BRENDER

Tutorial Guide

Getting Started with BRender

Copyright © 1996 Argonaut Technologies Ltd. All rights reserved.
Argonaut Technologies Ltd. makes no expressed or implied warranty with respect to the contents of this manual, and assumes no responsibility for any errors or omissions which it may contain. No liability is assumed for any direct, indirect or consequential damages or losses arising in connection with the information or examples described herein.
Use of BRender is governed by the terms of the licence agreement included with the product.

Author: Robbie McQuaid
Technical Direction: Crosbie Fitch, Sam Littlewood, Dan Piponi, Philip Pratt, Simon Everett, Vinay Gupta, John Gay, Neela Dass, Tony Roberts
Project Managers: Paul Ayscough, Stefano Zammattio

MS-DOS and Windows 3.1, Windows NT and Windows 95 are registered trademarks of Microsoft Corporation.
3D Studio and AutoCAD are registered trademarks of Autodesk Inc.

Argonaut Technologies Limited
Capitol House, Capitol Way
Colindale, London NW9 0DZ
United Kingdom

Introduction 1

1 Setting The Scene 3

Scene Description 5

Co-ordinate Systems 6
Perspective and Parallel Projection 8
Geometric Transformations 9
Matrix Algebra and BRender 12
Visualisation 17
Hidden-Surface Removal 19

Calculating Colour 20

Colour and the CRT 20
Double Buffering 25
More about Hidden-Surface Removal 27

Describing Scenes in BRender 30

BRender Data Types 31
The Registry 31
BRender Program Structure 32
Conventions 32

2 Getting Started 33

Initialisation and Termination 35

Setting up the World Database 40

Animation Loop 42

More About Background Colour 42

3 Positioning Actors 45

Your Second Program 47

Transformation Function 50
Adding the Sphere and Torus 53
The Animation Loop 54

4 Actor Hierarchies 55

5 Adding Colour 61

The Program 64

8-Bit Indexed Colour Mode 65

6 Texture Mapping 71

Loading the Texture Map 72

Loading the Material 73

Assigning the Material 73

8-Bit Colour 77

7 File Conversion 81

Converting 3D Studio (.3ds) Files 82

Importing Texture Maps 88

15-bit True Colour 88

8-bit Indexed Colour 94

8 BRender Tools 95

3DS2BR 98

GEOCONV 100

DXF2BR 102

TEXCONV 103

MKSHADES 105

MKRANGES 105

PALJOIN 106

VIEWPAL 107

Importing Models into BRender 107

Working with 8-bit Colour 107

Working with True Colour 124

Introduction

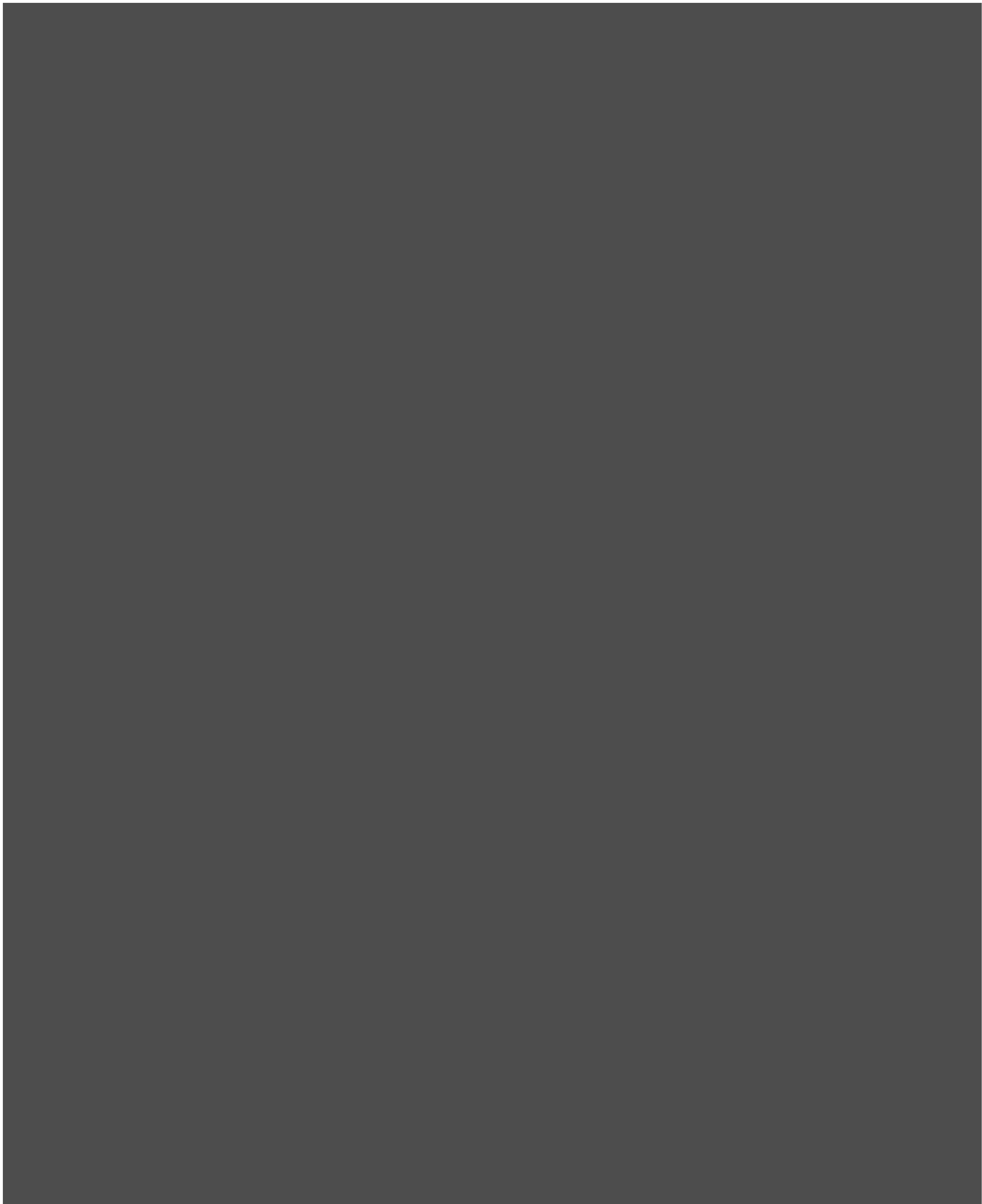
BRender is an extremely powerful real-time 3D graphics library. Its API (*Applications Programming Interface*) defines a set of C function calls. These function calls allow you to create interactive applications that generate three-dimensional images.

This book is your introduction to the world of BRender. It provides a step-by-step guide to the main features of the API and, we hope, the building blocks you will need to construct a conceptual model of BRender. Note that this is not a definitive guide to BRender and does not demonstrate the full range and power of BRender's features. It is intended to be read in conjunction with the technical reference manual.

The goal of this book is to help you produce your own 3D applications as quickly, and with as little fuss, as possible. To this end, sample programs are listed and dissected. We start with basic skeleton programs illustrating the structure of a BRender program and build these programs into more complex sequences. The source code for these programs can be found on the Tutorial Programs disk accompanying this guide. It is strongly recommended that you compile, edit and run these programs as you work through the tutorials. By experimenting with the supplied sample programs, you will quickly become familiar with BRender's vocabulary and learn how to implement its function calls.

Whilst you will need to be a proficient C programmer in order to make use of BRender, we do not assume an in-depth knowledge of 3D graphics principles. This is not a 3D graphics textbook however, only graphics principles relevant to BRender are described.

If you are new to writing 3D graphics applications, it is strongly recommended that you get your hands on a good 3D graphics textbook, for example *Computer Graphics – Principles and Practice*, Second Edition, by James D. Foley *et al.*, published by Addison Wesley Publishing Co., 1990. In addition, a general purpose maths primer covering vectors and matrices will prove invaluable.



1 Setting The Scene

This chapter presents an overview of BRender. It introduces a number of fundamental concepts you should be familiar with before you tackle your first BRender program. The structure and component features of a 3D computer graphics system are described, and how these features are realised in BRender. Readers new to 3D graphics will be introduced to a number of generic 3D graphics concepts.

So how does an imaginary three-dimensional world come to be represented by two-dimensional images on a computer screen? Consider someone in a room, looking through a window at the world outside. What that person sees through the window is determined by a number of factors: where they are positioned in the room, the direction and orientation of their gaze, the *field of view*, the lighting conditions etc.

A 3D computer graphics system attempts to mimic this viewing process.

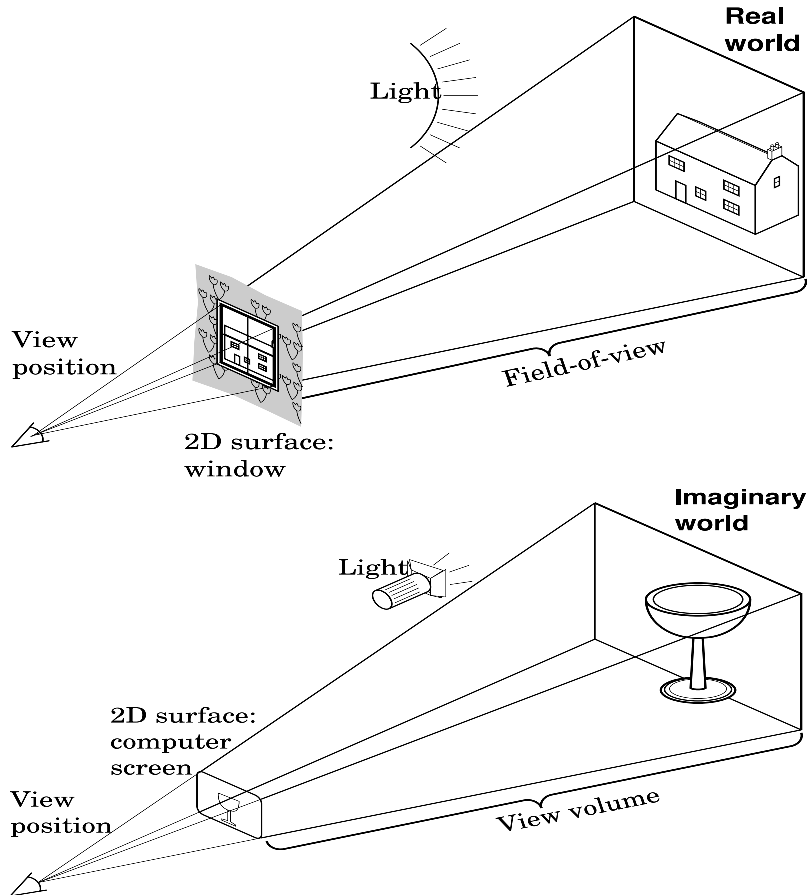


Figure 1 A 3D Graphics System mimics how we view the world through a window

A mathematical representation of an imaginary world is used to simulate the ‘real’ world. A number of user defined parameters determine what is actually displayed on (or projected onto) the computer screen. These include the view position (sometimes

referred to as the camera position), the view direction and orientation, the shape of the *view volume*, and the specified lighting conditions.

A 3D graphics simulation can be thought of as a three stage process. Firstly, an imaginary scene is described to the system in a language it understands. Then, a representation of what is currently 'visible' in the world is projected onto the screen. Finally, the colour of each picture element, or *pixel*, on the screen is calculated.

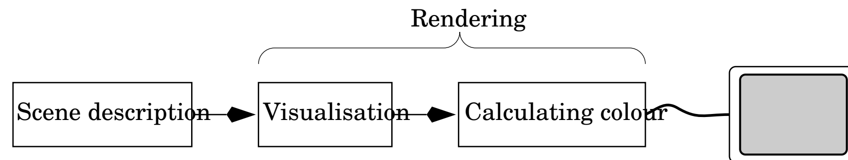


Figure 2 3D Graphics Simulation Sequence

The process whereby a scene gets drawn (stages 2 and 3 above), given an arrangement of previously defined models, is known as *rendering*. A very powerful rendering engine lies at the heart of BRender.

Scene Description

In a computer graphics system, a mathematical model of an imaginary scene is created and stored inside the computer. In order to accomplish this, the graphics system must provide some means of describing an imaginary scene to the computer. For the computer to generate a realistic simulation of the scene, it needs to know:

- the shape and position of all models (or model *actors* in BRender terminology) in the scene
- the colour and texture of these model actors
- the view (or camera) position, direction and orientation
- the lighting conditions.

The elements that make up a scene (typically models, lights and cameras) are described in BRender terms as **actors**. To avoid confusion at a later date, BRender terminology will be adopted from the beginning. What would be conventionally termed an 'object' will be here referred to as a model actor.

Co-ordinate Systems

In 3D computer graphics three-dimensional Cartesian co-ordinate systems are used to represent three-dimensional space. A model actor is defined initially in its own model co-ordinate system.

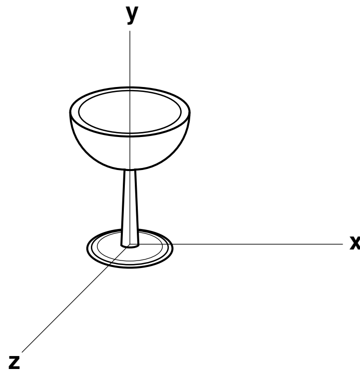


Figure 3 Models are defined in model co-ordinates

Note the orientation of the axes. With positive x to the right, and positive y pointing up, the positive z-axis points towards the viewer.

Actor data structures are designed to facilitate hierarchical relationships between elements of a scene. An actor may have a parent, and/or children. If an actor has no parent, it is the root actor of its hierarchy.

If an actor has a parent, its position and orientation is defined solely with respect to its parent's co-ordinate system, rather than with respect to some absolute co-ordinate system.

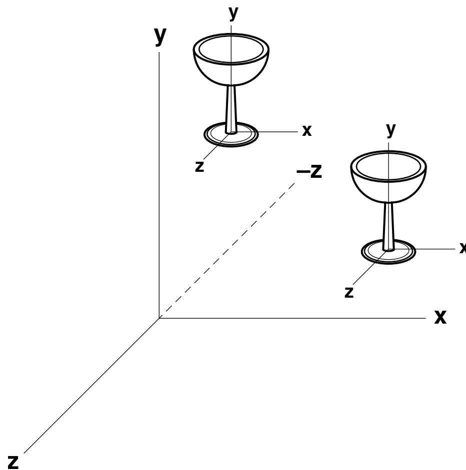


Figure 4 Actors transformed into parent's co-ordinate system

By default, an actor is centered at the origin in its parent's co-ordinate system. Geometric transformations, discussed below, are used to change the position, orientation and size of an actor in its parent's co-ordinate system.

If an absolute frame of reference is required, the application might define a universal parent actor (called 'World' perhaps), and make all other participants in a scene children of 'World'. 'World' 's co-ordinate system (call it 'world space') could then be used as an absolute frame of reference within the application.

Besides arranging the model actors that constitute the scene, the application must define a view position and orientation. These, together with other parameters such as lighting conditions and the shape of the view volume, determine which parts of a scene are visible.

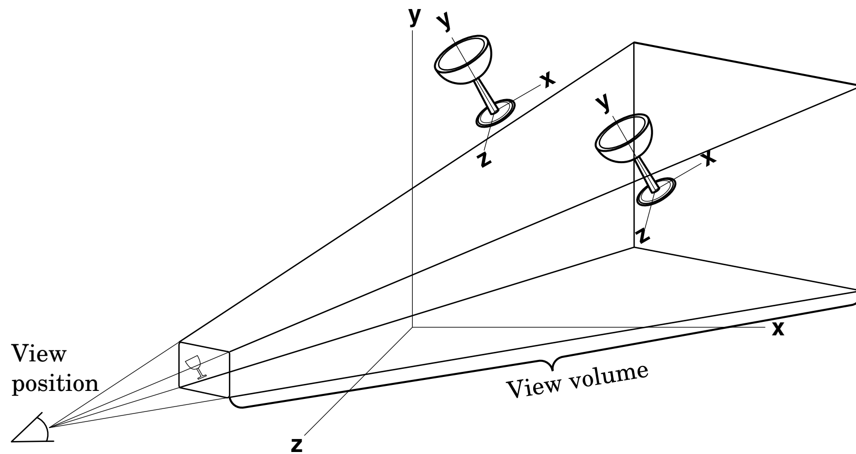


Figure 5 Everything outside the view volume is invisible

Model actors are tested, or *clipped*, against the sides of the view volume before being projected onto the screen. Everything lying entirely outside the view volume is eliminated. For lines or polygons that are partly inside and partly outside the view volume, the portions lying outside are eliminated. The edges of clipped polygons are reconstructed automatically.

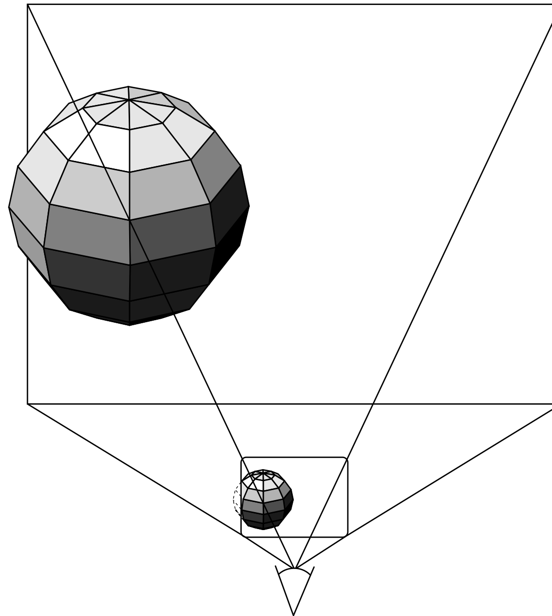


Figure 6 Model actors are clipped against the sides of the view volume

Perspective and Parallel Projection

The viewing process discussed above attempts to simulate how real world scenes are projected onto a 2D viewing surface (such as the viewer's 'window' or on the film in a camera). Projection rays (or vectors) reflected from the surfaces of models in the view volume converge at a focal point at the view position. This is known as *perspective projection*.

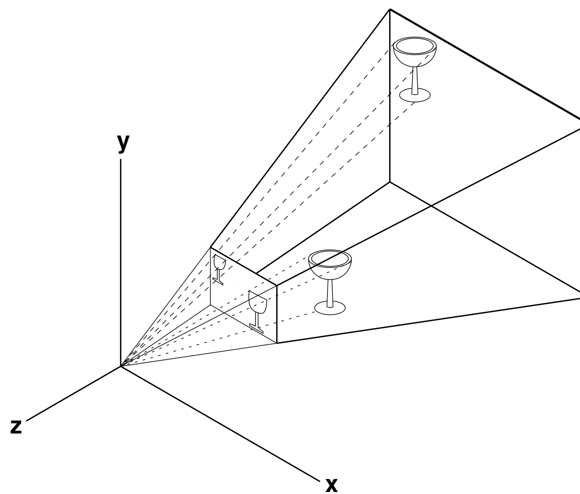


Figure 7 Perspective projection

It is important to realise that with perspective projection physical dimensions are relative. How big a model appears on the screen depends not only on its physical dimensions as defined in model co-ordinates, but also on its position in 3D space relative to the view position. The farther a model is from the camera, the smaller it appears on the screen.

The projection of a given model onto the screen could be enlarged, for instance, by increasing the model's dimensions (see 'Scaling' below), by moving it closer to the view position (see 'Translation' below), by moving the view position closer to the model or by decreasing the field-of-view angle (see 'Visualisation' below).

With a *parallel projection*, the view volume is a rectangular parallelepiped. Distance from the camera does not affect how large an object appears on the screen. Projection rays/vectors do not converge at a focal point but are projected along parallel lines.

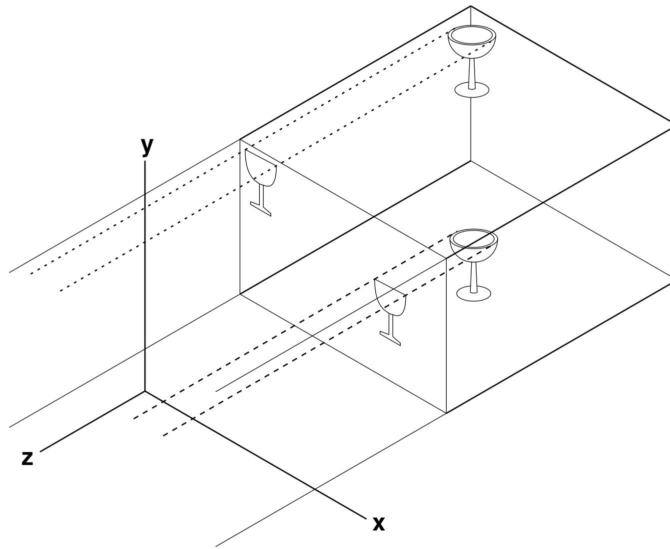


Figure 8 Parallel projection

Parallel projections are used in architectural and computer aided design applications where it is necessary to maintain relative sizes and angles after projection. In 3D computer graphics, perspective projections are usually specified. BRender supports both (the default is perspective).

Geometric Transformations

Geometric transformations are used to determine the position, orientation and size of an actor in its parent co-ordinate system. The translation, rotation and scaling transformations introduced below are essential to many graphics applications.They are the building blocks from which more complex transformations are constructed.

Translation

A translation transformation changes an actor's position in 3D space (relative to the origin in its parent's co-ordinate system). A translation $T(d_x,d_y,d_z)$ moves a point d_x units parallel to the x-axis, d_y units parallel to the y-axis and d_z units parallel to the z-axis.

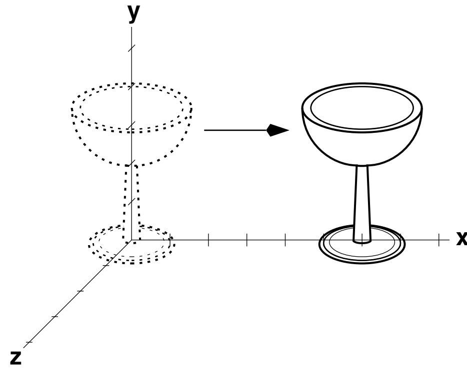


Figure 9 Translation: T(5,0,0)

Figure 9 illustrates a translation, T(5,0,0). The model actor is moved +5 units parallel to the x-axis (i.e. +5 is added to the x-components of the model's vertices). The y and z components are not altered.

Rotation

A rotation, as its name implies, rotates a model around an axis. A rotation $R_x(\theta)$, $R_y(\theta)$ or $R_z(\theta)$, rotates a point θ degrees around the specified axis.

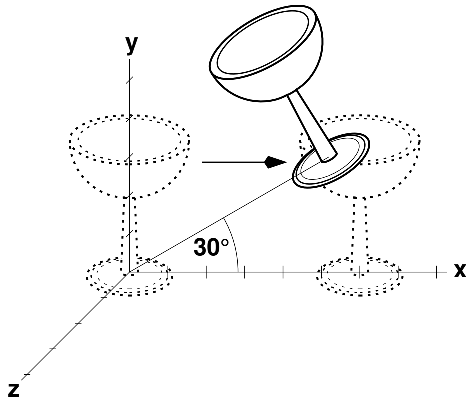


Figure 10 Rotation: $R_z(30)$

Figure 10 shows the, previously translated, model in Figure 9 rotated 30° around the z-axis.

Scaling

A scaling transformation changes the apparent size (and sometimes shape) of a model. A model can be scaled by multiplying the co-ordinates of its vertices by specified scaling factors. A scaling $S(s_x,s_y,s_z)$ multiplies the x component of a point by s_x, the y component by s_y, and the z component by s_z.

A scaling factor greater than 1 will 'stretch' a model in the appropriate x,y,z dimension. A value less than 1 will 'squeeze' it.

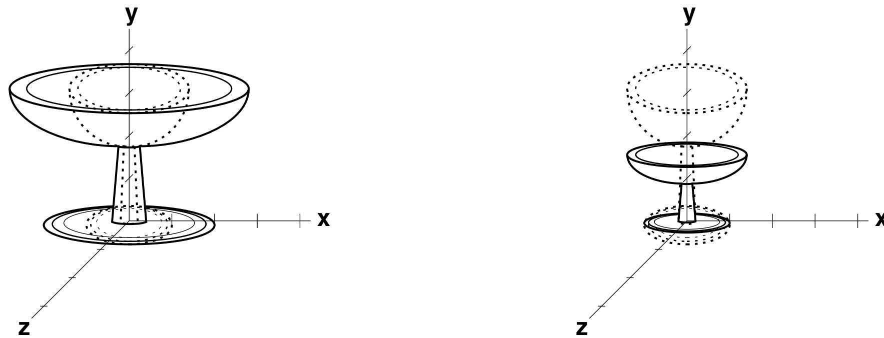


Figure 11 (a) Scaling: S(2,1,1); (b) Scaling: S(1,0.5,1)

The scaling applied in Figure 11(a) stretches the model horizontally along the x-axis (the x components of the model's vertices are multiplied by 2). The model appears smaller and fatter. In Figure 11(b) the model is squeezed vertically (all y values are halved).

If the same value is used for all three (x, y, and z) scaling elements, the model will retain its original proportions but change in size.

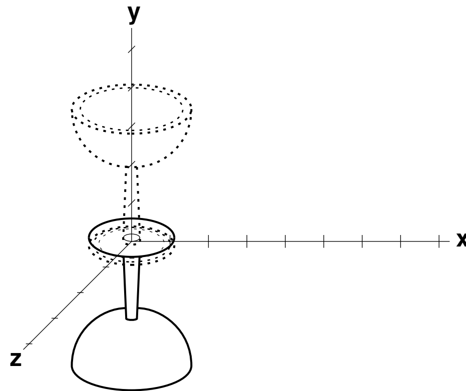


Figure 12 Scaling: S(1,-1,1)

An argument of -1 reflects the model in a plane. The scaling shown in Figure 12 reflects the model in the x-z plane.

A scaling of (2,2,2) doubles a model's length, width and depth – increasing its 'volume' by a factor of 8, as seen in Figure 13(a).

