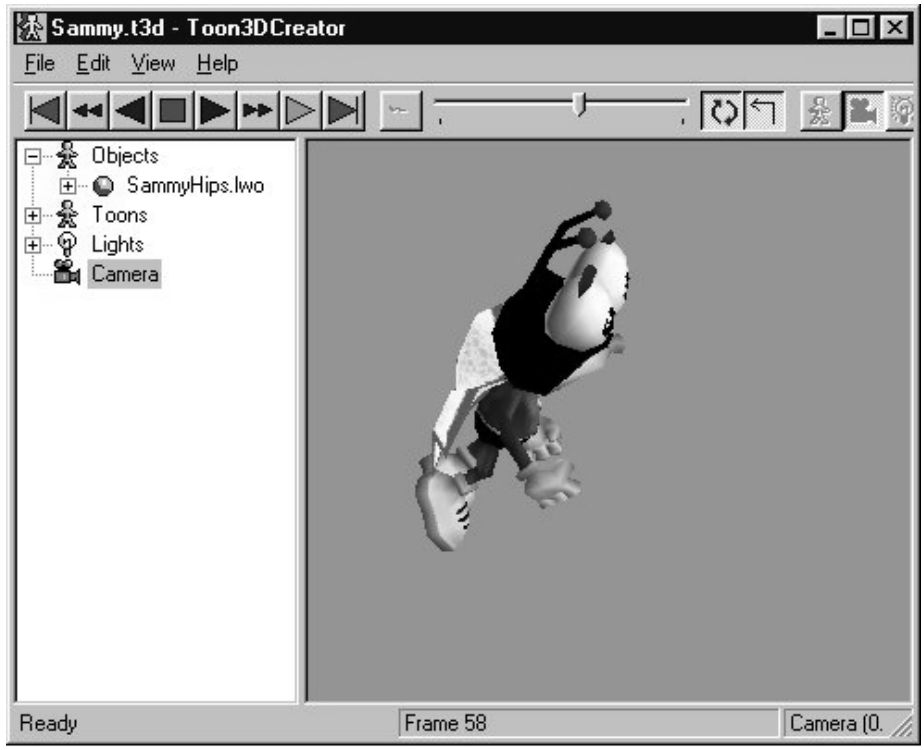
ISBN 13: 978-0-240-51664-6 (pbk)

Contents at a glance



The Toon3D Creator application.

Contents in summary



Animating with Toon3D.

The Toon3D logo.

Supplementary Resources Disclaimer

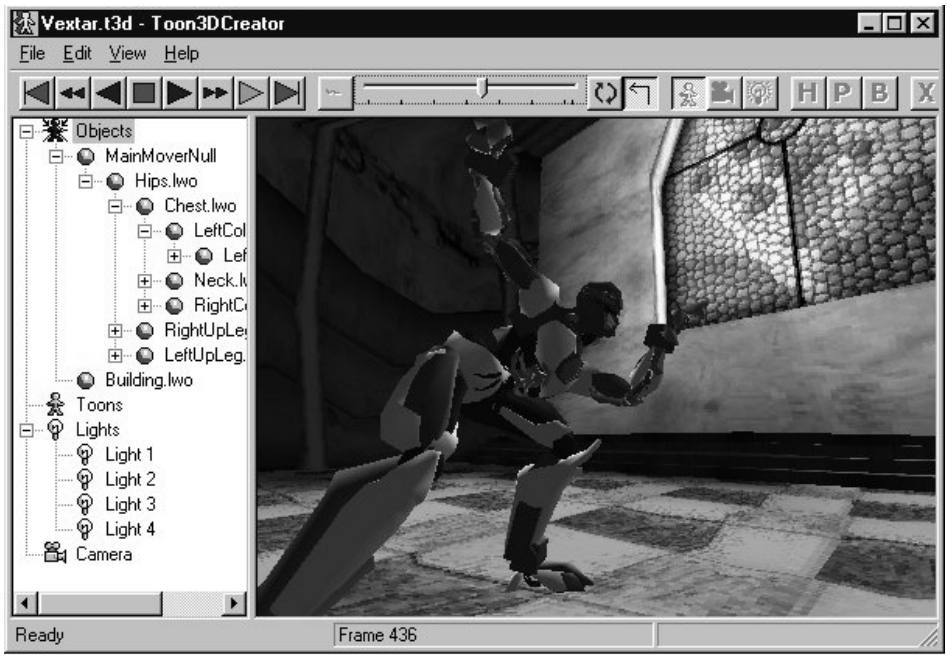
Additional resources were previously made available for this title on CD. However, as CD has become a less accessible format, all resources have been moved to a more convenient online download option.

You can find these resources available here: https://www.routledge.com/9780240516646

Please note: Where this title mentions the associated disc, please use the downloadable resources instead.

About the author

The author has been programming for about 20 years. Professionally, he started out as a drawn animator. These days he spends most of his time programming, but occasionally gets his pencil out. He is married with two children, one of whom spends far too long glued to his PS1!! He lives high in the Pennines in England and tries to get out sailing when it's not raining, which means he spends most of his time playing with computers, because it rains a lot in England.



Using the Toon3D application.

Introduction

Who should read this book?

To get the best from this book, you need some experience with C and a reasonable knowledge of C++. It does not attempt to teach the basics of C/C++ programming. If you are new to programming then I recommend getting a good introduction to C++ programming, particularly Visual C++.

If you have ever looked at a PC or Playstation game with characters running and leaping through an exciting landscape and wondered how it was done, then you should read this book. You may be a hobby programmer, a student or a professional.

Hobby programmer

The book takes you on an exciting adventure. From the basics of 3D manipulation to morph objects and subdivision. On the way, you get Visual C++ project files to load and software that runs on the Windows desktop. You get a full-featured development environment for 3D character animation, so even if you find the maths and the code hard to follow, you can still create games to impress the kids. The game engine even has an ActiveX control that allows you to distribute your work on the Internet.

Student

The computer games industry has become an important employer, always looking for new talent. After reading this book you will be ready to create the sample programs that will get you that first job. You will be guided through the maths and the principal ideas involved in displaying

complex moving characters. You will get an insight into the artist's problems in keeping the characters interesting while not exhausting the game engine.

Professional

You need to display characters in architectural walkthroughs or you may want to add this level of sophistication to multimedia kiosks that you produce. Maybe you use Director and want to add 3D support via an ActiveX control. If you are a web developer then you will find the chapter on streaming and compression particularly useful.

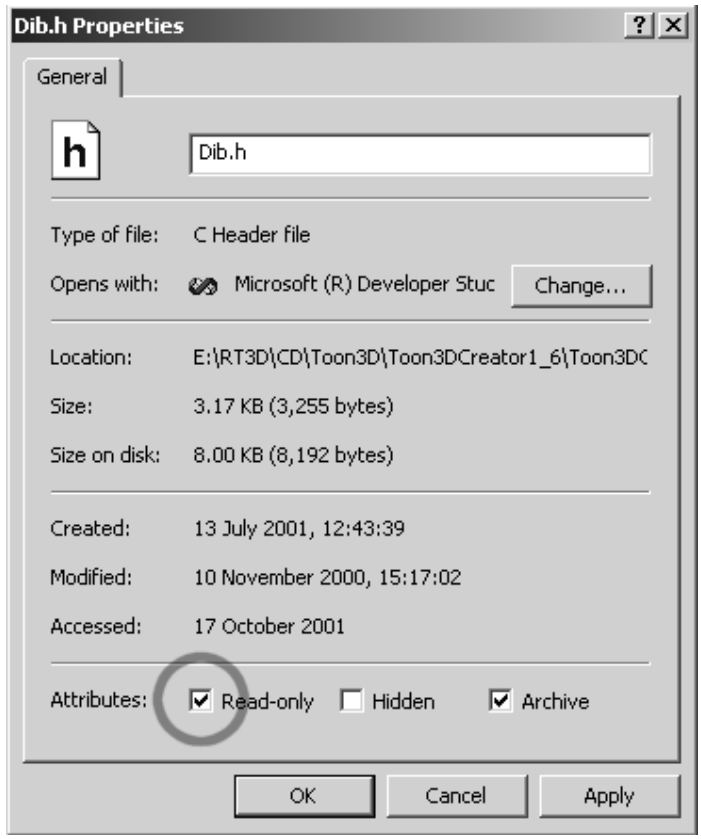
Using the CD

Most of the chapters have example programs to help illustrate the concepts described. These example programs provide you with source code to get you started with your own programs. The CD has two folders, Examples and Toon3D.

Inside the Examples folder you will find folders labelled Chapterxx, where xx is the chapter number. To find the examples for the chapter you are reading simply look in the appropriate Chapter folder. Many of the examples use Microsoft Foundation Classes (MFC). When programming with Visual C++, MFC is a common approach. You will find a brief introduction to MFC in Appendix B; if you have never used MFC then I recommend reading this and perhaps getting one of the many introductory MFC books.

