

Alan Watt

Third Edition

3D Computer Graphics



ADDISON-WESLEY





3D Computer Graphics

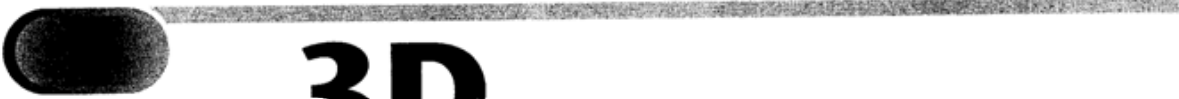


We work with leading authors to develop the strongest educational materials in computer science, bringing cutting-edge thinking and best learning practice to a global market.

Under a range of well-known imprints, including Addison-Wesley, we craft high quality print and electronic publications which help readers to understand and apply their content, whether studying or at work.

To find out about the complete range of our publishing please visit us on the World Wide Web at:


<http://www.pearsoneduc.com>



3D Computer Graphics

THIRD EDITION

ALAN WATT

 **ADDISON-WESLEY**

An imprint of PEARSON EDUCATION

Harlow, England · London · New York · Reading, Massachusetts · San Francisco · Toronto · Don Mills, Ontario · Sydney
Tokyo · Singapore · Hong Kong · Seoul · Taipei · Cape Town · Madrid · Mexico City · Amsterdam · Munich · Paris · Milan

Pearson Education Limited
Edinburgh Gate
Harlow
Essex CM20 2JE
England

and Associated Companies throughout the world

Visit us on the World Wide Web at:
www.pearsoneduc.com

First published 1989
Second edition 1993
This edition first published 2000

© 1989, 1993 Addison-Wesley Publishing Ltd, Addison-Wesley Publishing Company Inc.
© Pearson Education Limited 2000

This right of Alan Watt to be identified as author of
this work has been asserted by him in accordance with
the Copyright, Designs, and Patents Act 1988.

All rights reserved; no part of this publication may be reproduced, stored
in a retrieval system, or transmitted in any form or by any means, electronic,
mechanical, photocopying, recording, or otherwise without either the prior
written permission of the Publishers or a licence permitting restricted copying
in the United Kingdom issued by the Copyright Licensing Agency Ltd,
90 Tottenham Court Road, London W1T 4LP.

The programs in this book have been included for their instructional value.
The publisher does not offer any warranties or representation in respect of their
fitness for a particular purpose, nor does the publisher accept any liability for any
loss or damage (other than for personal injury or death) arising from their use.

Many of the designations used by manufacturers and sellers to distinguish their
products are claimed as trademarks. Pearson Education Limited has made every
attempt to supply trademark information about manufacturers and their products
mentioned in this book. A list of trademark designations and their owners
appears on page xxii.

ISBN 0 201 39855 9

British Library Cataloguing-in-Publication Data
A catalogue record for this book can be obtained from the British Library.

Library of Congress Cataloguing-in-Publication Data
Available from the publisher.

Typeset by 42
Printed and bound in The United States of America

10 9 8 7
07 06 05 04 03

Para Dionéa
a garota de Copacabana

Contents

Colour plates appear between pages 506 and 507

<i>Preface</i>	xvi
<i>Acknowledgements</i>	xxi
1 Mathematical fundamentals of computer graphics	1
1.1 Manipulating three-dimensional structures	1
1.1.1 Three-dimensional geometry in computer graphics – affine transformations	2
1.1.2 Transformations for changing coordinate systems	8
1.2 Structure-deforming transformations	9
1.3 Vectors and computer graphics	11
1.3.1 Addition of vectors	12
1.3.2 Length of vectors	12
1.3.3 Normal vectors and cross products	12
1.3.4 Normal vectors and dot products	14
1.3.5 Vectors associated with the normal vector reflection	15
1.4 Rays and computer graphics	17
1.4.1 Ray geometry – intersections	17
1.4.2 Intersections – ray–sphere	18
1.4.3 Intersections – ray–convex polygon	19
1.4.4 Intersections – ray–box	21
1.4.5 Intersections – ray–quadric	23
1.4.6 Ray tracing geometry – reflection and refraction	23
1.5 Interpolating properties in the image plane	25
2 Representation and modelling of three-dimensional objects (1)	27
Introduction	27
2.1 Polygonal representation of three-dimensional objects	33
2.1.1 Creating polygonal objects	37

2.1.2	Manual modelling of polygonal objects	38
2.1.3	Automatic generation of polygonal objects	38
2.1.4	Mathematical generation of polygonal objects	39
2.1.5	Procedural polygon mesh objects – fractal objects	44
2.2	Constructive solid geometry (CSG) representation of objects	46
2.3	Space subdivision techniques for object representation	51
2.3.1	Octrees and polygons	53
2.3.2	BSP trees	55
2.3.3	Creating voxel objects	56
2.4	Representing objects with implicit functions	56
2.5	Scene management and object representation	58
2.5.1	Polygon mesh optimization	59
2.6	Summary	64
3	Representation and modelling of three-dimensional objects (2)	66
	Introduction	66
3.1	Bézier curves	69
3.1.1	Joining Bézier curve segments	75
3.1.2	Summary of Bézier curve properties	77
3.2	B-spline representation	78
3.2.1	B-spline curves	78
3.2.2	Uniform B-splines	80
3.2.3	Non-uniform B-splines	84
3.2.4	Summary of B-spline curve properties	90
3.3	Rational curves	90
3.3.1	Rational Bézier curves	91
3.3.2	NURBS	93
3.4	From curves to surfaces	94
3.4.1	Continuity and Bézier patches	98
3.4.2	A Bézier patch object – the Utah teapot	100
3.5	B-spline surface patches	101
3.6	Modelling or creating patch surfaces	106
3.6.1	Cross-sectional or linear axis design example	107
3.6.2	Control polyhedron design – basic technique	110
3.6.3	Creating patch objects by surface fitting	115
3.7	From patches to objects	121
4	Representation and rendering	123
	Introduction	123
4.1	Rendering polygon meshes – a brief overview	124

4.2	Rendering parametric surfaces	125
4.2.1	Rendering directly from the patch descriptions	125
4.2.2	Patch to polygon conversion	128
4.2.3	Object space subdivision	128
4.2.4	Image space subdivision	135
4.3	Rendering a CSG description	138
4.4	Rendering a voxel description	140
4.5	Rendering implicit functions	141
5	The graphics pipeline (1): geometric operations	142
	Introduction	142
5.1	Coordinate spaces in the graphics pipeline	143
5.1.1	Local or modelling coordinate systems	143
5.1.2	World coordinate systems	143
5.1.3	Camera or eye or view coordinate system	143
5.2	Operations carried out in view space	147
5.2.1	Culling or back-face elimination	147
5.2.2	The view volume	147
5.2.3	Three-dimensional screen space	149
5.2.4	View volume and depth	152
5.3	Advanced viewing systems (PHIGS and GKS)	156
5.3.1	Overview of the PHIGS viewing system	157
5.3.2	The view orientation parameters	159
5.3.3	The view mapping parameters	159
5.3.4	The view plane in more detail	162
5.3.5	Implementing a PHIGS-type viewing system	164
6	The graphics pipeline (2): rendering or algorithmic processes	167
	Introduction	167
6.1	Clipping polygons against the view volume	168
6.2	Shading pixels	171
6.2.1	Local reflection models	173
6.2.2	Local reflection models – practical points	177
6.2.3	Local reflection models – light source considerations	179
6.3	Interpolative shading techniques	179
6.3.1	Interpolative shading techniques – Gouraud shading	180
6.3.2	Interpolative shading techniques – Phong shading	181
6.3.3	Renderer shading options	182
6.3.4	Comparison of Gouraud and Phong shading	183
6.4	Rasterization	183
6.4.1	Rasterizing edges	183
6.4.2	Rasterizing polygons	185

6.5	Order of rendering	187
6.6	Hidden surface removal	189
6.6.1	The Z-buffer algorithm	189
6.6.2	Z-buffer and CSG representation	190
6.6.3	Z-buffer and compositing	191
6.6.4	Z-buffer and rendering	192
6.6.5	Scan line Z-buffer	193
6.6.6	Spanning hidden surface removal	193
6.6.7	A spanning scan line algorithm	194
6.6.8	Z-buffer and complex scenes	196
6.6.9	Z-buffer summary	198
6.6.10	BSP trees and hidden surface removal	199
6.7	Multi-pass rendering and accumulation buffers	202
7	Simulating light-object interaction: local reflection models	205
	Introduction	205
7.1	Reflection from a perfect surface	206
7.2	Reflection from an imperfect surface	207
7.3	The bi-directional reflectance distribution function	208
7.4	Diffuse and specular components	211
7.5	Perfect diffuse – empirically spread specular reflection	212
7.6	Physically based specular reflection	213
7.6.1	Modelling the micro-geometry of the surface	214
7.6.2	Shadowing and masking effects	214
7.6.3	Viewing geometry	216
7.6.4	The Fresnel term	216
7.7	Pre-computing BRDFs	219
7.8	Physically based diffuse component	221
8	Mapping techniques	223
	Introduction	223
8.1	Two-dimensional texture maps to polygon mesh objects	228
8.1.1	Inverse mapping by bilinear interpolation	229
8.1.2	Inverse mapping by using an intermediate surface	230
8.2	Two-dimensional texture domain to bi-cubic parametric patch objects	234
8.3	Billboards	235
8.4	Bump mapping	236
8.4.1	A multi-pass technique for bump mapping	238
8.4.2	A pre-calculation technique for bump mapping	239

8.5	Light maps	240
8.6	Environment or reflection mapping	243
8.6.1	Cubic mapping	245
8.6.2	Sphere mapping	247
8.6.3	Environment mapping: comparative points	248
8.6.4	Surface properties and environment mapping	249
8.7	Three-dimensional texture domain techniques	251
8.7.1	Three-dimensional noise	251
8.7.2	Simulating turbulence	252
8.7.3	Three-dimensional texture and animation	254
8.7.4	Three-dimensional light maps	256
8.8	Anti-aliasing and texture mapping	256
8.9	Interactive techniques in texture mapping	260
9	Geometric shadows	263
	Introduction	263
9.1	Properties of shadows used in computer graphics	265
9.2	Simple shadows on a ground plane	265
9.3	Shadow algorithms	267
9.3.1	Shadow algorithms: projecting polygons/scan line	267
9.3.2	Shadow algorithms: shadow volumes	268
9.3.3	Shadow algorithms: derivation of shadow polygons from light source transformations	271
9.3.4	Shadow algorithms: shadow Z-buffer	271
10	Global illumination	275
	Introduction	275
10.1	Global illumination models	276
10.1.1	The rendering equation	277
10.1.2	Radiance, irradiance and the radiance equation	278
10.1.3	Path notation	281
10.2	The evolution of global illumination algorithms	283
10.3	Established algorithms – ray tracing and radiosity	284
10.3.1	Whitted ray tracing	284
10.3.2	Radiosity	286
10.4	Monte Carlo techniques in global illumination	288
10.5	Path tracing	292
10.6	Distributed ray tracing	294
10.7	Two-pass ray tracing	297
10.8	View dependence/independence and multi-pass methods	300

10.9	Caching illumination	301
10.10	Light volumes	303
10.11	Particle tracing and density estimation	304
11	The radiosity method	306
	Introduction	306
11.1	Radiosity theory	308
11.2	Form factor determination	310
11.3	The Gauss–Seidel method	314
11.4	Seeing a partial solution – progressive refinement	315
11.5	Problems with the radiosity method	318
11.6	Artefacts in radiosity images	319
11.6.1	Hemicube artefacts	319
11.6.2	Reconstruction artefacts	321
11.6.3	Meshing artefacts	323
11.7	Meshing strategies	325
11.7.1	Adaptive or <i>a posteriori</i> meshing	325
11.7.2	<i>A priori</i> meshing	332
12	Ray tracing strategies	342
	Introduction – Whitted ray tracing	342
12.1	The basic algorithm	343
12.1.1	Tracing rays – initial considerations	343
12.1.2	Lighting model components	344
12.1.3	Shadows	345
12.1.4	Hidden surface removal	346
12.2	Using recursion to implement ray tracing	347
12.3	The adventures of seven rays – a ray tracing study	350
12.4	Ray tracing polygon objects – interpolation of a normal at an intersection point in a polygon	352
12.5	Efficiency measures in ray tracing	354
12.5.1	Adaptive depth control	354
12.5.2	First hit speed up	355
12.5.3	Bounding objects with simple shapes	355
12.5.4	Secondary data structures	357
12.5.5	Ray space subdivision	363
12.6	The use of ray coherence	364
12.7	A historical digression – the optics of the rainbow	367

13	Volume rendering	370
	Introduction	370
	13.1 Volume rendering and the visualization of volume data	373
	13.2 'Semi-transparent gel' option	377
	13.2.1 Voxel classification	378
	13.2.2 Transforming into the viewing direction	379
	13.2.3 Compositing pixels along a ray	379
	13.3 Semi-transparent gel plus surfaces	380
	13.3.1 Explicit extraction of isosurfaces	382
	13.4 Structural considerations in volume rendering algorithms	384
	13.4.1 Ray casting (untransformed data)	385
	13.4.2 Ray casting (transformed data)	387
	13.4.3 Voxel projection method	388
	13.5 Perspective projection in volume rendering	390
	13.6 Three-dimensional texture and volume rendering	391
14	Anti-aliasing theory and practice	392
	Introduction	392
	14.1 Aliases and sampling	393
	14.2 Jagged edges	397
	14.3 Sampling in computer graphics compared with sampling reality	398
	14.4 Sampling and reconstruction	400
	14.5 A simple comparison	401
	14.6 Pre-filtering methods	402
	14.7 Supersampling or post-filtering	404
	14.8 Non-uniform sampling – some theoretical concepts	406
	14.9 The Fourier transform of images	411
15	Colour and computer graphics	418
	Introduction	418
	15.1 Colour sets in computer imagery	419
	15.2 Colour and three-dimensional space	420
	15.2.1 RGB space	423
	15.2.2 The HSV single hexcone model	424
	15.2.3 YIQ space	427
	15.3 Colour, information and perceptual spaces	427
	15.3.1 CIE XYZ space	429
	15.3.2 CIE xyY space	433
	15.4 Rendering and colour spaces	435

15.5 Monitor considerations	436
15.5.1 RGB _{monitor} space and other monitor considerations	436
15.5.2 Monitor considerations – different monitors and the same colour	437
15.5.3 Monitor considerations – colour gamut mapping	439
15.5.4 Monitor considerations – gamma correction	440
16 Image-based rendering and photo-modelling	443
Introduction	443
16.1 Reuse of previously rendered imagery – two-dimensional techniques	444
16.1.1 Planar impostors or sprites	445
16.1.2 Calculating the validity of planar impostors	445
16.2 Varying rendering resources	447
16.2.1 Priority rendering	447
16.2.2 Image layering	448
16.3 Using depth information	452
16.3.1 Three-dimensional warping	452
16.3.2 Layered depth images (LDIs)	456
16.4 View interpolation	458
16.4.1 View morphing	460
16.5 Four-dimensional techniques – the Lumigraph or light field rendering approach	463
16.6 Photo-modelling and IBR	465
16.6.1 Image-based rendering using photographic panoramas	469
16.6.2 Compositing panoramas	469
16.6.3 Photo-modelling for image-based rendering	470
17 Computer animation	473
Introduction	473
17.1 A categorization and description of computer animation techniques	476
17.2 Rigid body animation	477
17.2.1 Interpolation or keyframing	477
17.2.2 Explicit scripting	479
17.2.3 Interpolation of rotation	483
17.2.4 Using quaternions to represent rotation	484
17.2.5 Interpolating quaternions	488
17.2.6 The camera as an animated object	492
17.3 Linked structures and hierarchical motion	493
17.3.1 Solving the inverse kinematics problem	500

17.4 Dynamics in computer animation	504
17.4.1 Basic theory for a rigid body – particles	505
17.4.2 The nature of forces	506
17.4.3 Rigid bodies – extended masses	507
17.4.4 Using dynamics in computer animation	510
17.4.5 Simulating the dynamics of a lumped mass	511
17.4.6 Space–time constraints	515
17.5 Collision detection	517
17.5.1 Broad phase/narrow phase algorithms	518
17.5.2 Broad phase collision detection with OBBs	519
17.5.3 Narrow phase: pairs of convex polyhedra – exact collision detection	522
17.5.4 Single phase algorithms – object hierarchies	524
17.6 Collision response	526
17.7 Particle animation	529
17.8 Behavioural animation	531
17.9 Summary	534
18 Comparative image study	536
Introduction	536
18.1 Local reflection models	537
18.2 Texture and shadow mapping	538
18.3 Whitted ray tracing	539
18.4 Radiosity	541
18.5 RADIANCE	543
18.6 Summary	543
References	544
Index	553



Preface

This is the third edition of a book that deals with the processes involved in converting a mathematical or geometric description of an object – a computer graphics model – into a visualization – a two-dimensional projection – that simulates the appearance of a real object. The analogy of a synthetic camera is often used and this is a good allusion provided we bear in mind certain important limitations that are not usually available in a computer graphics camera (depth of field and motion blur are two examples) and certain computer graphics facilities that do not appear in a camera (near and far clipping planes).

Algorithms in computer graphics mostly function in a three-dimensional domain and the creations in this space are then mapped into a two-dimensional display or image plane at a late stage in the overall process. Traditionally computer graphics has created pictures by starting with a very detailed geometric description, subjecting this to a series of transformations that orient a viewer and objects in three-dimensional space, then imitating reality by making the objects look solid and real – a process known as rendering. In the early 1980s there was a coming together of research – carried out in the 1970s into reflection models, hidden surface removal and the like – that resulted in the emergence of a *de facto* approach to image synthesis of solid objects. But now this is proving insufficient for the new demands of moving computer imagery and virtual reality and much research is being carried out into how to model complex objects, where the nature and shape of the object changes dynamically and into capturing the richness of the world without having to explicitly model every detail. Such efforts are resulting in diverse synthesis methods and modelling methods but at the moment there has been no emergence of new image generation techniques that rival the pseudo-standard way of modelling and rendering solid objects – a method that has been established since the mid-1970s.

So where did it all begin? Most of the development in computer graphics as we know it today was motivated by hardware evolution and the availability of new devices. Software rapidly developed to use the image producing hardware. In this respect the most important development is the so-called raster display, a device that proliferated in the mass market shortly after the development of the PC. In this device the complete image is stored in a memory variously called a

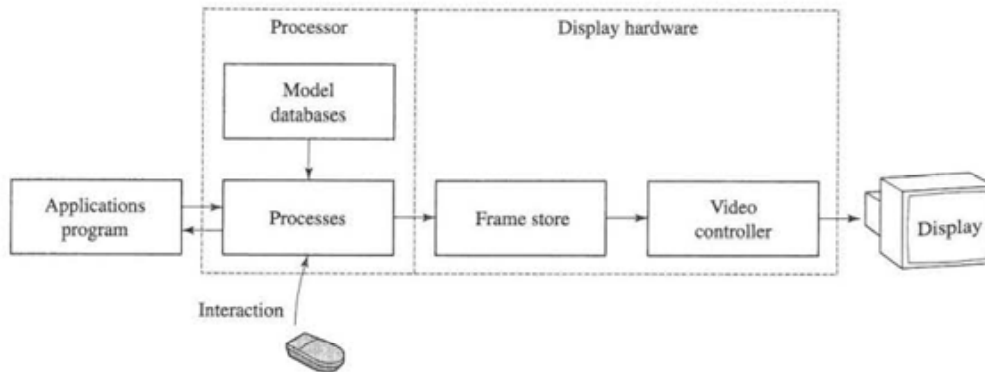


Figure P.1
The main elements of a graphics system.

frame store, a screen buffer or a refresh memory. This information – the discretized computer image – is continually converted by a video controller into a set of horizontal scan lines (a raster) which is then fed to a TV-type monitor. The image is generated by an application program which usually accesses a model or geometric description of an object or objects. The main elements in such a system are shown in Figure P.1. The display hardware to the right of the dotted line can be separate to the processor, but nowadays is usually integrated as in the case of an enhanced PC or a graphics workstation. The raster graphics device overshadows all other hardware developments in the sense that it made possible the display of shaded three-dimensional objects – the single most important theoretical development. The interaction of three-dimensional objects with a light source could be calculated and the effect projected into two-dimensional space and displayed by the device. Such shaded imagery is the foundation of modern computer graphics.

The two early landmark achievements that made shaded imagery possible are the algorithms developed by Gouraud in 1971 and Phong in 1975 enabling easy and fast calculation of the intensities of pixels when shading an object. The Phong technique is still in mainstream use and is undoubtedly responsible for most of the shaded images in computer graphics.



A brief history of shaded imagery

When we look at computer graphics from the viewpoint of its practitioners, we see that since the mid-1970s the developmental motivation has been photo-realism or the pursuit of techniques that make a graphics image of an object or scene indistinguishable from a TV image or photograph. A more recent strand of the application of these techniques is to display information in, for example, medicine, science and engineering.

The foundation of photo-realism is the calculation of light-object interaction and this splits neatly into two fields – the development of local reflection

models and the development of global models. Local or direct reflection models only consider the interaction of an object with a light source as if the object and light were floating in dark space. That is, only the first reflection of light from the object is considered. Global reflection models consider how light reflects from one object and travels onto another. In other words the light impinging on a point on the surface can come either from a light source (direct light) or indirect light that has first hit another object. Global interaction is for the most part an unsolved problem, although two partial solutions, ray tracing and radiosity, are now widely implemented.

Computer graphics research has gone the way of much modern scientific research – early major advances are created and consolidated into a practical technology. Later significant advances seem to be more difficult to achieve. We can say that most images are produced using the Phong local reflection model (first reported in 1975), fewer using ray tracing (first popularized in 1980) and fewer still using radiosity (first reported in 1984). Although there is still much research being carried out in light–scene interaction methodologies much of the current research in computer graphics is concerned more with applications, for example, with such general applications as animation, visualization and virtual reality. In the most important computer graphics publication (the annual SIGGRAPH conference proceedings) there was in 1985 a total of 22 papers concerned with the production techniques of images (rendering, modelling and hardware) compared with 13 on what could loosely be called applications. A decade later in 1995 there were 37 papers on applications and 19 on image production techniques.

Modelling surface reflection with local interaction

Two early advances which went hand-in-hand were the development of hidden surface removal algorithms and shaded imagery – simulating the interaction of an object with a light source. Most of the hidden surface removal research was carried out in the 1970s and nowadays, for general-purpose use, the most common algorithm is the Z-buffer – an approach that is very easy to implement and combine with shading or rendering algorithms.

In shaded imagery the major prop is the Phong reflection model. This is an elegant but completely empirical model that usually ends up with an object reflecting more light than it receives. Its parameters are based on the grossest aspects of reflection of light from a surface. Despite this, it is the most widely used model in computer graphics – responsible for the vast majority of created images. Why is this so? Probably because users find it adequate and it is easy to implement.

Theoretically based reflection models attempt to model reflection more accurately and their parameters have physical meaning – that is they can be measured for a real surface. For example, light reflects differently from an isotropic surface, such as plastic, compared to its behaviour with a non-isotropic surface

such as brushed aluminium and such an effect can be imitated by explicitly modelling the surface characteristics. Such models attempt to imitate the behaviour of light at a 'milliscale' level (where the roughness or surface geometry is still much greater than the wavelength of light). Their purpose is to imitate the material signature – why different materials in reality look different. Alternatively, parameters of a model can be measured on a real surface and used in a simulation. The work into more elaborate or theoretical local reflection models does not seem to have gained any widespread acceptance as far as its implementation in rendering systems is concerned. This may be due to the fact that users do not perceive that the extra processing costs are worth the somewhat marginal improvement in the appearance of the shaded object.

All these models, while attending to the accurate modelling of light from a surface, are local models which means that they only consider the interaction of light with the object as if the object was floating in free space. No object-object interaction is considered and one of the main problems that immediately arises is that shadows – a phenomenon due to global interaction – are not incorporated into the model and have to be calculated by a separate 'add-on' algorithm.

The development of the Phong reflection model spawned research into add-on shadow algorithms and texture mapping, both of which enhanced the appearance of the shaded object and tempered the otherwise 'floating in free space' plastic look of the basic Phong model.

Modelling global interaction

The 1980s saw the development of two significant global models – light reflection models that attempt to evaluate the interaction between objects. Global interaction gives rise to such phenomena as the determination of the intensity of light within a shadow area, the reflection of objects in each other (specular interaction) and a subtle effect known as colour bleeding where the colour from a diffuse surface is transported to another nearby surface (diffuse interaction). The light intensity within a shadow area can only be determined from global interaction. An area in shadow, by definition, cannot receive light directly from a light source but only indirectly from light reflecting from another object. When you see shiny objects in a scene you expect to see in them reflections of other objects. A very shiny surface, such as chromium plate, behaves almost as a mirror taking all its surface detail from its surroundings and distorting this geometrically according to surface curvature.

The successful global models are ray tracing and radiosity. However, in their basic implementation both models only cater for one aspect of global illumination. Ray tracing attends to perfect specular reflection – very shiny objects reflecting in each other, and radiosity models diffuse interaction which is light reflecting off matte surfaces to illuminate other surfaces. Diffuse interaction is common in man-made interiors which tend to have carpets on the floor and matte finishes on the walls. Areas in a room that cannot see the light source are

illuminated by diffuse interaction. Mutually exclusive in the phenomena they model, images created by both methods tend to have identifying 'signatures'. Ray-traced images are notable for perfect recursive reflections and super sharp refraction. Radiosity images are usually of softly-lit interiors and do not contain specular or shiny objects.

Computer graphics is not an exact science. Much research in light-surface interaction in computer graphics proceeds by taking existing physical models and simulating them with a computer graphics algorithm. This may involve much simplification in the original mathematical model so that it can be implemented as a computer graphics algorithm. Ray tracing and radiosity are classic examples of this tendency. Simplifications, which may appear gross to a mathematician, are made by computer graphicists for practical reasons. The reason this process 'works' is that when we look at a synthesized scene we do not generally perceive the simplifications in the mathematics unless they result in visible degeneracies known as aliases. However, most people can easily distinguish a computer graphics image from a photograph. Thus computer graphics have a 'realism' of their own that is a function of the model, and the nearness of the computer graphics image to a photograph of a real scene varies widely according to the method. Photo-realism in computer graphics means the image *looks* real not that it approaches, on a pixel by pixel basis, a photograph. This subjective judgement of computer graphics images somewhat devalues the widely used adjective 'photo-realistic', but there you are. With one or two exceptions very little work has been done on comparing a human's perception of a computer graphics image with, say, a TV image of the equivalent real scene.

Acknowledgements

The author would like to thank the following:

- Lightwork Design Ltd (Sheffield, UK) and Dave Cauldron for providing the facilities to produce the front cover image (model of the Tate Gallery, St Ives, UK) and the renderer, RadioRay.
- Daniel Teece for the images on the back cover which he produced as part of his PhD thesis and which comprise three-dimensional paint strokes interactively applied to a standard polygon model.
- Lee Cooper for producing Figures 6.12, 7.8, 8.7, 8.10, 10.4, 18.1, 18.3, 18.5, 18.6, 18.7, 18.8, 18.9, 18.10, 18.11, 18.12, 18.13, 18.14, 18.16, 18.17 and 18.19 together with the majority of images on the CD-ROM. These were produced using Lightworks Application Development System kindly supplied by Lightwork Design Ltd.
- Mark Puller for Figure 13.1.
- Steve Maddock for Figures 1.5, 4.9, 8.8, 8.26.
- Agata Opalach for Figure 2.20.
- Klaus de Geuss for Figures 13.10 and 13.11.
- Guy Brown for Figure 16.19.
- Fabio Policarpo for Figure 8.14.
- IMDM University, Hamburg, for Figure 13.3.

In addition the author would like to thank Keith Mansfield, the production staff at Addison-Wesley, Robert Chaundry of Bookstyle for his care with the manuscript and Dionea Watt for the cover design.

The publishers are grateful to the following for permission to reproduce copyright material:

Figure 2.1 reproduced with the permission of Viewpoint Digital, Inc; Figure 2.4 from *Tutorial: Computer Graphics*, Ze (Beatty and Booth, 1982), © 1982 IEEE, The Institute of Electrical and Electronics Engineers, Inc., New York; Figures 2.7 and 2.8 from *Generative Modelling for Computer Graphics and CAD* (Snyder, 1992), Academic

Press, London; Figure 2.20 reproduced with the permission of Agata Opalach; Figure 13.3 from *VOXEL-MAN, Part 1: Brain and Skull*, CD-ROM for UNIX workstations and LINUX PCs, Version 1.1 © Karl-Heinz Höhne and Springer-Verlag GmbH & Co. KG 1996, reproduced with kind permission; Figure 16.14 reproduced with the permission of Steven Seitz; Figure 17.28 from *ACM Transactions on Graphics*, 15:3, July 1996 (Hubbard, 1996), ACM Publications, New York.

Whilst every effort has been made to trace the owners of copyright material, in a few cases this has proved impossible and we take this opportunity to offer our apologies to any copyright holders whose rights we may have unwittingly infringed.

Trademark notice

Apple™ and QuickTime™ are trademarks of Apple Computer, Inc.
Luxo™ is a trademark of Jac Jacobson Industries.
Kodak™ is a trademark of Eastman Kodak Company.
RenderMan™ is a trademark of Pixar Corporation.
VAX™ is a trademark of Digital Equipment Corporation.
3D Dataset™ is a trademark of Viewpoint Digital, Inc.

1

Mathematical fundamentals of computer graphics

- 1.1 Manipulating three-dimensional structures
- 1.2 Structure-deforming transformations
- 1.3 Vectors and computer graphics
- 1.4 Rays and computer graphics
- 1.5 Interpolating properties in the image plane

1.1

Manipulating three-dimensional structures

Transformations are important tools in generating three-dimensional scenes. They are used to move objects around in an environment, and also to construct a two-dimensional view of the environment for a display surface. This chapter deals with basic three-dimensional transformations, and introduces some useful shape-changing transformations and basic three-dimensional geometry that we will be using later in the text.

In computer graphics the most popular method for representing an object is the polygon mesh model. This representation is fully described in Chapter 2. We do this by representing the surface of an object as a set of connected planar polygons and each polygon is a list of (connected) points. This form of representation is either exact or an approximation depending on the nature of the object. A cube, for example, can be represented exactly by six squares. A cylinder, on the other hand can only be approximated by polygons; say six rectangles for the curved surface and two hexagons for the end faces. The number of polygons used in the approximation determines how accurately the object is represented and this has repercussions in modelling cost, storage and rendering cost and quality. The popularity of the polygon mesh modelling technique in computer graphics is undoubtedly due to its inherent simplicity and the development of inexpensive shading algorithms that work with such models.

