Graphics and Visualisation

EE563

Dr. Derek Molloy and Dr. Robert Sadleir

Contents

Chapter 1

Introduction

1.1 Introduction

This chapter introduces some basic concepts in the area of computer graphics. The material is initially limited to a discussion of the 2D imaging and graphics functionality that is provided by Java. The concept of 3D content generation is subsequently introduced using VRML. A range of practical examples are provided to illustrate the various concepts that are discussed throughout this chapter. The material presented here is intended to act as a basis for a more in depth discussion of 3D computer graphics that will take place in subsequent chapters.

1.2 Java 2D Image and Graphics Support

The Java programming language from Sun Microsystems provides a high level of support for 2D graphics. The graphics functionality provided by Java can be divided into three main categories:

- **Interpretation -** reading various graphics file formats
- **Manipulation -** altering/processing graphical data
- **Display -** rendering graphics to a particular output device

There have been a number of incremental developments in relation to Java imaging since the initial release of the Java Developers Kit (JDK).

- **Java AWT Imaging -** Supports basic file formats, colour spaces, drawing capabilities and pixel level image process.
- **Java 2D Imaging -** Introduces more advanced support for colour spaces, drawing and image processing operations as well as providing support for printers.
- **Java Advanced Imaging -** Vastly increases the number of graphics file formats supported and provides a large library of image processing operations.

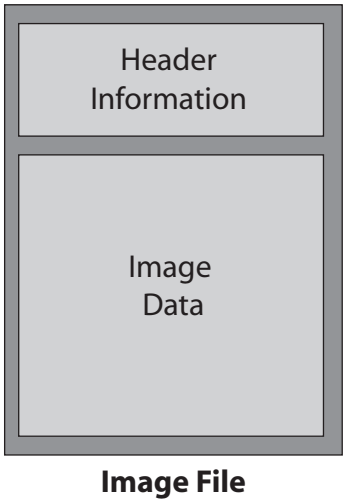


Figure 1.1: An illustration of the format of a standard image file that consists of separate header information and image data.

1.2.1 Image file formats

Java supports a variety of 2D image file formats through its various APIs. A file format designed specifically for representing graphical image data. An image file usually consists of header information and image data (either bitmap or vector). The format of a typical image file is outlined below and illustrated in Figure 1.1.

- Header information
  - Width in pixels
  - Height in pixels
  - Compression scheme
  - Colour Palette
  - Image Resolution
- Image data
  - Encoded and or compressed pixel data
  - Stored in a raster fashion as:
    - Several colour intensity planes
    - A plane of pixels with several colour components

There are a wide variety of image file formats that are available. Some of these are listed below. It is important to note that Java does not support all of these formats and in certain cases custom image file loaders must be developed.

- Windows Bitmap (BMP)
  - Developed by Microsoft
  - Supports RLE encoding
  - Maximum pixel depth of 32 bits

- Graphics Interchange Format (GIF)
  - 256 colour palette
  - Uses Lemple-Zev-Welch (LZW) compression
- Joint Photographic Experts Group (JPEG)
  - Supports 16.7 million colours
  - Compression based on the Discrete Cosine Transform (DCT)
- Digital Imaging and Communication is Medicine (DICOM)
  - Stores images from a large number of modalities
  - Header contains information about the patient, exam & image data
  - DICOM was developed by the American College of Radiology (ACR) and the National Electrical Manufactures Association (NEMA)

Reference Text:

- James D. Murray and William vanRyper "Encyclopedia of graphics file formats" O'Reilly & Associates, 2nd edition (June 1996) ISBN 1-56592-161-5

1.3 Java AWT Imaging

Java AWT (Advanced Windowing Toolkit) imaging was primarily designed to facilitate the display of images in an Internet browser based environment. Using this model the transfer of image data is based on the producer/consumer (push) model. Image data can be loaded from the local file system as well as from the internet. Java AWT imaging provides read only support for GIF and JPG images.

1.3.1 Graphics Support

Basic drawing is supported by the `Graphics` class. The types of data can be drawn using this class are text, basic shapes and images. The `Graphics` class can be used for drawing to onscreen GUI components or off-screen images. Some examples of the methods provided by the `Graphics` class include:

- `void setColor(Color c)`
  Sets the drawing colour of the associated `Graphics` object to the specified colour. All subsequent drawing operations associated with the `Graphics` object use this colour.
- `void drawOval(int x, int y, int width, int height)`
  Draws an ellipse or circle with the specified dimensions at the specified (x, y) coordinates.
- `void drawRect(int x, int y, int width, int height)`
  Draw a rectangle with the specified dimensions at the specified (x, y) coordinates.

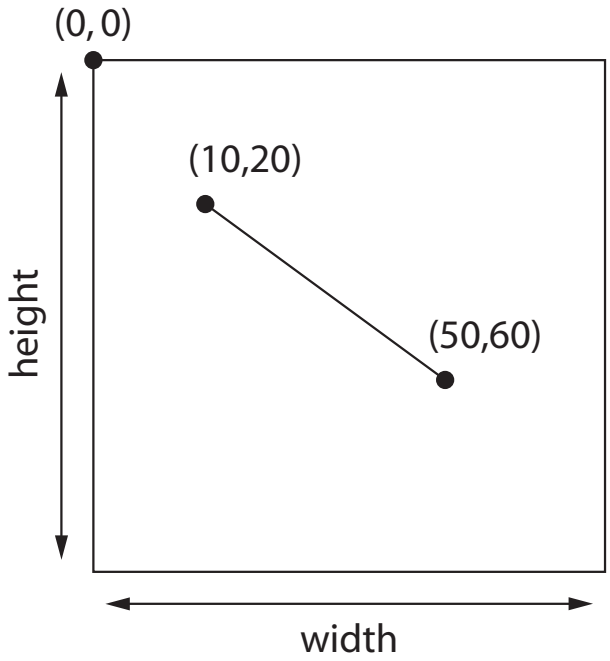


Figure 1.2: An illustration of the 2D graphics coordinates system used by Java.

- `void drawString(String str, int x, int y)`
  Draws the specified string at the specified (x, y) coordinates. The font used for drawing strings can be specified using the `setFont()` method of the `Graphics` class.

- `boolean drawImage(Image img, int x, int y, ImageObserver obs)`
  Draws the specified image at the specified (x, y) coordinates. This method return true if the specified image has completely loaded and false otherwise. The `ImageObserver` argument is notified once the image data becomes available.

Note: These methods use 2D (x, y) coordinates that related to points in the 2D Java coordinate system. This coordinate system is illustrated in Figure 1.2.

Note: In each of the methods that deal with drawing the specified (x, y) coordinates indicate the location of the top left hand corner of the object being drawn.

An example of an application that uses the `Graphics` class is listed below. The program draws a series of lines, using a for loop, to create a pattern.

```
0  import java.awt.*;

   public class GraphicsExample extends Canvas
   {
     public static void main(String args[]){new GraphicsExample();}
5
     Frame frame = new Frame();

     public GraphicsExample()
     {
10     // Initialise the display frame
```

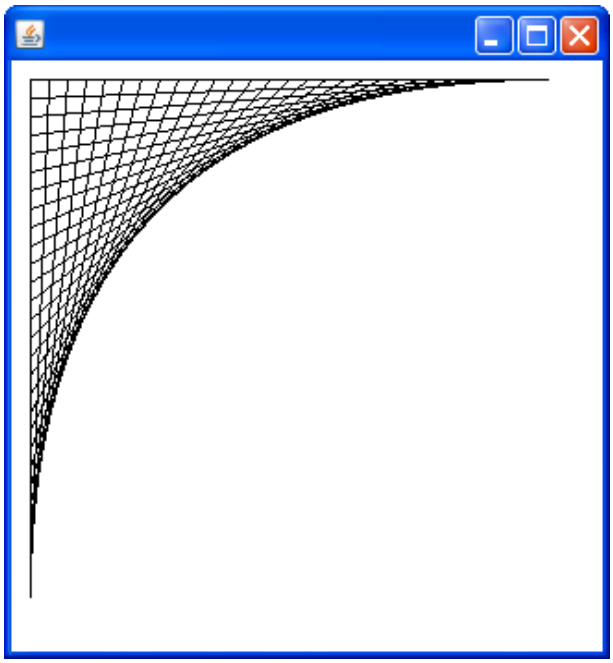


Figure 1.3: The pattern generated by the `GraphicsExample` application.

```
    frame.setLayout(new BorderLayout());
    setSize(320,320);
    frame.add(this, BorderLayout.CENTER);
    frame.pack();
    frame.setVisible(true);
  }

  public void paint(Graphics g)
  {
    g.setColor(Color.BLACK);

    // Draw a series of black lines
    for(int i=10; i<300; i+=10)
      g.drawLine(10,i,300-i,10);
  }
}
```

The class defined in this program extends the `Canvas` class and consequently represents a graphical user interface component that can be drawn upon. A `Frame` object is created in the constructor to display the `Canvas` and the `paint()` method of the `Canvas` object is overwritten to draw the desired pattern. The output generated by this program is illustrated in Figure 1.3.

1.3.2 Image Support

Bitmapped image data is represented using the `Image` class in the AWT imaging model. This class is essentially a container for image data and does not provide direct access to the pixel information. The `Image` class is an abstract class and consequently, an instance of this class cannot be constructed. However, it is possible

to create an image from a particular source (e.g. a URL or a local file path) using the the `createImage()` method of the `Toolkit` class. The `Image` class provides a limited number of methods, for example:

- `int getWidth(ImageObserver observer)`
  Returns the width of the image in pixels. If the image is not yet loaded then this method returns -1. The observer argument is a reference to object waiting for the image to be loaded.

- `int getHeight(ImageObserver observer)`
  Operates in a similar manner to the `getWidth()` method with the exception that this method returns the height of the image in pixels.

- `Graphics getGraphics()`
  Creates a graphics context for drawing to an off-screen image. This method can only be called for off-screen images. Note that an off-screen image is created using the `CreateImage(int w, int h)` method of the Component class.

The following example demonstrates how an instance of an `Image` object can be created and displayed using Java AWT imaging.

```
0  import java.awt.*;

   public class ImageLoadExample extends GraphicsExample
   {
     public static void main(String args[]){new ImageLoadExample();}
5
     Image i;

     public ImageLoadExample()
     {
10     super();

       // Load the image and resize the frame
       i = Toolkit.getDefaultToolkit().createImage("image.jpg");
       waitForImage(i);
15     int width = i.getWidth(this);
       int height = i.getHeight(this);
       setPreferredSize(new Dimension(width, height));
       frame.pack();
     }
20   public void paint(Graphics g)
     {
       // Paint the image on the canvas
       g.drawImage(i, 0, 0, this);
     }
25
     // Wait for image 'i' to be loaded
     public void waitForImage(Image i)
     {
       try{
30       MediaTracker tracker = new MediaTracker(this);
         tracker.addImage(i,0);
```

Figure 1.4: An illustration of the image loaded using the `ImageLoadExample` program.

```
        tracker.waitForAll();
        }catch(InterruptedException e){System.out.println(e);}
    }
}
```

This application loads the required image data in the constructor and displays the loaded image by overwriting the `paint()` method. It should be noted that this class extends the `GraphicsExample` class. The output generated by this program is illustrated in Figure 1.4.

Creating an image does not guarantee that the image will be immediately loaded into memory. However, it is possible to wait for the required image data to be loaded into memory using a `MediaTracker` object in order to ensure that the image data is available when required.

Note: An alternative to using the `MediaTracker` class would be monitor the dimensions of the image using the `getWidth()` or `getHeight()` methods. If the image data is not fully loaded then then these methods will return -1. When the image data is loaded then these methods will return the relevant image dimensions.

1.3.3 Image Processing Support

It is possible to get access to the pixel data indirectly using the `PixelGrabber` class. It is possible to reproduce an image from pixel data using the `MemorySourceImage` class in conjunction with the `createImage()` method of the `Toolkit` class. The following program performs the pixel level invert operation using this mechanism.

```
import java.awt.*;
import java.awt.image.*;
```

```
public class ImageProcessExample extends ImageLoadExample
{
  public static void main(String[] args){new ImageProcessExample();}

  public ImageProcessExample()
  {
    super();

    i = Toolkit.getDefaultToolkit().createImage("greatwall.jpg");
    waitForImage(i);

    int width = i.getWidth(this);
    int height = i.getHeight(this);
    int [] pixels = new int[width*height];

    try{
      // Grab the pixels and put them in the array called pixels
    PixelGrabber grabber = new PixelGrabber(i,0,0,width,height,
        pixels ,0, width);
    grabber.grabPixels();}
    catch(InterruptedException e){System.out.println(e.toString());}


    // Process each of the pixels individually
    for(int index = 0; index < width*height; index++)
      {
        int pixel = pixels[index];
        int red = (pixel & 0x00ff0000) >> 16;
        int green = (pixel & 0x0000ff00) >> 8;
        int blue = pixel & 0x000000ff;

        red = 255 - red;
        green = 255 - green;
        blue = 255 - blue;

        pixels[index] = 0xff000000|(red<<16)|(green<<8)|blue;
      }

    // Create a new image from the processed data
    MemoryImageSource data = new MemoryImageSource(width, height, pixels,0,width);
   i = Toolkit.getDefaultToolkit().createImage(data);

    waitForImage(i);

    setPreferredSize(new Dimension(width, height));
    frame.pack();
  }

  public void paint(Graphics g)
  {
    g.drawImage(i, 0, 0, this);
  }
```

(a) (b)

Figure 1.5: The pixel level invert operation (a) The input image and (b) an inverted representation of the input image.

```
55 }
```

The application obtains an integer array representation of the input image using the `grabPixels()` method of the `PixelGrabber` class. Each pixel in the image is decomposed into its red, green and blue colour components. The invert operation is performed on each of these colour components and the processed pixels are used to create a new image using the `createImage()` method of the `Toolkit` class. The output generated by this program is illustrated in Figure 1.5.

1.3.4 Defining Colours

The most basic way to represent a colour in Java is by using a single integer primitive. A java integer is 32-bits wide. This is divided into a total of four colour components: alpha, red, green and blue. Each component is allocated 8-bits of storage. Hence each component can have 2^8 (256) values. The alpha component represents opacity (the opposite of transparency). A low alpha value indicates that the colour is transparent and a high alpha value indicates that the colour is opaque. The format of the Java colour is illustrated in Figure 1.6.

Java also provides a class that is used to encapsulate colour information. The `Color` class represents ARGB colour information specified as a either integer primitives or float primitives. An instance of the `Color` class can be created using one of the following constructors:

- `Color(float r, float g, float b, float a)`
  Creates a colour object with the specified floating point colour components. It should be noted that when float primitives are used the colour components have a value in the range 0.0 - 1.0.

- `Color(int r, int g, int b, int a)`
  Creates a colour object with the specified integer colour components. It should

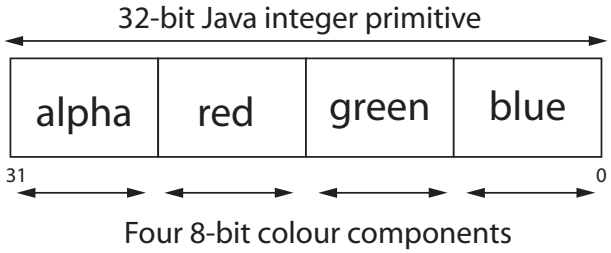


Figure 1.6: An illustration of the pixel format used by Java. The 32-bit bits of an integer primitive are divided into four 8-bit colour components representing: alpha or opacity, red, green and blue. The minimum value for each colour component is 0 and the maximum value for each component is 255.

be noted that when integer primitives are used the colour components have a value in the range 0 - 255.

1.4 Java 2D

The Java 2D API enhances the graphics, text and imaging capabilities of the Abstract Windowing Toolkit providing:

- Richer graphics, font and imaging support
- Enhanced colour definition
- A rendering model for printers and display devices

1.4.1 Graphics

The `Graphics2D` class extends the `Graphics` class to provide more sophisticated control over:

- Geometry
- Coordinate transformations
- Colour management
- Text layout

The `Graphics2D` class also provides an anti-aliasing feature. This facilites the generation of smoother, more visually appealing, graphics. The methods of the `Graphics2D` class include:

- `void setRenderingHint(RenderingHints.Key key, Object value)`
  Sets the value of a single preference from the rendering algorithms. Hint categories include controls for rendering quality and overall time/quality trade-off in the rendering process.
- `void scale(double sx, double sy)`
  Concatenaces the current `Graphics2D` transform with a scaling transformation. Subsequent renderings are resized according to the specified scaling factors relative to the previous scaling.

- `void rotate(double theta)`
  Concatenates the current `Graphics2D` transform with a rotation transform. Subsequent rendering is rotated by the specified number of radians relative to the previous origin.

- `void translate(double tx, double ty)`
  Concatenates the current `Graphics2D` transform with a translation transform. Subsequent rendering is translated by the specified distance relative to the previous position.

The following example illustrates how an instance of the `Graphics2D` class can be accessed through the `paint()` method of a swing component.

```
import java.awt.*;
import javax.swing.*;

public class Graphics2DExample extends JPanel
{
  public static void main(String args[]){new Graphics2DExample();}

  JFrame frame = new JFrame();

  public Graphics2DExample()
  {
    // Initialise the display frame
    frame.getContentPane().setLayout(new BorderLayout());
    frame.setSize(512,512);
    frame.getContentPane().add(this, BorderLayout.CENTER);
    frame.setVisible(true);
  }

  public void paint(Graphics g)
  {
    Graphics2D g2d = (Graphics2D)g;
    g2d.setColor(Color.BLACK);

    // Draw the pattern using a for loop
    for(int i=10; i<300; i+=10)
      g2d.drawLine(10,i,300-i,10);
  }
}
```

This code performs the same function as the earlier example that demonstrated the `Graphics` class. Consequently, the output generated by this example is similar to the output illustrated in Figure 1.3. The differences between this example and the earlier example are:

1. Swing graphical user interface components are used rather than AWT components. Note that swing components are preceded by the letter 'J'.

2. The `Graphics` object passed to the paint method is converted to a `Graphics2D` object (by casting). This enables access to the advanced graphics functionality provided by the `Graphics2D` class.

1.4.1.1 Anti-aliasing

The `Graphics2D` class provides support for anti-aliasing. Anti-aliasing is used to deal with the problems associated with drawing continuous shapes (e.g. lines and circles) using discrete pixels. There are a number of different approaches to anti-aliasing. The anti-aliasing approach supported by the `Graphics2D` class is called prefiltering. This method treats a pixel as an area, and computes the colour of the pixel based on the overlap of the scene's objects with the region occupied by the pixel. The colour of the pixel is based on how much of the pixel's area is covered by an object. Prefiltering thus amounts to sampling the shape of the object very densely within a pixel region. For shapes other than polygons, this can be very computationally intensive.

Anti-aliasing can be turned on and off using the `setRenderingHints()` method of the `Graphics2D` class. As mentioned earlier, this method expects two arguments: a key and a value. When dealing with anti-aliasing the key must be `KEY_ANTIALIASING` and the value can be one of the following:

- `VALUE_ANTIALIAS_OFF` rendering is done without anti-aliasing
- `VALUE_ANTIALIAS_ON` rendering is done with anti-aliasing
- `VALUE_ANTIALIAS_DEFAULT` rendering is done with a default anti-aliasing mode chosen by the implementation
  - **Note:** That the default anti-aliasing mode on the Windows XP implementation of Java Standard Edition 6 is `VALUE_ANTIALIAS_OFF`

The following example demonstrates how anti-aliasing can be used in a Java2D application:

```
0  import java.awt.*;
   import javax.swing.*;

   public class AntiAliasingExample extends Graphics2DExample
   {
5    public static void main(String args[]){new AntiAliasingExample();}

     public AntiAliasingExample()
     {
       setPreferredSize(new Dimension(200,200));
10     frame.pack();
     }

     public void paint(Graphics g)
     {
15     Graphics2D g2d = (Graphics2D)g;

       // Set the anti-aliasing to the default mode
       g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
           RenderingHints.VALUE_ANTIALIAS_ON);
20
       int centreX = getWidth()/2;
```

```
    int centreY = getHeight()/2;
    int radMax = getWidth()/2;

    // Draw a series of concentric circles
    for(int radius=10; radius<radMax; radius+=10)
    {
      g2d.drawOval(centreX-radius, centreY-radius, radius*2, radius*2);
    }
  }
}
```

The program draws a series of concentric circles. The output of the program is illustrated in Figure 1.7. Two versions of the output are illustrated. In the first version the anti-aliasing key is set to `VALUE_ANTIALIAS_OFF` and in the second version the anti-aliasing key is set to `VALUE_ANTIALIAS_ON`.

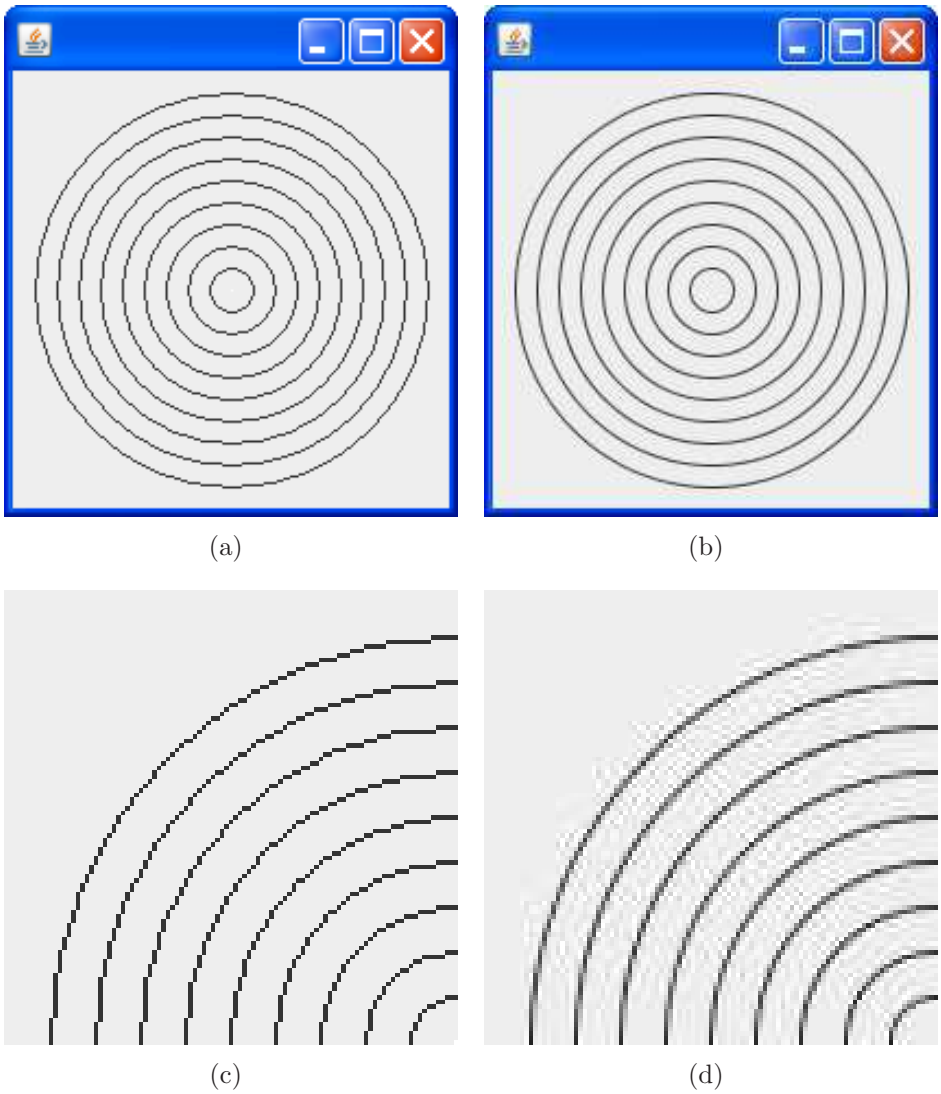


Figure 1.7: An example of anti-aliasing. A series of concentric circles drawn without anti-aliasing (a) and zoomed version (c). The same circles drawn with anti-aliasing (b) and zoomed version (d).

1.4.2 Image Support

Java 2D imaging is based on the immediate model for imaging. This makes it more suitable for use in imaging applications. The Java 2D API provides a new image class for the storage of bitmapped image data. This class, `BufferedImage`, extends the original `Image` class and can be constructed as follows:

- `BufferedImage(int width, int height, int imageType)`
  The `width` and `height` arguments give the dimensions of the image in pixels. The `imageType` argument specifies the type of image to be created. There are number of possible values for this argument:
  - `TYPE_BYTE_BINARY` represents a binary image where pixel values are mapped to either (0, 0, 0) or (255, 255, 255).
  - `TYPE_INT_ARGB` represents an image with 8-bit RGBA colour components packed into integer pixels. Note this is the default colour model.
  - `TYPE_INT_RGB` represents an image with 8-bit RGB colour components packed into integer pixels. If an alpha value is specified for a particular pixel then it is discarded.

The `BufferedImage` class also provides a number of useful methods that were not available in the original `Image` class:

- `int getRGB(int x, int y)`
  Returns the value of the specified pixel using the default RGB colour model i.e. `TYPE_INT_ARGB`. The returned value is in the form `0xAARRGGBB`. This method may throw an `ArrayOutOfBoundsException` if the specified coordinates are outside the bounds of the image.

- `void setRGB(int x, int y, int rgb)`
  Sets the value of the pixel at the specified coordinates. The default RGB colour model is assumed and the `rgb` argument must be in the form `0xAARRGGBB`. This methods may throw an `ArrayOutOfBoundsException` if the specified coordinates are outside the bounds of the image.

- `BufferedImage getSubImage(int x, int y, int w, int h)`
  Returns a subimage in the form of a `BuffferedImage` object that represents the region defined by the specified origin coordinates and dimensions.

Note: The `BufferedImage` class also provides versions of the `getWidth()` and `getHeight()` methods that do not require an `ImageObserver` argument.

It is possible to create a blank instance of a `BufferedImage` using the constructor outlined earlier. It is also possible to create an instance of a `BufferedImage` that is initialised from an image file. This is achieved using the `read()` method of the `ImageIO` class. The image source is specified as the argument to this method and can be a `File` object, a `URL` object or an `InputStream` object. There is no need to wait for the image data to load as this is handled automatically within the `read()` method. The following example demonstrates how a `BufferedImage` object can be initialised from a local file and displayed using Java2D imaging.

```
import java.io.*;
import javax.imageio.*;
import javax.swing.*;
import java.awt.*;
import java.awt.image.*;

public class BufferedImageLoadExample extends Graphics2DExample{

  public static void main(String args[]){new BufferedImageLoadExample();}

  BufferedImage b;

  public BufferedImageLoadExample()
  {
    super();

    try
    {
      // Load the image and resize the frame
        b = ImageIO.read(new File("image.jpg"));
        int width = b.getWidth();
        int height = b.getHeight();
        setPreferredSize(new Dimension(width, height));
        frame.pack();
    }
    catch(IOException ioe){System.out.println(ioe.toString());}
  }
  public void paint(Graphics g)
  {
    // paint the image on the JPanel
    Graphics2D g2d = (Graphics2D)g;
    g2d.drawImage(b, 0, 0, this);
  }
}
```

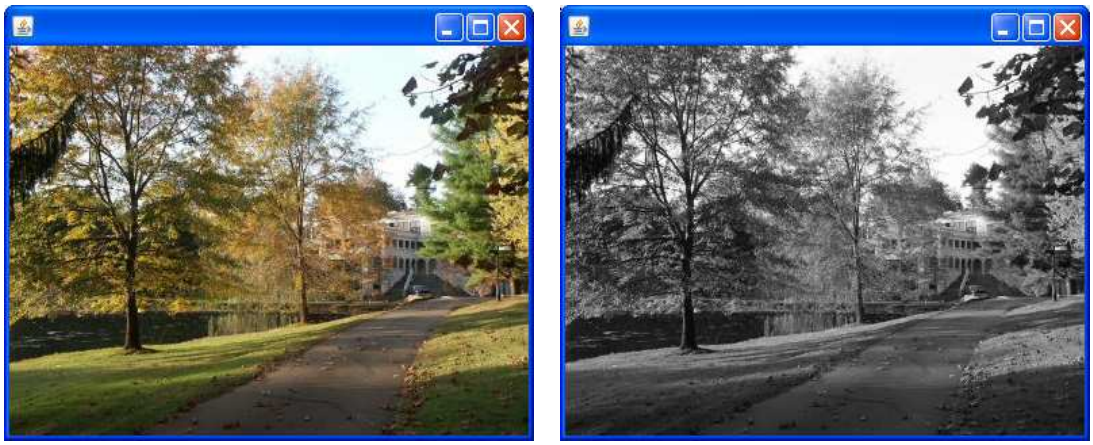
The output generated by this example is the same as the output generated by the AWT image load example illustrated in Figure 1.4.

1.4.3 Image Processing Support

It should be evident that the `BufferedImage` class provides direct access to pixel data using the `getRGB()` and `setRGB()` methods. Consequently the `BufferedImage` class provides a much more straightforward interface for image manipulation, and image processing operations can be carried out without the processing overhead associated with the AWT imaging model. The following example demonstrates how the colour to greyscale operation can be carried out using Java 2D imaging.

```
import java.io.*;

import javax.imageio.*;
```

```
import javax.swing.*;
import java.awt.*;
import java.awt.image.*;

public class BufferedImageProcessExample extends Graphics2DExample{

  public static void main(String args[]){new BufferedImageProcessExample();}

  BufferedImage b;

  public BufferedImageProcessExample()
  {
    super();

    try
    {
        b = ImageIO.read(new File("driveway.jpg"));
        int width = b.getWidth();
        int height = b.getHeight();

        for(int y=0; y<height; y++)
          for(int x=0; x<width; x++)
          {
            int pixel = b.getRGB(x,y);

            // Extract individual colour components
            int red =(pixel & 0x00ff0000) >> 16;
            int green = (pixel & 0x0000ff00) >> 8;
            int blue = pixel & 0x000000ff;

            // Perform the greyscale operation
            int grey = (red + green + blue)/3;

            // Create representation of pixel using default ARGB colour model
            pixel = 0xff000000 | (grey<<16) | (grey<<8) | grey;

            b.setRGB(x,y,pixel);
          }

        setPreferredSize(new Dimension(width, height));
        frame.pack();
    }
    catch(IOException ioe){System.out.println(ioe.toString());}
  }

  public void paint(Graphics g)
  {
      Graphics2D g2d = (Graphics2D)g;
      g2d.drawImage(b, 0, 0, this);
  }
}
```



(a) (b)

Figure 1.8: The pixel level grey-scale operation. (a) The input image and (b) a grey-scaled representation of the input image.

The program extracts the red, green and blue colour components for each pixel and averages them to generate a grey-scale representation of the image. The output generated by this program is illustrated in Figure 1.8.

Exercise: Update the code to perform a mid-level threshold operation. The mid-level threshold involves setting the output pixel to white if the grey-scale value of the input pixel is > 127, and setting it to black of the grey-scale value of the input pixel is ≤ 127.

1.4.4 2D Transformations

Transformations are very important in 2-D and 3-D graphics. This class represents a 2D affine transform that performs a linear mapping from 2D coordinates to other 2D coordinates that preserves the straightness and parallelness of lines. Affine transforms can be constructed using sequences of translations, scales, flips, rotations and shears. The affine transform equation for mapping from one coordinate system to another coordinates system is given below.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} m_{00}x + m_{01}y + m_{02} \\ m_{10}x + m_{11}y + m_{12} \\ 1 \end{bmatrix} \tag{1.1}$$

1.4.4.1 Scale

A scale transformation indicates a horizontal scaling by a factor of sx and a vertical scaling by a factor of sy. Note that a scale factor of 1.0 indicates that no scaling takes place in the relevant direction. The coordinate transformation associated with the scale operation is as follows:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Example: Scale the point (2.0, 3.0) by a factor of 0.5 along the x axis and a factor of 2.0 along the y axis.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} 2.0 \\ 3.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0.5 \times 2.0 + 0.0 \times 3.0 + 0.0 \\ 0.0 \times 0.0 + 2.0 \times 3.0 + 0.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 6.0 \\ 1.0 \end{bmatrix} \tag{1.2}$$

It is possible to create the affine transform that represents the scale operation by calling the static `getScaleInstance(double sx, double sy)` method of the `AffineTransform` class. The `sx` and `sz` arguments represent the factors by which the coordinates are scaled along the x and y axes.

1.4.4.2 Translate

A translation transformation indicates a horizontal translation by a distance tx and a vertical translation by a distance ty. Note that the translation distances are measured in pixels. The coordinate transformation associated with the translate operation is as follows:

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \tag{1.3}$$

Example: The following example uses classes from Java 2D imaging to implement a translation by 100 pixels in the positive x direction (across the screen to the left) and 40 pixels in the positive y direction (down the screen).

```
0  import java.io.*;
   import javax.imageio.*;
   import java.awt.*;
   import java.awt.image.*;
   import java.awt.geom.*;
5
   public class Translate2DExample extends BufferedImageLoadExample{

     public static void main(String args[]){new Translate2DExample();}

10   public void paint(Graphics g)
     {
         Graphics2D g2d = (Graphics2D)g;

         double tx = 100.0;
15       double ty = 40.0;

         // Degine the transformation matrix
         double[] matrix = {1.0, 0.0,
                            0.0,  1.0,
20                          tx,   ty};

       AffineTransform translateTransform = new AffineTransform(matrix);

       // Set the interpolation  type
```

Figure 1.9: A translated version of the image illustrated in Figure 1.4. The translation moves the origin of the image by 100 pixels in the x direction and 40 pixels in the y direction.

```
25      int interpolationType = AffineTransformOp.TYPE_NEAREST_NEIGHBOR;

        AffineTransformOp translateTransformOp =
          new AffineTransformOp(translateTransform,
                                interpolationType);
30
        // Perform the transformation
        BufferedImage result = translateTransformOp.filter(b, null);
        g2d.drawImage(result, 0, 0, this);
      }
35  }
```

The translation transformation is defined using a suitably constructed `AffineTransform` object. The `AffineTransform` object is used to create an instance of a `AffineTransformOp` object in conjunction with an argument that represents the type of interpolation type to be used. Three types of interpolation are supported:

- `TYPE_BICUBIC`
- `TYPE_BILINEAR`
- `TYPE_NEAREST_NEIGHBOUR`

The input image is subsequently filtered using the constructed `AffineTransformOp` to create a translated version of the image. Finally, the resulting image is displayed on the `JPanel`. The output generated by this application is illustrated in Figure 1.9. Note that the input to this operation was the image illustrated in Figure 1.4.

1.4.4.3 Rotate

A rotation transformation indicates a rotation about the origin by a specified angle θ. The coordinate transformation associated with the translate operation is as follows:

$$\begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.4}$$

Exercise: Update the translation example (`Translate2DExamle.java`) above to implement the rotation transformation using an angle of 45 degrees.

1.5 Java Advanced Imaging

Java advanced imaging (JAI) is an extension API that provides a set of object-orientated interfaces that support a simple, high-level programming model for image manipulation.

- Supports a wide range of image file formats (both read and write operations) e.g. BMP, TIFF, PNM, GIF and JPEG.
- Includes more than 80 image processing operations, most of which are native - optimised for performance
- Compatible with a variety of image formats and data types, support remote imaging and interoperates with the Java 2D API (immediate model for imaging).

The JAI API specification was developed by a consortium which includes:

- Sun Microsystems, Inc.
- Eastman Kodak, Inc.
- The Jet Propulsion Laboratory (JPL) @ NASA

JAI is currently being used in a variety of diverse applications.

- Defence and Intelligence
- Geospatial Data Processing
- Document Image Processing
- Bioinformatics

1.6 VRML

VRML is a text based language for describing 3D content. This language was originally known as Virtual Reality Markup Language and was subsequently renamed to Virtual Reality Modeling Language. VRML is based on the inventor file format from Silicon Graphics Inc. (SGI) and VRML version 1.0 is actually a subset of Inventor:

- it doesn't include the advanced interaction and animation capabilities supported by Inventor
- facilitated implementation on a wide variety of platforms

- Severely restricted flexibility and only facilitated the development of simple static worlds

VRML 1.1 was intended to meet some of the shortcomings of the VRML 1.0 specification by introducing support for:

- Audio clips
- Very primitive animation

VRML 1.1 was never made public, instead attention was focused on a complete overhaul of the language. This resulted in version 2.0 of the VRML language. VRML 2.0 was released in 1996 and provided rich support for:

- interaction
- animation
- 3D content

1.6.1 Software

VRML content can be viewed can be viewed using a standard web browser a suitable plug-in has been installed. The details for the VRML client used in the development of this material are as follows:

- The Cortona VRML client from Parallel Graphics version 5.1 (release 157)
- Available as a free download from: `http://www.parallelgraphics.com`

This plug-in can be used in conjunction with either Mozilla Firefox or Microsoft Internet Explorer.

1.6.2 VRML Coordinate System

VRML enables the definition of 3D content in a virtual world. VRML uses a cartesian coordinate system. Every point in a VRML world can be described by a set of x, y and z coordinates.

- The x coordinate determines position left and right of the origin. A positive x coordinate would indicate that a point is to the right of the origin.
- The y coordinate determines position above and below the origin. A positive y coordinate would indicate that a point is above the origin.
- The z coordinate determines position in front of and behind the origin. A positive z coordinate would indicate that a point is in front of the origin.

This coordinate system is illustrated in Figure 1.10.

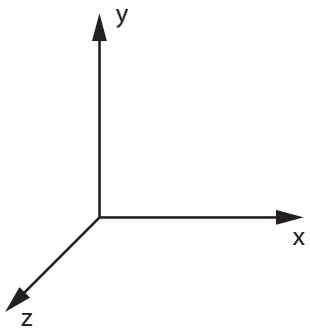


Figure 1.10: An illustration of the VRML coordinate system.

1.6.3 VRML Scene Graphs

A scene graph defines the relationship between objects contained in a virtual world. A common definition of a scene graph is a data structure composed of nodes and arcs.

- A node is a data element in a scene graph.
- An arc is a relationship between data elements. The arcs typically represent a parent-child relationship.

Scene graphs are constructed in the form of a directed-acyclic graph (DAG).

- A directed graph is a graph in which the arcs have direction.
- A directed-acyclic graph is a directed graph in which there are no other cycles i.e. beginning at any node in the graph, a path cannot be found to return to the same node.

There is only one path from the root of a scene graph to each of its leafs.

- The path from the root of a scene graph to a specific leaf node is known as a scene graph path and there is only one scene graph path for each leaf node.
- Each scene graph path completely specifies the state information of its leaf. In the case of a visual object, the state information would include the location, orientation and size of the object. Consequently, the visual attributes of each visual object depend only on its scene graph path.

Graphic representations of a scene graph can be used for design and/or documentation i.e. the scene graph can be used in the specification for the program. An example of a scene graph is illustrated in Figure 1.11.

1.6.3.1 Nodes

A node is a component in a VRML scene graph that describes some type of functionality. The nodes of a VRML scene graph can be divided into two main categories:

- **Leaf nodes:** A scene graph node without any children. Leaf nodes typically define content within a VRML scene. Examples of leaf nodes include:

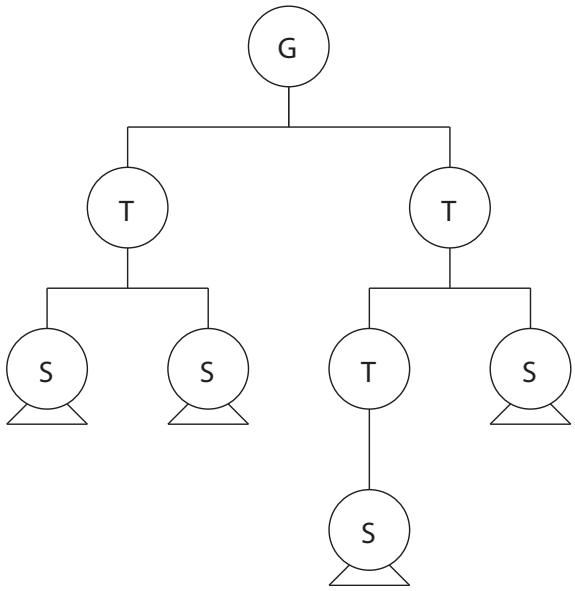


Figure 1.11: An example of a scene graph consisting of a ground node, three transform nodes and three shape nodes.

  - Shape nodes - Represent shapes that consist of nodes representing appearance and geometry
  - Sound nodes - Represent sound sources and can be associated with either WAV or MIDI files.
  - Light nodes - Represent a range of different light sources including ambient lights, point lights and spot lights.

- **Group nodes:** A scene graph node with children. Group nodes have one parent and an arbitrary number of children. Examples of group nodes include:
  - Transform node - Groups a series of leaf nodes together. The transformation associated with this node affects all of the grouped leaf nodes.
  - Switch node - Used to conditionally render a group of leaf nodes.
  - LOD node - Used to define many different representations of a particular object. The representation that is rendered is determined based on the distance between the view point and the LOD node.

1.6.3.2 Fields

Each node contains a list of fields that describe its functionality. A cone is an example of geometry node and it has two fields:

- base radius (default = 1 metre)
- height (default = 2 metres)

A field can have one of many data types:

- **Single Value Fields (SF)**
  - `SFBool` - A Boolean value, either true or false

  - `SFFloat` - A 32-bit floating point value
  - `SFInt32` - A 32-bit signed integer
  - `SFTime` - An absolute or relative time value
  - `SFVect2f` - A 2D coordinate (u, v) often used to represent texture coordinates
  - `SFVect3f` - A 3D coordinate (x, y, z) used to represent a position in space
  - `SFColor` - Three floating point values ranging from 0.0 to 1.0 that define red, green and blue colour components
  - `SFRotation` - Four floating point values. The first three values represent a point on the rotation axis (which goes through the origin). The fourth point represents the angle of rotation (in radians) around that axis
  - `SFImage` - A 2D image with between one and four colour components
    * One colour component ⇒ greyscale
    * Four colour components ⇒ RGB + transparency
  - `SFString` - A UTF8 string, supports the majority of international character sets
  - `SFNode` - A container for a VRML node

- **Multiple Value Fields (MF)**
  - `MFFloat` - An array of 32-bit floating point values
  - `MFInt32` - An array of 32-bit signed integer values
  - `MFVec2f` - An array of 2D floating point coordinates
  - `MFVec3d` - An array of 3D floating point coordinates
  - `MFColor` - An array of colour values, each with three components ranging from 0.0 to 1.0
  - `MFRotation` - An array of values representing axes and angles of rotation.
  - `MFString` - An array of UTF8 encoded strings

1.6.4 Shapes

A VRML node has two main properties:

- Geometry
  - Defines the structure of the shape
  - Can be a simple primitive (e.g. a sphere or cube) or a complex structure consisting of many faces
- Appearance
  - Material: Provides information about the colour, shininess, brightness and transparency of the shape
  - Texture: Defines an image to be stretched over the shape

The basic definition of a VRML shape node has the following format:

```
Shape
{
        exposedField SFNode appearance NULL
        exposedField SFNode geometry NULL
}
```

1.6.4.1 Geometry

VRML provides support for several primitive types of geometry that include: Box, Cone, Cylinder and Sphere. These primitives have the following definitions:

```
Box
{
        field    SFVec3f size              2 2 2
}
```

```
Cone
{
        field    SFFloat bottomRadius  1
        field    SFFloat height        2
        field    SFBool side           TRUE
        field    SFBool bottom         TRUE
}
```

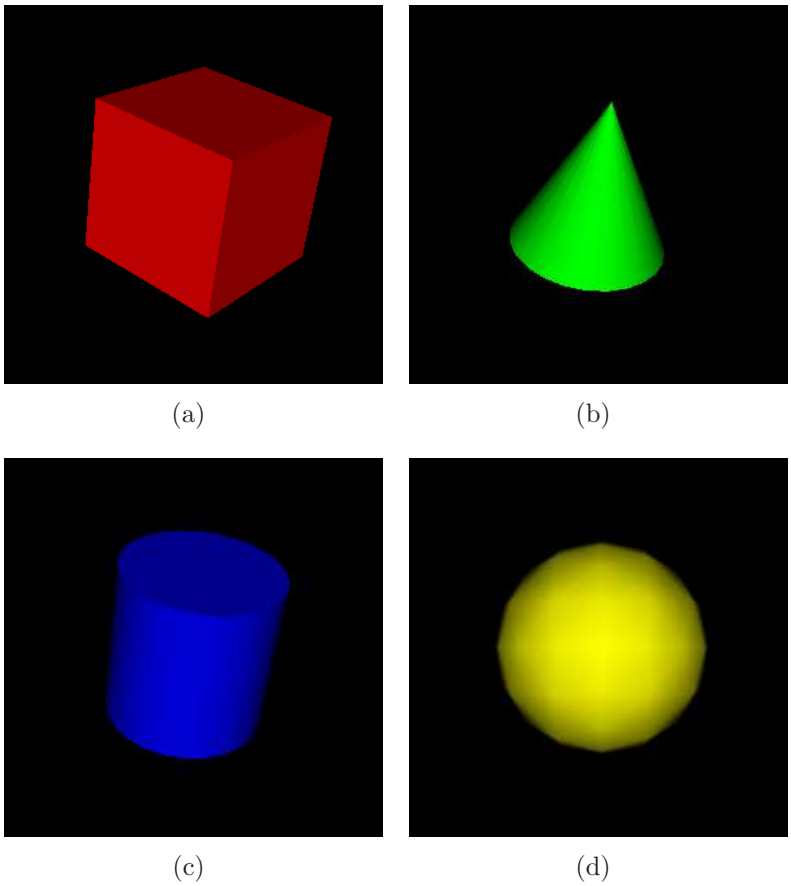
```
Cylinder
{
        field    SFBool  bottom  TRUE
        field    SFFloat height  2
        field    SFFloat radius  1
        field    SFBool  side    TRUE
        field    SFBool  top     TRUE
}
```

```
Sphere
{
        field SFFloat radius 1
}
```

An illustration of how these shapes are rendered in a VRML enabled browser is illustrated in Figure 1.2.

1.6.4.2 Appearance

The `Appearance` node provides support for all information that relates to the appearance of a shape. This includes information relating to the material and the texture that is to be applied to the geometry of the shape. The `Appearance` node



(a) (b)

(c) (d)

Figure 1.12: Renderings of four of the primitives supported by VRML (a) box, (b) Cone, (c) Cylinder, (d) Sphere.

has the following definition:

```
Appearance
{
        exposedField SFNode material NULL
        exposedField SFNode texture NULL
        exposedField SFNode textureTransform NULL
}
```

The `Material` node provides provides information about how a shape responds to different types of lighting e.g. ambient light and diffuse light. It can also be used to define an emissive colour so that the shape appears to emit light. A transparency value can also be set using the `Material` so that can shape can appear to be transparent.

```
Material
{
        exposedField SFFloat ambientIntensity 0.2
        exposedField SFColor diffuseColor 0.8 0.8 0.8
        exposedField SFColor emissiveColor 0 0 0
        exposedField SFFloat shininess 0.2
        exposedField SFColor specularColor 0 0 0
```

```
        exposedField SFFloat transparency 0
}
```

1.6.5 The VRML File Format

VRML content must be stored using a specific file format. The properties of this file format are as follows:

- The filename ends with the `.wrl` suffix
- The # symbol is used to indicate the start of a line of comments
- VRML 1.0 used 7-bit ASCII encoding
- VRML 2.0 uses UTF8 encoding
  - A multi-byte encoding in which each character can be encoded in as little as one byte and as many as four bytes
  - Supports the encoding of the character sets from most languages including Japanese
- The main components of a VRML file are:
  - The Header: `#VRML V1.0 ascii` or `#VRML V2.0 utf8`
  - The Body: Defines the structure and function of the virtual world

Example: Create a 3D world with a cone located at the origin. The cone should have the following properties:

- base radius = 1.2 meters
- height = 4.6 meters
- colour = red (r = 1.0, g = 0.0, b = 0.0)

This objective can be realised using the following VRML code:

```
0  #VRML V2.0 utf8

   Shape
   {
        geometry Cone
5       {
            bottomRadius 1.2
            height 4.6
        }
        appearance Appearance
10      {
            material Material
            {
                diffuseColor 1.0 0.0 0.0
            }
15      }
   }
```

The output generated when this file is loaded into a VRML enabled browser is illustrated in Figure 1.13.

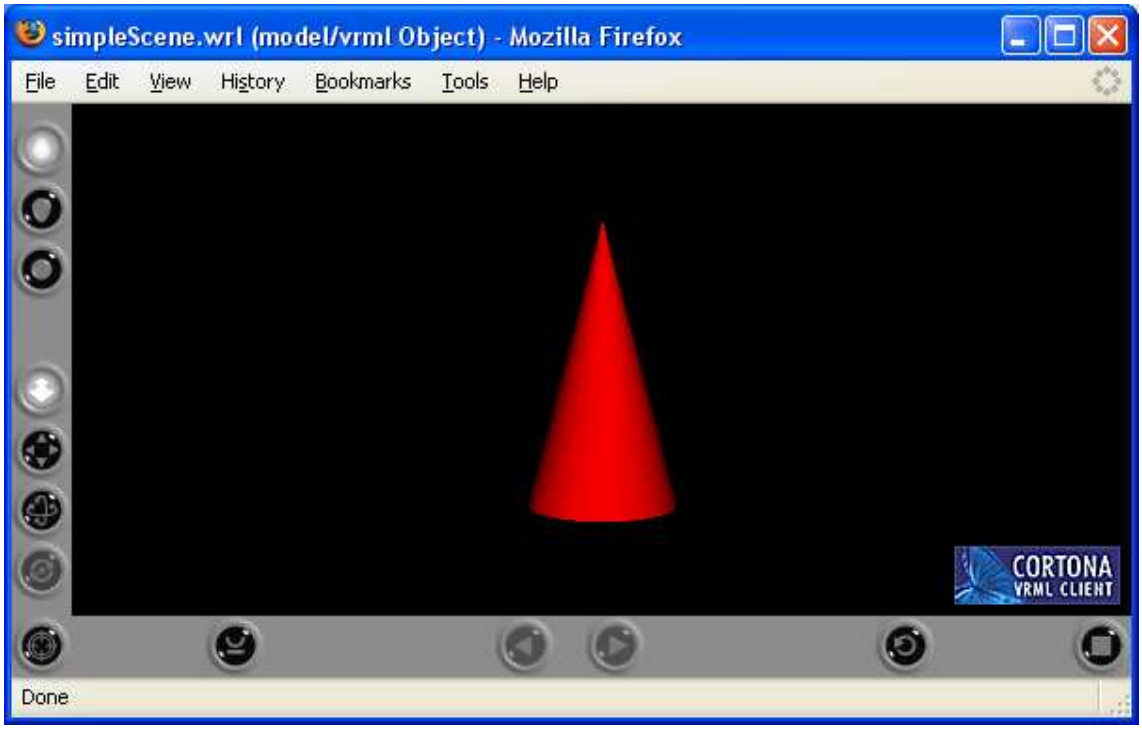


Figure 1.13: A cone VRML cone rendering using the Cortona VRML client plug-in for the Mozilla Firefox web browser.

1.6.6 Transformations

The `Transform` node represents a transformation from one 3D coordinates system into another preserving parallelness and straightness of lines. The `Transform` node can be used to implement:

- **Translations** - Translates the coordinates by the specified offsets in the x, y and z directions.
- **Scaling** - Scales the coordinates in the x, y and z directions by the specified ratios.
- **Rotation** - Rotates the coordinates about an axis, the rotation angle is measured in radians.

The general definition of a `Transform` node is as follows:

```
Transform
{
        eventIn MFNode addChildren
        eventIn MFNode removeChildren
        exposedField SFVec3f center 0 0 0
        exposedField MFNode children []
        exposedField SFRotation rotation 0 0 0 0
```

```
        exposedField SFVec3f scale 1 1 1
        exposedField SFRotation scaleOrientation 0 0 1 0
        exposedField SFVec3f translation 0 0 0
        field SFVec3f bboxCenter 0 0 0
        field SFVec3f bboxSize −1 −1 −1
}
```

The following examples illustrate the operation of the different types of transformation.