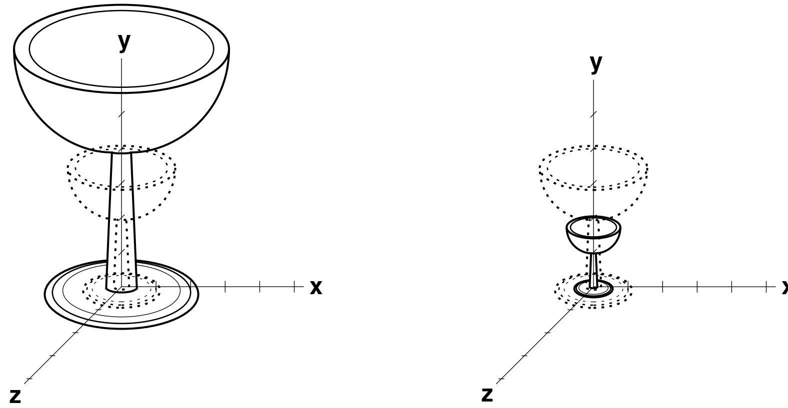


Figure 13 (a) Scaling: (2,2,2); (b) Scaling: (0.5,0.5,0.5)

Geometric transformations are most commonly represented in computer graphics by matrices. You don't need to understand matrix algebra in order to implement simple transformations. You specify the relevant parameters (direction and extent of the translation, the axis and angle of rotation etc.) and BRender sets up the transformation matrices.

However, in order to make full use of BRender's power and flexibility, you will need to understand how BRender implements matrix arithmetic. Each element of a BRender transformation matrix can be freely and individually accessed. If you are unfamiliar with the rudiments of matrix arithmetic, you may wish to consult your maths textbook before proceeding to the next section.

Matrix Algebra and BRender

General purpose 3D transformations are represented in BRender using `br_matrix34` data structures. These are effectively 4×4 matrices (in practice, the redundant fourth column is omitted for storage purposes and for speed).

Translation is conventionally treated as addition, whereas scaling and rotation are treated as multiplications. We want to be able to implement all three transformations in a consistent way, so that they can be easily combined. For this reason points are described in *homogeneous co-ordinates*. Homogeneous co-ordinates make it possible to implement all three transformations as multiplications.

In homogeneous co-ordinates, we add a fourth co-ordinate to a point – (x,y,z,W). The expression (x,y,z,W) represents the same point as (x/W,y/W,z/W,1). x/W, y/W and z/W are said to be the Cartesian co-ordinates of the homogeneous point. In practice, the W co-ordinate is almost always 1 and is omitted for storage purposes and

speed. See Section 5.2 of *Computer Graphics – Principles and Practice*, by James D. Foley *et al.*, for a more detailed discussion of homogeneous co-ordinates.

Translation

To translate point P, with co-ordinates (x,y,z), to a point P', with co-ordinates (x',y', z'), by means of translation $\mathbf{T}(\delta_x, \delta_y, \delta_z)$, BRender builds a translation matrix:

$$\mathbf{T}(\delta_x, \delta_y, \delta_z) = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \delta_x & \delta_y & \delta_z & 1 \end{pmatrix}$$

then pre-multiplies it by a row vector representing point P:

$$(x', y', z') = (x \ y \ z \ 1)\begin{pmatrix} 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \delta_x & \delta_y & \delta_z & 1 \end{pmatrix}$$

$$= (x + \delta_x, y + \delta_y, z + \delta_z)$$

Note that BRender's implementation of matrix arithmetic post-multiplies row vectors by matrices (rather than the alternative convention of post multiplying matrices by column vectors).

Rotation

BRender constructs the following matrix to implement a rotation θ about the z-axis:

$$\mathbf{R}(\theta) = \begin{pmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

This is easily verified. A unit vector along the x-axis, (1,0,0), rotated 90° around the z-axis, should produce a unit vector along the y-axis:

$$(1\ 0\ 0\ 1)\begin{pmatrix} 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} = (0\ 1\ 0\ 1)$$

The result (remembering that cos 90° = 0 and sin 90° = 1) is a unit vector along the y-axis, (0,1,0) as expected.

Scaling

A scaling matrix:

$$\mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ \delta_x & \delta_y & \delta_z & 1 \end{pmatrix}$$

would be built to implement a scaling transformation:

$$(x', y', z') = (x\ y\ z\ 1)\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= (xs_x, ys_y, zs_z)$$

Matrices can be concatenated to accumulate transformations. The compound transformation illustrated in Fig. 10 above for example, a translation followed by a rotation, could be represented by a single concatenated transformation matrix:

given that sin30°=0.5, and cos30°= $\frac{\sqrt{3}}{2}$. This transformation matrix would transform a point P = (0,0,0), for instance, to point P' = (5$\frac{\sqrt{3}}{2}$,2.5,0) as follows:

$$(x', y', z') = (0\ 0\ 0\ 1)\begin{pmatrix} \frac{\sqrt{3}}{2} & 0.5 & 0 & 0 \\ -0.5 & \frac{\sqrt{3}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{5\sqrt{3}}{2} & 2.5 & 0 & 1 \end{pmatrix}$$

$$= \left(\frac{5\sqrt{3}}{2}, 2.5, 0\right)$$

Matrix multiplication is not commutative, in other words:

$$\mathbf{M_1M_2 \neq M_2M_1}$$

$$\mathbf{T}(\delta_x, \delta_y, \delta_z)R_z(\theta) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} \cos 30° & \sin 30° & 0 & 0 \\ -\sin 30° & \cos 30° & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos 30° & \sin 30° & 0 & 0 \\ -\sin 30° & \cos 30° & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5\cos 30° & \cos 30° & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{\sqrt{3}}{2} & 0.5 & 0 & 0 \\ -0.5 & \frac{\sqrt{3}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ \frac{5\sqrt{3}}{2} & 2.5 & 0 & 1 \end{pmatrix}$$

The order in which you specify transformations is thus significant. Consider the consequences of reversing the order in which the above transformations are implemented. A 30° rotation about the z-axis, followed by a translation T(5,0,0), would generate the image in Figure 14(b).

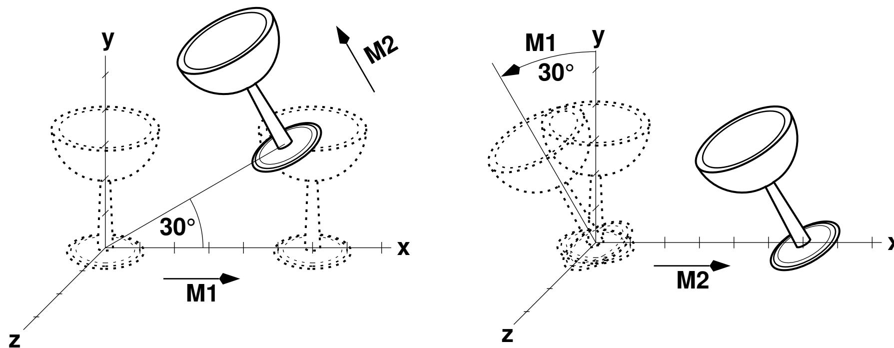


Figure 14 (a) Translate then Rotate: $\mathbf{T}(\delta_x\delta_y\delta_z)\mathbf{R}_z(\theta)$; (b) Rotate then Translate: $\mathbf{R}_z(\theta)\mathbf{T}(\delta_x\delta_y\delta_z)$

The resultant concatenated transformation matrix is given below.

$$R_z(\theta)\mathbf{T}(\delta_x,\ \delta_y, \delta_z) = \begin{pmatrix} \cos 30° & \sin 30° & 0 & 0 \\ -\sin 30° & \cos 30° & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \cos 30° & \sin 30° & 0 & 0 \\ -\sin 30° & \cos 30° & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & \cos 30° & 0 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} \frac{\sqrt{3}}{2} & 0.5 & 0 & 0 \\ -0.5 & \frac{\sqrt{3}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{pmatrix}$$

This transformation matrix would transform a point P = (0,0,0), for instance, to point P' = (5,0,0) as follows:

$$(x', y', z') = (0\ 0\ 0\ 1) \begin{pmatrix} \frac{\sqrt{3}}{2} & 0.5 & 0 & 0 \\ -0.5 & \frac{\sqrt{3}}{2} & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 5 & 0 & 0 & 1 \end{pmatrix}$$

$$= (5,0,0)$$

Clearly, matrix multiplication is not commutative.

Remember to carefully consider the order in which you specify transformations. You may wish to experiment with the tutorial programs presented in the following chapters by changing the order in which geometric transformations are implemented.

Visualisation

As noted above, what is actually displayed on the screen once an imaginary scene has been defined depends on the view position and orientation, and on the shape of the

view volume. The view volume corresponding to BRender's default camera is illustrated below.

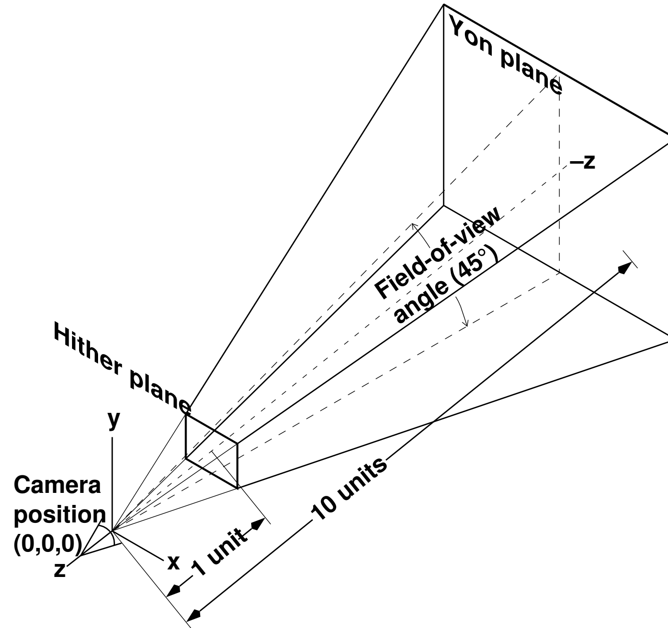


Figure 15 The Default Camera

Note that the default camera is perspective. The field of view is the angle subtended between the top and bottom of the view volume, typically 45° in BRender. Setting this angle is analogous to selecting the focal length of a photographic lens. A narrow-angle field of view simulates a telephoto lens. Models appear closer and larger.

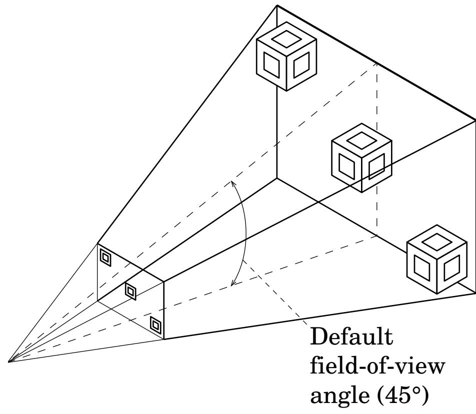


Figure 16 (a) Default Field of View (45°)

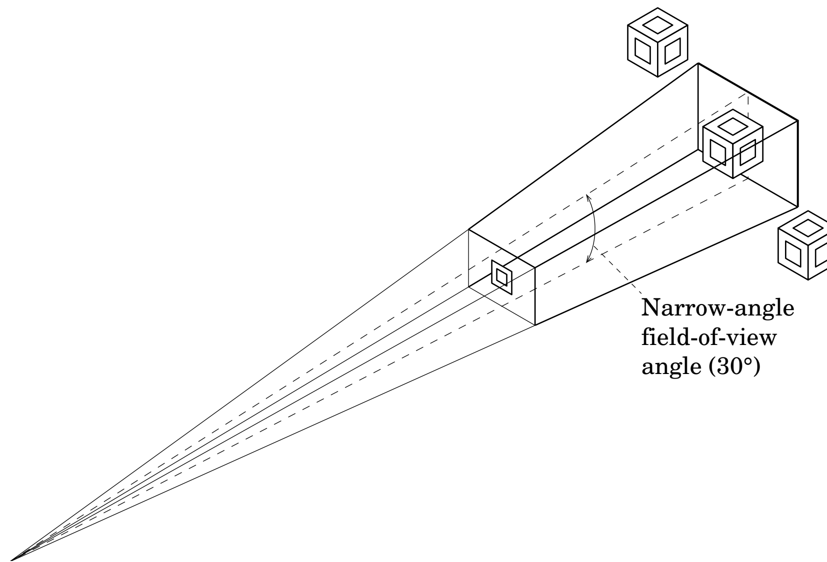


Figure 16 (b) A narrow-angle Field of View

A wide-angle field of view simulates a wide-angle lens. Models appear smaller, and further away.

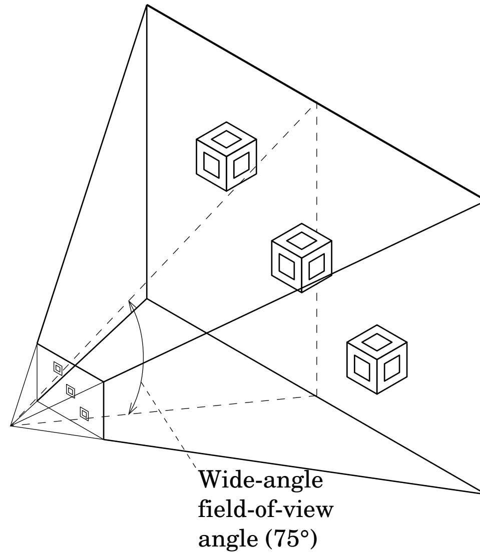


Figure 17 A wide-angle field of view

The field-of-view angle, together with the (user definable) *hither* and *yon* planes, determines the shape of the view volume. The shape of the view volume, in turn, determines what is potentially visible. The renderer tests models against the edges of the view volume to determine whether or not they are inside it. Models found to lie outside the view volume are not considered further.

Hidden-Surface Removal

Once the renderer has determined which models are inside the view volume, it must establish which surfaces are actually visible. A model may lie within the view volume, but be entirely obscured by another model positioned closer to the view position.

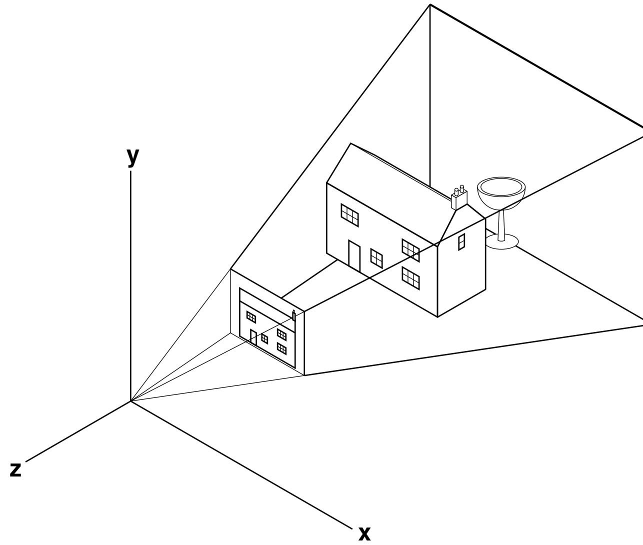


Figure 18 Hidden surfaces are not displayed

Hidden-Surface Removal techniques are used to ensure that only surfaces visible from the view position are displayed. Surfaces must be sorted according to their position in 3D space relative to the view position. This is necessary in order to determine which surfaces are visible, and which are obscured by surfaces nearer the view position.

Calculating Colour

Once an imaginary world has been described, and the renderer has determined which elements are currently visible, it must then decide how to display these visible elements – what colours to apply and where. To understand how BRender handles colour, let's start by briefly considering how monitors display colour.

Colour and the CRT

The inside of the glass screen on a colour monitor is covered with closely packed groups of red, green and blue phosphor dots. These phosphor dots emit red, green and blue light, respectively. Each group of dots is so small that light emanating from them is perceived by the viewer as a single colour.

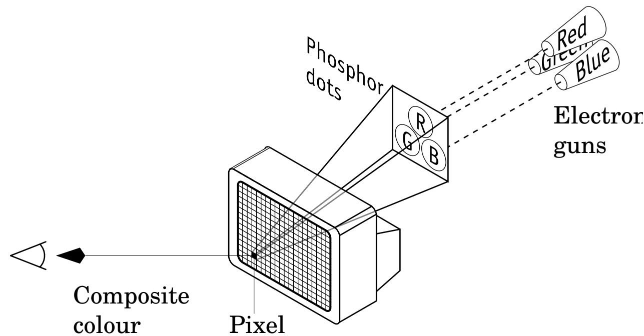


Figure 19 CRT screen

Three electron guns rapidly scan the screen from top to bottom, one line at a time. As each pixel is addressed, the intensities of the electron beams fired from the red, green and blue electron guns are varied to reflect the amount of red, green and blue light in that pixel's colour. In this way a picture is drawn on the screen. The entire screen is scanned many (typically 60 or more) times a second – too fast for the human eye to detect any movement. The viewer sees a constant unflickering picture (the same principle is at work in the cinema, where movie frames projected at 24 frames per second are blended together into a smooth sequence inside the viewer's brain).

The electron guns get their colour information, via a video signal, from a *screen buffer*. The screen buffer, also known as the *frame buffer*, is an area of memory mapped directly to the screen.

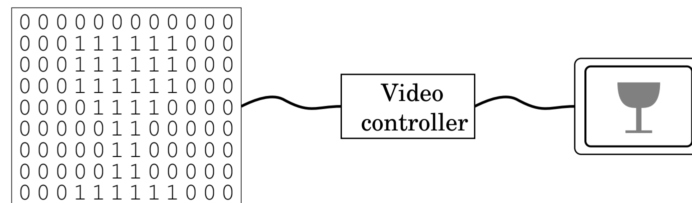


Figure 20 Screen buffer mapped to screen

Dedicated circuitry, known as the *Video Controller*, interprets the contents of the screen buffer, pixel by pixel, and automatically updates the screen. The renderer is responsible for updating the screen buffer. These operations are transparent to the programmer, who writes an application describing a scene and expects an accurate representation of this scene to be displayed on the screen.

The screen buffer 'depth' (or the number of bitplanes – a bitplane contains one bit of data for each pixel) determines how many colours can be displayed. In a 3-bit screen buffer, for example, a single bit is available for each of the *r*, *g* and *b* (red, green and

blue) components. Each is either fully ON or OFF. A total of eight colours are available to choose from (000 to 111).

Colour Components			Value	Colour
r	*g*	*b*		
0	0	0	0	black
0	0	1	1	blue
0	1	0	2	green
0	1	1	3	cyan
1	0	0	4	red
1	0	1	5	magenta
1	1	0	6	yellow
1	1	1	7	white

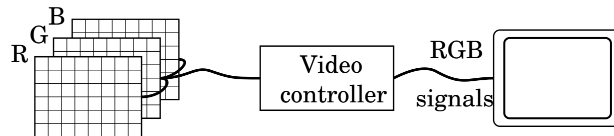


Figure 21 A 3-bit screen buffer contains three bit-planes

The number of colours that can be displayed may be changed by using a *colour lookup table* (or CLUT). The values stored in the frame buffer no longer generate colours directly, but are used to point to, or *index*, locations in the CLUT. A typical CLUT may store each colour as a 15-bit number, 5 bits for each r,g,b component.

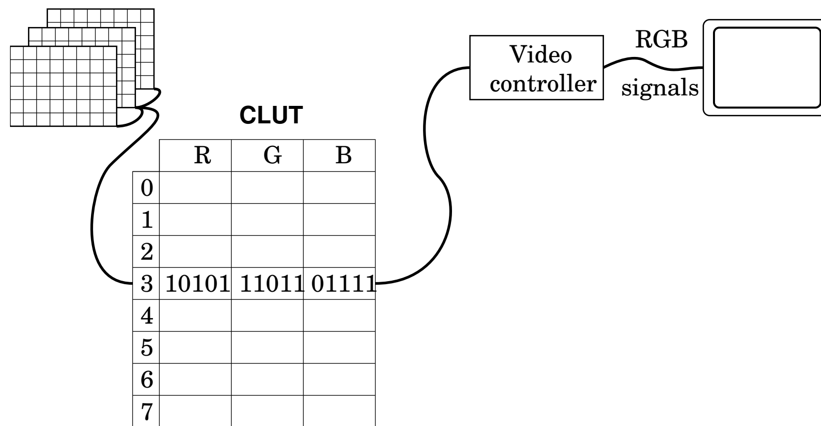


Figure 22 3-bit indexed mode

Note that the number of colours available at any one time does not change. In the 3-bit system depicted in Figure 22, there are still only 8 colours to choose from at any

one time. However, the CLUT may contain any 8 colours from a total of 2^{15} (or 32, 768). A selection of colours, or a colour palette, must be loaded into the CLUT before rendering begins. In indexed colour mode, one or more colours in a given image can be changed simply by loading a different colour palette. This is a potentially powerful feature. Colour manipulation and animation effects can be implemented using a fraction of the computational power that would be needed to achieve the same results by updating the stored image.

BRender colour modes include 8-bit indexed colour, and 15- and 24-bit non-indexed or true colour.

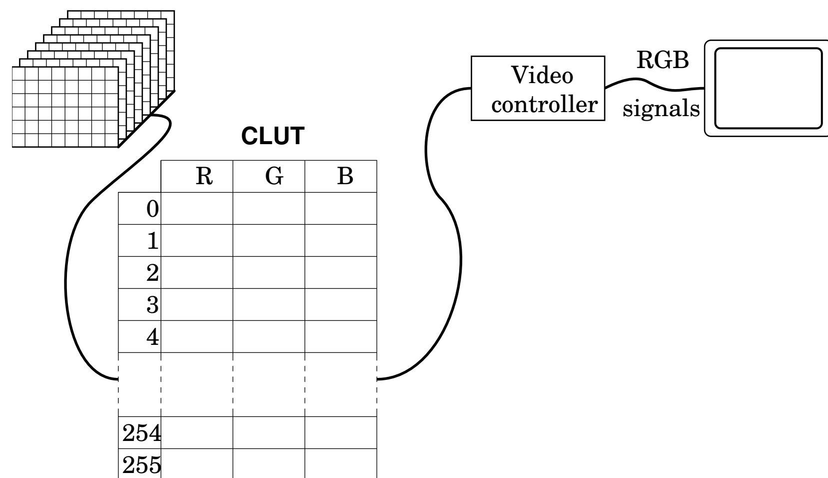


Figure 23 8-bit indexed mode

A total of 256 (2^8) colours are available at any one time in 8-bit indexed mode. In 15-bit true-colour mode, there are 32,768 (2^{15}) colours to choose from. In 24-bit mode 2^{24} (≅16.7 million) different colours are available.

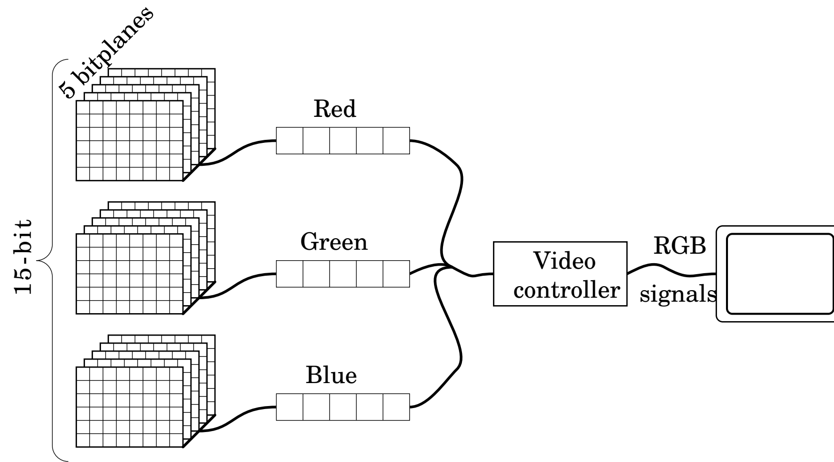


Figure 24 (a) 15-bit true colour

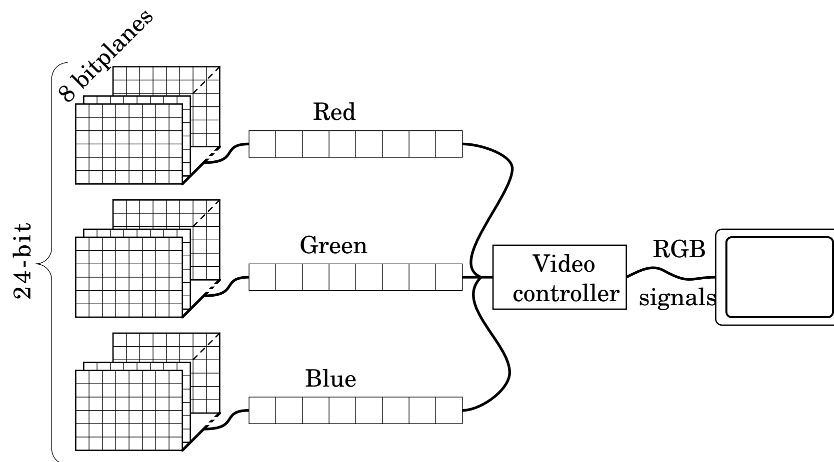


Figure 24 (b) 24-bit true colour

Indexed colour is more complex to work with than true colour. A colour palette must be set up and loaded before rendering begins. It must be sufficiently general to be useful, yet satisfy light intensity requirements. Faces on a model should appear lighter or darker according to their orientation with respect to the light source/s. In order to simulate realistic lighting conditions, various shades (from light to dark) of each colour in a scene should be available from the colour palette. In practice, the palette may be divided into a number of strips, or ramps. Each strip consists of a variety of shades of a single colour, ranging from very light to very dark. The palette must also be capable of displaying acceptable approximations of texture maps that may have been created using a completely different palette. For a more detailed discussion of indexed colour and texture mapping, see Chapters 6 to 8.

