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.

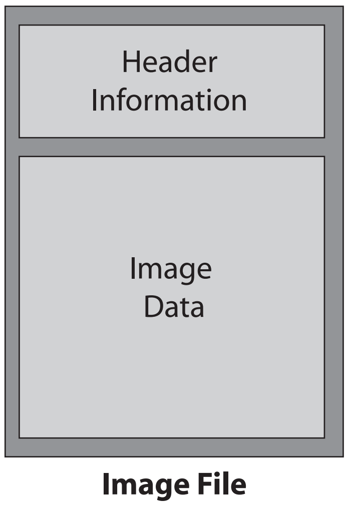


Figure 1.1: An illustration of the format of a standard image file that consists of separate header information and image data.

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.

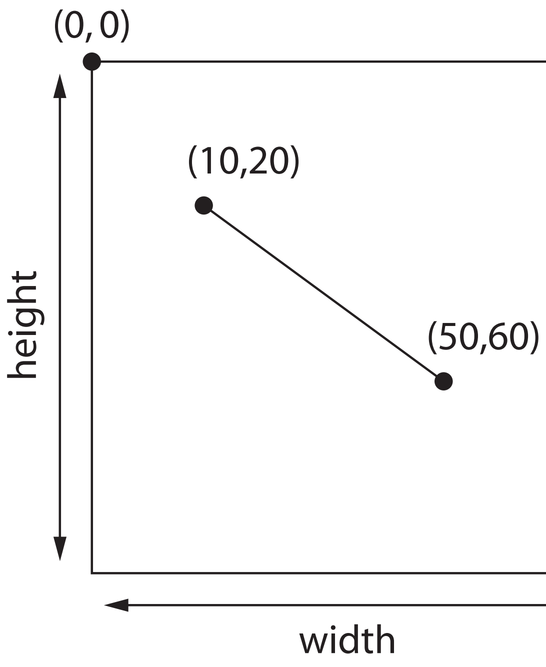


Figure 1.2: An illustration of the 2D graphics coordinates system used by Java.

- `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.

```
import java.awt.*;

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

  Frame frame = new Frame();

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

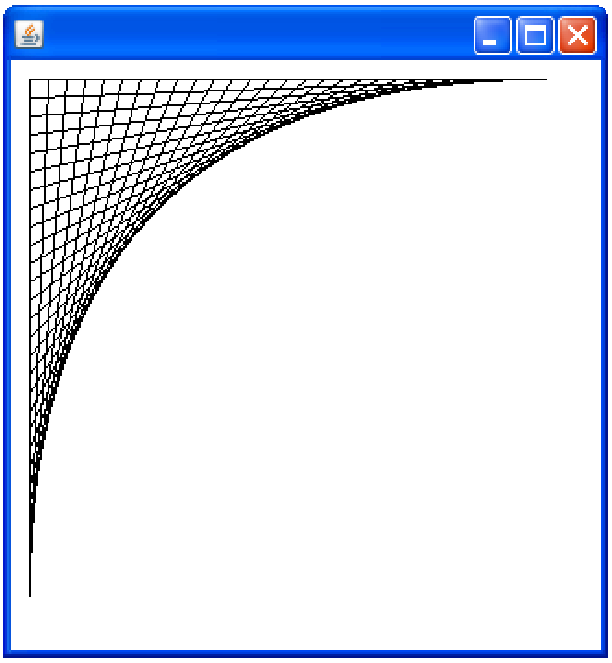


Figure 1.3: The pattern generated by the `GraphicsExample` application.

```
    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.

```java
import java.awt.*;

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

  Image i;

  public ImageLoadExample()
  {
    super();

    // Load the image and resize the frame
    i = Toolkit.getDefaultToolkit().createImage("image.jpg");
    waitForImage(i);
    int width = i.getWidth(this);
    int height = i.getHeight(this);
    setPreferredSize(new Dimension(width, height));
    frame.pack();
  }
  public void paint(Graphics g)
  {
    // Paint the image on the canvas
    g.drawImage(i, 0, 0, this);
  }

  // Wait for image 'i' to be loaded
  public void waitForImage(Image i)
  {
    try{
      MediaTracker tracker = new MediaTracker(this);
      tracker.addImage(i,0);
```