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} m_{00}x + m_{01}y + m_{02}z + m_{03}w \\ m_{10}x + m_{11}y + m_{12}z + m_{13}w \\ m_{20}x + m_{21}y + m_{22}z + m_{23}w \\ m_{30}x + m_{31}y + m_{32}z + m_{33}w \end{bmatrix} \tag{1.5}$$

1.6.6.1 Translation

The child of the translation is a cone with the default properties. The translation moves the cone 4 meters above the origin and 4 meters to the right of the origin. The output generated when this file is loaded into a VRML enabled browser is illustrated in Figure 1.14.

```
#VRML V2.0 utf8

Transform
{
        translation 4 4 0
        children Shape
        {
                appearance Appearance
                {
                        material Material
                        {
                                diffuseColor 1 0 0
                        }
                }
                geometry Cone{ }
        }
}
```

1.6.6.2 Scale

As before the child of the transformation is a cone with the default properties. The scale causes the size of the cone to change by the specified ratios in the x, y and z directions. The output generated when this file is loaded in a VRML enabled browser is illustrated in Figure 1.15.

```
#VRML V2.0 utf8
```

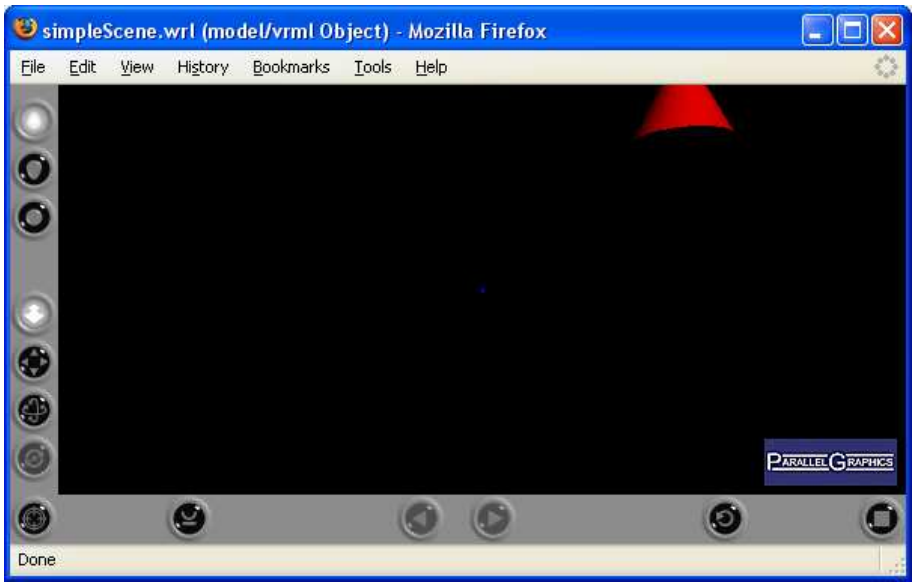


Figure 1.14: A VRML cone with the default geometry translated by 4 metres in the positive x direction and 4 metres in the positive y direction.

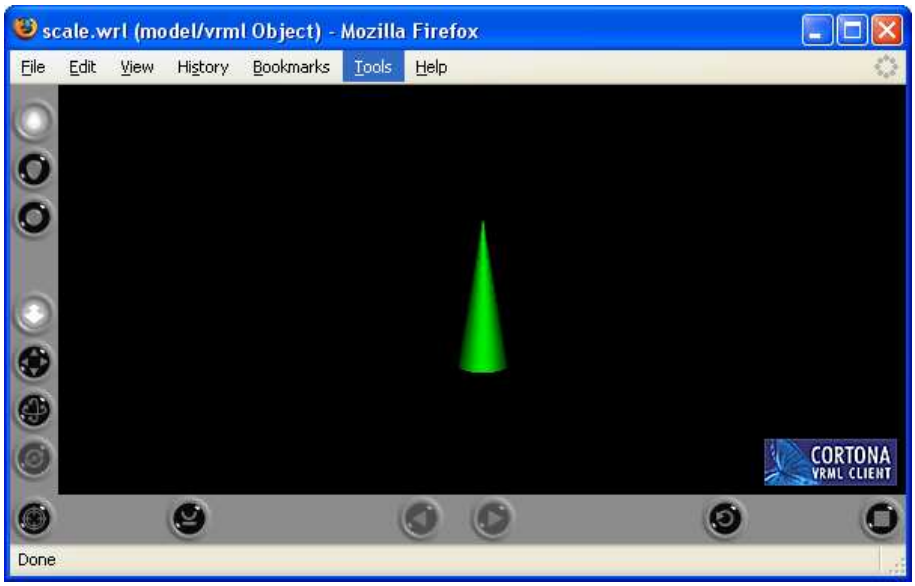


Figure 1.15: A VRML cone with the default geometry scaled by 0.5 in the x direction, 1.5 in the y direction and 1.0 in the z direction.

```
Transform
{
        scale  0.5  1.5  1.0
        children  Shape
        {
                appearance Appearance
                {
                        material Material
                        {
                                diffuseColor  0 1 0
                        }
```

```
                }
                geometry Cone{ }
        }
}
```

1.6.6.3 Rotation

In this example the child of the transformation is a box with the default properties. The rotation causes the box to rotate about a given axis by a given angle. The axis is defined by the vector (0, 1, 0), i.e. the y-axis, and the angle is 0.524 radians. This is equivalent to 30 degrees. The output generated when this file is loaded in a VRML enabled browser is illustrated in Figure 1.16.

```
#VRML V2.0 utf8

Transform
{
        rotation 0 1 0 0.524
        children Shape
        {
                appearance Appearance
                {
                        material Material
                        {
                                diffuseColor 0 0 1
                        }
                }
                geometry Box{ }
        }
}
```

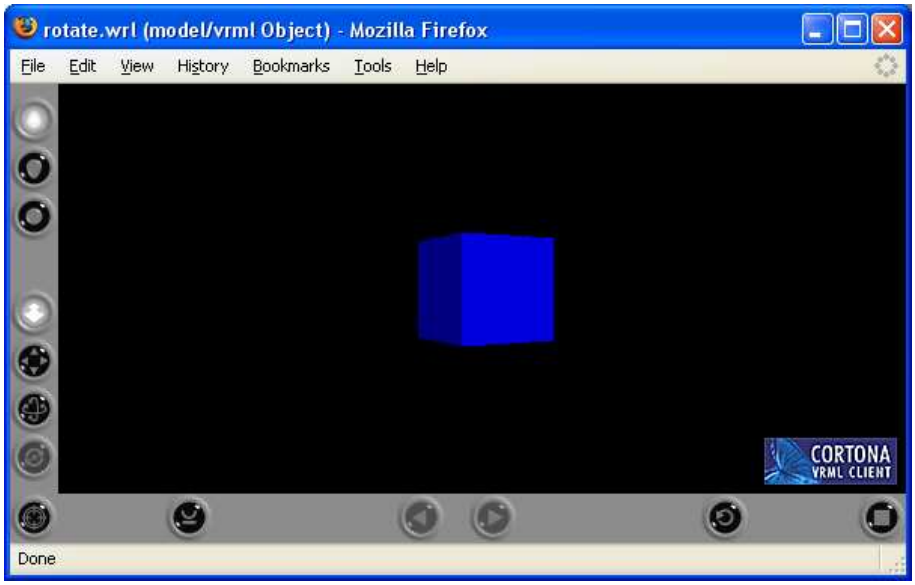


Figure 1.16: A VRML box with the default geometry rotated by 30 degrees (or 0.524) radians about the y axis.

1.6.7 Texture Mapping

Texture mapping involves applying a 2D image to the surface of a 3D object. The texture is scaled/streched to fit the particular surface. There are default texture mapping rules for all VRML shapes:

- `Box` - put one copy of the texture on each of the six faces of the box.
- `Cylinder` - Wrap once around the horizontal diameter and apply circular cutouts of the images to the top and bottom faces of the cylinder.
- `Sphere` - Wrap the image once around the horizontal diameter and squeeze it to a single point at the top and bottom.

The texture is specified as a URL. This can be located either remotely on the Internet or locally on the file system. The file formats supported for textures depend on the browser being used. GIF and JPEG images are generally supported by default. The following example shows how straightforward it is to create a 3D model of the planet earth in VRML using texture mapping. The texture mapping process used in this example is illustrated in Figure 1.17.

```
0  #VRML V2.0 utf8

   Shape
   {
           geometry Sphere{}
5          appearance Appearance
           {
                   material Material{}
                   texture ImageTexture
                   {
10                         url "textures/earth-low.jpg"
                   }

           }

15 }
```

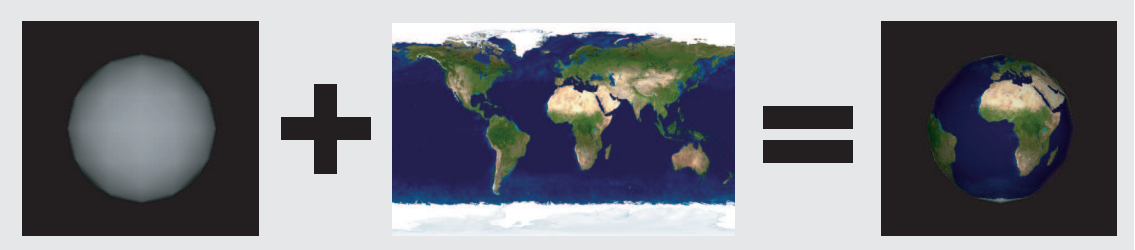


Figure 1.17: An illustration of the texture mapping process. The earth texture is mapped to the sphere by wrapping it once around the horizontal diameter and squeezing it to a single point at the top and bottom.

Mapping more than one texture to a shape with several faces is more complicated. Take a soft drinks can for example. This has three faces: top, bottom and label, see

Figure 1.18.

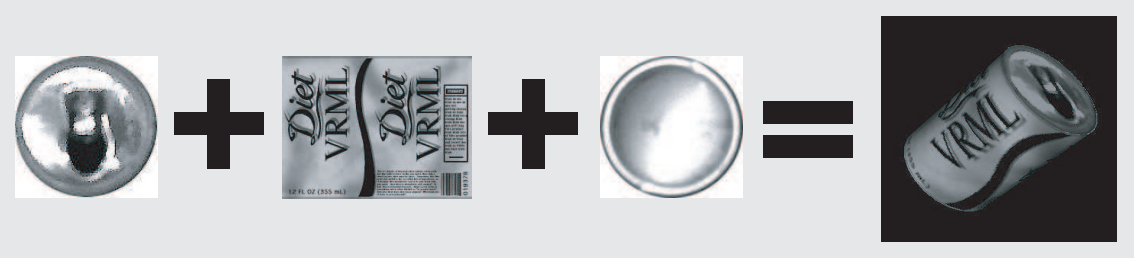


Figure 1.18: An illustration of the texture mapping process to create a soft drinks can from a cylinder geometry. Three separate textures must be specified in order to completely specify the appearance of the can.

It is only possible to map one texture to a particular shape, i.e. the same texture is mapped onto each face. In order to generate the can appearance three textured cylinders located at the same coordinates must be superimposed. The final textured cylinder is then generated by making the unwanted faces invisible, see Figure 1.19.

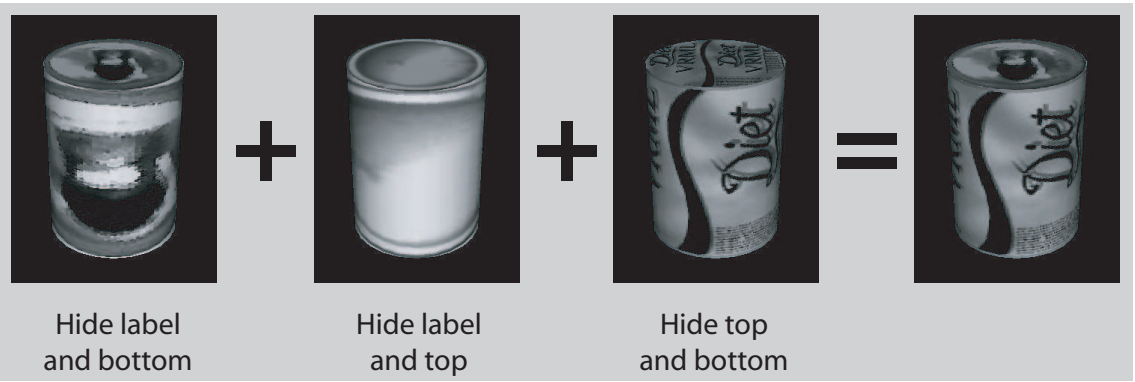


Figure 1.19: The stages of texture mapping that are involved to create a soft drinks can. Three separate cylinders must be created with the required textures. In each case the faces that are not required must be hidden and the cylinders must be co-located to give the final result.

The VRML code for the soft drinks can example is listed below:

```
0  #VRML V2.0 utf8

   Group {
       children [
       # Can top
5          Shape {
               appearance Appearance {
                   material Material { }
                   texture ImageTexture {
                       url "textures/cantop.jpg"
10                 }
```

```
            }
            geometry Cylinder {
                bottom FALSE
                side FALSE
                height 2.7
            }
        }
    # Can bottom
        Shape {
            appearance Appearance {
                material Material { }
                texture ImageTexture {
                    url "textures/canbot.jpg"
                }
            }
            geometry Cylinder {
                top FALSE
                side FALSE
                height 2.7
            }
        }
    # Can side
        Shape {
            appearance Appearance {
                material Material { }
                texture ImageTexture {
                    url "textures/canlabel.jpg"
                }
            }
            geometry Cylinder {
                top FALSE
                bottom FALSE
                height 2.7
            }
        }
    ]
}
```

In this example a total of three texture mapped cylinders are created. In each case the unwanted faces are hidden in order to create the final result.

1.6.8 Creating Custom Geomtery

Previous examples all used simple predefined shapes with limited flexibility. It was only possible to adjust the properties of the shapes and use them in conjunction with transformations. These simple shapes do not provide the power and flexibility required to create complex virtual worlds. Consequently, VRML provides a number of nodes for describing custom geometry e.g.

- `IndexFaceSet`
- `IndexLineSet`

- `PointSet`
- `ElevationGrid`

1.6.8.1 `IndexedFaceSet`

The `IndexedFaceSet` is used to define arbitrarily shaped flat surfaces. Surfaces are defined using a set of points with explicit ordering information:

- Straight lines are drawn between consecutive points
- A line is drawn between the first and last points
- The area with this closed boundary is then filled according to the associated appearance node

It should be noted that the coordinates of the points and the ordering information (indices) are specified in separate lists. It is possible for many indices to reference the same coordinate. Consequently, this is a very efficient way of defining geometry. Two rules govern the definition of these faces:

- **Rule 1:** All the points of the face must be coplanar
- **Rule 2:** A face must be convex (see Figure 1.20)

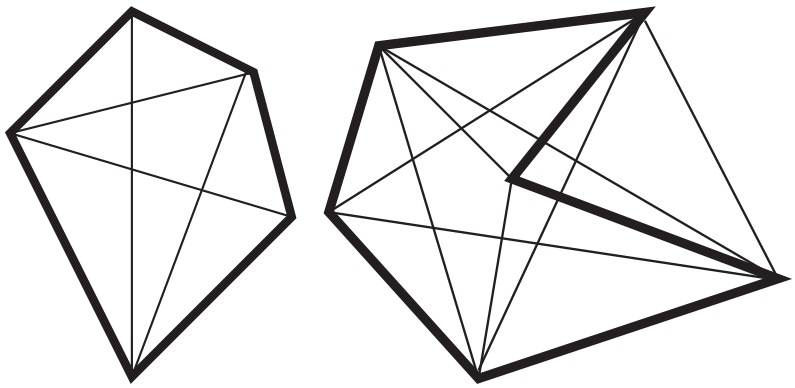


Figure 1.20: Examples of convex (a) and non-convex (b) faces. In the convex example the interconnections between each of the vertices are all located within the face. In the non-convex example some of the interconnections intersect with the boundaries of the face.

It should be noted that the convexity requirement does not restrict flexibility when creating complex geometry as a non-convex can be broken down into two or more convex faces (see Figure 1.21).

The `IndexedFaceSet` node is a very powerful and flexible node. As a result it is also a complex node with a large number of fields. A simplified representation is:

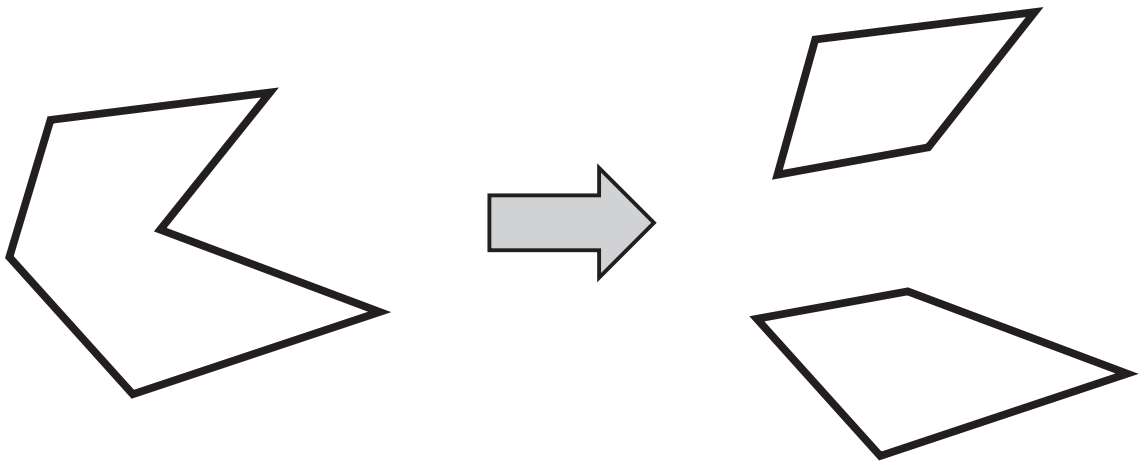


Figure 1.21: An example of a convex face divided into two non-convex faces.

```
IndexedFaceSet
{
        exposedField SFNode coord NULL
        field SFBool ccw TRUE
        field SFBool convex TRUE
        field MFInt32 coordIndex []
        field SFBool solid TRUE
}
```

One or both sides of a face can be rendered. This can be used to increase performance by removing the need to render faces that may never be visible, i.e. those inside a solid object. The order in which the points of a face are defined distinguishes the inside of a face from the outside of the face. Faces can ultimately be used as the building blocks for complex VRML models.

The `IndexedLineSet` node is very similar to the `IndexedFaceSet` node. Note that in the case of the `IndexedLineSet` node, the shape is not filled. The `PointSet` node defines a set of points and their associated colours and the `ElevationGrid` is intended to model terrain and consists of a 2D grid and an associated height map.

Example 1: The most straightforward example of a shape generated using an `IndexedFaceSet` node is a triangle, e.g. the equilateral triangle illustrated in Figure 1.22.

Such a triangle can be realised using the following VRML code:

```
#VRML V2.0 utf8

Shape
{
    geometry IndexedFaceSet
    {
        coord Coordinate
        {
```

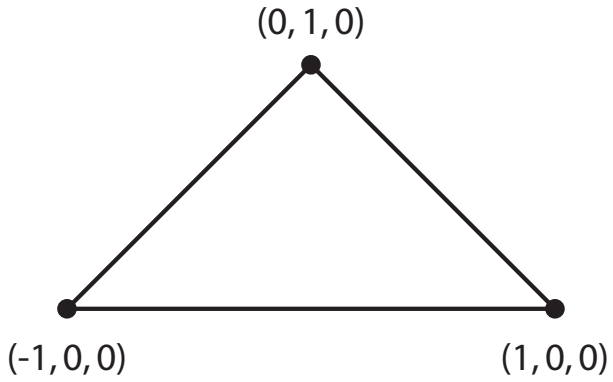


Figure 1.22: A simple equilateral triangle with vertices at (0,1,0), (-1,0,0) and (1,0,0)

```
            point [ −1 0 0, 1 0 0, 0 1 0 ]
        }
        coordIndex [0 1 2 −1]
    }
}
```

There are a total of three points and these correspond to the vertices of the triangle. The three points are indexed in the relevant order and the face is then closed by specifying an index of -1. The output obtained when this code is rendered in a VRML enabled browser is illustrated in Figure 1.23.

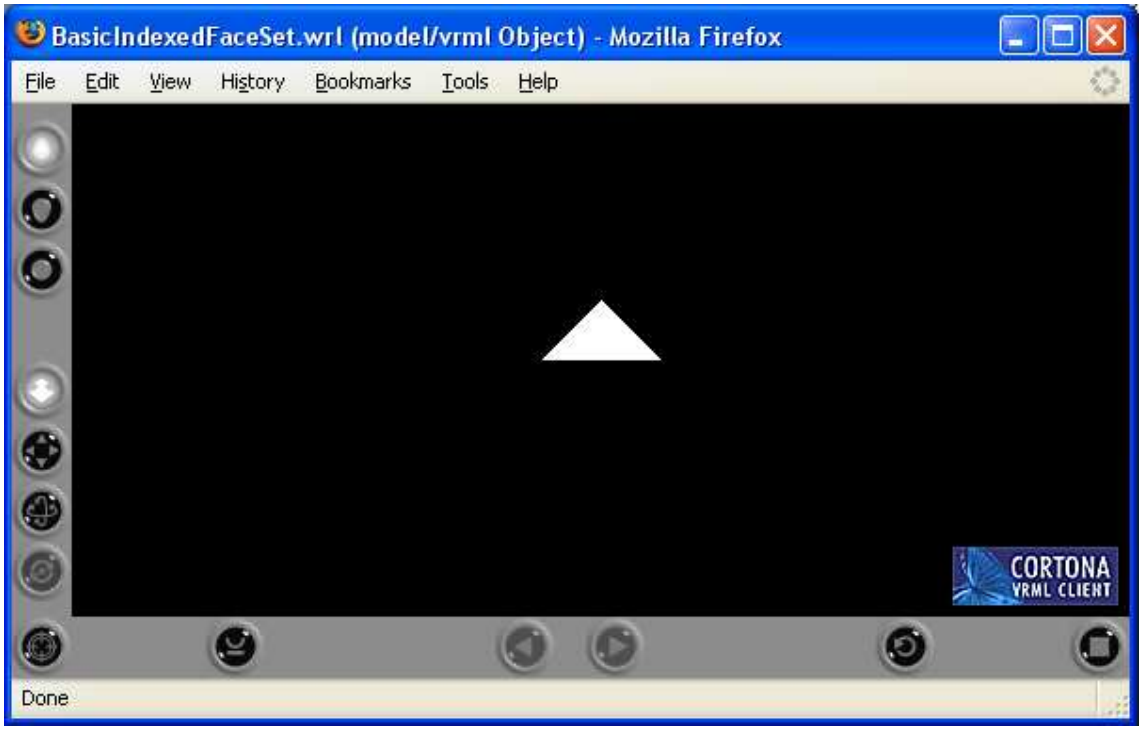


Figure 1.23: A simple equilateral triangle created using an IndexedFaceSet node.

Example 2: A more complicated example of shape that can be defined using an `IndexedFaceSet` node is a cube, see Figure 1.24. A cube has six faces and eight

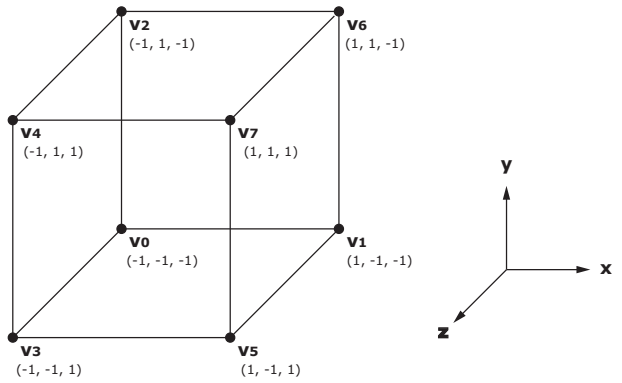


Figure 1.24: A simple cube consisting of six faces and eight vertices.

vertices. A cube can be realised using the following VRML code.

```
#VRML V2.0 utf8

Shape
{
appearance Appearance
        {
                material Material
                {
                        diffuseColor  0 0 1
                }
        }
    geometry IndexedFaceSet
    {
        coord Coordinate
        {
            point [ −1 −1 −1,  # v0
                     1 −1 −1,  # v1
                    −1  1 −1,  # v2
                    −1 −1 1,   # v3
                    −1  1  1,  # v4
                     1 −1  1,  # v5
                     1  1 −1,  # v6
                     1  1  1 ] # v7
        }
        coordIndex [0, 3, 4, 2, −1,   # left face
                    0, 1, 5, 3,−1,    # bottom face
                    0, 2, 6, 1, −1,   # back face
                    7, 5, 1, 6, −1,   # right face
                    7, 6, 2, 4, −1,   # top face
                    7, 4, 3, 5, −1]   # front face
    }
}
```

All eight vertices are defined in the point array and each face is subsequently created by specifying the relevant indices into the point array. In each case the point is closed

by specifying an index of -1. This is an efficient way to define geometry as it removes the need to define multiple instances of the same coordinate.

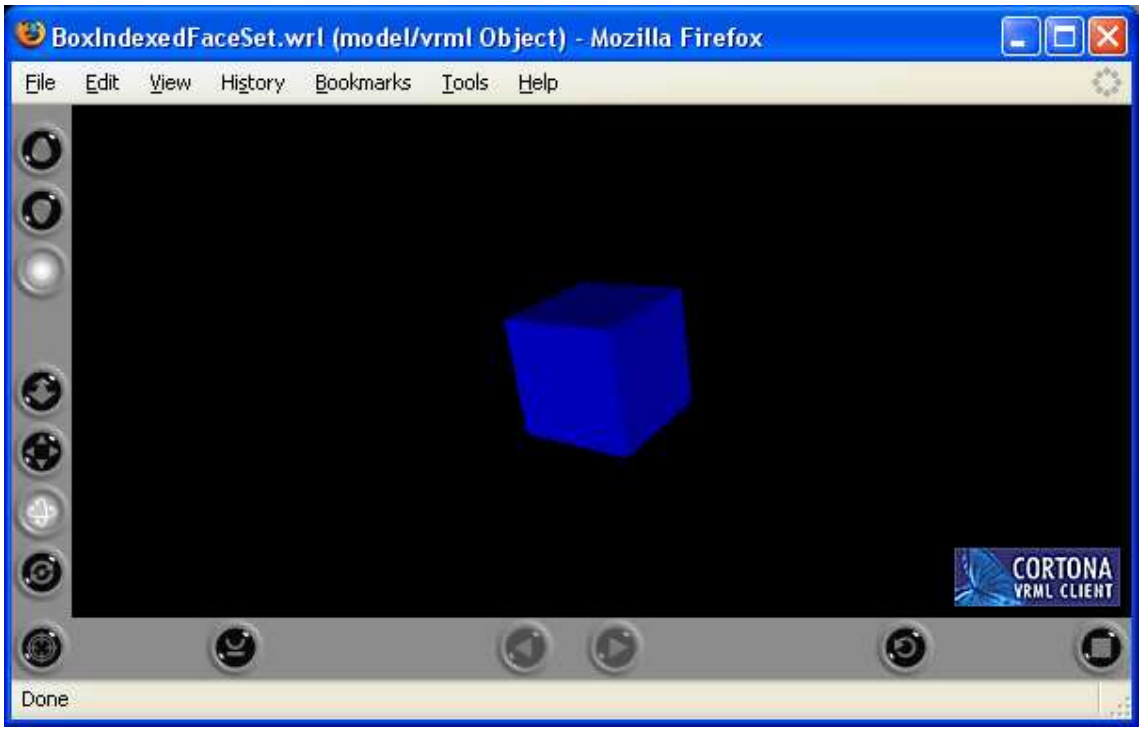


Figure 1.25: A custom cube structure created using an `IndexedFaceSet` node.

1.6.9 Other VRML features

VRML also provides support of a series of additional advanced features that include the following:

- Event handling i.e. support for user interaction
- Various modes of lighting
  - Point lights
  - Directional lights
  - Spot lights
- 3D sound
- Behaviors
  - Motion
  - Rotation
  - Morphing
- Atmospheric effects
  - Fog
  - Smoke

1.7 Summary

This chapter has introduced various concepts in relation to 2D and 3D graphics. The material dealing with 2D graphics demonstrated the graphics capabilities of the Java programming language and is relevant to the discussion on Java 3D that will take places in the next chapter. The use of VRML provided a straightforward introduction to 3D graphics is particularly relevant to the material discussed in the next chapter as Java 3D is based on VRML and many of the concepts including scene graphs will be discussed again in more detail in relation to Java 3D.

Chapter 2

Java 3D

Java 3D is a fully featured API. It uses the scene graph programming model to describe the structure of 3D worlds. The scene graph model allows the programmer to focus on the organization and functionality of the scene while Java 3D deals with all of the underlying rendering issues. Java 3D includes many of the features found in other popular 3D graphics APIs such as OpenGL and Direct3D. Java 3D is considered to be a higher level alternative to these low level APIs as it allows the programmer to focus on what to draw instead of how to draw it. Java 3D can be used to create virtual worlds with complex geometry, lighting, texture mapping and other features that enable the development of comprehensive, interactive scenes. Java 3D is also compatible with a variety of output devices from a standard monitors to advanced steroscopic visualisation systems that immerse the viewer in the virtual world.

2.1 Software

The software that was used in the development of the course material outlined in this chapter is as follows:

- Java SE Development Kit 6
- Java 3D 1.5.0

These packages can be downloaded from the Internet via the Sun Microsystems Java web site (`java.sun.com`). Installation of both packages is straightforward and in the case of the windows platform installation is handled automatically by an installer application. All examples outlined in this chapter were created and deployed using The Eclipse SDK version 3.2.1.

2.2 Basic Data Types

Java 3D supports a variety of data types to represents several different classes of data. The classes that represent these data types are defined in the `javax.vecmath` package. In the majority of cases the class name adheres to a format that includes:

- **Data type**
  - Point

  - Vector
  - Colour

- **Dimension**
  - Two-dimensional (2D), e.g. texture coordinates, these are used in the mapping of 2D texture to a 3D shape.
  - Three-dimensional (3D), e.g. a colour with red green and blue components.
  - Four-dimensional (4D), e.g. a four element axis angle that consists of a 3D vector as well as a rotational component.

- **Precision**
  - Byte, identified by the suffix *b*
  - Integer, identified by the suffix *i*
  - Float, identified by the suffix *f*
  - Double, identified by the suffix *d*

2.2.1 Colours

Colours are an example of a data type defined in the `javax.vecmath` package. Colors can be represented using three or four components and the individual colour components can be represented using either bytes or floating point values. In the case of byte values, the colour components range from 0 - 255 and the in the case of floating point values, the colour components range from 0.0 - 1.0. A specific colour can be created using one of the following constructors:

- `Color3b(byte r, byte g, byte b)`
  Creates a colour consisting of three components that are represented using byte values. The resulting colour consists of red, green and blue components with values in the range 0 - 255.

- `Color4f(float r, float g, float b, float a)`
  Creates a colour consisting of four components that are represented using float values. The resulting colour consists of red, green and blue components as well as an alpha component that defines opacity. All of the colour components have a value in the range 0.0 - 1.0.

2.3 Scene Graphs

Java 3D programs use scene graphs to define the structure of 3D virtual worlds. A scene graph is essentially a treelike data structure that is used to store, organise, and render 3D scene information including objects, lights and sounds. A scene graph consists of nodes which represent:

- Objects that are present in the virtual world represented by the scene graph.
- Aspects of the environment of the virtual world, for example, light or fog.

- Groups that contain nodes that can share a common property, for example, location.

A scene graph is usually defined graphically and then converted into code to enable rendering. The conversion process is reasonably straightforward and the resulting code can easily be related to the original scene graph diagram. An example of a scene graph is illustrated in Figure 2.1.

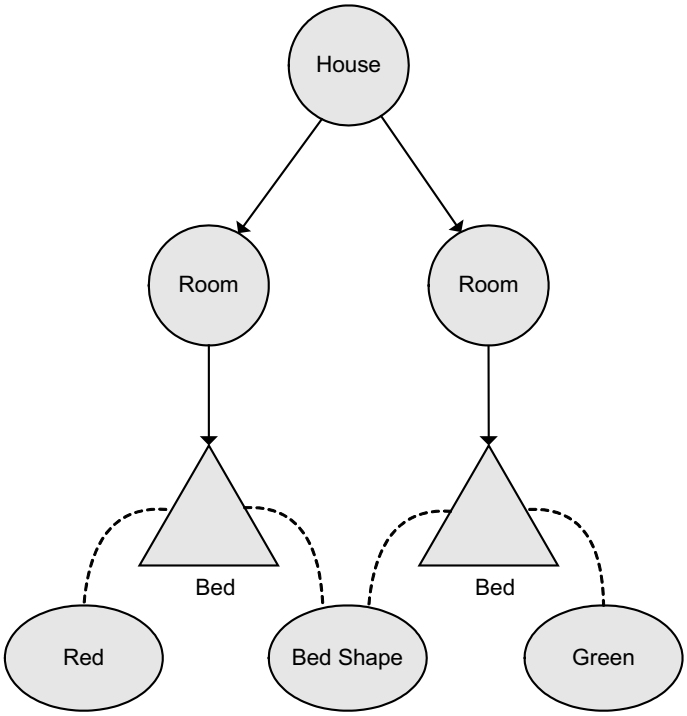


Figure 2.1: An example of a scene graph that represents a house. The house has two rooms and each room has a bed. Both of the beds have the same shape but different appearances.

This scene graph represents a house that has two rooms. There is a bed in each of the rooms. Both of the beds have the same shape but a different appearance. The house, the rooms and the beds are all represented by scene graph nodes. The visual and structural characteristics of the beds are represented using node components. Individual nodes cannot be shared by multiple parents but node components can. When this scene graph is implemented in Java the nodes and nodes components are represented by objects and the relationships between the nodes and node components are represented by references between the objects. The types of objects and object relationships that can appear in a scene graph are illustrated in Figure 2.2. The Java 3D scene graph is a directed acyclic graph (DAG). Connections between nodes in a scene graph are directed. This means that they are connected using a parent-child relationship. The connections are acyclic, which means that the parent-child relationship can't form loops, for example, the child of a group can't contain the parent of the group.

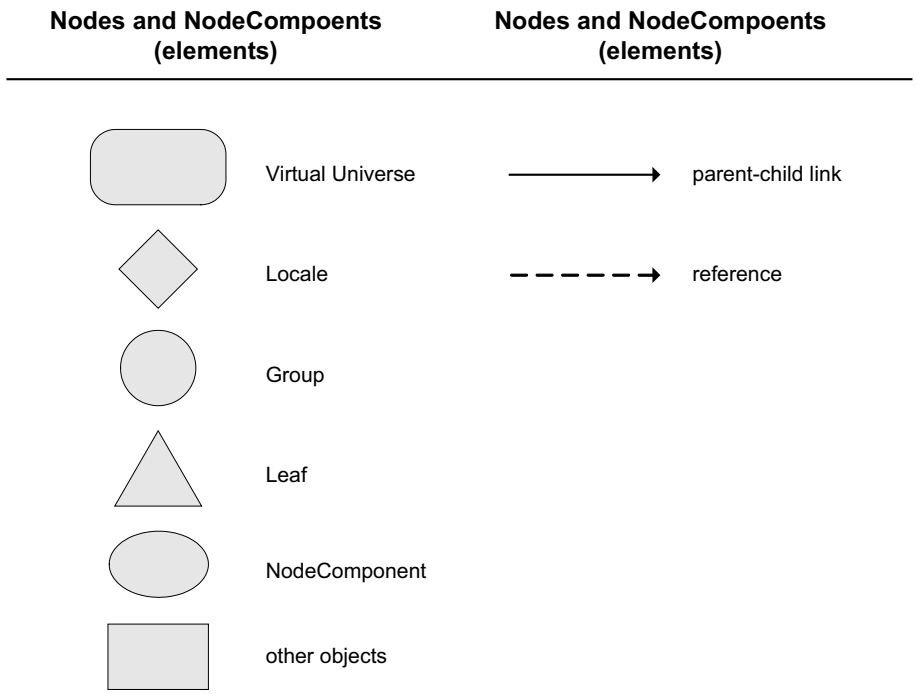


Figure 2.2: An overview of the range of symbols that are used to create Java 3D scene graphs.

2.3.1 `SceneGraphObject`

The class `SceneGraphObject` is the superclass for all classes that represents scene graph elements. There are two main types of scene graph elements:

- `Node` - represents a scene graph `Group` or `Leaf` node.
  - `Group` nodes are scene graph elements that can have children. `Group` nodes are used to organise the scene graph and can implement functionality that effect the way their children appear.
  - `Leaf` nodes are scene graph elements with no children. `Leaf` nodes can represent shapes, viewpoints or environmental properties.
- `NodeComponent` - represents information or data that is associated with a `Node` object. A single `NodeComponent` object can be shared between several `Node` objects and may be referred to many times in a scene graph.

Note: Strictly speaking, `NodeComponent` objects are not part of the scene graph, however, they can be referenced by other scene graph objects.

2.3.1.1 Group Nodes

The `Group` class represents a general purpose grouping node. `Group` nodes have one parent and an arbitrary number of children. It is possible for a `Group` node to have no children. The `Group` class defines functionality that enables its list of children to be added to, removed from or enumerated. The subclasses of the `Group` class define different types of grouping functionality.

- `BranchGroup` acts as the root of a scene graph branch (or subgraph). The subgraph represented by a `BranchGroup` can be added or removed from a scene graph that is currently being displayed.
- `OrderedGroup` ensures that its children are rendered in a particular order. An integer array of child indices is used to specify the order in which children are rendered.
- `TransformGroup` incorporates a 3D transformation that is used to alter the position, orientation and size of its children.
- `Switch` controls which of its children will be rendered. This group can be used to display all children, no children or a selection of children defined by a bitmask.
- `SharedGroup` allows multiple `Link` leaf nodes to share a subgraph that is represented by the `SharedGroup` object. This allows the same subgraph to appear several times in a scene.
- `ViewSpecificGroup` is a group node whose children are only rendered on a specified set of views.

2.3.1.2 Leaf Nodes

The `Leaf` class is an abstract class for all scene graph nodes that have no children, i.e. leaf nodes. The subclasses of the `Leaf` class are used to represent various entities that can be present within a virtual world. Leaf nodes include:

- `Shape3D` is used to represent graphical objects in a virtual world and consists of a geometry and an appearance.
- `ViewPlatform` controls the position, orientation and scale of the viewer. The viewer can be moved through the virtual world by updating the `TransformGroup` in the scene graph hierarchy above the `ViewPlatform`.
- Environmental nodes:
  - `Background` defines the background for the current scene. The background can be a solid colour or an image. The background can be drawn as a flat image, or alternatively, it can be associated with a geometry.
  - `Behavior` nodes make changes to the scene graph based on events such as time passing, moving the viewer or using the mouse. For example the `MouseRotate` behavior can be used to update the rotational aspect of the transform associated with a transform group.
  - `Clip` nodes keep objects that are far away from the viewer being drawn.
  - `Fog` nodes simulate atmospheric effects like fog or smoke. They can be used to increase realism in a scene by fading the appearance of objects that are farther from the viewer.
  - `Light` nodes are used to illuminate the scene. Types of light sources that are supported include `AmbientLight`, `DirectionalLight` and `PointLight`.
  - `Sound` nodes define sources of sound within the scene. Types of sound sources include `BackgroundSound` and `PointSound`.

2.3.1.3 Node Components

`NodeComponent` objects are used to represent information or data that is associated with either another `Node` or another `NodeComponent` object. A `Shape3D` object maintains references to two types of `NodeComponent` objects that define its structure and visual appearance. These are:

- `Geometry` defines the geometry component information required by a `Shape3D` object.
- `Appearance` defines all of the visual properties that can be associated with a `Shape3D` object.

The `Appearance` node component, in turn, maintains references to several other node components that are referred to as appearance components. These include:

- `Material` defines the response of a object to different types of light sources.
- `Texture` defines the texture image and texture mapping properties that are used when texture mapping is enabled.
- `TransparencyAttributes` defines the transparency characteristics for an object.

It is important to note that `NodeComponents` are not part of the scene graph but can be referenced by the scene graph. Consequently a single `NodeComponent` object can be referenced by several different scene graph nodes. An example of this is illustrated in Figure 2.3.

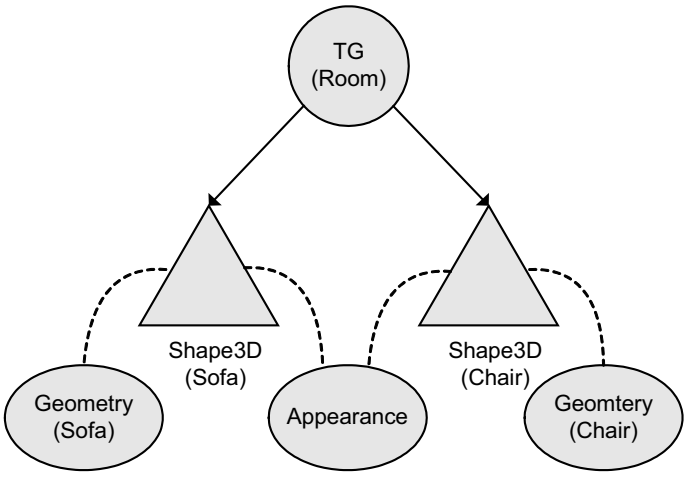


Figure 2.3: An example of how a single `NodeComponent` object can be shared by several scene graph nodes. In this example, both `Shape3D` objects share the same appearance. The relationship between the `Shape3D` nodes and the `Appearance` node component is represented by a dashed line to indicate a reference rather than a link.

2.3.2 `VirtualUniverse`, `Locale` and `SimpleUniverse`

A `VirtualUniverse` object is the top-level container for all scene graphs. A virtual universe consists of a set of `Locale` objects, each of which represents a high-resolution position within the virtual world. The scene graph is connected to a `Locale` via a `BranchGroup` object that is referred to as a "branch graph". A utility class called `SimpleUniverse` which is defined in the `com.sun.j3d.utils.universe` package is usually used to manage the `VirutalUniverse` and `Locale` objects.

The `SimpleUniverse` class extends `VirtualUniverse` and can be used to set up a minimal user environment to quickly and easily get a Java 3D program up and running. This utility class creates all the necessary objects on the "view" side of the scene graph. Specifically, this class creates a `Locale`, a single `ViewingPlatform`, and a `Viewer` object. The `SimpleUniverse` class provides all the necessary functionality required to enable the development of many basic Java 3D applications.

The `SimpleUniverse` class manages several objects that control the rendering of a scene:

- The `VirtualUniverse` and `Locale` objects that hold the virtual world that is to be displayed.
- The `ViewPlatform` that represents the location of the viewer in the virtual world.
- The `View` object that defines all the parameters required to render a 3D scene from a single viewpoint.
- The `Canvas3D` object that represents where the 3D scene is to be rendered to.

The scene graph diagram for the `SimpleUniverse` utility class is illustrated in Figure 2.4. A branch graph represented by a `BranchGroup` object can be added to the `Locale` of a `SimpleUniverse` object using the `addBrachGraph()` method. This is called the "content" branch of the scene graph . The `SimpleUniverse` utility class manages the "view" scene graph.

2.3.3 A Basic Scene

The following example uses the `SimpleUniverse` utility class to create a scene containing a single shape and a single behavior. The shape is a `ColorCube` which is a utility class that extends `Shape3D` to represent a cube structure with a different colour on each face. The behaviour is a `MouseRotate` behaviour that updates the transform associated with a `TransformGroup` based on mouse drag events that occur on the `Canvas3D`. The scene graph for this example is illustrated in Figure 2.5.
This scene graph can be converted into Java 3D to generate the following code:

```
0 import javax.swing.JFrame;
  import javax.vecmath.Point3d;

  import java.awt.BorderLayout;
  import java.awt.GraphicsConfiguration;
5
```

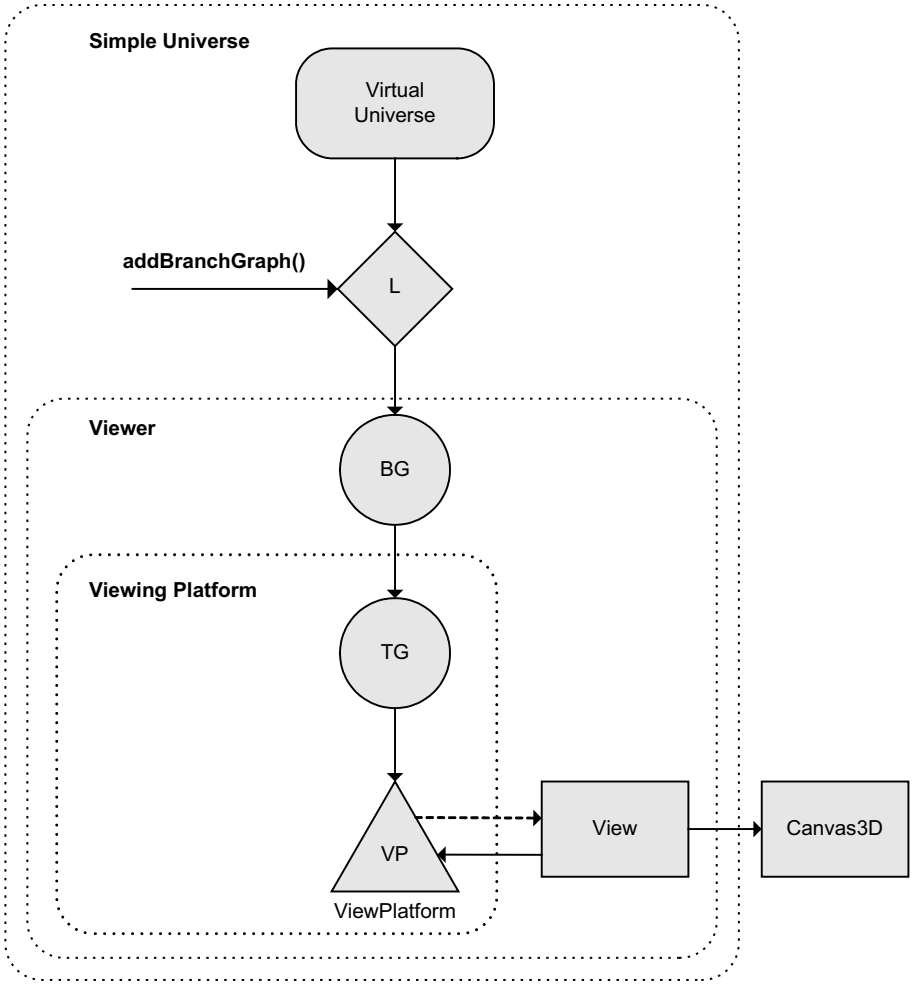


Figure 2.4: A simplified illustration of the scene graph diagram for the `SimpleUniverse` utility class. This scene graph represents the "view" branch graph. The "content" branch graph is added using the `addBranchGraph()` method.

```
import com.sun.j3d.utils.behaviors.mouse.MouseRotate;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.universe.*;
import javax.media.j3d.*;

public class BasicScene extends CustomFrame
{
  public static void main(String args[])
  {
    new BasicScene();
  }

  public Canvas3D canvas;

  public BasicScene()
  {
    getContentPane().setLayout(new BorderLayout());

    GraphicsConfiguration config =
      SimpleUniverse.getPreferredConfiguration();
```

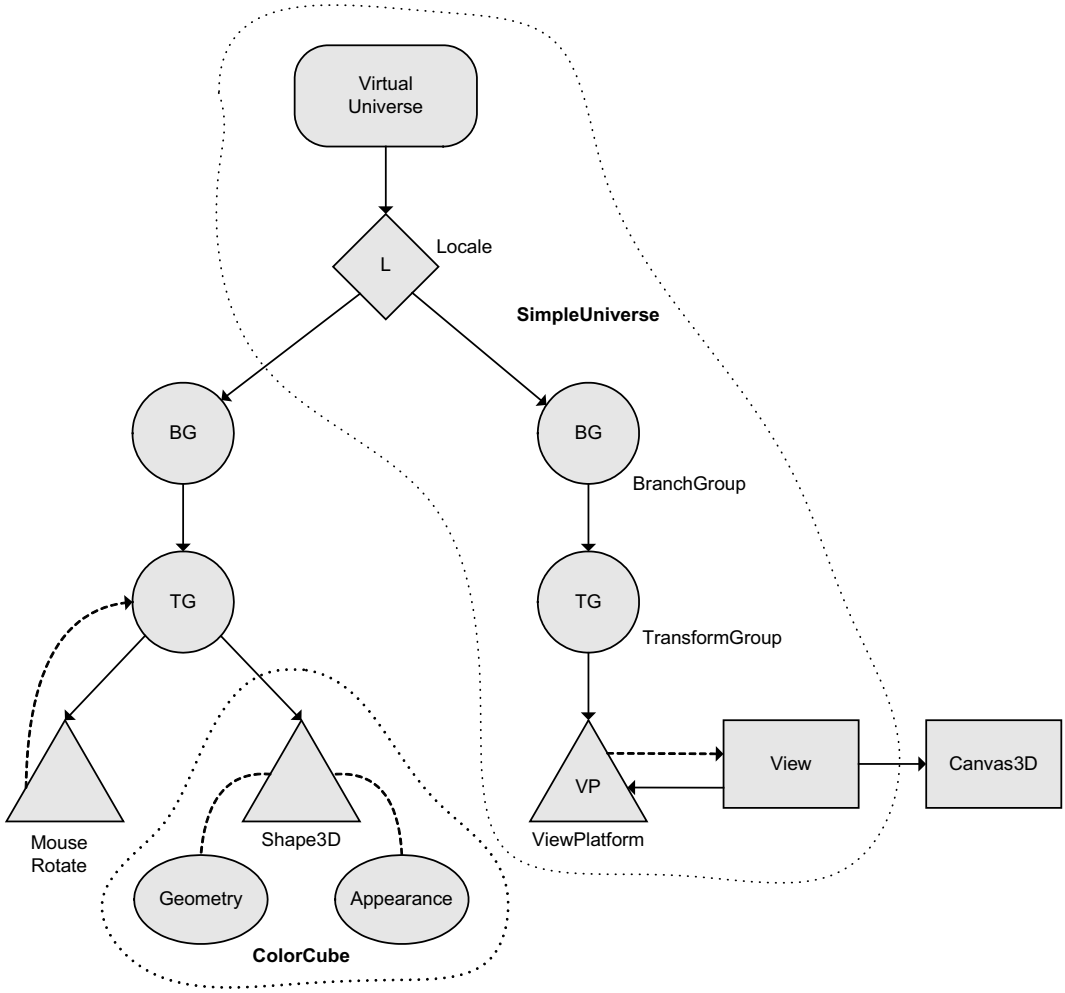


Figure 2.5: The scene graph used to generate `BasicScene.java`. The "content" branch graph consists of a root represented by a `BranchGroup` object, a `TransformGroup` that is modified by a `MouseRotate` behaviour and a `ColorCube` object that connected to the `TransformGroup` object.

```
    // Create a Canvas3D object and add it to the frame
     canvas = new Canvas3D(config);
     canvas.addMouseListener(this);
30  getContentPane().add(canvas, BorderLayout.CENTER);

    // Create a SimpleUniverse object to mange the "view" branch
    SimpleUniverse u = new SimpleUniverse(canvas);
    u.getViewingPlatform().setNominalViewingTransform();
35
    // Add the "content" branch to the SimpleUniverse
    BranchGroup scene = createContentBranch();
    u.addBranchGraph(scene);

40  setSize(256, 256);
    setVisible(true);
    }
```

```
  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // Create the transform group
    TransformGroup transformGroup = new TransformGroup();
    transformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_READ);
    transformGroup.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(transformGroup);

    // Create the mouse rotate behaviour
    MouseRotate rotate = new MouseRotate();
    rotate.setTransformGroup(transformGroup);
    rotate.setSchedulingBounds(new BoundingSphere(new Point3d(), 1000.0));
    transformGroup.addChild(rotate);

    // The color cube geometry
    ColorCube colorCube = new ColorCube(0.3);
    transformGroup.addChild(colorCube);

    root.compile();

    return root;
  }
}
```

The operation of the program can be described as follows:

- The main class of the application extends `JFrame` and can be used as a container to display the rendered scene. The layout of the `JFrame` is set to `BorderLayout`.
- The program begins by creating a new instance of a `Canvas3D` object. This will be ultimately used to display the rendered scene.
  - A `Canvas3D` object requires an instance of a `GraphicsConfiguration` object in the constructor.
  - A `GraphicsConfiguration` object defines the characteristics of a graphics destination such as a printer or a monitor.
  - A suitable instance of a `GraphicsConfiguration` object can be found by calling the `getPreferredConfiguration()` method of the `SimpleUniverse` class.
- A `SimpleUniverse` object is created and the `Canvas3D` object is specified in the constructor to indicate where the scene will ultimately be rendered.
- The nominal viewing transform is set for the viewing platform associated with the `SimpleUniverise` object.
  - This is achieved by calling the `setNominalViewingTransform()` method of the `ViewingPlatform` object obtained by calling the `getViewingPlatform()` method of the `SimpleUniverse` class.