The Toon3D folder contains all the source code for a Web3D application. Toon3D allows you to develop in Lightwave 3D or Max and import the geometry, surface data and animation into Toon3D. In the application you can add interactive behaviour and then publish the material suitable for the Internet. Toon3D is mentioned throughout the book because it is used to illustrate some concepts; the application is also in the Toon3D folder along with some content to play about with. Toon3D is explained in detail in Appendix A.

There is no installation program on the CD. If you want to use an example then copy it to your hard drive and remember to change the file attributes from Read-only. In Explorer you can do this by selecting the files and right clicking; in the pop-up menu select Properties and uncheck the Read-only check box.



Altering file attributes using Windows Explorer.

Typographic conventions

All code examples are set out using Courier New:

```
BOOL CMyClass::CodeExample(CString str){
        CString tmp;

        if (str.Find("code example")!=-1) return FALSE;
        tmp.Format("The string you passed was %s", str);
        AfxMessageBox(tmp);

        Return TRUE;
}
```

All C++ classes are prefixed with the letter C. When variables or function names are used in the text they are italicized; for example,

CodeExample uses the parameter *str*. I prefer not to use the *m_* to indicate a member variable in a class. Additionally, I do not use Hungarian notation to indicate the variable type. Source code style is a matter of heated debate but I prefer *name* rather than *m_szName*. Variables are all lower case and function names use capital letters to indicate the beginning of a word, for example *CodeExample*.

How much maths do I need?

3D computer graphics uses a lot of maths, there is no denying it. In this book I have kept the maths to a minimum. You will need basic school maths up to trigonometric functions, inverse trigonometric functions and algebra. When concepts are introduced they are explained fully and if you find some of the later chapters confusing in their use of the trig functions then I recommend reading Chapter 1 again, where the concepts are explained more fully. You will not find any proofs in this book. If you want to find why a particular technique for using a curve works rather than taking it on trust, then I suggest you look at Appendix C. Appendix C provides a list of alternative sources of information if you want to delve deeper into a particular topic.

All vertex transformations are done with the processor in the sample code. This helps illustrate what is going on, but it does mean that the accelerated graphics card that includes transformations is not being used to its best effect. Once you are familiar with the techniques you may choose to let the hardware look after transformations, leaving the processor to look after the logic.

Credits

The development of 3D graphics has been a combined effort by many people. In the text I explain most techniques with no clear indication of who should be given the credit for developing the technique in the first place. Appendix C on further information makes some attempt to give the credit to the individuals who devised the techniques and also to those who have provided much needed assistance to fledgling developers in the 3D industry.

Contacting the author

I hope you enjoy the book and find it useful. If you do then send me an email at nik@toon3d.com, if you don't then send me an email at nik@anywhere-else.com; just kidding, I would like to hear your views good and bad.

1 3D basics

In this chapter we are going to introduce the 3D basics. We will look at how to store the information required for a computer to display a 3D object. In addition, we will consider the maths required to manipulate this object in 3D space and then convert this to a 2D display. We need a sufficiently general scheme that will allow us to store and manipulate the data that can be displayed as a box, a teapot or an action hero. The method generally used is to store a list of points and a list of polygons. Throughout this book, all the source code is designed to handle polygons with three or four sides.

In later chapters we will leave most low-level operations to a graphics library, which will manage most of the mathematical manipulation. In this book we use the graphics library, OpenGL. But to ease the creation of seamless mesh characters, we will need to do some of our own manipulation of point data; to understand how this code operates you will need to follow the methods outlined in this chapter.

OpenGL is the most widely adopted graphics standard

From the OpenGL website www.opengl.org

'OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.'

Describing 3D space

First let's imagine a small box lying on the floor of a simple room (Figure 1.1).

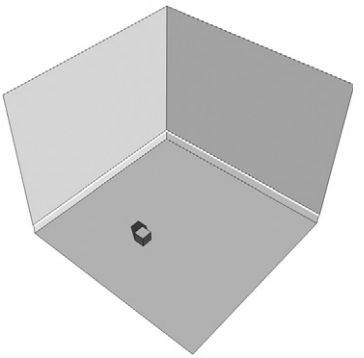


Figure 1.1 A simplified room showing a small box.

How can we create a dataset that describes the position of the box? One method is to use a tape measure to find out the distance of the box from each wall. But which wall? We need to have a frame of reference to work from.

Figure 1.2 shows the same room, only this time there are three perpendicular axes overlaid on the picture. The point where the three axes meet is called the origin. The use of these three axes allows you as a programmer to specify any position in the room using three numerical values.

In Figure 1.2, the two marked lines perpendicular to the axes give an indication of the scale we intend to use. Each slash on these lines represents 10 cm. Counting the slashes gives the box as 6 along the *x*-axis and 8 along the *z*-axis. The box is lying on the floor, so the value along the *y*-axis is 0. To define the position of the box with respect to the frame of reference we use a *vector*,

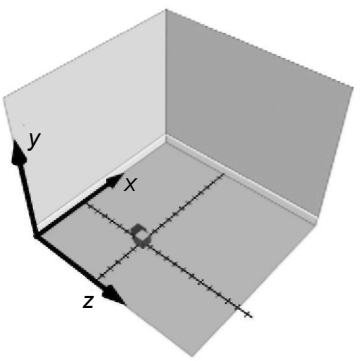


Figure 1.2 A simplified room with overlaid axes.

[6, 0, 8]

In this book, all vectors are of the form [*x*, *y*, *z*].

The direction of the axes is the scheme used throughout this book. The *y*-axis points up, the *x*-axis points to the right and the *z*-axis points out of the screen. We use this scheme because it is the same as that used by the OpenGL graphics library.

Transforming the box

To move the box around the room we can create a vector that gives the distance in the *x*, *y* and *z* directions that you intend to move the box. That

is, if we want to move the box 60 cm to the right, 30 cm up and 20 cm towards the back wall, then we can use the vector [6, 3, 2] (recall that the scale for each dash is 10 cm) to move the box. The sum of two vectors is the sum of the components.

$$[x, y, z] = [x1, y1, z1] + [x2, y2, z2]$$

where $x = x1 + x2$, $y = y1 + y2$ and $z = z1 + z2$

For example, $[12, 3, 10] = [6, 0, 8] + [6, 3, 2]$

Describing an object

The simplest shape that has some volume has just four points or *vertices*. A tetrahedron is a pyramid with a triangular base. We can extend the idea of a point in 3D space to define the four vertices needed to describe a tetrahedron. Before we can draw an object we also need to define how to join the vertices. This leads to two lists: a list of vertices and a list of faces or *polygons*.

The vertices used are:

A: [0.0, 1.7, 0.0]
B: [–1.0, 0.0, 0.6]
C: [0.0, 0.0, –1.1]
D: [1.0, 0.0, 0.6]

To describe the faces we give a list of the vertices that the face shares:

1: A,B,D
2: A,D,C
3: A,C,B
4: B,C,D

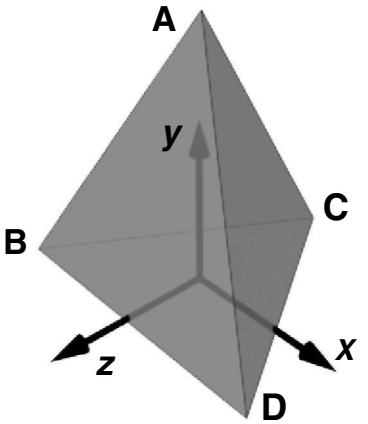


Figure 1.3 A tetrahedron.

Although the triangles ABD and ADB appear to be the same, the order of the vertices is clearly different. This ordering is used by many computer graphics applications to determine whether a face is pointing towards the viewer or away from the viewer. Some schemes use points described in a clockwise direction to indicate that this face is pointing towards the viewer. Other schemes choose counter-clockwise to indicate forward-facing polygons. In this book we used counter-clockwise. There are no advantages or disadvantages to either scheme, it is simply necessary to

be consistent. Think about the triangle ABD as the tetrahedron rotates about the *y*-axis. If this rotation is clockwise when viewed from above then the vertex B moves right and the vertex D moves left. At a certain stage the line BD is vertical. If the rotation continues then B is to the right of D. At this stage in the rotation the face ABD is pointing away from the viewer. Since we know that the order of the vertices read in a counter-clockwise direction should be ABD, when the order changes to ADB, the triangle has turned away from the viewer. This is very useful because in most situations it is possible to effectively disregard this polygon. (If an object is transparent then it will be necessary to continue to render back-facing polygons.) We will look at other techniques to determine back-facing polygons, but vertex order is always the most efficient to compute.

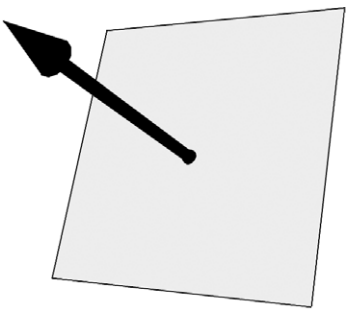


Figure 1.4 A polygon normal.