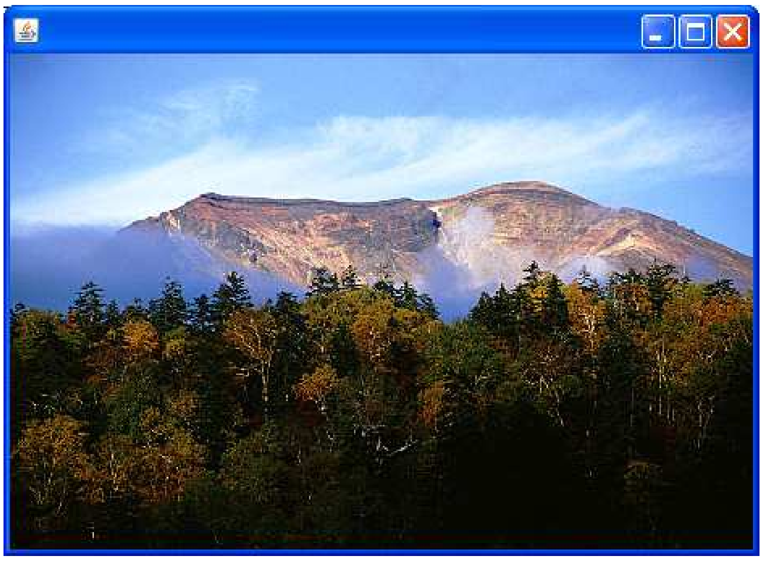


Figure 1.4: An illustration of the image loaded using the `ImageLoadExample` program.

```
      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.

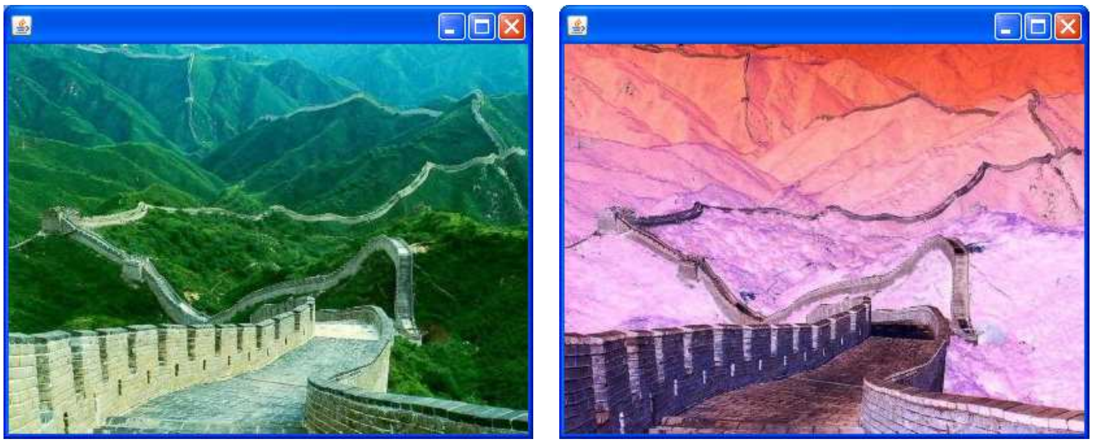
Note: An alternative to using the `MediaTracker` class would be monitor the dimensions of the image using the `getWidth()` or `getHeight()` methods. If the image data is not fully loaded then then these methods will return -1. When the image data is loaded then these methods will return the relevant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

```
}
```

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

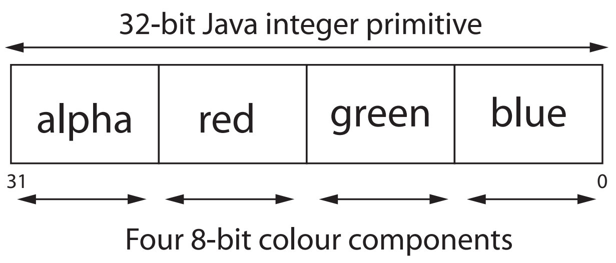


Figure 1.6: An illustration of the pixel format used by Java. The 32-bit bits of an integer primitive are divided into four 8-bit colour components representing: alpha or opacity, red, green and blue. The minimum value for each colour component is 0 and the maximum value for each component is 255.

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 `BufferedImage` 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.

```java
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:

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

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

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

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

    // Set the anti-aliasing to the default mode
    g2d.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
        RenderingHints.VALUE_ANTIALIAS_DEFAULT);

    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`.