  - The nominal viewing transform moves the `ViewPlatform` object back so that the viewer can see the area around the origin (0, 0, 0).

- The "content" branch graph is then created by a calling the `createContentBranch()` method.
- The "content" branch graph is then added to the `Locale` managed by the `SimpleUniverse` utility class by calling its `addBranchGraph()` method.
- Finally, the dimensions of the `JFrame` are specified and the `JFrame` is displayed.

The `createContentBranch()` method is used to define the "content" branch graph of the scene graph represented by the program. The operation of this method can be described as follows:

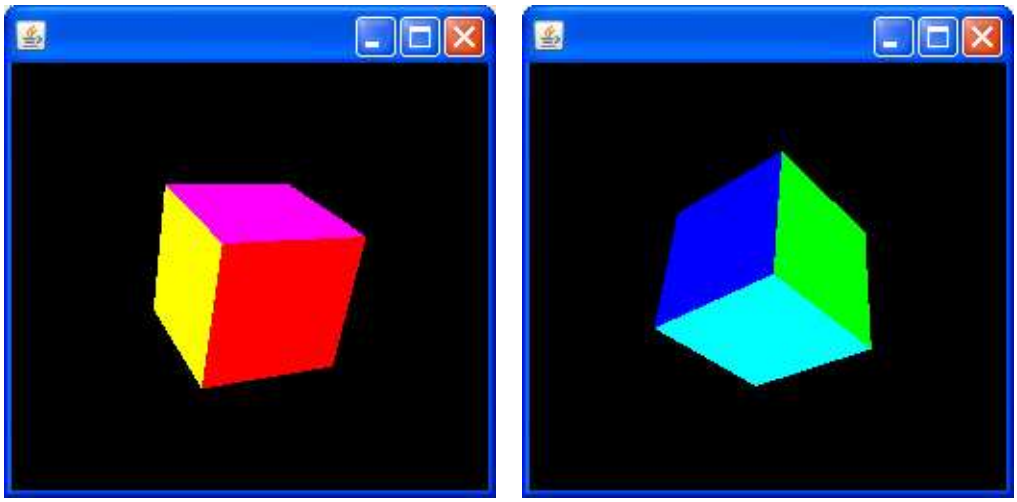
- The method begins by creating the root of the "content" branch graph. This is represented by a `BranchGroup` object.
- A `TransformGroup` object is then created. This is subsequently attached to the root of the scene graph. The `TransformGroup` is modified using the `setCapability()` method so that it can be updated when the scene graph is being displayed.
- A `MouseRotate` behaviour is created. This is attached to the `TransformGroup`. A reference is also created so the `MouseRotate` behaviour can update the transform associated with the `TransformGroup` object.
- A `ColorCube` object with dimensions 60 cm is then created and attached to the transform group. The default size of for a `ColorCube` object is 2 metres and this instance is scaled by a factor of 0.3.
- Finally, the "content" branch graph is compiled indicating that it is ready to be rendered.

Examples of the renderings obtained when this program is executed are illustrated in Figure 2.6. This example demonstrates the basic principles behind creating 3D content using the scene graph approach. This example will be built on by subsequent examples where this example is extended and the `createContentGraph()` method is overwritten to define different content and functionality. The "view" branch graph does not change between application so there is no need to redefine it each time.

2.3.4 Updating the Scene Graph

In order to render the "content" branch of a scene graph to a `Canvas3D` object it must be compiled and added to a suitably configured instance of a `SimpleUniverse` object. Once this has been done the "content" branch of the scene graph is considered to be live and the ability to modify the scene graph is greatly reduced. When a scene graph is compiled, it is converted to an optimised internal representation. This representation of the scene graph disables any functionality that is not required in order to optimise the scene graph for rendering.

It is possible to make some changes, for example, in the `BasicScene` example. The transformation associated with the `TransformGroup` object can be updated by the



(a) (b)

Figure 2.6: A `ColorCube` consists of six faces with different colours: red, green, blue, yellow, purple and cyan. This illustration shows two different renderings of a `ColorCube` object created using the `BasicScene` example.

user via the `MouseRotate` mouse behaviour. However, any changes that need to be supported must be specified prior to the scene graph going live. The different changes that can be specified are represented using capability bits, and every change that can be specified has a corresponding capability bit.

The following methods are defined by the `SceneGraphObject` class:

- `void setCapability(int bit)`
  Sets the capability indicated by the specified capability bit.
- `boolean getCapability(int bit)`
  Returns the status of the capability indicated by the specified capability bit.
- `void clearCapability(int bit)`
  Turns off the capability indicated by the specified capability bit.

The capability bits for a `SceneGraphObject` can only be changed when a scene graph is not live. Any attempt to change the capability bits after a scene graph has gone live will result in a `RestrictedAccessException`.

The default value for all read capability bits is true, i.e. by default all attributes may be read after the scene graph has gone live. The default value for all write capability bits is false, i.e. by default no attributes may be written after the scene graph has gone live. If the required capability bit has not been set, then any attempt to read or write the related property after the scene graph has gone live will result in a `CapabilityNotSetException` being thrown.

In the `BasicScene` example the `MouseRotate` mouse behaviour updates the transformation associated with the `TransformGroup` that holds the `ColorCube` object. In order for this operation to be supported, the `ALLOW_TRANSFROM_WRITE` capability bit must be set for the `TransformGroup` object using the `setCapability()` method.

2.4 Group Nodes

Group nodes have exactly one parent and an arbitrary number of children that are rendered in an unspecified order (or in parallel). It is possible for a group node to have no children. If this occurs then the group node is essentially ignored. Operations that can be carried out on group nodes include adding, removing and enumerating the children of the group node.

The `Group` class is the base class for all nodes that have children. It represents a general-purpose grouping node and defines a series of methods that include:

- `void addChild(Node child)`
  Append the specified child to the list of children maintained by this `Group` node.
- `Enumeration getAllChildren()`
  Returns an `Enumeration` object containing all of the children associated with this `Group` node.
- `Node getChild(int index)`
  Returns the child at the specified index from the list of children maintained by this `Group` node.
- `int numChildren()`
  Return a integer value that represents the number of children in the list maintained by this `Group` node.

The `Group` class defines the following capabilities:

- `ALLOW_CHILDREN_READ`
  Indicates that the `Group` node allows its children to be read after the scene graph has gone live.
- `ALLOW_CHILDREN_WRITE`
  Indicates that the `Group` node allows its children to be written to after the scene graph has gone live.
- `ALLOW_CHILDREN_EXTEND`
  Indicates that the `Group` node allows more children to be added after the scene graph has gone live.

The subclasses of Group node add additional semantics. These classes are discussed in the following text.

2.4.1 BranchGroup

A `BranchGroup` object serves as a pointer to the root of a scene graph branch. `BranchGroup` objects are the only objects that can be attached to, or removed from, a `Locale`. The following methods are defined by the `Locale` class to facilitate the attachment or removal of a `BranchGroup` object.

- `void addBranchGraph(BranchGroup branchGroup)`
  Attached the specified branch graph to the `Locale` object.

- `void removeBranchGraph(BranchGroup branchGroup)`
  Detach the specified branch graph from the `Locale` object.

The main method defined by a `BranchGroup` is the `compile()` method. This causes the branch graph represented by the `BranchGroup` object to be converted to an optimised internal representation. Once the `compile()` method has been called, only changes that have been explicitly enabled can be made to the scene graph.

The `BranchGroup` class defines the following capability bit:

- `ALLOW_DETACH`
  Indicates that the `BranchGroup` object can be detached from its parent.

2.4.2 OrderedGroup

The `OrderedGroup` node is a node that ensures its children are rendered in a specific order. In addition to the list of children inherited from the base `Group` class, the `OrderedGroup` class also maintains an integer array of child indices that indicates the rendering order for its children.

The methods defined by the `OrderedGroup` class include:

- `void addChild(Node child)`
  Appends the specified child to the `OrderedGroup` object and adds the index of the child to the end of the child index order array.
- `void addChild(Node child, int[] childIndexOrder)`
  Appends the specified child to the `OrderedGroup` object and sets the child index order array to the specified array.
- `int[] getChildIndexOrder()`
  Returns the current child index order array.

The `OrderedGroup` class also defines the following capability bits:

- `ALLOW_CHILD_INDEX_ORDER_READ`
  Indicates that the child index order array can be read after the scene graph has gone live.
- `ALLOW_CHILD_INDEX_ORDER_WRITE`
  Indicates that the child index order array can be written to after the scene graph has gone live.

2.4.3 TransformGroup

The `TransformGroup` class represents a group node that implements a 3D spatial transformation that can position, orient and scale all of its children. The transformation is represented by a `Transform3D` object. An instance of the `TransformGroup` can be created using one of the following constructors:

- `TransformGroup()`
  Creates a `TransformGroup` object that represents the identity transform. The resulting group has the same effect as a `BranchGroup` object.

- `TransformGroup(Transform3D transform)`
  Create a `TransformGroup` object that applies the transformation specified by the `transform` argument.

The `TransformGroup` class also defines the following methods to facilitate the specification and retrieval of its associated `Transform3D` object.

- `void setTransform(Transform3D transform)`
  Set the transform associated with this `TransformGroup` object to the specified value.

- `void getTransform(Transform3D transform)`
  Copy the `Transform3D` associated with this `TransformGroup` object to the `Transform3D` object passed as an argument.

The `Transform3D` class is represented internally as a 4×4 double-precision floating point matrix. A `Transform3D` object is used to perform translations, rotations and scaling transformation. The `Transform3D` object defines the following methods to facilitate these operations:

- `void setScale(double scale)`
  Sets the scale of the transformation to the specified value. Scale values below 1.0 cause a reduction in size and value above 1.0 cause an increase in size.

- `void setTranslation(Vector3f translation)`
  Causes the `Transform3D` object to represent the specified translation.

- `void setRotation(AxisAngle4f axisAngle)`
  Causes the `Transform3D` object to represent the specified rotation.

  - The `AxisAngle4f` object passed as an argument represents an axis and a rotational component. The axis is defined by a single point that represents a line through the origin and the angle represents the angle of rotation and is defined in radians.

The `TransformGroup` class defines the following capabilities:

- `ALLOW_TRANSFORM_READ`
  Indicates that the associated `Transform3D` object can be read after the scene graph has gone live.

- `ALLOW_TRANSFORM_WRITE`
  Indicates that the associated `Transform3D` object can be written to after the scene graph has gone live.

Example: The following example uses `TransformGroup` objects to place three shapes at different positions in a scene. The scene graph for the example is illustrated in Figure 2.7 and the Java 3D implementation of the scene graph is listed below:

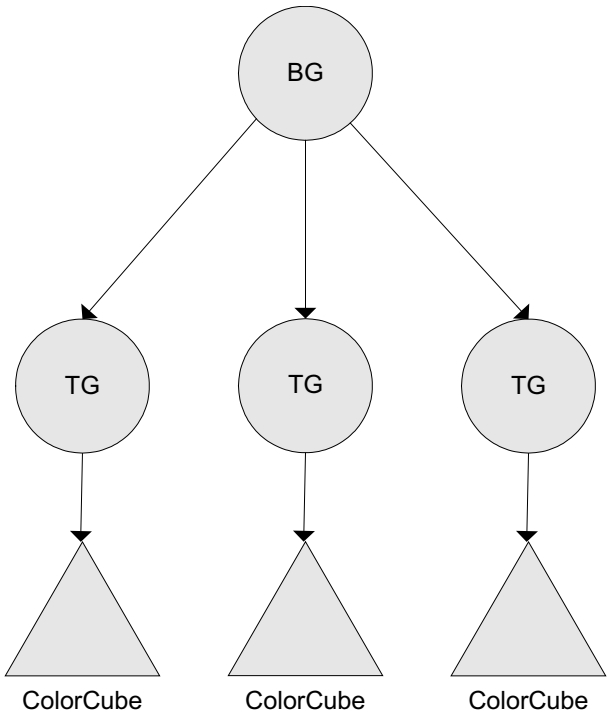


Figure 2.7: The scene graph for the `TransformGroup` example. This scene graph consists of a `BranchGroup` that represents the root, the root `BranchGroup` has three children. These children are `TransformGroup` objects which each have one `ColorCube` child. The location of the `ColorCube` nodes in the renered scene is depended on the transformation associated with the relevant `TransformGroup` object.

```
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.geometry.*;

public class TransformGroupExample extends BasicSceneWithMouseControl{

  public static void main(String args[]){new TransformGroupExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // bottom left ColorCube
    Transform3D t1 = new Transform3D();
    t1.setTranslation(new Vector3f(-0.25f, -0.25f, 0.0f));
    TransformGroup tg1 = new TransformGroup(t1);
    ColorCube c1 = new ColorCube(0.2);
    tg1.addChild(c1);

    // bottom right ColorCube
    Transform3D t2 = new Transform3D();
    t2.setTranslation(new Vector3f(0.25f, -0.25f, 0.0f));
    TransformGroup tg2 = new TransformGroup(t2);
    ColorCube c2 = new ColorCube(0.2);
```

```
        tg2.addChild(c2);

        // Top ColorCube
        Transform3D t3 = new Transform3D();
        t3.setTranslation(new Vector3f(0.0f, 0.25f, 0.0f));
        TransformGroup tg3 = new TransformGroup(t3);
        ColorCube c3 = new ColorCube(0.2);
        tg3.addChild(c3);

        root.addChild(tg1);
        root.addChild(tg2);
        root.addChild(tg3);

        return root;
    }
}
```

The main class of the program extends the `BasicSceneWithMouseControl` class. This is the same as the `BasicScene` class except three mouse behaviours have been added to the scene graph: `MouseRotate`, `MouseTranslate` and `MouseZoom`. The scene graph is implemented by overwriting the `createContentBranch()` method. The operation of the overwritten `createContentBranch()` method can described as follows:

- The root of the scene graph is created and represented by a `BranchGroup` object.
- The first child of the root node is a `TransformGroup` object.
  - A `Transform3D` object representing a translation 25 cm in the negative x direction and 25 cm in the negative y direction is associated with the `TransformGroup` object.
  - The `TransformGroup` object has a single `ColorCube` child with sides 40 cm in length (2.0 metre scaled by a factor of 0.2). The `ColorCube` is added to the `TransformGroup` using the `addChild()` method.
  - The capabilities for the `TransformGroup` are left unchanged, i.e. the `ALLOW_TRANSFORM_READ` capability is set (this is the default value for this capability) and the `ALLOW_TRANSFORM_WRITE` capability is not set (also by default).
- The root has two other children that represent `ColorCube` objects affected by different translations:
  - The second `ColorCube` object is translated 25 cm in the positive x direction and 25 cm in the negative y direction.
  - The third `ColorCube` object is translated 25 cm in the positive y direction.
- Finally, the three children are added to the root and the root is returned for compilation.

The output obtained when this scene graph is rendered is illustrated in Figure 2.8.

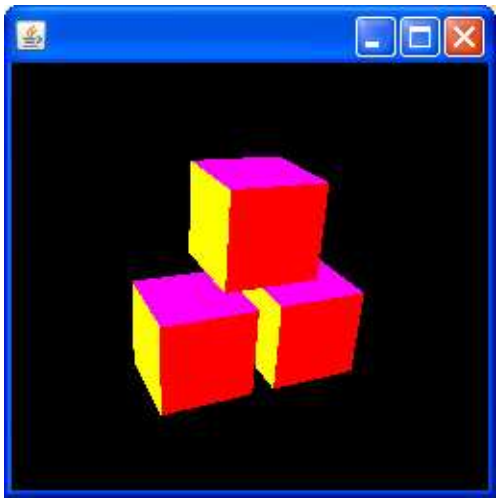


Figure 2.8: A rendering of the scene graph illustated in Figure 2.7. The three 40 cm `ColourCubes` appear in a triangular formation with a 10 cm gap between them

2.4.4 `Switch`

The `Switch` class represents a group node that can control which of its children are rendered. It defines a child selection value which defines the children to be rendered. The possible values are:

- `CHILD_NONE`
  None of the children of this `Switch` node are to be rendered.
- `CHILD_ALL`
  All of the children of this `Switch` node are to be rendered.
- `CHILD_MASK`
  Only the children of this `Switch` node that are indicated by a binary one in a mask are to be rendered.

If the `CHILD_MASK` selection value is specified then the children to be rendered are specified by a mask that is represented using a `BitSet` object from the `java.util` package.

An instance of a `Switch` node can be constructed using one of the following three constructors:

- `Switch()`
  Creates a `Switch` node where the child selection value is set to `CHILD_NONE`.
- `Switch(int whichChild)`
  Creates a `Switch` node with the specified child selection value.
- `Switch(int whichChild, BitSet childMask)`
  Creates a `Switch` node with the specified child selection value and child selection mask.

Once a `Switch` object has been created, the child selection value can be queried or set using:

- `int getWhichChild()`
  Returns the current child selection value that indicates how the children of the `Switch` node are to be rendered.

- `void setWhichChild(int childSelectionValue)`
  Set the current child selection value to the specified value.

If the child selection value is set to `CHILD_MASK` then a mask represented by a `BitSet` object determines which of the children are to be rendered. The number of bits in the mask is the same as the number of children. A bit value of 1 indicates the corresponding child is to be rendered, whereas, a bit value of 0 indicates that the corresponding child is not to be rendered. The mask can be queried or set using:

- `BitSet getChildMask()`
  Returns the current child selection mask in the form of a `BitSet` object.

- `void setChildMask(BitSet childMask)`
  Sets the current child selection mask to the specified value.

A `BitSet` object represents a vector of bits that grows if needed. A `BitSet` object can be created using the following constructor:

- `BitSet(int nbits)`
  Creates a `BitSet` object whose initial capacity is `nbits`.

The basic methods provided by the `BitSet` class facilitate querying and updating the individual bits contained within a `BitSet` object.

- `boolean get(int bitIndex)`
  Get the value of the bit at the specified location and return it in the form of a Boolean value.

- `void clear(int bitIndex)`
  Clear the bit at the specified location, i.e. set its value to false.

- `void set(int bitIndex)`
  Set the bit at the specified location, i.e. set its value to true.

The `Switch` class also defines two capability bits:

- `ALLOW_SWITCH_READ`
  Indicates that the `Switch` node can be read after the scene graph has gone live.

- `ALLOW_SWITCH_WRITE`
  Indicates that the `Switch` node can be written to or updated after the scene graph has gone live.

Example: The following example uses a `Switch` object to control the rendering of three child branches. In each case, the branch consists of a `TransformGroup` that implements a translation. Each `TransformGroup` has a single child which is a `ColorCube` object. The scene graph for the example is illustrated in Figure 2.9 and the Java 3D implementation of the scene graph is listed below:

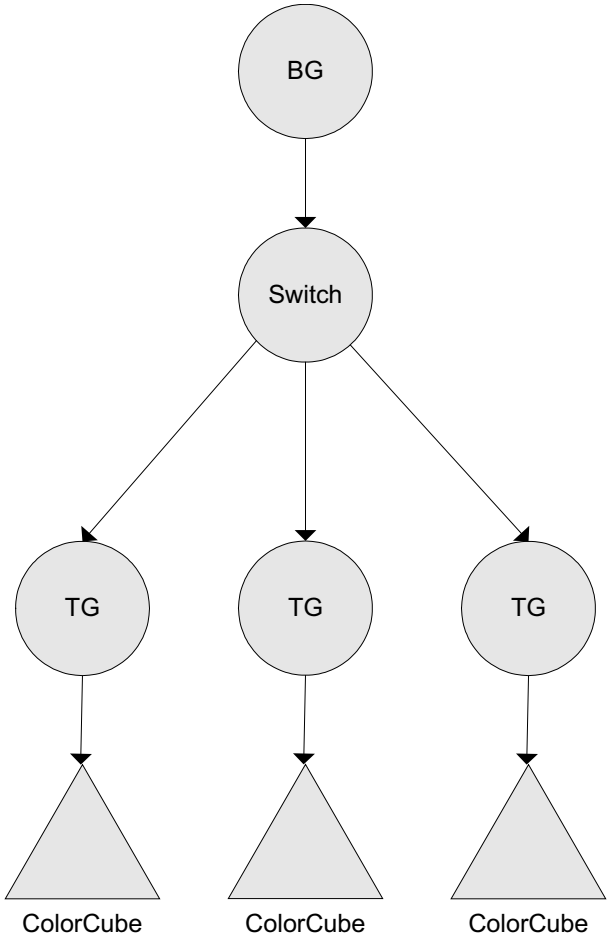


Figure 2.9: The scene graph for the `Switch` example program. This is similar to the scene graph from the `TransformGroup` example except that `Switch` node has been placed between the `BranchGroup` and the three `TransformGroup` nodes. The `Switch` node can ultimately be used to control which of the three `TransformGroup` branches are rendered.

```
0  import javax.media.j3d.*;
   import javax.vecmath.*;
   import com.sun.j3d.utils.geometry.*;
   import java.util.BitSet;

5  public class SwitchExample extends BasicSceneWithMouseControl{

     public static void main(String args[]){new SwitchExample();}

     public BranchGroup createContentBranch()
10   {
       BranchGroup root = new BranchGroup();

       // Bottom left ColorCube
       Transform3D t1 = new Transform3D();
15     t1.setTranslation(new Vector3f(-0.25f, -0.25f, 0.0f));
       TransformGroup tg1 = new TransformGroup(t1);
       ColorCube c1 = new ColorCube(0.2);
```

```
        tg1.addChild(c1);

        // Bottom right ColorCube
        Transform3D t2 = new Transform3D();
        t2.setTranslation(new Vector3f(0.25f, -0.25f, 0.0f));
        TransformGroup tg2 = new TransformGroup(t2);
        ColorCube c2 = new ColorCube(0.2);
        tg2.addChild(c2);

        // Top ColorCube
        Transform3D t3 = new Transform3D();
        t3.setTranslation(new Vector3f(0.0f, 0.25f, 0.0f));
        TransformGroup tg3 = new TransformGroup(t3);
        ColorCube c3 = new ColorCube(0.2);
        tg3.addChild(c3);

        // Child mask
        BitSet bitSet = new BitSet(3);
        //bitSet.set(0);
        bitSet.set(1);
        //bitSet.set(2);
        Switch switchGroup = new Switch(Switch.CHILD_MASK, bitSet);

        switchGroup.addChild(tg1);
        switchGroup.addChild(tg2);
        switchGroup.addChild(tg3);

        root.addChild(switchGroup);
        return root;
    }
}
```

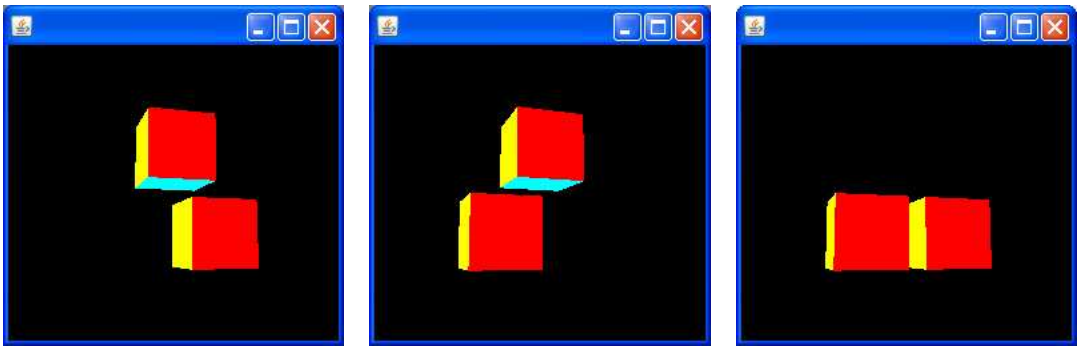
The main class of this program also extends the `BasicSceneWithMouseControl` class. As in the previous example, the scene graph is implemented by overwriting the `createContentBranch()` method. The main differences in this version are:

- A `Switch` node is added between the `BranchGroup` node and the `TransformGroup` nodes to control which of the `ColorCube` nodes are ultimately rendered.
- The children of the `Switch` node to be rendered are indicated using a `BitSet` object where each bit indicates whether or not the associated child should be displayed.

Three versions of the output obtained when this scene graph is rendered are illustrated in Figure 2.10.

2.4.5 `SharedGroup`

A `SharedGroup` enables a subgraph to be shared between different groups via `Link` leaf nodes. The essentially allows the same content to be replicated several times within a single scene. A `SharedGroup` node has a number of special properties:



(a) (b) (c)

Figure 2.10: Three renderings of the scene graph illustrated in Figure 2.9. In each case the `BitSet` object that controls which children are to be rendered has a difference configuration: (a) 011, (b) 101 and (c) 110.

- A `SharedGroup` may be referenced by one or more `Link` leaf nodes. Any runtime changes to a node or component object in a shared subgraph affect all graphs that refer to that subgraph.
- Only `Link` leaf nodes may refer to `SharedGroup` nodes. A `SharedGroup` node cannot have parents or be attached to a `Locale` object.
- A shared subgraph may contain any group node, except an embedded `SharedGroup` node as `SharedGroup` nodes cannot have parents. However, only the following leaf nodes may appear in a shared subgraph:
  - `Light`
  - `Link`
  - `Morph`
  - `Shape`
  - `Sound`
- An `IllegalSharingException` is thrown if any of the following leaf nodes appear in a shared subgraph represented by a `SharedGroup` object:
  - `AlternateAppearance`
  - `Background`
  - `Behaviour`
  - `BoundingLeaf`
  - `Clip`
  - `Fog`
  - `ModelClip`
  - `SoundScape`
  - `ViewPlatform`

The `SharedGroup` class defines the following capability bit:

- `ALLOW_LINK_READ`
  Indicates that this `SharedGroup` node allows list of `Link` objects that refer to this node to be read after the scene graph has gone live.

Example: The following example uses a `SharedGroup` object to create multiple instances of a single `ColorCube` object. The scene graph for the example is illustrated in Figure 2.11 and the Java 3D implementation of the scene graph is listed below:

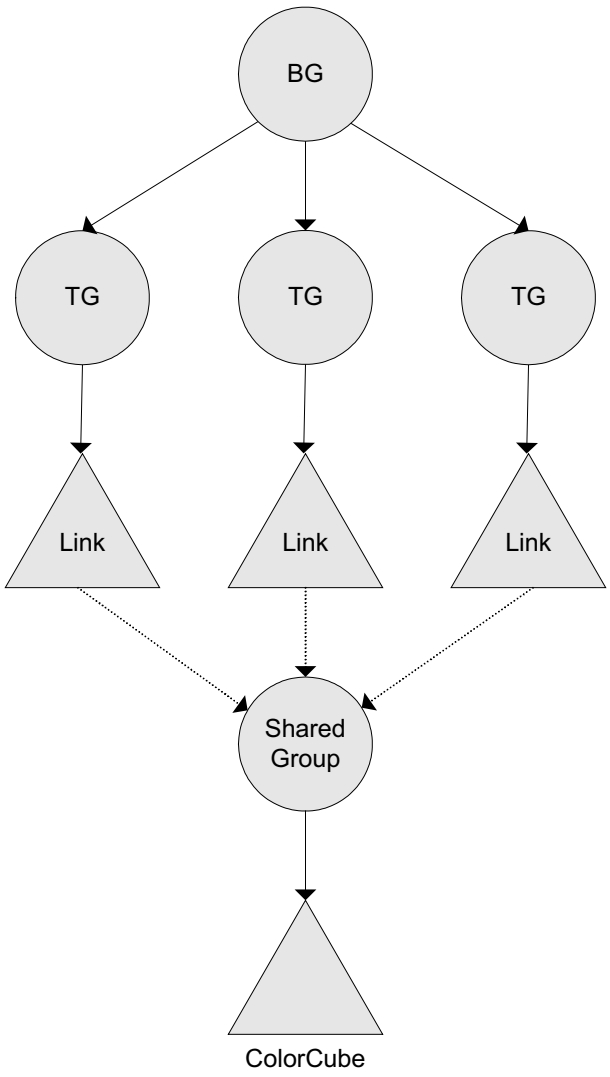


Figure 2.11: The scene graph for the `SharedGroup` example. This builds on the previous `TransformGroup` example, however, instead of having three separate instance of a `ColorCube` object, there is only one. This single instance is referenced three times via a `SharedGroup` object using three `Link` objects.

```
0 import javax.media.j3d.*;
  import javax.vecmath.*;
  import com.sun.j3d.utils.geometry.*;

  public class SharedGroupExample extends BasicSceneWithMouseControl{
5
    public static void main(String args[]){new SharedGroupExample();}
```

```
  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // Shared group containing ColorCube object
    SharedGroup sharedGroup = new SharedGroup();
    ColorCube cube = new ColorCube(0.2);
    sharedGroup.addChild(cube);

    // Bottom left Link object
    Transform3D t1 = new Transform3D();
    t1.setTranslation(new Vector3f(-0.25f, -0.25f, 0.0f));
    TransformGroup tg1 = new TransformGroup(t1);
    Link link1 = new Link();
    link1.setSharedGroup(sharedGroup);
    tg1.addChild(link1);

    // Bottom right Link object
    Transform3D t2 = new Transform3D();
    t2.setTranslation(new Vector3f(0.25f, -0.25f, 0.0f));
    TransformGroup tg2 = new TransformGroup(t2);
    Link link2 = new Link();
    link2.setSharedGroup(sharedGroup);
    tg2.addChild(link2);

    // Top link object
    Transform3D t3 = new Transform3D();
    t3.setTranslation(new Vector3f(0.0f, 0.25f, 0.0f));
    TransformGroup tg3 = new TransformGroup(t3);
    Link link3 = new Link();
    link3.setSharedGroup(sharedGroup);
    tg3.addChild(link3);

    root.addChild(tg1);
    root.addChild(tg2);
    root.addChild(tg3);

    return root;
  }
}
```

As in the previous examples, the main class of this program extends the version of the `BasicScene` class with mouse support and overwrites the `createContentBranch()` method. The main difference between this example and the earlier `TransformGroup` example is that instead of having three separate instances of a `ColorCube` object, there is only one. This is achieved by:

- Creating a `SharedGroup` rooted scene graph that contains a single `ColorCube` object.
- Replacing the `ColorCube` objects in the `TransformGroup` example with `Link` nodes.

- Creating references to the `SharedGroup` object from each of the `Link` nodes.

The output generated by this program is the same as the output generated by the `TransformGroup` example (see Figure 2.8), however, it should be noted that the level of repetition in this program is much less than in the `TransformGroup` example.

2.4.6 `ViewSpecificGroup`

The `ViewSpecificGroup` node is a Group whose descendants are rendered only on a specified set of views. It contains a list of view on which its descendants are rendered. Methods are provided to add views, removes views and enumerate the list of view maintained by this node. The list of views is initially empty. This means that by default, the children of this group will not be rendered on any view.

The `ViewSpecificGroup` defines a set of methods that are used to manage its associated views. These include:

- `void addView(View view)`
  Appends the specified `View` to this node's list of views.
- `Enumeration getAllViews()`
  Returns an enumeration containing this `ViewSpecificGroup` node's list of views.
- `View getView(int index)`
  Retrieves the `View` object at the specified index from the node's list of views.
- `int numViews()`
  Returns the number of `View` objects in this node's list of views.
- `void removeView(View view)`
  Removes the specified `View` object from this node's list of views.

The `ViewSpecificGroup` defines the following capability bits:

- `ALLOW_VIEW_READ`
  Indicates that the `ViewSpecificGroup` allows its list of views to be read after the scene graph has gone live.
- `ALLOW_VIEW_WRITE`
  Indicates that the `ViewSpecificGroup` allows it list of views to be modified after the scene graph has gone live.

2.5 Shapes Nodes

Geometric objects in a Java 3D scene are represented using shapes that are instances of the `Shape3D` class. The `Shape3D` class contains a list of one or more `Geometry` component object and a single `Appearance` component object (see Figure 2.12). The `Geometry` components define the shape node's structure. The `Appearance` object specifies the appearance attributes of the `Shape3D` object including: colour, material and texture.

An instance of a `Shape3D` object can be created using the following constructors:

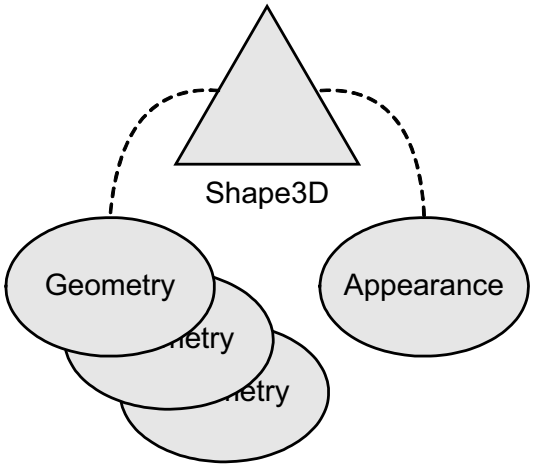


Figure 2.12: An illustration of a Shape3D object. This object maintains a single reference to an `Appearance` object and can maintain multiple references to `Geometry` objects.

- `Shape3D()`
  Constructs a new `Shape3D` object with a null appearance and a null geometry.
- `Shape3D(Geometry geometry, Appearance appearance)`
  Constructs a new `Shape3D` object with the specified appearance and geometry components.

Several methods are defined by the `Shape3D` class to manage its appearance and geometry. These include:

- `void addGeometry(Geometry geometry)`
  Appends the specified geometry component to the list of geometry components maintained by this `Shape3D` object.
- `Geometry getGeometry(int index)`
  Retrieves the geometry component at the specified index from the list of geometry components maintained by this `Shape3D` object.
- `void setAppearance(Appearance appearance)`
  Sets the appearance component of this `Shape3D` node.
- `Appearance getAppearance()`
  Retrieves the appearance component of this `Shape3D` node.

The `Shape3D` class also defines a series of capability bits including:

- `ALLOW_APPEARANCE_READ`
  Indicates that the `Shape3D` node allows read access to its appearance component after the scene graph has gone live.
- `ALLOW_APPEARANCE_WRITE`
  Indicates that the `Shape3D` node allows write access to its appearance component after the scene graph has gone live.

- `ALLOW_GEOMETRY_READ`
  Indicates that the `Shape3D` node allows read access to its geometry components after the scene graph has gone live.

- `ALLOW_GEOMETRY_WRITE`
  Indicates that the `Shape3D` node allows write access to its geometry components after the scene graph has gone live.

2.5.1 Subclasses of `Shape3D`

Two classes extend `Shape3D` to provide additional functionality. These classes are: `OrientatedShape3D` and `ColorCube`. `OrientatedShape3D` is used to represent a shape that is always oriented towards the viewer. The `OrientatedShape3D` class defines three modes of orientation:

- `ROTATE_ABOUT_AXIS`
  Causes the `OrientatedShape3D` to rotate about an axis in order to face the viewer.

- `ROTATE_ABOUT_POINT`
  Causes the `OrientatedShape3D` to rotate about a point to face the viewer.

- `ROTATE_NONE`
  Causes the `OrientatedShape3D` to be stationary. In this mode the `OrientatedShape3D` object acts the same as a standard `Shape3D` object.

An instance of an `OrientatedShape3D` object can be created using the following constructor:

- `OrientatedShape3D(Geometry g, Appearance a, int mode, Point3f pt)`
  Constructs an `OrientatedShape3D` object with the specified appearance and geometry attributes. The `mode` argument specifies whether the `OrientedShape3D` rotates about a point or an axis, and the `Point3f` argument defines the point around which the `OrientatedShape3D` rotates.

  – Note that there is another version of this constructor that takes a `Vector3f` parameter in place of the `Point3f` parameter. This version is used to construct an `OrientatedShape3D` object that rotates about an axis.

The `ColorCube` class was discussed earlier in this chapter. It represents a cube with sides two metres in size that has a different color on each of its six faces. An instance of a `ColorCube` object can be created using the following constructor:

- `ColorCube(double scale)`
  Constructs an instance of a `ColorCube` object that is scaled by the specified value. A scale value of 1.0 results in a `ColorCube` object with sides two metres in size.

Note that `ColorCube` is not part of the core Java 3D API, instead it is defined in the extension `com.sun.j3d.utils.geomtery` package.

2.5.2 Primitives

A series of primitive shapes are defined in the `com.sun.j3d.utils.geometry` package. All of these classes extend the class `Primitive` which is the base class for all Java 3D primitives. The class `Primitive` extends the class `Group` and manages the branch graph that represents the primitive shape. There are a total of four subclasses of primitive: `Box`, `Cone`, `Cylinder` and `Sphere`. Objects of these classes can be created using the following constructors:

- `Box()`
  Constructs a `Box` object with dimensions of 2.0 metres in the x, y and z directions and a null appearance.
- `Box(float xdim, float ydim, float zdim, Appearance app)`
  Constructs a `Box` object with the specified dimensions and appearance.
- `Cone()`
  Constructs a `Cone` object with a base radius of 1.0 metres, a height of 2.0 metres and a null appearance.
- `Cone(float radius, float height)`
  Constructs a `Cone` object with the specified dimensions and a null appearance.
- `Cylinder()`
  Constructs a `Cylinder` object with a radius of 1.0 metre, a height of 2.0 metres and a null appearance.
- `Cylinder(float radius, float height, Appearance appearance)`
  Constructs a `Cylinder` object with the specified dimensions and appearance.
- `Sphere()`
  Constructs a `Sphere` object with a radius of 1.0 metre and a null appearance.
- `Sphere(float radius, Appearance appearance)`
  Constructs a sphere object with the specified radius and appearance.

2.6 Geometry

The geometry of a shape represents its structure. There are several different types of geometry supported by Java 3D. Geometry is represented in Java 3D by the abstract class `Geometry`. There are four subclasses of the `Geometry` class that can be instantiated. These are:

- `Text3D` - A 3D representation of a specified text string that has an associated font as well as other characteristics that determine the structure of the geometric representation of the string.
- `Raster` - Allows a 2D raster image to be attached to a 3D location. The image is represented by a single point in the resulting scene.
- `GeometryArray` - Represents different types of geometry using a series of arrays that contain positional coordinates, colours, normals and texture coordinates.

- `CompressedGeometry` - Used to store geometry in a compressed format. Using compressed geometry can increase the speed of sending objects over the network.
  - **Note:** This class has been deprecated (as of Java 3D version 1.4) and will not be dealt with in this course.

2.6.1 `Text3D`

A `Text3D` object is used to represent a text string in the form of 3D geometry. A text 3D object has the following parameters:

- **A font** - in the form of a `Font3D` object describes the font characteristics of the text string represented by the `Text3D` object. The characteristics include the following:
  - The font family (Helvetica, Courier, etc.)
  - The font style (italic, bold, etc.)
  - The font size.
    - Note that the font size is specified in points but interpreted in metres. Consequently specifying a 12 point font will result in characters with a size of 12 metres. This means that the smallest possible font size is 1 meter since the size of a font is specified using an `integer` primitive.
  - An extrusion path that determines how the two-dimensional font should be rendered in 3D.
- **A text string** - that represents the text to be converted into geometry.
- **A position** - that determines the initial position of the `Text3D` object.
- **An alignment** - that specifies how the glyphs in the string are placed in relation to the position parameter. The valid values are:
  - `ALIGN_CENTER` - the centre of the string is place at the point represented by the position parameter.
  - `ALIGN_FIRST` - the first character of the string is placed at the point represented by the position parameter.
  - `ALIGN_LAST` - the last character of the string is placed at the point represented by the position parameter.
- `A path` - that specifies how succeeding glyphs in the string are placed in relation to the previous glyph. The valid values are:
  - `PATH_LEFT` succeeding glyphs are placed to the left of the current glyph.
  - `PATH_RIGHT` succeeding glyphs are placed to the right of the current glyph.
  - `PATH_UP` succeeding glyphs are placed above the current glyph.
  - `PATH_DOWN` succeeding glyphs are placed below the current glyph.
- `A character spacing` - that defines the spacing in addition to the regular spacing between glyphs as defined in the `Font` object.

Note: Characters and Glyphs
A character is a symbol that represents an item such as a letter, a digit, or a punctuation in an abstract way. For example, 'g' is a character. A glyph is a shape used to render a character or a sequence of characters. In simple writing systems, such as Latin, typically one glyph represents one character. In general, however, characters and glyphs do not have a one-to-one correspondence. For example, the character 'á' (a with acute), can be represented by two glyphs: one for 'a' and one for '´'. On the other hand, the two-character string 'fi' can be represented by a single glyph, an 'fi' ligature. In complex writing systems, such as Arabic or the South and South-East Asian writing systems, the relationship between characters and glyphs can be more complicated and involve context-dependent selection of glyphs as well as glyph reordering. A font encapsulates the collection of glyphs needed to render a selected set of characters as well as the tables needed to map the sequences of characters to corresponding sequences of glyphs.

The most comprehensive constructor for a `Text3D` object has the following format:

- `Text3D(Font3D font, String str, Point3f pos, int align, int path)` Creates a `Text3D` object with representing the string argument, with the specified font, location, alignment and path.

Example: The following example demonstrate how a `Text3D` object can be created, configured and displayed in a Java 3D application.

```
import java.awt.*;
import javax.media.j3d.*;
import javax.vecmath.Vector3f;

import com.sun.j3d.utils.geometry.ColorCube;

public class Text3DExample extends BasicSceneWithMouseControl{

  public static void main(String args[]){new Text3DExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // Create the Font3D object
    Font font2d = new Font("Monospaced", Font.PLAIN, 1);
    Font3D font3d = new Font3D(font2d, new FontExtrusion());

    // Create the Text3D object and add it to a Shape3D object
    Text3D text = new Text3D(font3d, "hello world!");
    text.setAlignment(Text3D.ALIGN_CENTER);
    text.setPath(Text3D.PATH_RIGHT);
    Shape3D shape = new Shape3D(text);

    // Scale the Text3D object using a TransformGroup
    Transform3D t = new Transform3D();
    t.setScale(0.2);
    TransformGroup tg = new TransformGroup(t);
```

```
        tg.addChild(shape);

30

        root.addChild(tg);

        return root;
    }
35  }
```

The scene graph in this example consists of a `BranchGroup` with a single child that is a `TransformGroup`. The `TransformGroup` represents a scaling of 20% and applies this scale transformation to its child, which is a Text3D node.

The `Text3D` object has a plain monospaced font, a font size of 1 metre and the default extrusion parameters. The default extrusion parameters indicate that the rendered text has a thickness of 20 cm. The text represented by the `Text3D` object is centre aligned and rendered from left to right. The output generated by this program is illustrated in Figure 2.13

Figure 2.13: The output generated by the `Text3D` example. The string is centre aligned and rendered from left to right.

2.6.2 `Raster`

The `Raster` class extends the `Geometry` class to allow a raster image to be attached to a 3D location in a virtual world. It contains a 3D point that is defined in the local object coordinate system of the `Shape3D` node that references the `Raster` object. It also contains:

- A type specifier that indicates the type of image data.
- A clipping mode
- A reference to a `ImageComponent2D` object that represents the raster image to be rendered.
- A reference to a `DepthComponent` object.

- An integer x, y source offset.
- Image dimensions.

Example: The following example demonstrates how a `Raster` object can be created and rendered in a Java 3D application.

```
import java.io.*;
import javax.media.j3d.*;
import javax.imageio.ImageIO;
import java.awt.image.BufferedImage;
import javax.vecmath.*;

public class RasterExample extends BasicSceneWithMouseControl{

  public static void main(String args[]){new RasterExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    try{
    BufferedImage b = ImageIO.read(new File("greatwall.jpg"));

    // Create the Raster object
    Raster raster = new Raster(new Point3f(-0.85f, 0.65f, 0.0f),
       Raster.RASTER_COLOR,0,0,b.getWidth(), b.getHeight(),
       new ImageComponent2D(ImageComponent2D.FORMAT_RGBA,b),
       new DepthComponentFloat(b.getWidth(), b.getHeight()));

    // Use the Raster geomtry to create a Shape3D object
    Shape3D shape = new Shape3D(raster);
    root.addChild(shape);
    root.compile();
    }catch(Exception e){System.out.println(e.toString());}

    return root;
  }
}
```

This example creates a new `Raster` object that is attached to a `Shape3D` node. The `Shape3D` node is the child of a `BranchGroup` object which represents the root of the scene graph. The output obtained when this program is executed is illustrated in Figure 2.14.

2.6.3 `GeometryArray`

The `GeometryArray` class is used to represent different types of geometry in the form of points, lines and polygons (triangles or quadrilaterals). In each case, the relevant geometry is defined using a series of arrays that contain various vertex properties, hence the name `GeometryArray`. The vertex properties include: location, colour, normals and texture coordinates. The vertex data can be associated with an instance of `GeometryArray` in one of two ways:

Figure 2.14: The output generated by the `Raster` example. The position of the image has been selected so that it is located in the centre of the frame.

- **By copying:** This is the default mode and involves the relevant instance of the `GeometryArray` class making an internal copy of the vertex properties that are specified using the various `set()` methods of the `GeomteryArray` class.
- **By reference:** This mode was made available in Java 3D version 1.2 and allows the relevant vertex data to be accessed by reference, directly from the user's arrays. Special methods are provided to enable the specification of vertex properties by reference. These `set()` methods all include the word `Ref`. Data that is referenced by a live or compiled `GeometryArray` object may only be modified via the `updateData()` method, subject to the `ALLOW_REF_DATA_WRITE` capability bit being set. It is important that this rule isn't violated as the results are undefined if any geometry is modified outside the `updateData()` method.

The `GeometryArray` class defines a series of methods to set the different kinds of vertex properties. Each method has several variations to allow the properties to be set using different forms of data. Other variations allow different sections of a vertex property array to be set, for example, individual vertex properties, a range of vertex properties within an array or an entire vertex property array. In the case of coordinates, the following methods can be used to set individual vertex coordinates.

- `void setCoordinate(int index, double[] coordinate)`
- `void setCoordinate(int index, float[] coordinate)`
- `void setCoordinate(int index, Point3d coordinate)`
- `void setCoordinate(int index, Point3f coordinate)`

The first two methods specify a single vertex index and a three element array that represents the x, y and z coordinate information. The last two methods use a higher

level point object to represent that coordinate information. A range of coordinates to be set can also be specified using the following methods:

- `void setCoordinates(int index, double[] coords, int start, int length)`
- `void setCoordinates(int index, float[] coords, int start, int length)`
- `void setCoordinates(int index, Point3d[] coords, int start, int length)`
- `void setCoordinates(int index, Point3f[] coords, int start, int length)`

In each of these methods, the `index` argument specifies the starting index where the range of coordinates is to be placed in the relevant array maintained by the `GeomteryArray` object. The `start` and `length` arguments indicate where the range is the be extracted from the list of coordinates represented by the `coordinates argument`. Finally, the entire set of coordinates for a `GeometryArray` object can be specified using the following methods:

- `void setCoordinates(int index, double[] coordinates)`
- `void setCoordinates(int index, float[] coordinates)`
- `void setCoordinates(int index, Point3d[] coordinates)`
- `void setCoordinates(int index, Point3f[] coordinates)`

The methods for setting colours, normals and texture coordinates are all a variation of the coordinate methods outlined above.

It should be noted that `GeometryArray` is an abstract class and can not be instantiated. Only subclasses of `GeometryArray` can be used to define geometry. In all cases the number of vertices and the vertex format must be specified in the constructor of the relevant subclass.

The vertex format argument is a mask indicating which components are present in each vertex. This is specified as one or more individual flags that are bitwise ORed together to indicate what data will be specified for each vertex. The flags include:

- `COORDINATES` indicates the inclusion of vertex locations. It should be noted that this flag must always be present.
- `NORMALS` indicates the inclusion of per-vertex normals.
- One of the following colour flags:
  - `COLOR_3` indicates the inclusion of per-vertex colour information (without alpha component).
  - `COLOR_4` indicates the inclusion of per-vertex colour information (with alpha component).
- One of the following texture coordinate flags to indicate the inclusion of 2D, 3D or 4D per-vertex texture coordinates:
  - `TEXTURE_COORDINATE_2D`

  - TEXTURE_COORDINATE_3D
  - TEXTURE_COORDINATE_4D

- BY_REFERENCE to indicate that the vertex data is passed by reference rather than by coping.

There are a range of subclasses of GeometryArray. Each subclass interprets the coordinate information in a different way to represent different types of basic geometry. These subclasses include:

- PointArray draws its array of vertices as individual points.
- LineArray draws its array of vertices as individual line segments. Each pair of verties defines a line to be drawn.
- TriangleArray draws its array of vertices as individual triangles. Each group of three vertices defines a triangle to be drawn.
- QuadArray draws its array of vertices as individual quadrilaterals. Each group of four vertices defines a quadrilateral to be drawn.
- GeometryStripArray is an abstract class that is extended to allow the definition of compound geometry. It is extended by the following classes:
  - LineStripArray draws its array of vertices as a set of connected lines strips. An array of per-strip vertex counts specifies where separate strips appear in the vertex array. For every strip in the set, each vertex, beginning with the second vertex in the array, defines a line segment to be drawn from the previous vertex to the current vertex.
  - TriangleStripArray draws its array of vertices as a set of connected triangle strips. An array of per-strip vertex counts specifies where the separate strips appear in the vertex array. For every strip in the set, each vertex, beginning with the third vertex in the array, defines a triangle to be drawn using the current vertex and the two previous vertices.
  - TriangleFanArray draws its array of vertices as a set of connected triangle fans. An array of per-strip vertex counts specify where the separate fans appear in the vertex array. For every strip in the set, each vertex, beginning with the third vertex in the array defines a triangle to be drawn using the current vertex, the previous vertex and the first vertex.
- IndexedGeometry contains separate integer arrays that index into the arrays of positional coordinates, colours, normals, texture coordinates, and vertex attributes. These index arrays specify how vertices are connected to form geometry primitives. This class is extended to create indexed versions of the primitive geometry types that were just discussed and include:
  - IndexedPointArray
  - IndexedLineArray
  - IndexedTriangleArray
  - IndexedQuadArray
  - IndexedGeometryStripArray

* `IndexedLineStripArray`
* `IndexedTriangleStripArray`
* `IndexedTriangleFanArray`

Examples of the types of geometry represented by these classes is illustrated in Figure 2.15.