A polygon mesh model consists of a structure of vertices, each vertex being a three-dimensional point in so-called world coordinate space. Later we will be concerned with how vertices are connected to form polygons and how polygons are structured into complete objects. But to start with we shall consider objects just as a set of three-dimensional vertices and look at how these are transformed in three-dimensional space using linear transformations.

1.1.1

Three-dimensional geometry in computer graphics – affine transformations

In this section we look at three-dimensional affine transformations. These are the transformations that effect rotation, scaling, shear and translation. Affine transformations can be represented by a matrix, and a set of affine transformations can be combined into a single overall affine transformation. Technically we say that an affine transformation is made up of any combination of linear transformations (rotation, scaling and shear) followed by translation.

Objects are defined in a world coordinate system which is conventionally a right-handed system. A right-handed and left-handed three-dimensional coordinate system is shown in Figure 1.1. Right-handed systems are the standard mathematical convention, although left-handed systems have, and still are, used in the special context of viewing systems in computer graphics. The difference between the two systems is the sense of the z axis as shown in the figure. Rotating your fingers around the z axis, from the positive x axis to the positive y axis, gives a different z direction for your thumb depending on which system is used.

It is sometimes convenient to define objects in their own local coordinate system. There are three reasons for this. When a three-dimensional object is modelled it is useful to build up the vertices with respect to some reference point in the object. In fact a complex object may have a number of local coordinate systems, one for each sub-part. It may be that the same object is to appear many times in a scene and a definition with a local origin is the only sensible way to

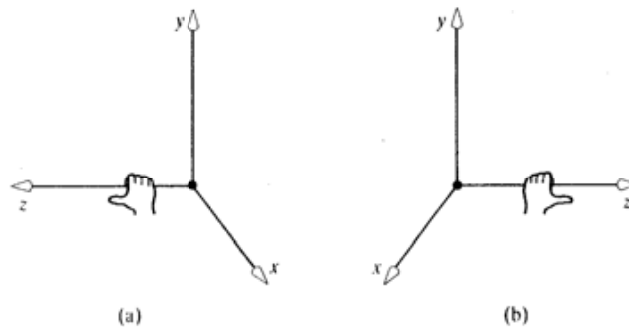


Figure 1.1
(a) Right-handed and
(b) left-handed coordinate
systems.

set this up. Instancing an object by applying a mix of translations, rotation and scaling transformations can then be seen as transforming the local coordinate system of each object to the world coordinate system. Finally when an object is to be rotated, it is easier if the rotation is defined with respect to a local reference point such as an axis of symmetry.

A set of vertices or three-dimensional points belonging to an object can be transformed into another set of points by a linear transformation. Both sets of points remain in the same coordinate system. Matrix notation is used in computer graphics to describe the transformations and the convention in computer graphics is to have the point or vector as a column matrix, preceded by the transformation matrix T .

Using matrix notation, a point V is transformed under translation, scaling and rotation as:

$$V' = V + D$$

$$V' = S V$$

$$V' = R V$$

where D is a translation vector and S and R are scaling and rotation matrices.

These three operations are the most commonly used transformations in computer graphics. In animation a rigid body can undergo only rotation and translation, and scaling is used in object modelling. To enable the above transformations to be treated in the same way and combined, we use a system called homogeneous coordinates which increase the dimensionality of the space. The practical reason for this in computer graphics is to enable us to include translation as matrix multiplication (rather than addition) and thus have a unified scheme for linear transformations. In a homogeneous system a vertex:

$$V(x, y, z)$$

is represented as

$$V(w \cdot X, w \cdot Y, w \cdot Z, w)$$

for any scale factor $w \neq 0$. The three-dimensional Cartesian coordinate representation is then:

$$x = X/w$$

$$y = Y/w$$

$$z = Z/w$$

If we consider w to have the value 1 then the matrix representation of a point is:

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Translation can now be treated as matrix multiplication, like the other two transformations and becomes:

$$\mathbf{V}' = \mathbf{T} \mathbf{V}$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This specification implies that the object is translated in three dimensions by applying a displacement T_x , T_y and T_z to each vertex that defines the object. The matrix notation is a convenient and elegant way of writing the transformation as a set of three equations:

$$\begin{aligned} x' &= x + T_x \\ y' &= y + T_y \\ z' &= z + T_z \end{aligned}$$

The set of transformations is completed by scaling and rotation. First scaling:

$$\mathbf{V}' = \mathbf{S} \mathbf{V}$$

$$\mathbf{S} = \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here S_x , S_y and S_z are scaling factors. For uniform scaling $S_x = S_y = S_z$, otherwise scaling occurs along these axes for which the scaling factor is non-unity. Again the process can be expressed less succinctly by a set of three equations:

$$\begin{aligned} x' &= x \cdot S_x \\ y' &= y \cdot S_y \\ z' &= z \cdot S_z \end{aligned}$$

applied to every vertex in the object.

To rotate an object in three-dimensional space we need to specify an axis of rotation. This can have any spatial orientation in three-dimensional space, but it is easiest to consider rotations that are parallel to one of the coordinate axes. The transformation matrices for anti-clockwise (looking along each axis towards the origin) rotation about the x , y and z axes respectively are:

$$\mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta & 0 \\ 0 & \sin \theta & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_y = \begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{R}_z = \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The z axis matrix specification is equivalent to the following set of three equations:

$$x' = x \cos \theta - y \sin \theta$$

$$y' = x \sin \theta + y \cos \theta$$

$$z' = z$$

Figure 1.2 shows examples of these transformations.

The inverse of these transformations is often required. \mathbf{T}^{-1} is obtained by negating T_x, T_y and T_z . Replacing S_x, S_y and S_z by their reciprocals gives \mathbf{S}^{-1} and negating the angle of rotation gives \mathbf{R}^{-1} .

Any set of rotations, scaling and translations can be multiplied or concatenated together to give a net transformation matrix. For example if:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \mathbf{M}_1 \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and

$$\begin{bmatrix} x'' \\ y'' \\ z'' \\ 1 \end{bmatrix} = \mathbf{M}_2 \begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix}$$

then the transformation matrices can be concatenated:

$$\mathbf{M}_3 = \mathbf{M}_2 \mathbf{M}_1$$

and

$$\begin{bmatrix} x'' \\ y'' \\ z'' \\ 1 \end{bmatrix} = \mathbf{M}_3 \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Note the order: in the product $\mathbf{M}_2 \mathbf{M}_1$ the first transformation to be applied is \mathbf{M}_1 . Although translations are commutative, rotations are not and

$$\mathbf{R}_1 \mathbf{R}_2 \neq \mathbf{R}_2 \mathbf{R}_1$$

This is demonstrated in Figure 1.2(e) and 1.2(f).

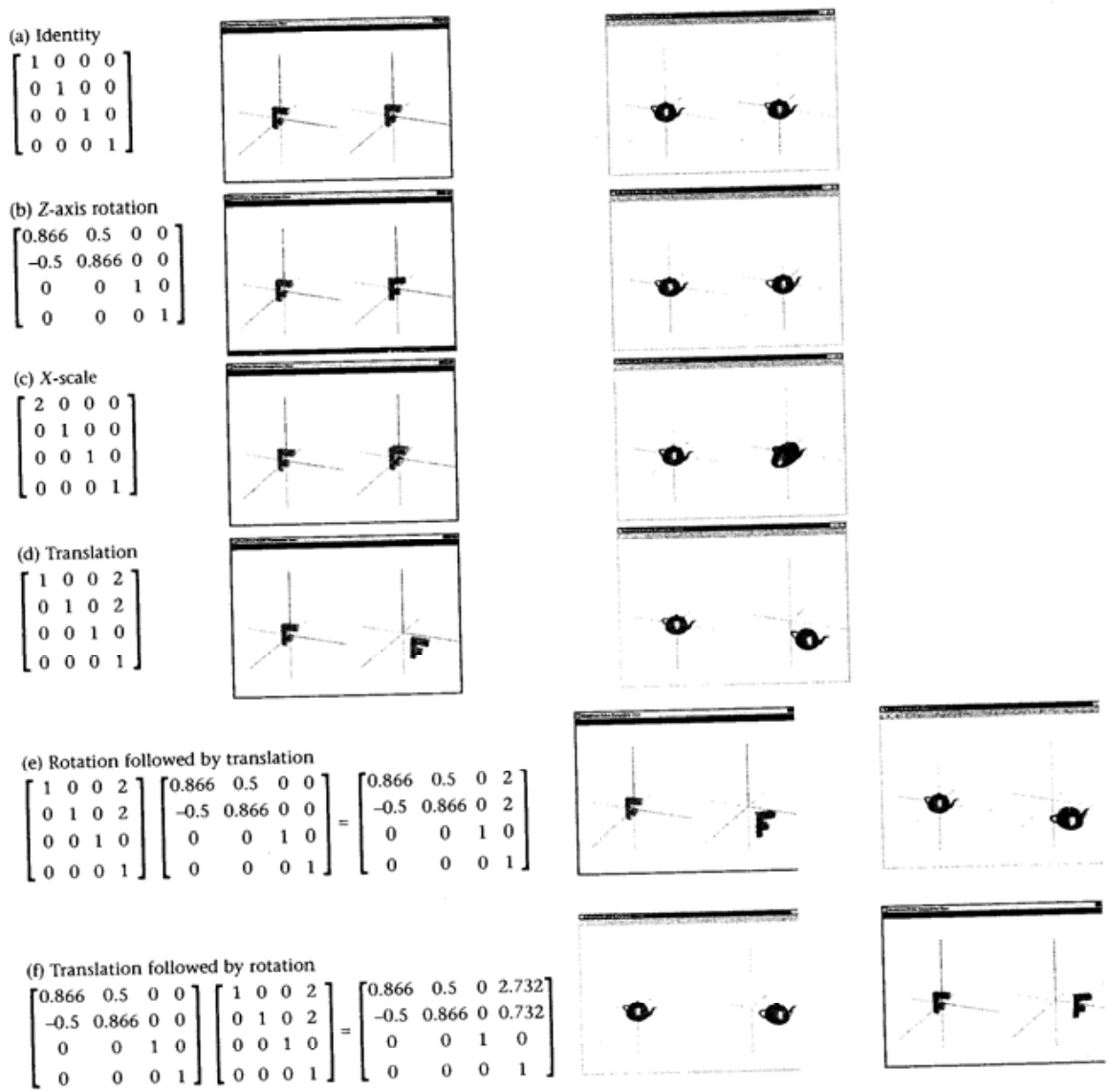


Figure 1.2
Examples of linear
transformations.

A general transformation matrix will be of the form:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & T_x \\ A_{21} & A_{22} & A_{23} & T_y \\ A_{31} & A_{32} & A_{33} & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The 3×3 upper-left sub-matrix A is the net rotation and scaling while T gives the net translation.

The ability to concatenate transformations to form a net transformation matrix is useful because it gives a single matrix specification for any linear transformation. For example, consider rotating a body about a line parallel to the z axis which passes through the point $(T_x, T_y, 0)$ and also passes through one of the vertices of the object. Here we are implying that the object is not at the origin and we wish to apply rotation about a reference point in the object itself. In other words we want to rotate the object with respect to its own coordinate system known as a local coordinate system (see also Section 1.1.2). We cannot simply apply a rotation matrix because this is defined with respect to the origin and an object not positioned at the origin would rotate and translate – not usually the desired effect. Instead we have to derive a net transformation matrix as follows:

- (1) Translate the object to the origin,
- (2) Apply the desired rotation, and,
- (3) Translate the object back to its original position.

The net transformation matrix is:

$$\begin{aligned}
 T_2RT_1 &= \begin{bmatrix} 1 & 0 & 0 & -T_x \\ 0 & 1 & 0 & -T_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos \theta & -\sin \theta & 0 & 0 \\ \sin \theta & \cos \theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \\
 &= \begin{bmatrix} \cos \theta & -\sin \theta & 0 & (-T_x \cos \theta + T_y \sin \theta + T_x) \\ \sin \theta & \cos \theta & 0 & (-T_x \sin \theta - T_y \cos \theta + T_y) \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

This process is shown in Figure 1.3 where θ is 30° .

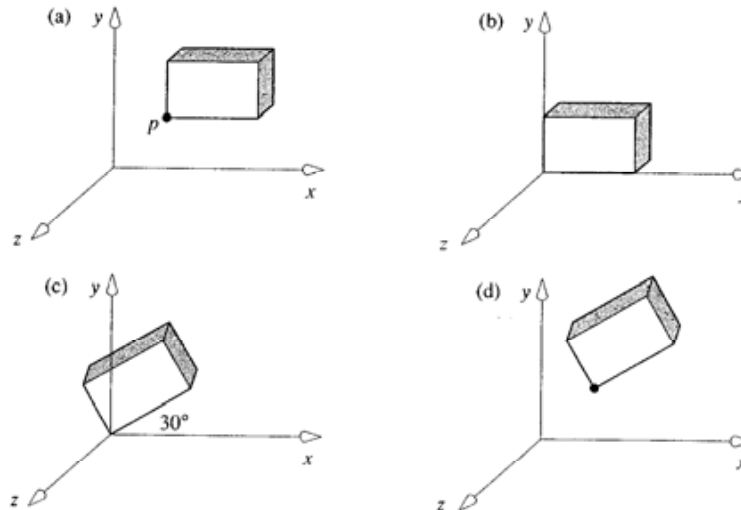


Figure 1.3
 Two stages in building up the rotation of an object about one of its own vertices. The rotation is about an axis parallel to the z axis at point $(T_x, T_y, 0)$. A two-dimensional projection (with the z axis coming out of the paper) is shown for clarity. (a) Original object at $(T_x, T_y, 0)$. (b) Translate to the origin. (c) Rotate about the origin. (d) Translate to $p(T_x, T_y, 0)$.

1.1.2

Transformations for changing coordinate systems

Up to now we have discussed transformations that operate on points all of which are expressed relative to one particular coordinate system. This is known as the world coordinate system. In many contexts in computer graphics we need to derive transformations that take points from one coordinate system into another. The commonest context is when we have a number of objects each specified by a set of vertices in a coordinate system embedded in the object itself. This is known as a local coordinate system. Every object will have a convenient local coordinate system; for example, a complex object that is basically cylindrical in shape may have a coordinate axis that coincides with the long axis of the cylinder. If we wish to bring a number of such objects together and position them in a scene then the scene would take the world coordinate system and we would apply translations, rotations and scale transformations to the objects to position them in the scene. Thus we can consider that the transformations operate on the object or equivalently on the local coordinate system of the object. Transformations that emplace an object with a local coordinate system into a position in a world coordinate system are called modelling transformations.

Another important context that involves a change of coordinate system is the transformation from the world coordinate system to the view coordinate system – a viewing transformation. Here we have a new coordinate system – an object if you like – defined with respect to the world coordinate system and we have to transform the vertices in the world coordinate system to this new system.

Consider two coordinate systems with axes parallel, that is the systems which only differ by a translation. If we wish to transform points currently expressed in system 1 into system 2 then we use the inverse of the transformation that takes the origin of system 1 to that of system 2. That is a point $(x, y, z, 1)$ in system 1 transforms to a point $(x', y', z', 1)$ by:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & -T_x \\ 0 & 1 & 0 & -T_y \\ 0 & 0 & 1 & -T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$= \mathbf{T}_{12} = (\mathbf{T}_{21})^{-1}$$

which is the transformation that translates the origin of system 1 to that of system 2 (where the point is still expressed relative to system 1). Another way of putting it is to say that the transformation generally required is the inverse of the transformation that takes the old axes to the new axes within the current coordinate system.

This is an important result because we generally find transformations between coordinate systems by considering transformations that operate on origins and axes. In the case of viewing systems a change in coordinate systems involves both translation and rotation and we find the required transformation in this way by considering a combination of rotations and translations.

1.2

Structure-deforming transformations

The above linear transformations either move an object (rotation and translation) or scale the object. Uniform scaling preserves shape. Using different values of S_x , S_y and S_z the object is stretched or squeezed along particular coordinate axes. In this section we introduce a set of transformations that deform the object. These are fully described in Barr (1984) where they are termed global deformations. The particular deformations detailed in this paper are tapering, twisting and bending.

Barr uses a formula definition for the transformations:

$$X = F_x(x)$$

$$Y = F_y(y)$$

$$Z = F_z(z)$$

where (x, y, z) is a vertex in an undeformed solid and (X, Y, Z) is the deformed vertex. Using this notation the scaling transformation above is:

$$X = S_x(x)$$

$$Y = S_y(y)$$

$$Z = S_z(z)$$

Tapering is easily developed from scaling. We choose a tapering axis and differentially scale the other two components setting up a tapering function along this axis. Thus, to taper an object along its Z axis:

$$X = rx$$

$$Y = ry$$

$$Z = z$$

where:

$$r = f(z)$$

is a linear or non-linear tapering profile or function. Thus, the transformation becomes a function of r . That is, we change the transformation depending on where in the space it is applied. In effect we are scaling a scaling transformation.

Global axial twisting can be developed as a differential rotation just as tapering is a differential scaling. To twist an object about its z axis we apply:

$$X = x \cos \theta - y \sin \theta$$

$$Y = x \sin \theta + y \cos \theta$$

$$Z = z$$

where:

$$\theta = f_5(z)$$

and $f'(z)$ specifies the rate of twist per unit length along the z axis.

A global linear bend along an axis is a composite transformation comprising a bent region and a region outside the bent region where the deformation is a rotation and a translation.

Barr defines a bend region along the Y axis as:

$$y_{\min} \leq y \leq y_{\max}$$

the radius of curvature of the bend is $1/k$ and the centre of the bend is at $y = y_0$.
The bending angle is:

$$\theta = k(y' - y_0)$$

where:

$$y' = \begin{cases} y_{\min} & y \leq y_{\min} \\ y & y_{\min} < y < y_{\max} \\ y_{\max} & y \geq y_{\max} \end{cases}$$

The deforming transformation is given by:

$$X = x$$

$$Y = \begin{cases} -\sin \theta \left(z - \frac{1}{k} \right) + y_0 & y_{\min} \leq y \leq y_{\max} \\ -\sin \theta \left(z - \frac{1}{k} \right) + y_0 + \cos \theta (y - y_{\min}) & y < y_{\min} \\ -\sin \theta \left(z - \frac{1}{k} \right) + y_0 + \cos \theta (y - y_{\max}) & y > y_{\max} \end{cases}$$

$$Z = \begin{cases} -\cos \theta \left(z - \frac{1}{k} \right) + y_0 & y_{\min} \leq y \leq y_{\max} \\ -\cos \theta \left(z - \frac{1}{k} \right) + \frac{1}{k} + \sin \theta (y - y_{\min}) & y < y_{\min} \\ -\cos \theta \left(z - \frac{1}{k} \right) + \frac{1}{k} + \sin \theta (y - y_{\max}) & y > y_{\max} \end{cases}$$

Figure 1.4 shows an example of each of these transformations. The deformation on the cube is an intuitive reflection of the effects and the same transformations are applied to the Utah teapot. Figure 1.5 (Colour Plate) shows a rendered version of a polygon mesh object (a corrugated cylinder) that has been twisted and tapered.

Non-constrained, non-linear deformations cannot be applied to polygon meshes in general. One problem is the connectivity constraints between vertices. For example, we cannot twist a cube, represented as six surfaces, without limit and retain a structure suitable for rendering. Another problem is that deformations where vertices move apart have the effect of reducing the polygonal resolution of the original model giving rise to a degradation in silhouette edge aliasing (dealt with in detail in Chapter 4). Thus the polygonal nature of the object model constrains the nature of the deformation and this can only be overcome by subdivision of the original mesh as a function of the 'severity' of the deformation.

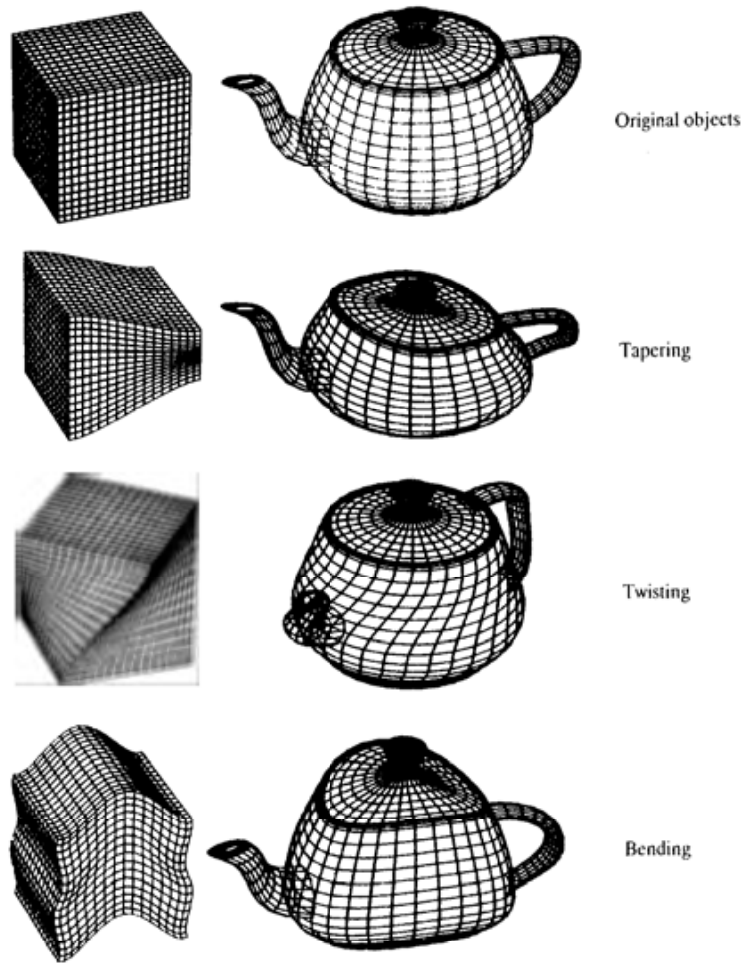


Figure 1.4
Structure-deforming
transformations.

1.3

Vectors and computer graphics

Vectors are used in a variety of contexts in computer graphics. A vector is an entity that possesses magnitude and direction. The common example of a vector is the velocity of a particle moving through space. The velocity possesses both a magnitude and a direction and this distinguishes it from a scalar quantity which only has magnitude. An example of a scalar is the temperature of a point in space. A three-dimensional vector is written as a triple:

$$\mathbf{V} = (v_1, v_2, v_3)$$

where each component v_i is a scalar.

1.3.1

Addition of vectors

Addition of two vectors \mathbf{V} and \mathbf{W} , for example, is defined as:

$$\begin{aligned}\mathbf{X} &= \mathbf{V} + \mathbf{W} \\ &= (x_1, x_2, x_3) \\ &= (v_1 + w_1, v_2 + w_2, v_3 + w_3)\end{aligned}$$

Geometrically this is interpreted as follows. The 'tail' of \mathbf{W} is placed at the 'head' of \mathbf{V} , and \mathbf{X} is the vector formed by joining the tail of \mathbf{V} to the head of \mathbf{W} . This is shown in Figure 1.6 for a pair of two-dimensional vectors together with an alternative, but equivalent, interpretation.

1.3.2

Length of vectors

The magnitude or length of a vector is defined as:

$$|\mathbf{V}| = (v_1^2 + v_2^2 + v_3^2)^{1/2}$$

and we interpret this geometrically as the distance from its tail to its head.

We normalize a vector to produce a unit vector which is a vector of length equal to one. The normalized version of \mathbf{V} is:

$$\mathbf{U} = \frac{\mathbf{V}}{|\mathbf{V}|}$$

which is a vector of unit length having the same direction as \mathbf{U} . We can now refer to \mathbf{U} as a direction. Note that we can write:

$$\mathbf{V} = |\mathbf{V}|\mathbf{U}$$

which is saying that any vector is given by its magnitude times its direction. Normalization is used frequently in computer graphics because we are interested in calculating and representing the orientation of entities, and comparative orientation requires normalized vectors.

1.3.3

Normal vectors and cross products

In computer graphics considerable processing is carried out using vectors that are normal to a surface. For example, in a polygon mesh model (see Chapter 2) a nor-

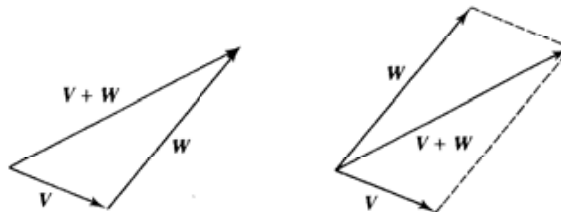


Figure 1.6
Two geometric
interpretations of the
sum of two vectors.

mal vector is used to represent the orientation of a surface when comparing this with the direction of the light. Such a comparison is used in reflection models to compute the intensity of the light reflected from the surface. The smaller the angle between the light vector and the vector that is normal to the surface, the higher is the intensity of the light reflected from the surface (see Chapter 7).

A normal vector to a polygon is calculated from three (non-collinear) vertices of the polygon. Three vertices define two vectors \mathbf{V}_1 and \mathbf{V}_2 (Figure 1.7) and the normal to the polygon is found by taking the cross product of these:

$$\mathbf{N}_p = \mathbf{V}_1 \times \mathbf{V}_2$$

The cross product of two vectors \mathbf{V} and \mathbf{W} is a vector \mathbf{X} and is defined as:

$$\begin{aligned} \mathbf{X} &= \mathbf{V} \times \mathbf{W} \\ &= (v_2w_3 - v_3w_2)\mathbf{i} + (v_3w_1 - v_1w_3)\mathbf{j} + (v_1w_2 - v_2w_1)\mathbf{k} \end{aligned}$$

where \mathbf{i} , \mathbf{j} and \mathbf{k} are the standard unit vectors:

$$\mathbf{i} = (1, 0, 0)$$

$$\mathbf{j} = (0, 1, 0)$$

$$\mathbf{k} = (0, 0, 1)$$

that is, vectors oriented along the coordinate axes that define the space in which the vectors are embedded.

Geometrically a cross product, as we have implied, is a vector whose orientation is normal to the plane containing the two vectors forming the cross product. When determining the surface normal of a polygon, the cross product must point outwards with respect to the object. In a right-handed coordinate system the sense of the cross product vector is given by the right-hand rule. If the first two fingers of your right hand point in the direction of \mathbf{V} and \mathbf{W} then the direction of \mathbf{X} is given by your thumb.

If the surface is a bi-cubic parametric surface (see Chapter 3), then the orientation of the normal vector varies continuously over the surface. We compute the normal at any point (u, v) on the surface again by using a cross product. This is done by first calculating tangent vectors in the two parametric directions (we outline the procedure here for the sake of completeness and give full details in Chapter 3). For a surface defined as $\mathbf{Q}(u, v)$ we find:

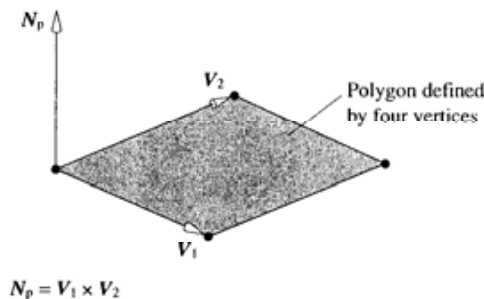


Figure 1.7
Calculating the normal
vector to a polygon.

$$\frac{\partial}{\partial u} \mathbf{Q}(u, v) \quad \text{and} \quad \frac{\partial}{\partial v} \mathbf{Q}(u, v)$$

We then define:

$$\mathbf{N}_s = \frac{\partial \mathbf{Q}}{\partial u} \times \frac{\partial \mathbf{Q}}{\partial v}$$

This is shown schematically in Figure 1.8.

1.3.4

Normal vectors and dot products

The most common use of a dot product in computer graphics is to provide a measure of the angle between two vectors, where one of the vectors is a normal vector to a surface or group of surfaces. Common applications are shading (the angle between a light direction vector and a surface normal) and visibility testing (the angle between viewing vector and a surface normal).

The dot product of vectors \mathbf{V} and \mathbf{W} is a scalar X which is defined as:

$$\begin{aligned} X &= \mathbf{V} \cdot \mathbf{W} \\ &= v_1 w_1 + v_2 w_2 + v_3 w_3 \end{aligned}$$

Figure 1.9(a) shows two vectors. Using the cosine rule we have:

$$|\mathbf{V} - \mathbf{W}|^2 = |\mathbf{V}|^2 + |\mathbf{W}|^2 - 2|\mathbf{V}||\mathbf{W}| \cos \theta$$

where θ is the angle between the vectors. Also it can be shown that:

$$|\mathbf{V} - \mathbf{W}|^2 = |\mathbf{V}|^2 - 2\mathbf{V} \cdot \mathbf{W} + |\mathbf{W}|^2$$

thus:

$$\mathbf{V} \cdot \mathbf{W} = |\mathbf{V}||\mathbf{W}| \cos \theta$$

giving:

$$\cos \theta = \frac{\mathbf{V} \cdot \mathbf{W}}{|\mathbf{V}||\mathbf{W}|}$$

or the angle between two vectors is the dot product of their normalized versions.

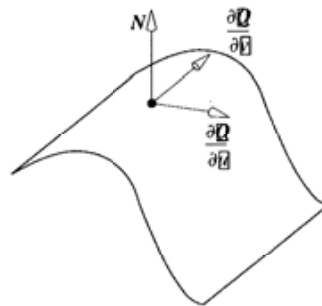


Figure 1.8
Normal \mathbf{N} to a point on a
parametric surface $\mathbf{Q}(u, v)$

We can use the dot product to project a vector onto another vector. Consider a unit vector \mathbf{V} . If we project any vector \mathbf{W} onto \mathbf{V} (Figure 1.9(b)) and call the result \mathbf{X} , then we have:

$$\begin{aligned} |\mathbf{X}| &= |\mathbf{W}| \cos \theta \\ &= |\mathbf{W}| \frac{\mathbf{V} \cdot \mathbf{W}}{|\mathbf{V}| |\mathbf{W}|} \\ &= \mathbf{V} \cdot \mathbf{W} \end{aligned} \quad (1.1)$$

because \mathbf{V} is a unit vector. Thus the dot product of \mathbf{V} and \mathbf{W} is the length of the projection of \mathbf{W} onto \mathbf{V} .