Video Modes

A variety of user-selectable video modes are usually available on a particular platform. Most systems support 8-bit indexed colour along with a variety of true-colour modes. The screen resolutions available will depend, in part, on the size of the video memory. By changing the video mode you can alter the screen resolution, as well as the number of colours available. Commonly used PC screen resolutions, for example, include the 320 × 200 and 640 × 480 VGA standards, 1024 × 768 and 1280 × 1024 SVGA standards and the intermediate 800 × 600. If you don't know which video modes are available to you, run the appropriate BRender hardware interrogation utility (if available on your platform – refer to your installation guide). For BRender x86, for instance, this is `VESAQ.EXE`, located in the Tools directory. `VESAQ.EXE` interrogates your video circuitry before displaying a list of the video modes supported by your system.

Note that the task of making BRender output visible to the user belongs to the application. BRender provides assistance whenever possible.

When you write a BRender program, you must select a video mode. This is discussed in detail in Chapter 2, where your first BRender program is introduced. In general, the lower the screen resolution, the faster your program will run (since there are fewer pixel values to calculate). Of course your images will lose sharpness at lower resolutions. In 3D computer graphics there is always a trade-off between speed and image quality. Since speed is a crucial factor in real time animation, you will probably want to work with low screen resolutions, certainly during the initial development stage (of course a higher, slower resolution may prove more useful for some applications). Most of the animations included on the Tutorial Programs disk are displayed using the lowest available screen resolution.

Double Buffering

As noted above, the Video Controller repeatedly scans the screen buffer, refreshing the screen as it does so. In effect, the contents of the screen buffer are permanently displayed.

Now consider what the viewer would see if scenes were rendered directly into the screen buffer. Since the screen buffer is mapped directly to the screen, the frame construction process would be visible and the viewer would be aware of discontinuities in rapidly changing scenes.

A simple solution is to use two buffers. The renderer constructs a scene in a back buffer while the screen, or *front*, buffer is being displayed. When the scene in the back buffer is complete and it is time to display it, the application 'swaps' the buffers so that the back buffer becomes the screen buffer. The next frame is then constructed in the new back buffer (the old screen buffer).

Some systems come with two hardware screen buffers, so that back and front buffers are literally swapped as described above. Where only a single screen buffer exists in hardware, BRender sets up a back buffer in memory. When it is time to display a newly constructed scene, BRender copies the contents of the back buffer into the screen buffer.

How double buffering is implemented is transparent to the applications programmer, who simply calls BRender's double buffering function at the appropriate point in the program.

More about Hidden-Surface Removal

Figure 25 illustrates the importance of hidden surface removal. Figure 25(a) looks correct. But what about Figure 25(b). This is what would be displayed if the model nearest the viewer were rendered into the screen buffer first.

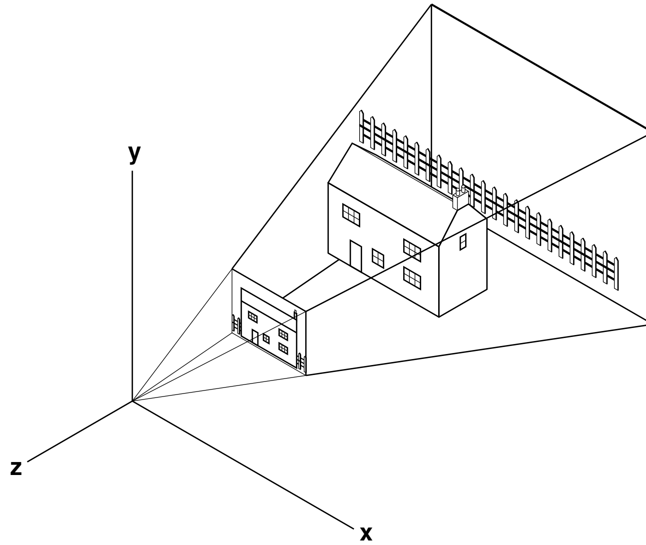


Figure 25 (a) Fence rendered first

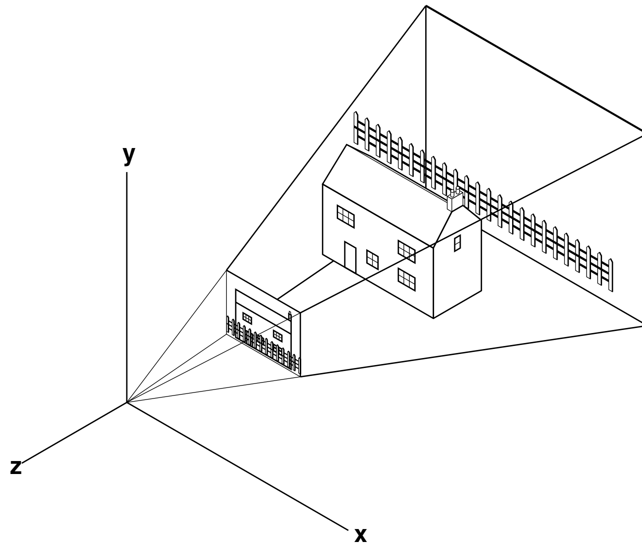


Figure 25 (b) House rendered first

The Z-Buffer Renderer

The Z- or *depth* buffer is an effective hidden-surface removal algorithm. Remember that colour information for each pixel is initially written to the back buffer. A depth buffer with the same number of entries is used to store a z, or depth, value for each pixel. The depth buffer is initialised to a large number beyond the visible range (e.g. FFFF for a 16-bit buffer). You could think of this value as the z-position of the background.

Each potentially visible surface is rendered in turn, in an arbitrary order. The magnitude of the z-component of each visible point on a surface is compared with the relevant value in the Z-Buffer. If it is nearer or as near as the point whose colour and depth are currently in the buffers, then the new point's colour and depth replace the old values. At any time, the Z-Buffer and the back buffer will store the information associated with the lowest (in magnitude) z value encountered thus far. When the back buffer is subsequently swapped with the screen buffer, the colour displayed at any pixel is that of the surface closest to the view position.

Suppose Surface 1 in Figure 26 is rendered first. Since 10, the magnitude of its z-component, is less than the z-component of the background (assumed to be initialised to FFFF), the colour and depth information associated with Surface 1 are copied to the buffers.

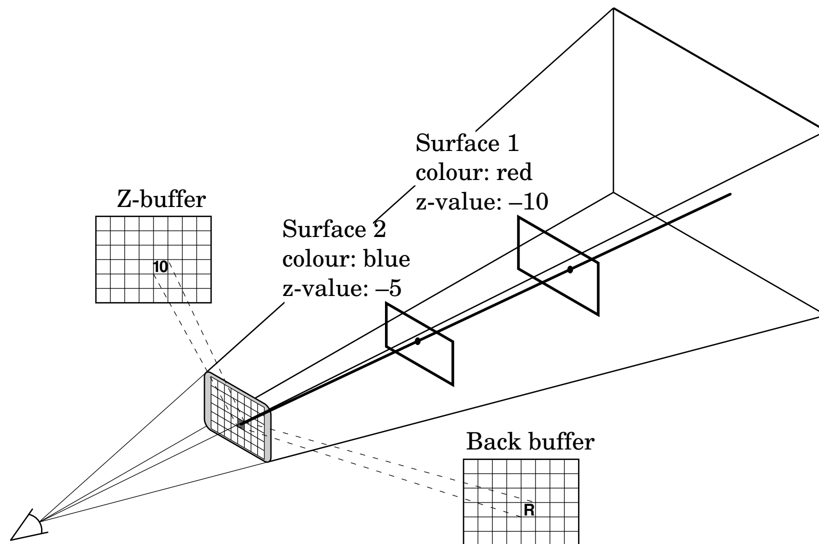


Figure 26 Surface 1 rendered first – since |-10| < |FFFF|, pixel colour is red

If Surface 2 is then rendered, its associated depth and colour information will replace those of Surface 1, since it is closer to the view position (5<10).

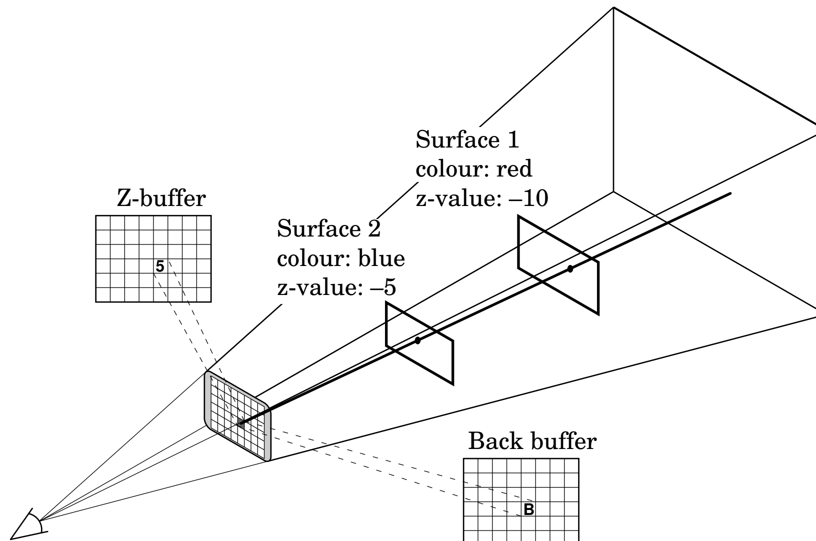


Figure 27 Surface 2 rendered next – since |-5| < |-10|, final pixel colour is blue

For each pixel the z-value of the closest surface so far considered will be stored in the depth-buffer and the colour value for this surface stored in the back buffer.

The applications programmer is responsible for setting up and initialising the depth buffer (in addition to the screen and back buffers) using BRender functions.

The Z-Buffer is a fast, high quality renderer. However, it requires a considerable amount of memory.

The Z-Sort Renderer

The Z-Sort renderer attempts to achieve hidden surface removal by sorting surfaces according to their distance from the view position, and then drawing them from back to front.

The Z-Sort is a fast renderer that requires relatively little memory (particularly for simple scenes). Hidden Surface Removal schemes are discussed in greater detail in your technical reference manual.

Describing Scenes in BRender

So how does the applications programmer go about describing a scene to BRender? What 'scene description language' does BRender understand? How is the interface between the applications programmer and BRender defined and structured?

BRender data structures are used to convey scene description information to the renderer. The **actor** data structure is fundamental to scene description. All participants in a scene are categorised as actors. An actor can have a number of children, each of whom is also an actor. In this way a tree of actors is built up representing the world.

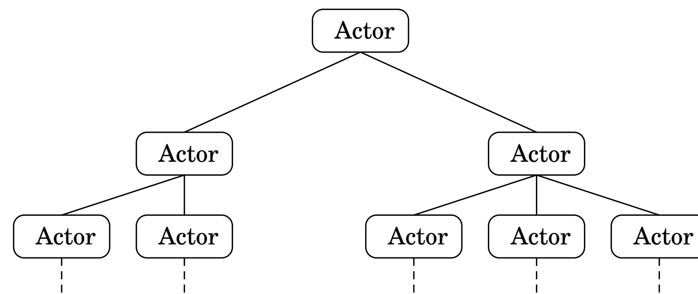


Figure 28 Simple tree of actors

Different types of actors are used to represent different entities. Commonly used actor types include **Models**, **Lights**, **Cameras** and **None** (a reference actor used to assist in the layout and organisation of actor hierarchies – invariably found at the root of an actor tree). The smallest actor hierarchy that will produce a rendered scene is depicted in Figure 29.

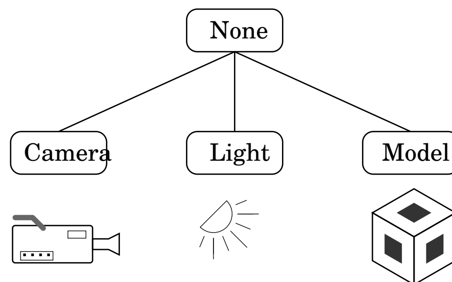


Figure 29 Four-actor tree

This scene contains a single model actor that references information describing its shape, colour, texture and position. The light actor references information describing the type of lighting used to illuminate the scene (point, direct or spot), its position, orientation and colour. The camera actor references information describing the view volume including the view position, the field of view and how the scene is projected onto the viewing surface. The None, or dummy, actor doesn't usually reference any

data (although it can) but is used to build the hierarchical tree structure. It forms the root of the actor tree used to describe the world.

A model actor may specify, or inherit from a parent, a default ‘material’. If a material is not explicitely assigned to a model, it inherits its parent’s default material. If no material is associated with the parent actor (or if no parent actor exists) a default flat-shaded grey material is used. The `br_material` data structure contains information about the appearance of a surface – its colour, whether the finish is flat or Gouraud, etc. The material data structure may reference a texture map. Texture mapping is considered in more detail in Chapter 6. Fundamentally, it’s a process whereby a two-dimensional pattern is wrapped around a three-dimensional model.

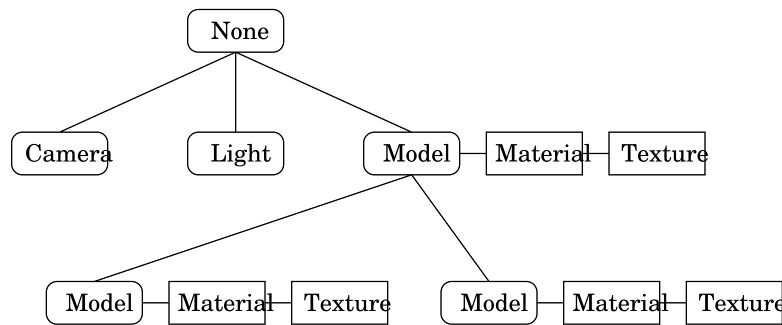


Figure 30 Models can reference materials and texture maps

This is how the world is represented in BRender – a tree of actors referencing models that reference materials that reference texture maps.

BRender Data Types

BRender defines certain data type classes in order to ensure cross platform portability. Macros are provided to convert standard C data types to appropriate BRender data types.

The Registry

Some items used to describe actors are preprocessed by BRender to minimize rendering time. These items are made available for such preprocessing by placing them in the Registry. The Registry is a database that keeps track of registered items (models, materials, texture maps and shade tables) and manages the resources used in rendering.

Registry operations are largely transparent to the user. However, it is important to remember that actors, models, materials and pixel maps (described below) must be registered before they can be used. They can then be referred to by name. A Registry update should be performed whenever registered items are changed.

BRender Program Structure

The structure of a typical BRender program is depicted below:

Initialise

- User Code
- Initialise BRender
- Select and enable the Rendering Engine

Set up the World Database

- Load/Create Models and associated Materials, Texture Maps and Palettes (if using indexed colour)
- Define the relationships between actors – the World Hierarchy

Event Loop

- Alter scene description as necessary: move/resize/delete Models etc.
- Pass current scene description to the Rendering Engine
- Display the image returned by the Renderer.
- User interaction

Terminate

- Close the Rendering Engine
- Close BRender
- Close User Code

Conventions

Some function calls, particularly those concerned with I/O operations, are platform specific. When platform specific function calls are included in program listings in this manual, they appear in *italics*. The sample programs on your Tutorial Programs disk contain the appropriate format for your platform. Refer to these sample program listings and to your installation guide for further details.

2 Getting Started

It's time to write your first program. BRender supplies suitable default values, where appropriate, for parameters not defined by the user. The colour of the default material, for instance, is matt grey. The default camera actor is positioned at the origin looking along the negative z-axis (in the camera's co-ordinate system). The default light actor is a direct light, with the light shining along its negative z-axis. BRender even provides a default model actor – a cube.

These defaults are depicted in Figure 31.

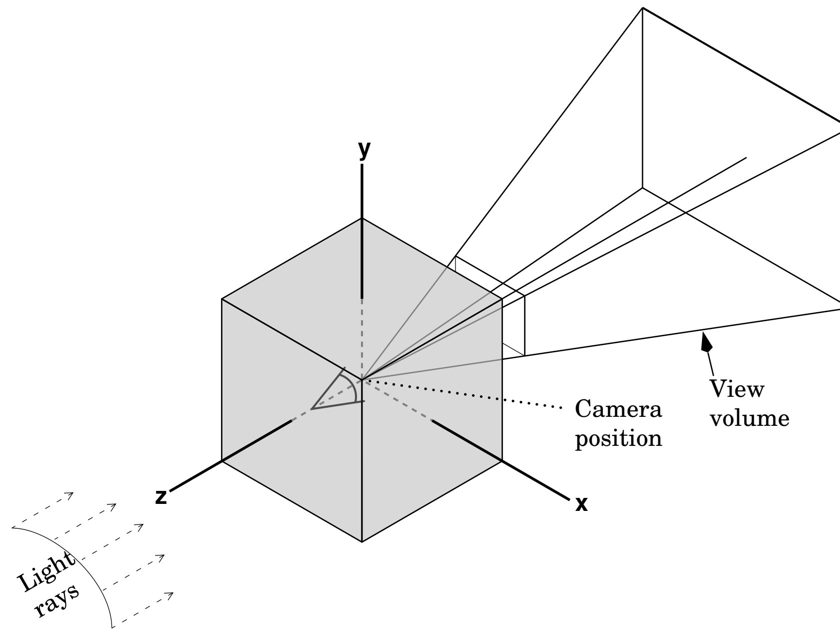


Figure 31 BRender defaults

Note the camera position in the centre of the unit cube. To display the cube, either the camera or the cube itself would need to be re-positioned.

The first program on your Tutorial disk is called `BRTUTOR1.C` and is listed below. Compile and run it to display a revolving grey cube. BRender programs don't come much simpler than this one. It accepts the defaults depicted above. However, the camera is re-positioned along the positive z-axis so that the cube becomes visible. The cube is then rotated 30° around the y-axis before being set in motion rotating around the x-axis.

Let's examine the program in detail.

Two fundamental BRender data structures are introduced in this program – `br_actor` and `br_pixelmap`. Models, lights and cameras are described by means of `br_actor` data structures. Note the widespread use of pointers. Most BRender functions return pointers to data structures. Be careful to avoid type mismatches when assigning variable values.

A `br_actor` data structure is really an index system – it contains pointers to other structures that describe the properties of the relevant actor. `br_pixelmap` structures contain information describing buffers, palettes and texture maps. Keep your technical reference manual handy when working through the BRender programs described in this guide. Familiarise yourself with BRender functions, data structures and argument data types as they are introduced. You may also wish to consult relevant BRender header files and to refer to the sample program listings on your Tutorial Programs disk.

Recall the structure of a typical BRender program outlined in Chapter 1. The Initialisation and Termination components of `BRTUTOR1.C` are isolated below.

Initialisation and Termination

`BrBegin` must be called before most BRender functions can be used. `BrEnd` frees internal resources and memory. These will normally be the first and last function calls in your BRender programs.

```
BrBegin();

/*
 * Initialise screen buffer and set up CLUT (ignored in true
             colour)
 */
screen_buffer = DOSGfxBegin(NULL);
palette = BrPixelmapLoad("std.pal");
if(palette)
   DOSGfxPaletteSet(palette);

/*
 * Initialise Z-Buffer renderer
 */
BrZbBegin(screen_buffer->type,BR_PMT_DEPTH_16);

          :          :          :

BrZbEnd();
DOSGfxEnd();
BrEnd();
```

BRTUTOR1.C

```
/*
 * Program to Display a Revolving Illuminated Cube
 */
#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"
int main(int argc, char **argv)
{
  /*
   * Need screen and back buffers for double-buffering, a Z-Buffer to store
                                                 current
   * depth information, and a storage buffer for the currently loaded palette
   */
  br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
  /*
   * The actors in the world: Need a root actor, a camera actor, a light actor,
   * and a model actor
   */
  br_actor *world, *observer, *light, *cube;
  int i;                                                     /*counter*/

/********** Initialise BRender and Graphics Hardware ****************/
  /*
   * Start BRender
   */
  BrBegin();
  /*
   * Initialise screen buffer and set up CLUT (ignored in true colour)
   */

                      :         :         :

  /*
   * Initialise Z-Buffer renderer
   */
  BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);
  /*
   * Allocate back buffer and depth buffer
   */
  back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
  depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database **********************/
  /*
   * Start with None actor at root of actor tree and call it 'world'
   */
  world = BrActorAllocate(BR_ACTOR_NONE,NULL);
  /*
   * Add a camera actor as a child of None actor 'World'
   */
  observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
```

```
  /*
   * Add, and enable, the default light source
   */
  light = BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL));
  BrLightEnable(light);
  /*
   * Move camera 5 units along +z axis so model becomes visible
   */
  observer->t.type = BR_TRANSFORM_MATRIX34;
  BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                        BR_SCALAR(5.0));
  /*
   * Add a model actor: The default unit cube
   */
  cube =  BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
  /*
   * Rotate cube to enhance visibility
   */
  cube->t.type = BR_TRANSFORM_MATRIX34;
  BrMatrix34RotateY(&cube->t.t.mat,BR_ANGLE_DEG(30));

/****************************** Animation Loop ******************************
  /*
   * Rotate cube around the x-axis
   */
  for(i=0; i < 360; i++) {
       /*
        * Initialise depth buffer and set background colour to black
        */
       BrPixelmapFill(back_buffer,0);
       BrPixelmapFill(depth_buffer,0xFFFFFFFF);
       /*
        * Render scene
        */
       BrZbSceneRender(world,observer,back_buffer,depth_buffer);
       BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
       /*
        * Rotate cube
        */
       BrMatrix34PostRotateX(&cube->t.t.mat,BR_ANGLE_DEG(2.0));
  }
  /* Close down */

  BrPixelmapFree(depth_buffer);
  BrPixelmapFree(back_buffer);
  BrZbEnd();

                    :          :          :

  BrEnd();
  return 0;
}
```

BRTutor1.c

Platform specific code, omitted from the program listings in this tutorial, is required to initialise display hardware, colour buffers and palettes. Sample hardware initialisation code is included in the program listings at the back of your platform-specific installation guide, and in the sample programs on your Tutorial Programs disk. In general, this sample code contains generic examples of how screen handling operations may be implemented on your platform. You may decide to optimize this code for your own purposes, or substitute your own screen handling and I/O routines.

The sample initialisation code listed above illustrates how a DOS-based application might initialise the graphics hardware. This code is discussed below, for illustrative purposes. If you are not running BRender x86 under DOS, refer to your installation guide and to the sample program listings on your Tutorial Programs disk.

A number of screen handling functions are provided for use with DOS-based BRender applications. These functions are provided to simplify graphics hardware initialisation and are documented in the BRender x86 Installation Guide.

DosGfxBegin and *DosGfxEnd* are used to initialise the graphics hardware.

DOSGfxBegin expects a character string argument in the following format:

```
VESA/MCGA, [W:<width>],[H:<height>],[B:<bits/pixel>]
```

The default string `MCGA,W:320,H:200,B:8` is assumed if NULL is passed as an argument to *DOSGfxBegin* (and the *BRENDER_DOS_GFX* environment variable has not been set). BRender x86 v1.2 supports 15- and 24-bit true colour, as well as 8-bit indexed colour.

If you don't know which graphics screen modes are supported by your system, run `VESAQ.EXE`, BRender's hardware interrogation utility for DOS based systems. This utility, located in the Tools directory, interrogates your video circuitry before displaying a list of the video modes supported by your system.

If NULL is passed as an argument to *DOSGfxBegin*, the screen mode can be altered after compilation by setting the *BRENDER_DOS_GFX* environment variable. For example, type:

```
SET BRENDER_DOS_GFX=VESA,W:640,H:400,B:8
```

followed by Return to select the 640 × 400, 8-bit VESA standard (assuming your video card supports this mode). Now run `BRTUTOR1.C` again. Notice that the program runs much slower at this higher screen resolution and that the quality of the image has improved (the 'staircase' effect at the edges of the cube is not as pronounced). Real-time rendering always involves a trade-off between image quality and speed.

To revert to the original screen mode in DOS type:

```
SET BRENDER_DOS_GFX=MCGA,W:320,H:200,B:8
```

followed by Return. Experiment with the screen modes available to you. In general, the lower the resolution the faster your animation will run. You may specify simply MCGA or VESA and BRender will select appropriate arguments for `W`, `H` and `B`.

`DOSGfxBegin` returns a pointer to a pixel map that references the screen. This pointer is assigned here to `screen_buffer`. This pixel map could be a hardware screen buffer mapped directly to the screen or a pixel map set up by BRender to simulate a hardware screen buffer. Either way, from the application programmer's point of view, the contents of this buffer are displayed on screen.

When rendering in 8-bit indexed colour mode, a colour palette must be loaded into the hardware palette (or CLUT). `BrPixelmapLoad` loads a pixel map from a specified file into memory. *`DOSfxPaletteSet`* copies the contents of a pixel map to the hardware palette. The initialisation code listed above loads the colour palette supplied with BRender, `std.pal`, into memory, then copies it to the hardware palette.

`BrZbBegin` is used to select and initialise the Z-Buffer renderer. `BrZbEnd` closes it down. Remember that a BRender scene description is designed to be independent of a particular rendering environment. This means that porting a BRender program between platforms that support different rendering engines can be accomplished by changing a couple of lines of code – in this case `BrZbBegin` and `BrZbEnd`. Refer to your installation guide for details of the rendering options available.

The renderer needs to know the 'type' of buffer it will be rendering into – whether it contains indexed or true colour information and the number of bits per pixel. It also needs to know the depth of the Z-Buffer. This information is passed to `BrZbBegin`.

`BRTUTOR1.C` contains additional initialisation code shown in the program extract below.

```
BrBegin();

   screen_buffer = DOSGfxBegin(NULL);
   palette = BrPixelmapLoad("std.pal");
   if(palette)
      DOSGfxPaletteSet(palette);

   BrZbBegin(screen_buffer->type,BR_PMT_DEPTH_16);

   /*
    * Allocate back buffer and depth buffer
    */
   back_buffer = BrPixelmapMatch(screen_buffer,
                 BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,
                 BR_PMMATCH_DEPTH_16);
```

⋮ ⋮ ⋮

```
BrZbEnd();
DOSGfxEnd();
BrEnd();
```

Before initialisation is complete, we must create a back buffer in the image of the screen buffer to facilitate double-buffering. Since we're using the Z-Buffer renderer, we also need a depth buffer of the same width and height. `BrPixelmapMatch` takes care of this. `BR_PIXELMAP_OFFSCREEN` tells it to replicate the screen buffer. `BR_PMMATCH_DEPTH_16` tells it to create a 16-bit depth, or Z-, buffer, of the same width and height as the screen buffer.