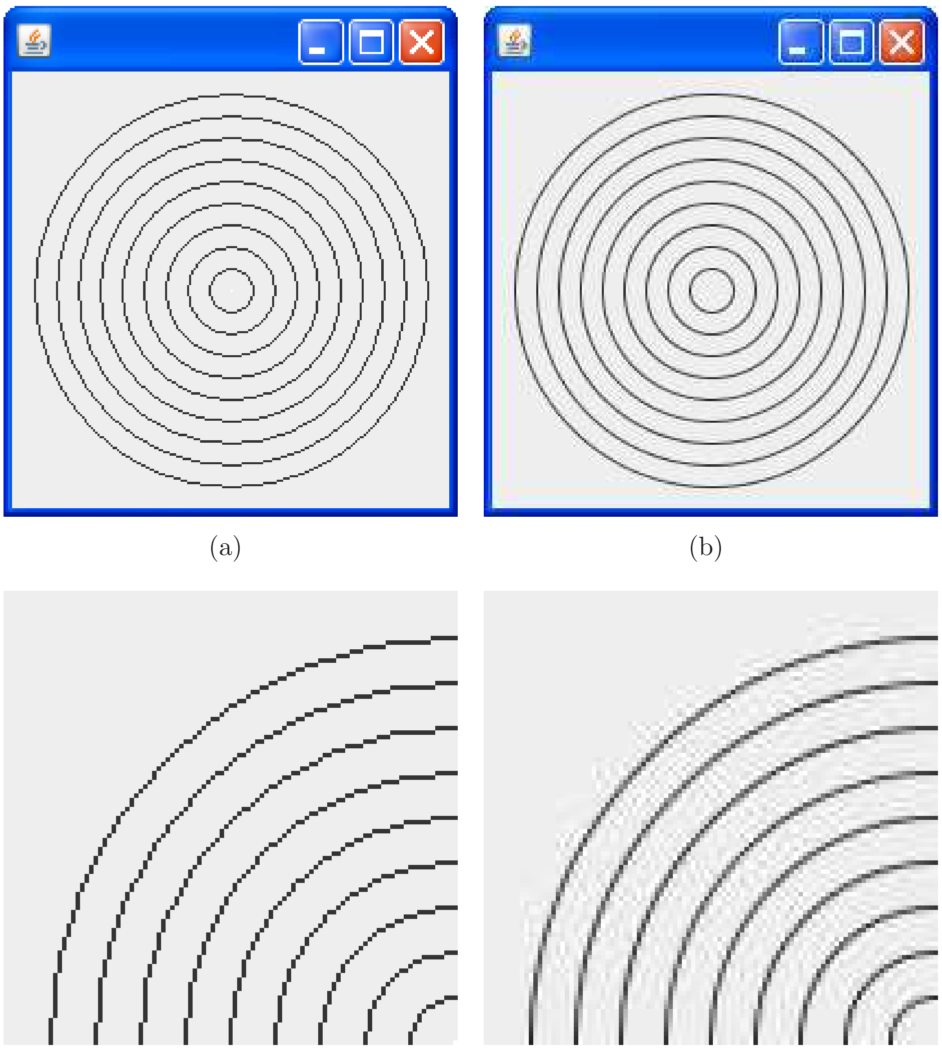


Figure 1.7: An example of anti-aliasing. A series of concentric circles drawn without anti-aliasing (a) and zoomed version (c). The same circles drawn with anti-aliasing (b) and zoomed version (d).

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.

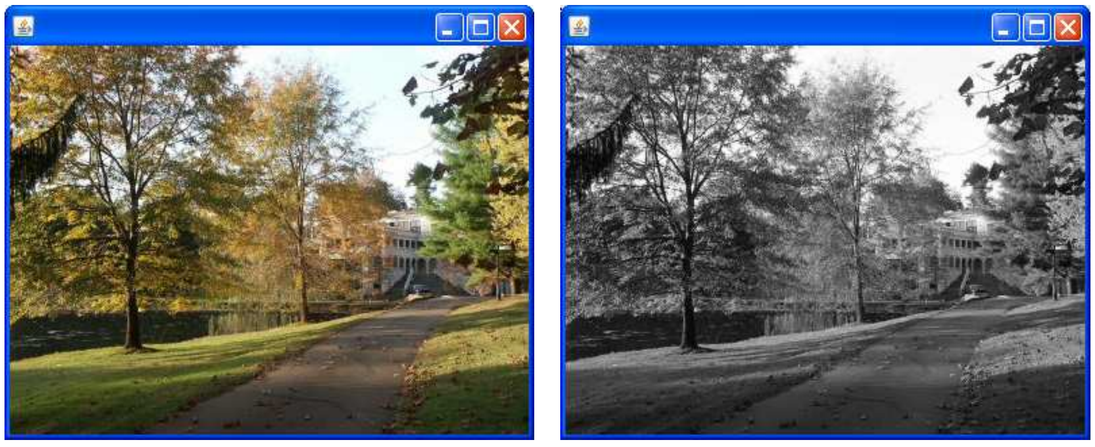
```
import java.io.*;
import javax.imageio.*;
import javax.swing.*;
import java.awt.*;
import java.awt.image.*;

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.

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

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:

$$\begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.2}$$

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.

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 0.5 & 0.0 & 0.0 \\ 0.0 & 2.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \end{bmatrix} \begin{bmatrix} 2.0 \\ 3.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 0.5 \times 2.0 + 0.0 \times 3.0 + 0.0 \\ 0.0 \times 0.0 + 2.0 \times 3.0 + 0.0 \\ 1.0 \end{bmatrix} = \begin{bmatrix} 1.0 \\ 6.0 \\ 1.0 \end{bmatrix} \tag{1.3}$$

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:

$$\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \tag{1.4}$$

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).

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

public class Translate2DExample extends BufferedImageLoadExample{

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

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

      double tx = 100.0;
      double ty = 40.0;

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

    AffineTransform translateTransform = new AffineTransform(matrix);

    // Set the interpolation type
    int interpolationType = AffineTransformOp.TYPE_NEAREST_NEIGHBOR;
```