Polygon normals

A normal is simply a vector that points directly out from a polygon. It is used in computer graphics for determining lighting levels, amongst other things. For the software accompanying this book we store the normal for every polygon in a scene. We have already seen how to deal with the sum of two vectors. The method is easily extended to allow us to subtract two vectors:

$$[x, y, z] = [x1, y1, z1] - [x2, y2, z2]$$
$$= [x1 - x2, y1 - y2, z1 - z2]$$

For example, $[6, 0, 8] - [6, 3, 2] = [6 - 6, 0 - 3, 8 - 2]$
$$= [0, -3, 6]$$

But what happens when we choose to multiply two vectors. In fact, there are two methods of 'multiplying' vectors. One is referred to as the *dot product*. This is defined as

$$\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos(\theta) \text{ where } 0 \leq \theta \leq 180^\circ$$

The symbol |**a**| refers to the magnitude of the vector **a**, which is defined as:

$$|\mathbf{a}| = \sqrt{(x*x + y*y + z*z)}$$

This is a method of measuring the length of the vector. It is a 3D version of the famous theorem of Pythagoras that gives the length of the hypotenuse of a right-angled triangle from the two other sides.

For example, if **a** = [6, 3, 2], then:

$$\begin{aligned} |a| &= \sqrt{(6*6 + 3*3 + 2*2)} \\ &= \sqrt{(36 + 9 + 4)} \\ &= \sqrt{49} = 7 \end{aligned}$$

The dot product is a scalar; this simply means it is a number with a single component not a vector. Given two vectors $\mathbf{a} = [a_x, a_y, a_z]$ and $\mathbf{b} = [b_x, b_y, b_z]$, the dot product is given by

$$\mathbf{a} \bullet \mathbf{b} = a_x \times b_x + a_y \times b_y + a_z \times b_z$$

The dot product is very useful for finding angles between vectors. Since we know that

$$\mathbf{a} \bullet \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos \theta$$

This implies that

$$\frac{\mathbf{a} \bullet \mathbf{b}}{|\mathbf{a}||\mathbf{b}|} = \cos \theta$$

Now we can calculate *cos* θ directly. We can then use the inverse function of *cos*, *acos*, to calculate the value of θ. Here is a code snippet that will pump out the angle between two vectors.

```
double angleBetweenVectors(VECTOR &v1, VECTOR &v2){
    double s,dot,mag1,mag2;
    //Calculate the magnitude of the two supplied vectors
    mag1=sqrt(v1.x*v1.x + v1.y*v1.y + v1.z*v1.z);
    mag2=sqrt(v2.x*v2.x + v2.y*v2.y + v2.z*v2.z);
    //Calculate the sum of the two magnitudes
    s=mag1 * mag2;
    //Avoid a division by zero
    if (s==0.0) s=0.00001;
    dot=v1.x*v2.x + v1.y*v2.y + v1.z*v2.z;
    //Cos theta is dot/s. Therefore theta=acos(dot/s)
    return acos(dot/s);
}
```

The alternative technique for 'multiplying' vectors is the *cross product*. This method creates a further vector that is at right angles or *orthogonal* to the two vectors used in the cross product. Unlike the dot product the operation is not commutative. This simply means that

$\boldsymbol{A} \times \boldsymbol{B}$ does not necessarily equal $\boldsymbol{B} \times \boldsymbol{A}$. Whereas $\boldsymbol{A} \bullet \boldsymbol{B} = \boldsymbol{B} \bullet \boldsymbol{A}$

The cross product of two 3D vectors is given by

$$A \times B = [Ay*Bz - Az*By,\ Az*Bx - Ax*Bz,\ Ax*By - Ay*Bx]$$

This is easier to remember if we look at the pattern for calculating determinants. Determinants are important scalar values associated with square matrices. The determinant of a 1×1 matrix $[a]$ is simply a. If A is a 2×2 matrix then the determinant is given by

$$\boldsymbol{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix}, \qquad \det \boldsymbol{A} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} = ad - bc$$

That is the diagonal top left, bottom right minus top right, bottom left. When extended to 3×3 matrices we have:

$$\boldsymbol{A} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}, \qquad \det \boldsymbol{A} = a \begin{bmatrix} e & f \\ h & i \end{bmatrix} - b \begin{bmatrix} d & f \\ g & i \end{bmatrix} + c \begin{bmatrix} d & e \\ g & h \end{bmatrix}$$

$$= a(ei - fh) - b(di - fg) + c(dh - eg)$$

Here we take the top row one at a time and multiply it by the determinant of the remaining two rows, excluding the column used in the top row. The only thing to bear in mind is that the middle term has a minus sign. If we apply this to the vectors A and B we get

$$\boldsymbol{A} = \begin{bmatrix} x & y & z \\ Ax & Ay & Az \\ Bx & By & Bz \end{bmatrix} \quad \det \boldsymbol{A} = x \begin{bmatrix} Ay & Az \\ By & Bz \end{bmatrix} - y \begin{bmatrix} Ax & Az \\ By & Bz \end{bmatrix} + z \begin{bmatrix} Ax & Ay \\ Bx & By \end{bmatrix}$$

$$= x(AyBz - AzBy) - y(AxBz - AzBx) + z(AxBy - AyBx)$$

$$= x(AyBz - AzBy) + y(AzBx - AxBz) + z(AxBy - AyBx)$$

The x, y and z terms are then found from the determinants of the matrix $\boldsymbol{A}$.

The purpose of all this vector manipulation is that, given three vertices that are distinct and define a polygon, we can find a vector that extends at right angles from this polygon. Given vertices A, B and C we can create two vectors. ***N*** is the vector from B to A and ***M*** is the vector from B to C. Simply subtracting B from A and B from C respectively creates these vectors. Now the cross product of the vectors ***N*** and ***M*** is the normal of the polygon. It is usual to scale this normal to unit length. Dividing each of the terms by the magnitude of the vector achieves this.

Rotating the box

There are many options available when rotating a 3D representation of an object; we will consider the three principal ones. The first option we will look at uses Euler angles.

Euler angles

When considering this representation it is useful to imagine an aeroplane flying through the sky. Its direction is given by its heading. The slope of the flight path is described using an angle we shall call pitch and the orientation of each wing can be described using another angle which we shall call bank. The orientation can be completely given using these three angles. Heading gives the rotation about the *y*-axis, pitch gives rotation about the *x*-axis and bank gives rotation about the *z*-axis.

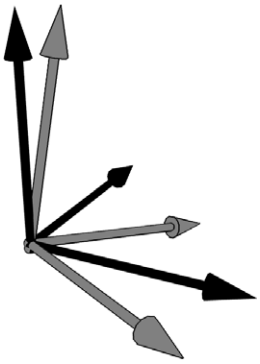


Figure 1.5 Euler angle rotation.

To describe the orientation of an object we store an angle for the heading, the pitch and the bank. Assuming that the rotation occurs about the point [0, 0, 0] as the box is modelled then heading is given from the 3 $\times$ 3 matrix:

$$\boldsymbol{H} = \begin{bmatrix} \cos(h) & 0 & \sin(h) \\ 0 & 1 & 0 \\ -\sin(h) & 0 & \cos(h) \end{bmatrix}$$

Rotation in the pitch is given by:

$$\boldsymbol{P} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(p) & -\sin(p) \\ 0 & \sin(p) & \cos(p) \end{bmatrix}$$

and bank rotation is given by:

$$\boldsymbol{B} = \begin{bmatrix} \cos(b) & \sin(b) & 0 \\ -\sin(b) & \cos(b) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Combining columns with rows as follows is another form of matrix multiplication:

$$\begin{bmatrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{bmatrix} \begin{bmatrix} A\ B\ C \\ D\ E\ F \\ G\ H\ I \end{bmatrix} = \begin{bmatrix} Aa + Db + Gc & Ba + Eb + Hc & Ca + Fb + Ic \\ Ad + De + Gf & Bd + Ee + Hf & Cd + Fe + If \\ Ag + Dh + Gi & Bg + Eh + Hi & Cg + Fh + Ii \end{bmatrix}$$

Using this method we can combine the ***H***, ***P*** and ***B*** rotation matrices:

$$\mathbf{HPB} = \begin{bmatrix} \cos(h)\cos(b) - \sin(h)\sin(p)\sin(b) & \cos(h)\sin(b) + \sin(h)\sin(p)\cos(b) & \sin(h)\cos(p) \\ -\cos(p)\sin(p) & \cos(p)\cos(b) & -\sin(p) \\ -\sin(h)\cos(b) - \cos(h)\sin(p)\sin(b) & -\sin(h)\sin(b) + \cos(h)\sin(p)\cos(b) & \cos(h)\cos(p) \end{bmatrix}$$

Matrix multiplication is non-commutative, so ***HPB***, ***HBP***, ***PHB***, ***PBH***, ***BHP*** and ***BPH*** all give different results.