The initialisation procedure introduced above could be used as a general purpose template for initialising BRender programs. You may well find yourself writing many BRender programs without making significant alterations to this sequence of function calls. Indeed, you are unlikely to alter any of this code unless you are selecting a different renderer or changing the graphics screen mode. Take time to familiarise yourself with the function calls and data structures introduced above. They are the foundations upon which your conceptual model of BRender will be built.

Setting up the World Database

Having initialised BRender, we must build the world database. The information contained here defines the shape, colour and texture of the models in a scene, as well as how they are related to one another in the world hierarchy. It also describes how the scene is lit and how much of it is initially 'visible'.

Remember we build scenes using actors – camera actors, light actors, model actors. Another actor type, the *None* actor, is used as a reference point for scene building – it is used to structure the world database. A none actor is usually found at the root of an actor tree hierarchy.

`BrActorAllocate` allocates, or assigns, a new actor to the world database. `BrActorAdd` declares one allocated actor to be the child of another (and returns a pointer to the child actor). This is how actor hierarchies are constructed.

`BRTUTOR1.C` starts with a none (or 'root') actor called 'world',

```
world = BrActorAllocate(BR_ACTOR_NONE,NULL);
```

It adds a model actor (the default cube) as a child of 'world' and calls it 'cube',

```
cube = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
```

then adds a camera actor, also a child of 'world', and calls it 'observer',

```
observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,
              NULL));
```

before adding and 'turning on' the default light source,

```
light = BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL));
BrLightEnable(light);
```

A simple renderable world tree has been created:

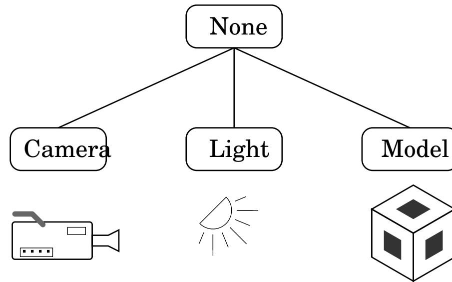


Figure 32 Four-actor world tree

The world database now consists of –

- a matt-grey unit cube positioned at the origin
- a camera or view position, also located at the origin
- an infinitely distant direct light source positioned directly behind the view position.

This is the default condition depicted in Figure 31.

In order to make the cube visible we move, or translate, the camera position 5 units backwards along the positive z-axis:

```
observer->t.type = BR_TRANSFORM_MATRIX34;
BrMatrix34Translate(&observer->t.t.mat, BR_SCALAR(0.0),
              BR_SCALAR(0.0),
                                          BR_SCALAR(5.0));
```

Note that the first line of the above code could have been omitted, as `BR_TRANSFORM_MATRIX34` is the default transformation type.

Note that BRender uses a custom data type, `br_scalar`, to represent scalar values – it doesn't use the standard data types int and float . `BR_SCALAR()`, implemented as a macro, converts constants to the `br_scalar` data type.

We then rotate the cube 30° around the y-axis (to accentuate its shape):

```
cube->t.type = BR_TRANSFORM_MATRIX34;
BrMatrix34RotateY(&cube->t.t.mat,BR_ANGLE_DEG(30));
```

`BR_ANGLE_DEG()` converts between degrees and BRender's angle representation `br_angle`. Before moving an actor, a transformation type must be defined. Refer to your technical reference manual for a list of the transformation types available.

Animation Loop

The code that generates the animation (in this case 360 frames), is listed below.

The back buffer is initialised to the background pixel value. `BrPixelmapFill(back_buffer,0)` fills the back buffer with 0's, setting the background pixel colour to black.

The Z-Buffer is then initialised to the largest representable z-value. This is 0xFFFF for a 16-bit buffer. 0xFFFFFFFF is passed to accommodate Z-Buffer depths up to 32 bits.

```
for(i=0; i < 360; i++) {
   BrPixelmapFill(back_buffer,0);
   BrPixelmapFill(depth_buffer,0xFFFFFFFF);
   BrZbSceneRender(world,observer,back_buffer,depth_buffer);
   BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
   BrMatrix34PostRotateX(&cube->t.t.mat,BR_ANGLE_DEG(2.0));
}
```

`BrZbSceneRender` renders the scene into `back_buffer`. `BrPixelmapDoubleBuffer` then 'swaps' `screen_buffer` and `back_buffer` (see section on 'Double Buffering' in Chapter 1) so the newly rendered image is displayed.

For each iteration of the loop, cube is rotated 2° about the x-axis.

More About Background Colour

In 8-bit colour mode, 256 colours are available at any time. Which 256 colours are available is determined by the contents of the hardware palette (or CLUT). In this case, the palette supplied with BRender (`std.pal`) has been loaded into the CLUT.

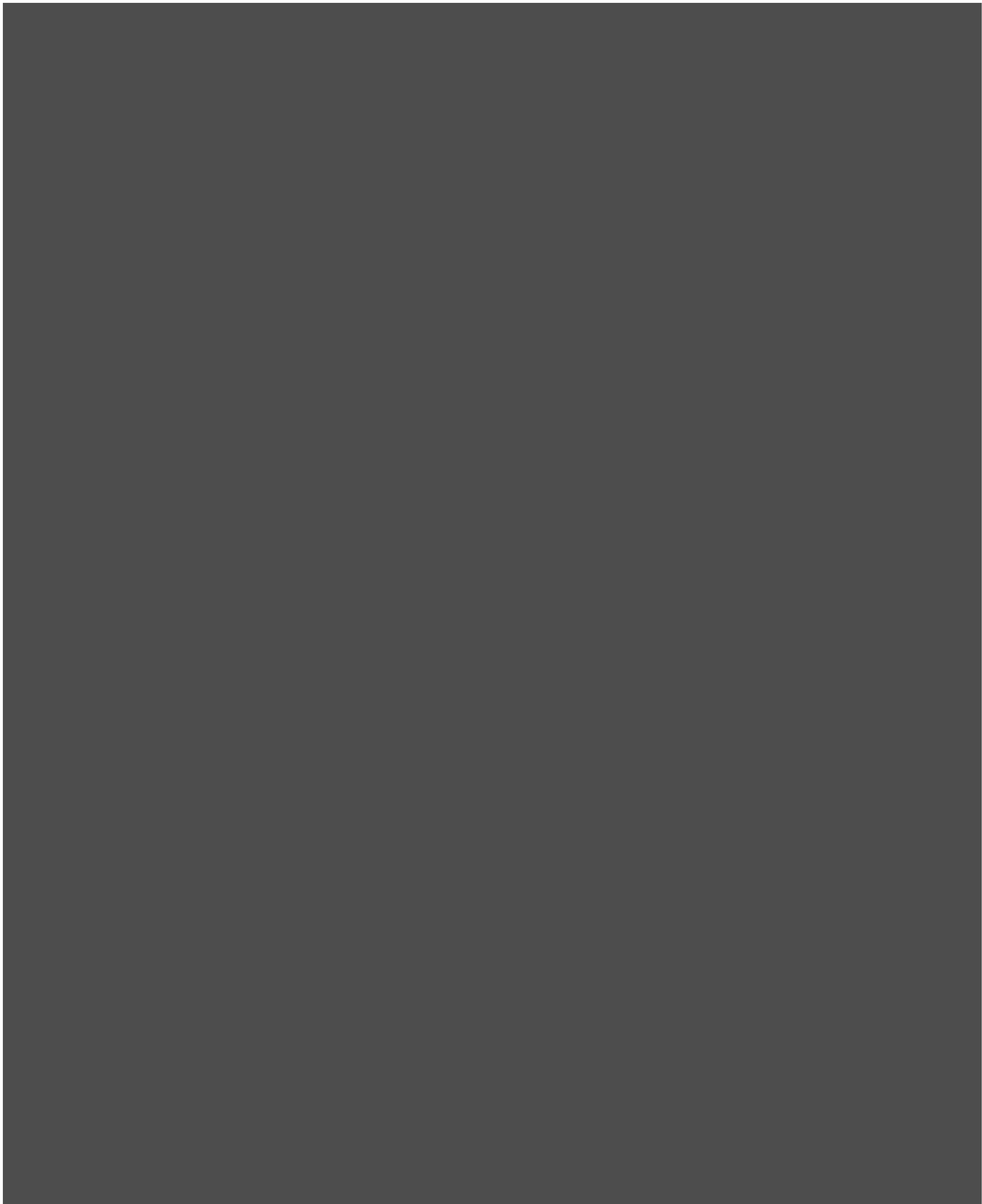
The 256 colours in `std.pal` are divided into seven ranges, or 'colour ramps'. The first 64 colours represent shades of grey ranging from very dark grey (black) to very light grey (white). The following colours are 32-element ramps for six colours as shown below.

Colour Ramps in std.pal	
Range	**Colour**
0–63	Grey
64–95	Blue
96–127	Green
128–159	Cyan
160–191	Red
192–223	Magenta
224–255	Yellow

`BrPixelmapFill(back_buffer,0)` sets the background to black. The background colour can easily be changed. In `std.pal`, colour 88 is a deep blue. Substitute 88 (or 0x55) for 0 in the above function call to change the background colour to blue. If you would prefer a red background, try 180 (0xB4).

Note that in 8-bit colour mode, only the least significant 8 bits of information in the colour argument are used by `BrPixelmapFill` to determine the colour – all others are ignored. So 0xFF is taken as white, as is 0xFFFFFFFF or 0x123FF or 255. 0xFFFFFF00, 0 and 256 are all black.

In true-colour mode, either 15-, 16-, 24- or 32- bit colour information is assumed (refer to your installation guide for details of which colour modes are supported by your version of BRender). In 15-bit colour mode, five bits are used for each of the R, G and B components of a pixel's colour – the least significant five bits for blue, the most significant five bits for red. 0x1F therefore represents fully saturated blue. Try setting the background to blue using this value. Remember to select 15-bit colour mode. If you are running BRender under DOS, for instance, you could substitute an appropriate string (e.g. `VESA,W:320,H:200,B:15`) in place of NULL in *`DOSgfxBegin(NULL)`* or set the *`BRENDER_DOS_GFX`* environment variable. Use 0x1F<<5 to change the background to green, or 0x1F<<10 to change it to red.



3 Positioning Actors

In chapter two, you were introduced to your first BRender program. You now know how to display a revolving grey cube! No doubt you are anxious to expand your BRender repertoire. You will be pleased to know that this will not be as difficult as you might have expected. You have already learnt the fundamentals. By the time you have worked through the following short chapters, your ability to produce complex 3D applications will be limited only by your imagination and the extent of your C programming skills. This chapter is devoted to 3D transformations – placing and moving models in a 3D scene.

The fundamental transformations: Translation, Rotation and Scaling, were introduced in Chapter 1. These can be represented in a number of different ways. In fact six different transformation types are available in BRender.

The simplest is the **Identity Transform**, which specifies the *identity matrix* as the transformation matrix. The identity matrix does not alter an actor's shape or position. It makes an actor effectively share its parent's co-ordinate space. The **Translation** transformation is used to implement a translation (useful when no rotation or scaling are involved in a transformation). The Translation transformation type is represented in BRender as a vector that is added to an actor's co-ordinates. **Euler**, **Look-Up** and **Quaternion** transformations all specify a translation and an orientation.

The Euler transform applies three separate rotations in turn, in a specified order. Euler angles are the generalisation of the pitch-yaw-and-roll of flight simulators.

A look-up transformation is a convenient method of making an actor, often a camera actor, point towards a particular position. A *look* vector is used to specify the view direction, and an *up* vector to define the orientation.

The unit quaternion transform represents a rotation about an arbitrary vector. Quaternions are used in computer animations for interpolating the positions of tumbling bodies between key frames.

The most general representation is the **Matrix Transform** (`BR_TRANSFORM_MATRIX34`). All the transformation types discussed above can be implemented using matrices (refer to Chapter 1 of this guide and to `br_matrix34` in your technical reference manual for details of how matrix transforms are performed). You will recall from Chapter 1 that matrices can be concatenated, allowing a complex sequence of transformations to be pre-processed into a single transformation matrix. Not surprisingly then, computer graphics systems normally represent transformations internally using matrices.

The transformations implemented in our tutorial programs use the `BR_TRANSFORM_MATRIX34` representation. Feel free to experiment with the other available implementations. Refer to your technical reference manual for details of relevant structures and data types.

Your Second Program

The remainder of this chapter is devoted to demonstrating how to implement translation, rotation, and scaling transformations. BRTUTOR1.C (remember the revolving grey cube) is used as a template. By the end of the chapter, BRTUTOR1.C will have evolved into BRTUTOR2.C, a copy of which is included on your Tutorial Programs disk. You may wish to monitor your progress by editing a copy of BRTUTOR1.C as we go along. This will allow you to compile and run the program at appropriate stages in its evolution. What better way to familiarise yourself with BRender functions than to experiment with them as they are introduced?

When complete, this program will display three models – a box, a sphere and a torus. The actor declarations are thus,

```
br_actor *world, *observer, *box, *sphere, *torus;
```

The camera position is moved 10 units backwards along the positive z-axis, instead of five as previously (the cube will appear smaller). The size of the view volume is increased by moving the yon-plane further away from the view position.

```
observer->t.type = BR_TRANSFORM_MATRIX34;
BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),
              BR_SCALAR(0.0),
                                                BR_SCALAR(10.0));
camera_data = (br_camera *)observer->type_data;
camera_data->you_z = BR_SCALAR(50);
```

Note the explicit cast conversion. Some actors, including camera actors, may have dynamically allocated type-specific data attached. The revised view volume is depicted in Figure 33.

BRTUTOR2.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a scene containing a Box, a Sphere and a Torus
 */
#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"
int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
   br_actor *world, *observer, *light, *box, *sphere, *torus;
   int i;
   br_camera *camera_data;

/********* Initialise BRender and Graphics Hardware ************/
BrBegin();
/*
 * Initialise screen buffer and set up CLUT (ignored in true colour)
 */

                       :          :          :

BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);
back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database *********************/
/*
 * Start with None actor at root of actor tree and call it 'world'
 */
world = BrActorAllocate(BR_ACTOR_NONE,NULL);

/*
 * Add, and enable, the default light source
 */
light = BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL));
BrLightEnable(light);
/*
 * Load and Position Camera
 */
observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
observer->t.type = BR_TRANSFORM_MATRIX34;
BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                      BR_SCALAR(10.0));
camera_data = (br_camera *)observer->type_data;
camera_data->you_z = BR_SCALAR(50);
/*
 * Load and Position Box Model
 */
box = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
box->t.type = BR_TRANSFORM_MATRIX34;
```

```
BrMatrix34RotateY(&box->t.t.mat,BR_ANGLE_DEG(30));
BrMatrix34PostTranslate(&box->t.t.mat,BR_SCALAR(-2.5),BR_SCALAR(0.0),
                                                      BR_SCALAR(0.0));
BrMatrix34PreScale(&box->t.t.mat,BR_SCALAR(2.0),BR_SCALAR(1.0),
                 BR_SCALAR(1.0));
/*
 * Load and Position Sphere Model
 */
sphere = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
sphere->model = BrModelLoad("sph32.dat");
BrModelAdd(sphere->model);
sphere->t.type = BR_TRANSFORM_MATRIX34;
BrMatrix34Translate(&sphere->t.t.mat,BR_SCALAR(2.0),BR_SCALAR(0.0),
                                                    BR_SCALAR(0.0));
/*
 * Load and Position Torus Model
 */
torus = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
torus->model = BrModelLoad("torus.dat");
BrModelAdd(torus->model);
torus->t.type = BR_TRANSFORM_MATRIX34;
BrMatrix34Translate(&torus->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                   BR_SCALAR(3.0));

/********************** Animation Loop ***********************/
   for(i=0; i < 360; i++) {
         BrPixelmapFill(back_buffer,0);
         BrPixelmapFill(depth_buffer,0xFFFFFFFF);
         BrZbSceneRender(world,observer,back_buffer,depth_buffer);
         BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
         BrMatrix34PostRotateX(&box->t.t.mat,BR_ANGLE_DEG(2.0));
         BrMatrix34PreRotateZ(&torus->t.t.mat,BR_ANGLE_DEG(4.0));
         BrMatrix34PreRotateY(&torus->t.t.mat,BR_ANGLE_DEG(-6.0));
         BrMatrix34PreRotateX(&torus->t.t.mat,BR_ANGLE_DEG(2.0));
         BrMatrix34PostRotateX(&torus->t.t.mat,BR_ANGLE_DEG(1.0));
         BrMatrix34PostRotateY(&sphere->t.t.mat,BR_ANGLE_DEG(0.8));
   }
/*
 * Close down
 */
BrPixelmapFree(depth_buffer);
BrPixelmapFree(back_buffer);
BrZbEnd();
                        :          :          :

   BrEnd();
   return 0;
}
```
BRTUTOR2.C

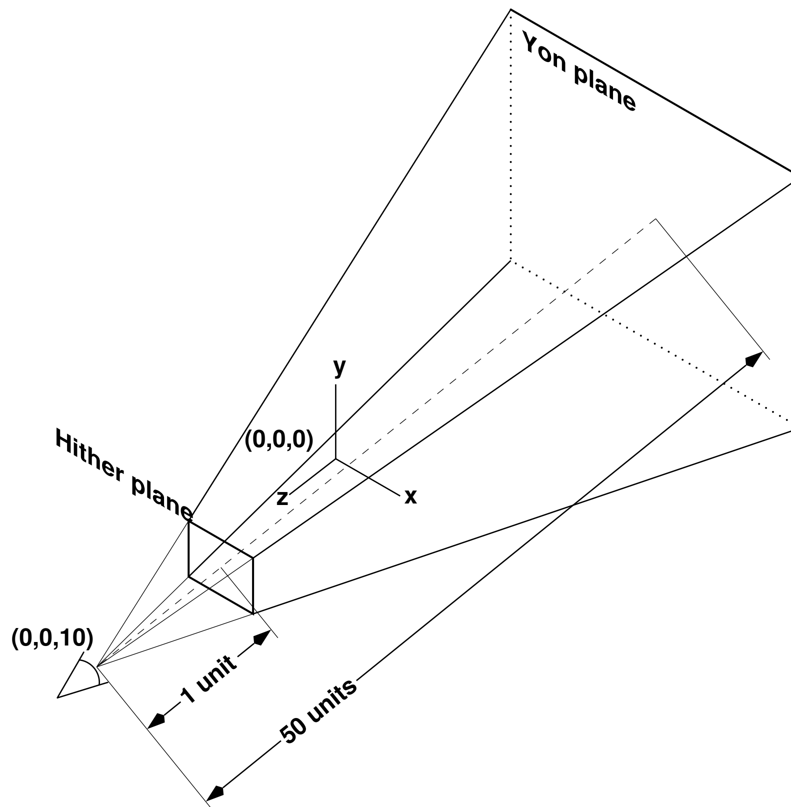


Figure 33 Revised view position and view volume

The following code loads and positions the cube or 'box':

```
box = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
/*Define box's transformation*/
box->t.type = BR_TRANSFORM_MATRIX34;
BrMatrix34RotateY(&box->t.t.mat,BR_ANGLE_DEG(30));
BrMatrix34PostTranslate(&box->t.t.mat,BR_SCALAR(-2.5),
             BR_SCALAR(1.0),
                                                BR_SCALAR(1
             .0));
```

The changes made to BRTUTOR1.C so far are highlighted in bold. If you were to implement these changes and re-compile the program, a revolving grey cube would be displayed close to the left edge of the screen.

Transformation Function

Remember that matrix multiplication is non-commutative. The order in which transformations are applied is critical. Three families of calls are provided for implementing transformations. The standard implementation,

```
BrMatrix34Translate()
BrMatrix34Rotate()
BrMatrix34Scale()
```

sets a target matrix to a matrix representing a translation, rotation or scaling.

The 'Pre-' family of transformation function calls,

```
BrMatrix34PreTranslate()
BrMatrix34PreRotate()
BrMatrix34PreScale()
```

pre-multiplies a target matrix by a matrix representing a translation, rotation or scaling, and puts the result back in the target matrix.

The 'Post-' family of transformation function calls,

```
BrMatrix34PostTranslate()
BrMatrix34PostRotate()
BrMatrix34PostScale()
```

post-multiplies a specified matrix by a matrix representing a translation, rotation or scaling, and puts the result back in the specified matrix.

Use the standard calls to initialise a matrix to represent a specified transformation.

Given a matrix representing a transformation, or series of transformations, use the 'Pre-' family of calls to add an additional transformation **at the beginning**.

Given a target matrix representing a transformation, or series of transformations, use the 'Post-' family of calls to append a transformation **at the end**.

Consider the 'box' transformation above – a rotation followed by a translation. If we neglected to specify a 'Post-' translation,

```
BrMatrix34RotateY(&box->t.t.mat,BR_ANGLE_DEG(30));
BrMatrix34Translate(&box->t.t.mat,BR_SCALAR(-2.5),BR_SCALAR(0.0),
                                                   BR_SCALAR(0.0))
                ;
```

the rotation would not be implemented. We would simply get a translation in x, as the rotation matrix has been replaced by the translation matrix.

Consider what would happen if we reversed the order,

```
BrMatrix34Translate(&box->t.t.mat,BR_SCALAR(-2.5),BR_SCALAR(0.0),
                                                   BR_SCALAR(0.0))
                ;
```

```
BrMatrix34PostRotateY(&box->t.t.mat,BR_ANGLE_DEG(30));
```

to give a translation followed by a rotation. The box would be translated in x, before being rotated around the y-axis.

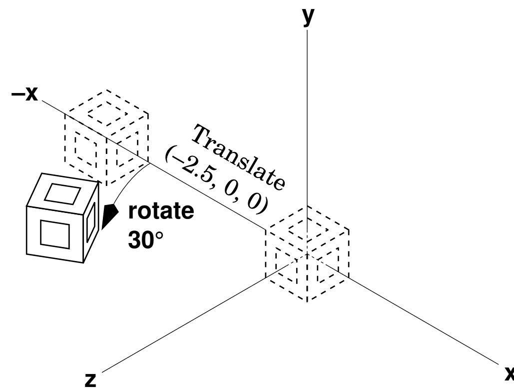


Figure 34 Rotation follows Translation

If this code were substituted in BRTUTOR2.C, the resulting animated sequence would be very different from that generated by the original program. Note that the following code would produce exactly the same result,

```
BrMatrix34RotateY(&box->t.t.mat,BR_ANGLE_DEG(30));
BrMatrix34PreTranslate(&box->t.t.mat,BR_SCALAR(-2.5),BR_SCALAR
            (0.0),
                                                        BR_SCALAR(0.
            0));
```

a translation followed by a rotation. Try out these and other variations for yourself. A little experimentation at this stage could pay dividends later.

Let's apply a scaling transformation to change the shape of the box from a cube to a rectangle.

```
BrMatrix34RotateY(&box->t.t.mat,BR_ANGLE_DEG(30));
BrMatrix34PostTranslate(&box->t.t.mat,BR_SCALAR(-2.5),BR_SCALAR
            (0.0),
                                                        BR_SCALAR(0
            .0));
BrMatrix34PreScale(&box->t.t.mat,BR_SCALAR(2.0),BR_SCALAR(1.0),
                                        BR_SCALAR(1.0));
```

We called the 'Pre-' scaling function because we wanted the scaling transformation to be applied to the box before it was positioned in the scene. The above scaling multiplies the x-components of the box's vertices by 2.

Note that, had we specified 'Post-' scaling, the *translation* factor would have been scaled along with the vertices. In this case, the previously specified translation factor (–2.5) would have been doubled (to –5). You are again invited to experiment.

Adding the Sphere and Torus

At this stage our program displays a scaled 'box' near the left edge of the screen, revolving around the x-axis. Let's introduce the other models, starting with the sphere.

```
/*Load and Position Sphere Model*/
sphere = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
sphere->model = BrModelLoad("sph32.dat");
BrModelAdd(sphere->model);
sphere->t.type = BR_TRANSFORM_MATRIX34;
BrMatrix34Translate(&sphere->t.t.mat,BR_SCALAR(2.0),
              BR_SCALAR(0.0),
                                                         BR_SCALAR(0.0
              ));
```

The first line of this code adds a new model actor to the database, as a child of 'world', and calls it 'sphere'. If you want to display a model other than the default cube, you must first load it from a model data file,

```
sphere->model = BrModelLoad("sph32.dat");
```

then add it to the registry,

```
BrModelAdd(sphere->model);
```

Remember that actors, models, materials and pixel maps must be added to the registry before they can be used. Model descriptions are stored in `.dat` files. A number of sample model files are contained on your Tutorial Programs disk (all with `.dat` extensions). You can create your own models in 3D Studio, then convert them to BRender `.dat` format using the supplied utilities 3DS2BR and GEOCONV. 3DS2BR converts 3D Studio models stored as `.3ds` binary files while GEOCONV handles ASCII (`.asc`) files.

The 'sphere' is translated in x to position it closer to the right hand edge of the screen. Note that the line

```
sphere->t.type = BR_TRANSFORM_MATRIX34;
```

could have been omitted as `BR_TRANSFORM_MATRIX34` is the default transformation type.

A 'torus' model is added and translated in z to bring it closer to the view position.