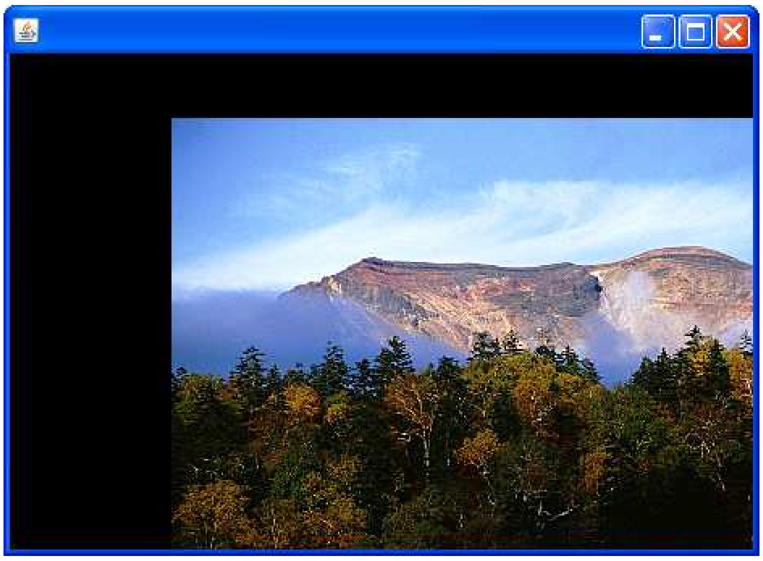


Figure 1.9: A translated version of the image illustrated in Figure 1.4. The translation moves the origin of the image by 100 pixels in the x direction and 40 pixels in the y direction.

```
    AffineTransformOp translateTransformOp =
      new AffineTransformOp(translateTransform,
                                interpolationType);

    // Perform the transformation
    BufferedImage result = translateTransformOp.filter(b, null);
    g2d.drawImage(result, 0, 0, this);
  }
}
```

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:

$$\begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1.5}$$

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.

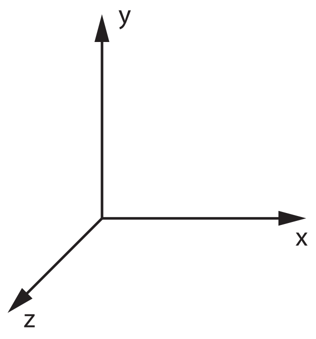


Figure 1.10: An illustration of the VRML coordinate system.

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:

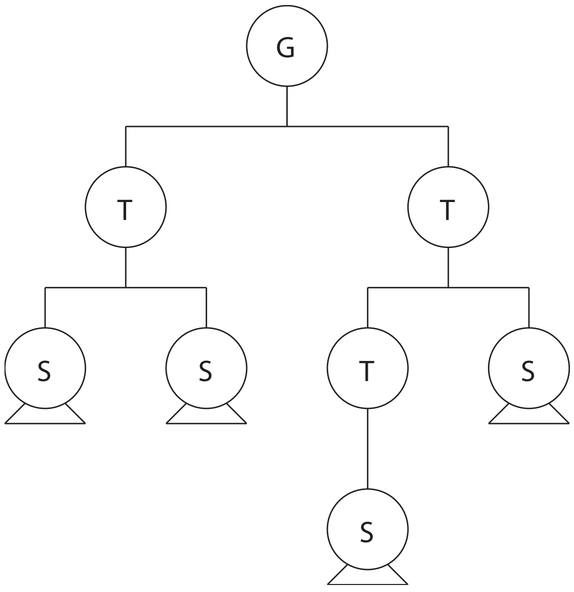


Figure 1.11: An example of a scene graph consisting of a ground node, three transform nodes and three shape nodes.

- 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

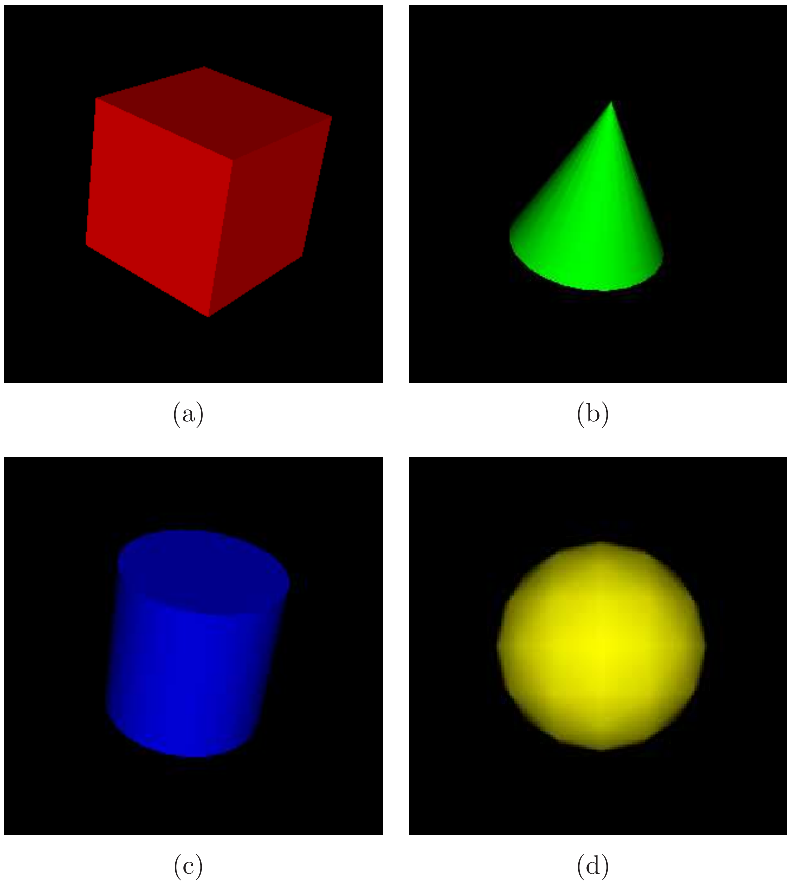


Figure 1.12: Renderings of four of the primitives supported by VRML (a) box, (b) Cone, (c) Cylinder, (d) Sphere.