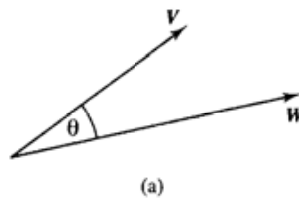
A property of the dot product used in computer graphics is its sign. Because of its relationship to $\cos \theta$ the sign of the dot product of \mathbf{V} and \mathbf{W} (where \mathbf{V} and \mathbf{W} are of any length) is:

$$\begin{aligned} \mathbf{V} \cdot \mathbf{W} &> 0 & \text{if } \theta < 90^\circ \\ \mathbf{V} \cdot \mathbf{W} &= 0 & \text{if } \theta = 90^\circ \\ \mathbf{V} \cdot \mathbf{W} &< 0 & \text{if } \theta > 90^\circ \end{aligned}$$

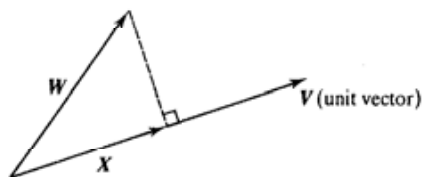
1.3.5

Vectors associated with the normal vector reflection

There are three important vectors that are associated with the surface normal. They are the light direction vector, \mathbf{L} , the reflection vector or mirror vector, \mathbf{R} , and the viewing vector, \mathbf{V} . The light direction vector, \mathbf{L} , is a vector whose direction is given by the line from the tail of the surface normal to the light source; which in simple shading contexts is defined as a point on the surface that we are currently considering. This vector is shown in Figure 1.10(a). The reflection vector, \mathbf{R} , is given by the direction of the light reflected from the surface due to light incoming along direction \mathbf{L} . Sometimes called the mirror direction, geometric optics tells us that the outgoing angle equals the incoming angle as shown in Figure 1.10(b).



(a)



(b)

Figure 1.9

(a) The dot product of the two vectors is related to the cosine of the angle between them:

$$\cos \theta = \frac{\mathbf{V} \cdot \mathbf{W}}{|\mathbf{V}| |\mathbf{W}|}$$

(b) $|\mathbf{X}| = \mathbf{V} \cdot \mathbf{W}$ is the length of the projection of \mathbf{W} onto \mathbf{V} .

Figure 1.10
 Vectors associated with the normal vector. (a) L , the light direction vector, (b) R , the reflection vector, (c) V , the view vector, is a vector of any orientation.

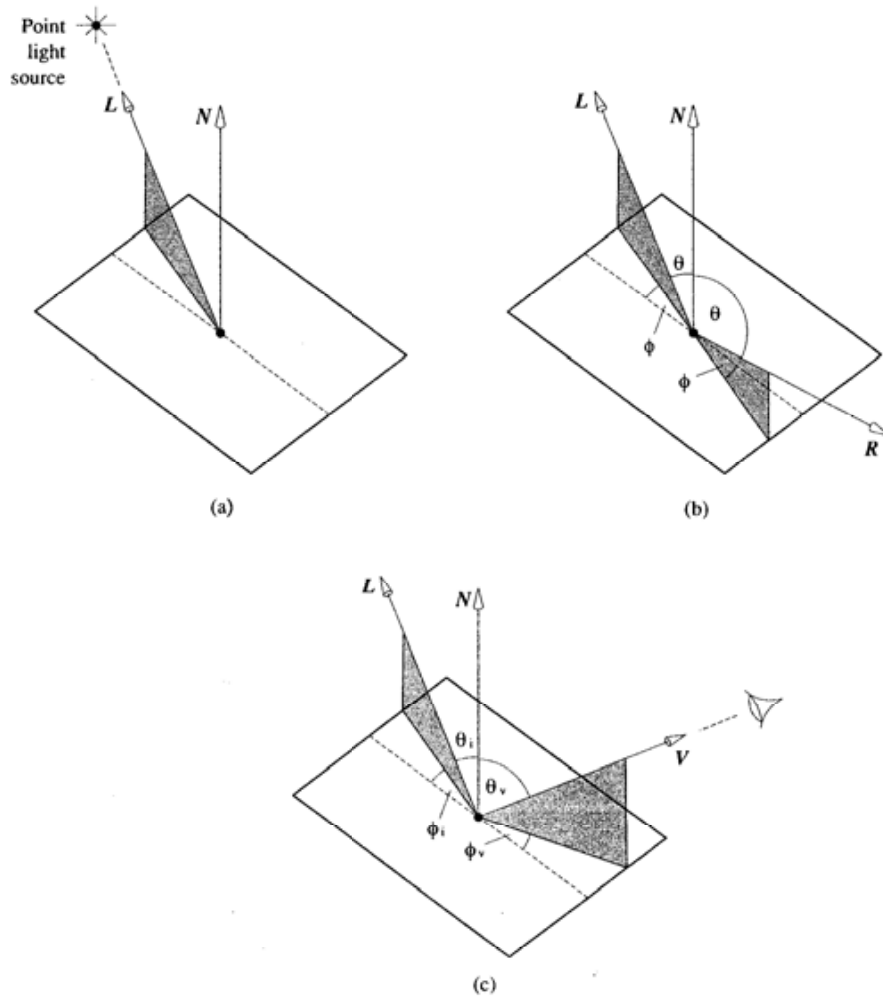
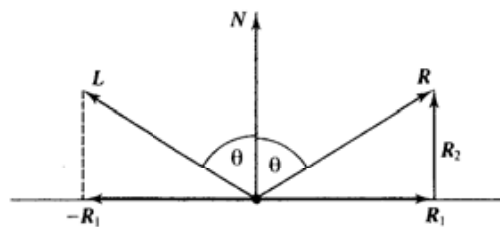


Figure 1.11
 Construction of the reflection vector R .



Consider the construction shown in Figure 1.11. This shows:

$$\mathbf{R} = \mathbf{R}_1 + \mathbf{R}_2$$

$$\mathbf{R}_1 = -\mathbf{L} + \mathbf{R}_2$$

Thus:

$$\mathbf{R} = 2\mathbf{R}_2 - \mathbf{L}$$

from Equation 1.1:

$$\mathbf{R}_2 = (\mathbf{N} \cdot \mathbf{L})\mathbf{N}$$

and

$$\mathbf{R} = 2(\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L} \tag{1.2}$$

Figure 1.10(c) shows a view vector \mathbf{V} . Note that this vector has any arbitrary orientation and we are normally interested in that component of light incoming in direction \mathbf{L} that is reflected along \mathbf{V} . This will depend in general on both the angles ϕ_v and θ_v . We also note that the intensity of outgoing light depends on the incoming angles ϕ_i and θ_i , and this is usually described as a bidirectional dependence because two angles, (ϕ_v, θ_v) and (ϕ_i, θ_i) , in three-dimensional space are involved.

1.4

Rays and computer graphics

In computer graphics we are interested in an entity called a ray (mathematically known as a directed line segment) that possesses position, magnitude and direction. We use this mostly to simulate light as an infinitesimally thin beam – a light ray. If we imagine a ray to be a physical line in three space, then its position is the position of the tail of the line, its magnitude the length of the line between its head and tail and its direction the direction of the line. A ray can be specified by two points or by a single point, and a vector. If the end points of the ray are (x_1, y_1, z_1) and (x_2, y_2, z_2) respectively, then the vector is given by:

$$\mathbf{V} = (x_2 - x_1, y_2 - y_1, z_2 - z_1)$$

Rays are not only used in ray tracing, but they find uses in volume rendering, rendering constructive solid geometry (CSG) volumes and in calculating form factors in radiosity. We will now look at some of the more important calculations associated with rays.

1.4.1

Ray geometry – intersections

Because ray tracing simulates the path of light through an environment, the most common calculation associated with rays is intersection testing – we see whether a ray has hit an object and if so where. Here we test a ray against all objects in the scene for an intersection. This is potentially a very expensive calculation and the most common technique used to make this more efficient is

to enclose objects in the scene in bounding volumes – the most convenient being a sphere – and test first for a ray–sphere intersection. The sphere encloses the object and if the ray does not intersect the sphere it cannot intersect the object. Another common bounding volume is a box.

Sphere and boxes are also used to bound objects for collision detection tests in computer animation (see Chapter 17). Pairs of objects can only collide if their bounding volumes intersect. The motivation here is the same as that for ray tracing – we first cull away pairs that cannot possibly collide before we undertake detailed intersection checking at the individual polygon level. Checking for sphere–sphere intersection is trivial and for boxes – if they are axis aligned – then we only need limit checks in the x , y and z directions.

1.4.2

Intersections – ray–sphere

The intersection between a ray and a sphere is easily calculated. If the end points of the ray are (x_1, y_1, z_1) and (x_2, y_2, z_2) then the first step is to parametrize the ray (Figure 1.12):

$$\begin{aligned}x &= x_1 + (x_2 - x_1)t = x_1 + it \\y &= y_1 + (y_2 - y_1)t = y_1 + jt \\z &= z_1 + (z_2 - z_1)t = z_1 + kt\end{aligned}\tag{1.1}$$

where:

$$0 \leq t \leq 1$$

A sphere at centre (l, m, n) of radius r is given by:

$$(x - l)^2 + (y - m)^2 + (z - n)^2 = r^2$$

Substituting for x , y and z gives a quadratic equation in t of the form:

$$at^2 + bt + c = 0$$

where:

$$a = i^2 + j^2 + k^2$$

$$b = 2i(x_1 - l) + 2j(y_1 - m) + 2k(z_1 - n)$$

$$c = l^2 + m^2 + n^2 + x_1^2 + y_1^2 + z_1^2 + 2(-lx_1 - my_1 - nz_1) - r^2$$

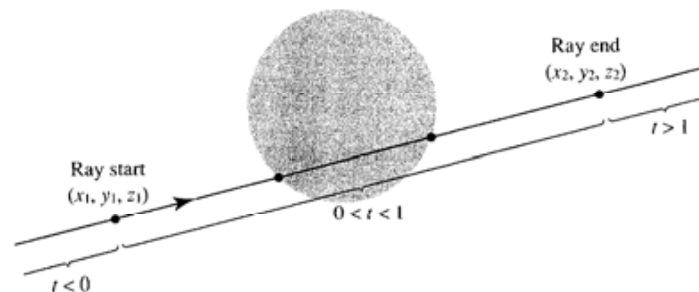


Figure 1.12
Values of parameter t along
a ray.

If the determinant of this quadratic is less than 0 then the line does not intersect the sphere. If the determinant equals 0 then the line grazes or is tangential to the sphere. The real roots of the quadratic give the front and back intersections. Substituting the values for t into the original parametric equations yields these points. Figure 1.12 shows that the value of t also gives the position of the points of intersection relative to (x_1, y_1, z_1) and (x_2, y_2, z_2) . Only positive values of t are relevant and the smallest value of t corresponds to the intersection nearest to the start of the ray.

Other information that is usually required from an intersection is the surface normal (so that the reflected and refracted rays may be calculated) although, if the sphere is being used as a bounding volume, only the fact that an intersection has occurred, or not, is required.

If the intersection point is (x_i, y_i, z_i) and the centre of the sphere is (l, m, n) then the normal at the intersection point is:

$$\mathbf{N} = \left(\frac{x_i - l}{r}, \frac{y_i - m}{r}, \frac{z_i - n}{r} \right)$$

1.4.3

Intersections – ray–convex polygon

If an object is represented by a set of polygons and is convex then the straightforward approach is to test the ray individually against each polygon. We do this as follows:

- (1) Obtain an equation for the plane containing the polygon.
- (2) Check for an intersection between this plane and the ray.
- (3) Check that this intersection is contained by the polygon.

A more common application of this operation is clipping a polygon against a view frustum (see Chapter 5). Here the ‘ray’ is a polygon edge and we need to find the intersection of a polygon edge and a view frustum plane so that the polygon can be split and that part outside the view frustum discarded.

For example, if the plane containing the polygon is:

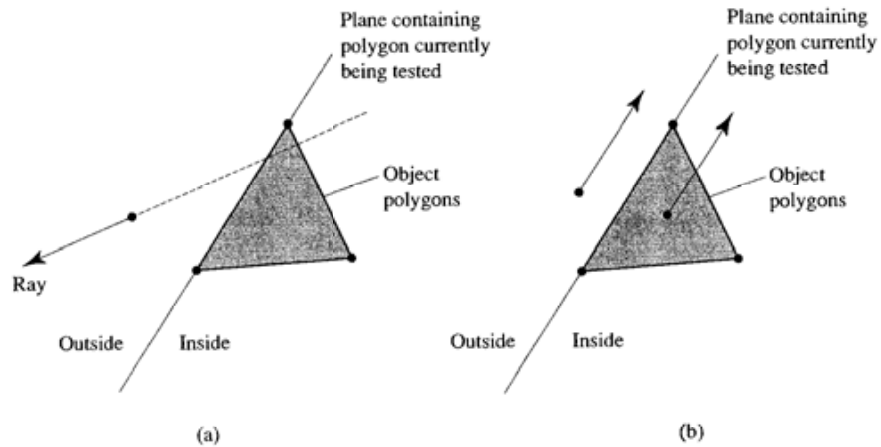
$$Ax + By + Cz + D = 0$$

and the line is defined parametrically as before, then the intersection is given by:

$$t = \frac{-(Ax_1 + By_1 + Cz_1 + D)}{(Ai + Bj + Ck)} \tag{1.2}$$

We can exit the test if $t < 0$. This means that the ray is in the half space, defined by the plane that does not contain the polygon (Figure 1.13(a)). We may also be able to exit if the denominator is equal to zero which means that the line and plane are parallel. In this case the ray origin is either inside or outside the polyhedron. We can check this by examining the sign of the numerator. If the numerator is positive then the ray is in that half space defined by the plane that is outside the object and no further testing is necessary (Figure 1.13(b)).

Figure 1.13
 (a) A ray in the half space that does not contain the object ($t < 0$). (b) A possible exit condition. The ray is parallel to the plane containing the polygon currently being tested. It is either inside or outside the object.



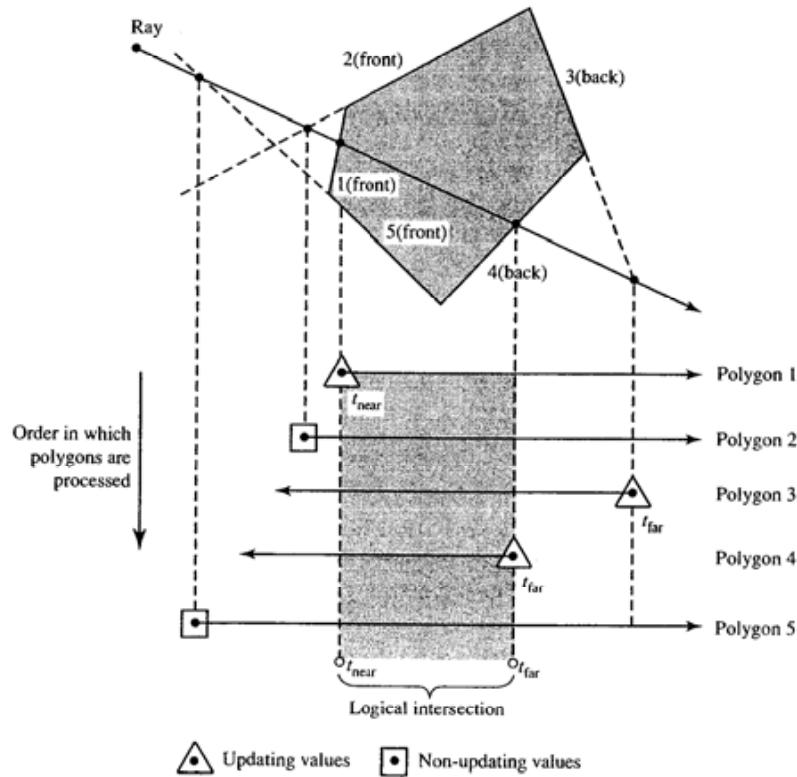
The straightforward method that tests a point for containment by a polygon is simple but expensive. The sum of the angles between lines drawn from the point to each vertex is 360° if the point is inside the polygon, but not if the point lies outside.

There are three disadvantages or inadequacies in this direct approach. We cannot stop when the first intersection emerges from the test unless we also evaluate whether the polygon is front- or back-facing with respect to the ray direction. The containment test is particularly expensive. It is also possible for errors to occur when a ray and a polygon edge coincide.

All of those disadvantages can be overcome by a single algorithm developed by Haines (1991). Again we use the concept of a plane that contains a polygon defining a half space. All points on one side of the plane are outside the polyhedron. Points on the other side may be contained by the polyhedron. The logical intersection of all inside half spaces is the space enclosed by the polyhedron. A ray that intersects a plane creates a directed line segment (unbounded in the direction of the ray) defined by the intersection point and the ray direction. It is easily seen that the logical intersection of all directed line segments gives the line segment that passes through the polyhedron. Proceeding as before we exit from the test when a parallel ray occurs with an 'outside' origin. Otherwise the algorithm considers every polygon and evaluates the logical intersection of the directed line segments. Consider the example shown in Figure 1.14. For each plane we categorize it as front-facing or back-facing with respect to the ray direction. This is given by the sign of the denominator in Equation 1.2 (positive for back-facing, negative for front-facing). The conditions that form the logical intersection of directed line segments are embedded in the algorithm which is:

```
{initialize  $t_{near}$  to large negative value
 $t_{far}$  to large positive value}
if {plane is back-facing} and ( $t < t_{far}$ )
then  $t_{far} = t$ 
```

Figure 1.14
Ray-convex polyhedron
intersection testing (after
Haines (1991)).



```

if {plane is front-facing} and ( $t > t_{near}$ )
then  $t_{near} = t$ 
if ( $t_{near} > t_{far}$ ) then {exit - ray misses}
    
```

1.4.4

Intersections – ray-box

Ray-box intersections are important because boxes may be more useful bounding volumes than spheres, particularly in hierarchical schemes. Also generalized boxes can be used as an efficient bounding volume.

Generalized boxes are formed from pairs of parallel planes, but the pairs of planes can be at any angle with respect to each other. In this section we consider the special case of boxes forming rectangular solids, with the normals to each pair of planes aligned in the same direction as the ray tracing axes or the object space axes.

To check if a ray intersects such a box is straightforward. We treat each pair of parallel planes in turn, calculating the distance along the ray to the first plane (t_{near}) and the distance to the second plane (t_{far}). The larger value of t_{near} and the smaller value of t_{far} is retained between comparisons. If the larger value of t_{near} is greater than the smaller value of t_{far} , the ray cannot intersect the box. This is

shown, for an example in the xy plane in Figure 1.15. If a hit occurs then the intersection is given by t_{near} .

A more succinct statement of the algorithm comes from considering the distance between the intersection points of a pair of parallel planes as intervals. Then if the intervals intersect, the ray hits the volume. If they do not intersect the ray misses.

Again because our convex polygon is reduced to a rectangular solid, we can define the required distances in terms of the box extent. Distances along the ray are given for the x plane pairs as follows: if the box extent is (x_{b1}, y_{b1}, z_{b1}) and (x_{b2}, y_{b2}, z_{b2}) then:

$$t_{1x} = \frac{x_{b1} - x_1}{x_2 - x_1}$$

is the distance along the ray from its origin to the intersection with the first plane, and:

$$t_{2x} = \frac{x_{b2} - x_1}{x_2 - x_1}$$

The calculations for t_{1y} , t_{2y} and t_{1z} , t_{2z} are similar. The largest value out of the t_1 set gives the required t_{near} and the smallest value of the t_2 set gives the required t_{far} . The algorithm can exit at the y plane calculations.

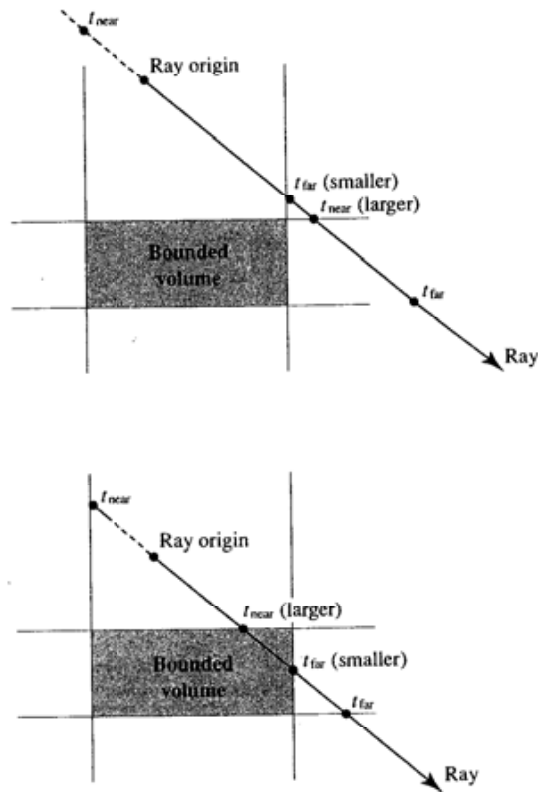


Figure 1.15
Ray-box intersection.

1.4.5
Intersections – ray–quadric

The sphere example given in Section 1.4.2 is a special case of rays intersecting with a general quadric. Ray–quadric intersections can be dealt with by considering the general case, or ‘special’ objects, such as cylinders, can be treated individually for reasons of efficiency.

The general implicit equation for a quadric is:

$$Ax^2 + Ey^2 + Hz^2 + 2Bxy + 2Fyz + 2Cxz + 2Dx + 2Gy + 2Iz + J = 0$$

Following the same approach as we adopted for the case of the sphere, we substitute Equation 1.1 into the above equation and obtain the coefficients a , b and c for the resulting quadratic as follows:

$$a = Ax_d^2 + Ey_d^2 + Hz_d^2 + 2Bx_d y_d + 2Cx_d z_d + 2Fy_d z_d$$

$$b = d(Ax_1 x_d + B(x_1 y_d + x_d y_1) + C(x_1 z_d + x_d z_1) + Dx_d + Ey_1 y_d + F(y_1 z_d + y_d z_1) + Gy_d + Hz_1 z_d + Iz_d$$

$$c = Ax_1^2 + Ey_1^2 + Hz_1^2 + 2Bx_1 y_1 + 2Cx_1 z_1 + 2Dx_1 + 2Fy_1 z_1 + 2Gy_1 + 2Iz_1 + J$$

The equations for the quadrics are:

- Sphere
 $(x - l)^2 + (y - m)^2 + (z - n)^2 = r^2$
 where (l, m, n) is, as before, the centre of the sphere.
- Infinite cylinder
 $(x - l)^2 + (y - m)^2 = r^2$
- Ellipsoid
 $\frac{(x - l)^2}{\alpha^2} + \frac{(y - m)^2}{\beta^2} + \frac{(z - n)^2}{\gamma^2} - 1 = 0$
 where α , β and γ are the semi-axes.
- Paraboloid
 $\frac{(x - l)^2}{\alpha^2} + \frac{(y - m)^2}{\beta^2} - z + n = 0$
- Hyperboloid
 $\frac{(x - l)^2}{\alpha^2} + \frac{(y - m)^2}{\beta^2} + \frac{(z - n)^2}{\gamma^2} - 1 = 0$

1.4.6
Ray tracing geometry – reflection and refraction

The formulae presented in this section are standard formulae in a form that is suitable for incorporation into a simple ray tracer. The source of the formulae is Fresnel's law given in Section 7.1.

Each time a ray intersects a surface it produces, in general, a reflected and refracted ray. The reflection direction, a unit vector, is given (as we saw in Section 1.3.2) by:

$$\begin{aligned} \mathbf{R} &= 2\mathbf{N} \cos \phi - \mathbf{L} \\ &= 2(\mathbf{N} \cdot \mathbf{L})\mathbf{N} - \mathbf{L} \end{aligned}$$

where \mathbf{L} and \mathbf{N} are unit vectors representing the incident ray direction, which is the same as the light vector, and the surface normal respectively. \mathbf{L} , \mathbf{R} and \mathbf{N} are co-planar. These vectors are shown in Figure 1.16, where $\mathbf{I} = -\mathbf{L}$.

A ray striking a partially or wholly transparent object is refracted due to the change in the velocity of light in different media. The angles of incidence and refraction are related by Snell's law:

$$\frac{\sin \phi}{\sin \theta} = \frac{\mu_2}{\mu_1}$$

where the incident and transmitted rays are co-planar with \mathbf{N} . The transmitted ray is represented by \mathbf{T} and this is given by:

$$\begin{aligned} \mathbf{T} &= \mu \mathbf{I} - (\cos \theta + \mu \cos \phi) \mathbf{N} \\ \mu &= \mu_1/\mu_2 \\ \cos \theta &= \frac{1}{\mu^2} (1 - \mu^2(1 - \cos^2 \phi))^{\frac{1}{2}} \end{aligned}$$

as shown in Figure 1.16.

If a ray is travelling from a more to a less dense medium then it is possible for the refracted ray to be parallel to the surface (Figure 1.17). ϕ_c is known as the critical angle. If ϕ is increased then total internal reflection occurs.

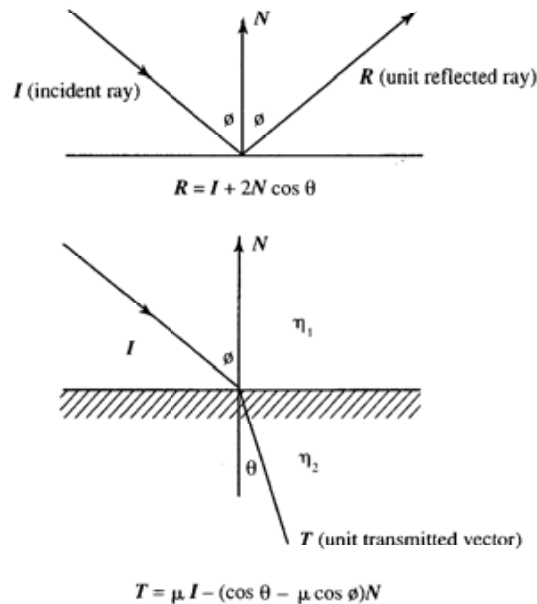
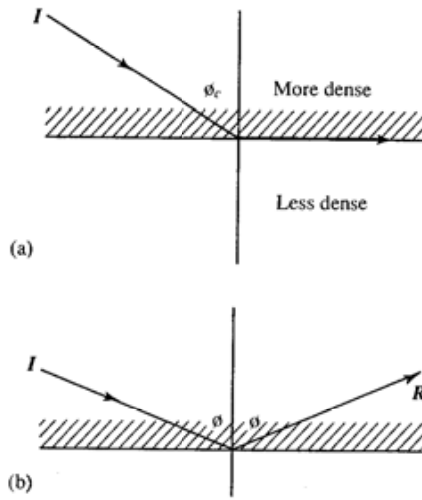


Figure 1.16
Reflection and refraction
geometry.

Figure 1.17
Internal reflection in an object. (a) ϕ_c = critical angle. (b) $\phi > \phi_c$.



1.5 Interpolating properties in the image plane

In mainstream rendering techniques – that is rendering polygons – various properties required for interior pixels are interpolated from the values of these properties at the vertices of the polygon (that is the pixels onto which the vertices project). Such interpolation is known as bilinear interpolation and it is the foundation of the efficiency of this kind of shading.

Referring to Figure 1.18, the interpolation proceeds by moving a scan line down through the pixel set representing the polygon and obtaining start and end values for a scan line by interpolating between the appropriate pair of vertex properties. Interpolation along a scan line then yields a value for the property at each pixel. The interpolation equations are (for the particular edge pair shown in the illustration):

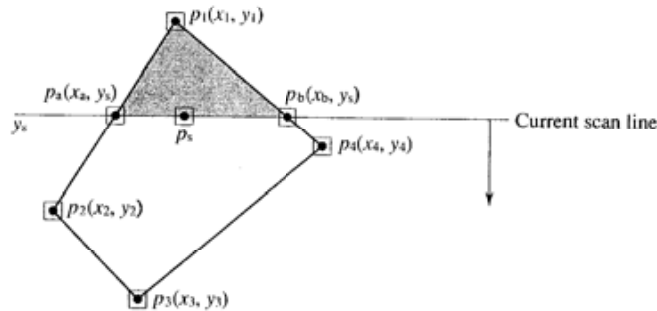


Figure 1.18
Interpolating a property at a pixel from values at the vertex pixels.

$$p_a = \frac{1}{y_1 - y_2} [p_1(y_s - y_2) + p_2(y_1 - y_s)]$$

$$p_b = \frac{1}{y_1 - y_4} [p_1(y_s - y_4) + p_4(y_1 - y_s)]$$

$$p_s = \frac{1}{x_b - x_a} [p_3(x_b - x_s) + p_b(x_s - x_a)]$$

These would normally be implemented using an incremental form, the final equation, for example, becoming:

$$p_s := p_s + \Delta p$$

with the constant value Δp calculated once per scan line.

Representation and modelling of three-dimensional objects (1)

- 2.1 Polygonal representation of three-dimensional objects
- 2.2 Constructive solid geometry (CSG) representation of objects
- 2.3 Space subdivision techniques for object representation
- 2.4 Representing objects with implicit functions
- 2.5 Scene management and object representation
- 2.6 Summary

Introduction

The primary purpose of three-dimensional computer graphics is to produce a two-dimensional image of a scene or an object from a description or model of the object. The object may be a real or existing object or it may exist only as a computer description. A less common but extremely important usage is where the act of creation of the object model and the visualization are intertwined. This occurs in interactive CAD applications where a designer uses the visualization to assist the act of creating the object. Most object descriptions are approximate in the sense that they describe the geometry or shape of the object only to the extent that inputting this description to a renderer produces an image of acceptable quality. In many CAD applications, however, the description has to be accurate because it is used to drive a manufacturing process. The final output is not a two-dimensional image but a real three-dimensional object.

Modelling and representation is a general phrase which can be applied to any or all of the following aspects of objects:

- Creation of a three-dimensional computer graphics representation.
- The technique or method or data structure used to represent the object.

- Manipulation of the representation – in particular changing the shape of an existing model.

The ways in which we can create computer graphics objects are almost as many and varied as the objects themselves. For example, we might construct an architectural object through a CAD interface. We may take data directly from a device such as a laser ranger or a three-dimensional digitizer. We may use some interface based on a sweeping technique where so-called ducted solids are created by sweeping a cross-section along a spine curve. Creation methods have up to now tended to be manual or semi-manual involving a designer working with an interface. As the demand for the representation of highly complex scenes increases – from such applications as virtual reality (VR) – automatic methods are being investigated. For VR applications of existing realities the creation of computer graphics representations from photographs or video is an attractive proposition.