Now, to translate the object vertices to world space we multiply all the vertices as vectors by the rotation matrix above. Vector and matrix multiplication is done in this way:

$$\boldsymbol{R} = \begin{bmatrix} a\ b\ c \\ d\ e\ f \\ g\ h\ i \end{bmatrix} \quad \boldsymbol{v} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \boldsymbol{Rv} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

So the vertex (x, y, z) maps to the vertex ($ax + by + cz$, $dx + ey + fz$, $gx + hy + iz$). If the object also moves in the 3D world by $\boldsymbol{T} = (tx, ty, tz)$, then the new position of the vertex should include this mapping. That is, the vertex maps to $\boldsymbol{Rv} + \boldsymbol{T}$, giving the world location

$$(x, y, z) \rightarrow (ax + by + cz + tx, dx + ey + fz + ty, gx + hy + iz + tz)$$

Euler angle rotation suffers from a problem that is commonly called gimbal lock. This problem arises when one axis is mapped to another by a rotation. Suppose that the heading rotates through 90°, then the *x*- and *z*-axes become aligned to each other. Now pitch and bank are occurring along the same axis. Whenever one rotation results in a mapping of one axis to another, one degree of freedom is lost. To avoid this problem, let's consider another way of describing rotations which uses four values.

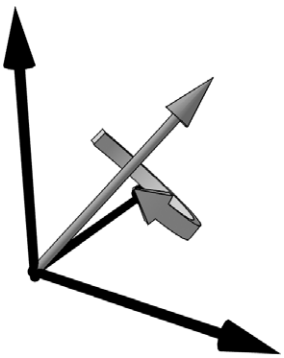


Figure 1.6 Angle and axis rotation.

Angle and axis rotation

The values used are an angle θ and a vector $A = [x, y, z]^T$ that represents the axis of rotation. When the orientation of the box is described in this way the rotation matrix is given by:

$$\boldsymbol{R} = \begin{bmatrix} 1 + (-z^2 - y^2)(1 - \cos(\theta)) & -z\sin(\theta) + yx(1 - \cos(\theta)) & y\sin(\theta) + zx(1 - \cos(\theta)) \\ z\sin(\theta) + yx(1 - \cos(\theta)) & 1 + (-z^2 - x^2)(1 - \cos(\theta)) & -x\sin(\theta) + zy(1 - \cos(\theta)) \\ -y\sin(\theta) + zx(1 - \cos(\theta)) & x\sin(\theta) + zy(1 - \cos(\theta)) & 1 + (-y^2 - z^2)(1 - \cos(\theta)) \end{bmatrix}$$

We can use this rotation matrix in the same way as described for Euler angles to map vertices in the object to a 3D world space location.

Quaternion rotation

Yet another way to consider an object's orientation uses quaternions. Devised by W. R. Hamilton in the eighteenth century, quarternions are used extensively in games because they provide a quick way to interpolate between orientations. A quaternion uses four values. One value is a scalar quantity *w*, and the remaining three values are combined into a vector $v = (x, y, z)$. When using quaternions for rotations they must be unit quaternions.

If we have a quaternion $q = w + x + y + z = [w, v]$, then:

The norm of a quaternion is $N(q) = w^2 + x^2 + y^2 + z^2 = 1$
A unit quaternion has $N(q) = 1$

The conjugate of a quaternion is $q^* = [w, -v]$
The inverse is $q^{-1} = q^*/N(q)$. Therefore, for unit quaternions the inverse is the same as the conjugate.
Addition and subtraction involves $q_0 \pm q_1 = [w_0 + w_1, v_0 + v_1]$
Multiplication is given by $q_0 q_1 = [w_0 w_1 - v_0 v_1, v_0 \times v_1 + w_0 v_1 + w_1 v_0]$; this operation is non-commutative, i.e. $q_0 q_1$ is not the same as $q_1 q_0$.
The identity for quaternions depends on the operation; it is [1, **0**] (where **0** is a zero vector (0, 0, 0)) for multiplication and [0, **0**] for addition and subtraction.

Rotation involves $v' = qvq^*$, where $v = [0, v]$.
Turning a unit quaternion into a rotation matrix results in

$$R = \begin{bmatrix} 1 - 2y^2 - 2x^2 & 2xy + 2wz & 2xz - 2wy \\ 2xy - 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx \\ 2xz + 2wy & 2yz - 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

We will consider the uses of quaternions for smooth interpolation of camera orientation and techniques for converting quickly between the different representations of rotation in Chapter 8.

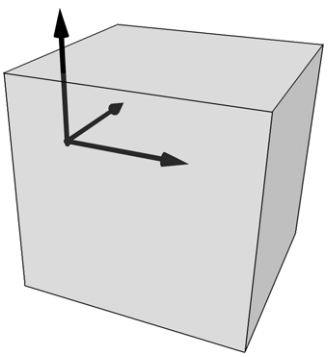


Figure 1.7 Rotation about a pivot point.

Rotation about a point other than the origin

To rotate about an arbitrary point, which in many CGI applications is called the *pivot point*, involves first translating a vertex to the origin, doing the rotation then translating it back. If the vertex $[1, 1, 1]^T$ were rotated about the point (2, 0, 0), then we want to consider the point (2, 0, 0) to be the origin. By subtracting (2, 0, 0) from $[1, 1, 1]^T$ we can now rotate as though this is the origin then add (2, 0, 0) back to the rotated vertex.

Scaling the object

The size of the object has so far been unaffected by the operations considered. If we want to scale the object up or down we can use another

matrix. The scaling in the *x*-axis is *Sx*, scaling in the *y*-axis is *Sy* and scaling in the *z*-axis is *Sz*. This results in the matrix

$$\boldsymbol{S} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & Sz \end{bmatrix}$$

Scaling should be applied before any other operations. We can concatenate our rotation matrix to ensure that scaling occurs first. If *R* is our rotation matrix from either Euler angles or from the angle/axis method, then the matrix becomes:

$$\boldsymbol{R} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad \boldsymbol{S} = \begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & Sz \end{bmatrix} \quad \boldsymbol{RS} = \begin{bmatrix} aSx & bSy & cSz \\ dSx & eSy & fSz \\ gSx & nSy & iSz \end{bmatrix}$$

The full operation to translate a vertex in the object to a location in world space including pivot point consideration becomes

RS(**v** − **p**) + **t** + **p**, where **R** is the rotation matrix, **S** the scaling matrix, **v** is the vertex, **p** is the pivot point and **t** is the translation vector.

For every vertex in a solid object, **t** + **p** and **RS** will be the same. Pre-calculating these will therefore speed up the transformation operations. It is highly likely that the pivot point of an object will remain constant throughout an animation, so the object could be stored already transformed to its pivot point. If this is the case then the equation becomes

RSv + **t**

So now we can move and rotate our box. We are now ready to transfer this to the screen.

Perspective transforms

Converting 3D world space geometry to a 2D screen is surprisingly easy. Essentially we divide the *x* and *y* terms by *z* to get screen locations (*sx*, *sy*). The technique uses similar triangles to derive the new value for (*sx*, *sy*) from the world coordinates. Referring to Figure 1.8, here we indicate the position of the camera, the screen and the object. Following the vertex *P* to the image of this on the screen at *P′*, we get two similar triangles, *CPP•z* and *CP′d*, where *d* is the distance from the camera to the screen. We want to know the position of *P′*:

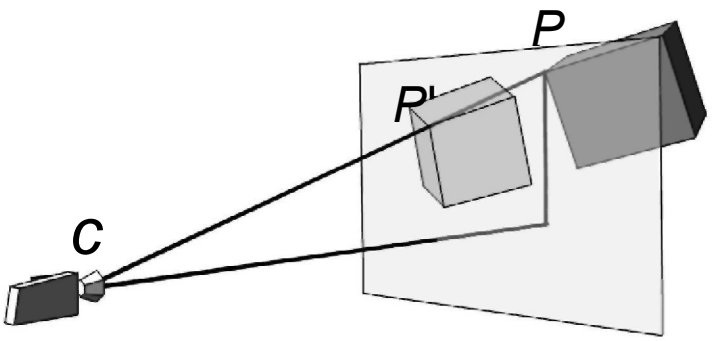


Figure 1.8 Perspective transform.

$$(Px - Cx)/(Pz - Cz) = (P'x - Cx)/d$$

$$(Py - Cy)/(Pz - Cz) = (P'y - Cy)/d$$

which can be rearranged to become

$$P'x = ((Px - Cx)*d)/(Pz - Cz) + Cx$$

$$P'y = ((Py - Cy)*d)/(Pz - Cz) + Cy$$

The value for *d*, the distance from the camera to the screen, should be of the order of twice the pixel width of the 3D display window to avoid serious distortion.