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:

```
#VRML V2.0 utf8

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

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

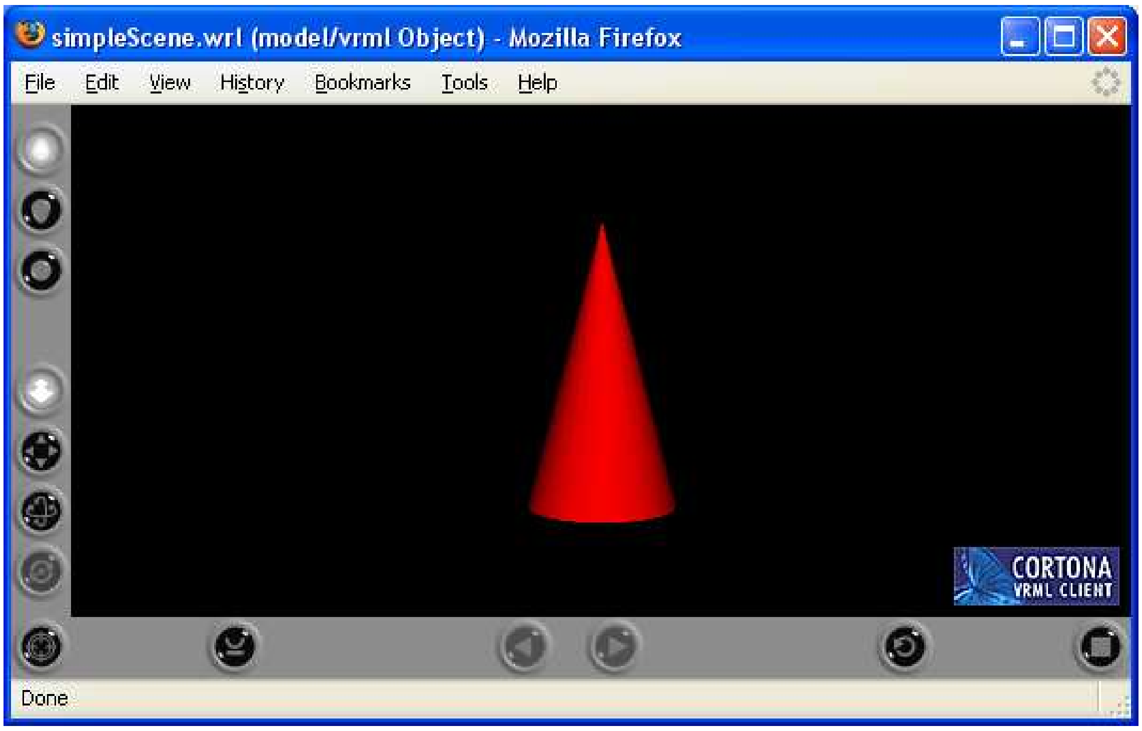

simpleScene.wrl (model/vrml Object) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
CORTONA VRML CLIENT
Done

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 i 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.

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} m_{00}x + m_{01}y + m_{02}z + m_{03}w \\ m_{10}x + m_{11}y + m_{12}z + m_{13}w \\ m_{20}x + m_{21}y + m_{22}z + m_{23}w \\ m_{30}x + m_{31}y + m_{32}z + m_{33}w \end{bmatrix} \tag{1.6}$$

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
```

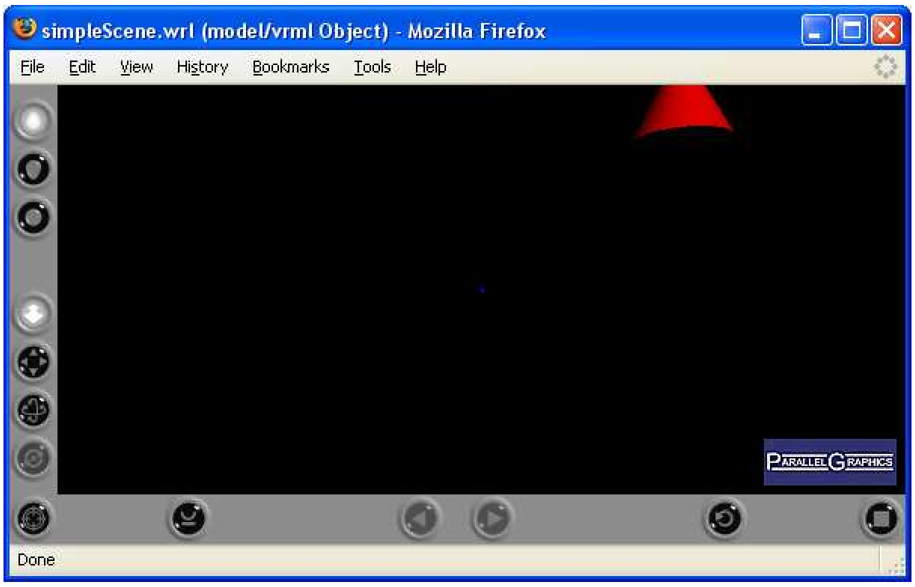

simpleScene.wrl (model/vrml Object) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
ParallelGraphics
Done

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.