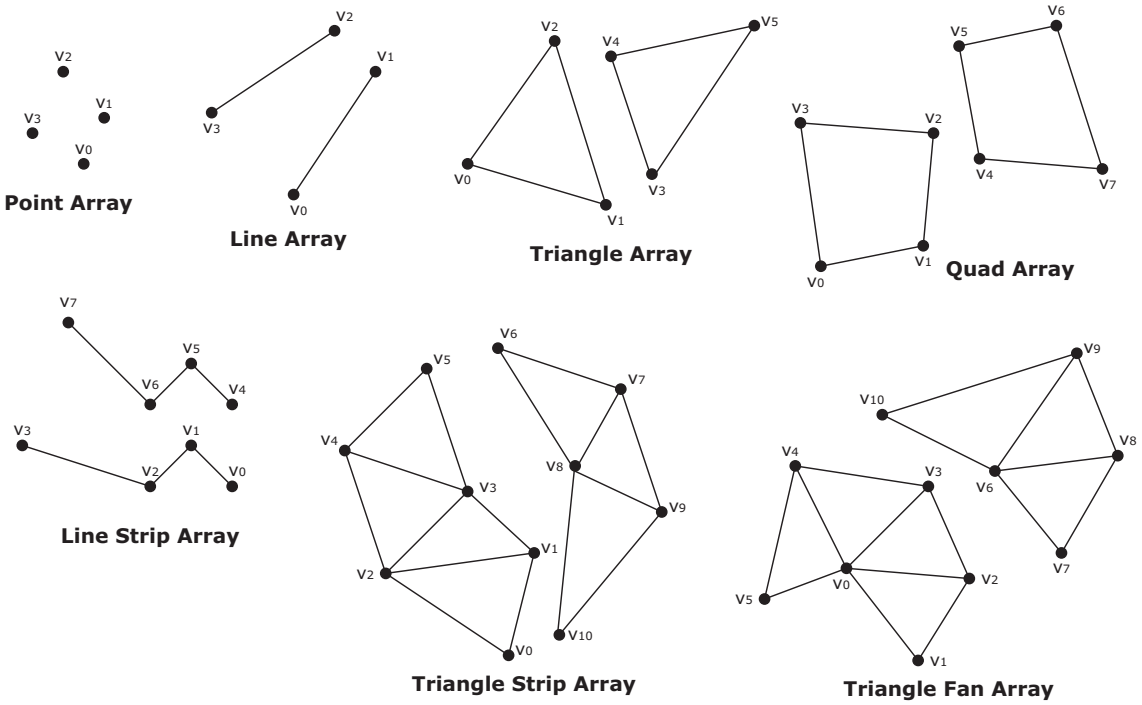


Figure 2.15: An illustration of the seven basic types of geometry supported by the Java 3D API

2.6.4 Defining polygons

There are two important facts that you need to be aware of when defining polygons:

1. The order of the vertices is important when defining the orientation of the polygon.

2. Vertices of a quad must form a convex, planar polygon.

Polygons are defined with front and back faces. The orientation is used to determine which way the polygon is facing for lighting operations. The orientation can also be used to remove, or cull, polygons that are facing away from the viewer. The front side of a polygon is defined to be the orientation where the vertices of the polygon form an anticlockwise loop. The two possible orientations of a polygon are illustrated in Figure 2.16.
The other detail is that the points of a quadrilateral must form a convex, planar polygon. The quadrilateral must be convex or some graphics hardware may render the quadrilateral incorrectly. The quadrilateral must be planar because a convex quadrilateral can turn non-convex if the quadrilateral is not planar. Examples of convex and non-convex quadrilateral are illustrated in Figure 2.17.

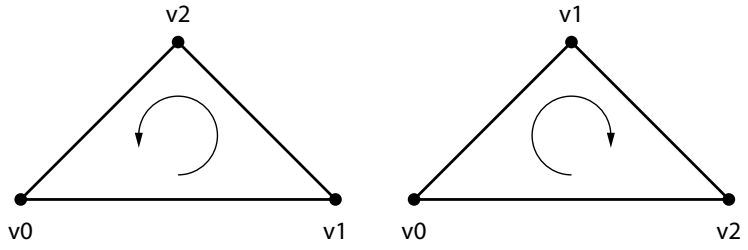


Figure 2.16: Examples of front facing (a) and rear facing (b) triangles.

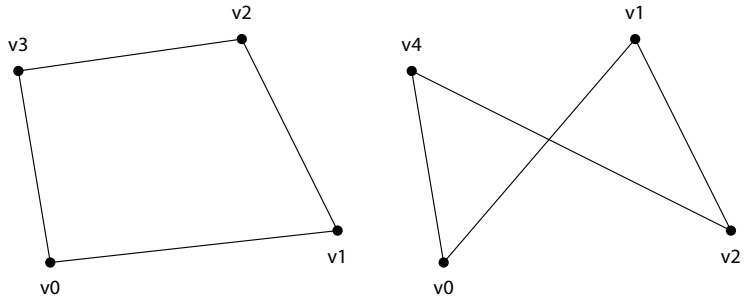


Figure 2.17: Examples of convex (a) and non-convex (concave) (b) quadrilaterals.

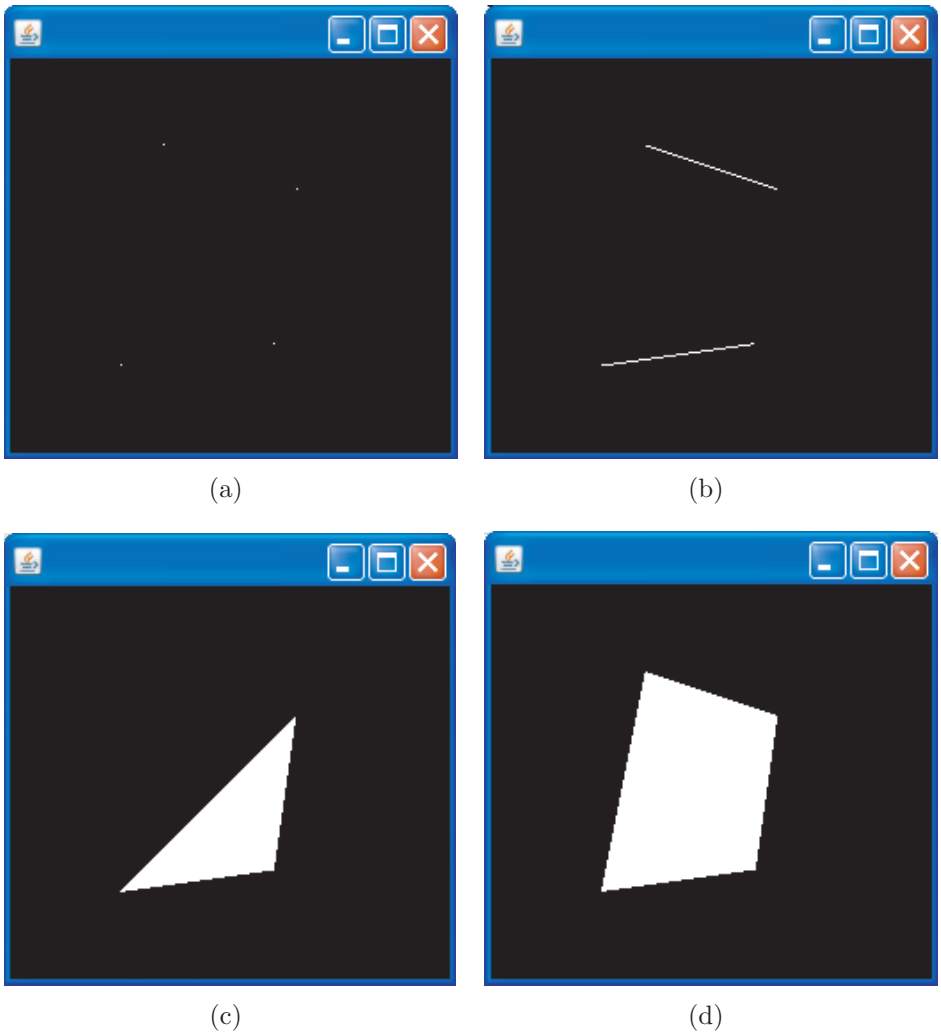
2.6.5 Simple Geometry

It is possible to create simple geometry using the `PointArray`, `LineArray`, `TriangleArray` and `QuadArray` classes. The following example demonstrates how these classes can be used in conjunction with the `Shape3D` class to create basic 3D content. Note that the type of object assigned to the `GeometryArray` reference must be updated to indicate the type of geometry being used.

```
0
import javax.media.j3d.*;

public class SimpleGeometryExample extends BasicSceneWithMouseControl
{
5   //
  public static void main(String args[]){new SimpleGeometryExample();}

  public BranchGroup createContentBranch()
  {
10   BranchGroup root = new BranchGroup();


    float [] coordinates = {-0.5f, -0.5f,  0.0f,
                             0.2f, -0.4f,  0.0f,
15                           0.3f,  0.3f,  0.0f,
                            -0.3f,  0.5f,  0.0f
                            };
```

```
        // Create a geometry array from the specified coordinates
        GeometryArray geometryArray = new LineArray(4,
            GeometryArray.COORDINATES
            );

        geometryArray.setCoordinates(0, coordinates);

        // Create a Shape3D object using the GeometryArray
        Shape3D shape = new Shape3D(geometryArray, null);
        root.addChild(shape);

        root.compile();

        return root;
    }
}
```

This program creates a `GeometryArray` object that contains only coordinates. This `GeometryArray` object is subsequently used to create a `Shape3D` object with a null appearance. The `Shape3D` object is then added to a `BranchGroup` that represents the root of the scene graph. The various outputs generated by this program are illustrated in Figure 2.18.



(a) (b)

(c) (d)

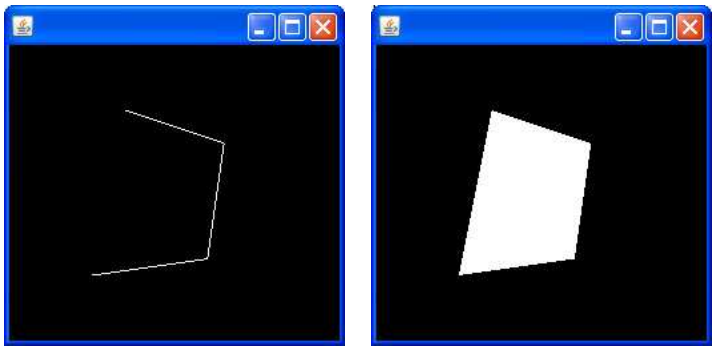
Figure 2.18: Examples of the geometry that can be created using the four simple geometry classes provided by Java 3D (a) A set of points defined using a `PointArray` object (b) A pair of lines defined using a `LineArray` object (c) A triangle defined using a `TriangleArray` object (d) A quadrilateral defined using a `QuadArray` object.

2.6.6 Strip Geometry

Strip Geometry reduces the need to repeatedly specify the same vertices when defining continuous pieces of geometry. It is possible to create strip geometry using instances of the `LineStripArray`, `TriangleFanArray` and `TriangleStripArray` classes. The following example demonstrates how the `LineStripArray` class can be used to create a line consisting of three connected segments.

```
0  import javax.media.j3d.*;

   public class StripGeometryExample extends BasicSceneWithMouseControl{

     public static void main(String args[]){new StripGeometryExample();}
5
     public BranchGroup createContentBranch()
     {
       BranchGroup root = new BranchGroup();

10     // Define the coordinates to be used
       float [] coordinates = {-0.5f, -0.5f,  0.0f,
                               0.2f, -0.4f,  0.0f,
                               0.3f,  0.3f,  0.0f,
                              -0.3f,  0.5f,  0.0f};
15
       // Specify a single strip that used all 4 coordinates
       int [] stripVertexCounts = {4};

       GeometryArray geometryArray = new TriangleFanArray(4,
20         GeometryArray.COORDINATES, stripVertexCounts);

       geometryArray.setCoordinates(0, coordinates);

       Shape3D shape = new Shape3D(geometryArray);
25     root.addChild(shape);

       root.compile();

       return root;
30   }
   }
```

This program begins by defining an array of single precision floating point coordinates that represent the vertices of the line strip that is being created. An array of vertex counts is also created. This size of this array represents the number of line strips to be generated, and each element defines the number of vertices to be used in the associated strip. The constructor to the `LineStripArray` object requires the total number of vertices, the vertex format and the array of vertex counts. The coordinates are associated with the constructed `LineStripArray` object and a `Shape3D` object is created to represent the line strip defined by the coordinates. The output of different versions of this program are illustrated in Figure 2.19



(a) (b)

Figure 2.19: The same set of vertices used to create a `LineStripArray` (a) and a `TriangleFanArray` (b)

Note that an additional argument is required in the `GeometryStripArray` constructor to specify the number of strips and the number of vertices associated with each strip. This is achieved using an integer array where the number of elements in the array specifies the number of strips, and each array element specifies the number of vertices associated with the relevant strip.

2.6.7 Creating Complex Geometry

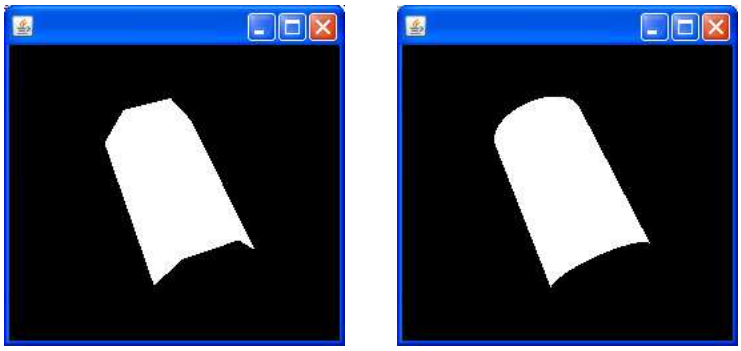
It would be tedious to define complex geometry by specifying all of the coordinates manually. An alternative would be to specify the attributes associated with the geometry and then use a program to automatically generate the required vertex coordinates. The following example demonstrates how a fully configurable cylinder can be created using a for loop and a `QuadArray` geometry.

```
0  import javax.media.j3d.*;

   public class CylinderExample extends BasicSceneWithMouseControl{

     public static void main(String args[]){new CylinderExample();}
5
     private Geometry createGeometry() {

       // Define the attributes for the cylinder
       float radius = 0.3f;
10     float height = 1.0f;
       int faces = 60;

       float angle = 0.0f;
       double angleIncrement = (2*Math.PI)/faces;
15
       // create an empty array of floats to hold the coordinates
       float coordinates[] = new float[faces*4*3];

       for(int f=0; f<faces; f++)
20     {
         // Generate the four coordinates required for each face
```

```
        float x1 = (float)(radius*Math.cos(angle));
        float z1 = (float)(radius*Math.sin(angle));
        angle -= angleIncrement;
        float x2 = (float)(radius*Math.cos(angle));
        float z2 = (float)(radius*Math.sin(angle));

        // Populate the coordinates array
        coordinates[f*4*3] = x1;
        coordinates[f*4*3+1] = -height/2.0f;
        coordinates[f*4*3+2] = z1;

        coordinates[f*4*3+3] = x2;
        coordinates[f*4*3+4] = -height/2.0f;
        coordinates[f*4*3+5] = z2;

        coordinates[f*4*3+6] = x2;
        coordinates[f*4*3+7] = height/2.0f;
        coordinates[f*4*3+8] = z2;

        coordinates[f*4*3+9] = x1;
        coordinates[f*4*3+10] = height/2.0f;
        coordinates[f*4*3+11] = z1;
      }

      QuadArray quadArray = new QuadArray(faces*4*3,
          GeometryArray.COORDINATES);
      quadArray.setCoordinates(0, coordinates);
      return quadArray;

    }

    public BranchGroup createContentBranch()
    {
      BranchGroup objRoot = new BranchGroup();

      // Add the cylinder to the scene graph
      objRoot.addChild(new Shape3D(createGeometry()));

      return objRoot;
    }
}
```

The cylinder geometry is created using a for loop. Each iteration of the for loop creates an individual face of the cylinder. The height of the cylinder is set to 1 metre and its radius is set to 30 cm. The number of faces to be generated is set to 60 so the resulting geometry will be smooth. The cylinder geometry is ultimately used to create a `Shape3D` object with the default appearance. Renderings generated by different versions of this program are illustrated in Figure 2.20.

Exercise: Write a program to create a sphere. The program should take inputs that represent the of the radius and complexity of the sphere. Instances of the `TriangleFanArray` and `TriangleStripArray` classes should be used to define the



(a) (b)

Figure 2.20: Examples of geometry representing a cylinder that was created using a for loop. (a) a cylinder with 6 faces and (b) a cylinder with 60 faces.

structure of the sphere geometry.

2.6.8 Indexed Geometry

In many cases the vertices in a geometric object are repeated as the boundaries of continuous regions are represented using common vertices. A good example of this is a simple cube. A cube has a total of eight corners, i.e. eight vertices must be specified in order to completely define a cube. The simplest way to create a cube using Java 3-D is to use quads to define each face of the cube. A total of six quads are required and each quad has four vertices, i.e. a total of 24 vertices where each vertex is used three times, see Figure 2.21.

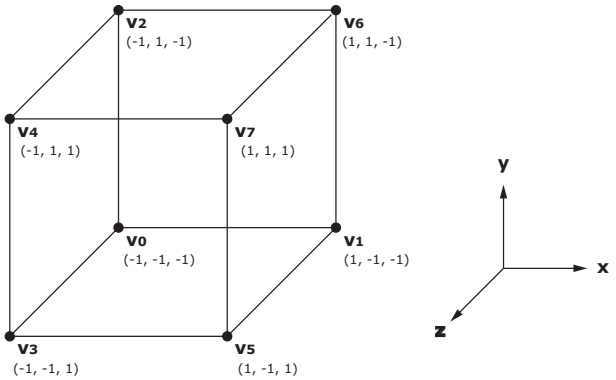


Figure 2.21: A simple cube consisting of six faces and eight vertices.

It is possible to avoid repeatedly defining the same vertices by using one of the subclasses of `IndexGeometryArray`. Using this class all of the required vertices are defined once and the structure of the geometry is determined by indices into the list of vertices. In the case of the cube, the eight required vertices are defined and stored in a `float` array. The structure of the geometry is then determined by specifying the required vertices using indices into the array of coordinates. The following example demonstrates how the cube geometry can be implemented using a `IndexedQuadArray`.

```
import javax.media.j3d.*;

public class IndexedGeometryExample extends BasicSceneWithMouseControl{

  public static void main(String args[]){new IndexedGeometryExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // Defines the 8 vertices for the cube geometry
    float [] points = {-0.5f, -0.5f, -0.5f,
                        0.5f, -0.5f, -0.5f,
                       -0.5f,  0.5f, -0.5f,
                       -0.5f, -0.5f,  0.5f,
                       -0.5f,  0.5f,  0.5f,
                        0.5f, -0.5f,  0.5f,
                        0.5f,  0.5f, -0.5f,
                        0.5f,  0.5f,  0.5f};

    // Defines the 24 indices for the cube geometry
    int [] indices = {0, 3, 4, 2,    // left face
                      0, 1, 5, 3,    // bottom face
                      0, 2, 6, 1,    // back face
                      7, 5, 1, 6,    // right face
                      7, 6, 2, 4,    // top face
                      7, 4, 3, 5};   // front face

    IndexedQuadArray quadArray = new IndexedQuadArray(8,
        GeometryArray.COORDINATES, 24);
    quadArray.setCoordinates(0, points);
    quadArray.setCoordinateIndices(0, indices);

    Shape3D cube = new Shape3D(quadArray);

    root.addChild(cube);
    root.compile();

    return root;
  }
}
```

The program defines the eight vertices that are illustrated in Figure 2.21. It also defines the indices that indicate how the list of vertices is to be used to generate the six faces of the cube. An `IndexedQuadArray` object is then created and the vertex count, the vertex format and the index count are all specified in the constructor. The array of vertex coordinates and the array of indices are subsequently supplied to the constructed `IndexedQuadArray` which is ultimately used to create a `Shape3D` object that is then added to the root of the scene graph.

It should be noted that the use of subclasses of `IndexedGeometryArray` allow ge-

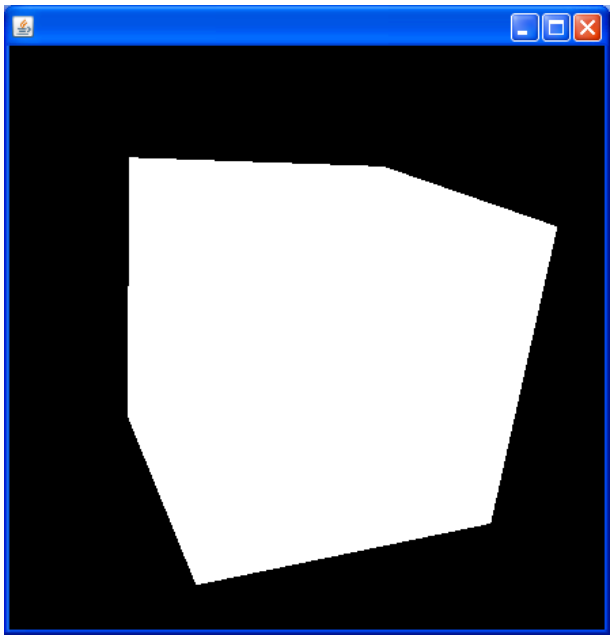


Figure 2.22: A cube created using a suitably constructed `IndexedQuadArray` geometry. The cube is rendered using the default appearance.

ometry to be defined in an efficient manner. In the case of the cube example, eight vertices (3 × `float`) and 24 integer primitives were defined. In Java, `float` and `int` primitives both require four bytes of storage. Consequently, the cube geometry requires $24 \times 4 + 3 \times 8 \times 4$ bytes or 192 bytes. If a standard `QuadArray` were used then the number of bytes required would be $24 \times 3 \times 4 = 288$ bytes. So, in addition to simplifying the definition of geometry, the use of subclasses of `IndexedGeometryArray` also reduce the amount of memory required to represent geometry.

2.6.9 Loading Geometry from Files

It is possible to import shapes into a virtual world from a file. Loading data from files facilitates the use of content created using other application, thus removing the need to create content explicitly or using the utility classes. There are a variety of applications that can be used to create 3D content and export it to a format that can be understood by Java 3D. The types of files discussed in this section are Wavefront .obj files. This is a commonly used format and many objects are available in this format on the net.

The interface for importing data from files consists of two classes, `Loader` and `Scene`. A `Loader` is used to load 3D content from a file. Once the content has been loaded it is represented in the form of a `Scene`. The `Scene` contains a scene graph for the content as well as methods that enabled individual nodes within the scene to be looked up.

The `Loader` interface is defined in the `com.sun.j3d.loaders` package. This interface defines the methods used to specify the content file as well as options for loading. The loader for .obj files is called `ObjectFile` and it implements the `Loader` interface. An `ObjectFile` object can be created using the following constructor:

- `ObjectFile(int flags)`

The `flags` parameter specifies how the data is to be created. The possible values for this parameter are:

- `RESIZE` - Indicates that the loaded geometry should be resized as it is loaded so that the object fits into the range (-1.0, -1.0, -1.0) to (1.0, 1.0, 1.0).
- `REVERSE` - Is used to correct content that was created with the polygons orientated the wrong way.
- `STRIPIFY` - Tells the loader to use the `Stripifier` utility to improve the rendering performance for the scene by combining adjacent triangles into strips.
- `TRIANGULATE` - Is used if the file contains complex (e.g. concave or nonplanar) polygons that must be broken up by the `Triangulator` utility so they can be rendered by Java 3D

These values can be OR'd together so that multiple flags can be specified simultaneously. A `Loader` can be used to load a file from the local system or a remote URL using the following methods:

- `Scene load(String filename)`
  Loads a scene from the specified file on the local system.
- `Scene load(URL url)`
  Loads a scene from the specified URL.

The load method returns an object that implements the `Scene` interface. This interface can be used to access several important pieces of scene information, the most significant being a `BranchGroup` containing the scene graph created by the loader. This branch group can be obtained by calling the following method:

- `BranchGroup getSceneGroup()`
  This method returns the overall `BranchGroup` containing the scene loaded by the loader.

The following example demonstrates how content can be loaded from a .obj file and rendered using Java 3D.

```
0  import javax.media.j3d.*;
   import javax.vecmath.*;

   import com.sun.j3d.loaders.*;
   import com.sun.j3d.loaders.objectfile.*;
5
   public class LoaderExample extends BasicSceneWithMouseControlAndLights
   {
     public static void main(String args[]){new LoaderExample();}

10   public BranchGroup createContentBranch()
     {
       // Resize the scene to fit the display
```

```
    int flags = ObjectFile.RESIZE;

    // Create a new .obj loader
    ObjectFile f = new ObjectFile(flags);
    Scene s = null;

    try
    {
      s = f.load("skull.obj");
    }
    catch(Exception e)
    {
      System.out.println("error:"+e.toString());
    }
    BranchGroup root = s.getSceneGroup();

    root.compile();

    return root;
  }
}
```

This program begins by creating an `ObjectLoader` object with the `RESIZE` flag set so that the loaded scene fits the display. The scene is then loaded from a local file using the `load()` method of the `ObjectLoader` object. The `BranchGroup` object representing the loaded scene is obtained using the `getSceneGroup()` method of the loaded scene. The output generated by this example is illustrated in Figure 2.23

2.7 Appearance

A `Shape3D` node references a geometry node component that specifies what to render and an appearance node component that specifies how the geometry should a appear. This section summarises the different appearance characteristics that can be specified using the appearance node component.

2.7.1 The `Appearance` Node Component

A `Shape3D` object maintains a reference to an `Appearance` node component object that defines how the geometry of the `Shape3D` object should appear in the rendered scene. The `Appearance` node component doesn't contain any appearance information, instead, it maintains references to appearance components that hold various different types of appearance information. There are a total of eleven appearance components:

- **Coloring attributes -** defines the attributes used in colour selection and shading. These attributes are defined using a `ColoringAttributes` object.
- **Line attributes -** defines attributes used to render lines, including the pattern, width, and whether or not antialiasing is to be used. These attributes are defined using a `LineAttributes` object.

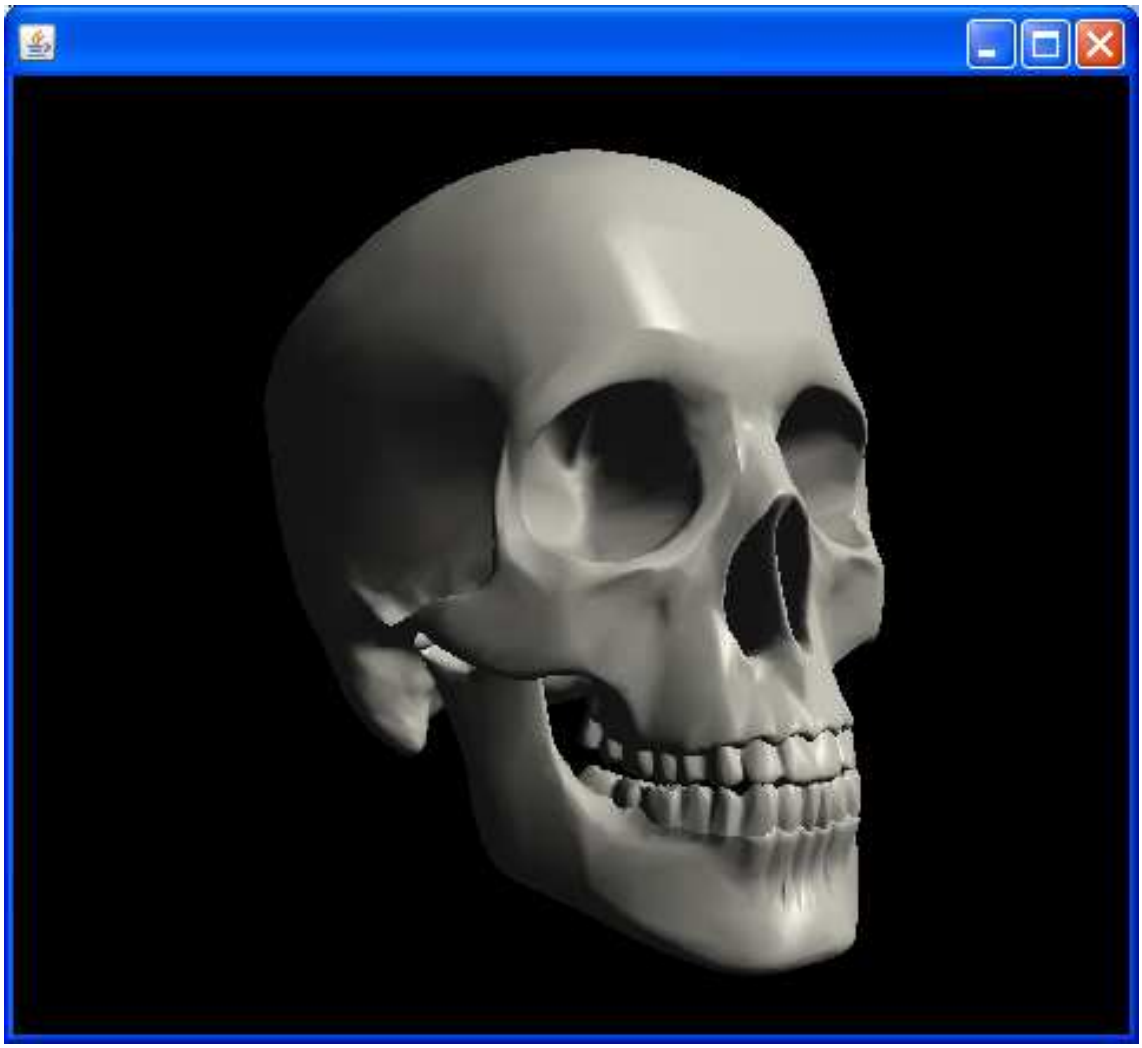


Figure 2.23: An example of the type of geometry that can be loaded into a Java 3D program using a custom `Loader`.

- **Point attributes -** defines attributes used to render points, including the point size and whether antialiasing is to be used. These attributes are defined using a `PointAttributes` object.
- **Polygon attributes -** defines the attributes used to render polygons, including culling, rasterization mode (filled, lines, or points), constant offset, offset factor and whether back facing normals are flipped. These attributes are defined using a `PolygonAttributes` object.
- **Rendering attributes -** defines rendering operations, including the alpha test function and test value, the raster operation, whether vertex colours are ignored, whether invisible objects are rendered, and whether the depth buffer is enabled. These attributes are defined using a `RenderingAttributes` object.
- **Transparency attributes -** defines the attributes that affect transparency mode (blended, screen-door), blending function (used in transparency and antialiasing operations), and a blend value that defines the amount of transparency to be applied. These attributes are defined using a `TransparencyAttributes` object.

- **Material -** defines the appearance of an object under illumination, such as the ambient colour, diffuse colour, specular colour, emissive colour, and shininess. These attributes are defined using a `Material` object.

- **Texture -** defines the texture image and filtering parameters used when texture mapping is enabled. These attributes are defined using a `Texture` object.

- **Texture attributes -** defines the attributes that apply to texture mapping, such as the texture mode, texture transform, blend colour, and perspective correction mode. These attributes are defined using a `TextureAttributes` object.

- **Texture coordinate generation -** defines the attributes that apply to texture coordinate generation, such as whether coordinate generation is enabled, coordinate format (2D or 3D coordinates), coordinate generation mode (object linear, eye linear, or spherical reflection mapping), and the R, S and T coordinate plane equations. These attributes are defined using a `TexCoordGeneration` object.

- **Texture unit state -** an array that defines the texture state for each of N separate texture units. This allows multiple textures to be applied to a geometry. Each `TextureUnitState` object contains a `Texture` object, a `TextureAttributes` object and a `TexCoordGeneration` object.

The `Appearance` class defines `set()` and `get()` methods for each of the eleven appearance components that it can reference. For example, the `set()` and `get()` methods for the `ColoringAttributes` appearance component have the following format:

- `void setColoringAttributes(ColoringAttributes coloringAttributes)`
  Sets the current `ColoringAttributes` reference to the specified object.

- `ColoringAttributes getColoringAttributes()`
  Returns the reference to the current `ColoringAttributes` appearance component.

The `Appearance` class also defines capability bits for each of the eleven appearance components that it can reference. In the case of the `ColoringAttributes` appearance component, the following capability bits are defined:

- `ALLOW_COLORING_ATTRIBUTES_READ`
  Indicates that the `ColoringAttributes` node component associated with this `Appearance` object can be read after the scene graph has gone live.

- `ALLOW_COLOURING_ATTRIBUTES_WRITE`
  Indicates that the `ColoringAttributes` node component associated with this `Appearance` object can be written to after the scene graph has gone live.

The `set()` and `get()` methods, and the capability bits, for the other ten appearance components have the same format as those for the `ColoringAttributes` appearance component outlined above.

The following subsections describe how these appearance components can be used to influence the appearance of objects in a rendered scene.

2.7.2 ColoringAttributes

The `ColoringAttributes` appearance component is used to set the colour and shading model for a shape if:

1. The appearance does not have a material appearance component associated with it.
2. The appearance does a have a material appearance component associated with it but the Material disables lighting.

The colour is set by specifying the relevant red, green and blue colour components, either individually as `float` primitives, or collectively using a suitably constructed `Color3f` object.

Note: If vertex colours are defined then they override the colour value associated with the `ColoringAttributes` appearance component.

If vertex colours are specified in the geometry then the shading model is used to indicate how these colours should be rendered. The shading model can be one of the following:

- `SHADE_FLAT` - indicates the flat shading model. This shading model does not interpolate between vertices. Each polygon is drawn with a single colour which is the colour at one of the vertices of the polygon[1].
- `SHADE_GOURAUD` - uses the Gouraud (smooth) shading model. This shading model interpolates the colour at each vertex across the primitive. The primitive is drawn with many different colours and the colour at each vertex is treated individually. For lines, the colours along the line segment are interpolated between the vertex colours at each end.
- `FASTEST` - uses the fastest method available for shading. This shading mode maps to whatever shading model the Java 3D implementor defines as the "fastest" shading model, which may be hardware-dependent.
- `NICEST` - uses the nicest (highest quality) available method for shading. This shading mode maps to whatever shading model the Java 3D implementor defines as the "nicest" shading model, which may be hardware dependent.

Note: In most Java 3D implementations, `SHADE_FLAT` shading is no faster than `SHADE_GOURAUD` shading. Consequently, `FASTEST` shading and `NICEST` shading both correspond to `SHADE_GOURAUD` shading.

Note: The default value for the colour is white (1.0, 1.0, 1.0) and the default value for the shading model is `SHADE_GOURAUD`.

A `ColoringAttributes` object can be created using one of the following constructors:

- `ColoringAttributes()`

[1] **Note:** In the example that follows, the colour of the polygon is defined by the last vertex in the case of the flat shading model.

- `ColoringAttributes(Color3f colour, int shadeModel)`
- `ColoringAttributes(float r, float g, float b, int shadeModel)`

The resulting `ColoringAttributes` object can be associated with an `Appearance` by calling the `setColoringAttributes()` method of the `Appearance` object. The methods defined by the `ColoringAttributes` object include:

- `void getColor(Color3f colour)`
  Returns the intrinsic colour of this `ColoringAttributes` class and stores it in the specified `Color3f` object.
- `void setColor(Color3f colour)`
  Sets the intrinsic colour of this `ColoringAttributes` class to specify colour.
- `int getShadeModel()`
  Returns the shade model for this `ColoringAttributes` component.
- `void setShadeModel(int shadeModel)`
  Sets the shade model for this `ColoringAttributes` component to the specified value.

The `ColoringAttributes` class also defines a series of capability bits that can be used to determine which capabilities are supported after the scene graph has gone live. These are:

- `ALLOW_COLOR_READ`
  Indicates that the colour information can be read after the scene graph has gone live.
- `ALLOW_COLOR_WRITE`
  Indicates that the colour information can be written to after the scene graph has gone live.
- `ALLOW_SHADE_MODEL_READ`
  Indicates that the shade model can be read after the scene graph has gone live.
- `ALLOW_SHADE_MODEL_WRITE`
  Indicates that the shade model can be written to after the scene graph has gone live.

The following example provides a comprehensive demonstration of how the `ColoringAttributes` appearance component can be used.

```
0
import javax.media.j3d.*;

public class ColoringAttributesExample extends BasicSceneWithMouseControl
{
5   public static void main(String args[]){new ColoringAttributesExample();}

    public BranchGroup createContentBranch()
    {
```

```
    BranchGroup root = new BranchGroup();

    // Define vertex colours
    float [] colours = {1.0f, 0.0f, 0.0f,
                        0.0f, 1.0f, 0.0f,
                        0.0f, 0.0f, 1.0f};

    // Defines vertex coordinates
    float [] coordinates = {-0.8f, -0.4f, 0.0f,
                             0.8f, -0.4f, 0.0f,
                             0.0f,  0.4f, 0.0f};

    GeometryArray geometryArray = new TriangleArray(3,
        GeometryArray.COORDINATES|
        GeometryArray.COLOR_3);

    geometryArray.setCoordinates(0, coordinates);
    geometryArray.setColors(0, colours);

    // Define the colouring attributes appearance component
    Appearance appearance = new Appearance();
    ColoringAttributes ca = new ColoringAttributes(1.0f, 1.0f, 0.0f,
        ColoringAttributes.SHADE_GOURAUD);
    appearance.setColoringAttributes(ca);

    Shape3D shape = new Shape3D(geometryArray, appearance);
    root.addChild(shape);

    root.compile();

    return root;
  }
}
```

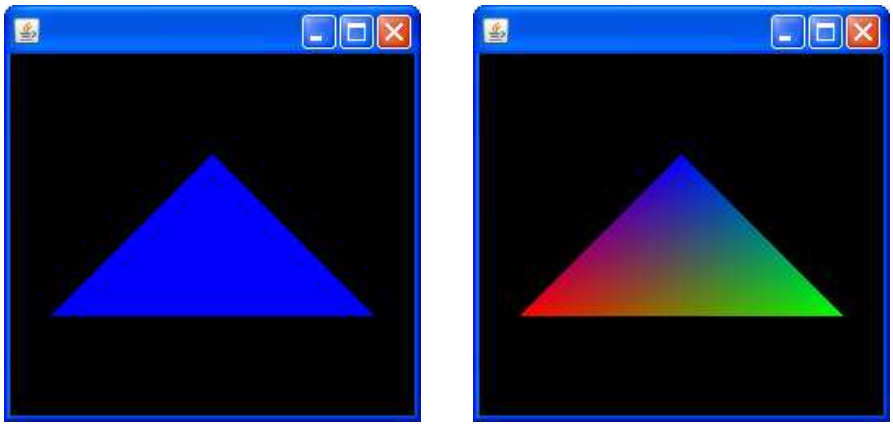
The program defines a single triangular polygon. Colours are also defined for each of the three vertices. The left vertex is assigned red, the right vertex is assigned green and the top vertex is assigned blue. A `ColoringAttributes` object is constructed with an intrinsic colour of yellow and a flat shading model. The resulting `ColoringAttributes` appearance component is associated with an `Appearance` object, which is in turn associated with the `Shape3D` object that represents the triangular geometry. Sample renderings generated by this program, and slight variations of this program, are illustrated in Figure 2.24.

Reference:

- H. Gouraud, "Continuous shading of curved surfaces", *IEEE Transactions on Computers* 20(6) :623-628, 1971.

2.7.3 `PointAttributes`

The `PointAttributes` appearance component defines all attributes that apply to point primitives. These are:



(a) (b)

Figure 2.24: Examples of a simple polygon with vertex colours rendered using (a) flat and (b) Gouraud shading. In the case of flat shading, the colour of the polygon is determined by the colour of the last vertex to be defined.

- **Size -** The size of the point, in pixels. The default value for this attribute is one pixel.
- **Antialiasing -** If the point size is greater than one-pixel in size then antialiasing smooths the outline of the point when it is rendered. Antialiasing is disabled by default.

An instance of a `PointAttributes` class can be defined as follows:

- `PointAttributes(float pointSize, boolean pointAntialiasing)`
  Contructs a `PointAttributes` object with the specified point size and antialiasing attributes.
- `PointAttributes()`
  Constructs a `PointAttributes` object with a point size of one pixel and antialiasing turned off. These are the default values for these attributes.

The `PointAttributes` class also defines methods to update and retrieve the attributes that it contains. In addition, the `PointAttributes` class defines a series of capability bits that can be used to specify whether the attributes can be updated or retrieved after the scene graph has gone live. This following program demonstrates how the `PointAttributes` appearance component can be used.

```
0
  import javax.media.j3d.*;

  public class PointAttributesExample extends BasicSceneWithMouseControl
  {
5   public static void main(String args[]){new PointAttributesExample();}

    public BranchGroup createContentBranch()
    {
      BranchGroup root = new BranchGroup();
10
```

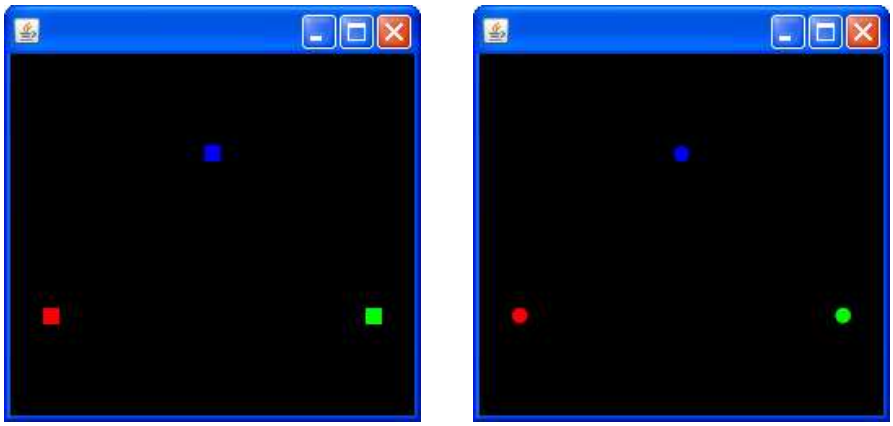
```
    // Define the vertex colours
    float [] colours = {1.0f, 0.0f, 0.0f,
                        0.0f, 1.0f, 0.0f,
                        0.0f, 0.0f, 1.0f};

    // Define the vertex coordinates
    float [] coordinates = {-0.8f, -0.4f,  0.0f,
                             0.8f, -0.4f,  0.0f,
                             0.0f,  0.4f,  0.0f};

    GeometryArray geometryArray = new PointArray(3,
        GeometryArray.COORDINATES|
        GeometryArray.COLOR_3);

    geometryArray.setCoordinates(0, coordinates);
    geometryArray.setColors(0, colours);

    // Define the point attributes appearance component
    Appearance appearance = new Appearance();
    PointAttributes pa = new PointAttributes(10.0f, true);
    appearance.setPointAttributes(pa);

    Shape3D shape = new Shape3D(geometryArray, appearance);
    root.addChild(shape);

    root.compile();

    return root;
  }
}
```

The program begins by creating a `PointArray` geometry where the vertex and colour information is the same as in the `ColoringAttributes` example. A `PointAttributes` object is constructed that represents antialiased points that are 10 pixels in size. Renderings generated using different versions of this program are illustrated in Figure 2.25

2.7.4 `LineAttributes`

The `LineAttributes` appearance component is used to specify all of the attributes associated with rendering lines. These include the line pattern, the line width in pixels and whether or not antialiasing is to be used. The possible values for the line pattern are:

- `PATTERN_SOLID` - draws a solid line with no pattern. This is the default.
- `PATTERN_DASH` - draws dashed lines. Ideally these will be drawn with a repeating pattern of 8 pixels on and 8 pixels off.
- `PATTERN_DOT` - draws dotted lines. Ideally these will be drawn with a repeating pattern of 1 pixel on and 7 pixels off.



(a) (b)

Figure 2.25: Examples of points 10 pixels in size the were rendered with (a) and without (b) antialiasing.

- `PATTERN_DASH_DOT` - draws dashed-dotted lines. Ideally, these will be drawn with a repeating pattern of 7 pixels on, 4 pixels off, 1 pixel on, and 4 pixels off.
- `PATTERN_USER_DEFINED` - draws used defined patterns. These will be discussed below.

Note: The default values are a solid line pattern with a width of 1 pixel that is not antialiased.

A `LineAttributes` object can be created using the following constructor:

- `LineAttributes(float width, int pattern, boolean antialiasing)`
  Constructs a `LineAttributes` object that represents lines with the specified width, pattern and antialiasing.

The `LineAttributes` class also defines the usual accessor methods to update and retrieve its attributes. It should be noted that if the line pattern is set to `PATTERN_USER_DEFINED`, then a pattern mask must be specified using:

- `void setPatternMask(int mask)`
  Defines a sixteen bit mask where a 1 indicates that a pixel is drawn and a 0 indicates that a pixel is not drawn.

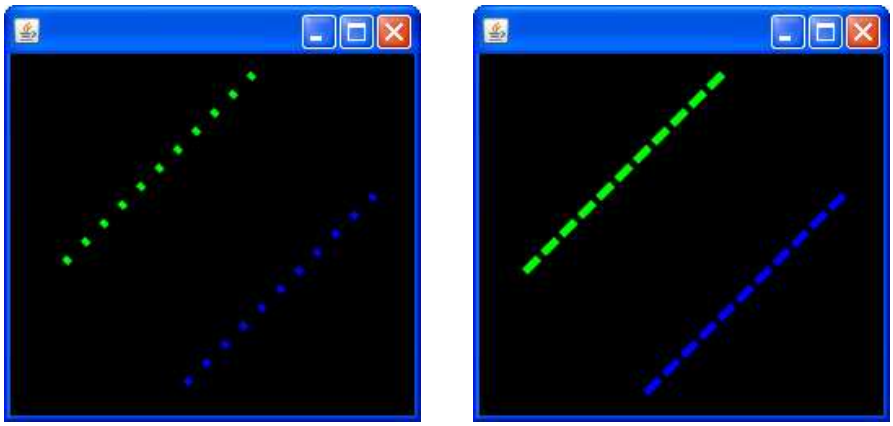
Capability bits are also defined to specified whether or not the attributes can be updated or retrieved after the scene graph has gone live.

The following example demonstrates how a `LineAttributes` object can be used to create a custom line style.

```
0
import javax.media.j3d.*;

public class LineAttributesExample extends BasicSceneWithMouseControl
```

```
{
  public static void main(String args[]){new LineAttributesExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // Define the vertex colours
    float [] colours = {0.0f, 1.0f, 0.0f,
                        0.0f, 1.0f, 0.0f,
                        0.0f, 0.0f, 1.0f,
                        0.0f, 0.0f, 1.0f};

    // Defines the vertex coordinates
    float [] coordinates = {-0.8f, -0.2f,  0.0f,
                             0.2f, 0.8f,  0.0f,
                            -0.2f, -0.8f,  0.0f,
                             0.8f, 0.2f,  0.0f};

    GeometryArray geometryArray = new LineArray(4,
        GeometryArray.COORDINATES|
        GeometryArray.COLOR_3);

    geometryArray.setCoordinates(0, coordinates);
    geometryArray.setColors(0, colours);

    // Define the line  attributes  appearance component
    Appearance appearance = new Appearance();
    LineAttributes la = new LineAttributes();
    la.setLinePattern(LineAttributes.PATTERN_USER_DEFINED);
    la.setPatternMask(0xFF00);
    la.setLineAntialiasingEnable(true);
    la.setLineWidth(5);
    appearance.setLineAttributes(la);

    Shape3D shape = new Shape3D(geometryArray, appearance);
    root.addChild(shape);

    root.compile();

    return root;
  }
}
```

This program defines two lines using a `LineArray` object the first line has blue vertex colours and the second line has green vertex colours. A `LineAttributes` object is created that represents a user defined line pattern with a pattern mask of `0xFFF0`. This mask represents a line with 12-bit segments followed by a 4-bit gap. The line width is set to 5 pixels and the line is rendered using antialiasing. Renderings generated by different versions of this program are illustrated in Figure 2.26.



(a) (b)

Figure 2.26: Examples of lines rendered using different patterns. (a) A line with 4 pixel segments followed by a 12 pixel gap. (b) A line with 12 pixel segments followed by a 4 pixel gap. In both cases the lines are 5 pixels wide and rendered with antialiasing.

2.7.5 `PolygonAttributes`

The `PolygonAttributes` appearance component defines the attributes that affect the rendering of polygons. These attributes include the following:

- Rasterization mode - defines how the polygons are drawn and can have one of the following three values:
  - `POLYGON_POINT` - The polygon is rendered as a set of points that are drawn at the vertices.
  - `POLYGON_LINE` - The polygon is rendered as a set of lines that are drawn between consecutive vertices.
  - `POLYGON_FILL` - The polygon is rendered by filling the interior region between the vertices. This is the default rasterization mode.
- Face culling - defines which polygons are culled, or discarded, before they are converted to screen coordinates. There are three possible mode of face culling:
  - `CULL_BACK` - Culls all back facing polygons. This is the default mode for face culling.
  - `CULL_FRONT` - Culls all front facing polygons.
  - `CULL_NONE` - Disables face culling and causes front and back facing polygons to be rendered.
- Back-face normal flip - specified whether vertex normals of back-facing polygons are flipped (negated) prior to lighting. The setting can be either true, meaning back-facing normals are flipped, or false. The default value is false.
- Offset - the depth values of all pixels generated by polygon rasterization can be offset by a value that is computed for that polygon. Two values are used to specify the offset.

- Offset bias - the constant polygon offset that is added to the final device coordinate Z value for the polygon primitive.
- Offset factor - the factor to be multiplied by the slope of the polygon and then added to the final device coordinate Z value of the polygon.

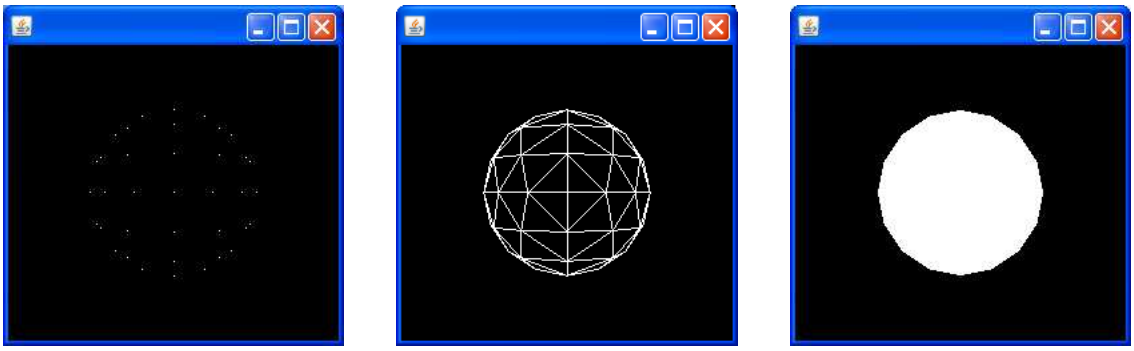
These values can be either positive or negative and the default for both of these values is 0.0.

Note: Polygon offset is used to solve a specific problem: drawing lines on top of polygons. The typical usage of this is in drawing a "hidden line" display. This is a form of wire-frame display in which a solid object is drawn as lines so that the lines hidden by the foreground are removed.

The following example demonstrates two of the attributes that are defined by the `PolygonAttributes` class.

```
0  import javax.media.j3d.*;
   import com.sun.j3d.utils.geometry.*;

   public class PolygonAttributesExample extends BasicSceneWithMouseControl
   {
5    public static void main(String args[]){new PolygonAttributesExample();}

     public BranchGroup createContentBranch()
     {
       BranchGroup root = new BranchGroup();
10
       Appearance appearance = new Appearance();

       // Define the polygon attributes appearance component
       PolygonAttributes pa = new PolygonAttributes();
15     pa.setPolygonMode(PolygonAttributes.POLYGON_POINT);
       pa.setCullFace(PolygonAttributes.CULL_BACK);
       appearance.setPolygonAttributes(pa);

       // Create a sphere primitive with the specified appearance
20     Sphere sphere = new Sphere(0.5f, appearance);
       root.addChild(sphere);

       root.compile();

25     return root;
     }
   }
```

This program begins by defining a `PolygonAttributes` object using the default constructor. Accessor methods are then used to set the polygon mode to `POLYGON_FILL` and the cull face attribute to `CULL_BACK`. The `PolygonAttributes` object is then associated with an `Appearance` object which is ultimately used to construct a `Sphere` primitive with a radius of 50 cm. Renderings generated by different versions of this program are illustrated in Figure 2.27.