The above equations assume that the centre of the display window is (0, 0) and that *y* values increase going up the screen. If (0, 0) for the display window is actually in the top left corner, then the *y* values should be subtracted from the height of the display window and half the width and height if the display is added to the result.

$$sx = ((Px - Cx)*d)/(Pz - Cz) + Cx + \text{screen width}/2$$

$$sy = \text{screen height}/2 - (((Py - Cy)*d)/(Pz - Cz) + Cy)$$

Using 4 × 4 matrix representations

Although rotation and scaling of an object can be achieved using 3 × 3 matrices, translation cannot be included. To get around this problem it is usual to add a row and column to the matrix. We move from

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

to

$$\begin{bmatrix} a & b & c & 0 \\ d & e & f & 0 \\ g & h & i & 0 \\ tx & ty & tz & 1 \end{bmatrix}$$

where (tx, ty, tz) is the translation in the x-, y- and z-axes respectively.

This technique requires us to add a component to the vector representation of a vertex. Now a vertex is defined as $[x, y, z, 1]^T$. Such coordinates are often referred to as homogeneous coordinates. The matrix can now include the perspective transform that converts world coordinates into the 2D screen coordinates that the viewer ultimately sees. By concatenating the above matrix with a matrix that achieves this perspective transform, all the calculations necessary to take a vertex from model space through world space to camera space and finally to screen space can be achieved by a single matrix.

Summary

The basic operations presented here will act as building blocks as we develop the character animation engine. To get the most out of this book, you need to be confident of the use of vectors, matrix multiplication and the simple algebra manipulation we used in this chapter. I have tried to present the material in a form that is suitable for those who are unfamiliar with mathematical set texts. If the reader wishes to explore the mathematics presented in this chapter in more depth, then please check Appendix C, where further references are mentioned.

2 Drawing points and polygons the hard way

Some people like to climb mountains, others prefer to fly over them sipping a chilled wine. If you are a climber then this chapter is for you. Most of this book uses the OpenGL library, which originated on the SGI platform and is now available for Windows, Mac and Unix boxes. The advantage of using such a library is that it shields much of the complexity of displaying the 3D characters we create. Another very definite benefit is that the library takes advantage of any hardware the user may have installed. The disadvantage to the climbers is that we have no understanding of how the display is actually generated at the individual pixel level. This chapter takes you through this process; if you are a climber then read on, if you are a flyer then feel free to skip this chapter, no one will ever know!

Creating memory for a background display

In this chapter we are trying to avoid using Windows-specific code wherever possible. For this reason we use a class library that deals with a memory-based *bitmap*. This class library, which is supplied as part of the sample code for this chapter on the CD, is called CCanvas. CCanvas has a constructor that can be supplied with a width and a height in pixels, together with the bit depth.

A colour can be specified in many ways. Generally you will need a red value, a green value and a blue value. Many applications allow for 256 levels of red, 256 levels of green and 256 levels of blue. Zero to 255 is the range of values available in 1 byte. One byte contains 8 bits of information. Hence with 8 bits for red, 8 bits for green and 8 bits for blue, we have a 24-bit colour value, 8 + 8 + 8.

When colour is specified using 3 bytes in this book, it is called an RGB value. If we define the colour value for an array of pixels then we can display this as a bitmap. If the value for each pixel were the same, then

the display would simply show a flat coloured rectangle. If we carefully choose the value for each pixel then we can display a photograph. If the range of colours in the displayed bitmap is limited then we could choose to store the bitmap as a combination of all the different colours used, followed by a list of where to use these colours. This type of display is called a palletized display and we are not supporting palletized displays in this book. Usually, palletized displays are limited to 256 colours. Creating an optimized palette for each frame of animation is time consuming. The alternative is to use a master palette, which has definite restrictions on the way that the display can handle lighting. Imagine the simplest of scenes with three bouncing balls all lit from above. Ball A is red, B is blue and C is green. If the master palette has about 80 levels of red, green and blue, then 240 slots in the palette have been used. Now in comes a purple, yellow and orange cube. Somehow, this has to be displayed using the remaining 16 colours; the results, while acceptable on desktop computer platforms 10 years ago, simply do not cut it by today's standards.

Another type of display uses 16 bits for the colour value of each pixel. This gives just 32 levels of red, 32 levels of green and 32 levels of blue. This standard is often used in computer games, resulting in faster frame rates with most hardware than 24-bit displays. A 32-bit display can use the additional 8 bits for alpha or stencil use or it can be used by the display driver to ensure all colour calculations use 32-bit integers. This results in fewer instructions for the processor to handle and consequently faster image transfers.

The code for the creation of the buffer is:

```
// Create a new empty Canvas with specified bitdepth
BOOL CCanvas::Create(int width, int height, int bitdepth)
{
    // Delete any existing stuff.
    if (m_bits)) delete m_bits;
    // Allocate memory for the bits (DWORD aligned).
    if (bitdepth==16) m_swidth =width*2;
    if (bitdepth==24) m_swidth =width*3;
    if (bitdepth==32) m_swidth =width*4;
    m_swidth =( m_swidth + 3) & ~3;
    m_size= m_swidth *height;

    m_bits = new BYTE[m_size];
    if (!m_bits) {
        TRACE("Out of memory for bits");
        return FALSE;
```

```
    }
    // Set all the bits to a known state (black).
    memset(m_bits, 0, m_size);
    m_width=width; m_height=height; m_bitdepth=bitdepth;
    CreateBMI();
    return TRUE;
}
```

Two things to notice here. First, the code to ensure that our line widths are exact multiples of 4.

```
m_swidth =( m_swidth + 3) & ~3;
```

We do this by adding 3 to the storage width and then bitwise And-ing this with the bitwise complement of 3. A bitwise complement has every bit in the number inverted. If we take an example, suppose that the width of the bitmap is 34 pixels and it is stored as a 24-bit image. A 24-bit image uses 3 bytes of information for each pixel, so if the storage width was simply the width times 3 then it would be 102. However, 102 is not a multiple of 4. We need to find the next multiple of 4 greater than 102. Three as a byte wide binary value is 00000011. The bitwise complement is 11111100. The algorithm adds 3 to the storage width, making it 105. Now 105 as a binary value is 01101001; note here that one of the lowest 2 bits is set, which means it cannot be a multiple of 4. 01101001 And 11111100 = 01101000, which is 104. This is divisible by 4 as required. The effect of the operation is to clear the last 2 bits of the number. This kind of alignment is used regularly in such buffers because it allows a pixel location in memory to be found with fewer instructions. The memory variable, m_swidth, holds the storage width of a single line and m_size keeps a check on the buffer size, so that we can easily check for out of bounds memory errors.

The other curiosity is the call to CreateBMI. Our canvas uses a Windows structure called BITMAPINFO, so that ultimately we can display the canvas on the user's screen using a simple Windows API call. A BITMAPINFO contains a BITMAPINFOHEADER and a single RGBQUAD. We are only interested in the BITMAPINFOHEADER, so we cast our member variable to a header to fill in the appropriate details. By keeping it in a function call, this minimizes the changes necessary to port this code to another platform.

```
BOOL CCanvas::CreateBMI(){
    // Clear any existing header.
    If (m_pBMI) delete m_pBMI;
```

```
    // Allocate memory for the new header.
    m_pBMI = new BITMAPINFO;
    if (!m_pBMI) {
        TRACE("Out of memory for header");
        return FALSE;
    }
    // Fill in the header info.
    BITMAPINFOHEADER *bmi=(BITMAPINFOHEADER*)m_pBMI;
    bmi->biSize = sizeof(BITMAPINFOHEADER);
    bmi->biWidth = m_width;
    bmi->biHeight = -m_height;
    bmi->biPlanes = 1;
    bmi->biBitCount = m_bitdepth;
    bmi->biCompression = BI_RGB;
    bmi->biSizeImage = 0;
    bmi->biXPelsPerMeter = 0;
    bmi->biYPelsPerMeter = 0;
    bmi->biClrUsed = 0;
    bmi->biClrImportant = 0;
    Return TRUE;
}
```

Blitting the display to the screen

My rule about not using Windows code falls down again here, since at some stage we need to display the result of our labours. Windows uses device contexts in such operations. This book does not go into any detail in this regard. There are many other books that explain graphics operation for Windows; the Appendix lists some of the author's favourites. We use a simple blit to get our memory-based bitmap onto the screen. Now you will realize why the BITMAPINFO structure was needed.

```
// Draw the DIB to a given DC.
void CCanvas::Draw(CDC *pdc)
{
    ::StretchDIBits(pdc->GetSafeHdc(),
                    0,              // Destination x
                    0,              // Destination y
                    m_width,        // Destination width
                    m_height,       // Destination height
                    0,              // Source x
```