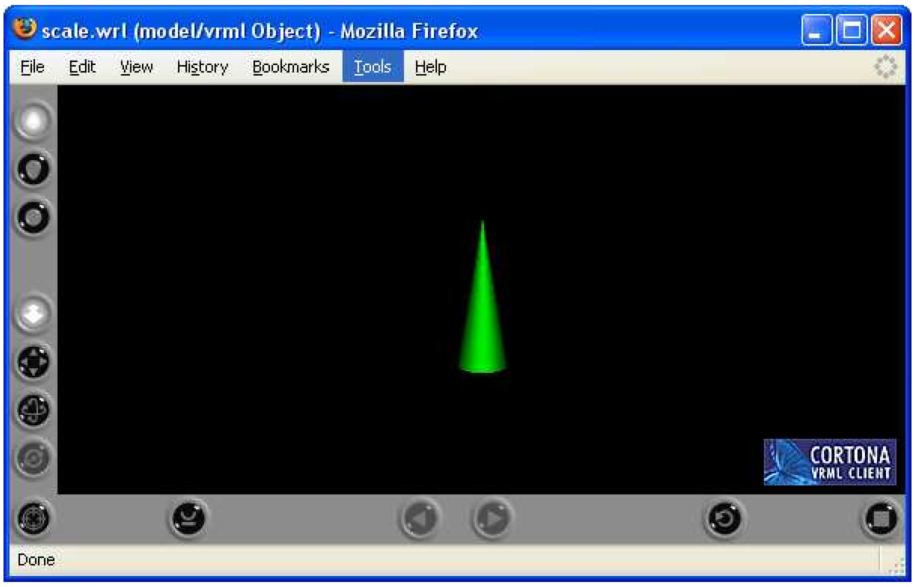

scale.wrl (model/vrml Object) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
CORTONA VRML CLIENT
Done

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 give 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{ }
        }
}
```

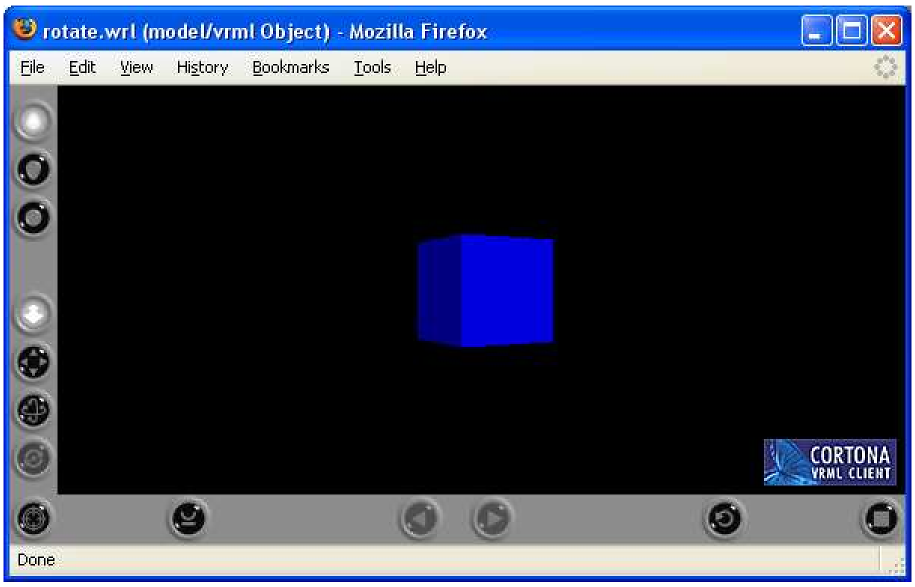

rotate.wrl (model/vrml Object) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
CORTONA VRML CLIENT
Done

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.