```
/*Load and Position Torus Model*/
torus = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
torus->model = BrModelLoad("torus.dat");
```

```
   BrModelAdd(torus->model);
   torus->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34Translate(&torus->t.t.mat,BR_SCALAR(0.0),
               BR_SCALAR(0.0),
                                                  BR_SCALAR(3.0)
               );
```

If you add the above code and re-compile the program, a torus will be displayed in the centre of the screen – flanked on one side by a revolving rectangle and on the other by a sphere.

The Animation Loop

Let's complete the animation. The revised animation loop is given below, with additional code highlighted:

```
for(i=0; i < 360; i++) {

   BrPixelmapFill(back_buffer,0);
   BrPixelmapFill(depth_buffer,0xFFFFFFFF);
   BrZbSceneRender(world,observer,back_buffer,depth_buffer);
   BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
   BrMatrix34PostRotateX(&box->t.t.mat,BR_ANGLE_DEG(2.0));
   BrMatrix34PreRotateZ(&torus->t.t.mat,BR_ANGLE_DEG(4.0));
   BrMatrix34PreRotateY(&torus->t.t.mat,BR_ANGLE_DEG(-6.0));
   BrMatrix34PreRotateX(&torus->t.t.mat,BR_ANGLE_DEG(2.0));
   BrMatrix34PostRotateX(&torus->t.t.mat,BR_ANGLE_DEG(1.0));
   BrMatrix34PostRotateY(&sphere->t.t.mat,BR_ANGLE_DEG(0.8));
}
```

Lets consider the torus. For each iteration of the loop, the specified PreRotations are added at the beginning of the transformation series (so they are implemented before the translation), and the PostRotation is added at the end. The concatenated matrix thus represents rotations in x, y and z (4°, –6°, and 2° respectively), followed by the compound transformation defined by the previous concatenation which is then followed by a 1° rotation in x.

The sphere is PostRotated in y. Since it had been previously translated +2 units in x, it circles the y-axis in a broad sweep.

`BRTUTOR1.C` has evolved into `BRTUTOR2.C`.

3

POSITIONING ACTORS

4 Actor Hierarchies

The animations we have created up to now have described uncomplicated scenes containing basic geometric shapes, each individually specified and positioned. Now, consider how much more difficult it would be to describe a complex 3D animation containing spatially interdependent models – say a representation of our solar system. How would you describe the path of the Moon orbiting the Earth, which in turn orbits the Sun? Or the motion of the hands of a clock located inside a space ship hurtling through space?

Complex models or scenes are constructed as actor hierarchies. A hierarchy allows you to think of an actor in terms of its spatial relationship to another, parent, actor. An actor hierarchy can can then be broken down into a series of fairly simple spatial relationships, or transformations, between actors.

A hierarchical model of part of our solar system is represented in Figure 35.

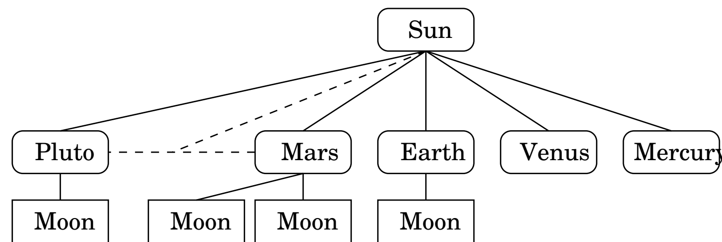


Figure 35 Hierarchical model of solar system

The motion of the Moon can be described relative to the position of the Earth, and that of the Earth relative to the position of the Sun. If we can keep track of the motion of the Moon relative to the Earth and of the motion of the Earth relative to the Sun, then the Moon's path through space can be computed. The position of the Moon at any time can be described by the concatenation of a series of matrices representing relative displacements, or transformations.

`BRTUTOR3.C` implements a simple three-layer hierarchical tree structure. Compile and run it to create a 'satellite' orbiting a 'moon' orbiting a 'planet' animation.

Note that 'moon' is defined as a child of 'planet',

```
moon = BrActorAdd(planet,BrActorAllocate(BR_ACTOR_MODEL,NULL));
```

and 'satellite' is defined as a child of 'moon',

```
satellite = BrActorAdd(moon,BrActorAllocate(BR_ACTOR_MODEL,
                NULL));
```

The actor hierarchy depicted in Figure 36 results.

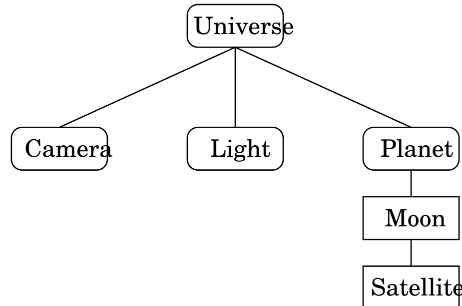


Figure 36 Planet, Moon, Satellite actor hierarchy

All three models are created using the same model data file, `sph16.dat`, in which a number of triangular polygons are combined to approximate the surface of a sphere. A more accurate approximation could be obtained using `sph32.dat` or `sph4096.dat`. Substitute `sph8.dat` for a cruder approximation. As always, you'll find there's a trade-off between speed and image quality.

The 'universe' (or root actor) co-ordinate system serves as an absolute frame of reference within the application and could be thought of as 'World Space' (or the application coordinate system).

The 'planet' is transformed into its parent co-ordinate system ('World Space'). The identity transform is assumed, since no transformation has been explicitly defined. Note that the camera has been translated 6 units along the positive z-axis to ensure that the 'planet' is visible.

The 'moon' is uniformly scaled before being translated +2 units in z, **in the planet's co-ordinate system**. Since the camera has been translated +6 units in z the net result, as far as the viewer is concerned, is to position the 'moon' between the view position and the planet.

BRTUTOR3.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display Planet, Moon Satellite Animation.
 */
#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"
int main(int argc, char **argv)
{
  br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
  br_actor *universe, *observer, *light, *planet, *moon, *satellite;
  int i;
  br_camera *camera_data;

/************* Initialise BRender and Graphics Hardware ************/
  BrBegin();
  /*
   * Initialise screen buffer and set up CLUT (ignored in true colour)
   */

                       :         :         :

  BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);

  back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
  depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database *****************/
  /*
   * Load Root Actor
   */
  universe = BrActorAllocate(BR_ACTOR_NONE,NULL);
  /*
   * Load and Enable Default Light Source
   */
  light = BrActorAdd(universe,BrActorAllocate(BR_ACTOR_LIGHT,NULL));
  BrLightEnable(light);
  /*
   * Load and Position Camera
   */
  observer = BrActorAdd(universe,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
  observer->t.type = BR_TRANSFORM_MATRIX34;
  BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                        BR_SCALAR(6.0));
  camera_data = (br_camera *)observer->type_data;
  camera_data->yon_z = BR_SCALAR(50);
  /*
   * Load Planet Model
   */
  planet = BrActorAdd(universe,BrActorAllocate(BR_ACTOR_MODEL,NULL));
  planet->model = BrModelLoad("sph16.dat");
```

```
  BrModelAdd(planet->model);
  /*
   * Load and Position Moon Model
   */
  moon = BrActorAdd(planet,BrActorAllocate(BR_ACTOR_MODEL,NULL));
  moon->model = BrModelLoad("sph8.dat");
  BrModelAdd(moon->model);
  moon->t.type = BR_TRANSFORM_MATRIX34;
  BrMatrix34Scale(&moon->t.t.mat,BR_SCALAR(0.5),BR_SCALAR(0.5),
                                                BR_SCALAR(0.5));
  BrMatrix34PostTranslate(&moon->t.t.mat,BR_SCALAR(0.0),BR_SCALAR (0.0),
                                                      BR_SCALAR(2.0));
  /*
   * Load and Position Satellite Model
   */
  satellite = BrActorAdd(moon,BrActorAllocate(BR_ACTOR_MODEL,NULL));
  satellite->model = BrModelLoad("sph8.dat");
  BrModelAdd(satellite->model);
  satellite->t.type = BR_TRANSFORM_MATRIX34;
  BrMatrix34Scale(&satellite->t.t.mat,BR_SCALAR(0.25),BR_SCALAR (0.25),
                                                    BR_SCALAR(0.25));
  BrMatrix34PostTranslate(&satellite->t.t.mat,BR_SCALAR(1.5),BR_SCALAR(0.0),
                                                           BR_SCALAR(0.0));

/********************* Animation Loop **********************/
  for(i=0; i < 1000; i++) {
       BrPixelmapFill(back_buffer,0);
       BrPixelmapFill(depth_buffer,0xFFFFFFFF);
       BrZbSceneRender(universe,observer,back_buffer,depth_buffer);
       BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
       BrMatrix34PreRotateY(&planet->t.t.mat,BR_ANGLE_DEG(1.0));
       BrMatrix34PreRotateY(&satellite->t.t.mat,BR_ANGLE_DEG(4.0));
       BrMatrix34PreRotateZ(&moon->t.t.mat,BR_ANGLE_DEG(1.5));
       BrMatrix34PostRotateZ(&satellite->t.t.mat,BR_ANGLE_DEG(-2.5));
       BrMatrix34PostRotateY(&moon->t.t.mat,BR_ANGLE_DEG(-2.0));
  }
  /*
   * Close down
   */
  BrPixelmapFree(depth_buffer);
  BrPixelmapFree(back_buffer);
  BrZbEnd();
                          :          :          :
  BrEnd();
  return 0;
}
```
BRTUTOR3.C

The satellite is also uniformly scaled before being translated +1.5 units in x, **in the moon's co-ordinate system**. From the viewer's perspective, (at x=0, y=0, z=6) in 'World Space', the satellite is initially centred at (1.5, 0, 2). The first frame rendered is depicted below.

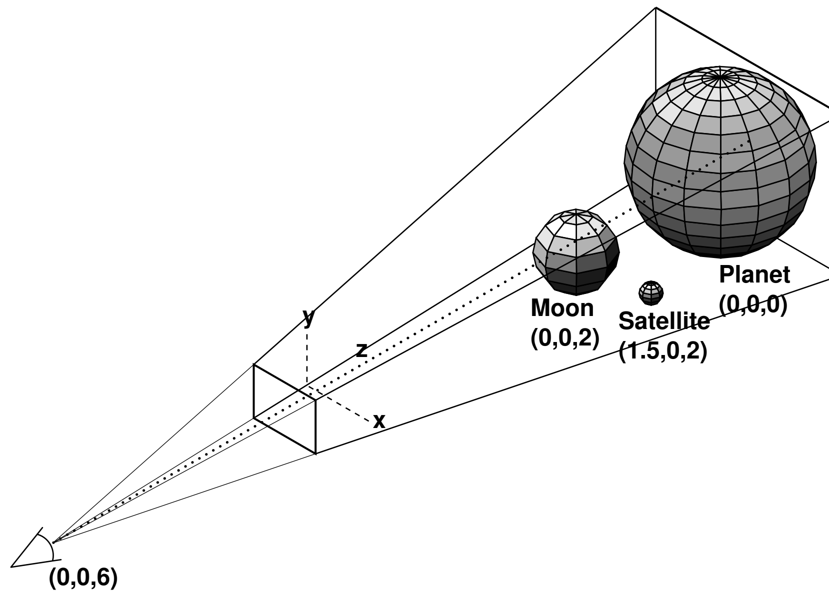


Figure 37 First frame

In the animation loop, PreRotate functions are used to set the planet and satellite rotating about their own y-axes, and the moon revolving around its own z-axis.

```
BrMatrix34PreRotateY(&planet->t.t.mat,BR_ANGLE_DEG(1.0));
BrMatrix34PreRotateY(&satellite->t.t.mat,BR_ANGLE_DEG(4.0));
BrMatrix34PreRotateZ(&moon->t.t.mat,BR_ANGLE_DEG(1.5));
```

The satellite is rotated to trace an orbit around the moon's z-axis. Note that the satellite's transformation is applied in the moon's co-ordinate space, not in 'World Space'.

```
BrMatrix34PostRotateZ(&satellite->t.t.mat,BR_ANGLE_DEG(-2.5));
```

The moon (and 'attached' satellite) is rotated to trace an orbit around the planet's y-axis.

```
BrMatrix34PostRotateY(&moon->t.t.mat,BR_ANGLE_DEG(-2.0));
```

To place either the moon or the satellite in geostationary orbit, simply delete the appropriate rotate command.

Feel free to experiment with this program and with `BRTUTOR4.C`. `BRTUTOR4.C` creates another planet-satellite animation using hierarchical models.

5 Adding Colour

A 'material' may be explicitly assigned to a model actor or to each face on a model. A material describes the appearance of a surface – its colour and texture, whether it's shiny or dull, smooth or rough, etc.

For each face on a model, BRender looks for an associated material. If none has been specified (or the associated material is not found in the registry), the model actor's material is assumed. If a material has not been assigned to the model actor, it inherits its parent's material. If the parent actor, or a previous ancestor, has not been assigned a material, a flat-shaded grey material is used by default.

The default material has been used with all the models you have displayed so far.

Let's design a material and apply it to the revolving grey cube of `BRTUTOR1.C`.

The information describing a material is stored in a `br_material` data structure. Refer to your technical reference manual for details of `br_material`.

Care should be taken when initialising data structures statically, as only public members of BRender data structures are documented in the technical reference manual.

A custom function, `BrFmtScriptMaterialLoad`, is provided for loading material descriptions from a script file. A material script file is a text file as in the following example:

sample material script file

```
# Comment
#
# Fields may be specified in any order, or omitted
# Where fields are omitted, sensible defaults will be supplied
# Extra white spaces are ignored

material = [
               identifier = "block";
               flags = [light,prelit,smooth,environment,
                  environment_local,perspective,decal,
                  always_visible,two-sided,force_z_0];
               colour = [0,0,255];
               ambient = 0.05;
               diffuse = 0.55;
               specular = 0.4;
               power = 20;
               map_transform = [[1,0], [0,1], [0,0]];
               index_base = 0;
               index_range = 0;
               colour_map = "brick"
               index_shade = "shade.tab"
            ];
```

The fields in the script file relate directly to `br_material` fields. Refer to your technical reference manual for futher details. Note that all material flags would never be set at the same time, as they are in the above example. They are shown here to illustrate how material flags are specified in script files.

The script file used to determine the appearance of the revolving cube in `BRTUTOR5.C` is called `cube.mat` and is included on your Tutorial Programs disk.

cube.mat

```
# This material script file describes the appearance
# of the material "BLUE MATERIAL"

material = [
              identifer = "BLUE MATERIAL";
              colour = [0,0,255];
              ambient = 0.05;
              diffuse = 0.55;
              specular = 0.4;
              power = 20;
              flags = [light,smooth];
           ];
```

A script file may contain a number of material descriptions. The *identifier* field allows you to specify a name by which each loaded material is subsequently known. When you want to assign a particular material to a model, you simply instruct BRender to find it by name before completing the assignation.

The material colour is pure blue (both red and green components are 0).

There are a number of material properties, besides colour, that determine how a surface will appear under given lighting conditions – whether it will appear rough or smooth, shiny or dull etc.

- *Ambient* light is the general 'atmospheric' light that is not associated with a specific light source. In a real scene it is background light generated by reflection from other surfaces, for example a bright (but cloudy) summer day would have a large amount of ambient light, whereas a moonlit scene would have practically none. In 3D graphics, ambient light illuminates an object with a uniform light which does not come from any specific direction, but rather all directions at once. As the ambient light is not dependent on the presence of light sources, an object with a high ambient lighting value in a dark unlit scene will appear to glow.
- *Diffuse* light is the light from a directional source which has been reflected equally in all directions from the surface of an object. The apparent brightness of a surface is independent of the position of the observer as it depends only on the angle between the surface of the object and the direction of the incident light. The closer the object surface and incident light are to being perpendicular, the brighter the diffuse lighting will be. An object with faces at different angles will

reflect varying intensities of light and hence have shading effects.

- *Specular* light is the effect that produces the highlights that can be seen on a shiny surface, for example as seen on a polished apple illuminated with a bright white light. Shiny metals or plastics have a high specular component but carpet or chalk have none. If the observer moves so does the position of the highlight on the object. The highlight is effectively the reflection of the light source in the surface of the object. Specular lighting depends on the angle between the line of sight of the observer and the direction of the incident light.

The ambient, diffuse and specular fields are used to specify, respectively, the `ka`, `kd` and `ks` members of the `br_material` data structure. Each ranges between 0 and 1, and the three should sum to 1. An additional field, power, determines how sharp highlights will appear. A more detailed discussion of material properties can be found in your technical reference manual (see `br_material`).

The most commonly specified flags are `light` and `smooth`. Light specifies that lighting effects should be taken into account when rendering. Smooth specifies Gouraud shading. The polygons that make up the surface of a model can be drawn with a single colour (flat shading) or with many colours (smooth or Gouraud shading). With flat shading, the colour of a single vertex is calculated and duplicated across the entire polygon. With smooth shading, the colour at each vertex is computed and colour values for the interior of the polygon interpolated linearly between the vertex colours. In our present example, if we hadn't specified smooth shading, each face on the cube would have been drawn using a single colour. With smooth shading a more realistic effect is achieved through interpolation.

For a demonstration of flat shading, simply remove 'smooth' from the flag field in the text file `cube.dat` and run the program again. This is the major advantage of using script files to define material properties; any of these properties can be changed, and the result viewed, without having to re-compile the program. Simply edit the script file as necessary. This makes it easy to experiment with different colour values, specular/diffuse/ambient properties and shading techniques.

The Program

Compile and run `BRTUTOR5.C` to display a revolving blue cube. Note the selected true colour mode. If you are running BRender under DOS, the following line in the hardware initialisation code selects 320 × 200, 15-bit true colour:

```
screen_buffer = DOSGfxBegin("VESA,W:320,H:200,B:15");
```

The following three lines of code are used to apply the material described in `cube.mat` to our revolving cube:

```
cube_material = BrFmtScriptMaterialLoad("cube.mat");
BrMaterialAdd(cube_material);
```

```
cube->material = BrMaterialFind("BLUE MATERIAL");
```

`BrFmtScriptMaterialLoad` loads a material from a material script and returns a pointer to an initialised `br_material` structure. `BrMaterialAdd` adds the new material to the registry. All materials must be added to the registry before they can be used in rendering.

`BrMaterialFind` searches for a material by name in the registry. You will recall that `BLUE MATERIAL` was the name entered in the identifier field in the material script file `cube.mat`. `BrMaterialFind` returns a pointer to a `br_material` (or NULL if the search was unsuccessful).

Use your text editor to experiment with the properties of `BLUE MATERIAL`, or to create your own material script files.

8-Bit Indexed Colour Mode

As previously noted, the complexities of palette management make 8-bit indexed colour more complex to work with than true colour. You can build your own palettes using the supplied utility MKRANGES. For now we will use the palette supplied with BRender (`std.pal`) when working in 8-bit mode.

The 256 colours in `std.pal` are divided into seven ranges, or 'colour ramps'. The first 64 colours represent shades of grey ranging from very dark grey (black) to very light grey (white). Colours 64 to 95 are various shades of blue.

Colour Ramps in std.pal

Range	Colour
0–63	Grey
64–95	Blue
96–127	Green
128–159	Cyan
160–191	Red
192–223	Magenta
224–255	Yellow

BRTUTOR5.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a Revolving Illuminated Blue Cube.
 */
#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"
int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
   br_actor *world, *observer, *cube;
   br_material *cube_material;
   int i;

/*********** Initialise BRender and Graphics Hardware ***********/
   BrBegin();
   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                        :         :         :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);
   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database *****************/
   /*
    * Load Root Actor. Load and Enable Default Light Source
    */
   world = BrActorAllocate(BR_ACTOR_NONE,NULL);
   BrLightEnable(BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL)));
   /*
    * Load and Position Camera
    */
   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
   observer->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                     BR_SCALAR(5.0));
   /*
    * Load and Position Cube Model
    */
   cube = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
   cube->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34RotateY(&cube->t.t.mat,BR_ANGLE_DEG(30));
   /*
    * Load and Apply Cube's Material
    */
   cube_material = BrFmtScriptMaterialLoad("cube.mat");
   BrMaterialAdd(cube_material);
```

```
  cube->material = BrMaterialFind("BLUE MATERIAL");

 /********************* Animation Loop *****************/
  for(i=0; i < 200; i++)  {
             BrPixelmapFill(back_buffer,0);
             BrPixelmapFill(depth_buffer,0xFFFFFFFF);
             BrZbSceneRender(world,observer,back_buffer,depth_buffer);
             BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
             BrMatrix34PostRotateX(&cube->t.t.mat,BR_ANGLE_DEG(2.0));
  }
  /*
   * Close down
   */
  BrPixelmapFree(depth_buffer);
  BrPixelmapFree(back_buffer);
  BrZbEnd();

                    ⋮          ⋮          ⋮

  BrEnd();
  return 0;
}
```

BRTUTOR5.C

So colour information in 8-bit colour mode specifies an index into a colour palette. A material script file describing a blue material (using `std`.pal) is given below.

cube.mat

```
# Material for Cube - 8-bit
# A plain blue texture

material = [
              identifier = "BLUE MATERIAL";
              ambient = 0.05;
              diffuse = 0.55;
              specular = 0.4;
              power = 10;
              flags = [light,smooth];
              index_base = 64;
              index_range = 30;
           ];
```

Colour number 64 indexes the start of the blue colour ramp. The `index_range` value defines the range of colours available for rendering this material. BRender uses lighting calculations to determine 'how blue' a model with this material should appear at any point. If no light is shining on the model, colour number 64 will be selected. If a bright light is shining directly onto the model, values closer to 90 will be selected.

Only two lines need to be changed in BRTUTOR5.C to display a revolving blue cube in 8-bit mode. One is highlighted in BRTUTR5b.C below. The other specifies 8-bit mode in the hardware initialisation code (refer to the program listing on your Tutorial Programs disk). If you are running BRender under DOS, the line:

```
screen_buffer = DOSGfxBegin(NULL);
```

replaces:

```
screen_buffer = DOSGfxBegin("VESA,W:320,H:200,B:15");
```

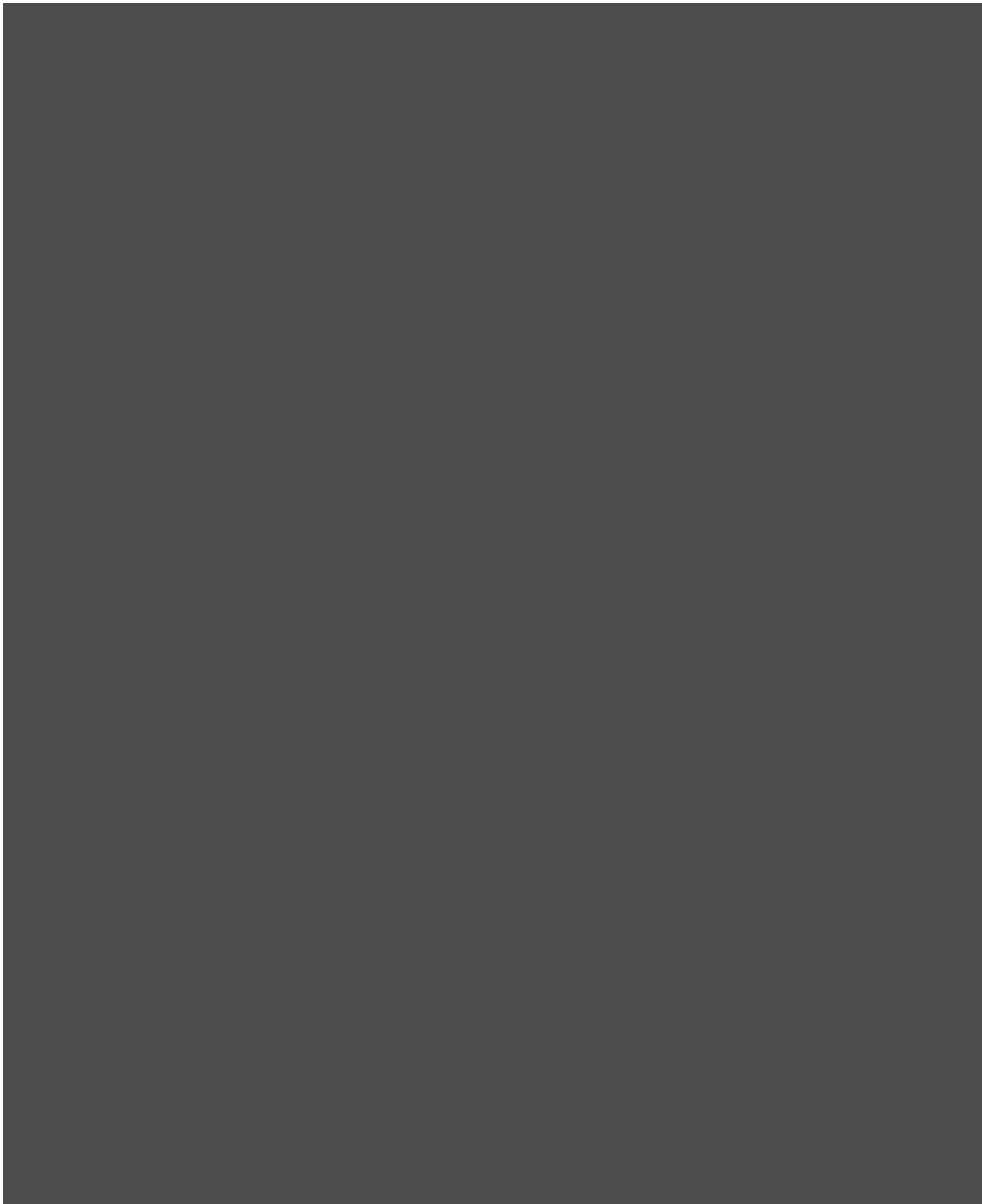
BRTUTR5B.C uses the material script file cube8.mat.

BRTUTR5B.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a Revolving Illuminated Blue Cube (8-bit mode)
 */

#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"

int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
   br_actor *world, *observer, *cube;
   br_material *cube_material;
   int i;

/******** Initialise BRender and Graphics Hardware ***********/

   BrBegin();
   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                      :          :          :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);
   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database **********************/

   world = BrActorAllocate(BR_ACTOR_NONE,NULL);
   BrLightEnable(BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL)));

   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
```

```
   observer->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                       BR_SCALAR(5.0));

   /*
    * Load and Position Cube Model
    */
   cube = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
   cube->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34RotateY(&cube->t.t.mat,BR_ANGLE_DEG(30));

   /*
    * Load and Apply Cube's Material
    */
   cube_material = BrFmtScriptMaterialLoad("cube8.mat");
   BrMaterialAdd(cube_material);
   cube->material = BrMaterialFind("BLUE MATERIAL");

/********************** Animation Loop ***********************/

   for(i=0; i < 200; i++) {
           BrPixelmapFill(back_buffer,0);
           BrPixelmapFill(depth_buffer,0xFFFFFFFF);
           BrZbSceneRender(world,observer,back_buffer,depth_buffer);
           BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
           BrMatrix34PostRotateX(&cube->t.t.mat,BR_ANGLE_DEG(2.0));
   }
   BrPixelmapFree(depth_buffer);/*Close down*/
   BrPixelmapFree(back_buffer);
   BrZbEnd();

                      :          :          :

   BrEnd();
   return 0;
}
```

BRTUTR5B.C



6 Texture Mapping

Imagine taking a photograph of the bark of a tree and somehow wrapping it around a cylinder to create a 3D image of a tree trunk, or superimposing a 2D image of a brick wall onto a flat rectangular face to make it look like a wall.