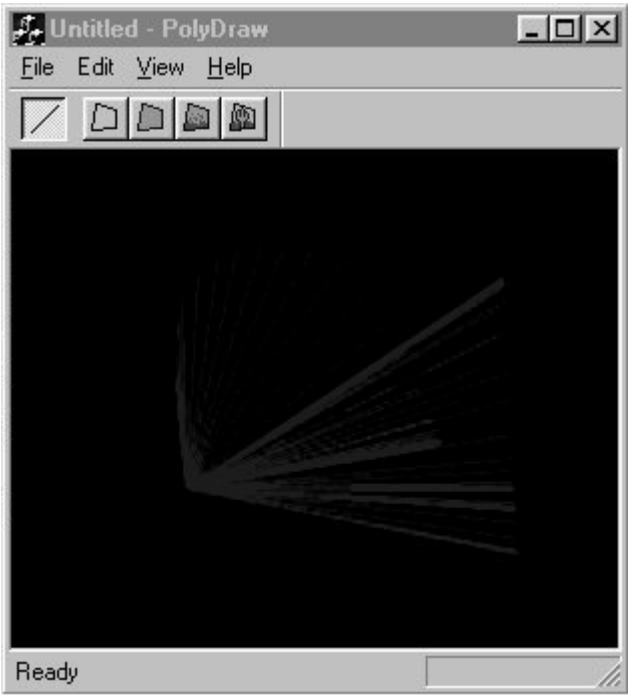


Figure 2.1 PolyDraw Sample drawing lines.

```
                        0,              // Source y
                        m_width,        // Source width
                        m_height,       // Source height
                        m_bits,         // Pointer to bits
                        m_bmi,          // BITMAPINFO
                        DIB_RGB_COLORS, // Options
                        SRCCOPY);       // Raster operation code (ROP)
}
```

So that is it for the Windows stuff. Now we can get down to the actual code.

Drawing a line with Bresenham's algorithm

We now have in memory a BYTE buffer. We can draw on this by setting the RGB values at a particular memory location. For convenience, the class includes a private function GetPixelAddress(x, *y*), which returns a

pointer to the pixel or NULL if it is out of range. The function includes a simple clip test. If either *x* is greater than the bitmap's width or *y* is greater than the bitmap's height, then it is out of range. Similarly, if *x* or *y* are less than 0 then they are out of range. To indicate this fact, the function returns a null pointer. The bitmap is stored in memory one line following another. We know how many bytes are required for a single line; this is the information that we ensured was divisible by 4, the storage width. To access the appropriate line, we simply need to multiply the storage width by the value for *y*. The distance along the line is dependent on whether we are using a 24-bit pixel or a 32-bit pixel. If we are using a 24-bit pixel then we need to multiply the *x* value by the number of bytes in a 24-bit pixel, that is 3. A 32-bit pixel needs 4 bytes for each pixel, hence the *x* value needs to be multiplied by 4. Having calculated the offset from the start of the bitmap in memory, all that remains is to add this to the start of the bitmap 'bits' memory to return the memory location of the pixel (*x*, *y*).

```
BYTE* CCanvas::GetPixelAddress(int x, int y)
{
    // Make sure it's in range and if it isn't return zero.
    if ((x >= m_width)|| (y >= m_height())||(x<0)||(y<0)) return↵
      NULL;
    // Calculate the scan line storage width.
    if (m_bitDepth()==24) x*=3;
    if (m_bitDepth()==32) x*=4;
    return m_pBits +y * m_swidth + x;
}
```

We want to create a function that will draw an arbitrary line that is defined by the starting and ending points. Before we go any further with such a function, we need to ensure that the start and end of the line are actually within the boundaries of the off-screen buffer we are using as a canvas. For this we will create a ClipLine function. The aim of the ClipLine function is to adjust the (*x*, *y*) values of each end of the line so that they are within the canvas area. That is $0 \leq x <$ width, where width is the CCanvas width, and $0 \leq y <$ height, where height is the CCanvas height.

The ClipLine function creates a code value for the start point, *cs*, and the end point, *ce*. This code value determines whether the point is within the canvas area, off to the left, right, above or below, or a combination of these. This is done using the code:

```
cs=((xs<0)<<3)|((xs>=m_width)<<2)|((ys<0)<<1)|(ys>=m_height);
```

Here, *xs* and *ys* are the *x*, *y* values of the starting point for the line. If *xs* is less than 0, then *cs* is given the value 1 shifted three places to the left, which is 8. If *xs* is greater than or equal to the width of the canvas, then 4 is added to the code value. If the *y* value is less than 0, then 2 is added to the code value, and finally if *ys* is greater or equal to the height of the canvas, then 1 is added to the code value. This places the point in one of the number sections of Figure 2.2, the number being the code value for a point in that section. For example, if we have the point (–3, 16) on a canvas that is 200 pixels square, then

```
cs = ((-3<0)<<3)|(-3>=200)<<2)|((16<0)<<1)|(16>=200)
cs = (1<<3)|(0<<2)|(0<<1)|0
cs = 8|0|0|0
cs = 8
```

From the code value, we know that the point (–3, 16) with respect to our canvas is to the left in the section labelled 8 in the diagram.

The next step is to determine the slope of the line. This is done using the *y* distance divided by the *x* distance. The *x* distance is the end *x* value minus the start *x* value. The *y* distance is the end *y* value minus the start *y* value. The slope of this line is a floating point value; since the values for *x* and *y* are all integer values, we must remember to cast the integer values as doubles to get a meaningful result for the slope. Now we have a point location and a slope. By doing a bitwise And-ing of the start and end locations, we determine whether the line remains off-screen throughout its length.

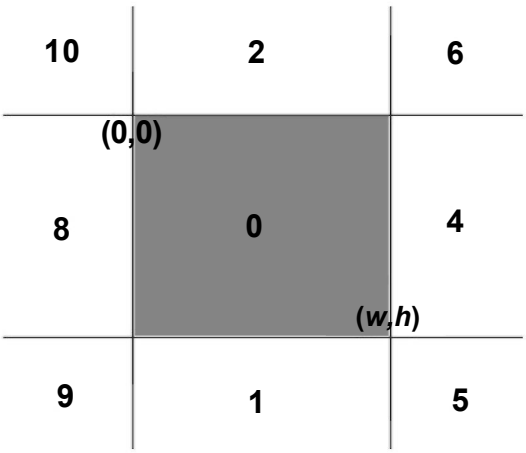


Figure 2.2 Determining point location.

Table 2.1

Codes	0	1	2	4	5	6	8	9	10
0	0	0	0	0	0	0	0	0	0
1	0	1	0	0	1	0	0	1	0
2	0	0	2	0	0	2	0	0	2
4	0	0	0	4	4	4	0	0	0
5	0	1	0	4	5	4	0	1	0
6	0	0	2	4	4	6	0	0	2
8	0	0	0	0	0	0	8	8	8
9	0	1	0	0	1	0	8	9	8
10	0	0	2	0	0	2	8	8	10

A table of values for a bitwise And-ing of the start and end point codes will help in the understanding of the result of this operation (Table 2.1).

Looking at Table 2.1, we can see that if the start point is in section 1 and the end point is in section 5, then the result of a bitwise And-ing is 1; since this is not zero the function returns FALSE, indicating that there is nothing to draw. If the bitwise test results in a zero value then the aim of the remainder of the function is to determine where the line crosses the canvas area and return both a TRUE to indicate that drawing is required and revised values for *xs*, *ys*, *xe* and *ye* that are within the canvas area.

To adjust the start and end points we use the slope or gradient of the line that we have calculated and stored as the variable *m*. If the code value for the point when And-ed with 8 does not equal zero, then the point must be off to the left; in this case we aim to set *x* to 0, but what value should we store for *y*? Here we use the fact that we have added $-x$ to the *y* value, so we must subtract *x* times the slope of the line to *y*. Similar principles are adopted for each off-screen area. Having adjusted the line the function loops, setting the code values for the start and end points. This continues until the point is totally within the canvas area at which point the function exits returning a TRUE value. This clever clipping routine is known as Cohen–Sutherland after its creators.

```
BOOL CCanvas::Clip(int &xs, int &ys, int &xe, int &ye){
    int cs, ce;
    double m;
    //Calculate the slope of the line (xs,ys)-(xe,ye)
    m = (double)(ye-ys)/(double)(xe-xs);
    while (cs|ce){
          cs=((xs<0)<<3)|((xs>=m_width)<<2)|((ys<0)<<1)|↵
            (ys>=m_height);
          ce=((xe<0)<<3)|((xe>=m_width)<<2)|((ye<0)<<1)|↵
            (ye>=m_height);
          if (cs & ce) return FALSE;
          if (cs){
              if (cs & 8) ys-=(int)((double)xs*m), xs=0; else
              if (cs & 4) ys+=(int)((double)(m_width-xs)*m),↵
                xs=m_width-1; else
              if (cs & 2) xs-=(int)((double)ys/m), ys=0; else
              if (cs & 1) xs+=(int)((double)(m_height-ys)/m),↵
                ys=m_height-1;
          }
          if (ce){
              if (ce & 8) ye+=(int)((double)(0-xe)*m), xe=0; else
              if (ce & 4) ye+=(int)((double)(m_width-xe)*m),↵
                xe=m_width-1; else
              if (ce & 2) xe+=(int)((double)(0-ye)/m), ye=0; else
              if (ce & 1) xe+=(int)((double)(m_height-ye)/m),↵
                ye=m_height-1;
          }
    }
    return TRUE;
}
```

Running this function with the values (–10, 100)–(150, 300) for a 200 pixel square canvas gives the following results:

```
Slope is 1.25
Loop 1 cs = 8 ce = 1 Starting point ( -10, 100) End point (150,300)
Loop 2 cs = 0 ce = 0 Starting point (  0, 112) End point (70,199)
```

We now know both the starting and ending points are on-screen. If neither were on-screen then there is nothing to do, so we exit. Assuming we actually have something to draw, we adjust the drawing width to the canvas storage width. Remember a 24-bit file will have a storage width

that is at least three times the pixel width, plus the extra up to 3 bytes to ensure divisibility by 4. We set two variables *x* and *y* to the starting value. Then we create two distance variables for the *x* distance, *dx*, and the *y* distance, *dy*.