```
#VRML V2.0 utf8

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

        }

}
```

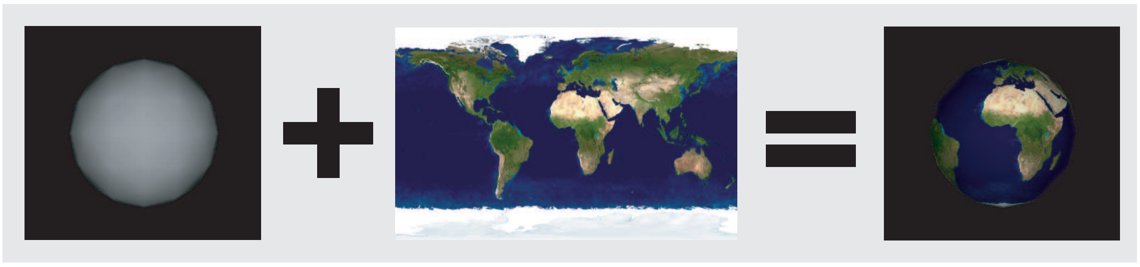


Figure 1.17: An illustration of the texture mapping process. The earth texture is mapped to the sphere by wrapping it once around the horizontal diameter and squeezing it to a single point at the top and bottom.

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.

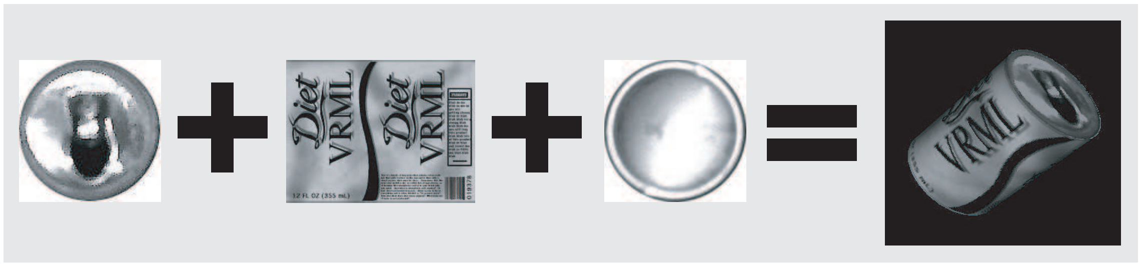


Figure 1.18: An illustration of the texture mapping process to create a soft drinks can from a cylinder geometry. Three separate textures must be specified in order to completely specify the appearance of the can.

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.

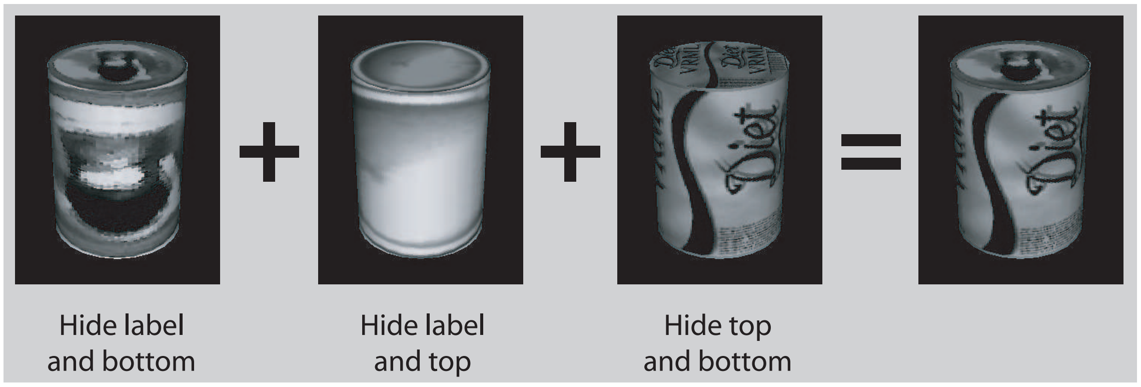


Figure 1.19: The stages of texture mapping that are involved to create a soft drinks can. Three separate cylinders must be created with the required textures. In each case the faces that are not required must be hidden and the cylinders must be co-located to give the final result.

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

```
#VRML V2.0 utf8

Group {
    children [
    # Can top
        Shape {
            appearance Appearance {
                material Material { }
                texture ImageTexture {
                    url ”textures/cantop.jpg”
                }
            }
            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)

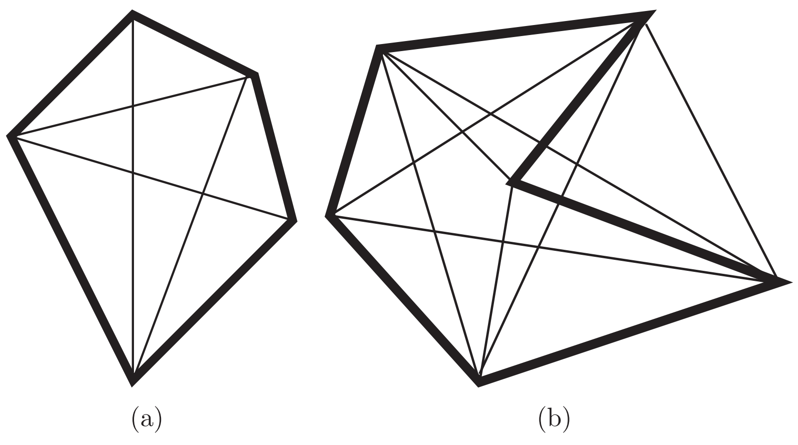


Figure 1.20: Examples of convex (a) and non-convex (b) faces. In the convex example the interconnections between each of the vertices are all located within the face. In the non-convex example some of the interconnections intersect with the boundaries of the face.

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:

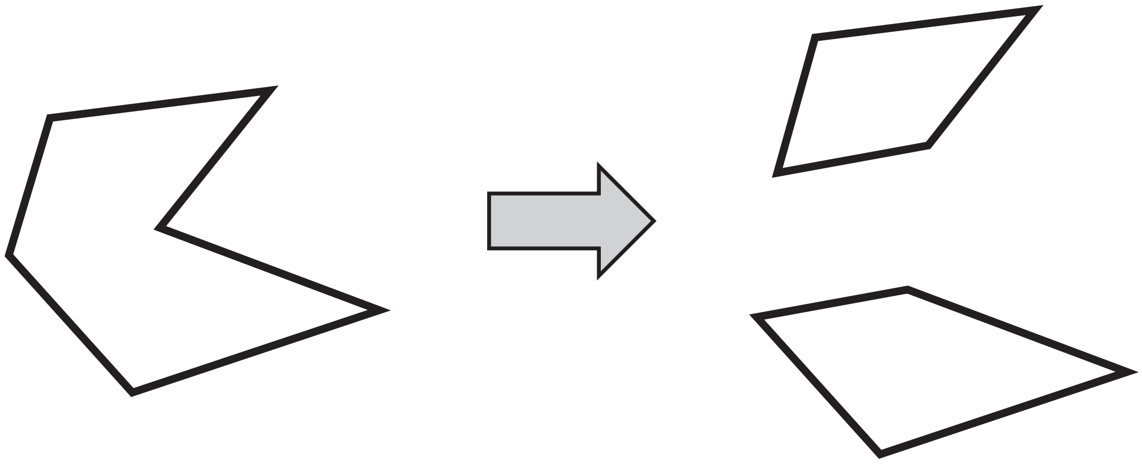


Figure 1.21: An example of a convex face divided into two non-convex faces.

```
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
        {
```