The representation of an object is very much an unsolved problem in computer graphics. We can distinguish between a representation that is required for a machine or renderer and the representation that is required by a user or user interface. Representing an object using polygonal facets – a polygon mesh representation – is the most popular machine representation. It is, however, an inconvenient representation for a user or creator of an object. Despite this it is used as both a user and a machine representation. Other methods have separate user and machine representations. For example, bi-cubic parametric patches and CSG methods, which constitute user or interface representations may be converted into polygon meshes for rendering.

The polygon mesh form suffers from many disadvantages when the object is complex and detailed. In mainstream computer graphics the number of polygons in an object representation can be anything from a few tens to hundreds of thousands. This has serious ramifications in rendering time and object creation cost and in the feasibility of using such objects in an animation or virtual reality environment. Other problems accrue in animation where a model has both to represent the shape of the object and be controlled by an animation system which may require collisions to be calculated or the object to change shape as a function of time. Despite this the polygon mesh is supreme in mainstream computer graphics. Its inertia is due in part to the development of efficient algorithms and hardware to render this description. This has resulted in a somewhat strange situation where it is more efficient – as far as rendering is concerned – to represent a shape with many simple elements (polygons) than to represent it with far fewer (and more accurate) but more complicated elements such as bi-cubic parametric patches (see Section 3.4.2).

The ability to manipulate the shape of an existing object depends strongly on the representation. Polygon meshes do not admit simple shape manipulation. Moving mesh vertices immediately disrupts the 'polygonal resolution' where a shape has been converted into polygons with some degree of accuracy that is related to the local curvature of the surface being represented. For example, imagine twisting a cube represented by six squares. The twisted object cannot be

represented by retaining only six polygons. Another problem with shape manipulation is scale. Sometimes we want to alter a large part of an object which may involve moving many elements at the same time; other times we may require a detailed change.

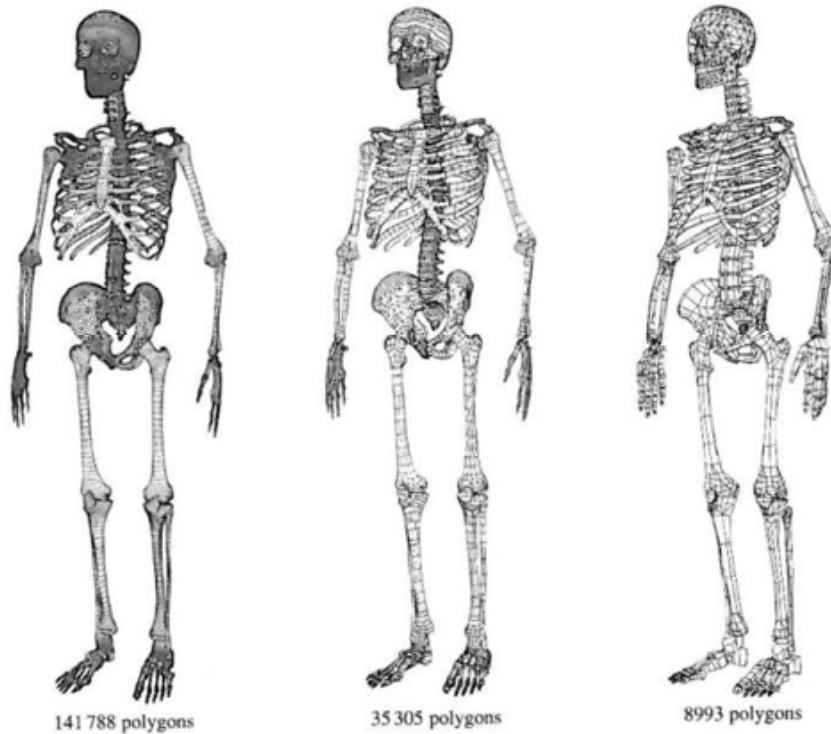
Different representational methods have their advantages and disadvantages but there is no universal solution to the many problems that still exist. Rather, particular modelling methods have evolved for particular contexts. A good example of this tendency is the development of constructive solid geometry methods (CSG) popular in interactive CAD because they facilitate an intuitive interface for the interactive design of complex industrial objects as well as a representation. CSG is a constrained representation in that we can only use it to model shapes that are made up of allowed combinations of the primitive shapes or elements that are included in the system.

How do we choose a representation? The answer is that it depends on the nature of the object, the particular computer graphics technique that we are going to use to bring the object to life and the application. All these factors are interrelated. We can represent some three-dimensional objects exactly using a mathematical formulation, for example, a cylinder or a sphere; for others we use an approximate representation. For objects that cannot be represented exactly by mathematics there is a trade-off between the accuracy of the representation and the bulk of information used. This is illustrated by the polygon mesh skeletons in Figure 2.1. You can only increase the veracity of the representation by increasing the polygonal resolution which then has high cost implications in rendering time.

The ultimate impossibility of this extrapolation has led to hybrid methods for very complex and unique objects such as a human head. For example, in representing a particular human head we can use a combination of a polygon mesh model and photographic texture maps. The solid form of the head is represented by a generic polygon mesh which is pulled around to match the actual dimensions of the head to be modelled. The detailed likeness is obtained by mapping a photographic texture onto this mesh. The idea here is that the detailed variations in the geometry are suggested by the texture map rather than by detailed excursions in the geometry. Of course, it's not perfect because the detail in the photograph depends on the lighting conditions under which it was taken as well as the real geometric detail, but it is a trick that is increasingly being used. Whether we regard the texture mapping as part of the representation or as part of the rendering process is perhaps a matter of opinion; but certainly the use of photographic texture maps in this context enables us to represent a complex object like a human head with a small number of polygons plus a photograph.

This compromise between polygonal resolution and a photographic texture map can be taken to extremes. In the computer games industry the total number of polygons rendered to the screen must be within the limiting number that can be rendered at, say, 15 frames per second on a PC. A recent football game consists of players whose heads are modelled with just a cube onto which a photographic texture is mapped.

Figure 2.1
The art of wireframe – an illustration from Viewpoint Digital's catalogue.
Source: '3D models by Viewpoint Digital, Inc.' Anatomy, Viewpoint's 3D Dataset™ Catalog, 2nd edn.



We now list, in order of approximate frequency of use, the mainstream models used in computer graphics.

- (1) **Polygonal** Objects are approximated by a net or mesh of planar polygonal facets. With this form we can represent, to an accuracy that we choose, an object of any shape. However, the accuracy is somewhat arbitrary in this sense. Consider Figure 2.1 again: are 142 000 polygons really necessary, or can we reduce the polygonal resolution without degrading the rendered image, and if so by how much? The shading algorithms are designed to visually transform the faceted representation in such a way that the piecewise linear representation is not visible in the shaded version (except on the silhouette edge). Connected with the polygonal resolution is the final projected size of the object on the screen. Waste is incurred when a complex object, represented by many thousands of polygons, projects onto a screen area that is made up of only a few pixels.
- (2) **Bi-cubic parametric patches** (see Chapter 3) These are 'curved quadrilaterals'. Generally we can say that the representation is similar to the polygon mesh except that the individual polygons are now curved surfaces. Each patch is specified by a mathematical formula that gives the position of

the patch in three-dimensional space and its shape. This formula enables us to generate any or every point on the surface of the patch. We can change the shape or curvature of the patch by editing the mathematical specification. This results in powerful interactive possibilities. The problems are, however, significant. It is very expensive to render or visualize the patches. When we change the shape of individual patches in a net of patches there are problems in maintaining 'smoothness' between the patch and its neighbours. Bi-cubic parametric patches can be either an exact or an approximate representation. They can only be an exact representation of themselves, which means that any object, say, a car body panel, can only be represented exactly if its shape corresponds exactly to the shape of the patch. This somewhat torturous statement is necessary because when the representation is used for real or existing objects, the shape modelled will not necessarily correspond to the surface of the object.

An example of the same object represented by both bi-cubic parametric patches and by polygonal facets is shown in Figure 3.28 (a) and (c). This clearly shows the complexity/number of elements trade-off with the polygon mesh representation requiring 2048 elements against the 32-patch representation.

- (3) **Constructive solid geometry (CSG)** This is an exact representation to within certain rigid shape limits. It has arisen out of the realization that very many manufactured objects can be represented by 'combinations' of elementary shapes or geometric primitives. For example, a chunk of metal with a hole in it could be specified as the result of a three-dimensional subtraction between a rectangular solid and a cylinder. Connected with this is the fact that such a representation makes for easy and intuitive shape control – we can specify that a metal plate has to have a hole in it by defining a cylinder of appropriate radius and subtracting it from the rectangular solid, representing the plate. The CSG method is a volumetric representation – shape is represented by elementary volumes or primitives. This contrasts with the previous two methods which represent shape using surfaces. An example of a CSG-represented object is shown in Figure 2.14.
- (4) **Spatial subdivision techniques** This simply means dividing the object space into elementary cubes, known as voxels, and labelling each voxel as empty or as containing part of an object. It is the three-dimensional analogue of representing a two-dimensional object as the collection of pixels onto which the object projects. Labelling all of three-dimensional object space in this way is clearly expensive, but it has found applications in computer graphics. In particular, in ray tracing where an efficient algorithm results if the objects are represented in this way. An example of a voxel object is shown in Figure 2.16. We are now representing the three-dimensional space occupied by the object; the other methods we have introduced are representations of the surface of the object.
- (5) **Implicit representation** Occasionally in texts implicit functions are mentioned as an object representation form. An implicit function is, for example:

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

which is the definition for a sphere. On their own these are of limited usefulness in computer graphics because there is a limited number of objects that can be represented in this way. Also, it is an inconvenient form as far as rendering is concerned. However, we should mention that such representations do appear quite frequently in three-dimensional computer graphics – in particular in ray tracing where spheres are used frequently – both as objects in their own right and as bounding objects for other polygon mesh representations.

Implicit representations are extended into implicit functions which can loosely be described as objects formed by mathematically defining a surface that is influenced by a collection of underlying primitives such as spheres. Implicit functions find their main use in shape-changing animation – they are of limited usefulness for representing real objects.

We have arranged the categories in order of popularity; another useful comparison is: with voxels and polygon meshes the number of representational elements per object is likely to be high (if accuracy is to be achieved) but the complexity of the representation is low. This contrasts with bi-cubic patches where the number of elements is likely to be much lower in most contexts but the complexity of the representation is higher.

We should not deduce from the above categorization that the choice of a representation is a free one. The representational form is decided by both the rendering technique and the application. Consider, for example, the continuous/discrete representation distinction. A discrete representation – the polygon mesh – is used to represent the arbitrary shapes of existing real world objects – it is difficult to see how else we would deal with such objects. In medical imaging the initial representation is discrete (voxels) because this is what the imaging technology produces. On the other hand in CAD work we need a continuous representation because eventually we are going to produce, say, a machine part from the internal description. The representation has, therefore, to be exact.

The CSG representation does not fit easily into these comparisons. It is both a discrete and a continuous representation, being a discrete combination of interacting primitives, some of which can be described by a continuous function.

Another important distinguishing factor is surface versus volume representation. The polygon mesh is an approximate representation of the surface of an object and the rendering engine is concerned with providing a visualization of that surface. With Gouraud shading the algorithm is only concerned with using geometric properties associated with the surface representation. In ray tracing, because the bulk of the cost is involved in tracking rays through space and finding which objects they intersect, a surface representation implies high rendering cost. Using a volume representation, where the object space is labelled according to object occupancy, greatly reduces the overall cost of rendering.

The relationship between a rendering method and the representation is critically important in the radiosity method and here, to avoid major defects in the final image, there has to be some kind of interaction between the representation and the execution of the algorithm. As the algorithm progresses the representation must adapt so that more accurate consideration is given to areas in the emerging solution that need greater consideration. In other words, because of the expense of the method, it is difficult to decide *a priori* what the level of detail in the representation should be. The unwieldiness of the concept of having a scene representation depend on the progress of the rendering algorithm is at the root of the difficulty of the radiosity method and is responsible for its (current) lack of uptake as a mainstream tool.

2.1

Polygonal representation of three-dimensional objects

This is the classic representational form in three-dimensional graphics. An object is represented by a mesh of polygonal facets. In the general case an object possesses curved surfaces and the facets are an approximation to such a surface (Figure 2.2). Polygons may contain a vertex count that emerges from the technology used to create the model, or we may constrain all polygons to be triangles. It may be necessary to do this, for example, to gain optimal performance from special-purpose hardware or graphics accelerator cards.

Polygonal representations are ubiquitous in computer graphics. There are two reasons for this. Creating polygonal objects is straightforward (although for complex objects the process can be time consuming and costly) and visually effective algorithms exist to produce shaded versions of objects represented in this way. As we have already stated, polygon meshes are strictly a machine representation – rather than a convenient user representation – and they often function in this capacity for other representations which are not directly renderable. Thus bi-cubic parametric patches, CSG and voxel representations are often converted into polygon meshes prior to rendering.

There are certain practical difficulties with polygon meshes. Foremost amongst these is accuracy. The accuracy of the model, or the difference between the faceted representation and the curved surface of the object, is usually arbitrary. As far as final image quality is concerned, the size of individual polygons should ideally depend on local spatial curvature. Where the curvature changes

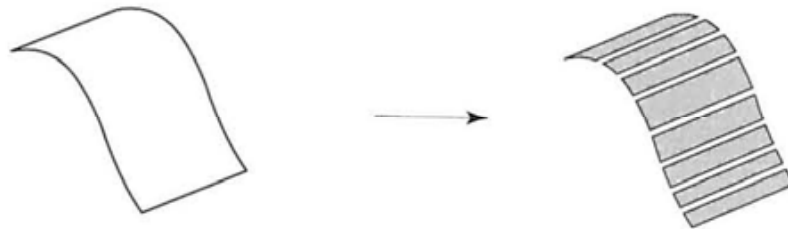


Figure 2.2
Approximating a curved
surface using polygonal
facets.

rapidly, more polygons are required per unit area of the surface. These factors tend to be related to the method used for creating the polygons. If, for example, a mesh is being built from an existing object, by using a three-dimensional digitizer to determine the spatial coordinates of polygon vertices, the digitizer operator will decide on the basis of experience how large each polygon should be. Sometimes polygons are extracted algorithmically (as in, for example, the creation of an object as a solid of revolution or in a bi-cubic patch subdivision algorithm) and a more rigorous approach to the rate of polygons per unit area of the surface is possible.

One of the most significant developments in three-dimensional graphics was the emergence in the 1970s of shading algorithms that deal efficiently with polygonal objects, and at the same time, through an interpolation scheme, diminish the visual effect of the piecewise linearities in the representation. This factor, together with recent developments in fixed program rendering hardware, has secured the entrenchment of the polygon mesh structure.

In the simplest case a polygon mesh is a structure that consists of polygons represented by a list of linked (x, y, z) coordinates that are the polygon vertices (edges are represented either explicitly or implicitly as we shall see in a moment). Thus the information we store to describe an object is finally a list of points or vertices. We may also store, as part of the object representation, other geometric information that is used in subsequent processing. These are usually polygon normals and vertex normals. Calculated once only, it is convenient to store these in the object data structure and have them undergo any linear transformations that are applied to the object.

It is convenient to order polygons into a simple hierarchical structure. Figure 2.3(a) shows a decomposition that we have called a conceptual hierarchy for reasons that should be apparent from the illustration. Polygons are grouped into surfaces and surfaces are grouped into objects. For example, a cylinder possesses three surfaces: a planar top and bottom surface together with a curved surface. The reason for this grouping is that we must distinguish between those edges that are part of the approximation – edges between adjacent rectangles in the curved surface approximation to the cylinder, for example – and edges that exist in reality. The way in which these are subsequently treated by the rendering process is different – real edges must remain visible whereas edges that form part of the approximation to a curved surface must be made invisible. Figure 2.3(b) shows a more formal representation of the topology in Figure 2.3(a).

An example of a practical data structure which implements these relationships is shown in Figure 2.3(c). This contains horizontal, as well as vertical, hierarchical links, necessary for programmer access to the next entity in a horizontal sequence. It also includes a vertex reference list which means that actual vertices (referred to by each polygon that shares them) are stored only once. Another difference between the practical structure and the topological diagram is that access is allowed directly to lower-level entities. Wireframe visualizations of an object are used extensively, and to produce a wireframe image requires direct access to the edge level in the hierarchy. Vertical links between the edges' and the

polygons' levels can be either backward pointers or forward pointers depending on the type of renderer that is accessing the structure. In a scan line renderer, edges are the topmost entity whereas in a Z-buffer renderer polygons are. A Z-buffer renderer treats polygons as independent entities, rendering one polygon at a time. A scan line renders all those polygons that straddle the scan line being rendered.

The approach just described is more particularly referred to as a vertex-based boundary model. Sometimes it is necessary to use an edge-based boundary

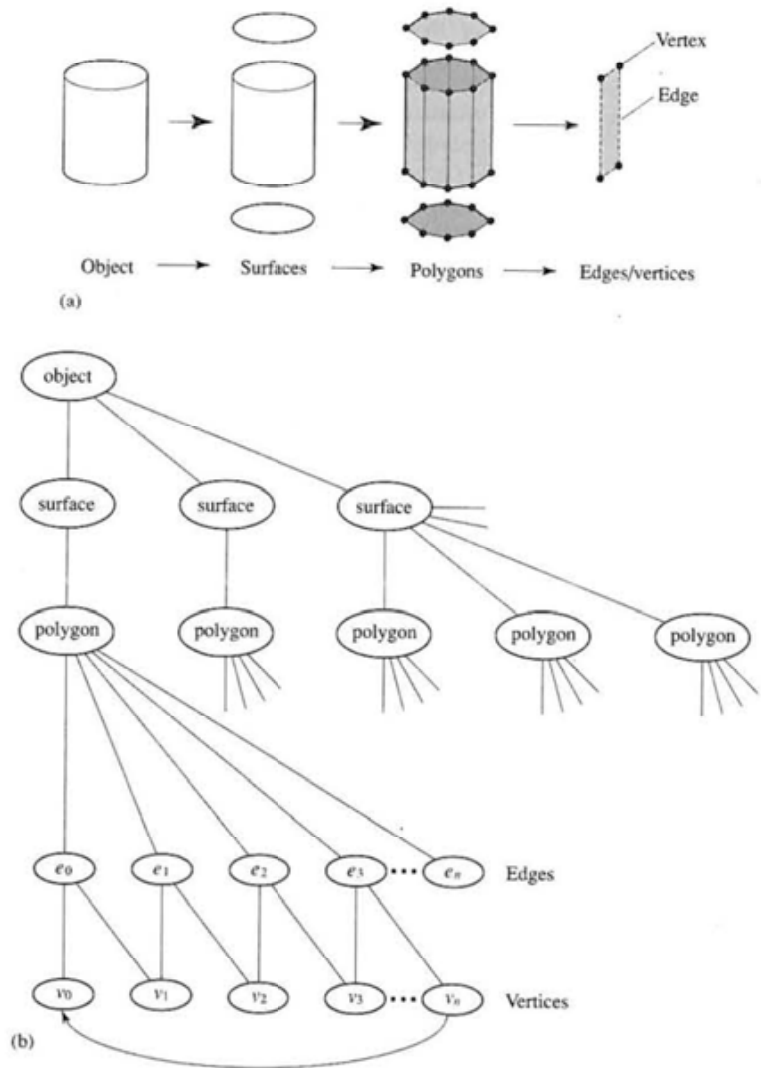
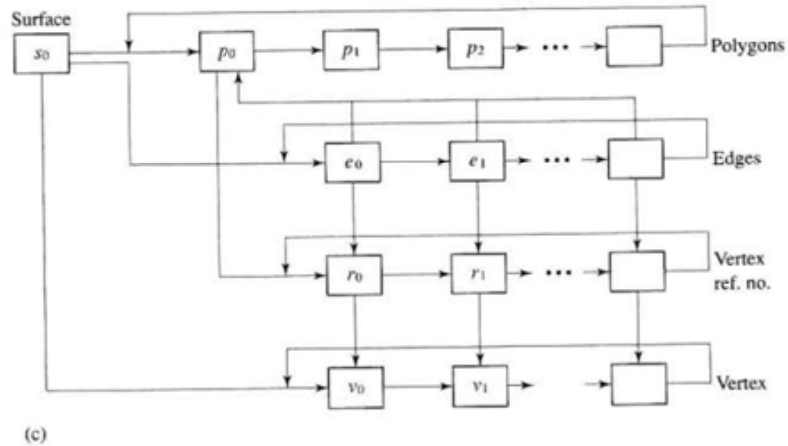


Figure 2.3
Representation of an object
as a mesh of polygons.
(a) Conceptual hierarchy.
(b) Topological
representation.

Figure 2.3 continued
(c) A practical data structure.



model, the most common manifestation of which is a winged-edge data structure (Mantyla 1988). An edge-based model represents a face in terms of a closing sequence of edges.

The data structure just described encapsulates the basic geometry associated with a polygonal facets of an object. Information required by applications and renderers is also usually contained in the scene/object database. The following list details the most common attributes found in polygon mesh structures. They are either data structure pointers, real numbers or binary flags. It is unlikely that all of these would appear in a practical application, but a subset is found in most object representations.

- Polygon attributes
 - (1) Triangular or not.
 - (2) Area.
 - (3) Normal to the plane containing the polygon.
 - (4) Coefficients (A, B, C, D) of the plane containing the polygon where $Ax + By + Cz + D = 0$.
 - (5) Whether convex or not.
 - (6) Whether it contains holes or not.
- Edge attributes
 - (1) Length.
 - (2) Whether an edge is between two polygons or between two surfaces.
 - (3) Polygons on each side of the edge.
- Vertex attributes
 - (1) Polygons that contribute to the vertex.

- (2) Shading or vertex normal – the average of the normals of the polygons that contribute to the vertex.
- (3) Texture coordinates (u, v) specifying a mapping into a two-dimensional texture image.

All these are absolute properties that exist when the object is created. Polygons can acquire attributes as they are passed through the graphics pipeline. For example, an edge can be tagged as a silhouette edge if it is between two polygons with normals facing towards and away from the viewer.

A significant problem that crops up in many guises in computer graphics is the scale problem. With polygonal representation this means that, in many applications, we cannot afford to render all the polygons in a model if the viewing distance and polygonal resolution are such that many polygons project onto a single pixel. This problem bedevils flight simulators (and similarly computer games) and virtual reality applications. An obvious solution is to have a hierarchy of models and use the one appropriate to projected screen area. There are two problems with this; the first is that in animation (and it is animation applications where this problem is most critical) switching between models can cause visual disturbances in the animation sequence – the user can see the switch from one resolution level to another. The other problem is how to generate the hierarchy and to decide how many levels it should contain. Clearly we can start with the highest resolution model and subdivide, but this is not necessarily straightforward. We look at this problem in more detail in Section 2.5.

2.1.1

Creating polygonal objects

Although a polygon mesh is the most common representational form in computer graphics, modelling, although straightforward, is somewhat tedious. The popularity of this representation derives from the ease of modelling, the emergence of rendering strategies (both hardware and software) to process polygonal objects and the important fact that there is no restriction whatever on the shape or complexity of the object being modelled.

Interactive development of a model is possible by ‘pulling’ vertices around with a three-dimensional locator device but in practice this is not a very useful method. It is difficult to make other than simple shape changes. Once an object has been created, any single polygon cannot be changed without also changing its neighbours. Thus most creation methods use either a device or a program; the only method that admits user interaction is item 4 on the following list.

Four common examples of polygon modelling methods are:

- (1) Using a three-dimensional digitizer or adopting an equivalent manual strategy.
- (2) Using an automatic device such as a laser ranger.
- (3) Generating an object from a mathematical description.
- (4) Generating an object by sweeping.

The first two modelling methods convert real objects into polygon meshes, the next two generate models from definitions. We distinguish between models generated by mathematical formulae and those generated by interacting with curves which are defined mathematically.

2.1.2

Manual modelling of polygonal objects

The easiest way to model a real object is manually using a three-dimensional digitizer. The operator uses experience and judgement to emplace points on an object which are to be polygon vertices. The three-dimensional coordinates of these vertices are then input to the system via a three-dimensional digitizer. The association of vertices with polygons is straightforward. A common strategy for ensuring an adequate representation is to draw a net over the surface of the object – like laying a real net over the object. Where curved net lines intersect defines the position of the polygon vertices. A historic photograph of this process is shown in Figure 2.4. This shows students creating a polygon mesh model of a car in 1974. It is taken from a classic paper by early outstanding pioneers in computer graphics – Sutherland *et al.* (1974).

2.1.3

Automatic generation of polygonal objects

A device that is capable of creating very accurate or high resolution polygon mesh objects from real objects is a laser ranger. In one type of device the object is placed on a rotating table in the path of the beam. The table also moves up and down vertically. The laser ranger returns a set of contours – the intersection of the object and a set of closely spaced parallel planes – by measuring the distance to the object surface. A 'skinning' algorithm, operating on pairs of contours, converts the boundary data into a very large number of triangles (Figure 2.5(a)). Figure 2.5(b) is a rendered version of an object polygonized in this way. The skinning algorithm produced, for this object, over 400 000 triangles. Given that only around half of these may be visible on screen and that the object

Figure 2.4
The Utah Beetle – an early example of manual modelling.
Source: Beatty and Booth
Tutorial: Computer Graphics,
2nd edn, The Institute of
Electrical and Electronics
Engineers, Inc.: New York.
© 1982 IEEE.

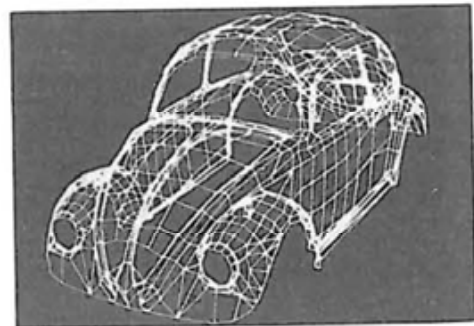
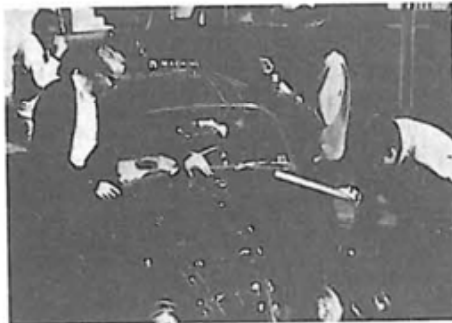
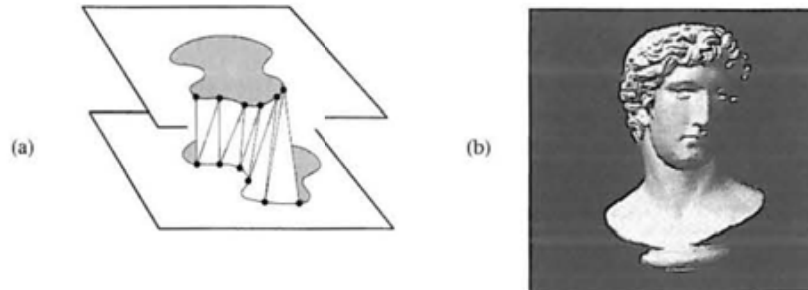


Figure 2.5
A rendered polygonal object scanned by a laser ranger and polygonized by a simple skinning algorithm. (a) A skinning algorithm joins points on consecutive contours to make a three-dimensional polygonal object from the contours. (b) A 400 000 polygonal object produced by a skinning algorithm.



projects onto about half the screen surface implies that each triangle projects onto one pixel on average. This clearly illustrates the point mentioned earlier that it is extremely wasteful of rendering resources to use a polygonal resolution where the average screen area onto which a polygon projects approaches a single pixel. For model creation, laser ranglers suffer from the significant disadvantage that, in the framework described – fully automatic rotating table device – they can only accurately model convex objects. Objects with concavities will have surfaces which will not necessarily be hit by the incident beam.

2.1.4

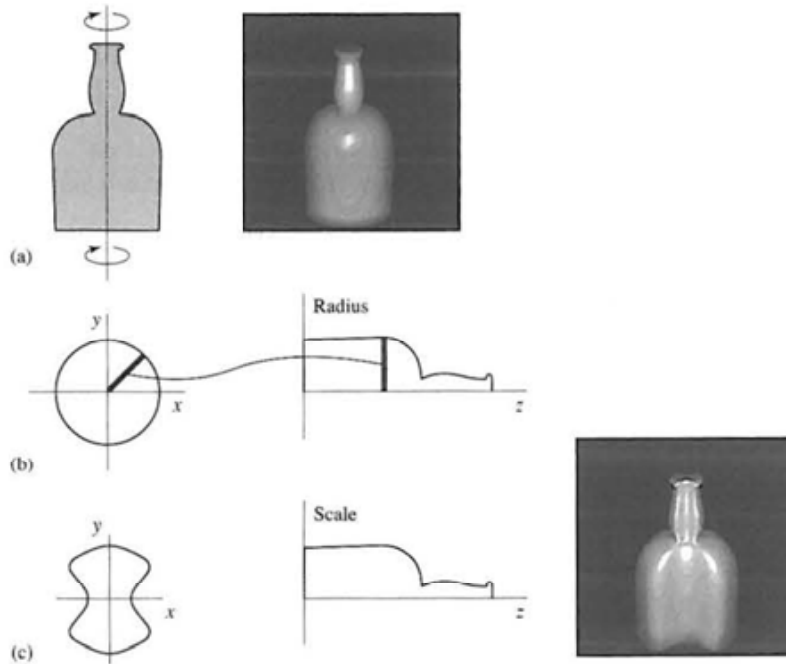
Mathematical generation of polygonal objects

Many polygonal objects are generated through an interface into which a user puts a model description in the form of a set of curves that are a function of two-dimensional or two-parameter space. This is particularly the case in CAD applications where the most popular paradigm is that of sweeping a cross-section in a variety of different ways. There are two benefits to this approach. The first is fairly obvious. The user works with some notion of shape which is removed from the low level activity of constructing an object from individual polygonal facets. Instead, shape is specified in terms of notions that are connected with the form of the object – something that Snyder (1992) calls 'the logic of shapes'. A program then takes the user description and transforms it into polygons. The transformation from the user description to a polygon mesh is straightforward. A second advantage of this approach is that it can be used in conjunction with either polygons as primitive elements or with bi-cubic parametric patches (see Section 3.6).