We set *xinc* and *yinc* to 1. The value for *xinc* indicates whether *xe* is greater than *xs*. If so, then we will reach *xe* by 1 to *xs*, a *dx* number of times. If, however, *xs* > *xe*, then we must subtract 1 from *xs*, *dx* number of times. We check *dx* to indicate which direction we are working in. If *dx* is less than 0, then the *xinc* is made to be –1 and the value for *dx* is negated. A similar technique is used for the *y* values.

The COLORREF parameter passed to the function is simply a 32-bit integer. The values for red, green and blue are embedded in this value, and we retrieve them with the code:

```
red= col&0xFF;
green=(col&0xFF00)>>8;
blue=(col&0xFF0000)>>16;
```

Now we use a switch statement to select based on bit depth. In our code we only support 24 bits, but it gives us flexibility for the future to support alternative bit depths.

```
void CCanvas::DrawLine(int xs,int ys,int xe,int ye,COLORREF col){
    int x, y, d, dx, dy, c, m, xinc, yinc, width;
    BYTE red,blue,green;

    if (!ClipLine(xs,ys,xe,ye)) return;
    //If ClipLine returns false then start and end are out off the
    //canvas so there is nothing to draw
    //m_swidth is the DWORD aligned storage width
    width=m_swidth;
    //x and y are set to the starting point
    x=xs; y=ys;
    //dx and dy are the distances in the x and y directions
    //respectively
    dx=xe-xs; dy=ye-ys;
    //ptr is the memory location of x,y
    BYTE* ptr=GetPixelAddress(x,y);

    xinc=1; yinc=1;
    if (dx<0){xinc=-1; dx=-dx;}
    if (dy<0){yinc=-1; dy=-dy; width=-width;}
```

```
        red =(BYTE)col&0xFF;
        green=(BYTE)(col&0xFF00)>>8;
        blue =(BYTE)(col&0xFF0000)>>16;

        d=0;
        switch (m_bitdepth){
        case 24:
             if (dy<dx){
                    c=2*dx; m=2*dy;
                    while (x!= xe){
                          *ptr++=blue; *ptr++=green; *ptr++=red; //Set↵
                            the pixel
                          x+=xinc; d+=m;
                          if (xinc<0) ptr-=6;
                          if (d>dx){y+=yinc; d-=c; ptr-=width;}
                    }
             }else{
                    c=2*dy; m=2*dx;
                    while (y!=ye){
                    *ptr++=blue; *ptr++=green; *ptr=red; //Set the↵
                      pixel
                    y+=yinc; d+=m; ptr-=width; ptr-=2;
                    if (d>dy){ x+=xinc;↵
                      d-=c;ptr+=xinc;ptr+=xinc;ptr+=xinc;}
             }
        }
        break;
      }
    }
```

When drawing the line we need to decide whether our principle increment axis is going to be *x* or *y*. The test involves simply testing which is greater, the *y* distance or the *x*. If *x* is greater, then we use the *x*-axis. For every column in the *x*-axis, we need to colour a pixel. If the line were horizontal, then simply incrementing along the *x*-axis would be sufficient. But we also need to find the *y* position. For each column the *y* value can increment by 1 or decrement by 1. The maximum slope of a line where *x* is the principle axis is a 45° slope. For this slope, *y* is incremented for each *x* increment. So the question we need to ask as we move to the next *x* column is: Do we need to alter our *y* value?

To answer this question we use three variables. Double the *x* distance, *c*, double the *y* distance, *m*, and an incremental value *d*. For each *x* value,

we add *m* to the incremental value *d*. If it tips over the value for *dx* then we need to increase *y*. When we increase *y* we decrease the incremental value *d* by *c* and adjust our memory pointer to point to another line. The operation for the *y*-axis works in the same way. So now we can draw arbitrary lines on our memory bitmap using integer arithmetic alone.

A simple class to implement a polygon

Now we have an off-screen buffer in the form of *CCanvas* and the ability to draw an arbitrary line on this buffer defined by two points. The next step is to implement a class to store and display a polygon. This class is defined as:

```
class CPolygon
{
public:
  //Member variables
  POINT3D pts[4];//Stores the vertex data
  CVector normal;//The unit length normal
  int numverts;//Number of vertices in polygon
  double h,p,b;//Euler angle rotation
  COLOUR col;//RGB values for the unshaded colour of the polygon
  BYTE red,green,blue;//Used for all drawing operations
  CTexture *tex;//Bitmap texture pointer
  CPolygon *next;//Used if the polygon is one of a list
  //Functions
  CPolygon();//Constructor
  CPolygon(int total, CVector *pts);//Constructor
  ~CPolygon();//Standard destructor
  BOOL Facing();//True if screen coordinates of vertices
                //are in counter clockwise order
  void SetColour(int red,int green, int blue);
  void AveragePointNormal(CVector &norm);//Define normal from an
                                         //average of the vertex
                                         //normals
  void SetNormal();//Calculates normal from vertex positions
  void HorzLine(CCanvas &canvas, int x1, int x2,
        int y, double light=1.0, double ambient=0.0);
  BOOL SetPoints(int total, CVector *pts);//Set the vertex values
  void SetColour(COLOUR col);//Set the colour value
  void SetTexture(CString &filename);//Set texture from bitmap
                                     //filename
```

```
  void DrawOutline(CCanvas &buffer);//Draw outline version of
                                    //polygon
  void DrawFlat(CCanvas &buffer);//Draw flat coloured version
  void DrawShaded(CCanvas &buffer, BOOL drawNormal=FALSE);↵
      //Draw shaded
  void DrawTextured(CCanvas &buffer);//Draw textured polygon
protected:
};
```

The POINT3D class is a structure defined as:

```
typedef struct stPOINT3D{
  double x,y,z;//Untransfromed position
  double nx,ny,nz;//Untransformed normal
  double wx,wy,wz;//Transformed position
  double wnx,wny,wnz;//Transformed normal
  int sx,sy;//Screen location
  int snx,sny;//Normal screen location
}POINT3D;
```

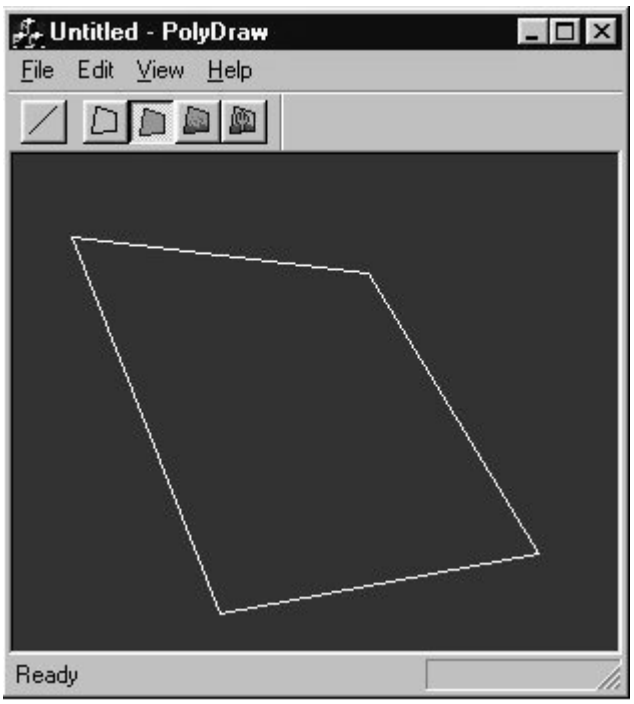


Figure 2.3 An outline polygon.