(a) (b) (c)

Figure 2.27: A `Sphere` primitive with 15 subdivisions rendered as (a) a set of points, (b) a wire frame and (c) filled polygons. Back facing polygons in all three renderings are culled.

2.7.6 `RenderingAttributes`

The `RenderingAttributes` class defines common rendering attributes for all primitive types. The attributes defined by this class include:

- **Alpha testing -** This is an advanced mechanism that is used to control rendering on a per-pixel basis. It uses the alpha value specified in RGBA colours, a test alpha value, and a comparison function to control whether a pixel gets drawn.
- **Raster operation -** This controls the per-pixel rendering operation. It specifies how the source and destination pixels are logically combined to produce the result written to the destination raster. The source pixel is the pixel to be rendered and the destination pixel is the pixel value that is currently stored in the frame buffer. Raster operations include AND, OR, XOR etc.
- **Ignore vertex colours -** This allows an appearance to override the vertex colours specified with the geometry for a shape. This can be useful when the program wants to highlight the geometry by changing its colour . If lighting is disabled then the object's colour will come from the `ColoringAttributes` appearance component, otherwise the object's colour comes from the `Material` appearance component.
- **Visibility flag -** This can be used to turn off the rendering of a shape without disabling the shape's pickability or collidability.
- **Depth buffer control -** This allows depth buffering, also known as Z-buffering, to be turned on and off. The depth buffer is used to determine which objects in a scene are rendered and which are occluded.

2.7.7 `TransparencyAttributes`

The `TransparencyAttributes` appearance component is used to specify the transparency characteristics for the associated `Appearance` object. There are a total of four transparency attributes.

- The transparency mode - this specifies the method used to generate a transparent rendering. There are four possible values:
  - `SCREEN_DOOR` - uses screen door transparency. This is done using an on/off stipple pattern in which the percentage of transparent pixels is approximately equal to the value specified by the transparency parameter.
  - `BLENDED` - uses alpha blended transparency. The blend equation is specified by source blend function and destination blend function attributes.
  - `FASTEST` - uses the fastest of the two blend functions.
  - `NICEST` - uses the nicest of the two blend functions.

  Note: In the Microsoft Windows XP implementation of Java 3D version `FASTEST` and `NICEST` modes both result in `BLENDED` being used.
- Transparency value, the amount of transparency to be applied to this appearance component object. The transparency values are in the range [0.0, 1.0], with 0.0 being fully opaque and 1.0 being fully transparent.
- Blend function - used in blended transparency and antialiasing operations. The source function specifies the factor that is multiplied by the source colour. This value is added to the product of the destination blend function and destination colour.

The default blend equation has the following format:

$$alpha_{src} \times src + (1 - alpha_{src}) \times dst \tag{2.1}$$

Note: The meaning of the terms *src* and *dst* is not explicitly stated in the API documentation. However, it is clear from this equation that *src* relates to the object that is being made transparent and *dst* relates to the background.

The possible values for both source and destination blend function are as follows:

- `BLEND_ZERO` - the blend function is: $f = 0$
- `BLEND_ONE` - the blend function is: $f = 1$
- `BLEND_SRC_ALPHA` - the blend function is: $f = alpha_{src}$
- `BLEND_ONE_MINUS_SRC_ALPHA` - the blend function is: $f = 1 - alpha_{src}$

The blend functions that can only be used in conjunction with the source pixel are:

- `BLEND_DST_COLOR` - the blend function is: $f = colour_{dst}$
- `BLEND_ONE_MINUS_DST_COLOR` - the blend function is: $f = 1 - colour_{dst}$

The possible values for the destination blend function are:

- `BLEND_SRC_COLOR` - the blend function is: $f = colour_{src}$
- `BLEND_ONE_MINUS_SRC_COLOR` - the blend function is: $f = 1 - color_{src}$

Note: Where the blend function is a colour, the individual colour components or their complements must be multiplied by either the *src* or *dst* values to give the required output.

The following example demonstrates how the `TransparencyAttributes` appearance component can be used to alter the transparency of an object.

```
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.geometry.*;

public class TransparencyAttributesExample extends BasicSceneWithMouseControl
{

  public static void main(String args[]){new TransparencyAttributesExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Appearance appearance1 = new Appearance();

    // Create the first colouring attibutes appearance component
    ColoringAttributes ca1 = new ColoringAttributes(new Color3f(0.0f, 0.0f, 1.0f),
        ColoringAttributes.SHADE_FLAT);
    appearance1.setColoringAttributes(ca1);

    // Create the transparency appearance compoennt that represents
    // a 50% blended transparency
    TransparencyAttributes ta = new TransparencyAttributes();
    ta.setTransparencyMode(TransparencyAttributes.BLENDED);
    ta.setTransparency(0.25f);
    appearance1.setTransparencyAttributes(ta);

    // Add the first sphere to the root of the scene graph
    Sphere sphere = new Sphere(0.5f, appearance1);
    root.addChild(sphere);

    Appearance appearance2 = new Appearance();

    // Create the second colouring attributes appearance component
    ColoringAttributes ca2 = new ColoringAttributes(new Color3f(1.0f, 0.0f, 0.0f),
        ColoringAttributes.SHADE_FLAT);
    appearance2.setColoringAttributes(ca2);

    Transform3D transform = new Transform3D();
    transform.setTranslation(new Vector3f(-0.5f, 0.0f, -1.0f));
    TransformGroup tg = new TransformGroup(transform);
    root.addChild(tg);

    // Add the second sphere to the transform group
    Sphere sphere2 = new Sphere(0.5f, appearance2);
    tg.addChild(sphere2);
```

```
        root.compile();

        return root;
    }
}
```

The program creates two `Sphere` primitives. The first sphere is positioned at the origin and its unlit colour is set to blue using a `ColoringAttributes` appearance component. A `TransparencyAttributes` appearance component is used to set the transparency mode for this sphere to `BLENDED` with a transparency value to 50%. The second sphere is positioned 1 metre behind and 50 cm to the left of the origin using a `TransformGroup`. The unlit colour of the second sphere is set to red using a `ColoringAttributes` appearance component. Renderings generated by different versions of this program are illustrated in Figure 2.28.

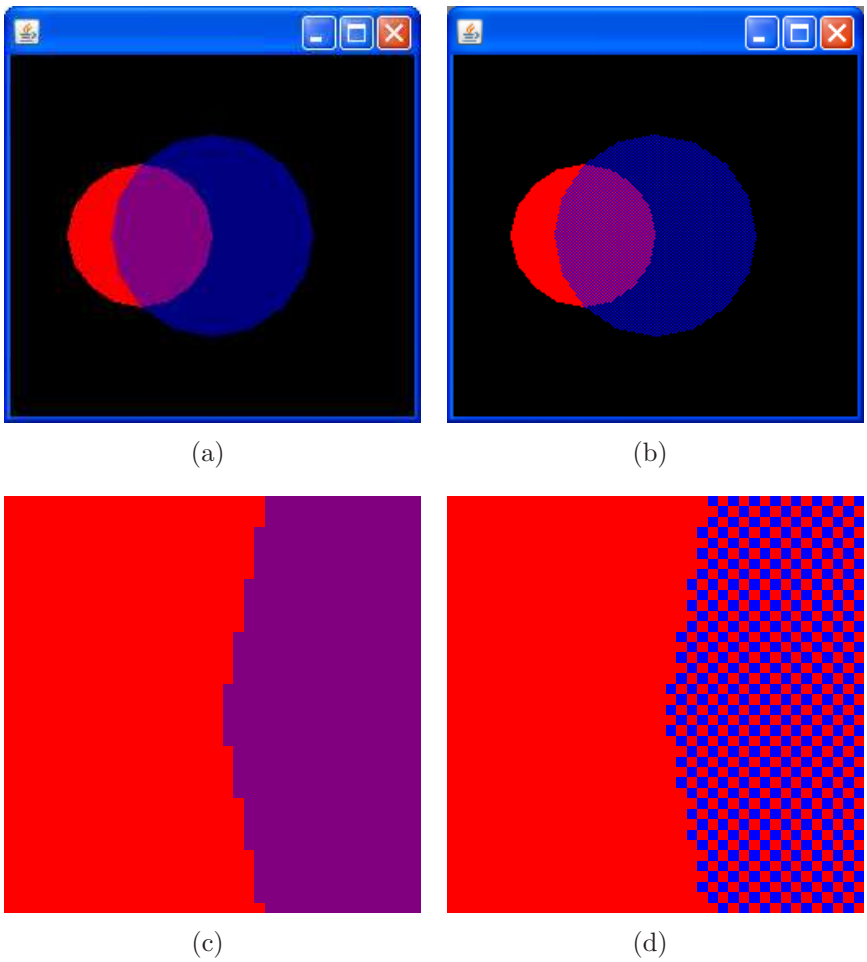


Figure 2.28: An scene containing a opaque red sphere located behind a semitransparent blue sphere. The `BLENDED` transparency mode is used in (a) and (c) and the `SCREEN_DOOR` mode is used in (b) and (d).

2.7.8 Material

The material appearance component specifies the colour of a shape when it is exposed to different types of lighting. A shape will only respond to light if:

1. The geometry of the shape has normals.
2. The appearance has a material.
3. The material enables lighting.

If any of these requirements are not met then the shape will not respond to lighting and the colour of the shape will be determined by either:

1. The geometry of the shape, i.e. the colours assigned to vertices of the geometry.
2. The relevant `ColoringAttributes` appearance component.

If both of these properties are defined then the vertex colours take precedence. If neither of these properties are specified then the colour comes from the default `ColorAttributes`. This causes the shape to be rendered in white.

The `Material` appearance component defines the appearance of a shape under illumination. These can be set in the constructor:

- `Material(Color3f ambientColor, Color3f, emissiveColor, Color3f diffuseColor, Color3f specularColor, float shininess)` Creates a new `Material` object with the specified colour characteristics.

2.7.8.1 Ambient Colour

Ambient light is the diffused light that fills a region, lighting regions that would not otherwise be illuminated. The ambient colour of `Material` appearance component indicates the response of the material to ambient light, i.e. the percentage of ambient light that is reflected.

If the ambient colour of a material is orange (1.0, 0.5, 0.0) and it is illuminated by purple ambient light (1.0, 0.0, 1.0) then the resulting reflected colour will be red (1.0, 0.0, 0.0), i.e. the colour reflected by the material is obtained by multiplying each of the colour components of the ambient light by the corresponding colour components of the ambient colour of the material.

In addition to using the constructor, the ambient colour of a `Material` appearance component object can also be set using either of the following methods:

- `void setAmbientColor(Color3f colour)`
- `void setAmbientColor(float r, float g, float b)`

It should be noted that once the scene graph is live these methods will throw a `CapabilityNotSetException` if the `ALLOW_COMPONENT_WRITE` capability bit has not been set using the `setCapability()` method.

The ambient colour of a `Material` component can be retrieved using:

- `void getAmbientColor(Color3f colour)`

The ambient colour information is stored in the supplied `Color3f` object. It should be noted that this method will throw a `CapabilityNotSetException` if the `ALLOW_COMPONENT_READ` capability bit has not been set using the `setCapability()` method.

Note: The default ambient colour is (0.2, 0.2, 0.2), i.e. by default the 20% of ambient light is reflected by the material.

2.7.8.2 Emissive Colour

The emissive colour of a material is the colour of the object independent of any light sources. It produces glowing colours that seem to emanate from within the shape itself. An emissive colour can be used to represent a light that is turned on.

In addition to using the constructor, the emissive colour of a `Material` appearance component can be set using either of the following methods:

- `void setEmissiveColor(Color3f colour)`
- `void setEmissiveColor(float r, float g, float b)`

The emissive colour of a `Material` appearance component can subsequently be retrieved using:

- `void getEmissiveColor(Color3f colour)`

The `ALLOW_COMPONENT_READ` and `ALLOW_COMPONENT_WRITE` capabilities must be set if the emissive colour is to be retrieved or updated after the scene graph has gone live.

Note: The default emissive colour for a `Material` appearance component is (0.0, 0.0, 0.0), i.e. the material does not appear to emit any colour.

Note: Setting the emissive colour for a material only causes the associated shape to appear to emit light. The shape does not illuminate the objects around it. In order to achieve this a suitable light source would have to be associated with the shape in addition to setting the emissive colour for the material.

2.7.8.3 Diffuse Colour

The diffuse colour defines the "true" colour of an object. It is the colour of the object when lit, excluding any light being reflected due to the shininess of the object. The diffuse colour is the colour produced by *diffuse reflection*, which is a term used to describe the light that bounces off objects in random directions. The intensity of diffuse lighting depends on the angle the light rays make with the surface of the object. If the light hits the object directly, it creates light of maximum intensity; the light intensity decreases as the angle increases. If the surface faces away from the light, then the light does not add any illumination to the surface.

The diffuse colour of a `Material` can be updated or retrieved using the similar methods as those described for ambient and emissive colours. The required capabilities

must also be set of the diffuse colour needs to be updated or retrieved after the scene graph has gone live.

Note: The default diffuse colour is (1.0, 1.0, 1.0), i.e. a material has a white diffuse colour by default.

2.7.8.4 Specular Colour

The specular colour comes from *specular reflection*, which is an approximation of the way a light bounces off a shiny object. Specular colour is combined with shininess settings to create shiny highlights on the surface of shapes. The colour of the highlight is a combination of the specular colour and the light colour. The maximum intensity is along the reflection of the light off the surface toward the viewer. The intensity decreases as the reflection is directed away from the viewer.

Note: The default specular colour is white (1.0, 1.0, 1.0).

2.7.8.5 Shininess

Shininess controls the size of reflective highlights created using specular colours. Shininess is a floating-point number in the range [1.0, 128.0], where 1.0 represents a surface that is not shiny at all, and 128.0 represents an extremely shiny surface. Shininess values outside this range are clamped.

Note: The default shininess value is 64.

2.7.8.6 Vertex Colours

If vertex colours are defined for the relevant geometry then they are used in place of the specified material colour or colours. By default the vertex colours replace the diffuse colour of the material. The relevant colour target for a `Material` object can be set using:

- `void setColorTarget(int colourTarget)`
  Indicates that the vertex colours are used in place of the specified material colour.

The possible values for the `colourTarget` argument are:

- `AMBIENT`
  Indicates that per-vertex colours replace the ambient material colour.
- `EMISSIVE`
  Indicatest that per-vertex colours replace the emissive material colour.
- `DIFFUSE`
  Indicates that per-vertex colours replace the diffuse material colour.
- `SPECULAR`
  Indicates that per-vertex colours replace the specular colour.
- `AMBIENT_AND_DIFFUSE`
  Indicates that per-vertex colours replace both the ambient and diffuse material colours.

2.7.9 Lighting

Light nodes are used to illuminate shapes in a virtual world. Java 3D supports four basic types of lights:

- `DirectionalLight` - provides a simple, fast light type that simulates light from a distant source, such as the sun.
- `PointLight` - a light source that radiates in all directions with a position defined by a point.
- `SpotLight` - a point light source that shines in a specific direction.
- `AmbientLight` - simulates the diffused light that fills a region, lighting areas that are not directly illuminated.

Each of these light nodes is a subclass of the `Light` class. The methods defined by the `Light` class include:

- `void setEnable(boolean state)`
  Turns the light on or off.
- `boolean getEnable()`
  Retrieves the current state of the light.
- `void setColor(Color3f color)`
  Sets the colour of the light to the specified value.
- `void getColor(Color3f color)`
  Retrieves the colour of the light and stores it in the supplied `Color3f` object.

In order to use these methods after the scene graph has go live the following capability bits will need to be set:

- `ALLOW_STATE_READ`
  Indicates that this light allows read access to its state information (i.e. whether the light is on of off) after the scene graph has gone live.
- `ALLOW_STATE_WRITE`
  Indicates that this light allows write access to its state information after the scene graph has gone live.
- `ALLOW_COLOR_READ`
  Indicates that this light allows read access to its colour information after the scene graph has gone live.
- `ALLOW_COLOR_WRITE`
  Indicates that this light allows write access to its colour information after the scene graph has gone live.

2.7.9.1 `DirectionalLight`

A `DirectionalLight` node defines an oriented light with an origin at infinity. It has the same attributes as a `Light` node, with the addition of a directional vector to specify the direction that the light shines in. A directional light has parallel light rays that travel in one direction along the specified vector. Directional light contributes to diffuse and specular reflections, which in turn depend on the orientation of an object's surface but not its position. A directional light does not contribute to ambient reflections. An illustration of how directional lighting works is presented in Figure 2.29.

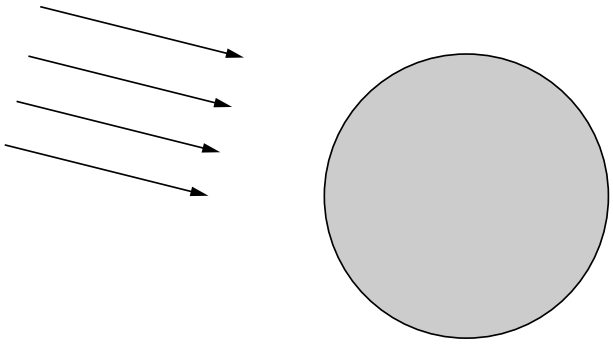


Figure 2.29: A `DirectionalLight` represents light from a distant source a can be thought of as a set of parallel rays originating from a specific direction.

The most comprehensive constructor for a directional light has the following format:

- `DirectionalLight(boolean on, Color3f colour, Vector3f direction)`
  Creates a directional light source with the specified state, colour and direction.

The direction of a `DirectionalLight` node can also be set or retrieved using:

- `void setDirection(Vector3f direction)`
  Sets the direction of this `DirectionalLight` to the specified value.
- `void getDirection(Vector3f direction)`
  Retrieves the direction of this `DirectionalLight` and stores it in the supplied `Vector3f` object.

If these methods are to be used after the scene graph has gone live then the following capability bits must be set:

- `ALLOW_DIRECTION_READ`
  Indicates that this light node allows read access to its direction information after the scene graph has gone live.
- `ALLOW_DIRECTION_WRITE`
  Indicates that this light node allows write access to its direction information after the scene graph has gone live.

A bounding region must also be specified in order to indicate the region where the light is active. The bounding region in set using the following method:

- `void setInfluencingBounds(Bounds region)`
  Sets the light's influencing bounds to the specified region.

The following example demonstrates how a `DirectionalLight` can be used to illuminate an object.

```
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.geometry.*;

public class DirectionalLightExample extends BasicScene
{
  public static void main(String args[]){new DirectionalLightExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Appearance appearance = new Appearance();

    // Create a material with a red diffuse colour
    Material material = new Material();
    material.setDiffuseColor(new Color3f(1.0f, 0.0f, 0.0f));
    appearance.setMaterial(material);

    Sphere sphere = new Sphere(0.5f,
        Sphere.GENERATE_NORMALS,
        100,
        appearance);

    root.addChild(sphere);

    // Create a directional light with a bright white colour
    DirectionalLight light = new DirectionalLight(new Color3f(1.0f, 1.0f, 1.0f),
        new Vector3f(-1.0f, -1.0f, -1.0f));
    light.setInfluencingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    root.addChild(light);

    root.compile();

    return root;
  }
}
```

This program creates a `Sphere` primitive located at the origin. The sphere has a radius of 50 cm and a red diffuse colour. The colour of the sphere is specified using a `Material` object which is ultimately associated with the `Appearance` object that is used to construct the sphere. A directional light is also constructed and attached to the scene. The colour of this light is bright white and it shines in the direction (-1.0, -1.0, -1.0). The output generated when this program is executed is illustrated in Figure 2.30.

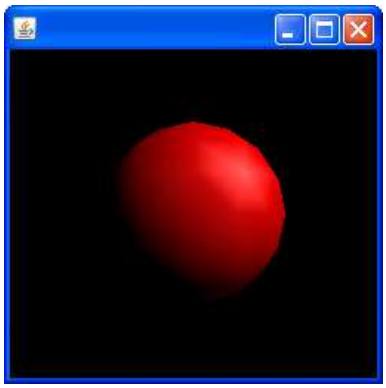


Figure 2.30: An example of a red sphere illuminated by white directional light shining in the direction (-1.0f, -1.0f, -1.0f).

2.7.9.2 `PointLight`

The `PointLight` class represents an attenuated light source that is located at a fixed point in space and radiates light equally in all directions away from the light source. A `PointLight` class has the same attributes as the `Light` class, with the addition of location and attenuation parameters.

A `PointLight` contributes to diffuse and specular reflections , which in turn depend on the orientation and position of a surface. A `PointLight` does not contribute to ambient reflections.

A `PointLight` is attenuated by multiplying the contribution of the light by an attenuation factor. The attenuation factor causes the brightness of the `PointLight` to decrease as the distance from the light source increases. The attenuation factor for a `PointLight` contains three values:

- Constant attenuation
- Linear attenuation
- Quadratic attenuation

A `PointLight` is attenuated by the reciprocal of the sum of:

- The constant attenuation factor.
- The linear attenuation factor times the distance between the light and the vertex being illuminated.
- The quadratic attenuation factor times the square of the distance between the light and the vertex.

By default, the constant attenuation value is 1.0 and the other two values are 0.0. The results in no attenuation being applied to the light source. The brightness of a `PointLight` source at a specific distance from the light source can be calculated using the following equation:

$$brightness = \frac{intensity}{const + lin \times dist + quad \times dist^2} \tag{2.2}$$

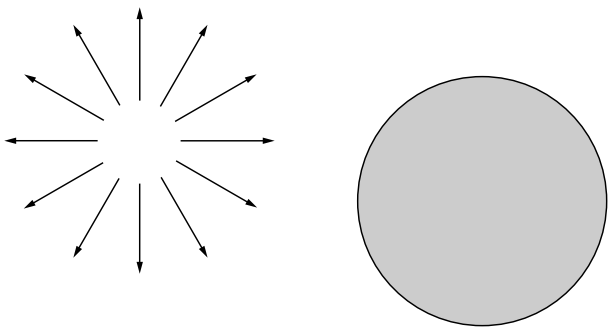


Figure 2.31: A `PointLight` represents light from a specific location. The light rays associated with the point light source all emanate from this point.

An illustration of how a point light source operates is presented in Figure 2.31. The most comprehensive constructor for a point light has the following format:

- `PointLight(boolean on, Color3f colour, Point3f position, Point3f attenuation)`
  Creates a point light source with the specified state, colour, position and attenuation.

The position and the attenuation of a `PointLight` can be updated or retrieved using:

- `void setAttenuation(Point3f attenuation)`
  Sets the attenuation attributes of this light source. The x coordinate of the point represents the constant attenuation factor, the y coordinate represents the linear attenuation factor and the z coordinate represents the quadratic attenuation factor.

- `void getAttenuation(Point3f attenuation)`
  Retrieves the attenuation attributes for this light source and stores them in the supplied `Point3f` object using the method described above.

- `void setPosition(Point3f position)`
  Sets the position of this point light source.

- `void getPosition(Point3f position)`
  Retrieves the position of this point light source and stores it in the supplied `Point3f` object.

If these methods are to be used after the scene graph has gone live, then the following capability bits must be set:

- `ALLOW_ATTENUATION_READ`
  Indicates that this point light allows read access to its attenuation information after the scene graph has gone live.

- `ALLOW_ATTENUATION_WRITE`
  Indicates that this point light allows write access to its attenuation information after the scene graph has gone live.

- `ALLOW_POSITION_READ`
  Indicates that this point light allows read access to its position information after the scene graph has gone live.

- `ALLOW_POSITION_WRITE`
  Indicates that this points light allows write access to it position information after the scene graph has gone live.

The following example demonstrates how a `PointLight` can be used to illuminate a pair of objects.

```
0  import javax.media.j3d.*;
   import javax.vecmath.*;
   import com.sun.j3d.utils.geometry.*;

   public class PointLightExample extends BasicSceneWithMouseControl
5  {
     public static void main(String args[]){new PointLightExample();}

     public BranchGroup createContentBranch()
     {
10     BranchGroup root = new BranchGroup();

       Transform3D transform1 = new Transform3D();
       transform1.setTranslation(new Vector3f(-0.5f, 0.0f, -0.5f));
       TransformGroup tg1 = new TransformGroup(transform1);
15     root.addChild(tg1);

       // Red appearance
       Appearance appearance1 = new Appearance();
       Material material1 = new Material();
20     material1.setDiffuseColor(new Color3f(1.0f, 0.0f, 0.0f));
       appearance1.setMaterial(material1);

       Sphere sphere1 = new Sphere(0.4f,
           Sphere.GENERATE_NORMALS,
25         100,appearance1);
       tg1.addChild(sphere1);

       Transform3D transform2 = new Transform3D();
       transform2.setTranslation(new Vector3f(0.5f, 0.0f, -0.5f));
30     TransformGroup tg2 = new TransformGroup(transform2);
       root.addChild(tg2);

       // Blue appearance
       Appearance appearance2 = new Appearance();
35     Material material2 = new Material();
       material2.setDiffuseColor(new Color3f(0.0f, 0.0f, 1.0f));
       appearance2.setMaterial(material2);

       Sphere sphere2 = new Sphere(0.4f,
40         Sphere.GENERATE_NORMALS,
           100,appearance2);
```

```
        tg2.addChild(sphere2);

        PointLight light = new PointLight();
        light.setInfluencingBounds(new BoundingSphere(new Point3d(),
            Double.MAX_VALUE));
        root.addChild(light);

        root.compile();

        return root;
    }
}
```

This program creates two spheres of radius 40 cm and positions them 50 cm behind the origin. The sphere with the red material is moved 50 cm in the negative x direction and the sphere with the blue material is moved 50 cm in the positive x direction. A single point light with the default parameters, white colour (1.0, 1.0, 1.0) and no attenuation, is placed at the origin in order to illuminate the two spheres. The output generated when this program is executed is illustrated in Figure 2.32.

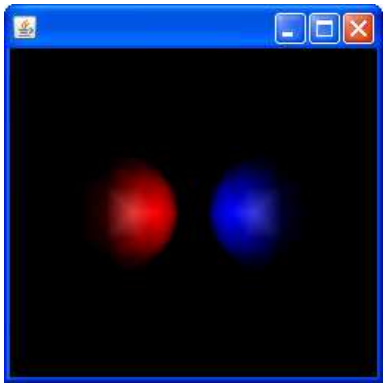


Figure 2.32: An examples of two spheres located left and right of the origin illuminated by a `PointLight` that is located at the origin.

2.7.9.3 `SpotLight`

The `SpotLight` class specifies an attenuated light source located at a fixed point in space that radiates light in a specified direction. The `SpotLight` class extends `PointLight` to include this additional functionality. A `SpotLight` node has the same attributes as a `PointLight` node, with the addition of the following:

- Direction - The axis of the cone of light. The default direction is (0.0, 0.0, -1.0), i.e. into the screen. The direction of the spotlight is only significant when the spread angle is not π radians (the default value for this attribute).
- Spread angle - The angle in radians between the direction axis and a ray along the edge of the cone of light.
  - Note that the angle of the cone at the apex is twice this angle.

  - The range of values for the spread angle is [0.0, $\frac{\pi}{2}$] radians, with a special value of π radians.
  - Spread angle values lower than 0 are clamped to 0 and values greater than $\frac{\pi}{2}$ are clamped to $\frac{\pi}{2}$.

- Concentration - Specifies how quickly the light intensity attenuates as a function of the angle of radiation as measured from the direction of radiation. The light's intensity is highest at the centre of the cone and is attenuated towards the edges of the cone by the cosine of the angle between the direction of the light and the direction from the light to the object being lit, raised to the power of the spot concentration exponent. The higher the concentration value, the more focused the light source. The range of values is [0.0, 128.0]. The default concentration is 0.0, which provides uniform light distribution.

A `SpotLight` contributes to diffuse and specular reflections, which depend on the orientation and position of an object's surface. A `SpotLight` does not contribute to ambient reflections. An illustration of the spread angle and the concentration for a `SpotLight` node are illustrated in Figure 2.33.

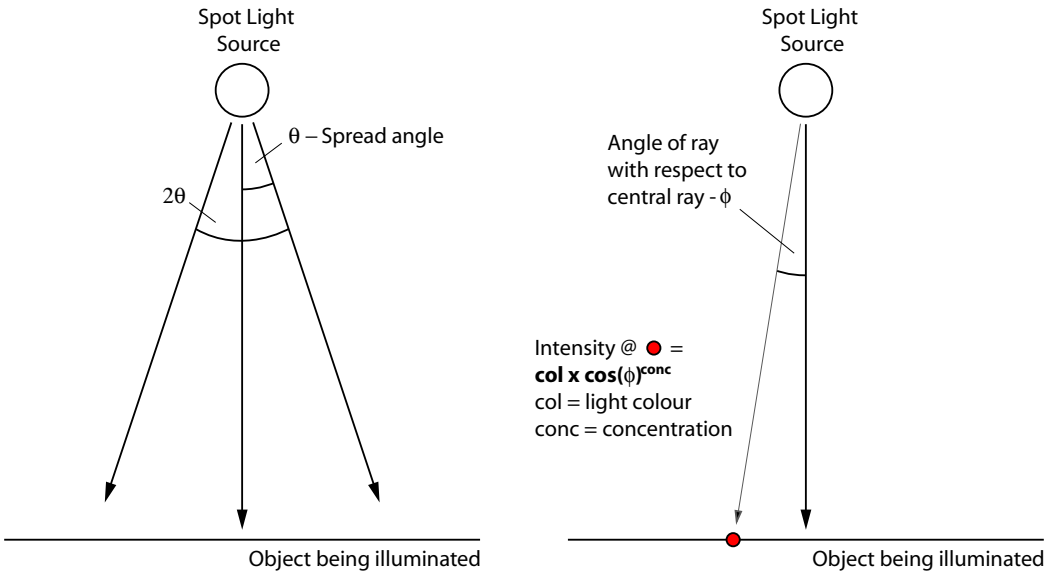


Figure 2.33: An illustration of two of the main attributes associated with a spot light. The spread angle (a) and the concentration (b).

The most comprehensive constructor for a `SpotLight` has the following format:

- `SpotLight(boolean on, Color3f color, Point3f position, Point3f attenuation, Vector3f dir, float spreadAngle, float conc)`
  Creates a new instance of a `SpotLight` object with the specified attributes.

The `SpotLight` class also defines methods to set or retrieve its various attributes:

- `void setConcentration(float concentration)`
  Sets the concentration for this `SpotLight` object.

- `float getConcentration()`
  Retrieves the concentration for this `SpotLight` object.

- `void setDirection(Vector3f direction)`
  Sets the direction for this `SpotLight` object to the specified vector.

- `void getDirection(Vector3f direction)`
  Retrieves the direction for this `SpotLight` and stores it in the specified vector.

- `void setSpreadAngle(float spreadAngle)`
  Sets the spread angle for this `SpotLight` object. The minimum spread angle is 0 radians and the maximum spread angle is $\frac{\pi}{2}$ radians.

- `float getSpreadAngle()`
  Retrieves the spread angle for this `SpotLight` object.

Note: If you are more comfortable specifying angles in degrees then the following method may be useful:

- `double Math.toDegrees(double radians)`
  Converts the specified angle from radians to degrees.

- `double Math.toRadians(double degrees)`
  Converts the specified angle from degrees to radians.

The `SpotLight` class also defines a series of capability bits that can be used to enable access to its attributes after the scene graph has gone live, these include:

- `ALLOW_CONCENTRATION_READ`
  Indicates that this `SpotLight` allows read access to its concentration information after the scene graph has gone live.

- `ALLOW_CONCENTRATION_WRITE`
  Indicates that this `SpotLight` object allows write access to its concentration information after the scene graph has gone live.

- `ALLOW_DIRECTION_READ`
  Indicates that this `SpotLight` object allows read access to its direction information after the scene graph has gone live.

- `ALLOW_DIRECTION_WRITE`
  Indicates that this `SpotLight` object allow write access to its direction information after the scene graph has gone live.

- `ALLOW_SPREAD_ANGLE_READ`
  Indicates that this `SpotLight` object allows read access to its spread angle information after the scene graph has gone live.

- `ALLOW_SPREAD_ANGLE_WRITE`
  Indicates that this `SpotLight` object allows write access to its spread angle information after the scene graph has gone live.

The following example demonstrates how a `SpotLight` can be used to illuminate an object.

```
import javax.media.j3d.*;
import javax.vecmath.*;
import java.lang.*;

import com.sun.j3d.utils.behaviors.mouse.MouseRotate;
import com.sun.j3d.utils.geometry.*;

public class SpotLightExample extends BasicScene
{
  public static void main(String args[]){new SpotLightExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Transform3D transform1 = new Transform3D();
    transform1.setTranslation(new Vector3f(0.0f, 0.0f, -20.0f));
    TransformGroup tg1 = new TransformGroup(transform1);
    root.addChild(tg1);

    // Red appearance
    Appearance appearance1 = new Appearance();
    Material material1 = new Material();
    material1.setDiffuseColor(new Color3f(1.0f, 0.0f, 0.0f));
    appearance1.setMaterial(material1);

    // A detailed sphere with a large radius
    Sphere sphere1 = new Sphere(15.0f, Primitive.GENERATE_NORMALS,
        500, appearance1);
    tg1.addChild(sphere1);

    TransformGroup tg2 = new TransformGroup();
    tg2.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg2);

    // A mouse rotate behaviour to control the direction of the light
    MouseRotate rotate = new MouseRotate();
    rotate.setTransformGroup(tg2);
    tg2.addChild(rotate);
    rotate.setSchedulingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));

    // A spot light
    SpotLight light = new SpotLight();
    light.setConcentration(45f);
    light.setSpreadAngle((float)Math.toRadians(30));
    light.setInfluencingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    tg2.addChild(light);

    root.compile();

    return root;
```

```
  }
}
```

This program begins by creating a large detailed sphere of radius 15 metres that is located 20 metres behind the origin. The creation of a detailed sphere is achieved by specifying a high number of subdivisions. The repositioning of the sphere with respect to the origin is achieved using a `TransformGroup`. A `SpotLight` is then created with a white colour, a concentration of 45.0 and a spread angle of 30 degrees. The `SpotLight` is attached to a `TransformGroup` which is associated with a `MouseRotation` behaviour. This setup enables the direction that the `SpotLight` is shining to be controlled by the mouse. The type of renderings obtained when this program is executed are illustated in Figure 2.34.

(a) (b)

Figure 2.34: An illustration of a highly detailed `Sphere` primitive illuminated by a `SpotLight` with a concentration of 45.0 and a spread angle of 30 degrees.

2.7.9.4 `AmbientLight`

An `AmbientLight` is used to represent light that appears to come from all directions. The `AmbientLight` class extends the `Light` class and consequently has the same attributes including colour, influencing bounds, scopes and a flag indicating whether the light source is on or off. Ambient reflections do not depend on the orientation or position of a surface. Ambient light only has an ambient reflection component. It does not have either diffuse or specular reflection components.

An `AmbientLight` object can be created using the following constructor:

- `AmbientLight(boolean lightOn, Color3f color)`
  Creates a new `AmbientLight` object with the specified state and colour information.

The `AmbientLight` class does not define any additional methods or capabilities as all of its required functionality is defined by the `Light` class.

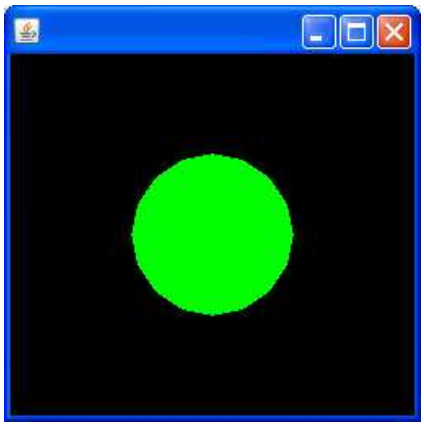
The following example demonstrates how an `AmbientLight` can be used to illuminate an object.

```
0
import javax.vecmath.*;
import javax.media.j3d.*;
import com.sun.j3d.utils.geometry.*;

5 public class AmbientLightExample extends BasicScene
{
  public static void main(String args[]){new AmbientLightExample();}

  public BranchGroup createContentBranch()
10  {
    BranchGroup root = new BranchGroup();

    // Create an yellow ambient light
    AmbientLight light = new AmbientLight(new Color3f(1.0f, 1.0f, 0.0f));
15  BoundingSphere bounds = new BoundingSphere(new Point3d(0.0, 0.0, 0.0),
        Double.POSITIVE_INFINITY);
    light.setInfluencingBounds(bounds);
    root.addChild(light);

20  // Create a cyan material
    Appearance appearance = new Appearance();
    Material material = new Material();
    material.setAmbientColor(new Color3f(0.0f, 1.0f, 1.0f));
    appearance.setMaterial(material);
25
    // Create a sphere of radius 40 cm
    Sphere sphere = new Sphere(0.4f, appearance);
    root.addChild(sphere);

30  root.compile();

    return root;
  }
}
```

The operation of this program is quite straightforward. An `AmbientLight` is created and positioned at the origin (the default location). The colour of the `AmbientLight` is set to yellow in the constructor. A `Sphere` primitive is created with a radius of 40 cm. The sphere has a `Material` with an ambient colour of cyan. Consequently, the colour of the sphere under the ambient light will be green. This is illustrated in Figure 2.35.

2.7.10 Texture Mapping

Texture mapping changes the appearance of a shape by wrapping an image around the structure of the shape. The use of an image, or texture, in this way enables the creation of extremely detailed content in a relatively straightforward manner. For example, a table textured with a wood grain texture would look more realistic than if a solid brown colour were used. Textures are applied, or mapped, to a surface using data that relates each vertex in the geometry to a location in the texture. The



(a)

Figure 2.35: A sphere with a material that has an ambient colour of cyan illuminated by yellow ambient light. This results in a sphere that appears to have a green colour.

locations in the texture are specified using texture coordinates and the texture is considered to be a rectangular array of colour values called texels.

Note: A texel is to a texture what a pixel is to a picture.

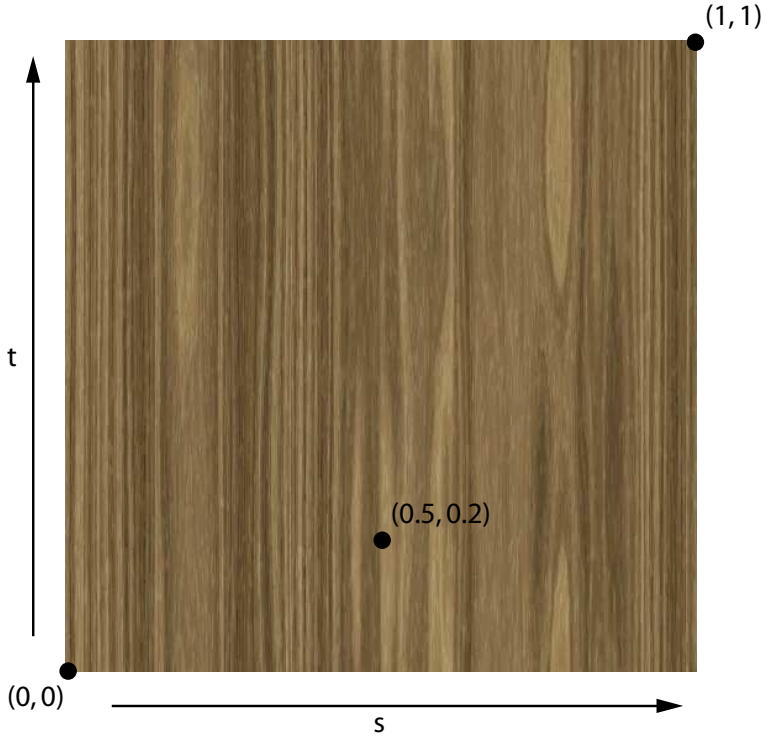
2.7.10.1 Texture Coordinates

The position of a pixel in an image is represented using x and y values. In a similar way, the position of a texel in a texture is represented using s and t values. These values are known as texture coordinates. The s coordinate corresponds to the horizontal axis of the texture and the t coordinate corresponds to the vertical axis of the texture. The lower left hand corner of the texture is at $(0, 0)$ and the upper right hand corner of the image is at $(1, 1)$. This coordinate system is illustrated in Figure 2.36.

Note: If a texture image is non-square the texture coordinate still have the same range. For example, if the texture is a 128×256 image the top right hand corner of the texture will have the coordinates $(1, 1)$ and not (0.5, 1).

Texture mapping stretches the texture to make the texture locations specified by the texture coordinates line up with the texture coordinates assigned to the vertices of the geometry being texture mapped. Texture mapping is controlled by several components:

- The `Texture` appearance component controls the texture image.
- The `TextureAttributes` appearance component control how the texture is applied to the surface of the geometry that is being texture mapped.
- The texture coordinates are specified by the `Geometry`. If the `Geometry` does not have texture coordinates, a `TextureCoordGeneration` appearance component can be used to generate texture coordinates from the geometric coordinates.



(a)

Figure 2.36: An illustration of the texture coordinate system. The location of each texel is represented by a pair of texture coordinates (s, t). The s coordinate represents the location of the texel in the horizontal direction and the t coordinate represents the location of the texel in the vertical direction. The origin of the texture coordinate system is located at the bottom left hand corner of the texture.

2.7.10.2 `Texture`

The base class for textures is `Texture`. The `Texture` class has two main subclasses:

- `Texture2D` - specifies a 2D image that is to be mapped to the exterior of a particular geometry.
- `Texture3D` - specified a 3D or volumetric texture that can be used in volume rendering.

The most straightforward method for the generation of a `Texture2D` object is to use a `TextureLoader`. The `TextureLoader` loads the texture and sets up several of its basic properties. A `TextureLoader` object can be created using one of the following constructors:

- `TextureLoader(BufferedImage image)`
  Constructs a `TextureLoader` object using the specified `BufferedImage` and the default format RGBA.
- `TextureLoader(Image image, Component observer)`
  Constructs a `TextureLoader` object using the specified `Image` and the default format RGBA.

- `TextureLoader(String filename, Component observer)`
  Constructs a `TextureLoader` object that will load a texture from the specified file location.

- `TextureLoader(URL url, Component observer)`
  Constructs a `TextureLoader` object that will load a texture from the specified URL.

Note: In the last three versions of the constructor listed here, an `ImageObserver` object must be specified. This is required to monitor the progress of `Image` object that are in the process of being loaded.

It is clear that a `TextureLoader` object can be used to load a texture image from a variety of sources. The texture loader also supports the image formats supported by the JDK i.e. GIF and JPEG. If the optional Java Advanced Imaging (JAI) package is installed then JAI will be used to load the texture image and consequently the loader will support BMP, FlashPix, PNG, PNM and TIFF file formats.

It is also possible to specify a series of flags in the constructor, for example:

- `TextureLoader(BufferedImage image, int flags)`
  Constructs a `TextureLoader` object using the specified `BufferedImage` object, the specified flag options and the default format RGBA.

The flags are used to specify options for the loader. The options that can be specified by the flags are integers that can be OR'd together. The possible flag values are:

- `GENERATE_MIPMAP` - Tells the `TextureLoader` to create the texture with multiple levels of resolution called mipmaps that are used when the texture is viewed at a variety of scales.

- `BY_REFERENCE` - Specifies that the `ImageComponent2D` object representing the texture will access the image data by reference.

- `Y_UP` - Indicates that the `ImageComponent2D` object representing the texture will have a y-orientation of y up, meaning that the origin of the image is in the lower left hand corner (i.e. so that the image coordinate system corresponds to the texture coordinate system).

It is also possible to specify an image format in the constructor for a `TextureLoader` object, for example:

- `TextureLoader(BufferedImage image, int format)`
  Constructs a `TextureLoader` object with the specified `BufferedImage` and the specified image format.

The format parameter is an advanced option that is used to specify the internal format of the image. The default format is RGBA, i.e. this is the format that is used if the version of the constructor being used does not have a format parameter. The RGBA format indicates that each texel has red, green, blue and alpha components. A variety of other formats are also available, for example ALPHA, indicates that only the transparency values of the loaded image are to be used.

Once a `TextureLoader` object has been created the `Texture` object that it represents can be obtained using the following method:

- `Texture getTexture()`
  Returns the relevant `Texture` object or null if the texture image failed to load.

Note: If the width or height of the image is not a power of 2 (i.e. 32, 64, 128, 256, etc.), then the image is scaled so that its dimensions are a power of two.

The other methods provided by the `TextureLoader` class can be used to load an `ImageComponent2D` representation of the texture image. This can be used in conjunction with the `Raster` geometry and `Background` environment node. These methods have the following format:

- `ImageComponent2D getImage()`
  Returns an `ImageComponent2D` representation of the texture image.
- `ImageComponent2D getScaledImage(float xScale, float yScale)`
  Returns a scaled `ImageComponent2D` representation of the texture image that has been scaled by the specified horizontal and vertical scale factors.
- `ImageComponent2D getScaledImage(int width, int height)`
  Returns a scaled `ImageComponent2D` representation of the texture image that has the specified dimensions.

The following program demonstrates how a texture can be applied to a simple triangle:

```
0  import javax.media.j3d.*;
   import com.sun.j3d.utils.image.*;

   public class TextureCoordinateExample extends BasicSceneWithMouseControl
   {
5    public static void main(String args[]){new TextureCoordinateExample();}

     public BranchGroup createContentBranch()
     {
       BranchGroup root = new BranchGroup();
10
       Texture woodTexture = null;

       try
       {
15       // Load the wood texture from the local file syste,
         TextureLoader loader = new TextureLoader("wood.jpg", this);
         woodTexture = loader.getTexture();
       }
       catch(Exception e){System.out.println(e.toString());}
20
       Appearance appearance = new Appearance();
       appearance.setTexture(woodTexture);

       float [] coordinates = {-0.5f, -0.5f,  0.0f,
25         0.5f, -0.5f,  0.0f,
           0.0f,  0.5f,  0.0f};
```

```
    // Define the texture coordinates for the vertices
    float [] texCoords = {0.0f, 0.0f,
        1.0f, 0.0f,
        0.5f,  1.0f};

    // Create a geometry array from the specified coordinates
    GeometryArray geometryArray = new TriangleArray(3,
    GeometryArray.COORDINATES|GeometryArray.TEXTURE_COORDINATE_2);

    geometryArray.setCoordinates(0, coordinates);
    geometryArray.setTextureCoordinates(0,0,texCoords);

    // Create a Shape3D object using the GeometryArray
    Shape3D shape = new Shape3D(geometryArray, appearance);
    root.addChild(shape);

    root.compile();

    return root;
  }
}
```

The program begins by creating a `Texture2D` object that represents the wood texture stored in the file `wood.jpg`. This is achieved using a `TextureLoader` object. An `Appearance` object is then created and its texture is set to be the loaded `Texture2D` object. A set of coordinates are then defined that represent a simple triangle located at the origin. A set of texture coordinates are subsequently defined. These associate the vertices of the triangle with points in the texture image. The first vertex is associated with the the bottom left corner of the texture image, the second vertex is associated with the bottom right corner of the image and the third vertex is associated with the central point at the top of the texture image. A `TriangleArray` object is then created and the `COORDINATE` and `TEXTURE_COORDINATE_2` flags are set to indicate that both coordinates and 2D texture coordinates will be specified for the vertices of the triangle array. A `Shape3D` object is then created using the previously discussed appearance and geometry. Finally, the resulting shape is added to the root of the scene graph. A rendering of the texture mapped triangle is illustrated in Figure 2.37.

The following example demonstrates how texture coordinates can be generated to example texture mapping for a primitive shape.

```

import javax.media.j3d.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;

public class TexturePrimitiveExample extends BasicSceneWithMouseControl
{
  public static void main(String args[]){new TexturePrimitiveExample();}

  public BranchGroup createContentBranch()
```

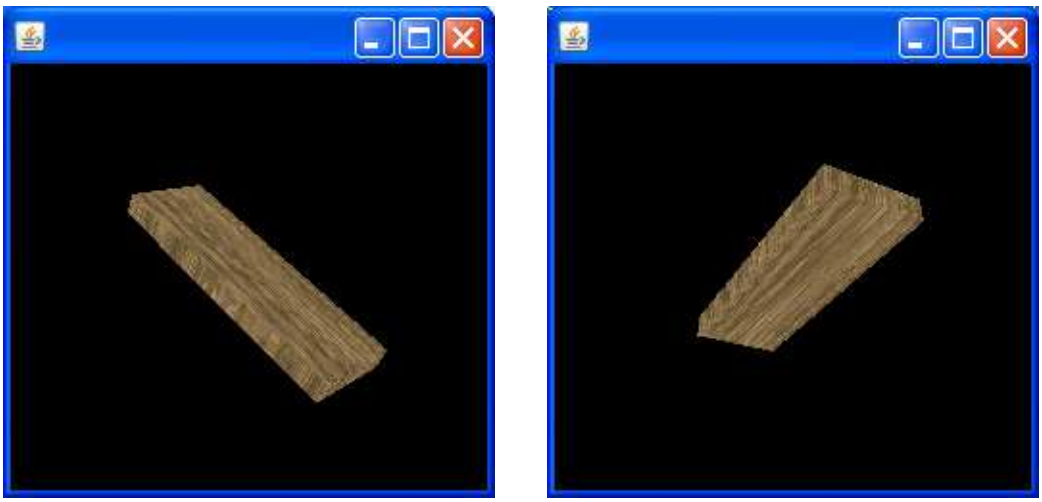
(a) (b)

Figure 2.37: The original wood texture (a) and a triangle that has been texture mapped using the wood texture (b).

```
10    {
        BranchGroup root = new BranchGroup();

        Appearance appearance = new Appearance();

15      Texture woodTexture = null;
        try
        {
          // Load the wood texture from the local file system
          TextureLoader loader = new TextureLoader("wood.jpg", this);
20        woodTexture = loader.getTexture();
        }
        catch(Exception e){System.out.println(e.toString());}

        appearance.setTexture(woodTexture);
25
        // Create a Box primitive with texture coordinates
        Box box = new Box(0.2f, 0.05f, 0.6f,
            Box.GENERATE_TEXTURE_COORDS,
            appearance);
30      root.addChild(box);

        root.compile();

        return root;
35    }
}
```

The operation of this example is similar to the operation of the previous example. The wood texture is loaded and associated with the `Appearance` in the same way. A `Box` primitive is then created with a width of 20 cm, a height of 5 cm and a depth of 60 cm. The `GENERATE_TEXTURE_COORDS` is specified to indicate that the

`Box` primitive should have texture coordinates associated with its vertices. Finally, the appearance with the associated wood texture is specified for use with the `Box` primitive. A rendering of the texture mapped `Box` is illustrated in Figure 2.38.



(a) (b)

Figure 2.38: Sample renderings of the texture mapped `Box` primitive.

The texture image is the most important attribute of the `Texture` class. However, the `Texture` class has a variety of other attributes that define how the texture appears when it is viewed, for example:

- The state attribute allows the texture to be enabled or disabled.
- The boundary mode specifies how the texture appears for texture coordinates outside the range [0, 1].
- The texture filtering mode specifies how the texture is drawn when it is larger or smaller than its original size.

Texture mapping is enabled if all three of the following are true:

- The shape has texture coordinates
- The appearance has a texture image associated with it
- The texture is enabled

The texture is enabled using the following method:

- `void setEnable(boolean state)`
  Sets the state of the texture to the specified value. A value of true indicates that the texture is enabled and a value of false indicates that the texture is disabled.

The state information for a texture can be accessed after the scene graph has gone live provided that the relevant capabilities are enabled, these are:

- `ALLOW_ENABLE_READ`
  Indicates that this `Texture` object allows read access to its state information after the scene graph has gone live.

- `ALLOW_ENABLE_WRITE`
  Indicates that this `Texture` object allows write access to its state information after the scene graph has gone live.

The `Texture` class also defines similar capabilities for the other attributes that it supports.