The most familiar manifestation of this approach is a solid of revolution where, say, a vertical cross-section is swept through 180° generating a solid with a circular horizontal cross-section (Figure 2.6(a)). The obvious constraint of solids of revolution is that they can only represent objects possessing rotational symmetry.

A more powerful generative model is arrived at by considering the same solid generated by sweeping a circle, with radius controlled by a profile curve,

Figure 2.6
 Straight spine objects –
 solid of revolution vs
 cross-sectional sweeping.
 (a) A solid of revolution
 generated by sweeping
 a (vertical) cross-section.
 (b) The same solid can be
 generated by sweeping
 a circle, whose radius is
 controlled by a profile
 curve, up a straight
 vertical spine. (c) Non-circular
 cross-section.

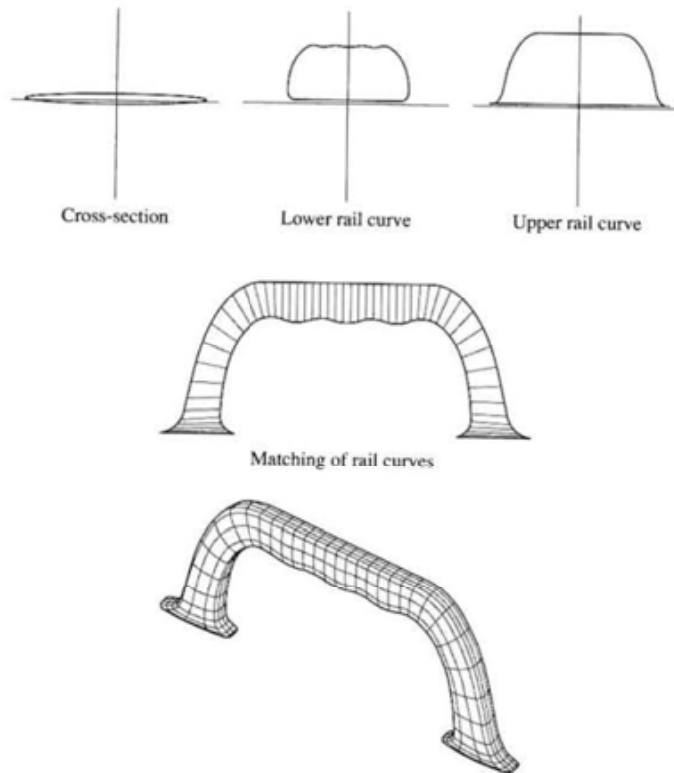


vertically up a straight spine (Figure 2.6(b)). In the event that the profile curve is a constant, we have the familiar notion of extrusion. This immediately removes the constraint of a circular cross-section and we can have cross-sections of arbitrary shape (Figure 2.6(c)).

Now consider controlling the shape of the spine. We can incorporate the notion of a curved spine and generate objects that are controlled by a cross-sectional shape, a profile curve and a spine curve as Figure 2.9 demonstrates.

Other possibilities emerge. Figure 2.7 shows an example of what Snyder calls a rail product surface. Here a briefcase carrying handle is generated by sweeping a cross-section along a path determined by the midpoints of two rail curves. The long axis extent of the elliptical-like cross-section is controlled by the same two curves – hence the name. A more complex example is the turbine blade shown in Figure 2.8. Snyder calls this an affine transformation surface – because the spine is now replaced by affine transformations, controlled by user specified curves. Each blade is generated by extruding a rectangular cross-section along the z axis. The cross-section is specified as a rectangle, and three shape controlling curves, functions of z , supply the values used in the transformations of the cross-section as it is extruded. The cross-section is, for each step in z , scaled separately in x and y , translated in x , rotated around, translated back in x , and extruded along the z axis.

Figure 2.7
Snyder's rail curve
product surfaces. Source:
J.M. Snyder, *Generative
Modelling for Computer
Graphics and CAD*,
Academic Press, 1992.



A complicated shape is thus generated by a general cross-section and three curves. Clearly implicit in this example is a reliance on a user/designer being able to visualize the final required shape in three-dimensions so that he is able to specify the appropriate shape curves. Although for the turbine blade example this may seem a somewhat tall order, we should bear in mind that shapes of such complexity are the domain of professional engineers where the use of such generative models for shape specification will not be unfamiliar.

Certain practical problems emerge when we generalize to curved spines. There are three difficulties in allowing curved spines that immediately emerge. These are illustrated in Figure 2.9. Figure 2.9(a) shows a problem in the curve to polygon procedure. Here it is seen that the size of the polygonal primitives depends on the excursion of the spine curve. The other is how do we orient the cross-section with respect to a varying spine (Figure 2.9(b))? And, finally, how do we prevent cross-sections self-intersecting (Figure 2.9(c))? It is clear that this will occur as soon as the radius of curvature of the path of any points traced out by the cross-sectional curve exceeds the radius of curvature of the spine. We will now look at approaches to these problems.

Figure 2.8
Snyder's affine transformation surface. The generating curves are shown for a single turbine blade. Source: J.M. Snyder, *Generative Modelling for Computer Graphics and CAD*, Academic Press, 1992.

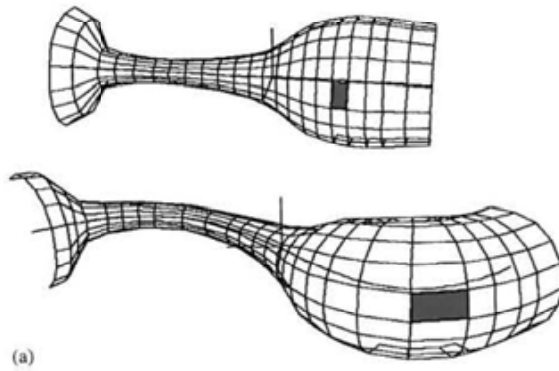
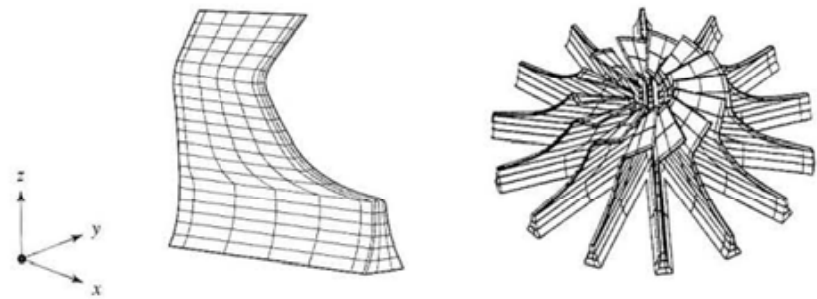
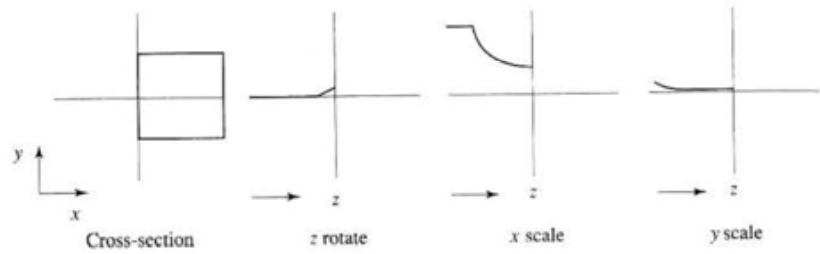
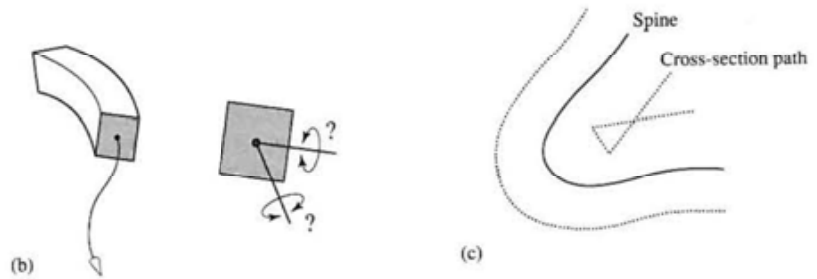


Figure 2.9
Three problems in cross-sectional sweeping. (a) Controlling the size of the polygons can become problematic. (b) How should the cross-section be oriented with respect to the spine curve? (c) Self-intersection of the cross-section path.



Consider a parametrically defined cubic along which the cross-section is swept. This can be defined (see Section 3.1) as:

$$\mathbf{Q}(u) = \mathbf{a}u^3 + \mathbf{b}u^2 + \mathbf{c}u + \mathbf{d}$$

Now if we consider the simple case of moving a constant cross-section without twisting it along the curve we need to define intervals along the curve at which the cross-section is to be placed and intervals around the cross-section curve. When we have these we can step along the spine intervals and around the cross-section intervals and output the polygons.

Consider the first problem. Dividing u into equal intervals will not necessarily give the best results. In particular the points will not appear at equal intervals along the curve. A procedure known as arc length parametrization divides the curve into equal intervals, but this procedure is not straightforward. Arc length parametrization may also be inappropriate. What is really required is a scheme that divides the curve into intervals that depend on the curvature of the curve. When the curvature is high the rate of polygon generation needs to be increased so that more polygons occur when the curvature twists rapidly. The most direct way to do this is to use the curve subdivision algorithm (see Section 4.2.3) and subdivide the curve until a linearity test is positive.

Now consider the second problem. Having defined a set of sample points we need to define a reference frame or coordinate system at each. The cross-section is then embedded in this coordinate system. This is done by deriving three mutually orthogonal vectors that form the coordinate axes. There are many possibilities.

A common one is the Frenet frame. The Frenet frame is defined by the origin or sample point, \mathbf{P} , and three vectors \mathbf{T} , \mathbf{N} and \mathbf{B} (Figure 2.10). \mathbf{T} is the unit length tangent vector:

$$\mathbf{T} = \mathbf{V}/|\mathbf{V}|$$

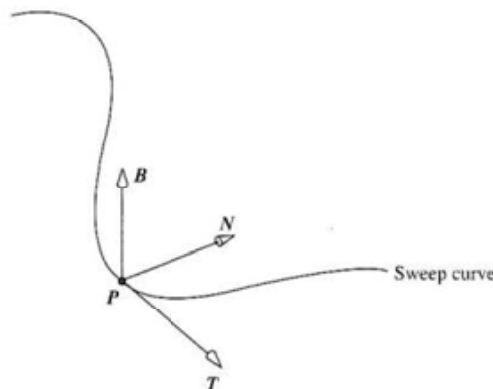


Figure 2.10
The Frenet frame at sample point \mathbf{P} on a sweep curve.

where \mathbf{V} is the derivative of the curve:

$$\mathbf{V} = 3\mathbf{a}u^2 + 2\mathbf{b}u + \mathbf{c}$$

The principal normal \mathbf{N} is given by:

$$\mathbf{N} = \mathbf{K}/|\mathbf{K}|$$

where:

$$\mathbf{K} = \mathbf{V} \times \mathbf{A} \times \mathbf{V}/|\mathbf{V}|^4$$

and \mathbf{A} is the second derivative of the curve:

$$\mathbf{A} = 6\mathbf{a}u + 2\mathbf{b}$$

Finally \mathbf{B} is given by:

$$\mathbf{B} = \mathbf{T} \times \mathbf{N}$$

2.1.5

Procedural polygon mesh objects – fractal objects

In this section we will look at a common example of generating polygon mesh objects procedurally. Fractal geometry is a term coined by Benoit Mandelbrot (1977; 1982). The term was used to describe the attributes of certain natural phenomena, for example, coastlines. A coastline viewed at any level of detail – at microscopic level, at a level where individual rocks can be seen or at ‘geographical’ level, tends to exhibit the same level of jaggedness; a kind of statistical self-similarity. Fractal geometry provides a description for certain aspects of this ubiquitous phenomenon in nature and its tendency towards self-similarity.

In three-dimensional computer graphics, fractal techniques have commonly been used to generate terrain models and the easiest techniques involve subdividing the facets of the objects that consist of triangles or quadrilaterals. A recursive subdivision procedure is applied to each facet, to a required depth or level of detail, and a convincing terrain model results. Subdivision in this context means taking the midpoint along the edge between two vertices and perturbing it along a line normal to the edge. The result of this is to subdivide the original facets into a large number of smaller facets, each having a random orientation in three-dimensional space about the original facet orientation. The initial global shape of the object is retained to an extent that depends on the perturbation at the subdivision and a planar four-sided pyramid might turn into a ‘Mont Blanc’ shaped object.

Most subdivision algorithms are based on a formulation by Fournier *et al.* (1982) that recursively subdivides a single line segment. This algorithm was developed as an alternative to more mathematically correct but expensive procedures suggested by Mandelbrot. It uses self-similarity and conditional expectation properties of fractional Brownian motion to give an estimate of the increment of the stochastic process. The process is also Gaussian and the only parameters needed to describe a Gaussian distribution are the mean (conditional expectation) and the variance.

A procedure recursively subdivides a line $(t_1, f_1), (t_2, f_2)$ generating a scalar displacement of the midpoint of the line in a direction normal to the line (Figure 2.11(a)).

To extend this procedure to, say, triangles or quadrilaterals in three-dimensional space, we treat each edge in turn generating a displacement along a midpoint vector that is normal to the plane of the original facet (Figure 2.11(b)). Using this technique we can take a smooth pyramid, say, made of large triangular faces and turn it into a rugged mountain.

Fournier categorizes two problems in this method – as internal and external consistency. Internal consistency requires that the shape generated should be the same whatever the orientation in which it is generated, and that coarser

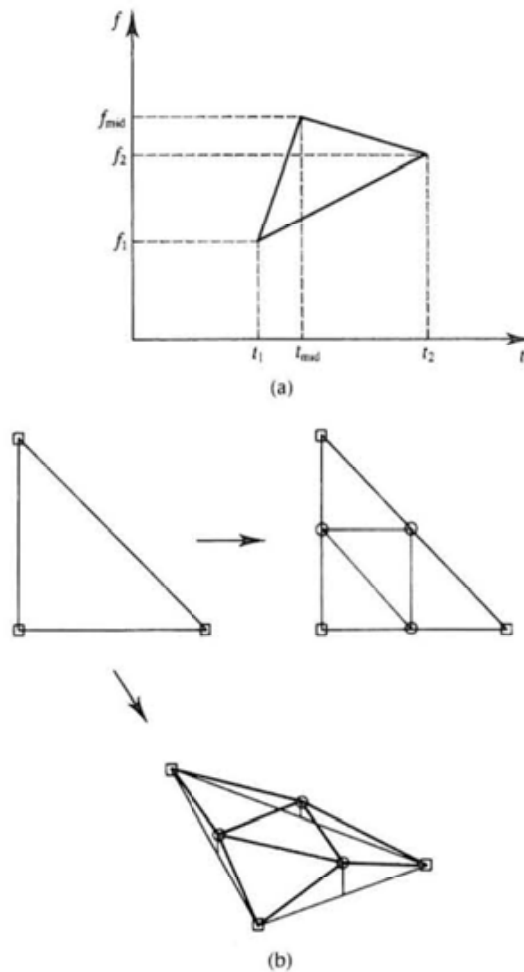


Figure 2.11
 An example of procedural generation of polygon mesh objects – fractal terrain.
 (a) Line segment subdivision. (b) Triangle subdivision.

details should remain the same if the shape is replotted at greater resolution. To satisfy the first requirement, the Gaussian randoms generated must not be a function of the position of the points, but should be unique to the point itself. An invariant point identifier needs to be associated with each point. This problem can be solved in terrain generation by giving each point a key value used to index a Gaussian random number. A hash function can be used to map the two keys of the end points of a line to a key value for the midpoint. Scale requirements of internal consistency means that the same random numbers must always be generated in the same order at a given level of subdivision.

External consistency is harder to maintain. Within the mesh of triangles every triangle shares each of its sides with another; thus the same random displacements must be generated for corresponding points of different connecting triangles. This is already solved by using the key value of each point and the hash function, but another problem still exists, that of the direction of the displacement.

If the displacements are along the surface normal of the polygon under consideration, then adjacent polygons which have different normals (as is, by definition, always the case) will have their midpoints displaced into different positions. This causes gaps to open up. A solution is to displace the midpoint along the average of the normals to all the polygons that contain it but this problem occurs at every level of recursion and is consequently very expensive to implement. Also, this technique would create an unsatisfactory skyline because the displacements are not constrained to one direction. A better skyline is obtained by making all the displacements of points internal to the original polygon in a direction normal to the plane of the original polygon. This cheaper technique solves all problems relating to different surface normals, and the gaps created by them. Now surface normals need not be created at each level of recursion and the algorithm is considerably cheaper because of this.

Another two points are worth mentioning. Firstly, note that polygons should be constant shaded without calculating vertex normals – discontinuities between polygons should not be smoothed out. Secondly, consider colour. The usual global colour scheme uses a height-dependent mapping. In detail, the colour assigned to a midpoint is one of its end point's colours. The colour chosen is determined by a Boolean random which is indexed by the key value of the midpoint. Once again this must be accessed in this way to maintain consistency, which is just as important for colour as it is for position.

2.2

Constructive solid geometry (CSG) representation of objects

We categorized the previous method – polygon mesh – as a machine representation which also frequently functions as a user representation. The CSG approach is very much a user representation and requires special rendering techniques or the conversion to a polygon mesh model prior to representation. It is a high-level representation that functions both as a shape representation and a record

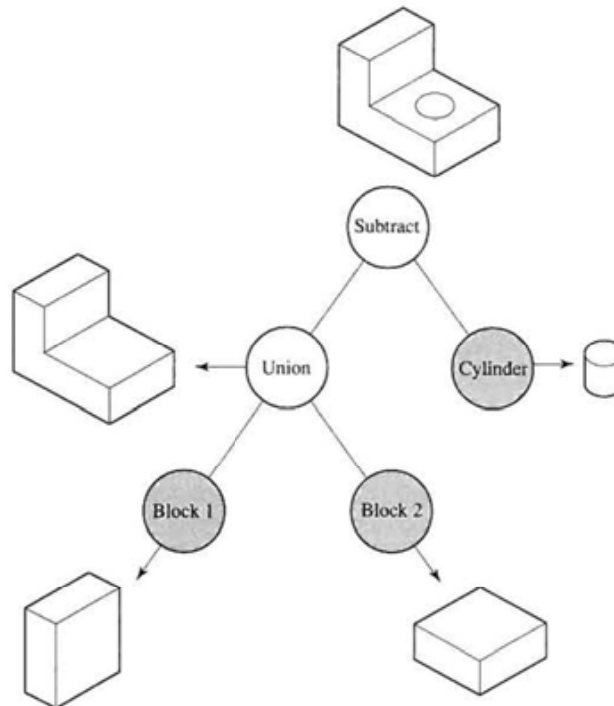
of how it was built up. The 'logic of the shape' in this representation is in how the final shape can be made or represented as a combination of primitive shapes. The designer builds up a shape by using the metaphor of three-dimensional building blocks and a selection of ways in which they can be combined. The high-level nature of the representation imposes a certain burden on the designer. Although with hindsight the logic of the parts in Figure 2.14 is apparent; the design of complex machine parts using this methodology is a demanding occupation.

The motivation for this type of representation is to facilitate an interactive mode for solid modelling. The idea is that objects are usually parts that will eventually be manufactured by casting, machining or extruding and they can be built up in a CAD program by using the equivalent (abstract) operations combining simple elementary objects called geometric primitives. These primitives are, for example, spheres, cones, cylinders or rectangular solids and they are combined using (three-dimensional) Boolean set operators and linear transformations. An object representation is stored as an attributed tree. The leaves contain simple primitives and the nodes store operators or linear transformations. The representation defines not only the shape of the object but its modelling history – the creation of the object and its representation become one and the same thing. The object is built up by adding primitives and causing them to combine with existing primitives. Shapes can be added to and subtracted from (to make holes) the current shape. For example, increasing the diameter of a hole through a rectangular solid means a trivial alteration – the radius of the cylinder primitive defining the hole is simply increased. This contrasts with the polygon mesh representation where the same operation is distinctly non-trivial. Even although the constituent polygons of the cylindrical surface are easily accessible in a hierarchical scheme, to generate a new set of polygons means reactivating whatever modelling procedure was used to create the original polygons. Also, account has to be taken of the fact that to maintain the same accuracy more polygons will have to be used.

Boolean set operators are used both as a representational form and as a user interface technique. A user specifies primitive solids and combines these using the Boolean set operators. The representation of the object is a reflection or recording of the user interaction operations. Thus we can say that the modelling information and representation are not separate – as they are in the case of deriving a representation from low-level information from an input device. The low-level information in the case of CSG is already in the form of volumetric primitives. The modelling activity becomes the representation. An example will demonstrate the idea.

Figure 2.12 shows the Boolean operations possible between solids. Figure 2.12(a) shows the union of two solids. If we consider the objects as 'clouds' of points the union operation encloses all points lying within the original two bodies. The second example (Figure 2.12(b)) shows the effect of a difference or subtraction operator. A subtract operator removes all those points in the second body that are contained within the first. In this case a cylinder is defined and

Figure 2.13
A CSG tree reflecting the construction of a simple object made from three primitives.

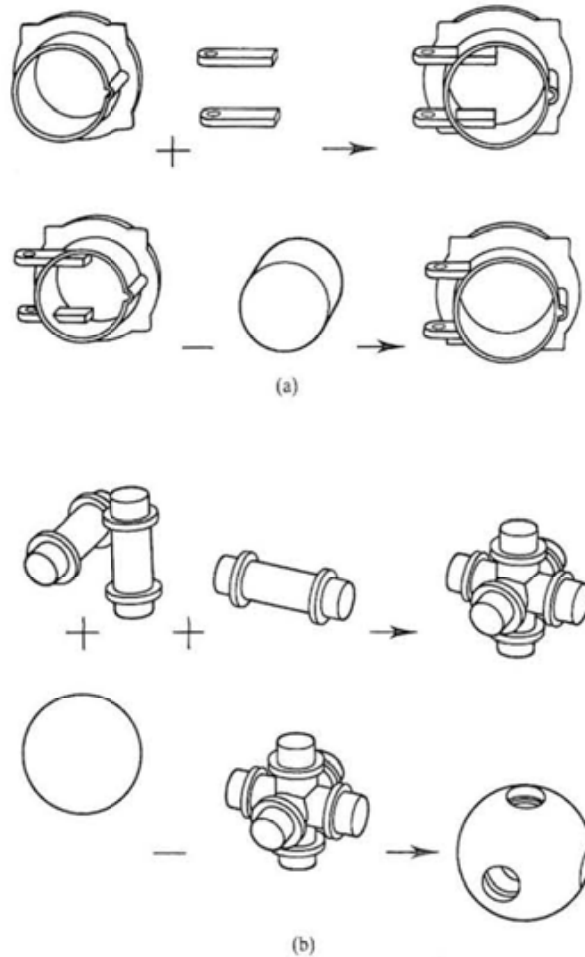


assembly. Thus the only information that has to be stored in the leaves of the tree is the name of the primitive and its dimensions. A node has to contain the name of the operator and the spatial relationship between the child nodes combined by the operator.

The power of Boolean operations is further demonstrated in the following examples. In the first example (Figure 2.14(a)) two parts developed separately are combined to make the desired configuration by using the union operator followed by a difference operator. The second example (Figure 2.14(b)) shows a complex object constructed only from the union of cylinders, which is then used to produce, by subtraction, a complex housing.

Although there are substantial advantages in CSG representation, they do suffer from drawbacks. A practical problem is the computation time required to produce a rendered image of the model. A more serious drawback is that the method imposes limitations on the operations available to create and modify a solid. Boolean operations are global – they affect the whole solid. Local operations, say a detailed modification on one face of a complex object cannot be easily implemented by using set operations. An important local modification required in many objects that are to be designed is blending surfaces. For example, consider the end face of a cylinder joined onto a flat base. Normally for practical manufacturing or aesthetic reasons, instead of the join being a right angle in cross-

Figure 2.14
Examples of geometrically complex objects produced from simple objects and Boolean operations.



section a radius is desired. A radius swept around another curve cannot be represented in a simple CSG system. This fact has led to many solid modellers using an underlying boundary representation. Incidentally there is no reason why Boolean operations cannot be incorporated in boundary representations systems. For example, many systems incorporate Boolean operations but use a boundary representation to represent the object. The trade-off between these two representations has resulted in a debate that has lasted for 15 years. Finally note that a CSG representation is a volumetric representation. The space occupied by the object – its volume – is represented rather than the object surface.

Space subdivision techniques for object representation

Space subdivision techniques are methods that consider the whole of object space and in some way label each point in the space according to object occupancy. However, unlike CSG, which uses a variety of volumetric elements or geometric primitives, space subdivision techniques are based on a single cubic element known as a voxel. A voxel is a volumetric element or primitive and is the smallest cube used in the representation. We could divide up all of world space into regular or cubic voxels and label each voxel according to whether it is in the object or in empty space. Clearly this is very costly in terms of memory consumption. Because of this voxel representation is not usually a preferred mainstream method but is used either because the raw data are already in this form or it is easiest to convert the data into this representation – the case, for example, in medical imagery; or because of the demands of an algorithm. For example, ray tracing in voxel space has significant advantages over conventional ray tracing. This is an example of an algorithmic technique dictating the nature of the object representation. Here, instead of asking the question: ‘does this ray intersect with any objects in the scene?’ which implies a very expensive intersection test to be carried out on each object, we pose the question: ‘what objects are encountered as we track a ray through voxel space?’ This requires no exhaustive search through the primary data structure for possible intersections and is a much faster strategy.

Another example is rendering CSG models (Section 4.3) which is not straightforward if conventional techniques are used. A strategy is to convert the CSG tree into an intermediate data consisting of voxels and render from this. Voxels can be considered as an intermediate representation, most commonly in medical imaging where their use links two-dimensional raw data with the visualization of three-dimensional structures. Alternatively the raw data may themselves be voxels. This is the case with many mathematical modelling schemes of three-dimensional physical phenomena such as fluid dynamics.

The main problem with voxel labelling is the trade-off between the consumption of vast storage costs and accuracy. Consider, for example, labelling square pixels to represent a circle in two-dimensional space. The pixel size /accuracy trade-off is clear here. The same notion extends to using voxels to represent a sphere except that now the cost depends on the accuracy and the cube of the radius. Thus such schemes are only used in contexts where their advantages outweigh their cost. A way to reduce cost is to impose a structural organization on the basic voxel labelling scheme.

The common way of organizing voxel data is to use an octree – a hierarchical data structure that describes how the objects in a scene are distributed throughout the three-dimensional space occupied by the scene. The basic idea is shown in Figure 2.15. In Figure 2.15(a) a cubic space is subject to a recursive subdivision which enables any cubic region of the space to be labelled with a number. This subdivision can proceed to any desired level of accuracy. Figure 2.15(b) shows an object embedded in this space and Figure 2.15(c) shows the subdivision and the

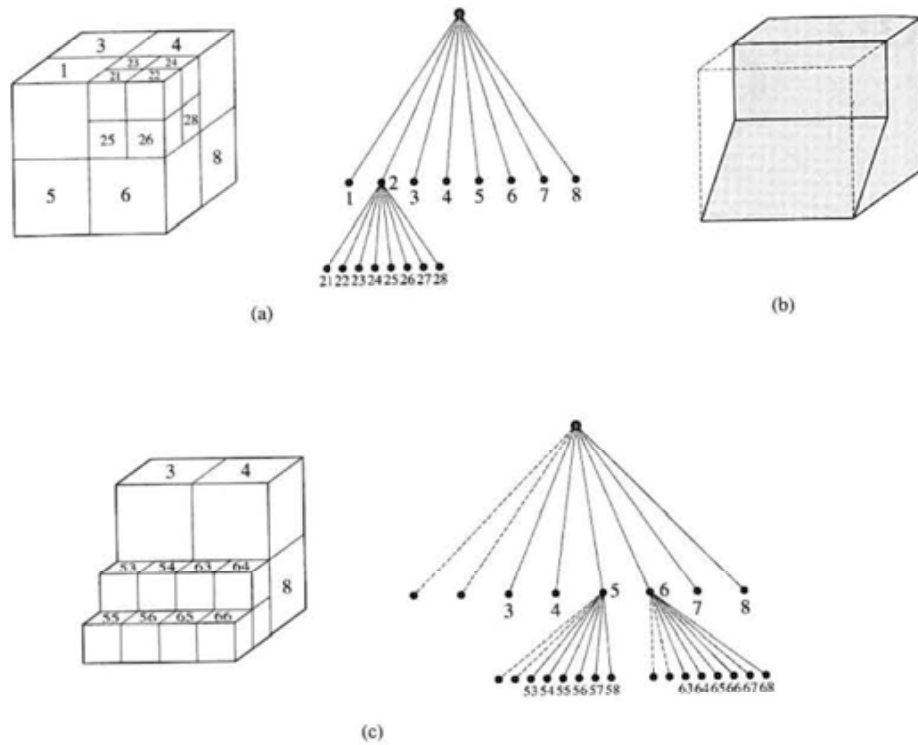


Figure 2.15
 Octree representation.
 (a) Cubic space and labelling scheme, and the octree for the two levels of subdivision. (b) Object embedded in space. (c) Representation of the object to two levels of subdivision.

related octree that labels cubic regions in the space according to whether they are occupied or empty.

There are actually two ways in which the octree decomposition of a scene can be used to represent the scene. Firstly, an octree as described above can be used in itself as a complete representation of the objects in the scene. The set of cells occupied by an object constitute the representation of the object. However, for a complex scene, high resolution work would require the decomposition of occupied space into an extremely large number of cells and this technique requires enormous amounts of data storage. A common alternative is to use a standard data structure representation of the objects and to use the octree as a representation of the distribution of the objects in the scene. In this case, a terminal node of a tree representing an occupied region would be represented by a pointer to the data structure for any object (or part of an object) contained within that region. Figure 2.16 illustrates this possibility in the two-dimensional case. Here the region subdivision has stopped as soon as a region is encountered that intersects only one object. A region represented by a terminal node is not necessarily completely occupied by the object associated with that region. The shape of the object within the region would be described by its data structure representation. In the case of a surface model representation of a scene, the

'objects' would be polygons or patches. In general, an occupied region represented by a terminal node would intersect with several polygons and would be represented by a list of pointers into the object data structures. Thus unlike the other techniques that we have described octrees are generally not self-contained representational methods. They are instead usually part of a hybrid scheme.