Texture mapping attempts to mimic this process. A texture map is a rectangular 2D image (perhaps scanned from a photograph or created using a paint package) that can be mapped onto the surface of a 3D model. The individual elements in a texture map are often called *texels*. Each of the polygons used in the construction of a model is mapped to a specific area of the texture map by means of texture co-ordinates assigned to the polygon's vertices. Texture co-ordinates are linearly interpolated between vertices in the same way that colour values for the interior of smooth shaded polygons are interpolated between the vertex colours.

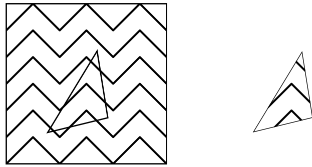


Figure 38 Texture Mapping

Texture mapping is by far the most effective way of constructing realistic images. Imagine trying to reproduce the wood grain in your kitchen table or a marble column without using a texture map.

`BRTUTOR6.C` creates an image of the Earth by superimposing a 2D texture map of the world onto a sphere. Note that a true colour mode has been selected. If you are running BRender under DOS, the following line in the hardware initialisation code selects 320 × 200, 15-bit true colour:

```
screen_buffer = DOSGfxBegin("VESA,W:320,H:200,B:15");
```

Loading the Texture Map

Texture maps must be loaded (or created) and registered, before they can be used in rendering:

```
BrMapAdd(BrPixelmapLoad("earth15.pix"));
```

The file `earth15.pix` stores a texture map depicting the Earth's surface. `BrPixelmapLoad` loads a `.pix` file and returns a pointer to `br_pixelmap`. `BrMapAdd` adds a texture to the registry. Note that textures, like materials, are stored in the registry by name. When you want to use a registered texture, you can refer to it by name. The texture loaded from `earth15.pix` is called, logically enough, 'earth'. It is assigned in the material script file `earth.mat`.

Loading the Material

The following code loads a material description from the material script file earth.mat and adds it to the registry:

```
planet_material = BrFmtScriptMaterialLoad("earth.mat");
BrMaterialAdd(planet_material);
```

Note that the same result could have been achieved using the following code:

```
BrMaterialAdd(BrFmtScriptMaterialLoad("earth.mat"));
```

Storing a pointer to the loaded material in planet_material allows the programmer to dynamically access fields in the structure.

Assigning the Material

The following code assigns the material earth_map to the planet actor:

```
planet->material = BrMaterialFind("earth_map");
```

'earth_map' is the name of the material described in earth.mat.

Let's take a look at the material script file earth.mat.

earth.mat

```
# This material script file describes the appearance
# of the material "earth_map"

material = [
              identifier = "earth_map";
              ambient = 0.05;
              diffuse = 0.55;
              specular = 0.4;
              power = 20;
              flags = [];
              map_transform = [[1,0], [0,1], [0,0]];
              colour_map = "earth";
           ];
```

'earth_map' is the name by which this material is known and registered.

BRTUTOR6.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a Texture Mapped Sphere (15-bit colour).
 */

#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"

int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
   br_actor *world, *observer, *planet;
   br_material *planet_material;
   int i;

/*********** Initialise BRender and Graphics Hardware ***********/

   BrBegin();

   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                      :        :        :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);

   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database *********************/

   world = BrActorAllocate(BR_ACTOR_NONE,NULL);
   BrLightEnable(BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL)));
   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
   observer->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                        BR_SCALAR(5.0));

   /*
    * Load and Position Planet Actor
    */

   planet = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
   planet->model = BrModelLoad("sph32.dat");
   BrModelAdd(planet->model);
   planet->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34RotateX(&planet->t.t.mat,BR_ANGLE_DEG(90));
```

```
    /*
     * Load and Register "earth" Texture
     */

    BrMapAdd(BrPixelmapLoad("earth15.pix"));

    /*
     * Load and Apply Earth Material
     */

    planet_material = BrFmtScriptMaterialLoad("earth.mat");
    BrMaterialAdd(planet_material);
    planet->material = BrMaterialFind("earth_map");

/********************* Animation Loop **********************/

    for(i=0; i < 200; i++) {
               BrPixelmapFill(back_buffer,0);
               BrPixelmapFill(depth_buffer,0xFFFFFFFF);

               BrZbSceneRender(world,observer,back_buffer,depth_buffer);
               BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
               BrMatrix34PostRotateY(&planet->t.t.mat,BR_ANGLE_DEG(2.0));
            }

    /*
     * Close down
     */
    BrPixelmapFree(depth_buffer);
    BrPixelmapFree(back_buffer);
    BrZbEnd();

                      .          .          .
                      .          .          .
                      .          .          .

    BrEnd();
    return 0;
}
```

BRTUTOR6.C

Note that, depending on your platform and the version of BRender installed, lighting calculations may not be performed for texture mapped models in true-colour modes. In addition smooth shading may not be supported. **Light and smooth shading may need to be turned off.** Refer to your installation guide for details. Note `flags = []` disables all rendering flags.

`map_transform` is a matrix representing a general texture map transformation. The transform is applied to texture coordinates. This enables textures to be translated, scaled and rotated. `[[1,0], [0,1], [0,0]]` represents the identity matrix (remember texture maps are two-dimensional).

`[[1,0], [0,1], [0.5,0]]` encodes a translation 0.5 units in x.

Scaling transformations can be used to produce a tiling effect. For example `[[3,0], [0,3], [0,0]]` generates a new texture map made up of 9 (3*3) copies of the original map (each reduced in size by a factor of 9).

`[[0.707,-0.707], [0.707,0.707], [0,0]]` is a rotation of approximately +45°.

See *Computer Graphics – Principles and Practice*, by James D. Foley *et al.*, Chapter 5, for a comprehensive overview of 2D transformations.

`colour_map = "earth"` specifies the name of the texture map to be retrieved from the registry. If you don't know, or can't remember, what a texture map is called, use the TEXCONV utility to retrieve its name from the relevant `.pix` file. TEXCONV is BRender's texture import and conversion utility, command line options are detailed in Chapter 8.

The texture map used in our example, called `'earth'`, was loaded from `earth15.pix`. To verify this name, enter the following command line:

```
texconv earth15.pix
```

The following will be displayed,

```
TEXCONV 1.5 Copyright (C) 1994 by Argonaut Technologies Limited
Loaded 'earth' as BR_PMT_RGB_555 (256,256)
```

confirming that the texture map is called `'earth'`. `BR_PMT_RGB_555` tells us that the pixel map format is RGB, 5 bits per pixel(15-bit true colour).

A number of texture maps are supplied with BRender to help get you started (all have `.pix` extensions). However, you will want to define and apply your own textures – perhaps scanned from a photograph, or created using a paint package such as Photoshop. TEXCONV is used for importing and converting textures to be used in BRender applications.

To convert the sample pixel map `bluebels.bmp`, included on your Tutorial Programs disk, to BRender 15-bit RGB format, enter the following command line:

```
texconv bluebels.bmp -n bluebells -c BR_PMT_RGB_555 -o bbell15.pix
```

The input file, `bluebels.bmp`, is in 8-bit, 256 indexed colour format. TEXCONV creates an output file, `bbell15.pix`, in 15-bit RGB (BRender `.pix`) format. The `-n` option assigns the name `'bluebells'` to the texture map stored in `bbell15.pix`. This newly created texture can now be used in BRender applications. You might want to try substituting it for `'earth'` in the above program. Remember that you can experiment with material properties without having to re-compile the program every time you make a change – simply edit the material script file as required.

8-Bit Colour

Lighting calculations are performed for textures in 8-bit colour mode. Textures are stored as pixel maps made up of indices into a *shade table*. A shade table is a two dimensional array (represented as a pixel map) containing, typically, 256 columns for colour index values and 64 rows for representing lighting levels.

Texture Map to be lit
100
Shade Table
0 100 255
0 30 64
40
Dark
Light
Colour Palette
0 40 255
RGB
Colour index value output to sc

Figure 39 A Shade Table

Each entry in the shade table is an index into a colour palette. The shade table is usually arranged so that each row contains a number of shades of the same colour, ranging from very dark to very bright.

You can build your own shade tables using the tool MKSHADES. The shade table supplied with BRender, `shade.tab`, references the palette `std.pal`. To display BRender's standard shade table, enter the following command line:

```
texconv -I pixelmap shade.tab -P std.pal -v
```

BRTUTR6B.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a Texture-Mapped Sphere (8-bit colour).
 */

#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"

int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette, *shade;
   br_actor *world, *observer, *planet;
   br_material *planet_material;
   int i;

/*********** Initialise BRender and Graphics Hardware ***********/

   BrBegin();

   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                        :         :         :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);
   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

   /*
    * Load Shade Table
    */
   shade = BrPixelmapLoad("shade.tab");
   if (shade==NULL)
        BR_ERROR("Couldn't load shade.tab");
   BrTableAdd(shade);

/*************** Build the World Database **********************/

   world = BrActorAllocate(BR_ACTOR_NONE,NULL);
   BrLightEnable(BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL)));

   /*
    * Load and Position Camera
    */
   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
   observer->t.type = BR_TRANSFORM_MATRIX34;
```

```
    BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                         BR_SCALAR(5.0));

    /*
     * Load and Position Planet Actor
     */
    planet = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
    planet->model = BrModelLoad("sph32.dat");
    BrModelAdd(planet->model);
    planet->t.type = BR_TRANSFORM_MATRIX34;
    BrMatrix34PostRotateX(&planet->t.t.mat,BR_ANGLE_DEG(90));

    /*
     * Load and Register "earth" Texture
     */
    BrMapAdd(BrPixelmapLoad("earth8.pix"));

    /*
     * Load and Apply Earth Material
     */
    planet_material = BrFmtScriptMaterialLoad("earth8.mat");
    BrMaterialAdd(planet_material);
    planet->material = BrMaterialFind("earth8_map");

/********************* Animation Loop **********************/

    for(i=0; i < 200; i++) {
                BrPixelmapFill(back_buffer,0);
                BrPixelmapFill(depth_buffer,0xFFFFFFFF);
                BrZbSceneRender(world,observer,back_buffer,depth_buffer);
                BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
                BrMatrix34PostRotateY(&planet->t.t.mat,BR_ANGLE_DEG(2.0));
    }

    /*
     * Close down
     */
    BrPixelmapFree(depth_buffer);
    BrPixelmapFree(back_buffer);
    BrZbEnd();
                    :         :         :
    BrEnd();
    return 0;
}
```

BRTUTR6B.C

BRTUTR6B.C displays a texture-mapped sphere in 8-bit colour mode. The material script file earth8.mat is listed below.

earth8.mat

```
# This material script file describes the appearance
# of the material "earth8_map"

material = [
              identifier = "earth8_map";
              ambient = 0.05;
              diffuse = 0.55;
              specular = 0.4;
              power = 20;
              flags = [light,smooth];
              map_transform = [[1,0], [0,1], [0,0]];
              colour_map = "earth";
              index_shade = "shade_table";
           ];
```

Note that the shade table, like the texture map, is referenced by name.

7 File Conversion

A number of utilities are supplied with BRender for importing and converting files.

- **3DS2BR** converts Autodesk 3D Studio files saved in `.3ds` binary file format to BRender `.dat` format.
- **GEOCONV** convert models from one geometry format to another.
- **TEXCONV** provides texture importing, scaling, quantizing and remapping functions.

Converting 3D Studio (.3ds) Files

3DS2BR converts model descriptions stored in `.3ds` binary file format to BRender `.dat` format. Type 3DS2BR, followed by return, to display a list of 3DS2BR command line options. The steps involved in converting a `.3ds` file are detailed below. This example converts the model description file `duck.3ds`, supplied with 3D Studio (and included on the Tutorial Programs disk for your convenience) to BRender format. `duck.3ds` describes a three component model hierarchy – a yellow duck body with two black eyes.

The command line,

```
3DS2BR duck.3ds -nomatrix -mod duck.dat -scr duck.mat
```

extracts information describing model geometry and stores it in `duck.dat`. Material descriptions are stored in the script file `duck.mat` (the `-nomatrix` option ensures that inverse mesh matrices are not applied – see Chapter 8 for more information on 3DS2BR).

Enter the following command line,

```
geoconv duck.dat -l
```

to display information on the newly created file `duck.dat`. The following listing should be displayed:

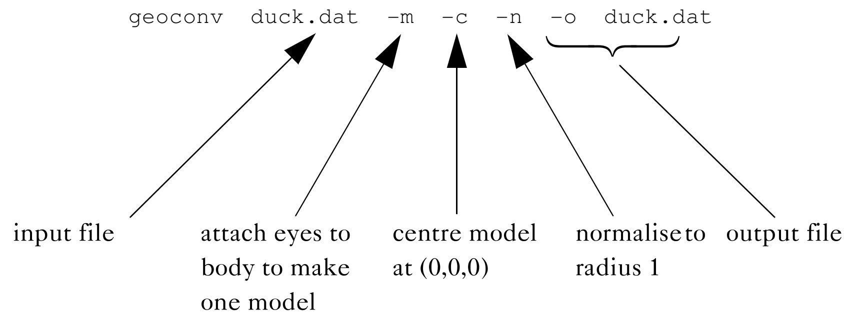
```
GEOCONV 1.21 Copyright (C) 1994-1995 by Argonaut Technologies Limited
Models: 3
```

	Vert s	Face s	Edges	Fgrps	Vgrp s	Sor t	DOTQ	Radius
Model List Object05	17	24	40	1	1	17	----	244.08 5
Model List Object04	17	24	40	1	1	17	----	1208.5 9
Model List Object03	268	516	783	1	1	268	----	2181.6 3
TOTAL	302	564	863	3	3			

```
Materials: 2
   BLACK PLASTIC
   YELLOW PLASTIC
```

This listing tells us that three models are combined to build the model described in `duck.dat` – one model describes the duck's body, the other two its eyes.

The following command line collapses the model hierarchy described in `duck.dat` into a single model. This model is then translated to the origin and scaled to fit within a sphere of radius 1.



If you again enter `geoconv duck.dat -l` the following listing will be displayed,

```
GEOCONV 1.21 Copyright (C) 1994-1995 by Argonaut Technologies Limited
Models: 1
             Vert    Face    Edge    Fgrp    Vgrp    Sor    DOT
             s       s       s       s       s       t      Q      Radius
Model         302     564     863      2       2     302    ---    0.9999
List                                                          -
Object03
Materials: 2
   BLACK PLASTIC
   YELLOW PLASTIC
```

`duck.dat` now describes a single model of radius 1 (0.9999). Note that material associations are retained.

To recap: the following command lines were used to import the 3D Studio `.3ds` model description file.

```
3DS2BR duck.3ds -nomatrix -mod duck.dat -scr duck.mat
geoconv duck.dat -m -c -n -o duck.dat
```

Two output files were created – `duck.dat` to store model geometry, `duck.mat` to store material descriptions. The model hierarchy described in `duck.dat` was collapsed into a single model. This model was then translated to the origin before being normalised to a radius of 1.

The imported 'duck' model is used in `BRTUTOR7.C`. Note that associated materials must be loaded and registered **before** the model is loaded (otherwise it will appear

matt grey). If associated materials are not found in the registry when a model is loaded the default material is used. The application may then assign any registered material to the model.

The following code loads materials from the script file `duck.mat`, and adds them to the registry.

```
i = BrFmtScriptMaterialLoadMany("duck.mat",mats,BR_ASIZE(mats));
BrMaterialAddMany(mats,i);
```

`BrFmtScriptMaterialLoadMany` and `BrMaterialAddMany` are called because more than one material is being loaded and registered. The macro `BR_ASIZE` (defined in header file `compiler.h`) calculates the size of an array. `BrFmtScriptMaterialLoadMany` returns the number of materials successfully loaded.

```
duck = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
duck->model = BrModelLoad("duck.dat");
BrModelAdd(duck->model);
```

The material script file `duck.mat` is listed below.

duck.mat

```
# BRender Material Script

material = [
              identifier = "BLACK PLASTIC";
              flags = [light,smooth];
              colour = [0,0,0];
              ambient = 0.000000;
              diffuse = 0.000000;
              specular = 1.000000;
              power = 69.309998;
           ];

material = [
              identifier = "YELLOW PLASTIC";
              flags = [light,smooth];
              colour = [202,179,52];
              ambient = 0.679216;
              diffuse = 0.679216;
              specular 0.741569;
              power = 23.770000;
           ];
```

`YELLOW PLASTIC` is used for the duck body, `BLACK PLASTIC` for the eyes. Note that the above material description will only work in true- colour mode. In 8-bit indexed mode, the `colour` field is ignored. Instead, BRender looks for an index into a colour palette. We will need to edit the script file if we want to use the duck model in 8-bit mode. In the palette provided with BRender (`std.pal`), the ‘yellow’ colour

ramp ranges from index 224 (yellow so dark it's black) to index 255 (yellow so bright it's white). Try editing `duck.mat` as follows, and running the program in 8-bit indexed mode (simply pass *NULL* to *`DOSfxBegin`* instead of *`"VESA,W:320, H:200,B:15"`*).

duck.mat

```
# BRender Material Script
#
material = [
              identifier = "BLACK PLASTIC";
              flags = [light,smooth];
              colour = [0,0,0];
              ambient = 0.000000;
              diffuse = 0.000000;
              specular = 1.000000;
              power = 69.309998;
              index_base = 0;
              index_range = 0;
           ];
material = [
              identifier = "YELLOW PLASTIC";
              flags = [light,smooth];
              colour = [202,179,52];
              ambient = 0.679216;
              diffuse = 0.679216;
              specular 0.741569;
              power = 23.770000;
              index_base = 224;
              index_range = 24;
           ];
```

BRTUTOR7.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a Revolving Yellow Duck.
 */

#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"

int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
   br_actor *world, *observer, *duck;
   int i;
   br_material *mats[10]; /*for storing pointers to material descriptions*/

/********* Initialise BRender and Graphics Hardware ***********/

   BrBegin();

   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                    :          :          :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);

   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database *********************/

   world = BrActorAllocate(BR_ACTOR_NONE,NULL);
   BrLightEnable(BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL)));

   /*
    * Load and Position Camera
    */
   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
   observer->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                         BR_SCALAR(5.0));

   /*
    * Load and Apply Duck Materials
    */
   i = BrFmtScriptMaterialLoadMany("duck.mat",mats,BR_ASIZE(mats));
   BrMaterialAddMany(mats,i);
```

```
   /*
    * Load and Position Duck Model
    */
   duck = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
   duck->model = BrModelLoad("duck.dat");
   BrModelAdd(duck->model);
   duck->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34RotateX(&duck->t.t.mat,BR_ANGLE_DEG(30));

/********************** Animation Loop ***********************/

   for(i=0; i < 200; i++) {
               BrPixelmapFill(back_buffer,0);
               BrPixelmapFill(depth_buffer,0xFFFFFFFF);
               BrZbSceneRender(world,observer,back_buffer,depth_buffer);
               BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
               BrMatrix34PostRotateX(&duck->t.t.mat,BR_ANGLE_DEG(2.0));
   }

   /*
    * Close down
    */
   BrPixelmapFree(depth_buffer);
   BrPixelmapFree(back_buffer);
   BrZbEnd();

                       :          :          :

   BrEnd();
   return 0;
}
```

BRTUTOR7.C

Importing Texture Maps

As we have seen, you can import models created in 3D Studio using 3DS2BR and GEOCONV. You will probably also want to import your own texture maps (perhaps created in a dedicated paint package or scanned from a photograph). TEXCONV, introduced in Chapter 6, allows you to do this.

15-bit True Colour

The following command line imports the pixel map `gold.gif` and generates a `.pix` file in BRender 15-bit format.

```
texconv gold.gif -n gold -c BR_PMT_RGB_555 -o gold15.pix
```

`gold.gif` can be found on your Tutorial Programs disk. Note that textures can only be applied to models that have been saved with texture co-ordinates. Remember to apply texture co-ordinates to models created in 3D Studio (or other 3D modelling package) before saving and importing. If, however, you forgot to apply texture co-ordinates in your modelling package, all is not lost. The BRender function `BrModelApplyMap` generates texture co-ordinates for a model's vertices. It allows you to select from a number of mapping options – planar, spherical, cylindrical, disk or none. The mapping option determines how a texture is wrapped around a model (refer to your technical reference manual for further details). Note that this function may not be as versatile as the texture mapping facility in your modelling package.

`BRTUTOR8.C` applies the imported texture `'gold'` to the `duck` model imported earlier. This model was originally saved without texture co-ordinates, so we need to call `BrModelApplyMap`.

The following lines of code load and register the `'gold'` texture map (type `texconv gold15.pix` to verify the name `'gold'`).

```
gold_pm = BrPixelmapLoad("gold15.pix");
if (gold_pm==NULL)
BR_ERROR0("Couldn't load gold15.pix");
BrMapAdd(gold_pm);
```

Note that:

```
BrMapAdd (BrPixelmapLoad("gold15.pix"));
```

would have achieved the same result. It would not, however, have informed us if it was unable to load the specified file. This would happen if, for instance, you forgot to run the command line:

```
texconv gold.gif -c BR_PMT_RGB_555 -o gold15.pix
```

to generate the BRender 15-bit texture map `gold15.pix`.

The material script file containing the material `gold` is loaded and registered:

```
BrMaterialAdd(BrFmtScriptMaterialLoad("gold15.mat"));
```

Texture co-ordinates are applied:

```
BrModelApplyMap(duck->model,BR_APPLYMAP_PLANE,NULL);
```

The `'gold'` material is assigned:

```
duck->material = BrMaterialFind("gold15");
```

Note that the script file `duck.mat`, containing duck's associated materials `BLACK PLASTIC` and `YELLOW PLASTIC`, has not been loaded. If these materials had been registered, they would have been automatically assigned and this line of code would have been ignored!

The material script file `gold15.mat` is given below.

gold15.mat

```
# This material script file describes the appearance
# of the material "gold15"
material = [
              identifier = "gold15";
              colour = [0,255,0];
              ambient = 0.05;
              diffuse = 0.55;
              specular = 0.4;
              power = 20;
              flags = [];
              map_transform = [[1,0], [0,1], [0,0]];
              colour_map = "gold";
           ];
```

Note that, depending on your platform and the version of BRender installed, light and smooth may need to be turned off (`flags = [];`) in 15-bit mode!

BRTUTOR8.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a Texture Mapped Duck (15-bit).
 */

#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"

int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
   br_actor *world, *observer, *duck;
   br_pixelmap *gold_pm;
   int i;

/*********** Initialise BRender and Graphics Hardware ***********/

   BrBegin();

   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                       :          :          :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);

   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database **********************/

   world = BrActorAllocate(BR_ACTOR_NONE,NULL);
   BrLightEnable(BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL)));

   /*
    * Load and Position Camera
    */
   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
   observer->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                         BR_SCALAR(5.0));

   /*
    * Load and Register "gold" Texture
    */
   gold_pm = BrPixelmapLoad("gold15.pix");
   if (gold_pm==NULL)
        BR_ERROR0("Couldn't load gold15.pix");
```

```
    BrMapAdd(gold_pm);

    /*
     * Load and Apply "gold" Material
     */
    BrMaterialAdd(BrFmtScriptMaterialLoad("gold15.mat"));

    /*
     * Load and Position Duck Model
     */
    duck = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
    duck->model = BrModelLoad("duck.dat");
    BrModelAdd(duck->model);
    BrModelApplyMap(duck->model,BR_APPLYMAP_PLANE,NULL);
    duck->t.type = BR_TRANSFORM_MATRIX34;
    BrMatrix34RotateX(&duck->t.t.mat,BR_ANGLE_DEG(30));
    duck->material = BrMaterialFind("gold15");

/********************* Animation Loop **********************/

    for(i=0; i < 200; i++){
                BrPixelmapFill(back_buffer,0);
                BrPixelmapFill(depth_buffer,0xFFFFFFFF);
                BrZbSceneRender(world,observer,back_buffer,depth_buffer);
                BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
                BrMatrix34PostRotateX(&duck->t.t.mat,BR_ANGLE_DEG(2.0));
    }

    /*
     * Close down
     */
    BrPixelmapFree(depth_buffer);
    BrPixelmapFree(back_buffer);
    BrZbEnd();
                        :           :           :
    BrEnd();
    return 0;
}
```

BRTUTOR8.C

BRTUTR8B.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a Texture Mapped Duck (8-bit).
 */

#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"

int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette, *shade;
   br_actor *world, *observer, *duck;
   br_pixelmap *gold_pm;
   int i;

/*********** Initialise BRender and Graphics Hardware ***********/

   BrBegin();

   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                        :          :          :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);

   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

   /*
    * Load Shade Table
    */
   shade = BrPixelmapLoad("shade.tab");
   if (shade==NULL)
        BR_ERROR0("Couldn't load shade.tab");
   BrTableAdd(shade);

/*************** Build the World Database **********************/

   world = BrActorAllocate(BR_ACTOR_NONE,NULL);
   BrLightEnable(BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL)));

   /*
    * Load and Position Camera
    */
   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
   observer->t.type = BR_TRANSFORM_MATRIX34;
```

```
    BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                        BR_SCALAR(5.0));

    /*
     * Load and Register "gold" Texture
     */
    gold_pm = BrPixelmapLoad("gold8.pix");
    if (gold_pm==NULL)
          BR_ERROR0("Couldn't load gold8.pix");
    BrMapAdd(gold_pm);

    /*
     * Load and Apply "gold" Material
     */
    BrMaterialAdd(BrFmtScriptMaterialLoad("gold8.mat"));

    /*
     * Load and Position Duck Model
     */
    duck = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
    duck->model = BrModelLoad("duck.dat");
    BrModelAdd(duck->model);
    BrModelApplyMap(duck->model,BR_APPLYMAP_PLANE,NULL);
    duck->t.type = BR_TRANSFORM_MATRIX34;
    BrMatrix34RotateX(&duck->t.t.mat,BR_ANGLE_DEG(30));
    duck->material = BrMaterialFind("gold8");

/********************* Animation Loop **********************/

    for(i=0; i < 200; i++) {
                BrPixelmapFill(back_buffer,0);
                BrPixelmapFill(depth_buffer,0xFFFFFFFF);
                BrZbSceneRender(world,observer,back_buffer,depth_buffer);
                BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
                BrMatrix34PostRotateX(&duck->t.t.mat,BR_ANGLE_DEG(2.0));
    }

    /*
     * Close down
     */
    BrPixelmapFree(depth_buffer);
    BrPixelmapFree(back_buffer);
    BrZbEnd();

                        :          :          :

    BrEnd();
    return 0;
}
```

BRUTR8B.C

8-bit Indexed Colour

Enter the following command line to generate an 8-bit pixel map using `gold.gif`.