The coordinates of a texture image always have values in the range [0, 1], however, the texture coordinates associated with the vertices of a geometry can have texture coordinates outside this range. The boundary mode of a texture specifies how texture coordinates outside the range [0, 1] are dealt with. There are two main types of boundary mode:

- `CLAMP` - clamps texture coordinate to be in the range [0, 1]. Texture boundary texels are used for values that fall outside this range.

- `WRAP` - repeat the texture by wrapping texture coordinates that are outside the range [0, 1]. Only the fractional portion of the texture coordinates will be used here. The integer portion in discarded, e.g. 1.5 would become 0.5.

The boundary mode for the horizonal and vertical directions are specified separately using the following methods:

- `void setBoundaryModeS(int mode)`
  Sets the horizontal boundary mode of this `Texture` object to the specified value.

- `void setBoundaryModeT(int mode)`
  Sets the vertical boundary mode of this `Texture` object to the specified value.

The following example demonstrated the operation of the two different boundary modes.

```
0  import javax.media.j3d.*;
   import com.sun.j3d.utils.image.*;

   public class TextureBoundaryModeExample extends BasicSceneWithMouseControl
   {
5    public static void main(String args[]){new TextureBoundaryModeExample();}

     public BranchGroup createContentBranch()
     {
       BranchGroup root = new BranchGroup();
10
       Texture earthTexture = null;

       try
       {
15       // Create the texture and set it horizontal and vertical
```

```
        // boundary modes
        TextureLoader loader = new TextureLoader("earth.jpg", this);
        earthTexture = loader.getTexture();
        earthTexture.setBoundaryModeS(Texture.CLAMP);
        earthTexture.setBoundaryModeT(Texture.WRAP);
      }
      catch(Exception e){System.out.println(e.toString());}

      Appearance appearance = new Appearance();
      appearance.setTexture(earthTexture);

      float [] coordinates = {-0.5f, -0.5f,  0.0f,
          0.5f, -0.5f,  0.0f,
          0.5f,  0.5f,  0.0f,
          -0.5f,  0.5f,  0.0f};

      // Specify the texture coordiantes for the vertices
      float [] texCoords = {-1.0f, -1.0f,
          2.0f, -1.0f,
          2.0f,  2.0f,
          -1.0f,  2.0f};

      // Create a geometry array from the specified coordinates
      GeometryArray geometryArray = new QuadArray(4,
      GeometryArray.COORDINATES|GeometryArray.TEXTURE_COORDINATE_2);

      geometryArray.setCoordinates(0, coordinates);
      geometryArray.setTextureCoordinates(0,0,texCoords);

      // Create a Shape3D object using the GeometryArray
      Shape3D shape = new Shape3D(geometryArray, appearance);
      root.addChild(shape);

      root.compile();

      return root;
    }
}
```

This example defines a `QuadArray` geometry consisting of a single quadrilateral facing the viewer. The quadrilateral has sides that are one meter in length and it is centred at the origin. The horizontal and vertical texture coordinates assigned to the vertices of the quadrilateral are in the range [-1.0, 2.0], i.e. they extend outside the range defined for the texture image. The horizontal boundary mode for the texture image is set to `CLAMP` and the vertical boundary mode for the texture image is set to `WRAP`. The output obtained when this program is executed is illustrated in Figure 2.39.

The `Texture` class also defines filtering modes that specify how the resolution for the texture image is increased or decreased during rendering. When a texture is mapped to a piece of geometry there is rarely a one-to-one correspondence between the pixels of the rendered geometry and the pixels of the texture image. Instead one

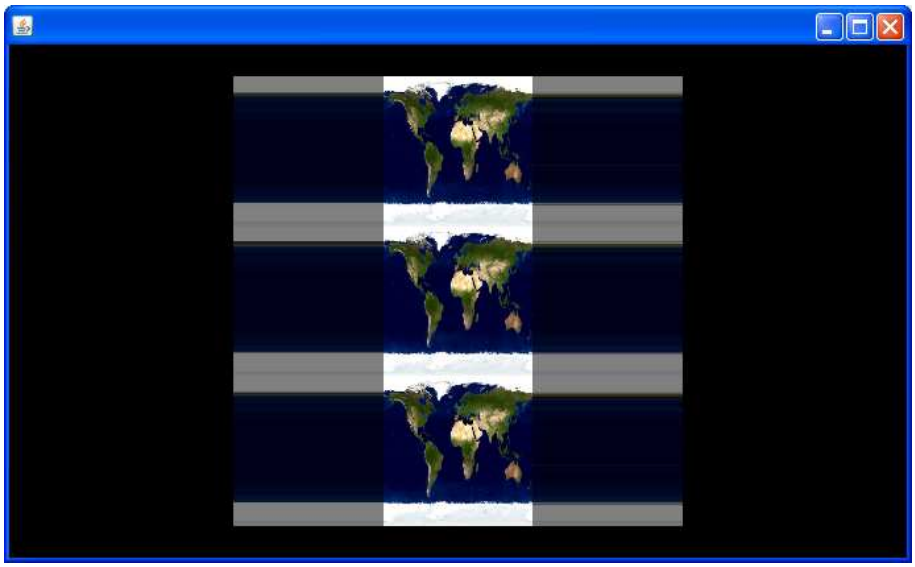


Figure 2.39: A rendering of a texture mapped quadrilateral where the horizontal texture mode has been set to `CLAMP` and the vertical texture mode has been set to `WRAP`.

of the following situations occurs:

- **Magnification** - where a single pixel of the rendered geometry corresponds to a small portion of a texel.
- **Minification** - where a single pixel of the rendered geometry corresponds to an area of the texture, i.e. several texels.

The texture filtering mode specifies the quality of the process used to magnify or minify the texture. The better the quality the less "blocky" the texture mapped geometry will appear. The possible the minification filter modes include:

- `BASE_LEVEL_POINT`
  Selects the nearest point in the base level texture image.
- `BASE_LEVEL_LINEAR`
  Performs bilinear interpolation on the four nearest texels in the base level texture image.
- `FILTER4`
  Applies a used defined weight function to the nearest 4×4 texels in the base level texture image.
  Note: The weight function is set using the `setFilter4Func()` method of the `Texture` class.
- `FASTEST`
  Uses the fastest minification filter.
- `NICEST`
  Uses the minification filter that generates the most visually appealing results.

The minification filter mode associated with a `Texture` object can be set or retrieved using the following methods:

- `void setMinFilter(int minFilter)`
  Sets the minification filter for this `Texture` object to the specified value.

- `int getMinFilter()`
  Returns the minification filter value for this `Texture` object.

The magnification filter can also use the modes listed above. The minification filter and the magnification filter mode associated with a `Texture` object can be set or retrieved using the following methods:

- `void setMagFilter(int magFilter)`
  Sets the magnification filter for this `Texture` object to the specified value.

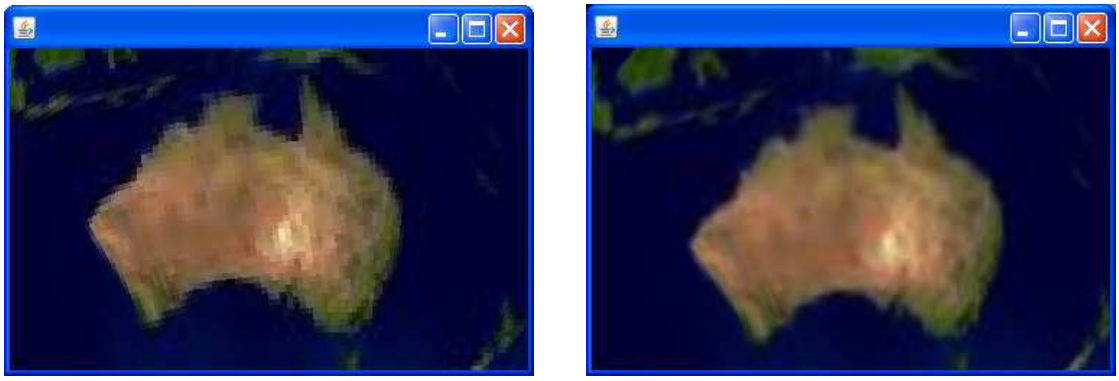
- `int getMagFilter()`
  Returns the magnification filter value for this `Texture` object.

The following example demonstrates how a specific magnification filter mode can be used in conjunction with a `Texture` object.

```
0
import javax.media.j3d.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;

5 public class MagnificationFilterExample extends BasicSceneWithMouseControl
{
  public static void main(String args[]){new MagnificationFilterExample();}

  public BranchGroup createContentBranch()
10 {
    BranchGroup root = new BranchGroup();

    Appearance appearance = new Appearance();

15  Texture earthTexture = null;
    try
    {
      // Load the texture and set its magnification filter
      TextureLoader loader = new TextureLoader("earth.jpg", this);
20    earthTexture = loader.getTexture();
      earthTexture.setMagFilter(Texture.BASE_LEVEL_LINEAR);

    }
    catch(Exception e){System.out.println(e.toString());}
25
    appearance.setTexture(earthTexture);

    // Use the texture in conjucntion with a sphere geometry
    Sphere sphere = new Sphere(0.5f,
30      Sphere.GENERATE_TEXTURE_COORDS,
        appearance);
    root.addChild(sphere);
```

```
        root.compile();
35
        return root;
    }
}
```

This program begins by loading a texture image that represents the surface of the planet earth using a suitably constructed `TextureLoader` object. The magnification filter mode of the loaded texture is set to `BASE_LEVEL_LINEAR`. The `Texture` object is then associated with an `Appearance` object and the `Appearance` object is used to construct a `Sphere` primitive which is ultimately added to the root of the scene graph. Examples of renderings obtained with different version of this program are illustrated in Figure 2.40.



(a) (b)

Figure 2.40: A sphere texture mapped with an image of the earth where Australia is visual rendered using two magnification modes: `BASE_LEVEL_POINT` (a) and `BASE_LEVEL_LINEAR` (b)

It should be noted that there are a variety of other attributes associated with the `Texture` class. These other attributes are discussed in detail in the Java 3D API specification.

2.7.10.3 `TextureAttributes`

The `TextureAttribute` class defines a range of attributes that apply to the texture mapping process. One of the main attributes defined in the `TextureAttributes` class is the texture mode. The texture mode can have one of the following values:

- `MODULATE` - mixes the colour of the texture with the colour of the underlying surface. This texture mode make it possible to use lighting with a texture.
- `DECAL` - applies the texture to the shape being mapped in the form of a decal. This means that the transparency defined in the texture image is preserved.
- `BLEND` - blends the texture blend colour with the colour of the underlying surface. This is an advanced mode where the blending that occurs at each point depends on the values of the pixels in the texture image.

- `REPLACE` - applies the texture directly onto the object overriding the underlying colour of the shape. This is the default texture mode. Textures applied using this mode are not affected by light.
- `COMBINE` - combines the object colour with the texture color or texture blend colour according to the combine operation specified by the texture combine mode. Possible values for the combine mode include:
  - `COMBINE_REPLACE`
  - `COMBINE_MODULATE`
  - `COMBINE_ADD`
  - `COMBINE_SUBTRACT`
  - `COMBINE_INTERPOLATE`

It is possible to set and retrieve the texture mode of a `TextureAttributes` object using the following methods:

- `void setTextureMode(int textureMode)`
  Sets the texture mode of this `TextureAttributes` object to the specified value.
- `int getTextureMode()`
  Returns the texture mode of this `TextureAttributes` object.

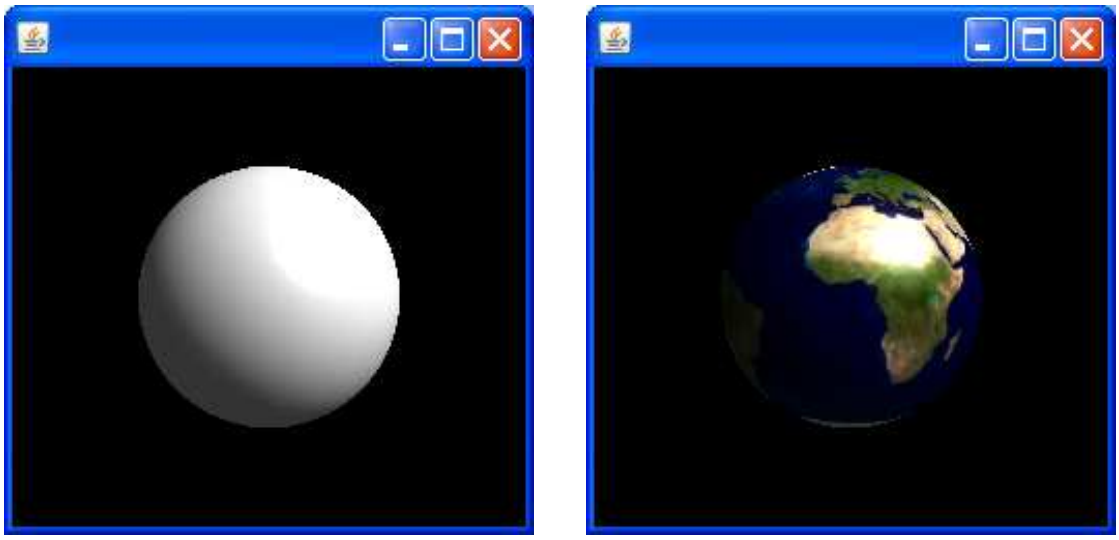
The following example demonstrates how the `TextureAttrubtes` appearance component can be used to make a texture appear to respond to lighting.

```
0
import javax.media.j3d.*;

import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;
5 import javax.vecmath.*;

public class TextureModeExample extends BasicSceneWithMouseControl
{
  public static void main(String args[]){new TextureModeExample();}
10
  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

15  Appearance appearance = new Appearance();

    // Create a white material
    Material material = new Material();
    material.setAmbientColor(new Color3f(1.0f, 1.0f, 1.0f));
20  material.setDiffuseColor(new Color3f(1.0f, 1.0f, 1.0f));
    appearance.setMaterial(material);

    Texture earthTexture = null;
    try
25  {
```

```
        // Load the earth texture
        TextureLoader loader = new TextureLoader("earth.jpg", this);
        earthTexture = loader.getTexture();

    }
    catch(Exception e){System.out.println(e.toString());}
    appearance.setTexture(earthTexture);

    // Set the texture mode to modulate
    TextureAttributes textureAttributes = new TextureAttributes();
    textureAttributes.setTextureMode(TextureAttributes.MODULATE);
    appearance.setTextureAttributes(textureAttributes);

    // Create a sphere primtive with texture coordinates and normals
    Sphere sphere = new Sphere(0.5f,
        Sphere.GENERATE_TEXTURE_COORDS|Sphere.GENERATE_NORMALS,
        50, appearance);
    root.addChild(sphere);

    // Create a bright white directional light
    DirectionalLight light = new DirectionalLight(new Color3f(1.0f, 1.0f, 1.0f),
        new Vector3f(-1.0f, -1.0f, -1.0f));
    light.setInfluencingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    root.addChild(light);

    // Create a dark white ambient light
    AmbientLight ambientLight = new AmbientLight(new Color3f(0.2f, 0.2f, 0.2f));
    ambientLight.setInfluencingBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));
    root.addChild(ambientLight);

    root.compile();

    return root;
  }
}
```

The program begins by creating an `Appearance` object. A `Material` object is then associated with the `Appearance` object. The ambient colour of the `Material` is white (1.0, 1.0, 1.0) and the diffuse colour of the `Material` is also white (1.0, 1.0, 1.0). The "earth" texture is loaded using a suitably constructed `TextureLoader` object. A `TextureAttributes` object is then created and its texture mode is set to `MODULATE`. The `TextureAttributes` object is also associated with the previously constructed `Appearance` object. The `Appearance` object is ultimately used in the construction of a `Sphere` primitive. Two light sources are also included in the scene: a directional light with a bright white colour (1.0, 1.0, 1.0) and an ambient light with a dim white colour (0.2, 0.2, 0.2). Renderings generated by variation of this program are illustrated in Figure 2.41.

It is possible to transform the texture coordinates for a piece of geometry using the a `TextureAttributes` object. This is achieved by specifying a suitable `Transform3D`



(a) (b)

Figure 2.41: Making textures respond to lighting. A white `Sphere` primitive illuminated by ambient and directional light sources (a). A texture applied to the same `Sphere` using the `MODULATE` texture mode appears to respond to both light sources (b).

object using the following method:

- `void setTextureTransform(Transform3D transform)`
  Transforms the texture coordinates using the transform represented by the specified `Transfrom3D` object.

The `TextureAttributes` class also support other attributes and these are discussed in detail in the Java 3D API specification.

2.7.10.4 `TexCoordGeneration`

The `TexCoordGeneration` class defines all of the parameters required for automatic texture coordinate generation and it is included as a part of an `Appearance` object.

Texture coordinates determine which texel in the texture map is assigned to a given vertex. Texture coordinates are interpolated between vertices in a similar method to the way colors are interpolated between vertices.

Rather than the programmer having to explicitly assign texture coordinates to a particular piece of geometry, Java 3D can automatically generate the texture coordinates to achieve texture mapping. The `TexCoordGeneration` class defines attributes that specify the functions for automatically generating texture coordinates. The attributes defines by the `TexCoordGeneration` class include:

- **The texture format -** defines whether the generated texture coordinates are 2D, 3D, or 4D. In the case of 2D texture coordinates this attribute has a value of `TEXTURE_COORDINATE_2`.
- **Texture generation mode -** defines how the texture coordinates are generated. The possible values for this attribute are:

- `OBJECT_LINEAR` - texture coordinates are generated as a linear function of the object coordinates, i.e. in the case of 2D texture coordinates the (s, t) texture coordinates are obtained directly from the (x, y) vertex coordinates.
- `EYE_LINEAR` - texture coordinates are generated as a linear function in eye coordinates. Note that this mode transforms the shapes coordinates to the viewers coordinate system before the texture coordinates are generated.
- `SPHERE_MAP` - texture coordinates are generated using spherical reflection mapping in eye coordinates. This mode is used to simulate the reflected image of a spherical environment onto a polygon.
- `NORMAL_MAP` - texture coordinates are generated to match vertices' normals in eye coordinates.
- `REFLECTION_MAP` - texture coordinates are generated to match vertices' reflection vectors in eye coordinates.

- **Plane equation coefficients** - defines the coefficients for the plane equations used to generate the coordinates in the `OBJECT_LINEAR` and `OBJECT_EYE` texture coordinate generation modes. The coefficients define a reference plane in either object coordinates or in eye coordinates, depending on the texture generation mode.

The following example demonstrates how texture coordinates can be automatically generated for a simple triangular polygon.

```
0  import javax.media.j3d.*;
   import com.sun.j3d.utils.image.*;

   public class TexCoordGenerationExample extends BasicSceneWithMouseControl
   {
5    public static void main(String args[]){new TexCoordGenerationExample();}

     public BranchGroup createContentBranch()
     {
       BranchGroup root = new BranchGroup();
10
       Texture woodTexture = null;

       try
       {
15       // Load the wood texture
         TextureLoader loader = new TextureLoader("wood.jpg", this);
         woodTexture = loader.getTexture();
       }
       catch(Exception e){System.out.println(e.toString());}
20
       Appearance appearance = new Appearance();
       appearance.setTexture(woodTexture);

       // Defines 2D texture coordinates generated by a linear mapping
25     TexCoordGeneration texCoordGeneration = new TexCoordGeneration();
```

```
        texCoordGeneration.setFormat(TexCoordGeneration.TEXTURE_COORDINATE_2);
        texCoordGeneration.setGenMode(TexCoordGeneration.OBJECT_LINEAR);
        appearance.setTexCoordGeneration(texCoordGeneration);

        float [] coordinates = {-0.5f, -0.5f,  0.0f,
            0.5f, -0.5f,  0.0f,
            0.0f,  0.5f,  0.0f};

        // Create a geometry array from the specified coordinates
        GeometryArray geometryArray = new TriangleArray(3,
        GeometryArray.COORDINATES);

        geometryArray.setCoordinates(0, coordinates);

        // Create a Shape3D object using the GeometryArray
        Shape3D shape = new Shape3D(geometryArray, appearance);
        root.addChild(shape);

        root.compile();

        return root;
    }
}
```

This program begins by loading the "wood" texture using a suitably constructed `TextureLoader` object. An `Appearance` object is then created and a `TexCoordGeneration` object is associated with the `Appearance` object. The format of the `TexCoordGeneration` object is set to `TEXTURE_COORD_2` and the texture coordinate generation mode is set to `OBJECT_LINEAR`. The vertices are specified for a `TriangleArray` geometry and a `Shape3D` object is created using the appearance and the geometry. Finally, the `Shape3D` object is added to the scene graph to be displayed. The output obtained when this program is executed is illustrated in Figure 2.42.

2.7.10.5 Using Multiple Textures

The texture options that have been discussed so far have dealt with applying a single texture to a surface. It is also possible to apply several layers of textures to a surface using a process referred to as multitexturing. This is an advanced approach to texturing that can be used to implement shadows and special kinds of lighting.

Multilayered textures are created by specifying a series of `TextureUnitState` objects for each layer of texture mapping. A `TextureUnitState` object holds the `Texture`, `TextureAttributes` and `TextCoordGeneration` objects that represent a specific texture. A `TextureUnitState` object can be associated with an `Appearance` object using one of the following methods:

- `void setTextureUnitState(int index, TextureUnitState state)`
  Set the `TextureUnitState` object for this `Appearance` object at the specified index to the specified value.

- `void setTextureUnitState(TextureUnitState[] stateArray)`
  Set the array of `TextureUnitState` objects for this `Appearance` object to the

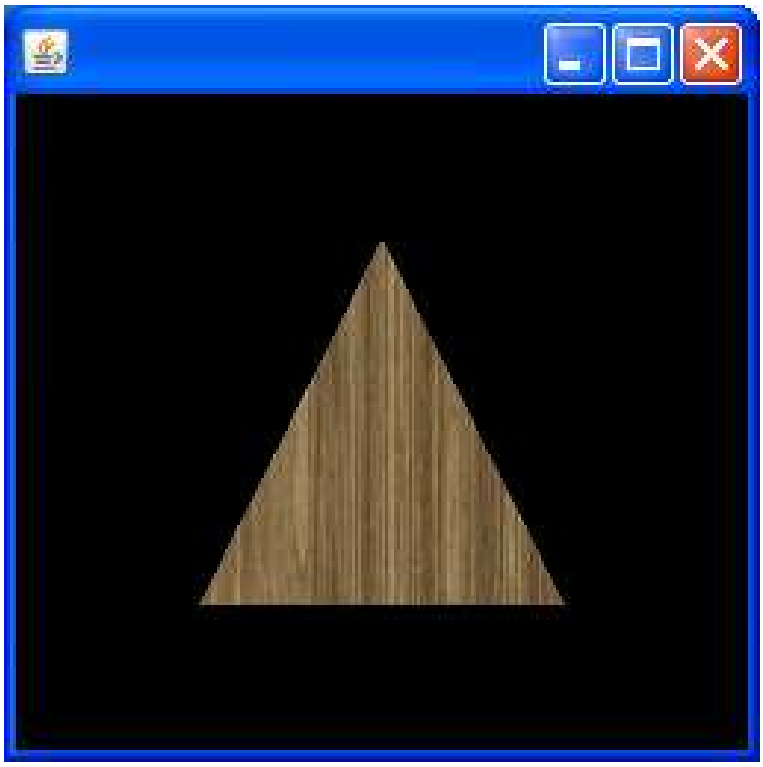


Figure 2.42: A simple triangular geometry where the texture coordinates have been automatically generated using a `TexCoordGeneration` appearance component.

specified array.

It is also possible to specify several sets of texture coordinates with an instance of `GeometryArray`. The mapping of the texture coordinates to a specific `TextureUnitState` object is defined in the constructor for the relevant subclass of `GeometryArray`, for example `TriangleArray`:

- `TriangleArray(int vertexCount, int vertexFormat, int texCoordSetCount, int[] texCoordSetMap)`
  Constructs a `TriangleArray` object that support multiple sets of texture coordinates. The number of sets is specified by the `texCoordSetCount` argument and the mapping between the individual sets of texture coordinates and the associated `TextureUnitState` object is indicated in the `texCoordSetMap` argument.

The texture coordinate associated with a particular `TextureUnitState` object can be specified using the following method:

- `void setTextureCoordinates(int texCoordSet, int index, TexCoord2f[] texCoords)`
  Sets the texture coordinates associated with the vertices starting at the specified index in the specified texture coordinate set for this `GeometryArray` object.

2.7.11 Environment Nodes

Java 3D provides a range of environment nodes that affect the affect the environment of a virtual world. Environment nodes can be used to control lighting, sound and the background of a scene.

2.7.11.1 Bounding Regions

Environment nodes typically affect the a particular region of a virtual world. For example, lights shine on shapes and sounds create audible content. Light and sound sources should not affect the entire virtual universe. Consequently, Java 3D requires bounding regions to be specified for environment nodes to define the region where they are active. There are two main types of bounding region and they are constructed as follows:

- `BoundingBox(Point3d lower, Point3d upper)`
  Creates a new `BoundingBox` object within the specified bounds.
- `BoundingSphere(Point3d centre, double radius)`
  Creates a new `BoundingSphere` object at the specified location with the specified radius.

If a bounding region is not specified for a particular environment node then it is considered to be inactive.

If a environment node is required to be active throughout the entire virtual universe then it is possible to approximate infinite bounds by creating a `BoundingSphere` with a radius of `Double.MAX_VALUE`.

2.7.11.2 Background

The `Background` environment node defines a solid background colour and a background image that are used to fill the window at the beginning of each new frame. The `Background` environment node also allows background geometry to be specified.

A `Background` environment node that represents a colour can be created using the following constructor:

- `Background(Color3f backgroundColour)`
  Creates a new `Background` environment node with the specified colour.

Alternatively, a background environment node that represents a image can be created:

- `Background(ImageComponent2D backgroundImage)`
  Creates a new `Background` environment node using the specified image.

An `ImageComponent2D` object representing an image resource can be obtained by calling the `getImage()` method of a suitably constructed `TextureLoader` object.

The following example demonstrates how a background image can be created and added to a 3D scene.

```
0
import javax.media.j3d.*;

import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;
5 import javax.vecmath.*;

public class BackgroundImageExample extends
    BasicSceneWithMouseControlAndLights
{
10   public static void main(String args[]){new BackgroundImageExample();}

   public BranchGroup createContentBranch()
   {
     BranchGroup root = new BranchGroup();
15
     ImageComponent2D starImage = null;
     try
     {
       // Load the stars image
20     TextureLoader loader = new TextureLoader("stars.gif", this);
       starImage = loader.getImage();
     }
     catch(Exception e){System.out.println(e.toString());}

25   // Create a background node from the stars image
     Background bg = new Background(starImage);
     bg.setApplicationBounds(new BoundingSphere(new Point3d(),
         Double.MAX_VALUE));
     root.addChild(bg);
30
     Appearance appearance = new Appearance();

     Material material = new Material();
     appearance.setMaterial(material);
35
     Texture earthTexture = null;
     try
     {
       // Load the earth texture
40     TextureLoader loader = new TextureLoader("earth.jpg", this);
       earthTexture = loader.getTexture();
     }
     catch(Exception e){System.out.println(e.toString());}
     appearance.setTexture(earthTexture);
45
     // Allow the texture to respond to lighting
     TextureAttributes textureAttributes = new TextureAttributes();
     textureAttributes.setTextureMode(TextureAttributes.MODULATE);
     appearance.setTextureAttributes(textureAttributes);
50
     // Create a details sphere and map the earth texture
     Sphere sphere = new Sphere(0.5f,
```

```
            Sphere.GENERATE_TEXTURE_COORDS|Sphere.GENERATE_NORMALS,
            100, appearance);
        root.addChild(sphere);

        root.compile();

        return root;
    }
}
```

This program is a modified version of the `TextureModeExample.java` example that was discussed earlier. The main difference here is that a `Background` environment node is added to the scene. This is achieved by loading the required background image using a suitably constructed `TextureLoader` object and then calling the `getImage()` method to obtain a `ImageComponent2D` object that represents the loaded image. This image is subsequently used to construct a `Background` object. The application bounds for the `Background` environment node are set to the maximum value and the `Background` is added to the root of the scene graph. The rendering obtained when this program is executed is illustrated in Figure 2.43

Figure 2.43: A texture mapped sphere representing the planet earth and a background image representing a backdrop of stars.

It is also possible to use a geometry rather than a flat image the create a background. This involves rendering a texture mapped `Sphere` at an infinite distance. The normals of the sphere must be flipped inwards so that the texture is applied to the interior of the `Sphere`.

The following example demonstrates how a background geometry can be specified for a scene.

```

import javax.media.j3d.*;

import com.sun.j3d.utils.behaviors.mouse.MouseRotate;
```

```
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.image.*;
import javax.vecmath.*;

public class BackgroundGeometryExample extends
  BasicSceneWithMouseControlAndLights
{
  public static void main(String args[]){new BackgroundGeometryExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    Texture starTexture = null;
    try
    {
      // Load the stars texture
      TextureLoader loader = new TextureLoader("stars.gif", this);
      starTexture = loader.getTexture();
    }
    catch(Exception e){System.out.println(e.toString());}

    // Create a backgound node with maximum bounds
    Background background = new Background();
    background.setApplicationBounds(new BoundingSphere(new Point3d(),
        Double.MAX_VALUE));

    BranchGroup backgroundGroup = new BranchGroup();

    // Set the texture for the background appearance
    Appearance backgroundAppearance = new Appearance();
    backgroundAppearance.setTexture(starTexture);

    // Create a detailed sphere with normal pointing inwards
    Sphere backgroundSphere = new Sphere(0.5f,
        Sphere.GENERATE_TEXTURE_COORDS|
        Sphere.GENERATE_NORMALS_INWARD,
        100, backgroundAppearance);
    backgroundGroup.addChild(backgroundSphere);
    background.setGeometry(backgroundGroup);
    root.addChild(background);

    // Create the earth sphere
    Appearance earthAppearance = new Appearance();

    Material earthMaterial = new Material();
    earthAppearance.setMaterial(earthMaterial);

    Texture earthTexture = null;
    try
    {
      TextureLoader loader = new TextureLoader("earth.jpg", this);
      earthTexture = loader.getTexture();
```

```
        }
        catch(Exception e){System.out.println(e.toString());}
        earthAppearance.setTexture(earthTexture);

        Sphere earthSphere = new Sphere(0.5f,
            Sphere.GENERATE_TEXTURE_COORDS|
            Sphere.GENERATE_NORMALS,
            100, earthAppearance);
        root.addChild(earthSphere);

        root.compile();

        return root;
    }
}
```

The main difference between this example and the previous background example is that a background geometry is used rather than at flat background. The background geometry is represented by a `BranchGroup` with a single `Sphere` child. The `Sphere` generates normals that are projected inwards so that the texture is mapped to the inside of the sphere rather than the outside. The `BranchGroup` representing the background geometry is associated with a `Background` object using the `setGeometry()` method and the `Background` object is ultimately added to the root of the scene graph using the `addChild()` method. The output obtained when this program is executed is illustrated in Figure 2.44.

Figure 2.44: A scene where the background is represented by a spherical geometry rendered at infinity. The normals of the sphere point inwards so that the background texture is mapped to the interior of the sphere rather than the exterior.

2.7.11.3 `Fog`

The `Fog` environment node simulates the way that objects appear to fade into the background when viewed from a distance. Fog is a useful feature and can be used to add a great deal of realism to a scene. The effect caused by fog is sometimes called "depth cueing" because it gives the brain a visual cue as to the depth of an object in the scene. Fog is implemented in Java 3D by blending the fog colour with the colour of the scene objects based on their distance from the viewer. Java 3D provides support for two types of `Fog` environment nodes:

- `LinearFog` - has a constant density, so the level of obscurity generated by the fog increases linearly as the viewer moves away from the object being viewed.
- `ExponentialFog` - the fog density increases exponentially so that the level of obscurity increases exponentially as the view moves away from the object being viewed.

`LinearFog` has three main attributes:

- **Colour -** defines the colour of the fog.
- **Front distance -** anything closer to the viewer than the front distance is not affected by the fog.
- **Back distance -** anything further away from the view than the back distance is completely obscured by the fog.

A `LinearFog` object can be created using the following constructor:

- `LinearFog(Color3f colour, double frontDistance, double backDistance)` Creates a `LinearFog` object with the specified colour, front distance and back distance.

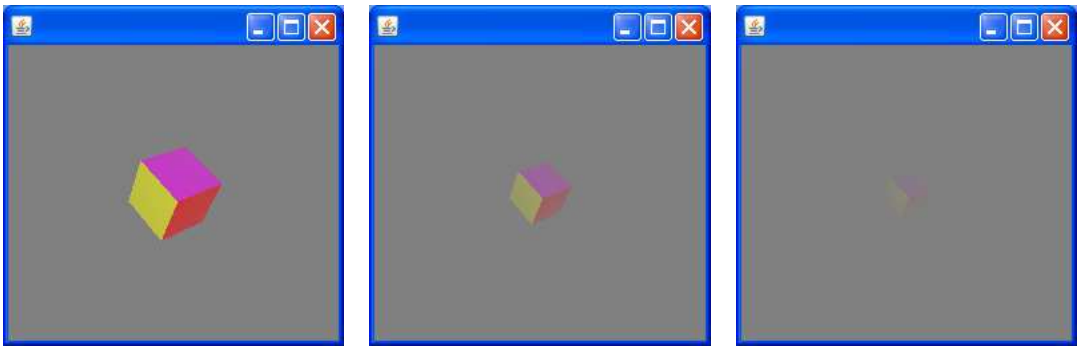
It should be noted that the influencing bounds for a `Fog` node must be set, otherwise the `Fog` node will be considered to be disabled.

The following example demonstrates how a `Fog` node can be used to obscure an object in a scene.

```
0  import javax.media.j3d.*;
   import javax.vecmath.*;

   import com.sun.j3d.utils.behaviors.mouse.*;
   import com.sun.j3d.utils.geometry.*;
5
   public class LinearFogExample extends BasicScene{

     public static void main(String args[]){new LinearFogExample();}

10   public BranchGroup createContentBranch()
     {
       BranchGroup root = new BranchGroup();

       // Define a backgound with a constant mid grey colour
15     Background background = new Background(new Color3f(0.5f, 0.5f, 0.5f));
       background.setApplicationBounds(new BoundingSphere(new Point3d(),
           Double.MAX_VALUE));
       root.addChild(background);

20     // Define a linear fog node with the same colour as the background
       LinearFog linearFog = new LinearFog(new Color3f(0.5f, 0.5f, 0.5f), 1.0, 10.0);
       linearFog.setInfluencingBounds(new BoundingSphere(new Point3d(),
           Double.MAX_VALUE));
```

```
        root.addChild(linearFog);

        TransformGroup tg = new TransformGroup();
        tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        root.addChild(tg);

        // Include support for mouse zoom
        MouseZoom zoom = new MouseZoom();
        zoom.setTransformGroup(tg);
        tg.addChild(zoom);
        zoom.setSchedulingBounds(new BoundingSphere(new Point3d(),
            Double.MAX_VALUE));

        // Include support for mouse rotate
        MouseRotate rotate = new MouseRotate();
        rotate.setTransformGroup(tg);
        tg.addChild(rotate);
        rotate.setSchedulingBounds(new BoundingSphere(new Point3d(),
            Double.MAX_VALUE));

        ColorCube colorCube = new ColorCube(0.4f);
        tg.addChild(colorCube);

        return root;
    }
}
```

This example begins by creating a background with a constant colour of mid grey (0.5, 0.5, 0.5). A `LinearFog` environment node of the same colour is then created and added to the root of the scene. The front distance for the `LinearFog` node is 1 metre and the back distance is 10 metres. This means that objects less than one metre away from the viewer are not affected by the fog and objects greater than 10 metres away from the viewer are completely obscured by the fog. `MouseZoom` and `MouseRotate` behaviours are added to the scene in order to demonstrate the effects of the fog. Examples of the types of renderings that are obtained when this program is executed are illustrated in Figure 2.45.



(a) (b) (c)

Figure 2.45: Examples of a `ColorCube` viewed from different distances in the presence of `LinearFog`.

2.7.12 Behaviours

Behaviours are nodes that makes changes to the scene graph in response to events, such as user input or the passing of time. In other words, behaviours are a general event handling mechanism for Java 3D. A behaviour indicates interest in a set of events called the behaviour's wakeup criterion. When an event occurs that matches the criterion, Java 3D calls the behaviour to process the event. A bounding region must be specified for all behaviours so that they are enabled. The bounding region is set using the following method of the `Behaviour` class:

- `void setSchedulingBounds(Bounds region)`
  Sets the bounding region for this `Behaviour` object to the specified value.

The remainder of this section discusses behaviours that respond to mouse event, behaviours that render an shape at different levels of detail, behaviours that cause a shape to always face the viewer and interpolators that change the property of a node over time.

2.7.12.1 Mouse Behaviours

The location, orientation and scale of a single scene graph node or a group of scene graph nodes can be controlled using a `TransformGroup` object. The transformation associated with the `TransformGroup` is specified using a `Transform3D` object.

Alternatively a subclass of the abstract `MouseBehavior` class can be used to update the transformation associated with a `TransformGroup` object. The subclasses of `MouseBehavior` are:

- `MouseRotate` - lets the user control the rotational component of the transform associated with a `TransformGroup` object using the mouse.
  - Pressing the left mouse button causes the mouse events to be passed to the `MouseRotate` behaviour.
- `MouseTranslate` - lets the user control the translational component of the transform associated with a `TransformGroup` object using the mouse.
  - Pressing the right mouse button causes the mouse events to be passed to the `MouseTranslate` behaviour.
- `MouseWheelZoom` - lets the user control the scale component of the transform associated with a `TransformGroup` object using the mouse wheel.
  - Rotation of the wheel away from the users causes the scale to decrease and rotation of the wheel towards the user cases the scale to increase.
- `MouseZoom` - lets the user control the scale component of the transform associated with a `TransformGroup` object using the mouse.
  - Pressing the centre mouse button causes the mouse events to be passed to the `MouseZoom` behaviour.
  - **Note:** In cases where the mouse has only two buttons, pressing the left mouse button and the ALT key simultaneously causes the mouse events to be passed to the `MouseZoom` behaviour.

These classes are all defined in the `com.sun.j3d.utils.behaviors.mouse` package. All of the four mouse behaviours operate in the same way. The following discussion describes how the `MouseRotate` behaviour operates.

An new instance of a `MouseRotate` object can be created using one of the following constructors:

- `MouseRotate()`
  Create a default `MouseRotate` behaviour that captures mouse events from the `Canvas3D` object associated with the scene graph.
- `MouseRotate(Component c)`
  Creates a `MouseRotate` behaviour that captures mouse events from the specified `Component`.
- `MouseRotate(TransformGroup tg)`
  Creates a `MouseRotate` behaviour that captures mouse events from the `Canvas3D` object associated with the scene graph and updates the specified `TransformGroup` object.

If the `TransformGroup` object that is to be modified is not specified in the constructor for a `MouseBehavior` object then it can be specified using the following method:

- `void setTransformGroup(TransformGroup tg)`
  Sets the `TransformGroup` object updated by the `MouseRotate` behaviour to the specified `TransformGroup` object.

The `MouseRotate` behaviour is an environment node and must have a bounding region associated with it so that it is enabled. A suitably constructed bounding region object can be associated with a `MouseRotate` behaviour using the following method:

- `void setSchedulingBounds(Bounds region)`
  Sets the bounding region of the `MouseRotate` behaviour to the specified region.

It is possible to set the rate of rotation caused by the mouse movements. This can be achieved using the following method:

- `void setFactor(double factor)`
  Sets the x-axis and y-axis movement multiplier to the specified value.

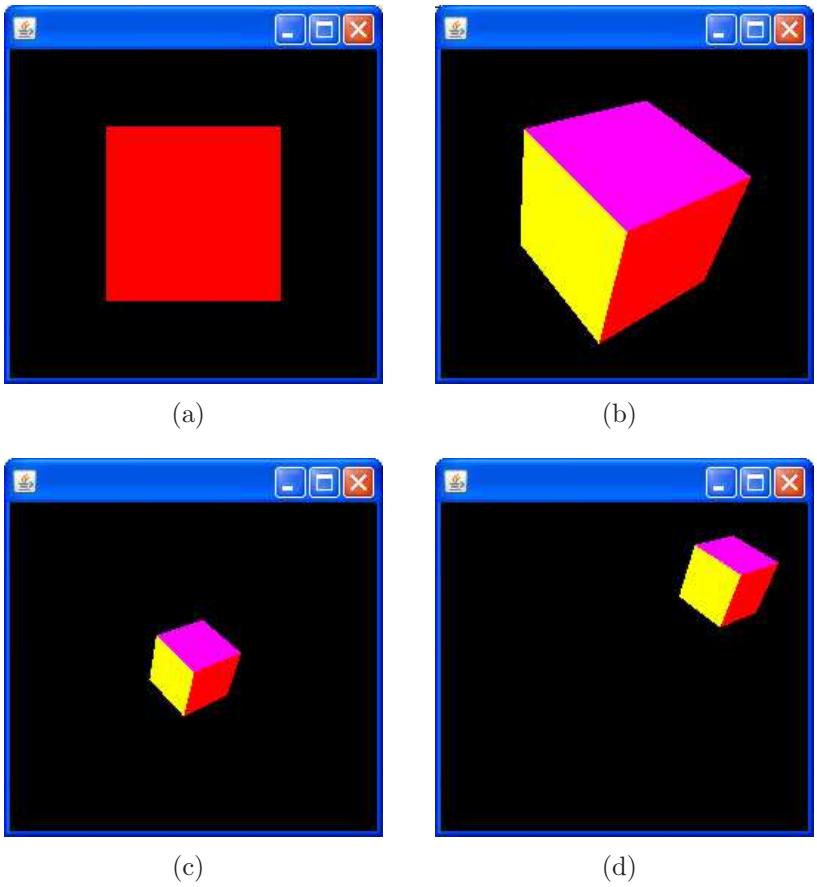
Once a `MouseRotate` behaviour has been constructed and configured it must be attached to the scene graph so that it is active. A mouse behaviour is usually attached to the `TransformGroup` that it updates. This is achieve by calling the `addChild()` method of the `TransformGroup` object.

The following example demonstrates how all four mouse behaviours can be used in conjunction with a single `TransformGroup` object.

```
0
import javax.media.j3d.*;
import javax.vecmath.Point3d;

import com.sun.j3d.utils.behaviors.mouse.*;
5 import com.sun.j3d.utils.geometry.*;

public class MouseBehaviourExample extends BasicScene
{
  public static void main(String args[]){new MouseBehaviourExample();}
10
  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

15  // Create the TransformGroup that is to be updated
    TransformGroup tg = new TransformGroup();
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg);

20  // Approximate infinite bounds
    BoundingSphere infiniteBounds = new BoundingSphere(new Point3d(),
        Double.POSITIVE_INFINITY);

    // Create the MouseRotate behaviour
25  MouseRotate rotate = new MouseRotate(tg);
    rotate.setSchedulingBounds(infiniteBounds);
    tg.addChild(rotate);

    // Create the MouseZoom behaviour
30  MouseZoom zoom = new MouseZoom(tg);
    zoom.setSchedulingBounds(infiniteBounds);
    tg.addChild(zoom);

    // Create the MouseWheelZoom behaviour
35  MouseWheelZoom wheelZoom = new MouseWheelZoom(tg);
    wheelZoom.setSchedulingBounds(infiniteBounds);
    tg.addChild(wheelZoom);

    // Create the MouseTranslate behaviour
40  MouseTranslate translate = new MouseTranslate(tg);
    translate.setSchedulingBounds(infiniteBounds);
    tg.addChild(translate);

    ColorCube colorCube = new ColorCube(0.4);
45  tg.addChild(colorCube);

    return root;
  }
}
```

This example begins by creating a `TransformGroup` object that is attached to the

root of the scene graph and allows its associated transform to be written after the scene graph has gone live. Then the four mouse behaviours are constructed, configured and added to the `TransformGroup`. Finally, a `ColorCube` object with sides of 80 cm is added to the `TransformGroup`. The scale, orientation and locations of this `ColorCube` will be controlled by different mouse events. Examples of the various outputs of this program are illustrated in Figure 2.46



(a) (b)

(c) (d)

Figure 2.46: Examples of a shape (a) that was rotated (b), scaled (c) and translated (d) using different mouse behaviours.

2.7.12.2 Level of Detail

A level of detail behaviour is used to control the level of detail of a shape based on its distance from the viewer. This allows a high resolution version of the shape to be used when the viewer is close to the shape. Then, as the viewer moves away from the shape lower resolution version of the shape can be used.

Level of detail control is provided by the `DistanceLOD` class which is a subclass of the abstract `LOD` class. Both of these classes are defined in the `javax.media.j3d` package. The `DistanceLOD` behaviour controls a `Switch` group to control which of its children are rendered based on the distance between the `DistanceLOD` node and the viewer.

An array of n monotonically increasing distance values must be specified, such that $distances[0]$ is associated with the highest level of detail and $distances[n-1]$ is

associated with the lowest level of detail. The index of the child of the `Switch` node that is rendered is based on the distance between the `DistanceLOD` and the viewer d is:

- $0, if d \leq distances[0]$
- $i, if distances[i-1] < d \leq distances[i]$
- $n, if d > distance[n-1]$

A `DistanceLOD` behaviour is created using one of the following constructors:

- `DistanceLOD(float[] distances)`
  Creates a `DistanceLOD` object with the specified list of distances that is positioned at the origin.
- `DistanceLOD(float[] distances, Point3f position)`
  Creates a `DistanceLOD` object with the specified list of distances and position.

One of more `Switch` nodes can be associated with the `DistanceLOD` behaviour using the following method:

- `void addSwitch(Switch switch)`
  Appends the specified `Switch` node to the list of `Switch` nodes maintained by this `DistanceLOD` object.
- `void insertSwitch(Switch switch, int index)`
  Inserts the specified `Switch` node into the list of `Switch` nodes maintained by this `DistanceLOD` object at the specified index.
- `void setSwitch(Switch switch, int index)`
  Sets the `Switch` node in the list of `Switch` nodes maintained by this `DistanceLOD` object at the specified index to the specified value.

The following two points should be noted regarding the usage of a `DistanceLOD` behaviour:

1. The `ALLOW_SWITCH_WRITE` capability must be set for the `Switch` group. Otherwise the child mask cannot be updated after the scene graph has gone live.
2. A bounding region must be specified for the `DistanceLOD` behaviour. Otherwise it will be disabled and none of the children of the associated `Switch` node will be displayed.

The following program demonstrates how a `DistanceLOD` behaviour can be used to render different version of a shape based on the distance between the `DistanceLOD` node and the viewer.

```
0
  import javax.media.j3d.*;
  import javax.vecmath.*;

  import com.sun.j3d.utils.behaviors.mouse.*;
5 import com.sun.j3d.utils.geometry.*;
```

```
public class DistanceLODExample extends BasicScene
{
  public static void main(String args[]){new DistanceLODExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // Create the TransformGroup that is to be updated
    TransformGroup tg = new TransformGroup();
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg);

    // Approximate infinite bounds
    BoundingSphere infiniteBounds = new BoundingSphere(new Point3d(),
        Double.POSITIVE_INFINITY);

    // Create the MouseZoom behaviour
    MouseZoom zoom = new MouseZoom(tg);
    zoom.setSchedulingBounds(infiniteBounds);
    tg.addChild(zoom);

    // Create a Switch and enable write operations
    Switch sg = new Switch();
    sg.setCapability(Switch.ALLOW_SWITCH_WRITE);

    // Create a red sphere
    Appearance appRed = new Appearance();
    ColoringAttributes caRed = new ColoringAttributes(
        new Color3f(1.0f, 0.0f, 0.0f),
        ColoringAttributes.SHADE_FLAT);
    appRed.setColoringAttributes(caRed);
    Sphere sphereRed = new Sphere(0.4f, appRed);
    sg.addChild(sphereRed);

    // Create a green sphere
    Appearance appGreen = new Appearance();
    ColoringAttributes caGreen = new ColoringAttributes(
        new Color3f(0.0f, 1.0f, 0.0f),
        ColoringAttributes.SHADE_FLAT);
    appGreen.setColoringAttributes(caGreen);
    Sphere sphereGreen = new Sphere(0.4f, appGreen);
    sg.addChild(sphereGreen);

    // Create a blue sphere
    Appearance appBlue = new Appearance();
    ColoringAttributes caBlue = new ColoringAttributes(
        new Color3f(0.0f, 0.0f, 1.0f),
        ColoringAttributes.SHADE_FLAT);
    appBlue.setColoringAttributes(caBlue);
    Sphere sphereBlue = new Sphere(0.4f, appBlue);
    sg.addChild(sphereBlue);
```

```
60      // Create DistanceLOD behaviour
        float [] distances = {4.0f, 8.0f};
        DistanceLOD lod = new DistanceLOD(distances);
        lod.setSchedulingBounds(infiniteBounds);
        lod.addSwitch(sg);
65
        tg.addChild(sg);
        tg.addChild(lod);

        return root;
70    }
}
```

This program begins by creating a `TransformGroup` object that allows its associated transform to be written to after the scene graph has gone live. A `MouseZoom` behaviour is then attached to this `TransformGroup`. Three shapes are subsequently attached to a `Switch` node which is in turn attached to the `TransformGroup`, these are: red, green and blue spheres all 40 cm in diameter. A `DistanceLOD` behaviour is then created, associated with the `Switch` group and added to the `TransformGroup`. This causes:

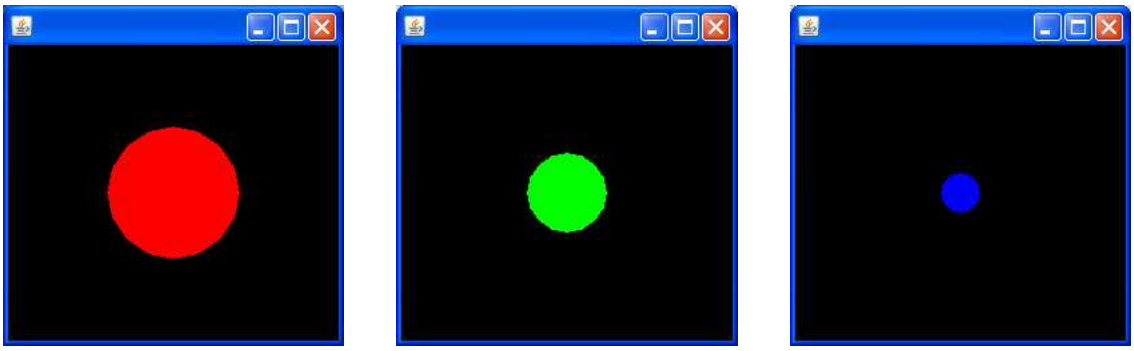
- This first child of the `Switch` group (i.e. the red sphere) to be displayed if the distance between the viewer and the `DistanceLOD` is ≤ 4 metres.
- The second child of the `Switch` group (i.e. the green sphere) to be displayed if the distance between the viewer and the `DistanceLOD` is > 4 metres but ≤ 8 metres.
- The third child of the `Switch` group (i.e. the blue sphere) to be displayed if the distance between the viewer and the `DistanceLOD` is > 8 metres.

It should be noted that in this example different coloured shapes were used rather than different resolution shapes. This is done in order to highlight the operation of the `DistanceLOD` behaviour. Examples of the output generated by this program are illustrated in Figure 2.47.

2.7.12.3 `Billboard`

The `Billboard` behaviour node operates on a `TransformGroup` node to cause the local +z axis of the `TransformGroup` to point at the viewer's eye position. This is done regardless of the transforms above the specified `TransformGroup` node in the scene graph. Two alignment modes are supported by the `Billboard` behaviour, these are:

- `ROTATE_ABOUT_AXIS`
  Causes the associated `TransformGroup` to rotate about the specified axis.
- `ROTATE_ABOUNT_POINT`
  Causes the associated `TransformGroup` to rotate about the specified point.