2.3.1

Octrees and polygons

As we have already implied, the most common use of octrees in computer graphics is not to impose a data structure, on voxel data, but to organize a scene containing many objects (each of which is made up of many polygons) into a structure of spatial occupancy. We are not representing the objects using voxels, but considering the rectangular space occupied as polygons as entities which are represented by voxel space. As far as rendering is concerned we enclose parts of the scene, at some level of detail, in rectangular regions in the sense of Figure 2.16. For example, we may include groups of objects, single objects, parts of objects or even single polygons in an octree leaf node. This can greatly speed up many aspects of rendering and many rendering methods, particularly ray tracing as we have already suggested.

We will now use ray tracing as a particular example. The high inherent cost in naive ray tracing resides in intersection testing. As we follow a ray through the scene we have to find out if it collides with any object in the scene (and what the position of that point is). In the case that each ray is tested against all objects in the scene, where each object test implies testing against each polygon in the object, the rendering time, for scenes of reasonable complexity, becomes unacceptably high. If the scene is decomposed into an octree representation, then tracing a ray means tracking, using an incremental algorithm from voxel to voxel. Each voxel contains pointers to polygons that it contains and the ray is tested against these. Intersection candidates are reduced from n to m , where:

$$n = \sum_{\text{objects}} \text{polygon count for object}$$

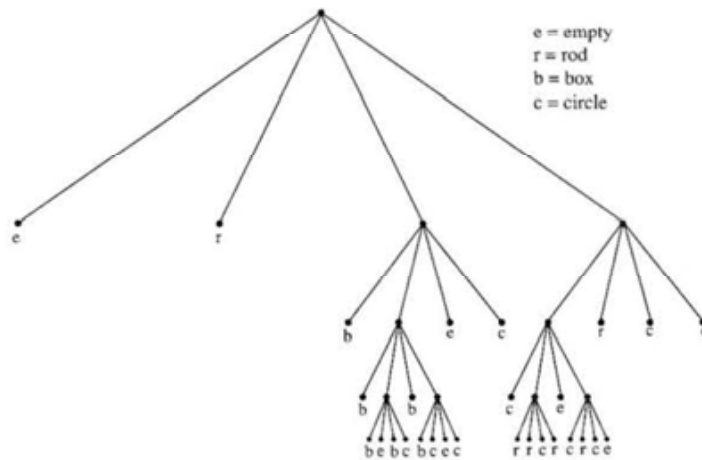
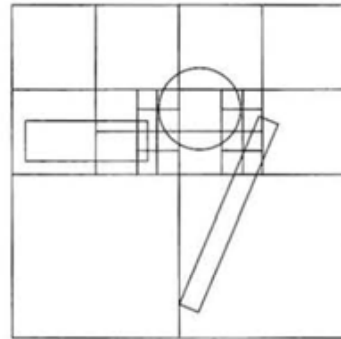
and m is the number of candidate polygons contained by the octree leaf.

However, decomposing a scene into an octree is an expensive operation and has to be judiciously controlled. It involves finding the 'minmax' coordinates of each polygon (the coordinates of its bounding box) and using these as an entity in the decomposition. Two factors that can be used to control the decomposition are:

- (1) The minimum number of candidate polygons per node. The smaller this factor, the greater is the decomposition and fewer intersection tests are made by a ray that enters a voxel. The total number of intersection tests per voxel for the entire rendering is approximately given by:

number of rays entering the voxel \times (0.5 \times number of polygons in voxel)
 assuming that on average a ray tests 50% of the candidate polygons before it finds an intersection.

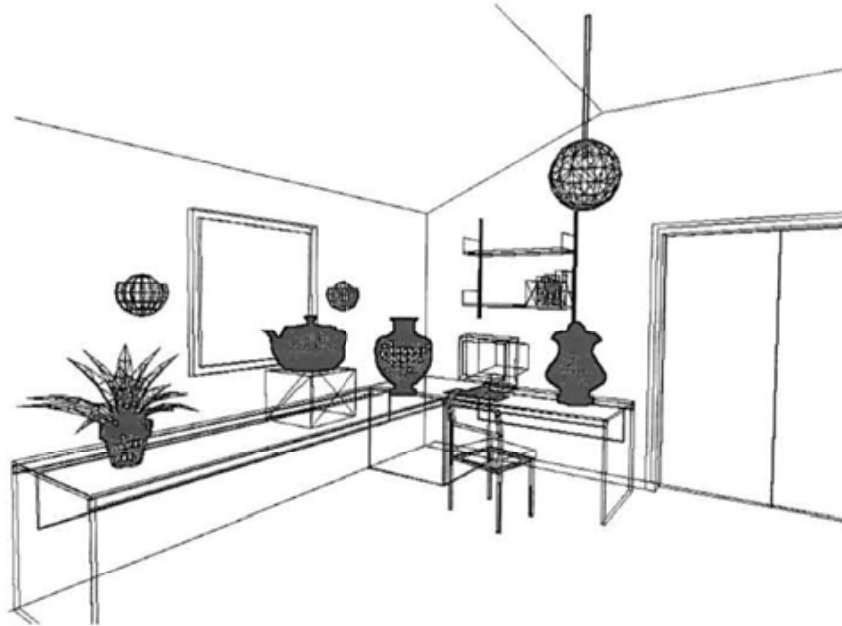
Figure 2.16
 Quadtree representation of a two-dimensional scene down to the level of cells containing at most a single object. Terminal nodes for cells containing objects would be represented by a pointer to a data structure representation of the object.



- (2) The maximum octree depth. The greater the depth the greater the decomposition and the fewer the candidate polygons at a leaf node. Also, because the size of a voxel decreases by a factor of 8 at every level, the fewer the rays that will enter the voxel for any given rendering.

In general the degree of decomposition should not be so great that the savings gained on intersection are wiped out by the higher costs of tracking a ray through decomposed space. Experience has shown that a default value of 8 for the above two factors gives good results in general for an object (or objects) distributed evenly throughout the space. Frequently scenes are rendered where this condition does not hold. Figure 2.17 shows an example where a few objects with high polygon count are distributed around a room whose volume is large compared to the space occupied by the objects. In this case octree subdivision will proceed to a high depth subdividing mostly empty space.

Figure 2.17
A scene consisting of a few objects of high polygon count. The objects are small compared with the volume of the room.



2.3.2

BSP trees

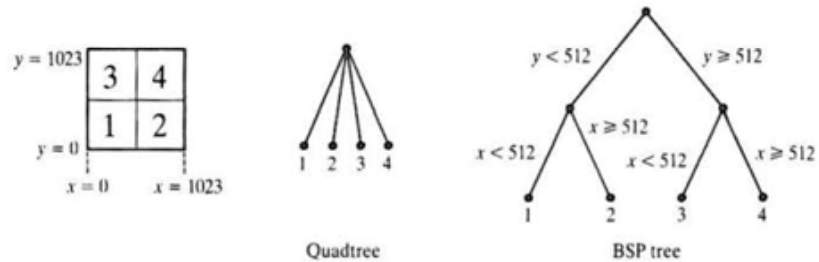
An alternative representation to an octree is a BSP or binary space partitioning tree. Each non-terminal node in the BSP tree represents a single partitioning plane that divides the space into two. A two-dimensional analogue illustrating the difference is shown in Figure 2.18. A BSP tree is not a direct object representation (although in certain circumstances it can be). Instead it is a way of partitioning space for a particular purpose – most commonly hidden surface removal. Because of this it is difficult and somewhat pointless to discuss BSP trees without dealing at the same time with HSR (see Chapter 6).

The properties of partitioning planes that can be exploited in computer graphics scenes are:

- Any object on one side of a plane cannot intercept any object on the other side.
- Given a view point in the scene space, objects on the same side as the viewer are nearer than any objects on the other side.

When a BSP tree is used to represent a subdivision of space into cubic cells, it shows no significant advantage over a direct data structure encoding of the octree. It is the same information encoded in a different way. However, nothing said above requires that the subdivision should be into cubic cells. In fact the

Figure 2.18
Quadtree and BSP tree
representations of a one-
level subdivision of a two-
dimensional region.



idea of a BSP tree was originally introduced in Fuchs (1980) where the planes used to subdivide space could be at any orientation. We revisit BSP trees in the context of hidden surface removal (Chapter 6).

2.3.3

Creating voxel objects

One of the mainstream uses of voxel objects is in volume rendering in medical imagery. The source data in such applications consist of a set of parallel planes of intensity information collected from consecutive cross-sections from some part of a body, where a pixel in one such plane will represent, say, the X-ray absorption of that part of the body that the pixel physically corresponds to. The problem is how to convert such a stack of planar two-dimensional information into a three-dimensional rendered object. Converting the stack of planes to a set of voxels is the most direct way to solve this problem. Corresponding pixels in two consecutive planes are deemed to form the top and bottom face of a voxel and some operation is performed to arrive at a single voxel value from the two pixel values. The voxel representation is used as an intermediary between the raw collected data, which are two-dimensional, and the required three-dimensional visualization. The overall process from the collection of raw data, through the conversion to a voxel representation and the rendering of the voxel data is the subject of Chapter 13.

Contours collected by a laser ranger can be converted into a voxel representation instead of into a polygon mesh representation. However, this may result in a loss of accuracy compared with using a skinning algorithm.

2.4

Representing objects with implicit functions

As we have already pointed out, representing a whole object by a single implicit formula is restricted to certain objects such as spheres. Nevertheless such a representation does find mainstream use in representing 'algorithmic' objects known as bounding volumes. These are used in many different contexts in computer graphics as a complexity limiting device.

A representation developed from implicit formulae is the representation of objects by using the concept of implicitly defined objects as components. (We use the term component rather than primitive because the object is not simply a set of touching spheres but a surface derived from such a collection.)

Implicit functions are surfaces formed by the effect of primitives that exert a field of influence over a local neighbourhood. For example, consider a pair of point heat sources shown in Figure 2.19. We could define the temperature in their vicinities as a field function where, for each in isolation, we have isothermal contours as spherical shells centred on each source. Bringing the two sources within influence of each other defines a combined global scalar field, the field of each source combining with that of the other to form a composite set of isothermal contours as shown. Such a scalar field, due to the combined effect of a number of primitives is used to define a modelling surface in computer graphics. Usually we consider an isosurface in the field to be the boundary of a volume which is the object that we desire to model. Thus we have the following elements in any implicit function modelling system:

- A generator or primitive for which a distance function $d(\mathbf{P})$ can be defined for all points \mathbf{P} in the locality of the generator.
- A 'potential' function $f(d(\mathbf{P}))$ which returns a scalar value for a point \mathbf{P} distance $d(\mathbf{P})$ from the generator. Associated with the generator can be an area of influence outside of which the generator has no influence. For a point generator this is usually a sphere. An example of a potential function is:

$$f(\mathbf{P}) = \left(1 - \frac{d^2}{R^2}\right)^2 \quad d \leq R$$

where d is the distance of the point to the generator and R is its radius of influence.

- A scalar field $F(\mathbf{P})$ which determines the combined effect of the individual potential functions of the generators. This implies the existence of a blending method which in the simplest case is addition – we evaluate a scalar field by evaluating the individual contributions of each generator at a point \mathbf{P} and adding their effects together.

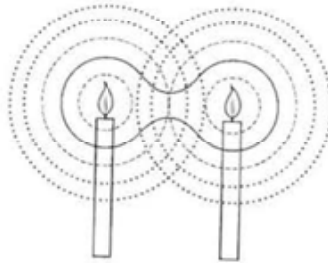


Figure 2.19
An isosurface of equal
temperature around two
heat sources (solid line).

- An isosurface of the scalar field which is used to represent the physical surface of the object that we are modelling.

An example (Figure 2.20 Colour Plate) illustrates the point. The Salvador Dali imitation on the left is an isosurface formed by point generators disposed in space as shown on the right. The radius of each sphere is proportional to the radius of influence of each generator. The dark spheres represent negative generators which are used to 'carve' concavities in the model. (Although we can form concavities by using only positive generators, it is more convenient to use negative ones as we require far fewer spheres.) The example illustrates the potential of the method for modelling organic shapes.

Deformable object animation can be implemented by displaying or choreographing the points that generate the object. The problem with using implicit functions in animation is that there is not a good intuitive link between moving groups of generators and the deformation that ensues because of this. Of course, this general problem is suffered by all modelling techniques where the geometry definition and the deformation method are one and the same thing.

In addition to this general problem, unwanted blending and unwanted separation can occur when the generators are moved with respect to each other and the same blending method retained.

A significant advantage of implicit functions in an animation context is the ease of collision detection that results from an easy inside-outside function. Irrespective of the complexity of the modelled surface a single scalar value defines the isosurface and a point \mathbf{P} is inside the object volume or outside it depending on whether $F(\mathbf{P})$ is less than or greater than this value.

2.5

Scene management and object representation

As the demand for high quality real time computer graphics continues to grow, from applications like computer games and virtual reality, the issue of efficient scene management has become increasingly important. This means that representational forms have to be extended to collections of objects; in other words the scene has to be considered as an object itself. This has generally meant using hierarchical or tree structures, such as BSP trees to represent the scene down to object and sub-object level. As rendering has increasingly migrated into real time applications, efficiency in culling and hidden surface removal has become as important as efficient rendering for complex scenes. With the advent of 3D graphics boards for the PC we are seeing a trend develop where the basic rendering of individual objects is handled by hardware and the evaluation of which objects are potentially visible is computed by software. (We will look into culling and hidden surface removal in Chapters 5 and 6). An equally important efficiency measure for objects in complex scenes has come to be known as Level of Detail, or LOD, and it is this topic that we will now examine.

2.5.1

Polygon mesh optimization

As we have discussed, polygon mesh models are well established as the *de facto* standard representational form in computer graphics but they suffer from significant disadvantages, notably that the level of detail, or number of polygons, required to synthesize the object for a high quality rendition of a complex object is very large. If the object is to be rendered on screen at different viewing distances the pipeline has to process thousands of polygons that project onto a few pixels on the screen. As the projected polygon size decreases, the polygon overheads become significant and in real time applications this situation is intolerable. High polygon counts per object occur either because of object complexity or because of the nature of the modelling system. Laser scanners and the output from programs like the marching cubes algorithm (which converts voxels into polygons) are notorious for producing very large polygon counts. Using such facilities almost always results in a model that, when rendered, is indistinguishable from a version rendered from a model with far fewer faces.

As early as 1976, one of the pioneers of 3D computer graphics, James H. Clark, wrote:

It makes no sense to use 500 polygons in describing an object if it covers only 20 raster units of the display . . . For example, when we view the human body from a very large distance, we might need to present only specks for the eyes, or perhaps just a block for the head, totally eliminating the eyes from consideration . . . these issues have not been addressed in a unified way.

Did Clark realize that not many years after he had written these words that 500 000 polygon objects would become fairly commonplace and that complex scenes might contain millions of polygons?

Existing systems tend to address this problem in a somewhat ad hoc manner. For example, many cheap virtual reality systems adopt a two- or three-level representation switching in surface detail, such as the numbers on the buttons of a telephone as the viewer moves closer to it. This produces an annoying visual disturbance as the detail blinks on and off. More considered approaches are now being proposed and lately there has been a substantial increase in the number of papers published in this area.

Thus mesh optimization seems necessary and the problem cannot be dismissed by relying on increased polygon throughput of the workstations of the future. The position we are in at the moment is that mainstream virtual reality platforms produce a visually inadequate result even from fairly simple scenes. We have to look forward not only to dealing with the defects in the image synthesis of such scenes, but also to being able to handle scenes of real world complexity implying many millions of polygons. The much vaunted 'immersive' applications of virtual reality will never become acceptable unless we can cope with scenes of such complexity. Current hardware is very far away from being able to deal with a complex scene in real time to the level of quality attainable for single object scenes.

An obvious solution to the problem is to generate a polygon mesh at the final level of detail and then use this representation to spawn a set of coarser descriptions. As the scene is rendered an appropriate level of detail is selected. Certain algorithms have emerged from time to time in computer graphics that use this principle. An example of a method that facilitates a polygon mesh at any level of detail is bi-cubic parametric patches (see Section 4.2.2). Here we take a patch description and turn it into a polygon description. At the same time we can easily control the number of polygons that are generated for each patch and relate this to local surface curvature. This is exactly what is done in patch rendering where a geometric criterion is used to control the extent of the subdivision and produce an image free of geometric aliasing (visible polygon edges in silhouette). The price we pay for this approach is the expense and difficulty of getting the patch description in the first place. But in any case we could build the original patch representation and construct a pyramid of polygon mesh representations off-line.

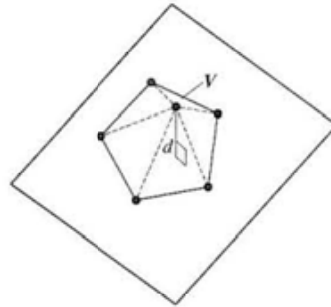
The idea of storing a 'detail pyramid' and accessing an appropriate level is established in many application areas. Consider the case of mip-mapping, for example (see Chapter 8). Here texture maps are stored in a detail hierarchy and a fine detail map selected when the projection of the map on the screen is large. In the event that the map projects onto just one pixel, then a single pixel texture map – the average of the most detailed map – is selected. Also, in this method the problem of avoiding a jump when going from one level to another is carefully addressed and an approximation to a continuous level of detail is obtained by interpolation between two maps.

The diversity of current approaches underlines the relative newness of the field. A direct and simple approach for triangular meshes derived from voxel sets was reported by Schroeder *et al.* in 1992. Here the algorithm considers each vertex on a surface. By looking at the triangles that contribute to, or share, the vertex, a number of criteria can be enumerated and used to determine whether these triangles can be merged into a single one exclusive of the vertex under consideration. For example, we can invoke the 'reduce the number of triangles where the surface curvature is low' argument by measuring the variance in the surface normals of the triangles that share the vertex. Alternatively we could consider the distance from the vertex to an (average) plane through all the other vertices of the sharing triangles (Figure 2.21). This is a local approach that considers vertices in the geometry of their immediate surroundings.

A more recent approach is the work of Hoppe (1996) which we will now describe. Hoppe gives an excellent categorization of the problems and advantages of mesh optimization, listing these as follows:

- Mesh simplification – reducing the polygons to a level that is adequate for the quality required. (This, of course, depends on the maximum projection size of the object on the screen.)
- Level of detail approximation – a level is used that is appropriate to the viewing distance. In this respect, Hoppe adds: 'Since instantaneous

Figure 2.21
A simple vertex deletion criterion. Delete V ? Measure d , the distance from V to the (average) plane through the triangles that share V .



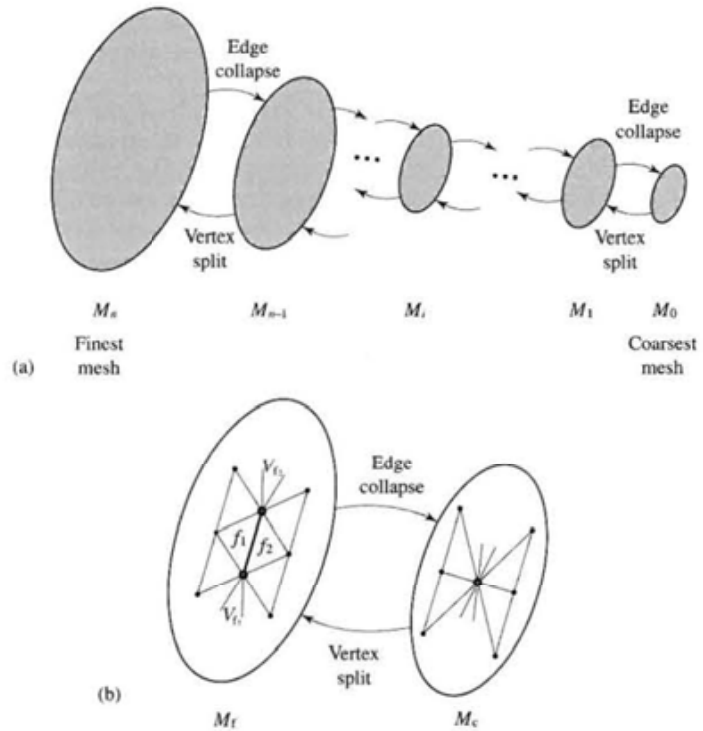
switching between LOD meshes may lead to perceptible “popping”, one would like to construct smooth visual transitions, *geomorphs*, between meshes at different resolutions’.

- Progressive transmission – this is a three-dimensional equivalent of the common progressive transmission modes used to transmit two-dimensional imagery over the Internet. Successive LOD approximations can be transmitted and rendered at the receiver.
- Mesh compression – analogous to two-dimensional image pyramids. We can consider not only reducing the number of polygons but also minimizing the space that any LOD approximation occupies. As in two-dimensional imagery, this is important because an LOD hierarchy occupies much more memory than a single model stored at its highest level of detail.
- Selective refinement – an LOD representation may be used in a context-dependent manner. Hoppe gives the example of a user flying over a terrain where the terrain mesh need only be fully detailed near the viewer.

Addressing mesh compression, Hoppe takes a ‘pyramidal’ approach and stores the coarsest level of detail approximation together, for each higher level, with the information required to ascend from a lower to a higher level of detail. To make the transition from a lower to a higher level the reverse of the transformation that constructed the hierarchy from the highest to the lowest level is stored and used. This is in the form of a vertex split – an operation that adds an additional vertex to the lower mesh to obtain the next mesh up the detail hierarchy. Although Hoppe originally considered three mesh transformations – an edge collapse, an edge split and an edge swap – he found that an edge collapse is sufficient for simplifying meshes.

The overall scheme is represented in Figure 2.22(a) which shows a detail pyramid which would be constructed off-line by a series of edge collapse transformations that take the original mesh M_n and generate through repeated edge collapse transformations the final or coarsest mesh M_0 . The entire pyramid can then be stored as M_0 together with the information required to generate, from M_0 to any finer level M_i in the hierarchy – mesh compression. This inter-level transformation is the reverse of the edge collapse and is the information required

Figure 2.22
Hoppe's (1996) progressive
mesh scheme based on
edge collapse
transformations.



for a vertex split. Hoppe quotes an example of an object with 13 546 faces which was simplified to an \$M_0\$ of 150 faces using 6698 edge collapse transformations. The original data are then stored as \$M_0\$ together with the 6698 vertex split records. The vertex split records themselves exhibit redundancy and can be compressed using classical data compression techniques.

Figure 2.22(b) shows a single edge collapse between two consecutive levels. The notation is as follows: \$V_{f1}\$ and \$V_{f2}\$ are the two vertices in the finer mesh that are collapsed into one vertex \$V_c\$ in the coarser mesh, where

$$V_c \in \left\{ V_{f1}, V_{f2}, \frac{V_{f1} + V_{f2}}{2} \right\}$$

From the diagram it can be seen that this operation implies the collapse of the two faces \$f_1\$ and \$f_2\$ into new edges.

Hoppe defines a continuum between any two levels of detail by using a blending parameter \$\alpha\$. If we define:

$$d = \frac{V_{f1} + V_{f2}}{2}$$

then we can generate a continuum of geomorphs between the two levels by having the edge shrink under control of the blending parameter as:

$$V_{11} := V_{11} + \alpha d \quad \text{and} \quad V_{12} := V_{12} - \alpha d$$

Texture coordinates can be interpolated in the same way as can scalar attributes associated with a vertex such as colour.

The remaining question is: how are the edges selected for collapse in the reduction from M_i to M_{i-1} ? This can be done either by using a simple heuristic approach or by a more rigorous method that measures the difference between a particular approximation and a sample of the original mesh. A simple metric that can be used to order the edges for collapse is:

$$\frac{|V_{11} - V_{12}|}{|\mathbf{N}_{11} \cdot \mathbf{N}_{12}|}$$

that is, the length of the edge divided by the dot product of the vertex normals. On its own this metric will work quite well, but if it is continually applied the mesh will suddenly begin to 'collapse' and a more considered approach to edge selection is mandatory. Figure 2.23 is an example that uses this technique.

Hoppe casts this as an energy function minimization problem. A mesh M is optimized with respect to a set of points X which are the vertices of the mesh M_n together (optionally) with points randomly sampled from its faces. (Although this is a lengthy process it is, of course, executed once only as an off-line pre-process.) The energy function to be minimized is:

$$E(M) = E_{\text{dist}}(M) + E_{\text{spring}}(M)$$

where

$$E_{\text{dist}} = \sum_i d^2(x_i, M)$$

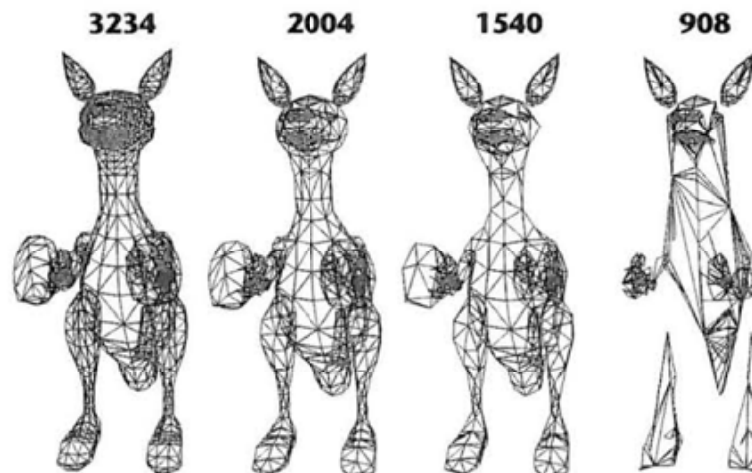


Figure 2.23
The result of applying the simple edge elimination criterion described in the text – the model eventually breaks up.

is the sum of the squared distances from the points X to the mesh – when a vertex is removed this term will tend to increase.

$$E_{\text{spring}}(M) = \sum \kappa \|v_j - v_k\|^2$$

is a spring energy term that assists the optimization. It is equivalent to placing on each edge a spring of rest length zero and spring constant κ .

Hoppe orders the optimization by placing all (legal) edge collapse transformations into a priority queue, where the priority of each transformation is its estimated energy cost ΔE . In each iteration, the transformation at the front of the queue (lowest ΔE) is performed and the priorities of the edges in the neighbourhood of this transformation are recomputed. An edge collapse transformation is only legal if it does not change the topology of the mesh. For example, if V_{i1} and V_{i2} are boundary vertices, the edge $[V_{i1}, V_{i2}]$ must be a boundary edge – it cannot be an internal edge connecting two boundary points.

2.6

Summary

Object representations have evolved under a variety of influences – ease of rendering, ease of shape editing, suitability for animation, dependence on the attributes of raw data and so on. There is no general solution that is satisfactory for all practical applications and the most popular solution that has served us for so many years – the polygon mesh – has significant disadvantages as soon as we leave the domain of static objects rendered off-line. We complete this chapter by listing the defining attributes of any representation. These allow a (very) general comparison between the methods. (For completeness we have included comments on bi-cubic patches which are dealt with in the next chapter.)

- **Creation of object/representation** A factor that is obviously context dependent. We have the methods which can create representations automatically from physical data (polygon mesh from range data via a skinning algorithm, bi-cubic parametric patches via interpolation of surface data). Other methods map input data directly into a voxel representation. Some methods are suitable for interactive creation (CSG and bi-cubic parametric patches) and some can be created by interacting with a ‘mathematically’ based interactive facility such as sweeping a cross-section along a spine (polygon mesh and bi-cubic parametric patches).
- **Nature of the primitive elements** The common forms are either methods that represent surfaces – boundary representations (polygon mesh and bi-cubic parametric patches) or volumes (voxels and CSG).
- **Accuracy** Representations are either exact or approximate. Polygon meshes are approximate representations but their accuracy can be increased to any degree at the expense of an expansion in the data. Increasing the accuracy of a polygon mesh representation in an intelligent way is difficult. The easy ‘brute force’ approach – throwing more polygons at the shape – may result in areas being ‘over represented’. Bi-cubic patches can either be exact or approximate depending on the application. Surface interpolation will result in

an approximation but designing a car door panel using a single patch results in an exact representation. CSG representations are exact but we need to make two qualifications. They can only describe that subset of shapes that is possible by combining the set of supplied primitives. The representation is abstract in that it is just a formula for the composite object – the geometry has to be derived from the formula to enable a visualization of the object.

- **Accuracy vs data volume** There is always a trade-off between accuracy and data volume – at least as far as the rendering penalty is concerned. To increase the accuracy of a boundary representation or a volume representation we have to increase the number of low-level elements. Although the implicit equation of a sphere is 100% accurate and compact, it has to be converted for rendering using some kind of geometric sampling procedure which generates low-level elements.
- **Data volume vs complexity** There is usually a trade-off also between data volume and the complexity of the representation which has practical ramifications in the algorithms that operate with the representation. This is best exemplified by comparing polygon meshes with their counterpart using bi-cubic parametric patches.
- **Ease of editing/animation** This can mean retrospective editing of an existing model or shape deformation techniques in an animation environment. The best method for editing the shape of static objects is, of course, the CSG representation – it was designed for this. Editing bi-cubic parametric patches is easy or difficult depending on the complexity of the shape and the desired freedom of the editing operations. In this respect editing a single patch is easy, editing a net of patches is difficult. None of the representation methods that we have described is suitable for shape-changing in animated sequences, although bi-cubic parametric patches and implicit functions have been tried. It seems that the needs of accuracy and ease of animating shape change are opposites. Methods that allow a high degree of accuracy are difficult to animate, because they consist of a structure with maybe thousands of low-level primitives as leaves. For example, the common way to control a net of bi-cubic parametric patches representing, say, the face of a character is to organize it into a hierarchy allowing local changes to be made (by descending the hierarchy and operating on a few or even a single patch) and making more global changes by operating at a high level in the structure. This has not resulted in a generally accepted animation technique simply because it does not produce good results (in the case of facial animation anyway). It seems shape-change animation needs a paradigm that is independent of the object model and the most successful techniques involve embedding the object model in *another* structure which is then subject to shape-change animation. Thus we control facial animation by attaching a geometric structure to a muscle control model or immerse a geometric model in the 'field' of an elastic solid and animate the elastic solid. In other words for the representations that we currently use, animation of shape does not seem to be possible by operating directly on the geometry of the object.