```
texconv gold.gif -n gold -c BR_PMT_INDEX_8 -Q std.pal -o gold8.pix
```

The `-Q std.pal` option quantizes the input pixel map to the standard palette. The colours in `std.pal` that best approximate those in the pixel map associated with the `.gif` file are substituted in the `.pix` file.

`BRTUTR8b.C` renders the texture-mapped duck in 8-bit mode.

gold8.mat

```
# This material script file describes the appearance
# of the material "gold8"

material = [
              identifier = "gold8";
              colour = [0,255,0];
              ambient = 0.05;
              diffuse = 0.55;
              specular = 0.4;
              power = 20;
              flags = [light,smooth];
              map_transform = [[1,0], [0,1], [0,0]];
              colour_map = "gold";
              index_shade = "shade_table";
           ];
```

8 BRender Tools

A number of tools are provided to assist you in pre-preparing data for use in BRender programs:

- **3DS2BR** converts Autodesk 3D Studio files saved in `.3ds` file format to BRender `.dat` format.
- **GEOCONV** manipulates (scales, centres etc.) model geometry and converts models from one geometry format to another.
- **DXF2BR** is a geometry converter that converts AutoCad `.dxf` files to BRender `.dat` format.
- **TEXCONV** is used to import and manipulate textures.
- **MKSHADES** is a tool for creating shade tables.
- **MKRANGES** is used to construct customised palettes using colour ramps.
- **PALJOIN** is used to cut and paste palettes.
- **VIEWPAL** is a tool for displaying palettes on the screen.

You were introduced to 3DS2BR, GEOCONV and TEXCONV in Chapter 7. They will be discussed in more detail here. DXF2BR converts AutoCad files to BRender format. MKRANGES, MKSHADES, PALJOIN and VIEWPAL are used in 8-bit indexed mode to construct and view user defined palettes and shade tables.

The techniques described in this chapter for importing models and textures are summarised in Figure 40. These techniques are intended as illustrative examples of how you might approach the task of importing models, materials and textures (and that of palette management if working in 8-bit mode). You should devise your own strategy, according to the requirements of your application, and adapt the above techniques to your needs.

3D Studio
.asc file
.3ds file
GEOCONV
.dat file
3DS2BR
Model file (.dat)
ACT file (.act)
Material file (.mat)
Material Script file (.mat)
Log file (.log)
BrModelLoa
BrModelLoadMa
BrActorLoa
BrActorLoadMa
BrMaterialLo
BrMaterialLoadM
BrFmtScriptMaterialL
BrFmtScriptMaterialLoad
text file
MKRANGES
strip palette
PALJOIN
palette
texture palette
MKSHADES
shade table
Textures (.gif, .tga)
TEXCONV -R -O palette
TEXCONV -Q -O pixelmap
pixelmap (.pix)
BrPixelmapLo
BrPixelmapLoadM

Figure 40

3DS2BR

3DS2BR is a DOS command line tool for converting Autodesk 3D Studio (3DS) models to one or more BRender format files.

3DS2BR interprets the information in the `.3ds` file and converts it to its nearest BRender equivalent. Depending on which command line options are specified, it will generate one or more of the following files:

- a binary file containing the meshes in the model
- a binary file containing all the materials in the model
- a binary file containing a BRender hierarchy tying these together
- a text file storing conversion information
- a material script file that can be loaded directly into BRender, describing the materials used with the model.

Command line format: `3DS2BR 3ds_file.3ds {options}` where:
`3ds_file.3ds` is the 3D Studio Binary file you wish to convert.

3DS2BR command line options are listed below:

Option	Description
`[-h]`	Display a help screen detailing command line options
`[-v]`	Turn on warning and information messages. By default, only error messages are displayed.
`[-mod <model file name>]`	Save models to file
`[-mat <material file name>]`	Save materials to file
`[-scr <material script file name>]`	Save materials to script file
`[-act <actor file name>]`	Save actor hierarchy to file
`[-log <log file name>]`	Save logfile containing conversion information
`[-flat]`	Build a flat hierarchy, ignoring keyframer data
`[-nopivot]`	Ignore keyframer pivot point
`[-nomatrix]`	Ignore mesh matrix
`[-noaxis]`	Do not remap axes to correspond to 3DS user interface
`[-nl]`	Replace lights with dummy actors
`[-nc]`	Replace cameras with dummy actors
`[-hither <distance>]`	Set all camera hither distances to this value

[-yon <distance>]	Set all camera yon distances to this value
[-pc]	Set perspective correction for all textured materials

The -mod option extracts mesh information describing 3DS model geometry, and stores it in the specified binary file. The -mat option extracts material description information, and stores it as a binary file. The -act option extracts the 3DS actor hierarchy. The -log option stores conversion information in a text file. The -scr option extracts material description information and stores it in a BRender material script file (a text file).

You can specify any, or a combination, of the above options. For example, if you are only interested in importing the mesh data describing model geometry, use the -mod option:

```
3DS2BR 3ds_file.3ds -mod model_file.dat
```

This command line saves the converted mesh data in model_file.dat. Models stored as .dat files can be loaded into BRender programs using BrModeLoad/Many.

Command line options can also be read from a file. The following command line performs a full conversion, generating and saving all five BRender format files, while displaying warning and information messages:

```
3DS2BR 3ds_file.3ds @options -v
```

where the options file contains the following lines:

```
-mod    model_file.dat
-mat    material_file.mat
-act    actor_file.act
-log    log_file.log
-scr    mat_script_file.mat
```

Note that the above extensions are adopted by convention, and are not obligatory.

If you want to convert a hierarchical 3DS model and use it directly in BRender, use the -act and -mod options. For example, suppose a hierarchical model of a human body was contained in the file person.3ds:

```
3DS2BR person.3ds -act person.act -mod person.dat
```

would create a file person.dat containing the models defined in person.3ds, and a file person.act defining the hierarchical relationships between these models. The 3DS hierarchy could be reproduced in BRender by loading and registering the converted models,

```
no_of_models = BrModelLoadMany("person.dat", * model_array,
              max_no);
               BrModelAddMany(* model_array,no_of_models);
```

```
root_actor = BrActorLoad("person.act");
```

If you don't wish to retain an actor hierarchy, you can collapse the contents of a `.dat` file containing multiple models into a single model (using GEOCONV's `-m` option). In this case you should specify the `-nomatrix` option when converting model data using 3DS2BR. This model can then be loaded using `BrModelLoad`.

Options in this file can be separated by any whitespace characters. Lines in this file are limited to 255 characters.

Examples illustrating the use of 3DS2BR are given later in this chapter.

GEOCONV

GEOCONV converts models from one geometry format to another, allowing models generated using 3D modelling packages to be used with BRender. It can also be used to manipulate imported models.

Command line format: `geoconv {options}`

The options are treated as commands, executed from left to right:

-?	Display command line options
<input file>	Load a model file
-I <input-type>	Set input file type
-o <file>	Generate output file
-O <output-type>	Set output data type
-n	Normalise models to radius of 1.0
-c	Centre models on 0,0,0
-f	Flip face normals
-w <map-axes>	Fix wrapped texture co-ordinates
-F <flag>	Set or clear a model flag
-p	Remove identical vertices
-P <FLOAT>	Merge vertices within a given tolerance
-C	Remove degenerate faces, unused vertices and duplicate faces
-m	Collapse current data to one actor and one model
-r <pattern>	Restrict subsequent operations to items matching <pattern>
-l	List current data
-g	Set each model to a different smoothing group
-S <sort_type>	Set sorting on output
-N <material-name>	Set all models to use named material
-N default	Set all models to use default material
-M <map-type>,<axis>,<axis>	Apply a default U,V mapping to models
-s <float>	Uniform scale applied to models
-s <float>,<float>,<float>	Non-uniform scale applied to models
-t <float>,<float>,<float>	Translation applied to models
-a <axis>,<axis>,<axis>	Remap axes
-D <name>	Remap materials
-L <name>	Rename materials
<input-type> = dat nff asc	 BRender model files Eric Haines' Neutral File Format 3d STUDIO .asc FILES

If a type is not specified, it will be guessed from the filename extension.

`<output-type> =`
`models`
`c-models`
	
	.dat
	Source code for model data structures	

`<map-axes> =`
`u`
`v`
`uv`
	
	fix wrapping along u axis
	fix wrapping along v axis
	fix wrapping along uv axis	

`<axis> =`
`x y z`
`+x +y +z`
`-x -y -z`
	
	positive input axes
	positive input axes
	negative input axes	

`<map-type> =`
`none`
`disc`
`plane`
`cylinder`
`sphere`		

`<material-name> =`
`<string>`
`default`		

`<sort-type> =`
`none`
`name`		

`<flag> =`
`+d`
`-d`
`+o`
`-o`
`+t`
`-t`
`+q`
`-q`
	
	Set
	Clear
	Set
	Clear
	Set
	Clear
	Set
	Clear
		
		`BR_MODF_DONT_WELD`
		`BR_MODF_DONT_WELD`
		`BR_MODF_KEEP_ORIGNAL`
		`BR_MODF_KEEP_ORIGNAL`
		`BR_MODF_GENERATE_TAGS`
		`BR_MODF_GENERATE_TAGS`
		`BR_MODF_QUICK_UPDATE`
		`BR_MODF_QUICK_UPDATE`

There are three related points worth noting:

- 3D Studio saves models with their longest axis down Z, and may need to be re-oriented with the remap axis option (`-a`).
- For the sake of consistency, it is advisable to pre-scale models as necessary with this tool, rather than scale them within a BRender application.
- By default, if there are many models in a source file, they will be separated into individual models (but saved as a single file).

DXF2BR

DFX2BR converts AutoCad `.dxf` files to BRender `.dat` format.

Command line format: `DFX2BR {options}`

Command line options are given below:

`-o <file>`	Generate output file
`-d <value>`	Extrusion depth of 2D objects
`-n`	Normalise models to radius 1.0
`-c`	Center models on 0,0,0
`-sx <value>`	Scale in x direction
`-sy <value>`	Scale in y direction
`-sz <value>`	Scale in z direction
`-l <comma separated list>`	Specify layers required

Type `DFX2BR` to display a list of command line options.

TEXCONV

TEXCONV is a command-line tool for importing and manipulating textures.

Command line format: `texconv {options}`

Command line options are given below:

`-?`	Display command line options
`<input file>`	Load a file
`-I <input-type>`	Set input file type
`-O <output-type>`	Set output file type
`-a`	Toggle current 32-bit pixelmaps to exclude/ include alpha data (default exclude)
`-c <pixelmap-type>[, t]`	Convert to pixelmap type, may involve quantizing. 't' is the alpha channel threshold (0–255) for conversions involving 32-bit pixelmaps.
`-f`	'Forget' all current data
`-n <name>`	Assign identifier 'name' to data
`-o <file>`	Generate output file
`-r <file>,<pixelmap-type>, offset,x,y[,P]`	Load raw data file as pixelmap-type, with pixel dimensions x,y, from offset into file. P is specified to load as a palette

-v	View snapshot of current data (only BR_PMT_INDEX_8)
-x	Flip left/right
-y	Flip top/bottom
-P <name>[,RAW]	Apply palette from a file to all current indexed pixelmaps. If RAW is specified, the palette file is 768 byte RGB, otherwise pixelmap format
-Q <name>[,b,r]	Quantize to palette supplied (pixelmap format) using (b)ase and (r)ange palette entries
-R b,r	Quantize and remap to (b)ase,(r)ange colours (both values in the range 0–255)
-S x,y[,<float>]	Scale to new x,y dimensions using optional filter size (default 3.0)
@file	Perform all operations contained in <file>
<input-type> = pix bmp gif tga iff	 BRender Pixelmap format Windows BMP format (4, 8, 24, RLE4, RLE8 CompuServe GIF format (1- to 8-bit) Targa TGA format (8-, 15-, 16-, 24-, 32-bit) Amiga IFF/LBM format (1- to 8-bit)

If an input type is not specified, it will be guessed from the filename extension.

<output type> = palette image pixelmap targa	Palette information stripped from bitmap (.pix) Pixelmap without palette (.pix) Image with any palette information (.pix) Output a .tga file

The default output type is pixelmap.

BR_PMT_INDEX	8-bit indexed
BR_PMT_RGB_555	RGB 16-bit; 5 bits per colour
BR_PMT_RGB_565	RGB 16-bit; 5, 6, 5 bits per colour
BR_PMT_RGB_888	RGB 24-bit; 8 bits per pixel
BR_PMT_RGBX_888	RGB 32-bit; 8 bits per pixel
BR_PMT_RGBA_8888	RGB 32-bit; 8 bits per component

Note that TEXCONV can take a BRender .pix file and convert it to .tga format. You could save a scene generated in BRender:

```
BrPixelmapSave(back_buffer, "filename.pix")
```

then convert it to .tga format:

```
TEXCONV filename.pix -P std.pal -o filename.tga
```

MKSHADES

MKSHADES is a command-line tool that takes a source palette (usually a 1 by n BR_PMT_RGBX_888 pixelmap generated by TEXCONV) and generates an indexed shade table. The shade table is used when rendering into an 8-bit indexed pixelmap to perform shading across a lit textured material. Normally, the range of the shade table corresponds to the current hardware palette and the indices in the texture itself.

Command line format: mkshades {options}

Command line options are given below:

<palette>	Source BRender palette
[-o <shade-table>]	Output shade table pixelmap
[-d <dest-palette>]	Destination palette if different from source
[-n <num_shades>]	Number of shades to generate (default 64)
[-r <base>,<size>]	Range of colours in output palette (default 0,256)

MKRANGES

MKRANGES is a command line tool used to build custom ramped palettes.

Command line format: mkranges <input-text-file> <output-palette>

The input text file is read a line at a time. Lines starting with '#' are ignored. Each line describes a ramp in the palette. There a five groups of numbers. Each group is separated by white space. The numbers in each group are separated by commas. The groups are respectively:

Range (2 integers)	The starting index and size of the ramp
Ambient (3 integers)	The colour at the start of the ramp
Diffuse (3 integers)	The colour at the cut point in the ramp
Specular (3 integers)	The colour at the end of the ramp
Cut (1 float)	The point where the ramp cuts from diffuse to specular

The following is an example script that generates `std.pal`, the palette supplied with BRender:

```
stdpal.txt
# Standard palette
#

#   Range   Ambient     Diffuse       Specular      Cut
#
     0,64     0,0,0    147,147,147   255,255,255    0.75   #Grey
    64,32     0,0,0     60, 60,238   255,255,255    0.75   #Blue
    96,32     0,0,0     60,238, 60   255,255,255    0.75   #Green
   128,32     0,0,0     60,238,238   255,255,255    0.75   #Cyan
   160,32     0,0,0    238, 60, 60   255,255,255    0.75   #Red
   192,32     0,0,0    238, 60,238   255,255,255    0.75   #Magenta
   224,32     0,0,0    238,238, 60   255,255,255    0.75   #Yellow
```


PALJOIN

PALJOIN is a command line tool that allows you to construct a colour palette using sections of existing palettes.

Command line format: `paljoin <input-text-file> <output-palette>`

The input text file is read a line at a time. Lines beginning with `;` or # are treated as comments. Blank lines are ignored. A valid command lines contains a string identifying the source palette, followed by `index_base` and `range` values specifying which colours are to be copied from the source palette. An additional `index_base` value locates the copied colour range in the output palette.

The text file illustrated below could be used to combine, in a single palette, the first 128 colours in a strip palette (`strip.pal`) and the last 128 colours in a texture palette (`texture.pal`).

```
newpal.txt

#
# Source_Palette   Source_Index_Base  Range  Destination_Index_Base
strip.pal                  0           128               0
texture.pal              128           128             128
```

The following command line would generate the palette described in `newpal.txt`, and save it as `new.pal`:

```
paljoin newpal.txt new.pal
```

VIEWPAL

VIEWPAL is a command line tool that displays a palette on the screen.

Command line format: viewpal <input_palette>

All 256 colours are displayed, along with their respective palette indices (0–255). To view std.pal, enter the following command line:

```
viewpal std.pal
```

Importing Models into BRender

When importing a model into BRender from a 3D modeller, there are three classes of entity which you may need to import – texture maps, materials and geometry. Each needs to be imported separately, then re-united inside BRender. If you are only interested in importing model geometry, you can use GEOCONV (or the 3DS2BR –mod option) to import and convert .asc files. If you want to import materials associated with a model, you will use 3DS2BR. TEXCONV is used to import texture maps. If you are working in true colour mode, importing models is an reasonably straightforward process. Indexed colour is altogether more complicated due to the complexities of palette management. Indexed colour is discussed first below. A shorter section towards the end of the chapter covers true colour. This section is largely a review of ground already covered.

Working with 8-bit Colour

When working in 8-bit colour you will need to formulate a strategy for palette management. How you set up and manage your palette will depend on personal preference and the requirements of your application. You may be happy to use the palette supplied with BRender (std.pal) which offers a reasonably wide selection of colours. If all your materials reference imported texture maps you may wish to construct a palette and a shade table using only the colours in your texture maps. If you are not using texture maps you may wish to construct a custom palette containing the colours of the materials in your scenes. Alternatively, your palette may be divided into two parts, one part containing strip colours for your non-textured materials, the other containing texture colours. Some possible palette management strategies are considered below.

Using the supplied palette (std.pal) and shade table (shade.tab)

You may find that the palette and shade table supplied with BRender adequately represent your scenes. In this case you will need to ensure suitable `index_base` and `index_range` values are added to your material descriptions. You will also need to quantize your texture maps to `std.pal` using TEXCONV. When you quantize an imported texture map to a palette, TEXCONV takes each colour in the texture map in turn, and selects the colour from the palette that most closely approximates that colour. This new colour then replaces each instance of the original colour in the output pixelmap. Note that `std.pal` contains a range of colours sufficiently general to reproduce a reasonable representation of most scenes.

Colour Ramps in std.pal

Range	Colour
0–63	Grey
64–95	Blue
96–127	Green
128–159	Cyan
160–191	Red
192–223	Magenta
224–255	Yellow

The material script file illustrated below will display a blue shaded material when rendering using `std.pal`. Note the `index_base` and `index_range` values. You will need to ensure that the `index_base` and `index_range` values for all your materials (whether saved as script files or in `.mat` format) reference the appropriate colour in your palette.

cube8.mat

```
# A plain blue material

material = [
              identifier = "BLUE MATERIAL";
              ambient = 0.05;
              diffuse = 0.55;
              specular = 0.4;
              power = 10;
              flags = [light,smooth];
              index_base = 64;
              index_range = 31;
           ];
```

3DS2BR doesn't know how your hardware palette is set up. Materials imported using the -mat command line option contain dummy index_base and index_range values. Your application should set these fields to point to the correct palette indices when the material is loaded. Material script files generated by 3DS2BR for non-textured materials contain true colour information. You can use this data to add appropriate index_base and index_range values to the script file for use in 8-bit mode. Some examples will help clarify these issues.

Non-textured Materials

In Chapter 7 we extracted information describing the geometry and appearance of a model from the 3D Studio file duck.3ds using the following command line:

```
3DS2BR duck.3ds -nomatrix -mod duck.dat -scr duck.mat
```

For the purposes of this exercise, enter the following command line to extract material descriptions from duck.3ds and store them in duck.mat:

```
3DS2BR duck.3ds -scr duck.mat
```

The script file generated by 3DS2BR is shown below:

duck.mat

```
# BRender Material Script

material = [
              identifier = "BLACK PLASTIC";
              flags = [light,smooth];
              colour = [0,0,0];
              ambient = 0.000000;
              diffuse = 0.000000;
              specular = 1.000000;
              power = 69.309998;
           ];

material = [
              identifier = "YELLOW PLASTIC";
              flags = [light,smooth];
              colour = [202,179,52];
              ambient = 0.679216;
              diffuse = 0.679216;
              specular 0.741569;
              power = 23.770000;
           ];
```

This script file can be used unaltered in true colour mode, as indeed it was in BRTUTOR7.C described in Chapter 7. In order to use the materials described in duck.mat in 8-bit mode, we need to add appropriate index_base and

index_range values. It is likely that the exact colour will not be available in std.pal, due to the 256 colour limitation. It should be possible, however, to find a reasonable approximation.

duck8.mat

```
# BRender Material Script

material = [
              identifier = "BLACK PLASTIC";
              flags = [light,smooth];
              ambient = 0.000000;
              diffuse = 0.000000;
              specular = 1.000000;
              power = 69.309998;
              index_base = 0;
              index_range = 0;
           ];

material = [
              identifier = "YELLOW PLASTIC";
              flags = [light,smooth];
              ambient = 0.679216;
              diffuse = 0.679216;
              specular 0.741569;
              power = 23.770000;
              index_base = 224;
              index_range = 25;
           ];
```

An index_base value of 224 points to the beginning of the yellow colour ramp. The colour range for this material is 224–248, essentially the yellow colour ramp. When rendering BRender will select a shade of yellow from within this range according to the current lighting level. Note that true colour information is redundant in 8-bit mode and has been removed. This modified material script is stored on your Tutorial Programs disk as duck8.mat. Compile and run BRTUTR7B.C to display a revolving yellow duck in 8-bit mode.

Textured Materials

When 3DS2BR converts a textured material for use in BRender programs, the resultant material file (binary or script) will contain a dummy texture map and a dummy shade table. For instance, suppose the 3D Studio model being imported contains a material named 'FACE OF THE MOON', which references a colour (or texture) map called moonshot.gif. 3DS2BR will save a material file containing a BRender material called 'FACE OF THE MOON'. This material will reference a dummy

BRender pixelmap called 'moonshot' (note the .gif file extension has been stripped away) and a dummy shade table named 'shade'.

You could convert moonshot.gif to a BRender .pix file using TEXCONV. Your application would then need to load a file containing a pixelmap called 'moonshot' (it doesn't matter what the file is called) before loading and registering the file containing the material 'FACE OF THE MOON'. If you don't wish to use the texture map in moonshot.gif, simply replace 'moonshot' with the identifier of any previously loaded and registered texture map. If you are using shade.tab, the shade table supplied with BRender, you will need to change 'shade' to 'shade_table' in the colour_map field. This is necessary because shade.tab uses the string 'shade_table' as an identifier (an alternative would be to change the identifier in shade.tab to 'shade' using TEXCONV -n). A further example should help illuminate the above discussion.

The following command line imports a model of a fork and its associated material from the standard 3D Studio model description file fork.3ds:

```
3DS2BR fork.3ds -mod fork.dat -scr fork.mat
```

Both fork.3ds and the file containing its associated texture map, refmap.gif, can be found on your Tutorial Programs disk. You will need to enter the command lines in this example in order to display the imported fork model. Mesh data describing the geometry of the converted model is stored in fork.dat. Enter the command line:

```
geoconv fork.dat -l
```

to display the following information on this newly created file:

```
GEOCONV 1.24 Copyright (C) 1994-1995 by Argonaut Technologies Limited
Models: 1
       Verts   Faces  Edges  Fgrps  Vgrps  Sort  DOTQ   Radius
 fork   858     812    1648    1      1     858  ----   3.05932
Materials: 1
 CHROME GIFMAP
```

Scale the model (set radius to 1) and translate it to the origin (in its parent's co-ordinate space) as follows:

```
geoconv fork.dat -c -n -o fork.dat
```

BRTUTR7B.C

```
/* Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a Revolving Yellow Duck (8-bit).
 */

#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"

int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
   br_actor *world, *observer, *duck;
   int i;
   br_material *mats[10]; /*for storing pointers to material descriptions*/

/******** Initialise BRender and Graphics Hardware **************/

   BrBegin();

   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                       :          :          :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);

   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database **********************/

   world = BrActorAllocate(BR_ACTOR_NONE,NULL);
   BrLightEnable(BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL)));

   /*
    * Load and Position Camera
    */
   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
   observer->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                        BR_SCALAR(5.0));

   /*
    * Load and Apply Duck Materials
    */
   i = BrFmtScriptMaterialLoadMany("duck8.mat",mats,BR_ASIZE(mats));
   BrMaterialAddMany(mats,i);
```

```
   /*
    * Load and Position Duck Model
    */
   duck = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
   duck->model = BrModelLoad("duck.dat");
   BrModelAdd(duck->model);
   duck->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34RotateX(&duck->t.t.mat,BR_ANGLE_DEG(30));