(a) (b) (c)

Figure 2.47: A `Switch` group rendering a different child based on the distance between a `DistanceLOD` behaviour and the viewer. A red sphere is rendered when the viewer is ≤ 4 metres away (a). A green sphere is rendered when the viewer is > 4 metres away but ≤ 8 metres away (b). Finally a blue sphere is rendered if the viewer is > 8 metres away.

`Billboard` behaviours are ideal for drawing screen aligned-text or for drawing roughly symmetrical objects. A typical use might consist of a quadrilateral that contains a tree structure. A `Billboard` behaviour can be created using one of the following constructors:

- `Billboard(TransformGroup tg, int mode, Point3f point)`
  Creates a `Billboard` behaviour with the specified rotation point and mode that operates on the specified `TransformGroup`.
- `Billboard(TransformGroup tg, int mode, Vector3f axis)`
  Creates a `Billboard` behaviour with the specified axis and mode that operates on the specified `TransformGroup`.

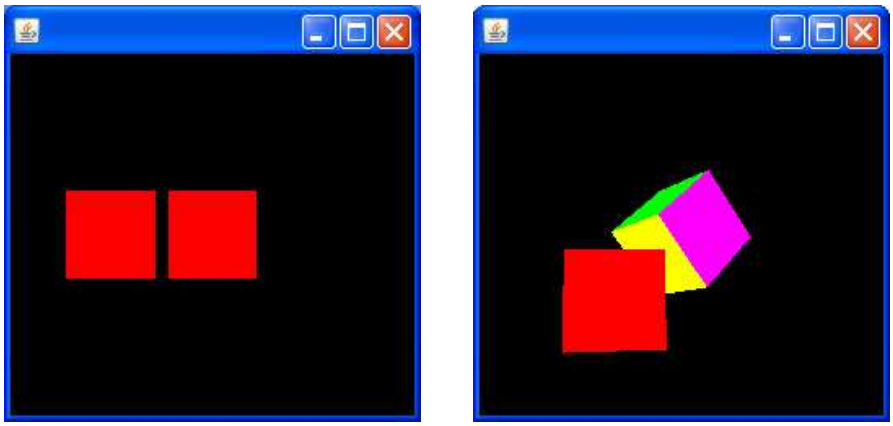
It should be noted that the `OrientatedShape3D` node provides the same kind of functionality as the `Billboard` behaviour, except that only a single `Shape3D` object is affected. `OrientatedShape3D` is generally faster than `Billboard` and should be used where possible.

The following example demonstrates how a `Billboard` behaviour can be used.

```
0
import javax.media.j3d.*;
import javax.vecmath.*;

import com.sun.j3d.utils.behaviors.mouse.*;
5 import com.sun.j3d.utils.geometry.*;

public class BillboardExample extends BasicScene
{
  public static void main(String args[]){new BillboardExample();}
10
  public BranchGroup createContentBranch()
  {
```

```
    BranchGroup root = new BranchGroup();

    // Create the TransformGroup for the MouseRotate
    TransformGroup tg1 = new TransformGroup();
    tg1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg1);

    tg1.addChild(new ColorCube(0.2));

    // Approximate infinite bounds
    BoundingSphere infiniteBounds = new BoundingSphere(new Point3d(),
        Double.POSITIVE_INFINITY);

    // Create the MouseRotate behaviour
    MouseRotate rotate = new MouseRotate(tg1);
    rotate.setSchedulingBounds(infiniteBounds);
    tg1.addChild(rotate);

    // Create the TransformGroup for the translation
    Transform3D trans = new Transform3D();
    trans.setTranslation(new Vector3d(-0.5f, 0.0f, 0.0f));
    TransformGroup tg2 = new TransformGroup(trans);
    tg2.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg1.addChild(tg2);

    // Create the TransformGroup for use with the Billboard
    TransformGroup tg3 = new TransformGroup();
    tg3.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg2.addChild(tg3);

    tg3.addChild(new ColorCube(0.2));

    Billboard billboard = new Billboard(tg3,
        Billboard.ROTATE_ABOUT_POINT,
        new Point3f(0.0f, 0.0f, 0.0f));
    billboard.setSchedulingBounds(infiniteBounds);
    tg3.addChild(billboard);

    return root;
  }
}
```

The program creates a hierarchy that consists of three `TransformGroup` objects. The first `TransformGroup` is associated with a `MouseRotate` behaviour and has a `ColorCube` child and a `TransformGroup` child. The second `TransformGroup` is associated with a translation 50 cm left of the origin and has a `TransformGroup` child. This final `TransformGroup` has a `ColorCube` child and is associated with a `Billboard` behaviour so that its child is always orientated towards the viewer. The types of output generated when this program is executed are illustrated in Figure 2.48.



(a) (b)

Figure 2.48: The initial output of the `BillboardExample` program (a) and an example of the output after the scene has been rotated slightly. Note that the `ColorCube` that was originally on the left is still facing towards the viewer.

2.7.12.4 Interpolators

Java 3D provides support for a range of behaviours that implement some type of interpolation. Interpolators are used to change an attribute of a node over time. The core interpolation functionality is defined in the abstract `Interpolator` base class. The types of interpolation that are defined by the subclasses of this class include:

- `ColorInterpolator` - This class defines a behaviour that modifies the ambient, emissive, diffuse, or specular colour of its target `Material` object by linearly interpolating between a pair of specified colours.
- `TransparencyInterpolator` - This class defines a behaviour that modifies the transparency of its target `TransparencyAttributes` object by linearly interpolating between a pair of specified transparency values.
- `SwitchValueInterpolator` - This class defines a behaviour that modifies the selected child of the target `Switch` node by linearly interpolating between a pair of specified child index values.
- `TransformInterpolator` - This is an abstract class that extends `Interpolator` to provide common methods used by various transform related interpolator subclasses.

`PositionInterpolator`

One example of a subclass of `TransformInterpolator` is `PositionInterpolator`. This class defines a behaviour that modifies the translation component of its target `TransformGroup` by linearly interpolating between a pair of specified positions. The interpolated position is used to generate a translation transform along the local X-axis of this interpolator. An instance of a `PositionInterpolator` can be created using one of the following constructors:

- `PositionInterpolator(Alpha alpha, TransformGroup target)`
  Constructs a position interpolator with a specified target, an axis-of-translation

set to the identity transformation, a start position of 0.0f and an end position of 1.0f.

- `PositionInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfTransform, float start, float finish)`
  Constructs a position interpolator with a specified target, a specified axis-of-translation, a specified start position and a specified end position.

Both of these constructors also require a `Alpha` object to be passed as an argument. An `Alpha` object is a node component that provides common methods for converting a time value into an alpha value (i.e. a value in the range 0 to 1). Some of the attributes defined by the `Alpha` alpha class are as follows:

- **Loop count -** This is the number of times to run this `Alpha`. A value of -1 indicates that the `Alpha` loops indefinitely.
- **Trigger time -** This is the time in milliseconds since the start time that this object first triggers. If the start time plus the trigger time is $\geq$ the current time, then the `Alpha` starts running.
- **Increasing alpha duration -** This is the period of time during which the `Alpha` object transitions from zero to one.

An instance of an `Alpha` object can be created using the following constructor:

- `Alpha(int loopCount, long increasingAlphaDuration)`
  Creates an `Alpha` object that will loop for the specified number of durations where each loop lasts for the specified duration in milliseconds.

The attributes of the constructed `Alpha` object can then be access using the relevant accessor methods, for example the trigger time can be set or retrieved using the following methods:

- `void setTriggerTime(long time)`
  Sets the trigger time to the specified value, e.g. a value of 4000 would cause the `Alpha` to start iterating four seconds after the application was launched.
- `long getTriggerTime()`
  Retreves the trigger time in milliseconds associated with this alpha object.

The following example demonstrates how a `PositionInterpolator` can be used to move an object from one location to another, a specified number of times, over a specified period.

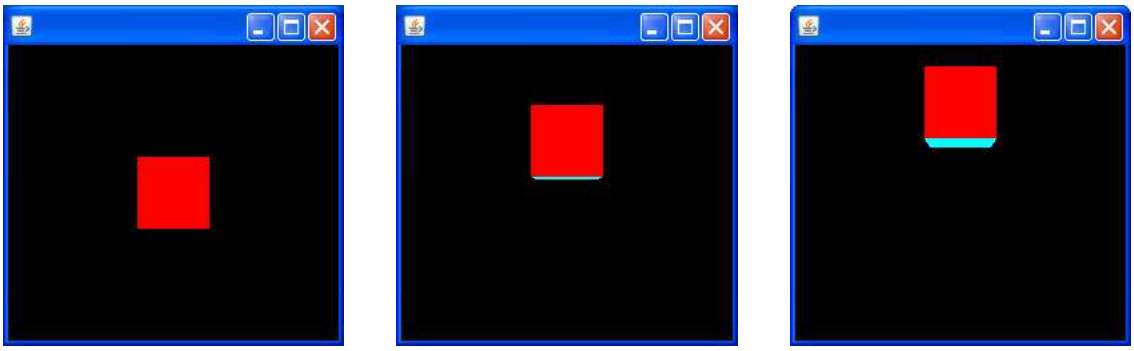
```
0
import javax.media.j3d.*;
import javax.vecmath.*;

import com.sun.j3d.utils.behaviors.mouse.*;
5 import com.sun.j3d.utils.geometry.*;

public class PositionInterpolatorExample extends BasicScene
{
```

```
  public static void main(String args[]){new PositionInterpolatorExample();}

  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    // Create the TransformGroup associated with the PositionInterpolator
    TransformGroup tg = new TransformGroup();
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg);

    // Add a ColorCube child
    tg.addChild(new ColorCube(0.2));

    // Create an alpha that will start after two seconds, loop four times
    // where each loop lasts 1 second
    Alpha alpha = new Alpha(4, 1000);
    alpha.setTriggerTime(2000);

    // A Transform that rotates the x axis onto the y axis
    Transform3D transform = new Transform3D();
    transform.setRotation(new AxisAngle4f(0.0f, 0.0f, 1.0f,
        (float)(Math.PI/2.0)));

    // Create the positional interpolator to move the ColorCube
    // between 0.0 and 0.5 on the y-axis
    PositionInterpolator pi = new PositionInterpolator(alpha,
        tg, transform, 0.0f, 0.5f);
    pi.setSchedulingBounds(new BoundingSphere(new Point3d(),
        Double.POSITIVE_INFINITY));
    tg.addChild(pi);

    return root;
  }
}
```

The program begins by creating a `TransformGroup` object that will ultimately be updated by a `PositionInterpolator` behaviour. A `ColorCube` with sides 40 cm in length is added to the `TransformGroup`. An `Alpha` is then created with a loop count of 4, a loop duration of 1 second and a trigger time 2 seconds after the application starts. A `Transform3D` is then created that transforms the x-axis onto the y-axis. Finally, a `PositionInterpolator` is created that updates the `TransformGroup` and moves the `ColorCube` from 0.0 to 0.5 along the y-axis. Examples of the output renderings obtained from this program are illustrated in Figure 2.49.

Spline Path Interpolator

A B-spline curve is a smooth path that is defined by a series of control points and blending functions. The origin of B-splines relates to industries such as ship building. where a designer was required to draw a life-size curves representing, for example, the cross-section through the hull of a ship.



(a) (b) (c)

Figure 2.49: An example of the output generated by the PositionInterpolator example. The `ColorCube` is initially positioned at the original. Its position is then interpolated along the positive y-axis (b) to y = 0.5 (c).

For small scale drawings draughtsmen would use French curves[2]. They would draw complete curves by putting together segments formed from different parts of different French curves. For full-scale plans this method was completely impractical and the draftsmen would employ long thin strips of metal. These were pushed into the required shape and secured using lead weights called ducks (see Figure 2.50). These ducks are analogous to the control points for a B-spline.

Figure 2.50: An example of how lead weights known as ducks can be used to generate a curved shape from a straight rod.

This is the physical basis for B-splines. The metal shape takes up a shape that minimises its internal strain. In addition, the effect of a duck is local and the shape of the curve is only altered in its vicinity.

A B-spline curve does not pass through its control points. It is a complete piecewise cubic polynomial consisting of any number of curve segments. The B-spline formulation is defined as follows:

[2]French curves are a set of small, flat preformed curve sections

$$Q_i(u) = UB_sP \tag{2.3}$$

Or alternatively in matrix notation:

$$Q_i(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix} \tag{2.4}$$

where Q_i is the ith B-spline segment and P_i is a set of four points in a sequence of control points. The value for u over a single curve segment is $0 \leq u \leq 1$. Using this notation u represents a local parameter, locally varying over the parametric range 0 to 1 to define a single B-spline curve segment.

It is clear that a B-spline curve is a series of $m-2$ curve segments that are labeled $Q_3, Q_4, ..., Q_m$ defines or determined by $m+1$ control points $P_0, P_1, ..., P_m, m \geq 3$. Each curve segment is defined by four control points and each control point influences four and only four curve segments. An example of a B-spline curve is illustrated in Figure 2.51.

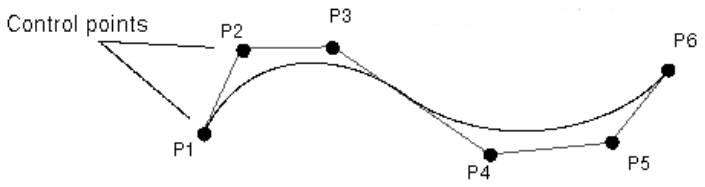


Figure 2.51: An example of a B-spline curve with 6 control points,

Java 3D provides support for a variation on B-splines known as Kochanek-Bartels cubic splines. These types of spline are also known as a TCB splines as they have configurable tension, continuity and bias characteristics. Unlike B-splines, TCB splines do go through the control points.

The `RotPosScaleTCBSplinePathInterpolator` class defines a behaviour that varies the rotational, translational and scale components of its target `TransformGroup` using Kochanek-Bartels cubic spline interpolation to interpolate among a series of key frames using the value generated by a specified `Alpha` object. An object of this class can be constructed as follows:

- `RotPosScaleTCBSplinePathInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfTransform, TCBKeyFrame[] keys)`
  Constructs a new `RotPosScaleTCBSplinePathInterpolator` object that varies the rotation, translation and scale of the transformation associated with the target `TransformGroup`.

The `TCBKeyFrame[]` argument defines a list of key points and their associated attributes. A `TCBKeyFrame` object can be created using the following constructor:

- `TCBKeyFrame(float k, int l, Point3f pos, Quat4f q, Point3f s, float t, float c, float b)`
  Creates a key frame with the specified attributes:

- `k` - Defines the knot value. The first knot must have a value of 0.0. The last knot must have a value of 1.0. An intermediate knot with index k must have a value strictly greater than any knot with index less than k.
- `l` - Indicate whether to use linear (1) or spline (0) interpolation.
- `pos` - The position of the key frame.
- `q` - The rotation at the key frame.
- `s` - The scale at the key frame.
- `t` - The tension at the key frame ($-1.0 < t < 1.0$).
- `c` - The continuity at the key frame ($-1.0 < c < 1.0$).
- `b` - The bias at the key frame ($-1.0 < b < 1.0$).

The follow example demonstrates how a `RotPosScaleTCBSplinePathInterpolator` behaviour can be used to generate a spline interpolated path along a series of control points.

```
0
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.behaviors.interpolators.*;
5
public class TCBSplineExample extends BasicScene
{
  public static void main(String args[]){new TCBSplineExample();}

10  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

    TransformGroup tg = new TransformGroup();
15    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg);


    ColorCube colorCube = new ColorCube(0.1);
20    tg.addChild(colorCube);

    TCBKeyFrame[] keyFrame = new TCBKeyFrame[5];

    // Create the first key frame at (0.5, 0.5)
25    keyFrame[0] = new TCBKeyFrame(0.0f,
        0,
        new Point3f(0.5f,0.5f,0.0f),
        new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
        new Point3f(1.0f, 1.0f, 1.0f),
30        0.0f, 0.0f, 0.0f);

    // Create the second key frame at (-0.5, 0.5)
    keyFrame[1] = new TCBKeyFrame(0.25f,
        0,
```

```
            new Point3f(-0.5f,0.5f,0.0f),
            new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
            new Point3f(1.0f, 1.0f, 1.0f),
            0.0f, 0.0f, 0.0f);


        // Create the thrid key frame at (-0.5, -0.5)
        keyFrame[2] = new TCBKeyFrame(0.50f,
            0,
            new Point3f(-0.5f,-0.5f,0.0f),
            new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
            new Point3f(1.0f, 1.0f, 1.0f),
            0.0f, 0.0f, 0.0f);


        // Create the fourth key frame at (0.5, -0.5)
        keyFrame[3] = new TCBKeyFrame(0.75f,
            0,
            new Point3f(0.5f,-0.5f,0.0f),
            new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
            new Point3f(1.0f, 1.0f, 1.0f),
            0.0f, 0.0f, 0.0f);


        // Create the final key frame at (0.0, 0.0)
        keyFrame[4] = new TCBKeyFrame(1.0f,
            0,
            new Point3f(0.0f,0.0f,0.0f),
            new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
            new Point3f(1.0f, 1.0f, 1.0f),
            0.0f, 0.0f, 0.0f);

        // Create an Alpha that loops indefinitely for
        // a duration of 4 seconds
        Alpha alpha = new Alpha(-1, 4000);

        // Create the TCB spline path interpolator
        RotPosScaleTCBSplinePathInterpolator i =
          new RotPosScaleTCBSplinePathInterpolator(alpha,
              tg, new Transform3D(),keyFrame);
        BoundingSphere s = new BoundingSphere(new Point3d(),
            Double.POSITIVE_INFINITY);
        i.setSchedulingBounds(s);
        tg.addChild(i);

        return root;
    }
}
```

The program begins by creating a `TransformGroup` whose associated transformation will ultimately be controlled by a TCB spline path interpolator. The child of the `TransformGroup` is a `ColorCube` with sides 20 cm in length.

A total of five key frames are defined. Each key frame has a unique position and knot value and uses spline interpolation. None of the key frames affect the rotation or scale of the transform group and the tension, continuity and bias are all set to 0.

A `RotPosScaleTCBSplinePathInterpolator` object is created using an `Alpha` object that loops infinitely with a period of 4 seconds, the `TransformGroup` that is to be modified, the identity transform and the array of key frames. The bounds for the interpolator are set to approximate infinite bounds and the interpolator is added to the scene graph. The path taken by the `ColorCube` under the control of the TCB spline path interpolator is illustrated in Figure 2.52.

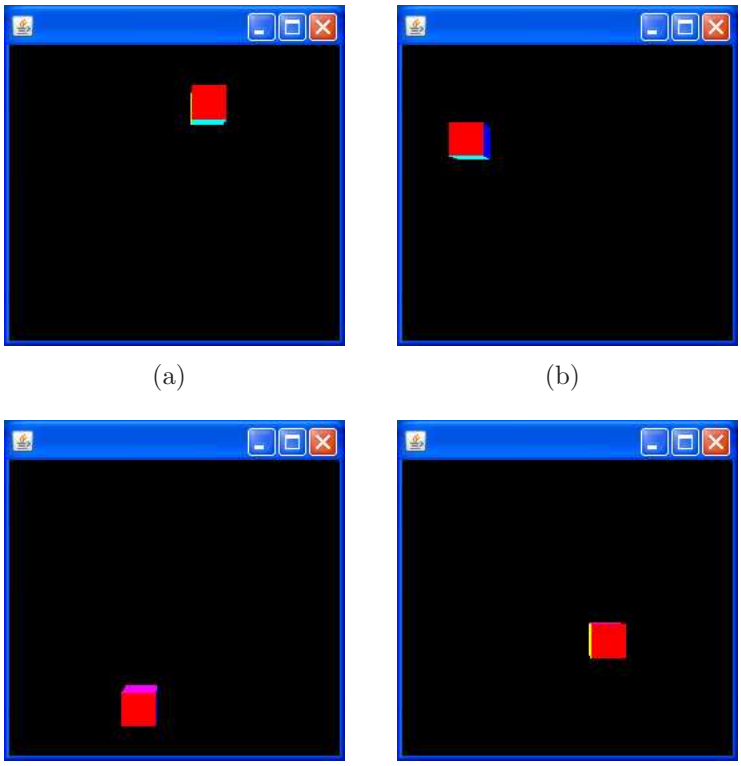


Figure 2.52: A illustration of the path taken by the `ColorCube` under control of the TCB spline path interpolator.

2.7.13 Picking

Picking is essentially the opposite operation to viewing. It enables the selection of a specific shape by projecting 2D screen coordinates into the virtual world and identifying the shape associated with the coordinates.

The `PickCanvas` class is used to turn the mouse coordinates into an area of space or a `PickShape`, that projects from the viewer through the mouse location into the virtual world. The `PickCanvas` class extends a more general `PickTool` class that defines basic picking operations.

When a pick is requested, Java 3D figures out the pickable shapes that intersect with the `PickShape`. These shapes are stored in a list of `PickResult` objects.

A `PickCanvas` object can be created using the following constructor:

- `PickCanvas(Canvas3D canvas, BranchGroup b)`
  Creates a `PickCanvas` object that monitors the specified `Canvas3D` object for mouse events and uses these events to pick objects from the specified `BranchGroup` rooted scene graph.

A range of methods are available for configuring the attributes of a `PickCanvas` object, these include:

- `void setMode(int mode)`
  Sets the picking detail mode for this `PickCanvas` object.
- `void setTolerance(float tolerance)`
  Sets the picking tolerance. Objects within this distance in pixels to the mouse x,y location will be picked. The default tolerance is 2.0 pixels.

If a mouse click occurs on the canvas then the list of shapes that were picked can be obtained using the following method:

- `PickResult[] pickAll()`
  Results an array that represents all of the nodes that intersect with the pick location.

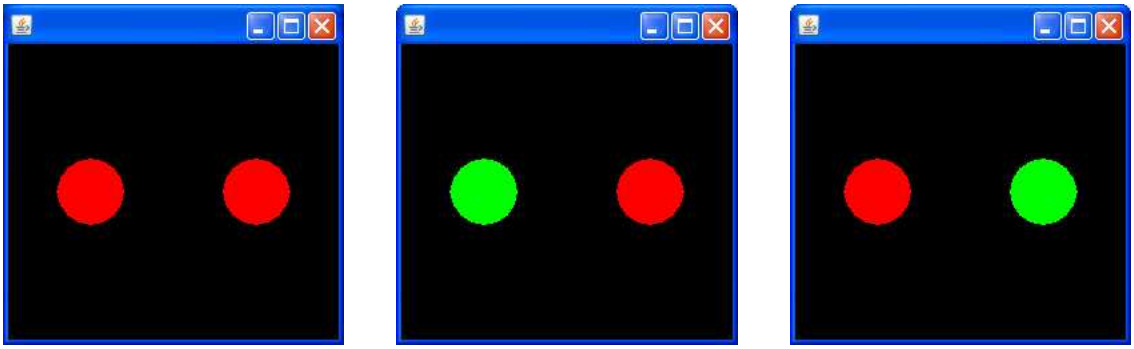
The following example demonstrates how the `PickCanvas` and its associated classes can be used to select objects from a 3D scene using the mouse.

```
0
import java.awt.event.*;
import javax.media.j3d.*;

import com.sun.j3d.utils.picking.*;
5 import com.sun.j3d.utils.geometry.*;
import javax.vecmath.*;

public class PickExample extends BasicScene
{
10   public static void main(String args[]){new PickExample();}

   PickCanvas pickCanvas;
   Appearance redAppearance;
   Appearance greenAppearance;
15   Shape3D rightShape;
   Shape3D leftShape;

   public BranchGroup createContentBranch()
   {
20     BranchGroup root = new BranchGroup();

     // Create and configure the PickCanvas
```

```
    pickCanvas = new PickCanvas(canvas, root);
    pickCanvas.setMode(PickTool.GEOMETRY);
    pickCanvas.setTolerance(4.0f);

    // Create a red appearance
    redAppearance = new Appearance();
    ColoringAttributes red = new ColoringAttributes(
        new Color3f(1.0f, 0.0f, 0.0f),
        ColoringAttributes.SHADE_FLAT);
    redAppearance.setColoringAttributes(red);

    // Create a green appearance
    greenAppearance = new Appearance();
    ColoringAttributes green = new ColoringAttributes(
        new Color3f(0.0f, 1.0f, 0.0f),
        ColoringAttributes.SHADE_FLAT);
    greenAppearance.setColoringAttributes(green);

    // Create a TransformGroup whose children will be located
    // 50 cm to the left of the origin
    Transform3D left = new Transform3D();
    left.setTranslation(new Vector3f(-0.5f, 0.0f, 0.0f));
    TransformGroup leftGroup = new TransformGroup(left);
    root.addChild(leftGroup);

    // Add a red sphere to the left TransformGroup
    Sphere leftSphere = new Sphere(0.2f, redAppearance);
    leftShape = leftSphere.getShape();
    leftShape.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);
    leftGroup.addChild(leftSphere);

    // Create a TransformGroup whose children will be located
    // 50 cm to the right of the origin
    Transform3D right = new Transform3D();
    right.setTranslation(new Vector3f(0.5f, 0.0f, 0.0f));
    TransformGroup rightGroup = new TransformGroup(right);
    root.addChild(rightGroup);

    // Add a red sphere to the right TransformGroup
    Sphere rightSphere = new Sphere(0.2f, redAppearance);
    rightShape = rightSphere.getShape();
    rightShape.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);
    rightGroup.addChild(rightSphere);

    return root;
  }

  public void mouseClicked(MouseEvent me)
  {
    // Set the colour of both spheres to red
    leftShape.setAppearance(redAppearance);
    rightShape.setAppearance(redAppearance);

```

```
    // Get the picked nodes
    pickCanvas.setShapeLocation(me);
    PickResult[] results = pickCanvas.pickAll();

    if(results != null)
      for(int r=0; r<results.length; r++)
      {
        // Set the colour of the picked shape to green
        PickResult result = results[r];
        Shape3D sphere = (Shape3D)result.getObject();
        sphere.setAppearance(greenAppearance);
      }
  }
}
```

This program begins by associating a `PickCanvas` with the main `Canvas3D` of the application. Then two red spheres with radius 20cm are created and positioned 50 cm left and right of the origin. When the mouse is clicked the colour of both of the spheres is set to red. The list of picked nodes is obtained and the colour of each picked shape is set to green. Examples of the renderings obtained when this program is executed are illustrated in Figure 2.53



(a) (b) (c)

Figure 2.53: The output generated by the picking example. Intially both of the spheres are coloured red (a). When the user clicks on the left sphere its colour changes to green (b). Then when the user clicks on the right sphere its colour changes to green and the colour of the left sphere changes back to red (c).

2.8 Summary

This chapter has provided a detailed discussion of the Java 3D API. The concept of scene graphs forms the basis for the organisation of content in a 3D scene. The scene graph can contain group nodes, leaf nodes and node components. Relationships and references can be created to link the various scene graph elements in order to create a coherent structure.

The group nodes in the scene graph allow other nodes to be grouped together. In some cases the group nodes implemented some kind of functionality, for example

conditional rendering of the groups children, or implementing a specified transformation that is applied to all the children of the group.

The main visible content that can be contained in a scene graph is represented by a `Shape3D` object. A `Shape3D` object is essentially a container for an appearance and one or possibly more geometries.

The `Appearance` node component defines a series of attributes that indicate how a shape appears in the rendered scene. These attributes define things like how the shape appears when there is no light, how different types of light affect the appearance of the shape, whether the shape is to be rendered using points, lines or polygons and whether texture mapping is to be used in conjunction with the shape.

The `Geometry` node component essentially defines the structure of the shape. At the most basic level the geometry can be a set of point, lines or polygons. Several different approaches to polygon definition are also available to optimise the way that geometry can be defined. These include fan and strip arrays of triangles. Indexed arrays of vertices can also be used to reduce the amount of repetition that occurs when defining adjoining polygons.

A series of environment nodes are also defined. These nodes are used to determine different aspects of the environment. For example, they can be used to define a background image or fog in order to add realism to a scene. A series of behaviours are also defined. These enable the scene to react to various situations, for example mouse events or the passage of time.

It is evident from the material discussed in this chapter that Java 3D is a comprehensive, fully featured, 3D graphics API that allow the programmer to develop 3D content at a high level, i.e. it allows the developer to focus on what to render and not how it is rendered.

Chapter 3

Surface Extraction

Modern medical imaging modalities typically generate 3D volumetric data that represents the characteristics at each point in the scanned region. Examples of medical imaging modalities that generate volumetric data include:

- Computed Tomography (CT)
- Magnetic Resonance Imaging (MRI)
- Single Photon Emission Computed Tomography (SPECT)
- Positron Emission Tomography (PET)

Examples of images obtained from an abdominal CT study of a patient are illustrated in Figure 3.1.

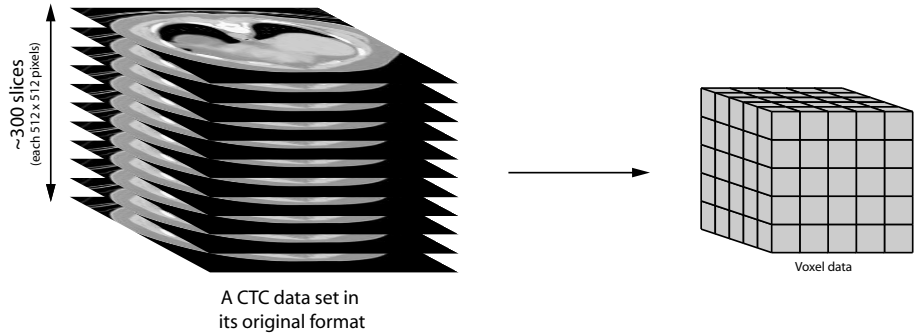


Figure 3.1: A series of slices obtained from an abdominal CT study of a patient. Although the data is obtained as a series of slices it essentially represents a volumetric data set consisting of voxels.

The images in this figure are represented as 2D slices. However, they actually represent thin volumetric regions with a thickness of approximately 1.5 mm. Consequently, the series of images represents a volumetric data set consisting of individual voxels. In the case of CT, the value of a voxel represents the density at a particular point in the scanned region.

If the volumetric data contains continuous isosurfaces, then these surfaces can be explicitly extracted and converted into a polygonal mesh. The resulting mesh can be rendered using the methods outlined in the previous chapter. The techniques used to extract an isosurface from a volumetric data sets is known as the marching cubes algorithm and was originally reported by Lorenson and Cline in 1987.

- Lorensen, W. E. & Cline, H. E. (1987), 'Marching cubes: A high resolution 3D surface construction algorithm', *Computer Graphics* **41**(4), 163-169.

This chapter will provide an overview of the standard marching cubes algorithm. A series of enhancements are then introduced that enhance the operation of the standard algorithm. The marching cubes algorithm is described in relation to a specific area of medical imaging known as virtual endoscopy (specifically virtual colonoscopy).

3.1 The Standard Marching Cubes Algorithm

The marching cubes algorithm (MCA) is used to extract a surface represented by an isosurface value (d_{iso}) from a volumetric data set. The value of d_{iso} is dependent on the surface being extracted and in the case of the colon surface d_{iso} has a value of -800 HU[1].

The MCA begins by thresholding the data set, assigning a 1 to voxels $\geq d_{iso}$ (inside the isosurface) and a 0 to voxels $< d_{iso}$ (outside the isosurface). The isosurface associated with the colonic mucosa cannot be uniquely identified using a simple threshold operation due to the number of gas/soft tissue interfaces that are present in a CTC data set (i.e. those due to the lung bases, the small intestine, the stomach and the exterior of the patient). In this case, segmentation information is used in conjunction with the isosurface value to identify the region associated with the colon surface.

A cubic mask of size $2 \times 2 \times 2$ is then passed through the volume and at each mask location the configuration of the eight underlying voxels is examined and the relevant surface patches are generated. This process is illustrated in Figure 3.2.

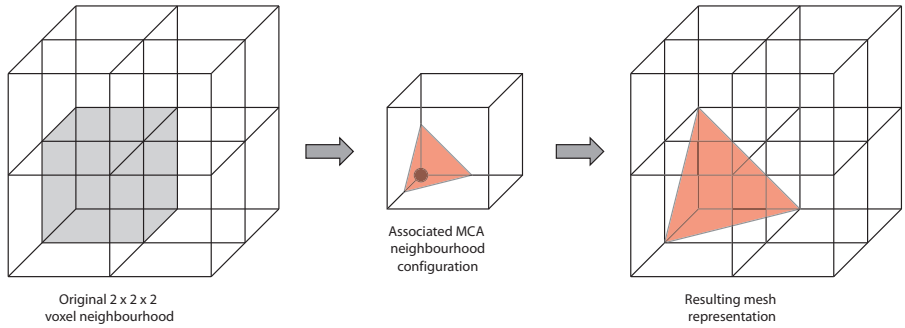


Figure 3.2: An illustration of the marching cubes isosurface generation process. Each $2 \times 2 \times 2$ voxel neighbourhood is examined and the surface patch associated with the neighbourhood configuration is generated and added to the output mesh. In this example, the original eight voxels in the input volume are replaced by a single triangle in the output mesh. Note that the corner sphere in the central image indicates the presence of a voxel located inside the isosurface i.e. the shaded voxel in the original $2 \times 2 \times 2$ volume.

There are 256 (2^8) possible configurations of eight binary voxels and although possible, the task of manually specifying the surface patches associated with each config-

[1]The units of CT attenuation are named after the inventor of the CT technique, G. N. Hounsfield. The CT attenuation of a material is related to its density.

uration is both tedious and prone to error. Lorensen & Cline observed that this task could be greatly simplified by considering all rotations and complementary cases for each configuration.

Using this approach, the number of possible configurations is reduced from 256 down to just 15, as illustrated in Figure 3.3. This significant reduction in the number of configurations makes the task of manually specifying surface patches associated with particular voxel configurations much more manageable and a lot less prone to error.

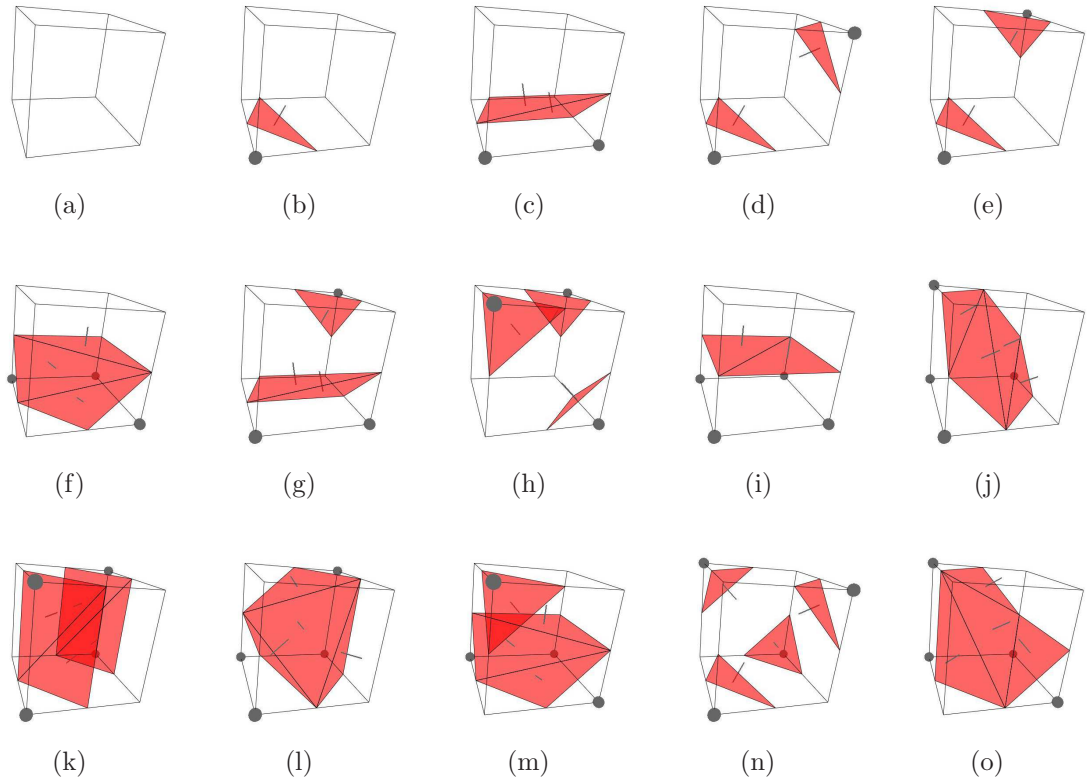
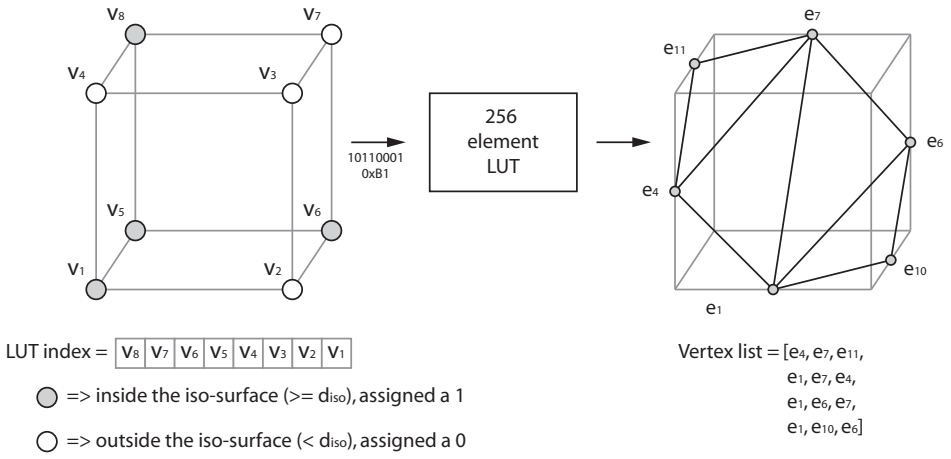


Figure 3.3: The 15 possible configurations of eight binary voxels arranged in a cubic formation. Voxels located inside the isosurface are indicated by cube corners with spheres and voxels outside the isosurface are indicated by cube corners without spheres. The relevant surface patches (highlighted using red) have been inserted as documented in the original marching cubes algorithm specification.

Once the relevant surface patches have been specified for the 15 base configurations, the original 256 configurations are regenerated by applying a series of rotations to the original 15 configurations and their complements. In each case the associated list of vertices (i.e. surface patches) and their complements are also rotated.

The resulting information is used to populate a 256 element lookup table associating voxel configurations with edge lists (i.e. lists of vertices that are positioned relative to the local origin of the cube). The lookup table index is generated based on the voxel configuration. This process is illustrated in Figure 3.4.

By default a vertex will lie midway between the two complementary valued voxels (v_a & v_b) that led to its creation. This is demonstrated in general by all of the vertices associated with each of the 15 cases illustrated in Figure 3.3 and in particular,



(a)

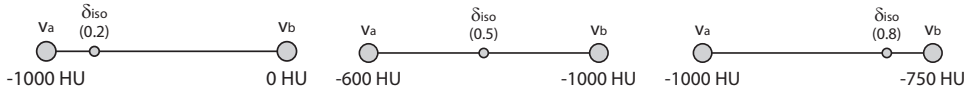
Figure 3.4: A sample voxel configuration where v_1, v_5, v_6 & v_8 are inside the isosurface and all other voxels are outside. A LUT index of 0xB1 is generated which results in the specified edge list. Note that the order of the edges is important as it defines the front face for the associated triangle in Java 3D.

by the vertex resulting from v_1 & v_2 in Figure 3.4.

In order to fit the extracted surface more accurately to the actual isosurface identified by d_{iso}, each vertex must be interpolated between v_a & v_b based on the relationship between their densities d_a & d_b and the isosurface density d_{iso}. This is achieved by calculating the normalised distance (δ_{iso}) between the voxel that is closest to the origin (either v_a or v_b) and the isosurface (see Equation 3.1).

$$\delta_{iso} = \frac{d_{iso} - d_a}{d_b - d_a} \tag{3.1}$$

The value for δ_{iso} represents the intersection location relative to the reference voxel in terms of the intervoxel spacing (see Figure 3.5).



(a) (b) (c)

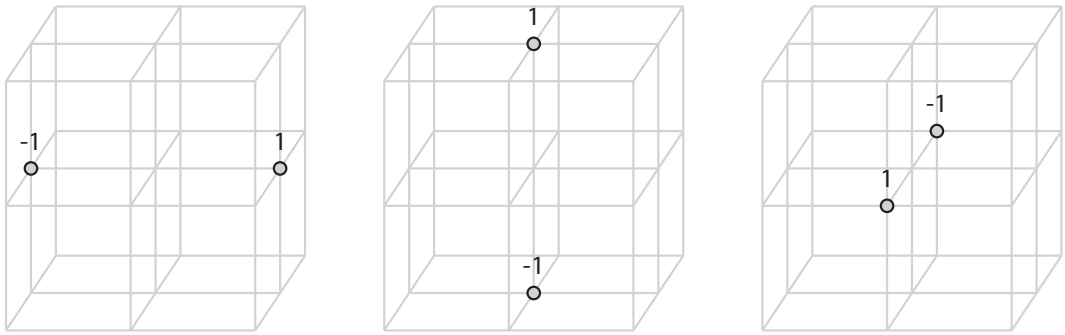
Figure 3.5: The calculation of δ_{iso} at three sample boundaries. In each case, the value of δ_{iso} is calculated using linear interpolation and the density at the point represented by d_{iso} is -800 HU, i.e. the isosurface density d_{iso}.

The closest voxel to the origin is used as the reference point v_a to ensure δ_{iso} has a positive value and to standardise the interpolation process throughout the surface extraction algorithm. The δ_{iso} value can then be used to calculate an interpolated vertex location that is more representative of the actual isosurface. Assuming that v_a is the closest voxel to the origin, the interpolated vertex location (x_i, y_i, z_i) located between v_a and v_b is calculated using:

$$
\begin{aligned}
x_i &= x_a + \delta_{iso}(x_b - x_a) \\
y_i &= y_a + \delta_{iso}(y_b - y_a) \\
z_i &= z_a + \delta_{iso}(z_b - z_a)
\end{aligned} \tag{3.2}
$$

where (x_a, y_a, z_a) and (x_b, y_b, z_b) are the locations of v_a and v_b respectively.

The final stage of the MCA involves generating unit normals for each vertex in the extracted isosurface. The normals are used to facilitate surface shading (Gouraud shading in the case of Java 3D). Normals are calculated at each voxel in the original data set using a 3-D gradient operator. The three masks for the gradient operator proposed by Lorensen and Cline are illustrated in Figure 3.6. The resulting gradient magnitude in each direction is divided by the overall gradient magnitude to yield the unit normal.



(a) (b) (c)

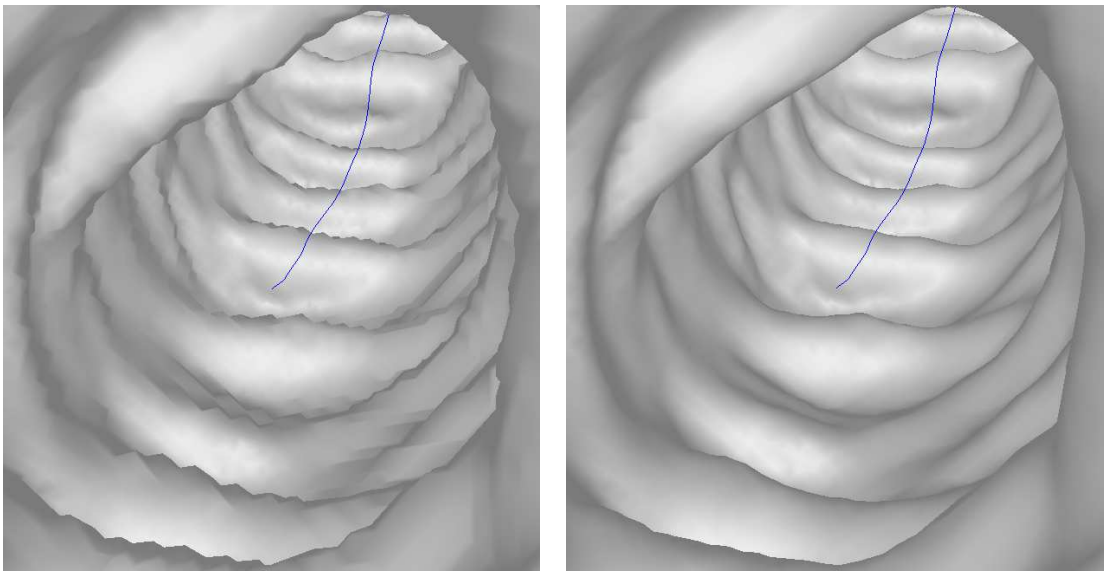
Figure 3.6: The three masks (a, b & c) that are used by (lorensen1987a) to calculate the edge magnitudes in the x, y & z directions respectively. Note that the scaling factors are omitted as the implementation discussed here is intended for use with isotropic data.

The normals of the two voxels (e.g. v_a & v_b as above) associated with a particular vertex must be interpolated in order to give an approximation of the normal value at the vertex location (a_i, b_i, c_i). As with the vertex locations, the normals are interpolated using δ_{iso} from Equation 3.1 as follows:

$$
\begin{aligned}
a_i &= (1 - \delta_{iso})a_a + \delta_{iso}a_b \\
b_i &= (1 - \delta_{iso})b_a + \delta_{iso}b_b \\
c_i &= (1 - \delta_{iso})c_a + \delta_{iso}c_b
\end{aligned} \tag{3.3}
$$

Where (a_a, b_a, c_a) and (a_b, b_b, c_b) are the normals associated with voxels v_a and v_b respectively. The effect of vertex and normal interpolation on the quality of the extracted surface is illustrated in Figure 3.7.

This completes the basic description of the standard MCA. The vertices and their associated normals can now be rendered using conventional 3-D graphics techniques.



(a) (b)

Figure 3.7: An illustration of an isosurface model for the colonic mucosa before (a) and after (b) the use of vertex and normal interpolation.

The standard MCA is only suitable for display purposes. In order to use the MCA in virtual colonoscopy, a number of modifications and enhancements are required. These modifications, which are summarised below, are dealt with in the remainder of this chapter.

1. The standard MCA does not generate airtight surfaces. In certain cases holes may be inadvertently introduced into the generated mesh. The standard MCA must be updated so that such surface discontinuities do not occur.

2. The standard MCA uses a very basic method to generate normals. A more accurate normal generation technique is proposed as normals will be used for surface analysis as well as surface visualisation.

3. The mesh generated by the standard MCA is wasteful of memory as it contains a vast amount of repeated information. A more streamlined alternative is used to reduce the amount of memory required to store vertex coordinates and associated information (e.g. normals, colours, etc.).

4. In the streamlined mesh, mentioned in 3. above, a vertex is no longer represented by an (x, y, z) coordinate. Instead, it is represented by a unique index into a list of common coordinates. A neighbour list is generated for each vertex index that identifies all of the directly connected neighbouring vertices. This is extremely useful for region growing in a triangular mesh and crucial in identifying the surface of the mesh with polyp-like properties.

3.2 Topology Errors (Holes)

Upon visual inspection of the output generated by the standard MCA, it is clear that topology errors (or holes) are present in the generated surface, see Figure 3.10 (a). These holes are due to ambiguous cases resulting from mismatches between the

surface patches of adjoining cubes. An example of an ambiguous case is illustrated in Figure 3.8.

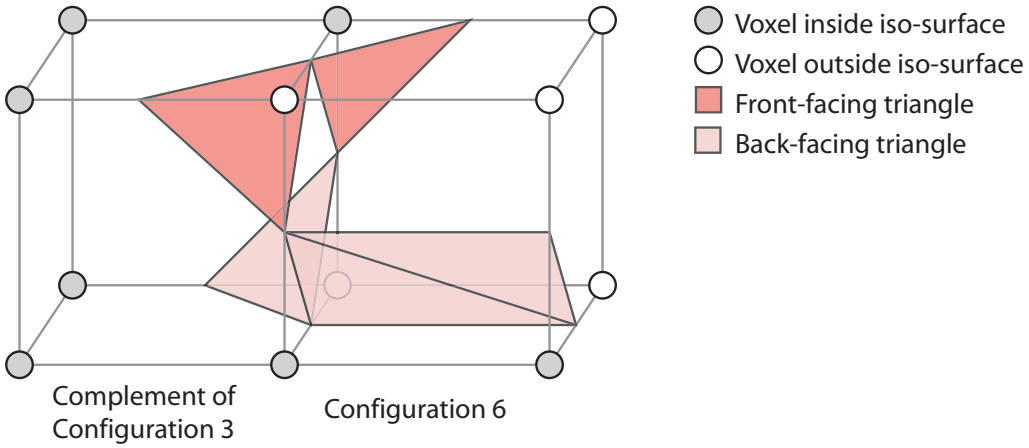


Figure 3.8: An example of the ambiguous case that results when the complement of configuration 3 (Figure 3.3 (d)) occurs next to configuration 6 (Figure 3.3 (g)). A hole is evident at the interface between these two cubes.

The ambiguous cases that result in unwanted holes are a direct result of the use of complementary cases in the standard MCA to reduce the number of core cube configurations that must be specified. By disregarding complementary cases and using only rotation to identify equivalent cube configurations the number of core configurations increases from 15 to 23. The eight additional cases and their associated surface patches are illustrated in Figure 3.9.

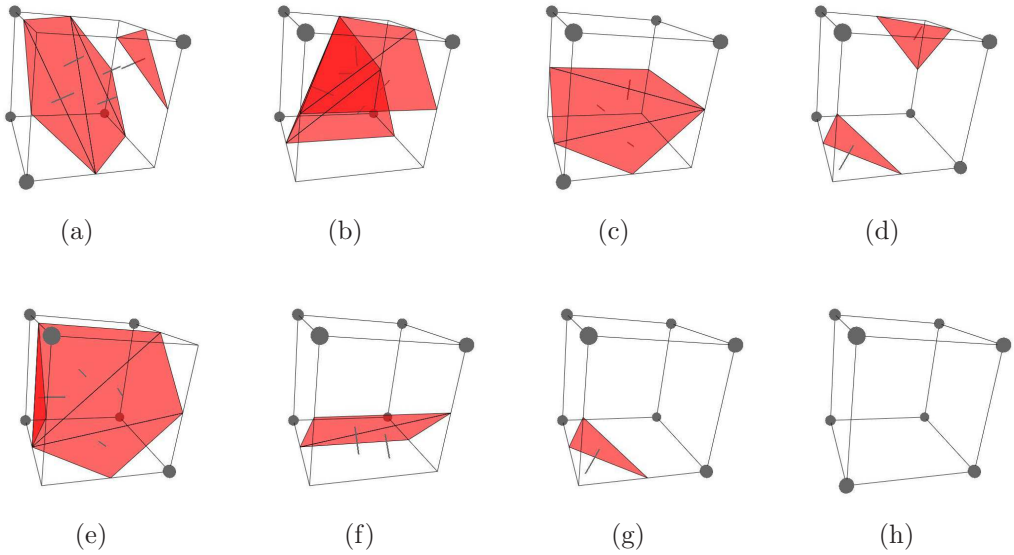
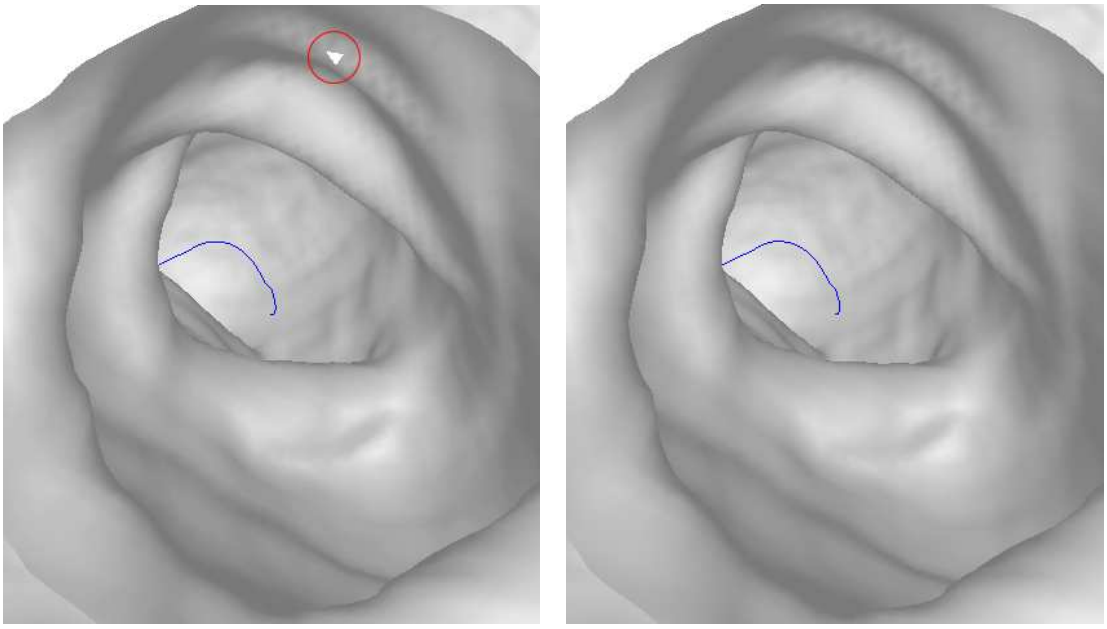


Figure 3.9: The eight additional configurations of eight binary voxels arranged in a cubic formation. These extra configurations remove the need to generate complementary cases and as a result, solve the topology problem associated with the original MCA.