3

Representation and modelling of three-dimensional objects (2)

- 3.1 Bézier curves
- 3.2 B-spline representation
- 3.3 Rational curves
- 3.4 From curves to surfaces
- 3.5 B-spline surface patches
- 3.6 Modelling or creating patch surfaces
- 3.7 From patches to objects

Introduction

In the previous chapter we concentrated mainly on the polygon mesh representation where a polygon was, for example, a (flat) quadrilateral made up of four vertices joined by four straight lines. This chapter is devoted entirely to a representational form where the primitive element – a bi-cubic parametric patch – is a curvilinear quadrilateral. It has four corner points joined by four edges which are themselves cubic curves. The interior of the patch is a curved (cubic) surface where every point on the surface is defined. This contrasts with the polygon mesh approximation where surface points on an object are only defined at the polygon vertices.

Representing surfaces of objects using bi-cubic parametric patches finds two main applications in computer graphics:

- (1) As a basis for interactive design in CAD. Here we may obtain the model by an interactive process – a designer building up a model by interacting with a program. In many CAD applications the representational form is transformed *directly* into a real object (or a scaled-down model of the real object). The

computer graphics representation is used to control a device such as a numerical milling machine which sculpts the object in some material. This is exactly the opposite of the 'normal' computer graphics modelling methodology – instead of transforming a real object into a representation we are using the computer graphics model to make the real object.

- (2) As an alternative representational form to the polygon mesh – the representation which services the normal computer graphics requirement of transforming a real object into a representational form. In this use we usually wish to exploit the accuracy of the parametric representation over the polygon mesh approximation. Here we may obtain a parametric representation from a real object by some (surface) interpolation technique.

The apparent advantages of this representation over the polygon mesh representation are:

- It is an exact analytical representation.
- It has the potential of three-dimensional shape editing.
- It is a more economical representation.

Given these advantages it is somewhat surprising that this form is not the mainstream representation in computer graphics. It is certainly no more difficult to render an object represented by a net of patches and so we must conclude that its lack of popularity in mainstream computer graphics (it is, of course, used in industrial CAD), is due to the mathematical formalities associated with it.

The exactness of the representation factor needs careful qualification. A real object (or a physical model of a real object) can be represented by a net or mesh of patches (Figure 3.28 and Figure 3.43 are two such objects) but the representation may not be wholly 'exact'. In the first example, the teapot cannot have a perfectly circular cross-section because the representational method, in this case the Bézier form or Bernstein basis, cannot represent a circle exactly. The patches representing the face in the second example may not everywhere be coincident with the real object. We can obtain a suitable set of points that lie in the surface of the object from a three-dimensional digitizer and we could, say, use the same set of points that we would use to build a polygon mesh model. We then use an interpolation technique known as surface fitting, to determine a set of patches that represents the surface. However, the patch surface and the object surface will not necessarily be identical. The exactness of the fit depends on both the nature of the interpolation process and how closely the physical surface conforms to the shape constraints of the bi-cubic patch representation. But we do end up with an object representation that is a smooth surface which has certain advantages over the polygon mesh representation – the silhouette edge problem, which accounts for the most prominent visual defect in rendered polygon mesh objects, is cured.

It is possible to model subtly shaped objects such as the human face with a net of patches. An adequate representation of such an object using a polygon mesh would need an extremely high polygonal resolution. Despite this there is a perceived complexity associated with bi-cubic parametric patches and in many applications we can

avoid this by using the polygon mesh representation. When we digitize real objects we are normally working with an application that does not demand exact representation. We may be building a model of a product for an animated TV commercial, for example, in which case a good polygon mesh model will do.

In fact the most common applications of the bi-cubic parametric patch representation are not to build very complex models but as a representation for fairly simple objects in industrial CAD or CAGD applications. The real value of the representation here is that it can be used to transform an abstract design, built up within an interactive program, directly into a physical reality. The description can be made to drive a sculpting device such as a numerically controlled milling machine to produce a prototype object without any human intervention. It is this single factor more than any other that makes bi-cubic parametric patches important in CAD.

Part of their value in CAD comes from the ability to change the shape of an object represented by patches in a way that maintains a smooth surface. Sometimes the allusion to sculpting is made. We can view the representation as a kind of 'abstract clay' model that can be pulled around and deformed into any desirable shape – giving the same freedom to create as a sculptor would have with a real clay model. Here we should be wary of the claims that are made in the computer graphics literature concerning the efficacy of free-form sculpting using bi-cubic parametric patches. We can distinguish between methods that attempt a free-form sculpting model, which places no constraints on the shape complexity of the object formed, and the much more well-established techniques in CAD where the object tends to be fairly simple. A common, early example of this category is the design of car body panels. Bi-cubic parametric patches are manifestly successful in such applications; their success as a metaphor for clay sculpting is more debatable.

We distinguish between objects that are represented by a single patch and objects whose form demands that they are represented by a net of patches. Shape editing a single patch is straightforward but the objects that we can design with a single patch are restricted. Shape editing an object that is represented by a net of patches is much more difficult. One problem is that if we have to alter the shape of one patch in a net, we have to maintain its smoothness relationship with the neighbouring patches in which it is embedded. Another difficulty is yet another manifestation of the scale problem. Say we want to effect a shape change that involves many patches. We have to move these patches together and maintain their continuity with all their neighbouring patches.

Despite these difficulties we should recognize that this representation has a strong potential for shape editing compared with the polygon mesh representation. This is already an approximation and pulling vertices around to change the shape of the represented object results in many difficulties. The accuracy of the polygon mesh representation changes as soon as vertices are moved resulting perhaps in visual defects. It is almost certain that we would always have to move groups of points rather than move a single polygon vertex around in three space. Pulling a single vertex would just result in a local peak.

In this chapter we will mainly confine ourselves to the study of single patches and simple shapes formed from nets of a few patches using rudimentary but powerful CAD techniques, such as generating a solid object by sweeping a profile through 360°.

The analytical representation of patches differs according to the formulation and some have been named after their instigators. One of the most popular formulations is the Bézier patch developed in the 1960s by Pierre Bézier for use in the design of Renault cars. His CAD system called UNISURF was one of the first to be used. In what follows we will concentrate mainly on the Bézier and B-spline formulation.

The usual approach in considering parametric representation is to begin with a description of three-dimensional space curves and then to generalize to surfaces or patches. A three-dimensional space curve is a smooth curve that is a function of the three spatial variables. An example would be the path that a particle traced as it moved through space. Incidentally, curves by themselves also find applications in computer graphics. For example, we can script the path of an object in three-dimensional computer animation by using a space curve. We can model a 'ducted' solid by sweeping a cross-section along a space curve as we saw in the previous chapter.

3.1

Bézier curves

In this section we will look at the pioneering developments of Bézier, who was amongst the first to develop computer tools in industrial design. We will draw on Bézier's own descriptions of the evolution of his method, not just because of their historical interest but also because they give a real insight into the relationship between the representation, the physical reality and the requirements of the designers who were to use his methods.

Bézier's development work was carried out in the Renault car factory in the 1960s and he called his system UNISURF. Car designers are concerned with styling free-form surfaces which are then used to produce master dies which produce the tools that stamp out the manufactured parts. Many other industries use free-form surfaces. Some parts such as ship's hulls, airframes and turbine blades are constrained by aerodynamic and hydrodynamic considerations and shapes evolve through experience and testing in wind tunnels and test tanks, but a designer still needs freedom to produce new shapes albeit within these constraints. Before the advent of this representational form, such free-form surfaces could not be represented analytically and once developed could only be stored for future reproduction and evolution by sampling and storing as coordinates.

Prior to Bézier's innovation the process of going from the abstract design to the prototype was lengthy and involved many people and processes. The following description, abstracted from Bézier's account in Piegl's book (Piegl 1993), is of the process of car design at the time:

- (1) Stylists defined a general shape using small-scale sketches and clay mock-ups.
- (2) Using offsets (world coordinates in computer graphics terminology) measured on the mock-up, designers traced a full-scale shape of the skin of the car body.
- (3) Plasterers built a full-scale model, weighing about eight tons, starting from plywood cross-sections duplicating the curves of the drawing. The clay model was then examined by stylists and sales managers, and modified according to taste.
- (4) When at last the model was accepted, offsets were again measured and the final drawings were made. During this period, which could be a year or more, tooling and production specialists often suggested minor changes to avoid difficult and costly operations during production.
- (5) The drawings were finalized, and one three-dimensional master was built as the standard for checking the press tools and stamped parts.
- (6) The plaster copies of the master were used for milling punches and dies on copy-machine tools.

Bézier's pioneer development completely changed most aspects of these processes by enabling a representation of free-form surfaces. Before, a designer would produce curves using say a device such as a French curve. The designer used his skill and experience to produce a complete curve that was built, step-by-step, using segments along some portion of the French curve. A curve generated in this way could not be stored conveniently except as a set of samples. Bézier's development was a definition that enabled such curves to be represented as four points, known as control points, and an implicit set of basis or blending functions. When the four points are injected into the definition, the curve is generated or reproduced. This has two immediate consequences. The definition can be used directly to drive a numerically controlled milling machine and the part can be produced exactly without the intervention of complications and delays. (Numerically controlled milling machines have been in existence since 1955 and were another motivation for the development of CAGD.) The definition can be used as a basis of a CAD program in which modifications to the curve can be made to a computer visualization.

Bézier describes an intriguing difficulty that he experienced at the time:

When it was suggested that these curves replace sweeps and French curves, most stylists objected that they had invented their own templates and would not change their methods. It was therefore solemnly promised that their secret curves would be translated into secret listings and buried in the most secret part of the computer's memory, and that nobody but them would keep the key of the vaulted cellar. In fact, the standard curves were flexible enough and secret curves were soon forgotten; designers and draughtsmen easily understood the polygons and their relationship with the shape of the corresponding curves.

Many simultaneous developments were occurring in other industries – notably aircraft and ship manufacture, and much of the research was carried out under

the auspices of particular manufacturers, who, like Bézier at Renault, developed their own CAD systems and surface representations suited to their own requirements. This has led to a number of parametric definitions of surfaces and the interested reader is best referred to Piegl's book in which each chapter is written by a pioneer in this field.

Bézier states that one of the most important requirements of his representation was that it should be founded on geometry and that the underlying mathematics should be easily understood. He introduced the concept of a space curve being contained in a cube which when distorted into a parallelepiped distorts the curve (Figure 3.1). The curve is 'fixed' within the parallelepiped as follows:

- The start and end points of the curve are located at opposite vertices of the parallelepiped.
- At its start point the curve is tangential to Ox .
- At its end point the curve is tangential to Oz .

This geometric concept uniquely defines any space curve (if it is understood that the curve is a polynomial of a certain degree) and also gives an intuitive feel for how the curve changes shape as the parallelepiped changes. Now the parallelepiped, and thus the curve, can be completely defined by four points – known as control points – P_0, P_1, P_2 and P_3 which are just vertices of the parallelepiped as shown in the figure. Given that the position of the end points of the curve is fixed and its behaviour at the end points is determined, the shape that the curve traces out in space between its extremities needs to be defined. A parametric

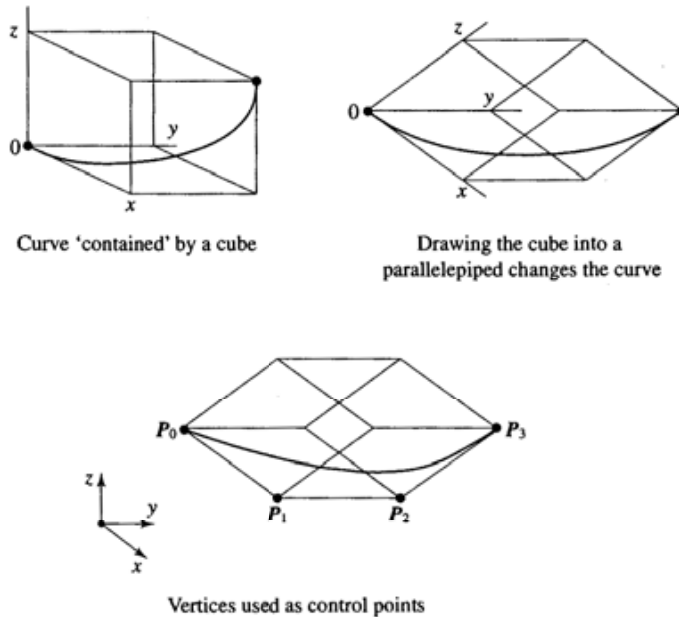


Figure 3.1
Bézier's concept of curve representation.

definition was chosen which means that the space curve $Q(u)$ is defined in terms of a parameter u ($0 \leq u \leq 1$). As u varies from 0 to 1 we arrive at values for the position of a point on $Q(u)$ by scaling or blending the control points. That is, each point on the curve is determined by scaling each control point by a cubic polynomial known as a basis or blending function. The curve is then given by:

$$Q(u) = \sum_{i=0}^3 P_i B_i(u) \quad [3.1]$$

and in the case of a Bézier curve the basis or blending functions are the Bernstein cubic polynomials:

$$\begin{aligned} B_0(u) &= (1-u)^3 \\ B_1(u) &= 3u(1-u)^2 \\ B_2(u) &= 3u^2(1-u) \\ B_3(u) &= u^3 \end{aligned}$$

Figure 3.2 shows these polynomials and a Bézier curve (projected into the two-dimensional space of the diagram).

A useful intuitive notion is the following. As we move physically along the curve from $u = 0$ to $u = 1$ we simultaneously move a vertical line in the basis function space that defines four values for the basis functions. Weighting each basis function by the control points and summing, we obtain the corresponding point in the space of the curve. We note that for any value of u (except $u = 0$ and $u = 1$) all the functions are non-zero. This means that the position of all the control points contribute to every point on the curve (except at the end points). At $u = 0$ only B_0 is non-zero. Therefore:

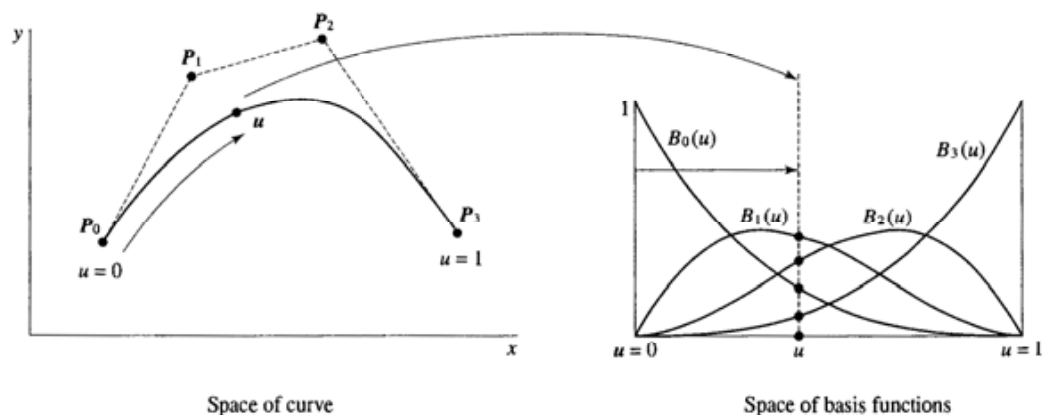
$$Q(0) = P_0$$

similarly

$$Q(1) = P_3$$

Figure 3.2

Moving along the curve by increasing u is equivalent to moving a vertical line through the basis functions. The intercepts of this line with the basis functions give the values of B for the equivalent point.



We also note that:

$$B_0(u) + B_1(u) + B_2(u) + B_3(u) = 1$$

Joining the four control points together gives the so-called control polygon and moving the control points around produces new curves. Moving a single control point of the curve distorts its shape in an intuitive manner. This is demonstrated in Figure 3.3. The effect of moving the end points is obvious. When we move the inner control points P_1 and P_2 we change the orientation of the tangent vectors to the curves at the end points – again obvious. Less obvious is that the positions of P_1 and P_2 also control the magnitude of the tangent vectors and it can be shown that:

$$Q_u(0) = 3(P_1 - P_0)$$

$$Q_u(1) = 3(P_2 - P_3)$$

where Q_u is the tangent vector to the curve (first derivative) at the end point. It can be seen that the curve is pulled towards the tangent vector with greater magnitude which is controlled by the position of the control points.

Bézier curves find uses not just in highly technical applications but also in popular software. Drawing packages that are found nowadays in word processors and DTP applications almost always include a sketching facility based on Bézier curves. Another well-known application of Bézier curves is shown in Figure 3.4. Here a typeface is in the process of being designed. The outline of the filled character is a set of Bézier curves to which the designer can make subtle alterations by moving the control points that specify curves that describe the outline.

Bézier's original cube concept, encapsulating a curve of three spatial variables, seems to have been lost and most texts simply deal with the curves of two spatial variables enclosed in a control polygon. Applications where three-dimensional space curves have to be designed, three-dimensional computer animation for example, can have interfaces where two-dimensional projections

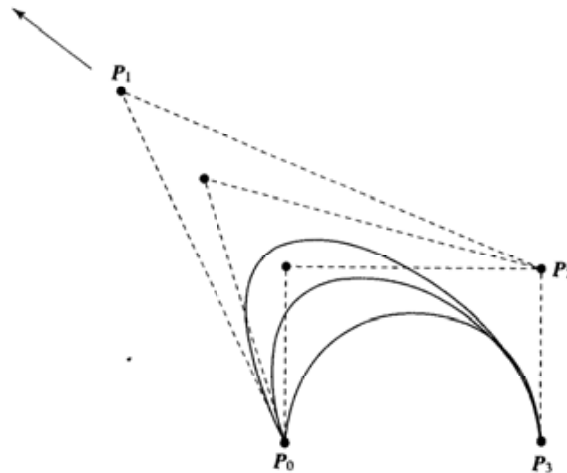
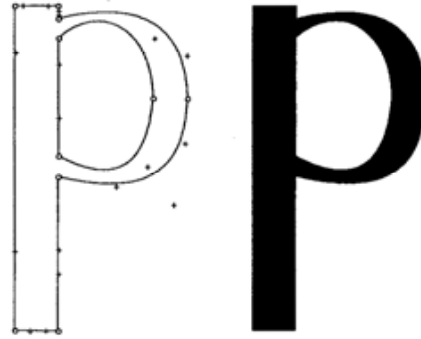


Figure 3.3
Effects of moving control
point P_1 .

Figure 3.4
Using Bézier curves in font design. Each curve segment control points are symbolized by O + + O.



of the curve are used. An example of this application is given in Section 17.2.2. (Note, that for a three-dimensional curve the parallelepiped determines the plane in which the tangents to the curve – the edges of the control polygon – are oriented.)

At this point it is useful to consider all the ramifications of representing a curve with control points. The most important property, as far as interaction is concerned, is that moving the control points gives an intuitive change in curve shape. Another way of putting it is to say that the curve mimics the shape of the control polygon. An important property from the point of view of the algorithms that deal with curves (and surfaces) is that a curve is always enclosed in the convex hull formed by the control polygon. The convex hull of a two-dimensional space curve is illustrated in Figure 3.5 and can be considered to be the polygon formed by placing an elastic band around the control points. This follows from the fact that the basis functions sum to unity for all u .

Now consider transforming curves. Since the curves are defined as linear combinations of the control points, the curve is transformed by any affine transformation (rotation, scaling, translation etc.) in three-dimensional space by

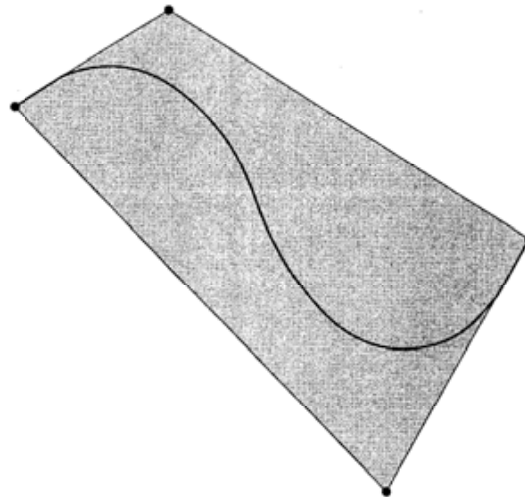


Figure 3.5
Convex hull property for cubic spline. The curve is contained in the shaded area formed from the control points.

applying the appropriate transformations to the set of control points. Thus, to transform a curve we transform the control points then compute the points on the curve. In this context, note that it is not easy to transform a curve by computing the points then transforming (as we might do with an implicit description). For example, it is not clear in scaling, how many points need to ensure smoothness when the curve has been magnified. Note here that perspective transformations are non-affine, so we cannot map control points to screen space and compute the curve there. However, we can overcome this significant disadvantage by using rational curves as we describe later in this chapter.

Finally, a useful alternative notation to the summation form is the following. First, we expand Equation 3.1 to give:

$$Q(u) = P_0 (1 - u)^3 + P_1 3u(1 - u)^2 + P_2 3u^2 (1 - u) + P_3 u^3$$

this can then be written in matrix notation as:

$$Q(u) = \mathbf{UB}_2\mathbf{P}$$

$$= [u^3 \ u^2 \ u \ 1] \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

3.1.1

Joining Bézier curve segments

Curve segments, defined by a set of four control points, can be joined to make up 'more complex' curves than those obtainable from a single segment. This results in a so-called piecewise polynomial curve. An alternative method of representing more complex curves is to increase the degree of the polynomial, but this has computational and mathematical disadvantages and it is generally considered easier to split the curve into cubic segments. Connecting curve segments implies that constraints must apply at the joins. The default constraint is positional continuity, the next best is first order (or tangential continuity). The difference between positional and first order continuity for a Bézier curve is shown in Figure 3.6. Positional continuity means that the end point of the first segment is coincident with the start point of the second. First order continuity means that the edges of the characteristic polygon are collinear as shown in the figure. This means that the tangent vectors, at the end of one curve and the start of the other, match to within a constant. In shaded surfaces, maintaining only positional continuity would possibly result in the joins being visible in the final rendered object.

If the control points of the two segments are \mathbf{S}_i and \mathbf{R}_i then first order continuity is maintained if:

$$(\mathbf{S}_3 - \mathbf{S}_2) = k(\mathbf{R}_1 - \mathbf{R}_0)$$

Using this condition a composite Bézier curve is easily built up by adding a single segment at a time. However, the advantage of being able to build up

Figure 3.6
Continuity between Bézier curve segments.
(a) Positional continuity between Bézier points.
(b) Tangential continuity between Bézier points.

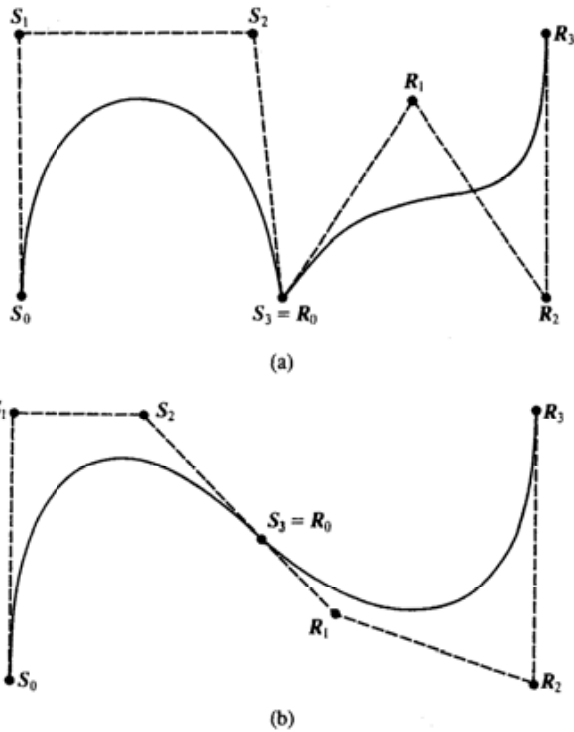
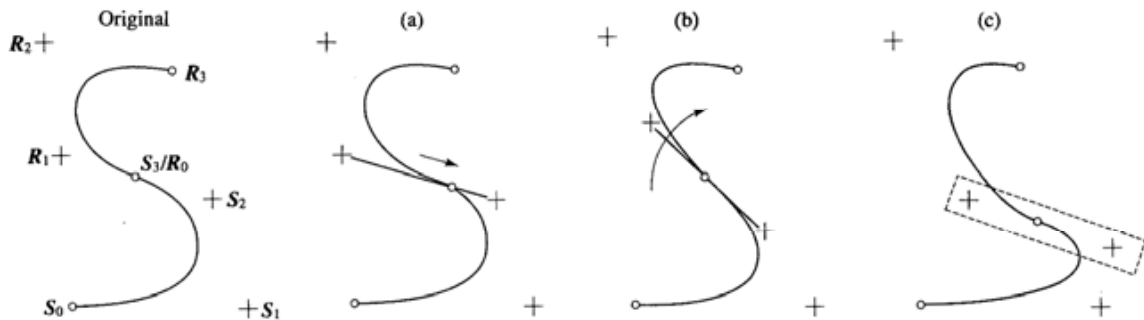


Figure 3.7
Examples of possible shape editing protocols for a two-segment Bézier curve.
(a) Maintain the orientation of R_1S_2 and move any of the three control points R_1 , S_1/R_0 , S_2 along this line.
(b) Rotate the line R_1S_2 about S_1/R_0 .
(c) Move the three control points R_1 , S_1/R_0 , S_2 as a 'locked' unit.

a composite form from segments is somewhat negated by the constraints on local control that now apply because of the joining conditions.

Figure 3.4 is an example of a multi-segment Bézier curve. In this case a number of curves are joined to represent the outline of the character and first order continuity is maintained between them. It is useful to consider the ramifications for an interface through which a user can edit multi-segment curves and maintain continuity. Figure 3.7 shows some possibilities. The illustration assumes that the user has already constructed a two-segment curve whose shape is to be



altered around the area of the join point S_3/R_0 . To maintain continuity we must operate simultaneously on R_1 , R_0/S_3 and S_2 . We can do this by:

- Maintaining the orientation of the line R_1, S_2 and moving the join point up and down this line (Figure 3.7(a)).
- Maintaining the position of the join point and rotating the line R_1, S_2 about this point (Figure 3.7(b)).
- Moving all three control points as a locked unit (Figure 3.7(c)).

These three editing possibilities or constraints will enable the user to change the shape of curves made up of any number of segments while at the same time maintaining first order continuity between the curve segments. We will see later that this complication of Bézier curves can be overcome in another way – by using B-spline curves.

3.1.2

Summary of Bézier curve properties

- A Bézier curve is a polynomial. The degree of the polynomial is always one less than the number of control points. In computer graphics we generally use degree 3. Quadratic curves are not flexible enough and going above degree 3 gives rise to complications and so the choice of cubics is the best compromise for most computer graphics applications.
- The curve 'follows' the shape of the control point polygon and is constrained within the convex hull formed by the control points.
- The control points do not exert 'local' control. Moving any control point affects all of the curve to a greater or lesser extent. This can be seen by examining Figure 3.2 which shows that all the basis functions are everywhere non-zero except at the point $u = 0$ and $u = 1$.
- The first and last control points are the end points of the curve segment.
- The tangent vectors to the curve at the end points are coincident with the first and last edges of the control point polygon.
- Moving the control points alters the magnitude and direction of the tangent vectors – the basis of the intuitive 'feel' of a Bézier curve interface.
- The curve does not oscillate about any straight line more often than the control point polygon – this is known as the variation diminishing property. This has implications concerning the nature of the surface that can be represented.
- The curve is transformed by applying any affine transformation (that is, any combination of linear transformations) to its control point representation. The curve is invariant (does not change shape) under such a transformation.

3.2**B-spline representation**

The simplicity and power of the Bézier representation is no doubt responsible for its enduring popularity. It does, however, suffer from limitations and we will address these in this section by looking at how these are overcome by using the B-spline representation. We will as before introduce B-splines by first examining B-spline curves.

Historically, B-splines preceded Bézier curves and their origin relates to industries such as shipbuilding where a designer was required to draw life-size curves representing such entities as the cross-section through the hull of a ship. For small-scale drawing, draughtsmen would use French curves – a set of small, flat pre-formed curve sections. They would draw complete curves by putting together segments formed from different parts of different French curves. For full-scale plans this method was completely impractical and the draughtsmen (called in ship-building loftsmen) would employ long, thin strips of metal. These were pushed into the required curve shape and secured using lead weights called ducks, and the analogue between ducks and control points should be clear. We can push the spline into any desired shape that the system can take up and we can have as many ducks as we require. This is the physical basis of B-splines and we can compare the idea with either a single segment Bézier curve or a multi-segment Bézier curve. If we compare it with a single segment curve we see that adding extra control points or ducks removes the variation diminishing property – the curve can oscillate as we require. Comparing it with a multi-segment Bézier curve we can say that it is equivalent but we do not have to explicitly maintain continuity anywhere. Imagine a loftsmen inserting an extra duck – the physical properties of the metal spline ensures that the new shape that is taken up around the point where the duck was inserted is continuous. The metal takes up a shape that minimizes its internal strain energy. Yet another point that comes out of this real piece of engineering is that the effect of a duck insertion is local. The shape of the curve is only altered in its vicinity. We now deal with these points in a formal manner.

3.2.1**B-spline curves**

Two drawbacks associated with Bézier curves that are overcome by using B-spline curves are their non-localness and the relationship between the degree of the curve and the number of control points. The first property – non-localness – implies that although a control point heavily influences that part of the curve most close to it, it also has some effect on all the curve and this can be seen by examining Figure 3.2. All the basis functions are non-zero over the entire range of u . The second disadvantage means that we cannot use a Bézier cubic curve to approximate or represent n points without the inconvenience of using multiple curve segments (or by increasing the degree of the curve).

Like a Bézier curve a B-spline curve does not pass through its control points. A B-spline is a complete piecewise cubic polynomial consisting of any number of curve segments. (For notational simplicity we will only consider cubic B-splines. We can, however, have B-splines to any degree.) It is a cubic segment over a certain interval, and going from one interval to the next, the coefficients change. For a single segment only, we can compare the B-spline formulation with the Bézier formulation by using the same matrix notation.

The B-spline formulation is:

$$\begin{aligned}
 Q_i(u) &= \mathbf{UB}_i\mathbf{P} \\
 &= [u^3 \ u^2 \ u \ 1] \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{P}_{i-3} \\ \mathbf{P}_{i-2} \\ \mathbf{P}_{i-1} \\ \mathbf{P}_i \end{bmatrix}
 \end{aligned}$$

where Q_i is the i th B-spline segment and \mathbf{P}_j is a set of four points in a sequence of control points. Alternatively we can write:

$$Q_i(u) = \sum_{k=0}^3 \mathbf{P}_{i-3+k} B_{i-3+k}(u) \tag{3.2}$$

where i is the segment number and k is the local control point index – that is the index for the segment i . The value of u over a single curve segment is $0 \leq u \leq 1$. Using this notation we can describe u as a local parameter – locally varying over the parametric range of 0 to 1 – to define a single B-spline curve segment.