/********************** Animation Loop ***********************/

   for(i=0; i < 200; i++)
      BrPixelmapFill(back_buffer,0);
      BrPixelmapFill(depth_buffer,0xFFFFFFFF);
      BrZbSceneRender(world,observer,back_buffer,depth_buffer);
      BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
      BrMatrix34PostRotateX(&duck->t.t.mat,BR_ANGLE_DEG(2.0));
   }

   /*
    * Close down
    */
   BrPixelmapFree(depth_buffer);
   BrPixelmapFree(back_buffer);
   BrZbEnd();

                        :          :          :

   BrEnd();
   return 0;
}
```

BRTutr7b.c

The model described in `fork.dat` can now be used in any BRender program. The material file, `fork.mat`, generated by 3DS2BR is given below:

fork.mat

```
# BRender Material Script
#

material = [
              identifier = "CHROME GIFMAP";
              flags = [light,smooth,environment];
              ambient = 0.000000;
              diffuse = 0.000000;
              specular =0.992941;
              power = 69.340004;
              index_base = 64;
              index_range = 31;
              colour_map = "REFMAP";
              index_shade = "shade";
           ];
```

Note that the identifier used for BRender's standard shade table is `'shade_table'`, and not `'shade'`. The script file, `fork.mat`, will need to be edited accordingly.

With texture mapped materials, `index_base` is used as a row offset into a shade table. `index_range` determines the number of shades (or lighting levels) available, starting at `index_base`. The standard shade table has 64 rows (this is also the default number of rows generated by MKSHADES, the utility used to create shade tables discussed later in this chapter). Not surprisingly, then, the default `index_base` is 0 and the default `index_range` is 63.

Unless you are using a shade table with non standard dimensions (or you want to restrict the range of shades available), you can omit the `index_base` and `index_range` fields completely. Alternatively, specify the default values, 0 and 63.

The revised script file is given below:

```
fork.mat

# BRender Material Script

material = [
              identifier = "CHROME GIFMAP";
              flags = [light,smooth,environment];
              ambient = 0.000000;
              diffuse = 0.000000;
              specular =0.992941;
              power = 69.340004;
              colour_map = "REFMAP";
              index_shade = "shade_table";
           ];
```

This material references a dummy texture map called 'REFMAP'. In order to replicate the image created in 3D Studio, the original `.gif` file needs to be converted to BRender `.pix` format:

```
TEXCONV refmap.gif -n REFMAP -c BR_PMT_INDEX_8 -Q std.pal -o
                refmap.pix
```

The following is displayed:

```
TEXCONV 1.6 Copyright (C) 1994 by Argonaut Technologies Limited
Loaded 'refmap.gif' as BR_PMT_INDEX_8(320,200)
Palette 'refmap.gif' BR_PMT_RGBX_888(256)
'refmap.gif' assigned new identifier 'REFMAP'
Converted 'REFMAP' BR_PMT_INDEX_8 (8 bit) to BR_PMT_INDEX_8 (8 bit)
Quantizing 'REFMAP' to palette 'Palette' using 256 colours from the
range 0-255
Output pixelmap 'refmap.pix'
```

The `-n` option ensures that the identifier, 'REFMAP', is the same as that used in the material script file (alternatively, we could have changed the line in the script file specifying the texture map to `colour_map = refmap.gif`)

The `-c` option converts the input file to the required BRender pixelmap type (where necessary).

The `-Q` option quantizes the colours in the texture map to the specified palette.

The `-o` option generates an output file in accordance with the specified command line options.

Compile and run `BRTUTOR9.C` to display a chrome-textured fork. Please note that this program will not work unless the command lines specified above have been entered. Only `fork.3ds` and `refmap.gif` are included on the Tutorial Programs disk. You will need to run 3DS2BR to generate `fork.dat` and `fork.mat`,

GEOCONV to convert `fork.dat`, and TEXCONV to convert and quantize `refmap.gif`. You will also need to edit `fork.mat` as indicated.

`BRTUTOR9.C` is almost identical to `BRTUTR8B.C`, encountered in Chapter 7. A fork model is loaded instead of the duck model of `BRTUTR8B.C`, and the `'gold'` textured material is replaced with a material that references the chrome texture in `refmap.pix`.

To recap:

To import 3D Studio models and associated materials:

- Run 3DS2BR to generate BRender files (`.dat`, `.mat`, `.log` etc.)
  e.g. `3DS2BR fork.3ds -mod fork.dat -scr fork.mat`
- Run GEOCONV to tailor model geometry to your BRender program:
  e.g. `geoconv fork.dat -c -n -o fork.dat`
- Run TEXCONV to convert texture maps to BRender format and to quantize them to your palette (`std.pal` if using the palette supplied with BRender):
  e.g. `texconv refmap.gif -n REFMAP -c BR_PMT_INDEX_8 -Q std.pal`
  `-o refmap.pix`
- For non-textured materials, ensure that appropriate `index_base` and `index_range` values are added. For textured materials, ensure that the texture map references used in material descriptions match the identifiers stored in the relevant `.pix` files. Also, ensure that shade table references match the appropriate shade table identifier (`shade_table` in the case of the standard shade table). Ensure that appropriate `index_base` and `index_range` values are entered if using a shade table with non-standard dimensions. Otherwise accept the default values.

Creating a Texture Palette

If all your materials reference texture maps, you may wish to construct a palette and a shade table using only the colours in these texture maps. TEXCONV can be used to generate an optimal palette for a number of texture maps. This palette could then be defined as the texture palette, to be used for all textures.

Begin by assembling your textures and listing them in a text file (to keep command line length within manageable proportions). Suppose your project uses the following textures, listed in a text file `textures`:

```
--------------------------- textures
gold.gif
refmap.gif
buttons.gif
canada.gif
cloud.gif
```

```
cruiser.gif
dash.gif
graymarb.gif
```

Note that all the above textures are standard 3D Studio `.gif` files. You may want to copy these files (or your own textures) to your current directory before proceeding, so you can work through this example.

It is worth noting that colour 0 is **always** transparent for textures in 8-bit colour mode. For this reason it is advisable to quantize and remap between colours 1 and 255.

The following command line loads the specified pixelmaps, converts them to BRender 8-bit format, quantizes and remaps to 255 colours in the range 1–255 (this avoids reassigning colour 0, which would be treated as transparent by the renderer), and creates an output palette, `texture.pal`, containing these colours.

```
texconv @textures -c BR_PMT_INDEX_8 -R 1,255 -O palette -o
            texture.pal
```

In short, TEXCONV examines the specified textures maps and comes up with what, in its estimation, is the best combination of 255 colours that could be used to reproduce these textures. Note that you will still need to run TEXCONV to generate a BRender `.pix` file for each or your textures.

Enter the following command line to display the palette:

```
viewpal texture.pal
```

VIEWPAL displays all 256 colours in a palette, along with their respective palette indices (0–255). The next step is to create a shade table based on this palette.

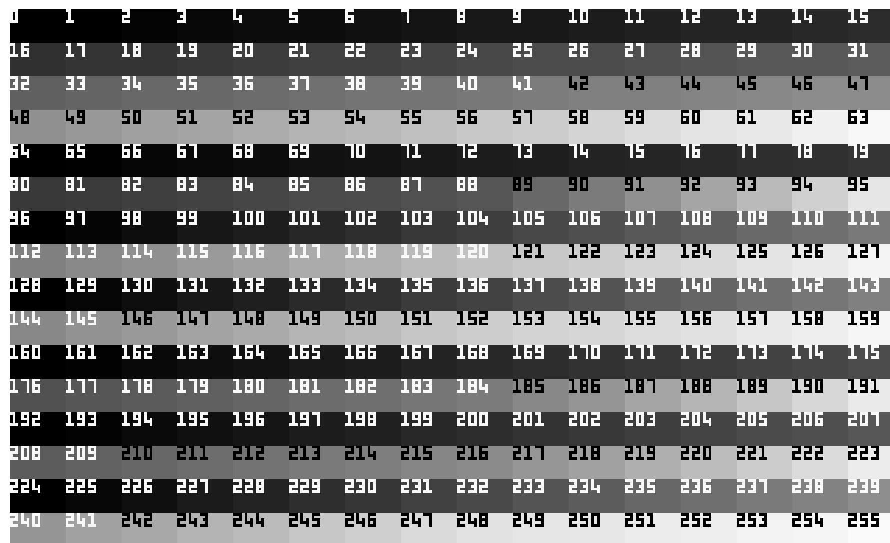


Figure 41 A palette displayed using VIEWPAL

BRTUTOR9.C

```
/* Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
 * Program to Display a Chrome-Textured Fork (8-bit)
 */

#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"

int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette, *shade;
   br_actor *world, *observer, *fork;
   br_pixelmap *chrome_pm;
   int i;

/****** Initialise Renderer and Set Screen Resolution ********/

   BrBegin();

   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                     :          :          :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);

   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

   /*
    * Load Shade Table
    */
   shade = BrPixelmapLoad("shade.tab");
   if (shade==NULL)
        BR_ERROR0("Couldn't load shade.tab");
   BrTableAdd(shade);

/*************** Build the World Database **********************/

   world = BrActorAllocate(BR_ACTOR_NONE,NULL)
   BrLightEnable(BrActorAdd(world,BrActorAllocate( BR_ACTOR_LIGHT,NULL)));

   /*
    * Load and Position Camera
    */
   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
   observer->t.type = BR_TRANSFORM_MATRIX34;
```

```
   BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                          BR_SCALAR(5.0));

   /*
    * Load and Register chrome Texture
    */
   chrome_pm = BrPixelmapLoad("refmap.pix");
   if (chrome_pm==NULL)
      BR_ERROR0("Couldn't load refmap.pix");
   BrMapAdd(chrome_pm);
   /*
    * Load and Apply fork Material
    */
   BrMaterialAdd(BrFmtScriptMaterialLoad("fork.mat"));

   /*
    * Load and Position fork Actor
    */
   fork = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
   fork->model = BrModelLoad("fork.dat");
   BrModelAdd(fork->model);
   BrModelApplyMap(fork->model,BR_APPLYMAP_PLANE,NULL);
   fork->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34RotateX(&fork->t.t.mat,BR_ANGLE_DEG(30));
   fork->material = BrMaterialFind("CHROME GIFMAP");

/********************* Animation Loop **********************/

   for(i=0; i < 200; i++) {
            BrPixelmapFill(back_buffer,0);
            BrPixelmapFill(depth_buffer,0xFFFFFFFF);
            BrZbSceneRender(world,observer,back_buffer,depth_buffer);
            BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
            BrMatrix34PostRotateX(&fork->t.t.mat,BR_ANGLE_DEG(2.0));
   }

   /*
    * Close down
    */
   BrPixelmapFree(depth_buffer);
   BrPixelmapFree(back_buffer);
   BrZbEnd();

                    :         :         :

   BrEnd();
   return 0;
}
```

BRTUTOR9.C

The following command line generates a shade table, `newshade.tab` from the source palette `texture.pal`:

```
mkshades texture.pal -r 1,255 -o newshade.tab
```

Enter the following command line to display the shade table:

```
texconv -I pix newshade.tab -P texture.pal -c BR_PMT_INDEX_8 -v
```

`newshade.tab` is an array of 255 columns by 64 rows. Each column indexes a colour in `texture.pal`. Each row contains indices to shades of its associated colour representing 64 lighting levels.

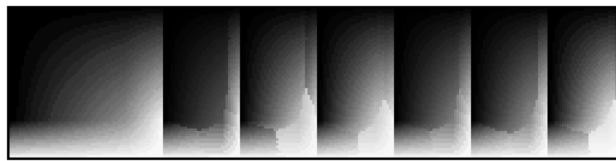


Figure 42 A shade table displayed using TEXCONV

Note that the identifier `shade_table` has been assigned to `newshade.tab`. This could be changed using the `-n` option:

```
texconv -I pix newshade.tab -n newname -O pix -o newshade.tab
```


To recap:

To create a custom texture palette and shade table:

- Assemble your textures and list them in a text file.
- Run TEXCONV to generate a palette based on your textures:
  `texconv @textures -c BR_PMT_INDEX_8 -R 1,255 -O palette -o texture.pal`
- For each of your textures, run TEXCONV to convert them to BRender format and to quantize them to your palette:
  e.g. `texconv gold.gif -c BR_PMT_INDEX_8 -Q texture.pal -o gold.pix`
- Run MKSHADES to generate a shade table from the texture palette:
  e.g. `mkshades texture.pal -r 1,255 -o newshade.tab`

Creating a Custom Palette With Strip Colours

If you are not using texture maps, you may wish to build a custom palette containing the colours of the materials in your scenes using MKRANGES. MKRANGES is a utility for building ramped, or strip, palettes from information contained in a text file.

For imported materials, material colours are specified in the text files generated using the 3DS2BR `-mod` and `-scr` options. If you are intending to construct a palette based on the colours of your materials, compile a list of colours from your material script files or log files (or your material data structure if not using material script files). Then use

this information to create a text file for MKRANGES. Remember to ensure appropriate `index_base` and `index_range` values have been specified for all your materials.

Supposing you have compiled the following list of colours from your material script files or log files:

Colour	RGB Value
Aqua Glaze	0,162,160
Green Plastic	65,204,77
Plum Plastic	130,0,91
Old Gold	152,82,0
Orange	166,66,0
White Paint	204,204,189
Red Plastic	166,0,0
Blue Metallic	39,42,79

You could then prepare a text file for MKRANGES as follows:

strippal.txt

```
# Range   Ambient   Diffuse     Specular    Cut
#
    0,32   0,0,0    0,162,160  255,255,255  0.75 #Aqua Glaze
   32,32   0,0,0    65,204,77  255,255,255  0.75 #Green
                                                  Plastic
   64,32   0,0,0     130,0,91  255,255,255  0.75 #Plum Plastic
   96,32   0,0,0     152,82,0  255,255,255  0.75 #Old Gold
  128,32   0,0,0     166,66,0  255,255,255  0.75 #Orange
  160,32   0,0,0     204,204,  255,255,255  0.75 #White Paint
                          189
  192,32   0,0,0      166,0,0  255,255,255  0.75 #Red Plastic
  224,32   0,0,0     39,42,79  255,255,255  0.75 #Blue
                                                  Metallic
```

Then run MKRANGES to create your custom palette:

```
mkranges strippal.txt strip.pal
```

The text file '`strippal.txt`' can be found on your Tutorial Programs disk. Having created a new palette, you can view it by entering:

```
texconv -I pix strip.pal -c BR_PMT_INDEX_8 -v
```

or:

```
viewpal strip.pal
```

You could experiment by loading strip.pal in BRTUTR7B.C above, instead of std.pal, and varying the index_base and index_range values to select the newly generated colours.

Creating a Custom Palette With Strip Colours and Texture Colours

If you are using both textured and non-textured materials, your palette may be divided into two parts, one part containing strip colours for your non-textured materials, the other containing texture colours. To achieve this you will need to create separate strip colour and texture palettes using the utilities provided and subsequently combine them using PALJOIN.

Suppose your project uses the textures in textures above, and the colours specified in strippal.txt. Let's assume that you have decided to divide your palette equally between strip colours and textures.

Project Palette

```
-----------------------------------
| strip colours | texture colours |
-----------------------------------
|<----0-127---->|<----128-255---->|
```

1. You could begin by generating a texture palette and shade table:

Run TEXCONV to generate a texture palette:

```
texconv @textures -c BR_PMT_INDEX_8 -R 128,128 -O palette
                 -o texture.pal
```

For each of the texture maps in textures, quantize and remap to the specified range: e.g. texconv gold.gif -c BR_PMT_INDEX_8 -Q texture.pal -R 128, 128
-O pixelmap -o gold.pix

Run MKSHADES to create a shade table from the texture palette:

```
mkshades texture.pal -r 128,128 -o newshade.tab
```

2. Generate a strip palette using only the first 128 colours:

Edit strippal.txt as follows:

strippal.txt

```
#  Range   Ambien     Diffuse      Specular     Cut
             t
#
     0,16   0,0,0     0,162,160  255,255,255   0.75   #Aqua Glaze
    16,16   0,0,0     65,204,77  255,255,255   0.75   #Green Plastic
    32,16   0,0,0      130,0,91  255,255,255   0.75   #Plum Plastic
```

```
   48,16   0,0,0      152,82,0  255,255,255  0.75  #Old Gold
   64,16   0,0,0      166,66,0  255,255,255  0.75  #Orange
   80,16   0,0,0   204,204,189  255,255,255  0.75  #White Paint
   96,16   0,0,0       166,0,0  255,255,255  0.75  #Red Plastic
  112,16   0,0,0      39,42,79  255,255,255  0.75  #Blue Metallic
```

Run MKRANGES to generate a strip palette:

```
MKRANGES strippal.txt strip.pal
```

The strip colours are located in the first 128 locations in `strip.pal`. The remaining locations are 'empty'. Texture colours occupy the second half of `texture.pal`. This can be visually verified by running VIEWPAL to view the palettes on the screen. Enter:

```
viewpal strip.pal
```

to display the strip palette. Enter:

```
viewpal texture.pal
```

to display the texture palette.

3. Combine your texture and strip palettes into a single palette.

Generate the following text file to combine in a single palette the first 128 colours in the strip palette and the last 128 colours in the texture palette.

newpal.txt

```
# Source_Palette Source_Index_Bas Range Destination_Index_Base
                 e
   strip.pal             0         128             0
   texture.pal          128        128            128
```

Then run the following command line to generate the output palette, `new.pal`, containing all the required colours:

```
paljoin newpal.txt new.pal
```

Enter:

```
viewpal new.pal
```

to view `new.pal`.

The newly generated palette will need to be assigned an identifier:

```
texconv -I pix new.pal -n palette -O pix -o new.pal
```

The best way to handle indexed colour is to pre-define a set of colours for a given project, use MKANGES to generate a strip palette using these colours, then define a range of palette-indexed materials based on these colours. Similarly with texture maps – prepare your textures, then run TEXCONV to generate an optimal texture palette and MKSHADES to generate a shade table based on the optimised palette.

Thereafter, use only the pre-defined materials and textures. This involves much less work than starting with your artwork and trying to hand-convert and optimise the entire project's artwork once it is completed.

Working with True Colour

If you are working in true colour mode, you won't need to use MKSHADES or MKRANGES, since there are no palettes or shade tables to worry about. 3DS2BR can be used to import model geometry and materials. TEXCONV is used to import textures. If you are only interested in a model's geometry, GEOCONV can be used to import `.asc` files etc. Much of what you need to know to import model geometry, materials and textures has already been covered, either above or in Chapters 6 and 7. What follows, therefore, is a brief review.

Importing Model Geometry

Enter the following command line to import the 3D Studio `.asc` file `fork.asc` (`fork.asc` can be found on your Tutorial Programs disk),

```
geoconv fork.asc -c -n -o fork.dat
```

Importing Geometry and Materials

1. Use 3DS2BR to convert 3D Studio `.3DS` files to BRender format. The following command line extracts data describing model geometry from the 3D Studio file `duck.3ds` and stores it in `duck.dat`, and creates a material script file describing the converted materials:

   ```
   3DS2BR duck.3ds -nomatrix -mod duck.dat -scr duck.scr
   ```

2. Use GEOCONV to convert model geometry to a format suitable for BRender. The following command line collapses data into a single model, centres the model at the origin and normalises it to fit a sphere of radius 1:

   ```
   geoconv duck.dat -m -c -n -o duck.dat
   ```

Importing Texture Maps

Use TEXCONV to convert a pixelmap to BRender pixelmap format. The following command line generates a 15-bit BRender pixelmap from a `.gif` file:

```
texconv gold.gif -c BR_PMT_RGB_555 -o gold15.pix
```

A Worked Example

Let's work through an example.

Suppose you want to import a model whose geometry is described in the file `fork.asc`. Suppose further that the material you want to assign to this model is called GRIDMAP, and is defined in the 3D Studio file `light.3ds`. GRIDMAP is a textured material that references a texture map called TILE0011 in the file `tile0011.tga`. Note that `fork.asc`, `light.3ds` and `tile0011.tga` are all on your Tutorial Programs disk.

Start by importing the model geometry:

```
geoconv fork.asc -c -n -o fork.dat
```

Then import the material and store it in a material script file:

```
3DS2BR light.3ds -scr fork.mat
```

Finally, import the texture map:

```
texconv tile0011.tga -n TILE0011 -c BR_PMT_RGB_555 -o tile.pix
```

The `-n` option sets the identifier to TILE0011 to be consistent with that contained in the `colour_map` field in the recently generated script file `fork.mat`. The `-c` and `-o` options generate a `.pix` file in BRender 15-bit format.

The material script file generated by 3DS2BR is shown below:

fork.mat

```
# BRender Material Script
#

material = [
              identifier = "WHITE PLASTIC";
              flags = [light,smooth];
              colour = [206,206,206];
              ambient = 0.807843;
              diffuse = 0.807843;
              specular = 1.000000;
              power = 52.480000;
           ];
material = [
              identifier = "GRIDMAP";
              flags = [light,smooth];
              colour = [0,0,0];
              ambient = 0.000000;
              diffuse = 0.000000;
              power = 1.000000;
              index_base = 64;
              index_range = 31;
              colour_map = "TILE0011";
              index_shade = "shade";
           ];
```

8

BRENDER TOOLS

BRTUTR10.C

```
/*
 * Copyright (c) 1996 Argonaut Technologies Limited. All rights reserved.
126
126
#include <stddef.h>
#include <stdio.h>
#include "brender.h"
#include "dosio.h"

int main(int argc, char **argv)
{
   br_pixelmap *screen_buffer, *back_buffer, *depth_buffer, *palette;
   br_pixelmap *tile_pm;
   br_actor *world, *observer, *fork;
   int i;
   br_material *mats[10];

/******** Initialise Renderer and Set Screen Resolution *********/

   BrBegin();

   /*
    * Initialise screen buffer and set up CLUT (ignored in true colour)
    */

                      :         :         :

   BrZbBegin(screen_buffer->type, BR_PMT_DEPTH_16);

   back_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_OFFSCREEN);
   depth_buffer = BrPixelmapMatch(screen_buffer,BR_PMMATCH_DEPTH_16);

/*************** Build the World Database *********************/

   world = BrActorAllocate(BR_ACTOR_NONE,NULL)
   BrLightEnable(BrActorAdd(world,BrActorAllocate(BR_ACTOR_LIGHT,NULL)));

   /*
    * Load and Position Camera
    */
   observer = BrActorAdd(world,BrActorAllocate(BR_ACTOR_CAMERA,NULL));
   observer->t.type = BR_TRANSFORM_MATRIX34;
   BrMatrix34Translate(&observer->t.t.mat,BR_SCALAR(0.0),BR_SCALAR(0.0),
                                                         BR_SCALAR(5.0));

   /*
    * Load and Register TILE0011 Texture
    */
   tile_pm = BrPixelmapLoad("tile.pix");
   if (tile_pm==NULL)
        BR_ERROR0("Couldn't load tile.pix");
```

```
   BrMapAdd(tile_pm);

   /*
    * Load and Apply fork Material
    */
   i = BrFmtScriptMaterialLoadMany("fork.mat",mats,BR_ASIZE(mats));
   BrMaterialAddMany(mats,i);

   /*
    * Load and Position fork Actor
    */
   fork = BrActorAdd(world,BrActorAllocate(BR_ACTOR_MODEL,NULL));
   fork->model = BrModelLoad("fork.dat")
   BrModelAdd(fork->model);
   BrModelApplyMap(fork->model,BR_APPLYMAP_PLANE,NULL);
   fork->t.type = BR_TRANSFORM_MATRIX34;

   /*
    * Assign fork Material
    */
   fork->material = BrMaterialFind("GRIDMAP");

/********************** Animation Loop **********************/

   for(i=0; i < 200; i++)  {
           BrPixelmapFill(back_buffer,0);
           BrPixelmapFill(depth_buffer,0xFFFFFFFF);

           BrZbSceneRender(world,observer,back_buffer,depth_buffer);
           BrPixelmapDoubleBuffer(screen_buffer,back_buffer);
           BrMatrix34PostRotateX(&fork->t.t.mat,BR_ANGLE_DEG(2.0));
   }

   /*
    * Close down
    */
   BrPixelmapFree(depth_buffer);
   BrPixelmapFree(back_buffer);
   BrZbEnd();

                       :         :         :

   BrEnd();
   return 0;
}
```

BRTUTR10.C

The material WHITE PLASTIC, imported from `light.3ds` along with GRIDMAP, could be deleted. The `index_base`, `index_range` and `index_shade` fields could also be deleted, since we are working in non-indexed mode. The colour field is also irrelevant for a textured material. If your version of BRender does not support lit textures, the ambient, diffuse and power fields could also be eliminated (refer to your installation guide for details). Finally, your version of BRender might not support smooth shading, in which case this flag should be reset. If you were running BRender x86, Version 1.2, for example, which doesn't support light or smooth shaded textures in true colour mode, the following minimalist script would suffice:

fork.mat

```
# BRender Material Script
#
material = [
              identifier = "GRIDMAP";
              flags = [];
              colour_map = "TILE0011";
           ];
```


We now have:

- a file, `fork.dat`, describing model geometry
- a material, GRIDMAP, described in the material script file `fork.mat`
- GRIDMAP references a texture map `TILE0011`, contained in `tile.pix`.

Note that the model described in `fork.asc` was saved without texture co-ordinates. It was therefore necessary to call `BrModelApplyMap` to generate texture co-ordinates for the model's vertices:

```
BrModelApplyMap(fork->model,BR_APPLYMAP_PLANE,NULL);
```

Compile and run `BRTUTR10.C` to display a texture mapped fork. Note that only the source files `fork.asc`, `light.3ds` and `tile0011.tga` are included on your Tutorial Programs disk. You will need to run the command lines described above to generate `fork.dat`, `fork.mat` and `tile.pix`, before attempting to compile `BRTUTR10.C`.

8

BRENDER TOOLS