Altering the standard MCA to include these eight extra cases removes the necessity to generate complementary cases and thus results in the generation of airtight surfaces that do not contain unwanted holes. The result of using the revised algorithm is presented in Figure 3.10 (b) where the solitary hole that is evident at the top of Figure 3.10 (a) is no longer present.



(a) (b)

Figure 3.10: An isosurface extracted using the 15 core neighbourhood configurations of the standard marching cubes algorithm (a) and the extended 23 neighbourhood configurations of the modified algorithm (b). Note that the standard approach creates an unwanted hole, indicated by a red circle, whereas the extended approach results in an airtight isosurface.

3.3 Improved Normal Calculation

The edge detector that is used for normal generation by the standard MCA (see Figure 3.6) is very basic and provides only a rough estimation of 3-D edge direction. Vertex normals are usually only used for visualisation purposes i.e. to enable shading so that surfaces generate a more realistic response to lighting. In the enhanced MCA a more accurate estimation of 3-D edge direction is required. The Zucker-Hummel edge operator was selected for this task due to its inherent smoothing effect.

- Zucker, S. W. & Hummel, R. A. (1981), 'A three-dimensional edge operator', *IEEE Transactions on Pattern Analysis and Machine Intelligence* **3**(3), 324-331.

The use of this edge operator gives a more global indication of the normal at each vertex location. The three masks for the 3-D Zucker-Hummel operator are illustrated in Figure 3.11.

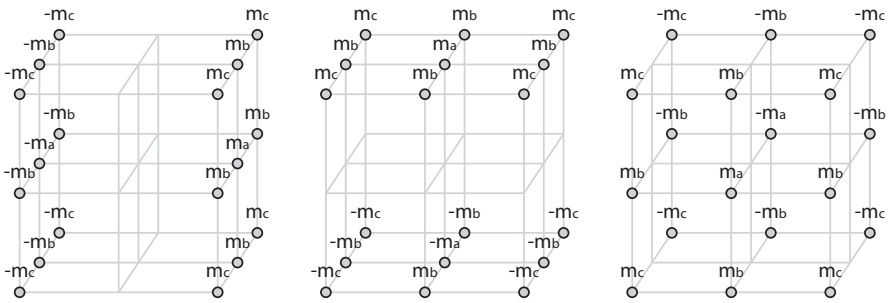


Figure 3.11: The three 26-neighbour masks representing the Zucker-Hummel edge operator where $m_a = 1.0$, $m_b = \frac{\sqrt{2}}{2}$ & $m_c = \frac{\sqrt{3}}{3}$.

3.4 Mesh Representation

The mesh generated by the standard MCA consists of a disjoined set of triangular patches where there is a high degree of vertex repetition. This representation, although suitable for visualisation purposes, is not ideal for analysis. It is also extremely wasteful of memory due to the high degree of vertex repetition.

An alternative mesh representation involves storing each vertex in an array structure where a particular vertex can be present only once. Using this approach, triangles that represent the mesh are specified as indices into the vertex array and not as actual vertex locations. An array of unit normals is generated and populated in the same manner.

This approach to mesh storage has the potential to significantly reduce the amount of memory required to store a fully characterised isosurface representation of the colonic mucosa. A simple example illustrating the difference between the standard mesh and the indexed mesh is presented in Figure 3.12.

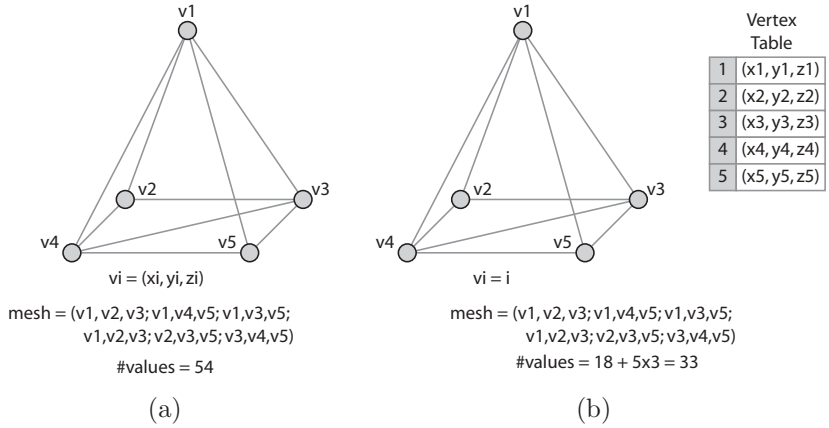


Figure 3.12: An illustration of how indexing can be used to reduce the amount of data required to specify a mesh structure. A pyramid consists of five vertices and six triangles. The unoptimised representation yields 54 values (a) and the optimised alternative yields only 33 (b). Note that in Java the data type for vertex (float) and index (int) both require four bytes of storage space.

The indexing process requires a modification to the standard MCA. As each vertex is encountered it is assigned an index and stored in the vertex list. This index is then used to represent the relevant vertex coordinates (i.e. an Integer primitive (four bytes) is used to reference three floating point primitives (12 bytes)).

If a vertex that was already assigned an index is encountered then that index is used. Conversely, a new index is generated if a new vertex is encountered. Only vertices associated with the same slice or two adjacent slices can be shared. As a result, only two slices need to be resident in memory at any one time. Noting that in the context of the marching cubes algorithm a slice is actually two voxels thick.

3.5 Neighbour Identification

The final modification to the standard MCA involves identifying all of the directly connected neighbouring vertices within the mesh. This step is required to facilitate the automatic detection of polyps from the isosurface representation of the colon surafce.
As each triangle, consisting of vertices e_0, e_1 and e_2, is identified, its vertex relationships are added to an array structure where: e_0 is associated with e_1 & e_2 (i.e. two vertex pairs (e_0, e_1) and (e_0, e_2)), e_1 is associated with e_0 & e_2 and e_2 is associated with e_0 & e_1. A vertex pair is only added to the array structure if this vertex pair is not already present.

An example illustrating the neighbour identification process is illustrated in Figure 3.13. Representing the vertex neighbours in this way reduces the task of neighbourhood identification from an extensive mesh search to a simple table lookup i.e. by specifying the index of one vertex the indices of all of the neighbouring vertices are returned.

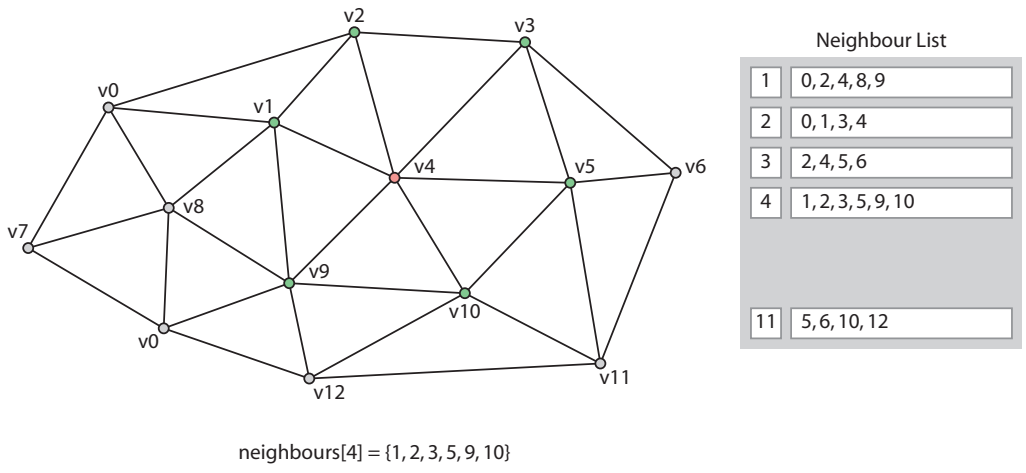


Figure 3.13: The neighbour indexing process: The neighbouring vertices for each mesh vertex are stored in a list to streamline the process of searching for vertex neighbours.

3.6 Summary

This chapter provided a complete description of the marching cubes algorithm. The use of this approach to extract an isosurface enables the indirect rendering of volumetric data using conventional surface rendering techniques, i.e. those described in the previous chapter. The implementation of the marching cubes algorithm described in this chapter deals with some of the issues associated with the standard marching cubes algorithm e.g. fixing topology errors and improving the normal calculation stage. It has also been demonstrated that the use of indexed geometry can be used to reduce the amount of memory required to store the extracted polygonal mesh structure.

Chapter 4

Volume Rendering

The basic idea of volume rendering is that a viewer should be able to perceive a data volume from a rendered projection on the view plane. There are two main types of volume rendering:

- **Direct volume rendering -** involves generating a 2D representation of a 3D volumetric data set by projecting the voxels of the data set on to a 2D view plane.
- **Indirect volume rendering -** involves extracting a polygonal mesh representation of an isosurface of interest from a volumetric data set and rendering it to a 2D view plane using conventional 3D surface rendering techniques.

Volume rendering involves assigning transparency and colour to voxels within the data set. The data set can then be viewed from any angle by projecting the voxels values on to a suitably orientated view plane.

In medical applications this process is sometimes referred to as a computed X-ray as it is analogous to a conventional X-ray. It is possible to generate a computed X-ray from any viewing angle, including angles that may be physically impossible with conventional X-ray equipment.

Currently, the main application of volume rendering is in medical imaging due to the volumetric nature of data acquired using 3D medical imaging modalities. A 2D representation of the volume rendering process is illustrated in Figure 4.1.

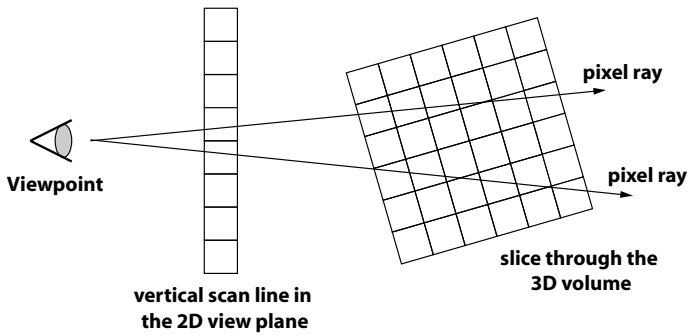


Figure 4.1: A 2D representation of the volume rendering process. Pixel rays are fired from the viewpoint through the volume. The colour of the a pixel in the 2D view plane is defined by the path of its associated pixel ray.

The illustration of the volume rendering process in Figure 4.1 uses perspective projection where all the rays emanate from a single viewpoint. An alternative would be to use parallel projection where a parallel rays is generated from each pixel in the view plane. In either case the volume rendering process is referred to as ray casting.

Note: The term ray casting is used to distinguish the method from ray tracing, which models the path taken by light rays as they interact with optical surfaces.

References:

- Levoy M. (1988) "Display of surfaces from volume data" IEEE Computer Graphics Applications, **8**(3), 29-37.
- Levoy M. (1990) "Efficient ray tracing of volume data" ACM Transactions on Graphics, **9**(3), 263-270.

The following sections describe different approaches to direct volume rendering that can be used to create a 2D representation of a 3D volumetric data set.

4.1 Maximum Intensity Projection

A maximum intensity projection (MIP) is a simple technique for transforming a 3D data sets to a 2D image. It involves identifying the maximum valued pixel along each ray projected from the pixels of the 2D view plane. The resulting maximum value is ultimately assigned to the associated pixel. An illustration of this process is illustrated in Figure 4.2.

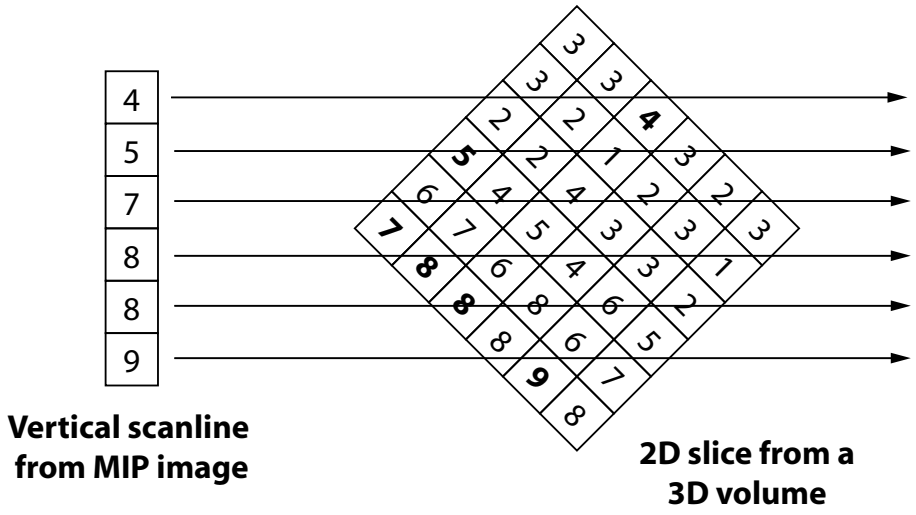


Figure 4.2: An illustration of the maximum intensity projection process. Maximum valued voxels are identified along each pixel ray. The resulting value is assigned to the associated pixel in the MIP image.

Maximum intensity projection is particularly useful in medical imaging applications where contrast enhanced material has been used to highlight a abnormal region of anatomy. One example of this is in a medical imaging technique known as magnetic resonance cholangiopancreatography. This involves the use of magnetic resonance imaging to visualise the biliary tree and pancreatic ducts in a non-invasive manner.

This procedure can be used to determine if gallstones are lodged in any of the ducts surrounding the gallbladder. An illustration of the types of images obtained in an MRCP examination and the resulting MIP representation are illustrated in Figure 4.3.

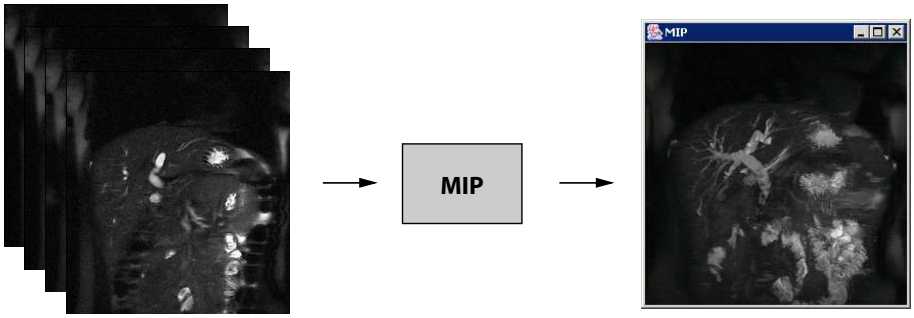


Figure 4.3: An illustration of the MIP process used in conjunction with MRCP data. The maximum value voxels projected in the the MIP image give a much better representation of the imaged anatomy than the individual slices.

It is clear that the structure of the anatomy is much more apparent from the MIP image. The structure of the biliary tree and liver can easily been seen in this image.

4.2 Average Intensity Projection

An average intensity projection is the same a computed X-ray (mentioned earlier). The average value of the voxels encountered by each pixel ray is assigned to the associated pixel in the 2D view plane. The theory behind this process is illustrated in Figure 4.4. It is clear that the approach for average intensity projection is a variation of the approach outlined for maximum intensity projection.

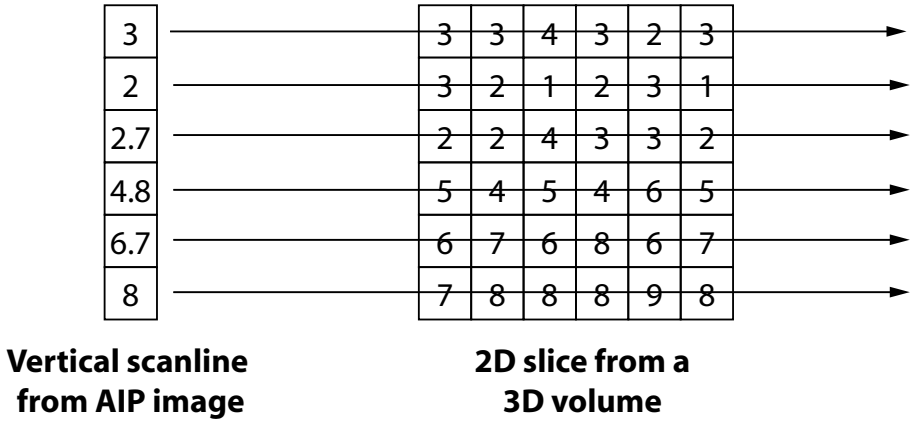
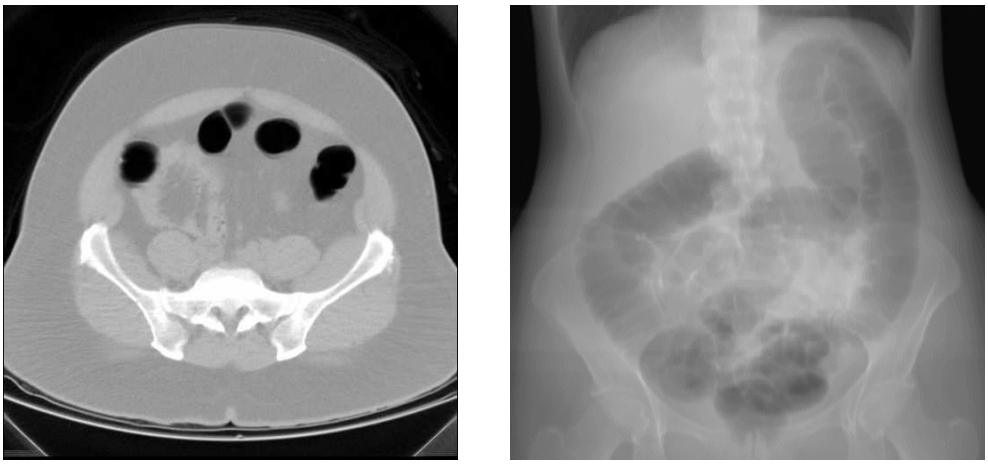


Figure 4.4: An illustration of the average intensity projection process. The average voxel value is identified along each pixel ray. The resulting value is assigned to the associated pixel in the AIP image.

An example of how the average intensity projection process can be applied to a CT data set is illustrated in Figure 4.5. It is clear that the output of this process is very similar to an X-ray image.



(a) (b)

Figure 4.5: An example of the average intensity projection process. (a) A single slice from the data set being projected. (b) The resulting average intensity projection. Note that the AIP image is very similar to an X-ray image.

4.3 Transparent Voxel Rendering

The previous examples dealt with voxels that have a single property, i.e. the voxel colour. A more comprehensive approach to volume rendering involves giving the viewer the facility to see all the data. This involves assigning an additional opacity property to each voxel. The physical analogue to this model is that each voxel is considered to be made of a coloured transparent gel.

Each voxel is assigned a colour, C, and a transparency, α. The colour associated with the material type can be chosen aesthetically. For example, in the case of CT:

- White could be selected for the colour of bone voxels and an opacity could be selected that makes bone appear completely opaque.
- Mid-grey could be selected for soft tissue voxels and an opacity could be selected that makes soft tissue appear semi transparent.
- Black could be selected for air voxels and an opacity could be selected that makes air appear completely transparent.

In addition, it would be possible to use RGB colours to provide a greater degree of realism that can be achieved when limited to using grey scale colours.

Rendering volumes consisting of transparent voxels involves casting a ray into a volume which has been rotated into the desired viewing orientation. Compositing is performed along the ray path to generate the relevant pixel value in the 2D view plane. The compositing process involves accumulating the colours and opacities along the pixel ray.

This approach to volume visualisation involves the following three steps:

1. Classify each voxel in the original data set and assign the desired colour and opacity values.

2. Transform the classified volume data into the viewing direction.

3. For each pixel, cast a ray and find, by composition along the ray, a colour for that pixel.

4.3.1 Compositing Pixels Along a Ray

The most straightforward compositing operation involves the recursive application of the following formula:

$$C_{out} = C_{in}(1 - \alpha) + C\alpha \tag{4.1}$$

where:

- C_{out} is the accumulated colour emerging from the current voxel
- C_{in} is the accumulated colour going into the current voxel
- α is the opacity at the current voxel.
- C is the colour if the current voxel.

This process is illustrated in Figure 4.6

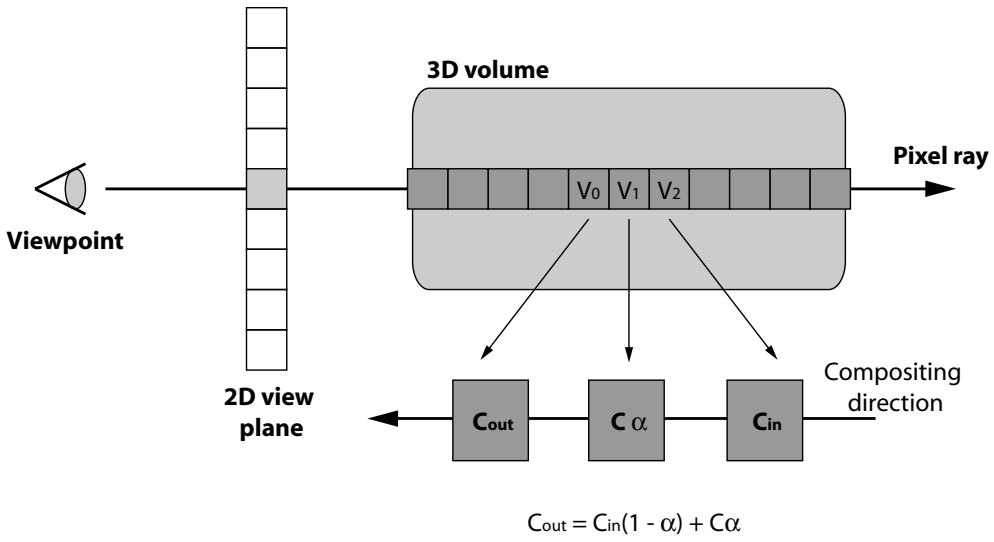


Figure 4.6: An illustration of the ray compositing process.

The direction implied by C_{in} and C_{out} is from back to front with respect to the view plane, i.e. the operation starts at the voxel that is furthest from the view plane.

Using this approach, the colour that comes out of a particular voxel is the product of the colour of the voxel and the opacity of the voxel plus the product of the incoming colour and the transparency of the voxel.

Note: The colour coming into the voxel that is furthest away from the view plane is black.

The purpose of this operation is to make voxels with high opacity values predominate, obscuring voxels that are behind them and being made visible through voxels

in front of them.

Example: Calculate the value of the shaded pixel in the view plane in Figure 4.7 by compositing the voxel values along the pixel ray.

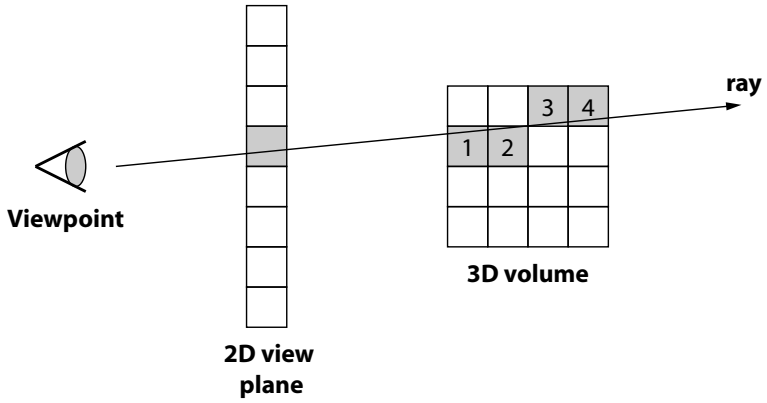


Figure 4.7: An illustration of the ray compositing process.

Given the following grey scale colour values and opacity values for the numbered voxels in the volume.

Voxel #	Colour	Opacity
1	20	0.05
2	25	0.07
3	130	0.55
4	225	0.80

- Voxel 4
  - $C_{in} = 0$, $C = 225$, $\alpha = 0.8$
  - $C_{out} = C_{in}(1 - \alpha) + C\alpha$
  - $C_{out} = 0 \times (1 - 0.80) + 225 \times 0.8$
  - $C_{out} = 180$
- Voxel 3
  - $C_{in} = 180$, $C = 130$, $\alpha = 0.55$
  - $C_{out} = C_{in}(1 - \alpha) + C\alpha$
  - $C_{out} = 180 \times (1 - 0.55) + 130 \times 0.55$
  - $C_{out} = 81 + 71.5 = 152.5$
- Voxel 2
  - $C_{in} = 152.5$, $C = 25$, $\alpha = 0.07$
  - $C_{out} = C_{in}(1 - \alpha) + C\alpha$
  - $C_{out} = 152.5 \times (1 - 0.07) + 25 \times 0.07$

- $C_{out} = 141.825 + 1.75 = 143.575$

- Voxel 1
  - $C_{in} = 144$, $C = 20$, $\alpha = 0.05$
  - $C_{out} = C_{in}(1 - \alpha) + C\alpha$
  - $C_{out} = 144 \times (1 - 0.05) + 20 \times 0.05$
  - $C_{out} = 136.8 + 1 = 137.8$

Consequently, the value of the shaded pixel in the view plane is 138. It is clear that the voxels with the highest opacity values have the greatest influence on the composited pixel value.

4.3.2 Voxel Projection

This is a variant on volume rendering that involves traversing the data set and projecting each voxel onto the view plane. This process is illustrated in Figure 4.8.

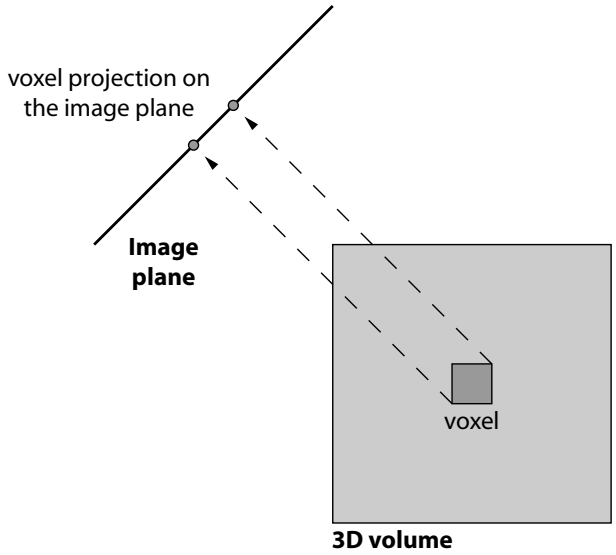


Figure 4.8: An illustration of the voxel projection process.

If the image plane is moved through the data, then the frame buffer is used as an accumulator and all pixels are updated simultaneously until the entire data set has been traversed and the pixels have their final values.

Possibly the most well know voxel projection algorithm is known as "splatting". This term is used to describe the effect that one voxel has on the image plane. This algorithm considers how the contribution of a voxel should be spread or splatted in the image plane. There are two possible ways that splatting can be used:

1. A point in the data set at the centre of a particular voxel projects onto a single pixel. The three dimensional region surrounding the sampled voxel is filtered to determine its contribution to the pixel.

2. A single pixel value can be spread over a number of pixels in the image plane.

Both of these approaches are equivalent. Representations of the two approaches are illustrated in Figure 4.9.

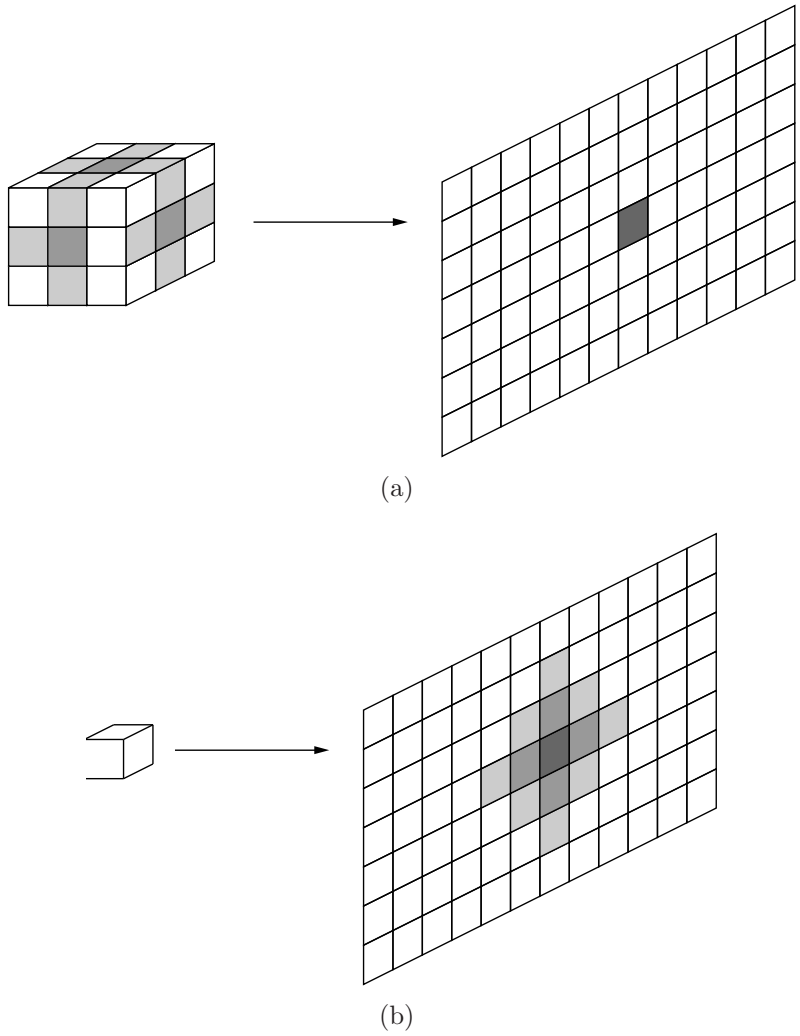


Figure 4.9: The two different approaches to splatting. (a) Many voxels are filtered over a spherical region to provide a single value for a pixel. (b) One voxel contributes value in a footprint region weighted as shown.

Reference:

- Westover L. (1990) "Footprint evaluation for volume rendering" Computer Graphics, **24**(4), 367-376.

4.3.3 Volume Rendering Implementation

The following example uses ray casting to generate a volume rendering of a CT data set. The program begins by loading a data set called `dataset.iso`. The dimensions of the data set are obtained by calling the relevant methods of the `DataSet` class. The dimension of the data set used in this example are:

- `width = 512 voxels`
- `height = 593 voxels`

- `depth = 512 voxels`

An initial raster scan of the data set is performed to determine the minimum and maximum voxel values. The extrema for the data set used in this example are:

- `minValue = -1024 HU`
- `maxValue = 1833 HU`

Recall that the values assigned to voxels in a CT data set are measured in Hounsfield Units. These units provide a measure of attenuation which is equivalent to density. A low value (-1024 HU) is equivalent to a low density material, e.g. air, whereas a high value (1833 HU) is equivalent to a high density material, e.g. bone.

In order to display a set of attenuation values in the form of a grey scale image, the values must be normalised in the range [0 - 255]. This is achieved by calculating two values:

1. **The offset -** this is the value that must be added to all attenuation values to ensure that each value is at least zero.
2. **The range -** this is the range of attenuation values. Dividing an offset attenuation value by the range always gives a value in the range [0 - 1].

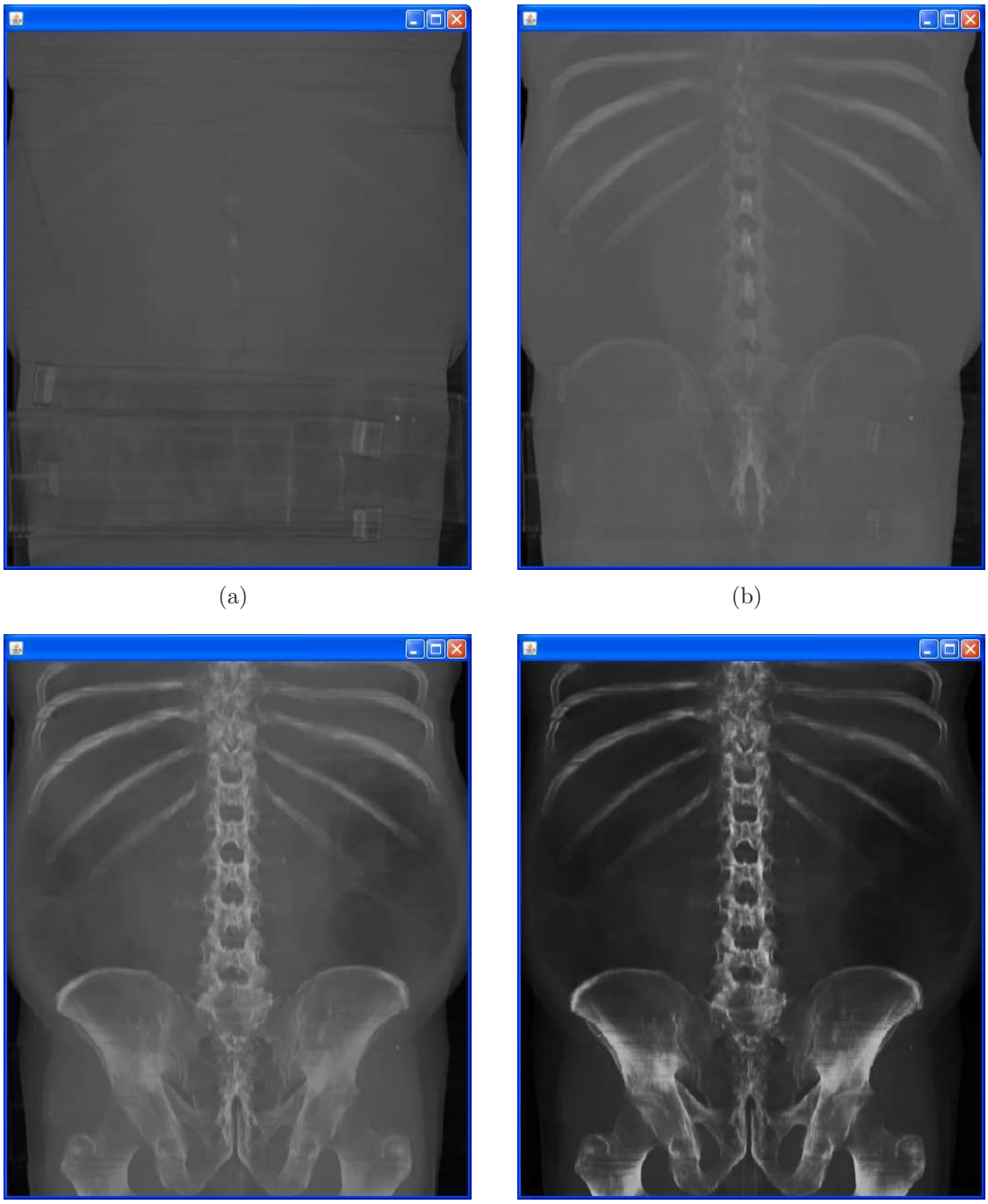
In order to create a grey scale value in the range [0 - 255] from an attenuation value, the following equation can be used.

$$grey = \frac{255 \times (attenuation + offset)}{range} \tag{4.2}$$

A `BufferedImage` object is created to hold the 2D volume rendering and for loop is entered to calculate each pixel value for the `BufferedImage` object. At each (x, y) the ray composition algorithm is applied along the z direction. The colour of each voxel is calculated using equation 4.2 and a corresponding alpha value is calculated by dividing the colour value by 255.0. This results in a value in the range [0 - 1]. The initial alpha value is scaled using an exponential function to increase the transparency of low density voxels.

The `JFrame` is configured and the `BufferedImage` is displayed on the `JFrame` using a `JLabel`.

```
0
import java.awt.image.*;
import javax.swing.*;
import java.awt.*;

5 public class VolumeRenderingExample extends JFrame {

  public static void main(String args[]){new VolumeRenderingExample();}

  public VolumeRenderingExample()
10 {
    // Load the data set and get dimensions
```



(a) (b)

(c) (d)

Figure 4.10: Volume renderings of the same abdominal CT data set where the opacity is scaled using different exponential functions. It is clear that as the scale of the exponential function is increased, the lower density material, i.e. the soft tissue, becomes more and more transparent. In (a) the exterior of the patient and the outer covering of the patient can be seen, whereas in (d) only the skeleton is visible.

```
    DataSet dataset = new DataSet("dataset.iso");

    int width = dataset.getWidth();
15  int height = dataset.getHeight();
    int depth = dataset.getDepth();
```

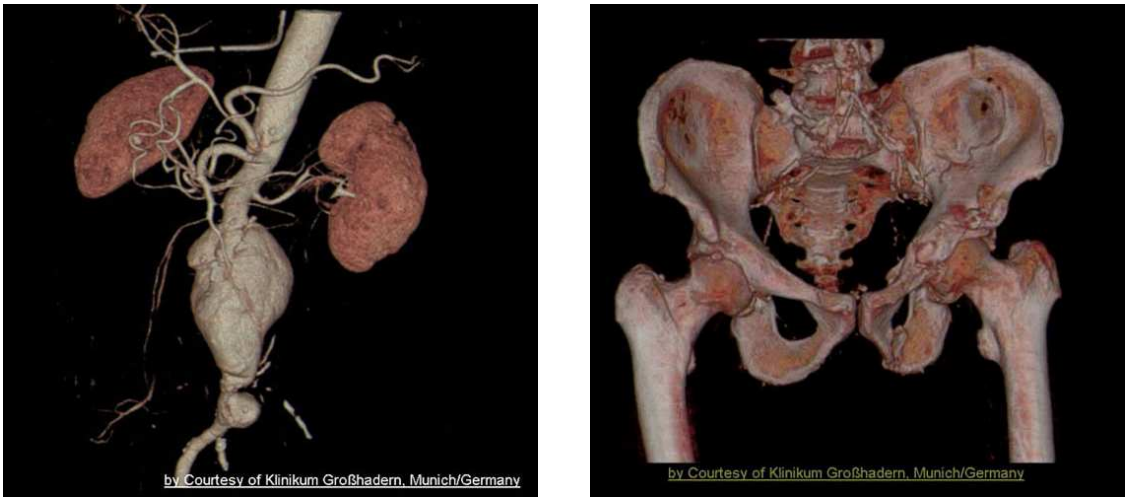
```
    // Calculate the exrema of voxel values
    int minValue = Integer.MAX_VALUE;
    int maxValue = Integer.MIN_VALUE;

    for(int x=0; x<width; x++)
      for(int y=0; y<height; y++)
        for(int z=0; z<depth; z++)
        {
          int voxel = dataset.getVoxel(x, y, z);
          if(voxel<minValue) minValue = voxel;
          if(voxel>maxValue) maxValue = voxel;
        }

    System.out.println("Min value: "+minValue+" HU");
    System.out.println("Max value: "+maxValue+" HU");

    // Calculate values required for normalisation
    int range = maxValue - minValue;
    int offset = - minValue;

    // Create a BufferedImage to store the rendered output
    BufferedImage bufferedImage = new BufferedImage(width, height,
        BufferedImage.TYPE_INT_ARGB);

    // Configure the exponential scaling function
    float exponentialFactor = 1.0f;
    float divisor = (float)Math.exp(exponentialFactor);

    // Enter the volume rendering loop
    float C_in, C_out = 0.0f;
    for(int x=0; x<width; x++)
      for(int y=0; y<height; y++)
      {
        C_in = 0.0f;
        for(int z=depth-1; z>=0; z--)
        {
          // Calculate the colour and opacity
          float C = dataset.getVoxel(x, y, z);
          C = (255*(C + offset))/range;

          float alpha = C/255.0f;
          alpha = (float)(alpha * Math.exp(exponentialFactor*alpha)/divisor);

          // Applied the ray compositing formula
          C_out = C_in*(1.0f - alpha) + C*alpha;
          C_in = C_out;
        }

        // Generate a pixel value for ray and store in output image
        int grey = (int)C_out;
        int pixel = 0xff000000 | (grey << 16) | (grey << 8) | grey;
        bufferedImage.setRGB(x, height-1-y, pixel);
      }
```

```
        // Configure the JFrame and display the image
        this.getContentPane().setLayout(new BorderLayout());
        JLabel label = new JLabel();
75      label.setIcon(new ImageIcon(bufferedImage));
        this.getContentPane().add(label, BorderLayout.CENTER);
        this.pack();
        this.setVisible(true);
    }
80  }
```

A series of different volume renderings can be obtained by changing the exponential scaling function. Examples of these volume renderings are illustrated in Figure 4.10

4.4 Sample Volume Renderings

As mentioned earlier, volume rendering is typically used with volumetric data obtained using a medical imaging modality. The following examples, illustrated in Figure 4.11, demonstrate how volume rendering can be used to generate extremely detailed representations of human anatomy.



(a) (b)

Figure 4.11: Two examples of high quality volume renderings obtained using commercial volume rendering software. In both cases pseudo colouring has been used to help distinguish between different voxel classes and enhance the appearance of the rendered output.

4.5 Medical Imaging Modalities

Volumetric data sets are typically acquired using a medical imaging modality that sample the characteristics of the region occupied by the patient at regular intervals. The images generated by 3D medical imaging modalities typically consist of thin slices of voxels that can be stack to create a volumetric representation of the scanned

region. This section will describe two medical imaging modalities: Computed tomography (CT) and magnetic resonance imaging (MRI).

4.5.1 Computed Tomography

Computed tomography (CT), originally known as computed axial tomography (CAT), is a medical imaging method employing tomography where digital geometry processing is used to generate a 3D volumetric representation of a scanned region from a large series of 2D X-ray images taken around a single axis of rotation.

X-ray slice data is generated using an X-ray source that rotates around the object. X-ray sensors are positioned on the opposite side of the circle from the X-ray source. Many data scans are progressively taken as the object is gradually passed through the gantry. The scans are ultimately combined together using a mathematical procedure known as tomographic reconstruction. There have been several generations of CT scanners and these are illustrated in Figure 4.12. Examples of the techniques used by two commercial CT scanners are illustrated in Figure 4.14.

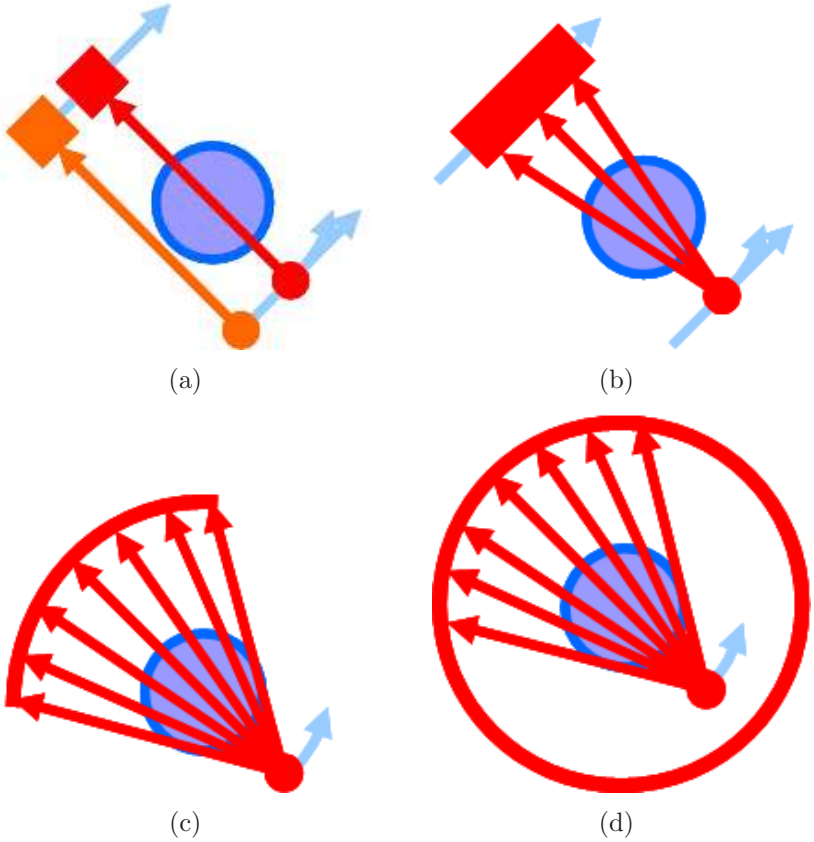
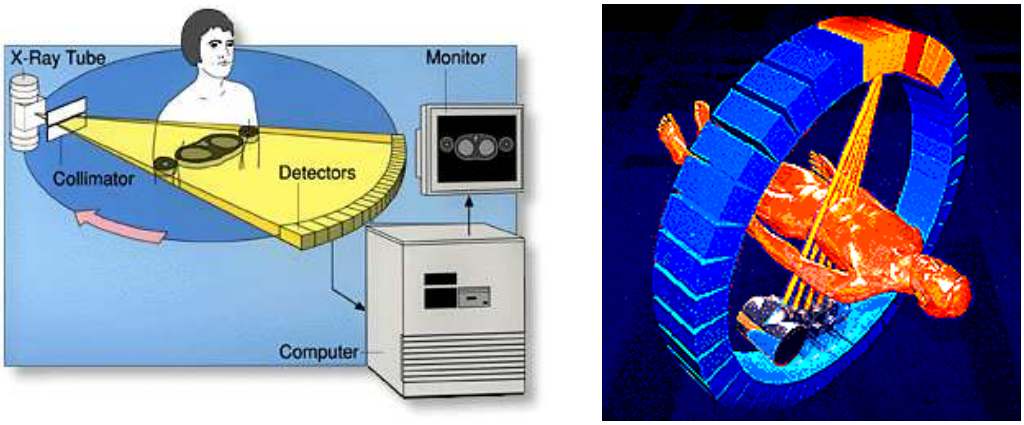


Figure 4.12: An illustration of the four generations of CT scanners. (a) The first generation: one detector, translation rotation, parallel beam. (b) Second generation: multiple detectors, translation-rotation, small fan-beam. (c) Third generation: multiple detectors, source rotation, large fan beam. (d) Fourth generation: Detector ring, source-rotation, large fan-beam.

CT is regarded as a moderate to high radiation diagnostic technique. While technical advances have improved radiation efficiency, there has been a simultaneous pressure to obtain higher-resolution image and use more complex scan techniques,

both of which require higher doses of radiation. The effective radiation dose for a chest CT is 290 times the dose for a chest X-ray.



(a) (b)

Figure 4.13: An illustration of the operation of two commercially available CT scanning systems. (a) A siemens system, third generation. (b) A Picker system, fourth generation.

An example of an actual Siemens CT scanner is illustrated in Figure 4.14. This is a Siemens Somatom Sensation 16 scanner. It has 0.5 mm resolution and a 0.4 second rotation time. This type of high performance rotation is key for scanning the beating heart (4D medical imaging).

Figure 4.14: An image of a Siemens Somatom Sensation 16 scanner. This scanner has 0.5 mm resolution and 0.4 second rotation time.

4.5.2 Magnetic Resonance Imaging

Magnetic resonance imaging (MRI) is a non-invasive method used to render an image of the interior of an object. It is primarily used in medical imaging to demonstrate pathological[1] or other physiological[2] alterations of living tissues.

A CT scanner uses X-rays, a type of ionising radiation, to acquire its images. MRI, on the other hand uses non-ionising radio frequency signals to acquire its images. MRI relies on the properties of hydrogen molecules in water when influenced by a strong magnetic field. Even though the strength of the magnet is typically 100,000 the strength of the earths magnetic field, an MRI examination poses no threat to the subject being imaged.

It should be noted that although CT and MRI both generate volumetric representations of the interior of a patient, MRI takes longer to scan a volume than CT, MRI responds differently to materials than CT and MRI does not provide as high a resolution as CT. An MRI scanner provides a maximum resolution of $1mm^3$. A modern commercially available MRI scanner is illustrated in Figure 4.15.

Figure 4.15: A three tesla Philips MRI scanner.

[1] Caused by a disease.
[2] Relating to the functions and activities of living organisms.

4.6 Stereo 3D Visualisation

All of the 3D content discussed in this course has been rendered to a 2D display. This essentially flattens the 3D content so that it can be viewed using conventional visualisation systems, e.g. LCD and CRT displays.

In order to get that full effect of 3D content, it must be viewed using a stereo 3D visualisation system where a different views of the content are created for the left and the right eyes. Using this approach the content appears to have a depth associated with it. This adds a great deal of realism to a 3D scene. An overview of the stereo viewing process is illustrated in Figure 4.16.

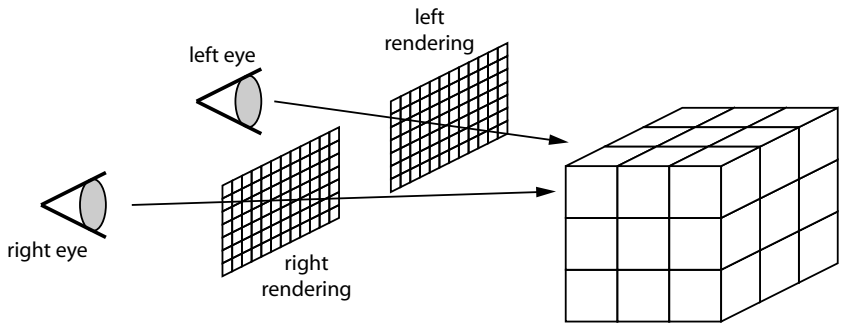


Figure 4.16: An illustration of the stereo viewing process where separate renderings are created for the left and right eyes.

One example of a stereo 3D viewing system is CrystalEyes from StereoGraphics (RealD). This system utilises a pair of LCD shutter glasses in conjunction with a CRT display and a compatible high refresh rate view card to facilitate stereo 3D viewing.

Assume that the CRT monitor has a refresh rate of 120 Hz. The CrystalEyes system displays the left eye and right eye renderings of a 3D scene in subsequent frames. A signal is sent to the CrystalEyes glasses so that right lens is opaque when the left eye image is being displayed and the left lens is opaque when the right eye image is being displayed. The CrystalEyes equipment is illustrated in Figure 4.17.

Figure 4.17: The CrystalEyes LCD shutter glasses and remote control unit.

4.7 Summary

This chapter has presented an overview of volume rendering techniques dealing with ray casting and voxel projection based approaches. A implementation of a ray casting based volume rendering system demonstrated the simplicity of the approach. It should be noted that the direct volume rendering techniques discussed in this chapter provide a very different output to the indirect volumetric rendering technique based on the marching cubes algorithm that was discussed in the previous chapter. This chapter also provided a overview of common medical imaging modalities and introduced the concept of immersive 3D stereoscopic visualisation.