Thus in this notation we see that a B-spline curve is a series of $m - 2$ curve segments that we conventionally label Q_3, Q_4, \dots, Q_m defined or determined by $m+1$ control points $\mathbf{P}_0, \mathbf{P}_1, \dots, \mathbf{P}_m, m \geq 3$. Each curve segment is defined by four control points and each control point influences four and only four curve segments. This is the local control property of the B-spline curve and its main advantage over the Bézier curve.

Here we must be careful. Barsky (in Bartels *et al.* 1988) points out that comparing Bézier curves and B-spline curves can be misleading because it is not a comparison of like with like but a comparison of a single segment Bézier curve (which may have the control vertex set extended and the degree of the curve raised) with a piecewise or composite B-spline curve. A single segment Bézier curve is subject to global control because moving a control point affects the complete curve. In a composite B-spline curve moving a control point only affects a few segments of the curve. The comparison should be between multi-segment Bézier curves and B-splines. The difference here is that to maintain continuity between Bézier segments the movement of the control points must satisfy constraints, while the control points of a B-spline composite can be moved in any way.

A B-spline exhibits positional, first derivative and second derivative (C^2) continuity and this is achieved because the basis functions are themselves C^2 piecewise polynomials. A linear combination of such basis functions will also be C^2 continuous. We define the entire set of curve segments as one B-spline curve in u :

$$Q(u) = \sum_{i=0}^m P_i B_i(u)$$

In this notation i is now a non-local control point number and u is a global parameter discussed in more detail in the next section.

3.2.2

Uniform B-splines

Equation 3.2 shows that each segment in a B-spline curve is defined by four basis functions and four control vertices. Hence there are three more basis functions and three more control vertices than there are curve segments. The join point on the value of u between segments is called the knot value and a uniform B-spline means that knots are spaced at equal intervals of the parameter u . Figure 3.8 shows a B-spline curve that is defined by (the position of) six control vertices or control points P_0, P_1, \dots, P_5 . It also shows the effect of varying the degree of the polynomials, and curves are shown for degree 2, 3 and 4. We are generally interested in cubics and this is a curve of three segments with the left-hand end point of Q_3 near P_0 and the right-hand end point of Q_5 near P_5 . (Thus we see that a uniform B-spline does not in general interpolate the end control points, unlike a Bézier curve. Also it is the case that a Bézier curve more closely approximates its control point polygon. However, the continuity-maintaining property of the B-spline curve outweighs these disadvantages.)

The notation gives us the following organization (where each curve segment is shown as an alternating full/dotted line):

Q_3 is defined by $P_0 P_1 P_2 P_3$ which are scaled by $B_0 B_1 B_2 B_3$

Q_4 is defined by $P_1 P_2 P_3 P_4$ which are scaled by $B_1 B_2 B_3 B_4$

Q_5 is defined by $P_2 P_3 P_4 P_5$ which are scaled by $B_2 B_3 B_4 B_5$

The fact that each curve segment shares control points is the underlying mechanism whereby C^2 continuity is maintained between curve segments. Figure 3.9

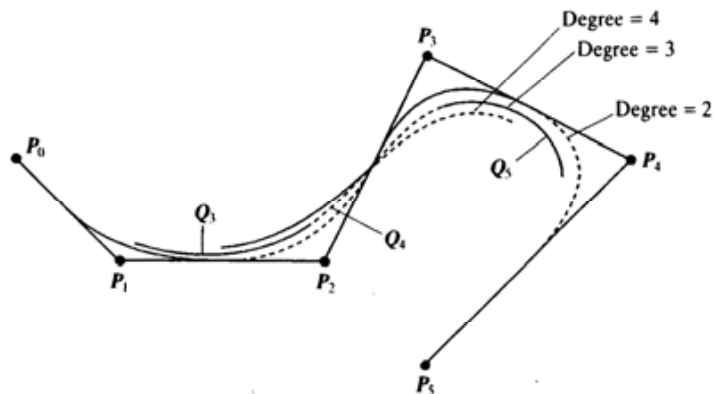
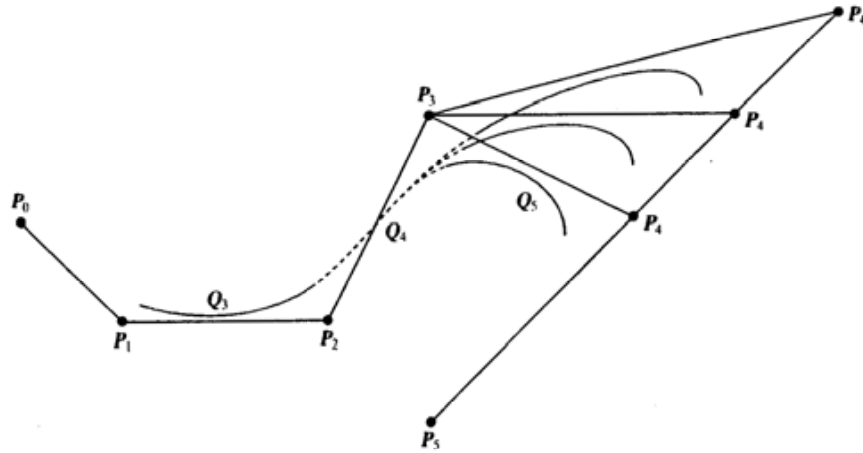


Figure 3.8
A three-segment B-spline cubic curve defined by six control points.

Figure 3.9
 Demonstrating the locality property of B-spline curves. Moving P_1 changes Q_3 and Q_4 to a lesser extent. Q_5 is unchanged.



shows the effect of changing the position of control point P_4 . This pulls the segment Q_5 in the appropriate direction and also affects, to a lesser extent, segment Q_4 (which is also defined by P_4). However, it does not affect Q_3 and this figure demonstrates the important locality property of B-splines. In general, of course, a single control point influences four curve segments.

We now consider the underlying basis functions that define the curve. Each basis function is non-zero over four successive intervals in u (Figure 3.10). It is, in fact, a cubic composed itself of four segments. The B-spline is non-zero over the intervals $u_i, u_{i+1}, \dots, u_{i+4}$ and centred on u_{i+2} . Now each control point is scaled by a single basis function and if we assume that our knot values are equally spaced, then each basis function is a copy or translate and the set of basis functions used by the curve in Figure 3.8 is shown in Figure 3.11.

The basis functions sum to unity in the range $u = 3$ to $u = 6$ in this case, the values of the parameter u over which the curve is defined. A consequence of this is that the entire B-spline curve is contained within the convex hull of its control points. If we consider a single segment in the curve, then this defines a parameter range u_i to u_{i+1} . The basis functions that are active in the i th parametric interval, u_i to u_{i+1} , that is the functions that define a single curve segment, are shown highlighted in Figure 3.12. This gives a useful interpretation of the behaviour of the functions as u is varied. In general, for values of u that are not knot values, four basis functions are active and sum to unity. When a knot value $u = u_i$ is

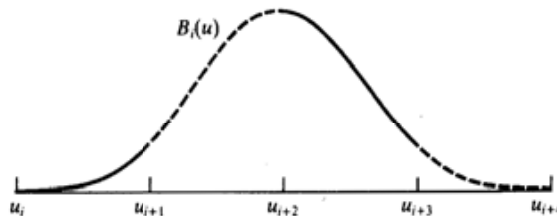
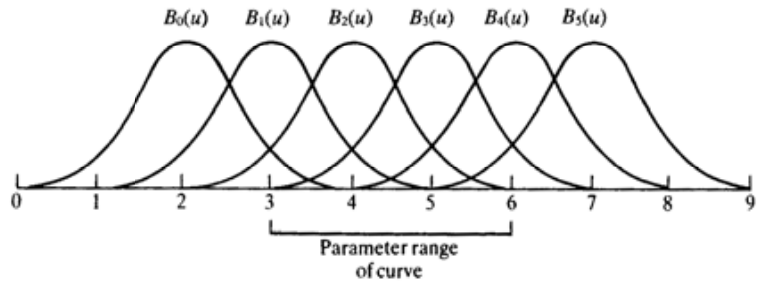


Figure 3.10
 The uniform cubic B-spline $B(u)$.

Figure 3.11
The six B-splines used in constructing the curve of Figure 3.8. They are all translates of each other.



reached one basis function 'switches off' and another 'switches on'. At the knot value there are three basis functions that sum to unity.

At this stage we can summarize and state that a B-spline curve is made up of $m - 2$ segments defined by the position of $m + 1$ basis functions over $m + 5$ knot values. Thus in Figure 3.7 we have three segments, six control points and six basis functions over ten knot values.

Now consider again Figure 3.12. In the parameter range $u_i \leq u \leq u_{i+1}$ we evaluate the four B-splines B_i , B_{i-1} , B_{i-2} and B_{i-3} by substituting $0 \leq u \leq 1$ and computing:

$$\begin{aligned} B_i &= \frac{1}{6} u^3 \\ B_{i-1} &= \frac{1}{6} (-3u^3 + 3u^2 + 3u + 1) \\ B_{i-2} &= \frac{1}{6} (3u^3 - 6u^2 + 4) \\ B_{i-3} &= \frac{1}{6} (1 - u)^3 \end{aligned} \quad [3.3]$$

It is important to note that this definition gives a single segment from each of the four B-spline basis functions over the range $0 \leq u \leq 1$. It does *not* define a single B-spline basis function which consists of four segments over the range $0 \leq u \leq 4$.

We now come to consider the end control vertices and note again that the curve does not interpolate these points. In general, of course, a B-spline curve does not interpolate any control points. We can make a B-spline curve interpolate control points by introducing multiple vertices. However, this involves a loss of continuity as we shall see. Intuitively we can think of increasing the influence of a control point by repeating it. The curve is attracted to the repeated point. A segment is made by basis functions scaling control points. If a control point is

The Figure 3.12
The four B-splines that are non-zero or active for the first curve segment in Figure 3.8.

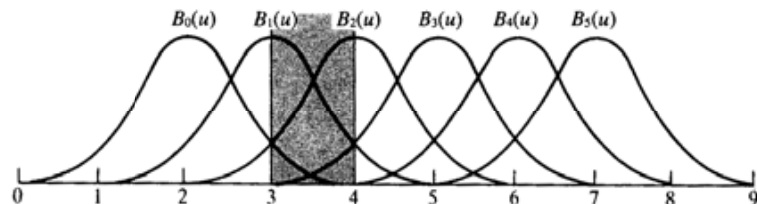
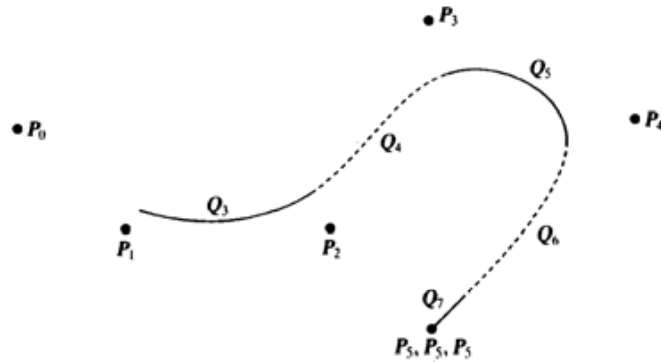


Figure 3.13
 Demonstrating the effect of multiple end control points. P_3 is repeated three times forcing the curve to interpolate it.



repeated it will be used more than once in the evaluation of a single segment. For example, consider Figure 3.13 and compare it with Figure 3.8. The last control point in the example in Figure 3.8 is now repeated three times. There are now five segments and P_3 is used once in the determination of Q_5 , twice in Q_6 and three times in Q_7 . The curve now ranges over $3 \leq u \leq 8$. At $u = 8$ the curve is coincident with P_3 .

Such a technique can be used to make the curve interpolate both the intermediate control points and the end points. Figure 3.14(a) shows the effect

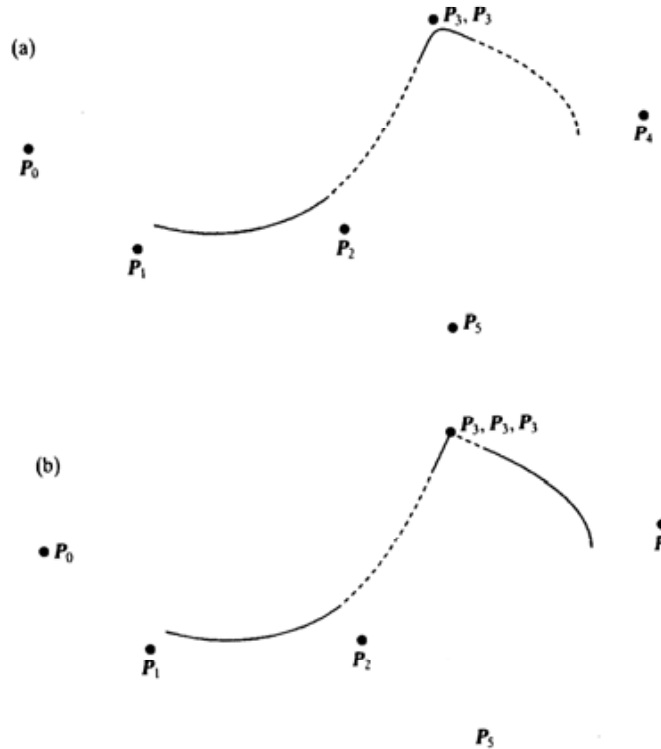


Figure 3.14
 Demonstrating the effect of multiple intermediate control points. (a) P_3 is duplicated. (b) P_3 is triplicated.

of introducing multiple intermediate control points. In this figure P_3 has been doubled. P_3 is almost interpolated and an extra segment is introduced. The continuity changes from C^2G^2 to C^2G^1 . This means that the continuity across the two segments is reduced by one although the continuity within each segment is still C^2 . Figure 3.14(b) shows P_3 made into a triple control point. This time the curve interpolates the control point and the curve becomes a straight line on either side of the control point. The continuity reduces now to C^2G^0 .

3.2.3

Non-uniform B-splines

In the previous section we considered a family of curves that we referred to as uniform B-splines because the basis functions were translates of each other. We now look at non-uniform B-splines.

A non-uniform B-spline is a curve where the parametric intervals between successive knot values are not necessarily equal. This implies that the blending functions are no longer translates of each other but vary from interval to interval. The most common form of a non-uniform B-spline is where some of the intervals between successive knot values are reduced to zero by inserting multiple knots. This facility is used to interpolate control points (both end points and intermediate points) and it possesses certain advantages over the method used in the previous section – inserting multiple control points. In particular a control point can be interpolated without the effect that occurred with multiple control vertices – namely straight line curve segments on either side of the control point.

Consider the curve generated in Figure 3.8. The knot values for this curve are $u = 3, 4, 5, 6$. We define a knot vector for this curve as $[0, 1, 2, 3, 4, 5, 6, 7]$ and a useful parametric range (within which the basis functions sum to unity) as $3 \leq u \leq 6$. The interval between each knot value is 1. If non-uniform knot values are

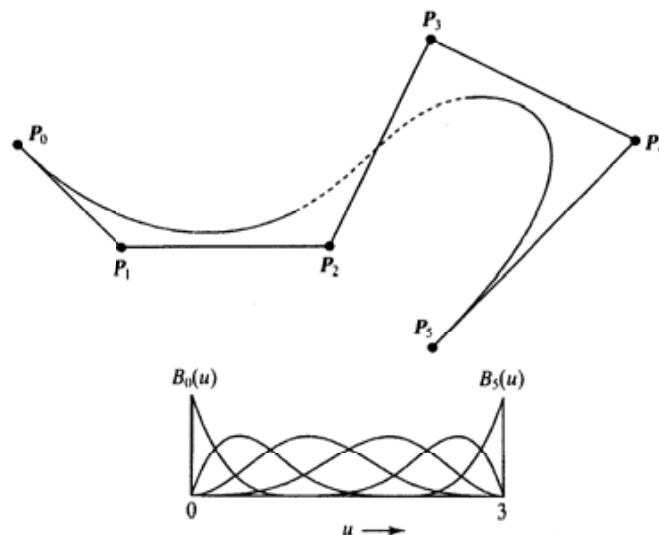


Figure 3.15
A non-uniform B-spline that interpolates the end points by using a knot vector $[0, 0, 0, 0, 1, 2, 3, 3, 3, 3]$.

used, then the basis functions are no longer the same for each parametric interval, but vary over the range of u . Consider Figure 3.15. This uses the same control points as Figure 3.8 and the B-spline curve is still made up of three segments. However, the curve now interpolates the end points because multiple knots have been inserted at each end of the knot vector. The knot vector used is $[0,0,0,0,1,2,3,3,3,3]$. The basis functions are also shown in the figure. The curve now possesses nine segments Q_0 to Q_8 . However, Q_0, Q_1, Q_2 are reduced to a single point. Q_3, Q_4 and Q_5 are defined over the range $0 \leq u \leq 3$. Q_6, Q_7 and Q_8 are reduced to a single point $u = 3$. In practice the knot sequence $[0,0,0,0,1,2,\dots,n-1,n,n,n,n]$ is often used. That is, interpolation is forced at the end points but uniform knots are used elsewhere. A second example showing the flexibility of a B-spline curve is given in Figure 3.16. Here we have nine control points and thirteen knots. The knot vector is $[0,0,0,0,1,2,3,4,5,6,6,6,6]$.

In general a knot vector is any non-decreasing sequence of knot values u_0 to u_{m+4} . As we have seen, successive knot values can be equal and the number of identical values is called the multiplicity of the knot. Causing a curve to interpolate the end points by using multiple control vertices does not have precisely the same effect as using multiple control vertices and Figure 3.17 shows the final control point P_5 in our standard example interpolated using both a multiple control point and a knot vector with multiplicity 4 on the final knot value.

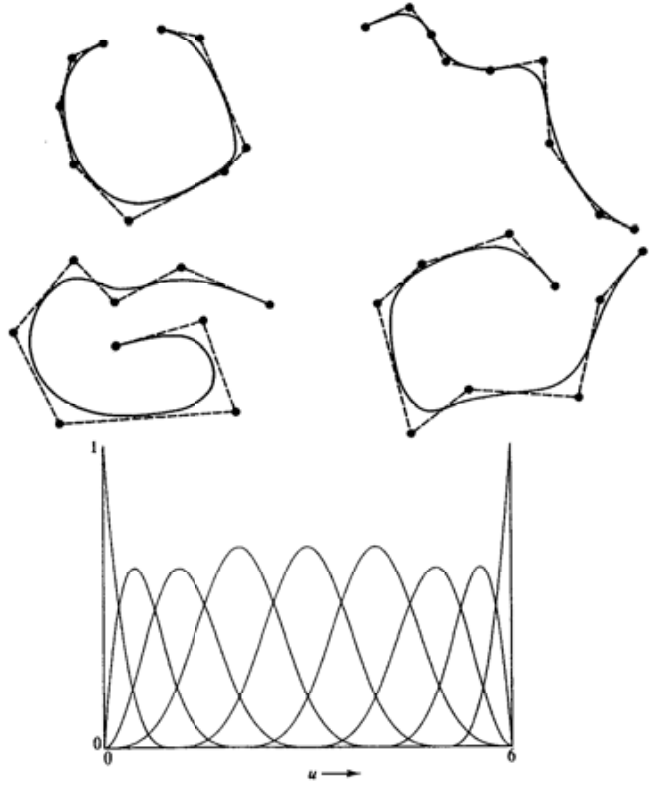
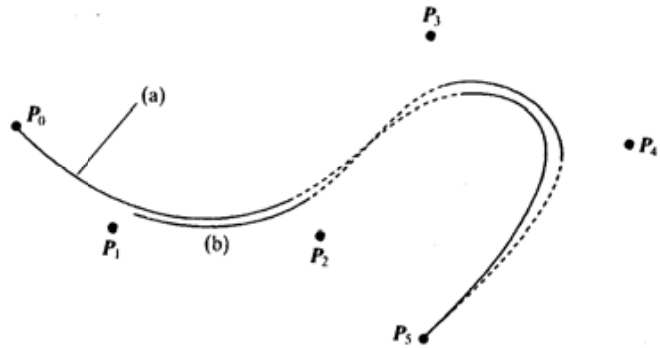


Figure 3.16
Showing the flexibility
of B-spline curves.
The knot vector is
 $[0,0,0,0,1,2,3,4,5,6,6,6,6]$.

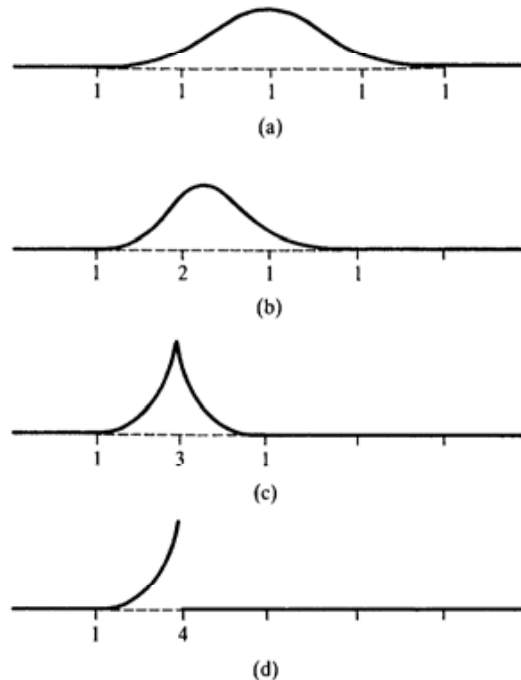
Figure 3.17

Comparing multiple knots with multiple control points.
 (a) The curve is generated by a knot vector with multiplicity 4 on the start and end values. (b) P_3 is repeated three times.



Note that if we use the knot vector $[0,0,0,0,1,1,1,1]$ then we have single segment curve interpolating P_0 and P_3 . In this instance the basis functions are the Bézier basis functions (Figure 3.2) and the resulting curve is a Bézier curve. Thus we see that a Bézier curve is just a special case of a non-uniform B-spline.

The effect of a multiple knot on the shape of a basis function is easily seen. Consider Figure 3.18(a) shows the uniform B-spline basis function defined over the knots $0, 1, 2, 3, 4$. As we have explained in the previous section, this is itself made up of four cubic polynomial segments defined over the given ranges. These are generated by using Equation 3.3 and translating each cubic segment by $0, 1, 2, 3$ and 4 units in u . Alternatively we can use:

**Figure 3.18**

The effect of knot multiplicity on a single cubic B-spline basis function.
 (a) All knot multiplicities are unity: $[0,1,2,3,4]$.
 (b) Second knot has multiplicity 2: $[0,1,1,2,3]$.
 (c) Second knot has multiplicity 3: $[0,1,1,1,2]$.
 (d) Second knot has multiplicity 4: $[0,1,1,1,1]$.

$$\begin{aligned}
 B_0(u) = & \begin{aligned}
 & b_{-0}(u) = \frac{1}{6} u^3 & 0 \leq u \leq 1 \\
 & b_{-1}(u) = -\frac{1}{6} (3u^3 - 12u^2 + 12u - 4) & 1 \leq u \leq 2 \\
 & b_{-2}(u) = \frac{1}{6} (3u^3 - 24u^2 + 60u - 44) & 2 \leq u \leq 3 \\
 & b_{-3}(u) = -\frac{1}{6} (u^3 - 12u^2 + 48u - 64) & 3 \leq u \leq 4
 \end{aligned}
 \end{aligned}$$

Compared with Equation 3.3 note that this defines a single B-spline basis function over the range $0 \leq u \leq 4$. If we double the second knot and use $[0,1,1,2,3]$, $b_{-1}(u)$ shrinks to zero length and the function becomes asymmetric as shown in Figure 3.18(b). The double knot eliminates second derivative continuity but first derivative continuity remains. Tripling the second knot by using knot vector $[0,1,1,1,2]$ gives the symmetrical function shown in Figure 3.18(c) which now only has positional continuity. Quadrupling this knot $[0,1,1,1,1]$ produces the function shown in Figure 3.18(d) where even positional continuity is eliminated.

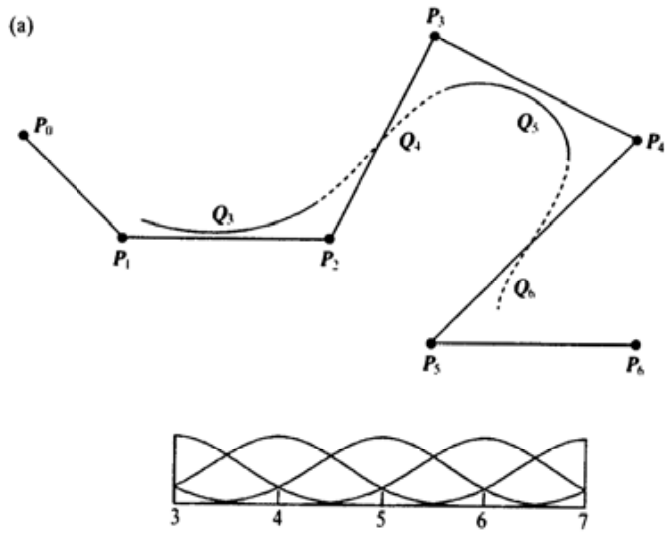
If we now return to the context shown in Figure 3.15. The first basis function is defined over $[0,0,0,0,1]$ and is asymmetric with no positional continuity. The second is defined over a set of knot values that contains a triple knot $- [0,0,0,1,2]$, the third over the sequence $[0,0,1,2,3]$ and is also asymmetric. In this case all functions are asymmetric and summarizing we have:

<i>Knot vector</i>	<i>Basis function</i>
0 0 0 0 1	B_0
0 0 0 1 2	B_1
0 0 1 2 3	B_2
0 1 2 3 3	B_3
1 2 3 3 3	B_4
2 3 3 3 3	B_5

We can further see from this set of basis functions that they sum to unity over the entire range of u and that at $u = 0$ and $u = 3$ the only non-zero basis functions are B_0 and B_5 (both unity) which cause the end points to be interpolated by Q_3 and Q_5 respectively.

We now consider altering the knot multiplicity for interior knots where the issue of continuity changes becomes apparent. Consider the examples given in Figure 3.19. This is the same example as we used in Figure 3.7 except that an extra control point has been added to give us a four segment curve. The knot vector is $[0,1,2,3,4,5,6,7,8,9,10]$ and Figure 3.19(a) shows the curve. Figure 3.19(b) shows the effect of introducing a double knot using vector $[0,1,2,3,4,4,5,6,7,8,9]$. The number of segments is reduced to three. Q_4 shrinks to zero. The convex hulls containing Q_3 and Q_5 meet on edge P_2P_3 and the join point between Q_3Q_4 and Q_4Q_5 is forced to lie on this line. In Figure 3.19(c) a triple knot is introduced $- [0,1,2,3,4,4,4,5,6,7,8]$. The curve is reduced to two segments. Q_4 and Q_5 shrink to zero at P_3 . There is only positional continuity between Q_3 and Q_6 but the segments on either side of the control point P_3 are

Figure 3.19
 The effect of interior knot multiplicity on a B-spline curve.
 (a) A four-segment B-spline curve. The knot vector is $[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$. All B-splines are translates of each other.



(b) Knot vector is $[0, 1, 2, 3, 4, 4, 5, 6, 7, 8, 9]$. Q_4 shrinks to zero.

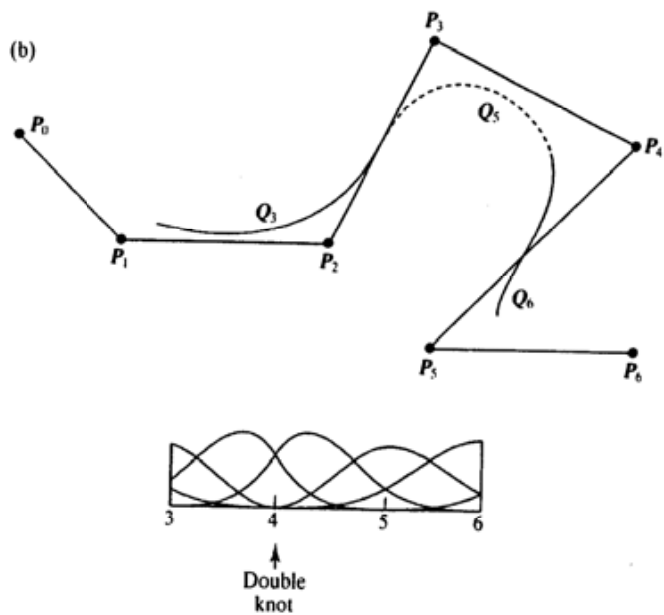
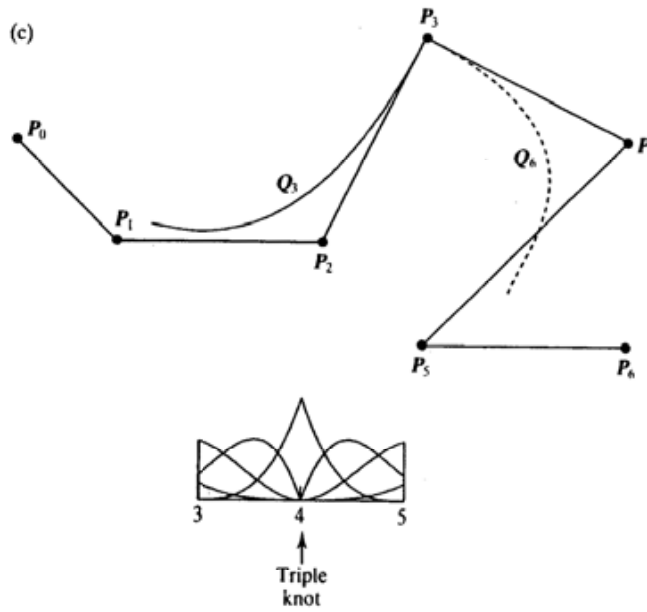


Figure 3.19 continued
 (c) Knot vector is $[0, 1, 2, 3, 4, 4, 4, 5, 6, 7, 8]$.
 Q_4 and Q_5 shrink to zero.
 Continuity between Q_3 and Q_6 is positional.



(d) Knot vector is $[0, 1, 2, 3, 4, 4, 4, 4, 5, 6, 7, 8]$.
 The curve reduces to a single segment Q_3 . Another control point has been added to show that the curve now 'breaks' between P_3 and P_4 .