Drawing an outline polygon

We can use this new class to draw an outline polygon simply by connecting all the points in the polygon. Once the polygon has been transformed to screen coordinates using the techniques from the previous chapter, this involves just this simple code. Note that, once a polygon has been transformed, the screen coordinates are stored in the *sx* and *sy* members of the POINT3D structure points.

```
void CPolygon::DrawOutline(CCanvas &buf){
    if (numverts<3) return;
    for (int i=0;i<numverts-1;i++){
        buf.DrawLine(pts[i].sx, pts[i].sy, pts[i+1].sx,↵
          pts[i+1].sy,col);
    }
    //Finally join the last point to the first
    buf.DrawLine(pts[numverts-1].sx, pts[numverts-1].sy,↵
      pts[0].sx, pts[0].sy,col);
}
```

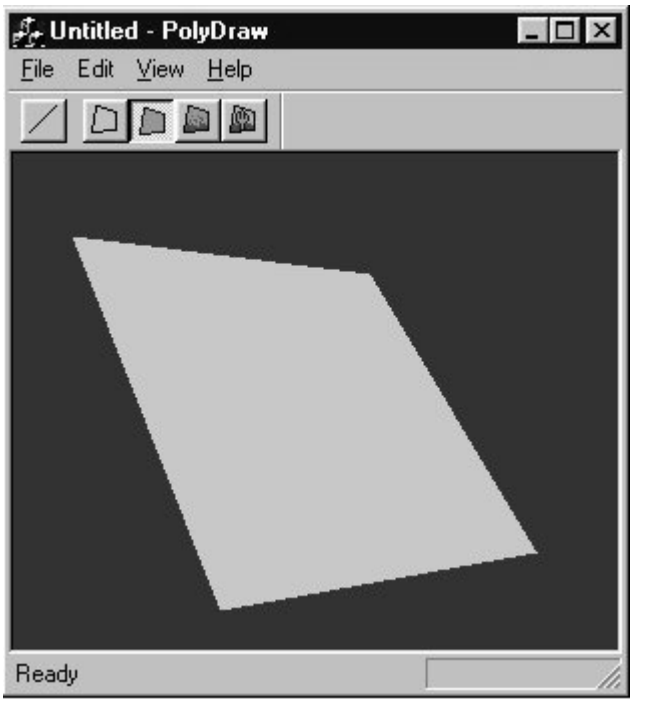


Figure 2.4 A flat coloured polygon.

Drawing a flat coloured polygon

In order to paint a filled polygon we need to raster scan the polygon. That is, we need to break up the polygon into horizontal lines and draw each of these in turn. To keep things simple we will work only with triangles. If the polygon has four sides then we draw one half first and then the next. A triangle can sometimes be orientated so that one of its sides is horizontal, but an arbitrary triangle can be split into two triangles, each with one horizontal side.

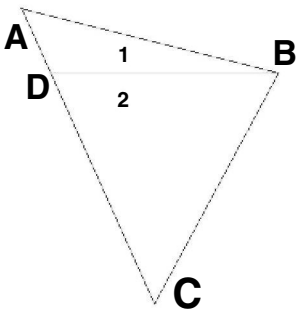


Figure 2.5 Dividing a triangle into two, each with a horizontal line.

To determine the starting point of each horizontal line in the traingle we need the slope of all three lines. In our code we first order the points by height; to create variables *y*[max], *y*[mid] and *y*[min]. The slope of each side is the *y* distance divided by the *x* distance. Using this information, we can determine the start and end points of each horizontal line and draw the line for each *y* value. The starting point for each horizontal line is given by the starting point for this slope, the slope of the line and the current *y* value.

If we look at an example then it will be clearer. The vertices of the triangle in Figure 2.5 are

A(28, 16) B(268, 76) C(150, 291)

Taking the line AC, we can calculate point D using the following technique. First determine the slope of line AC.

The slope of the line is the *y* distance divided by the *x* distance:

(291 – 16)/(150 – 28) = 275/122 = 2.254

Now we want the *x* value when the *y* value on this line is 76, i.e. the *y* value for vertex B. A line is defined as

$$y = mx + c$$

where *m* is the slope and *c* is a constant.

Since we know that the vertex (28, 16) is on the line, we can calculate *c* as

$$c = y - mx = 16 - 2.254 \times 28 = -47.112$$

We also know that the vertex (150, 291) is on this line, so a quick check gives

$$y = mx + c = 2.254 \times 150 - 47.112 = 290.98$$

which rounded up is the 291 *y* value of this vertex.

Having calculated this constant, we can rearrange the equation for a line to derive the *x* value:

$$x = (y - c)/m$$

For our line we know that *y* is 76 and the slope is 2.254, so the *x* value on the line AC when *y* is 76 is

$$(76 - (-47.112))/2.254 = 54.62$$

So the point D is (54.62, 76).

When we draw the triangle, we use the same technique that we have used to determine the point D to determine the horizontal values for the start and end of each horizontal line. To calculate the end points of each horizontal line in the triangle ABD we will also need to know the slope and constant value for the line AB. Using this information, we know the start and end *x* values for each integer *y* value in the triangle ABD. Having drawn the upper triangle ABD, we go on to draw the lower triangle DBC. For this triangle we need to know the slope and constant value for the line BC. We can then go on to draw each horizontal line in the triangle DBC.

One end of the line will be the slope from *y*[min] to *y*[max] and the other end will change when the *y* value reaches the mid value. The code works through each section in turn.

```
void CPolygon::DrawFlat(CCanvas &buf){
    //This function will draw a triangle to the off screen buffer
    //class CCanvas
    //Only 24 bit supported a the moment
    if (buf.GetBitDepth()!=24) return;
    //The polygon is facing away from camera
    if (!Facing()) return;
    int count,min,max,mid,i;
    int x[c],y[c];
    double m1, d1, m2, d2, q, u, v;
    count=1;
    //Set the values for red, green and blue directly. No shading
    red=col.red; green=col.green; blue=col.blue;

    while(count<(numverts-1)){
        x[0]=pts[0].sx; x[1]=pts[count].sx; x[2]=pts[count+1].sx;
        y[0]=pts[0].sy; y[1]=pts[count].sy; y[2]=pts[count+1].sy;
        //Sort points by height
        max=(y[0]<y[1])?1:0;
        max=(y[max]<y[2])?2:max;
        min=(y[0]<y[1])?0:1;
        min=(y[min]<y[2])?min:2;
        mid=3-(max+min);
        //x distance
        q=(double)(x[max]-x[min]);
        //Avoid division by zero
        q=(q)?q:EPSILON;
        m2=(double)(y[max]-y[min])/q;
        d2=y[max]-m2*x[max];
        //Now we know the highest. middle and lowesty positions
        //Draw horizontal lines from highest to middle position
        if (y[max]!=y[min]){
            q=(double)(x[mid]-x[max]);
            q=(q)?q:EPSILON;
            m1=(double)(y[mid]-y[max])/q;
            d1=(double)y[mid]-m1*x[mid];
            for (i=y[max];i>y[mid];i-){
                u=((double)i-d1)/m1; v=((double)i-d2)/m2;
                HorzLine(buf, (int)u, (int)v, i);
            }
        }
        //Reached the mid point
```

```
        if (y[mid]!=y[min]){
            q=(double)(x[mid]-x[min]);
            q=(q)?q:EPSILON;
            m1=(double)(y[mid]-y[min])/q;
            d1=(double)y[mid]-m1*x[mid];
            for (i=y[mid];i>y[min];i-){
                u=((double)i-d1)/m1; v=((double)i-d2)/m2;
                HorzLine(buf, (int)u, (int)v, i);
            }
        }
        count++;
    }
}
```

The function to draw a horizontal line is quite simple. We do some simple clipping to ensure we stay within memory, get a memory pointer and paint the pixels one after the other. Notice the class *CPolygon* has colour values red, green and blue defined by member variables.

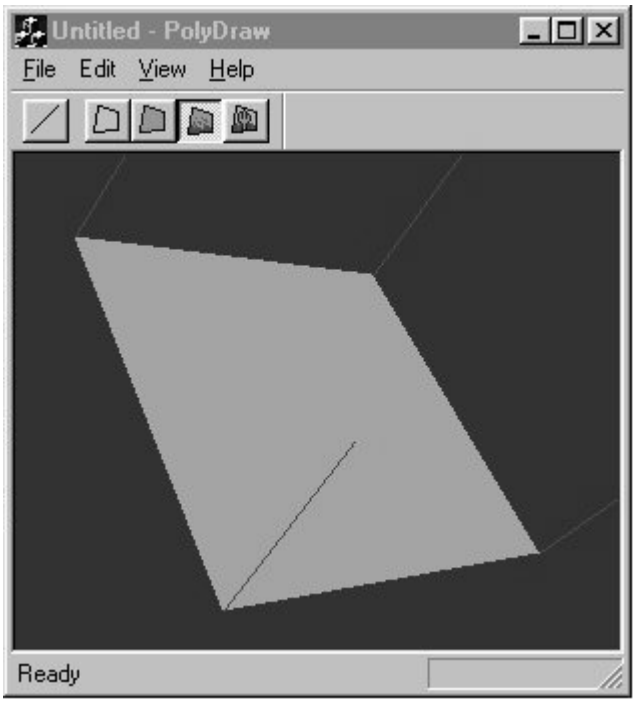


Figure 2.6 Drawing a shaded polygon.