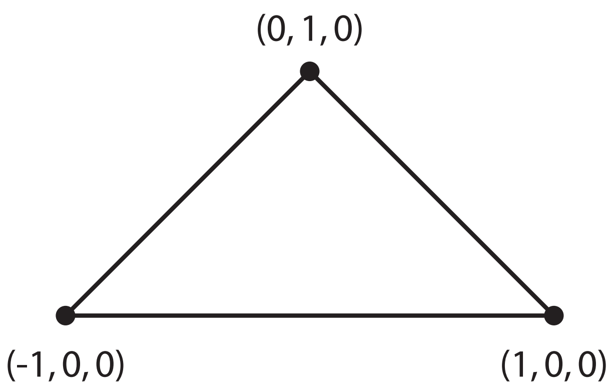


Figure 1.22: A simple equilateral triangle with vertices at (0,1,0), (-1,0,0) and (1,0,0)

```
                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.

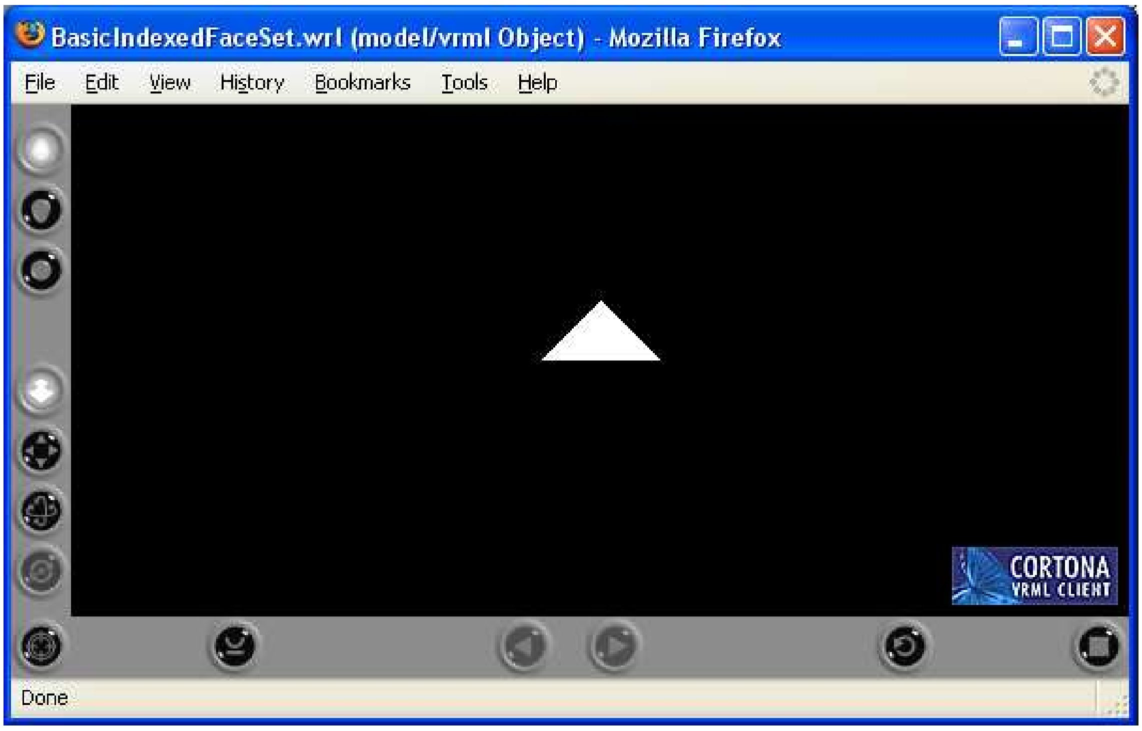

BasicIndexedFaceSet.wrl (model/vrml Object) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
CORTONA VRML CLIENT
Done

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

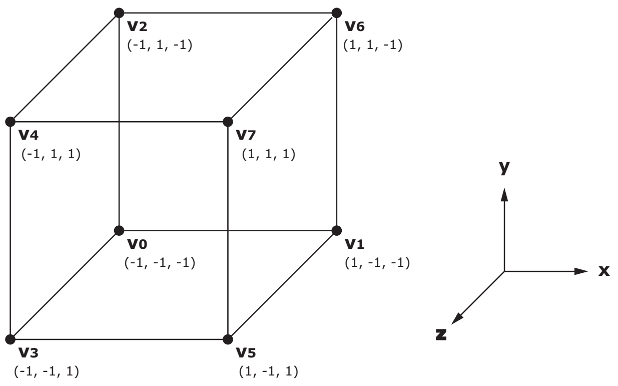


Figure 1.24: A simple cube consisting of six faces and eight vertices.

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.

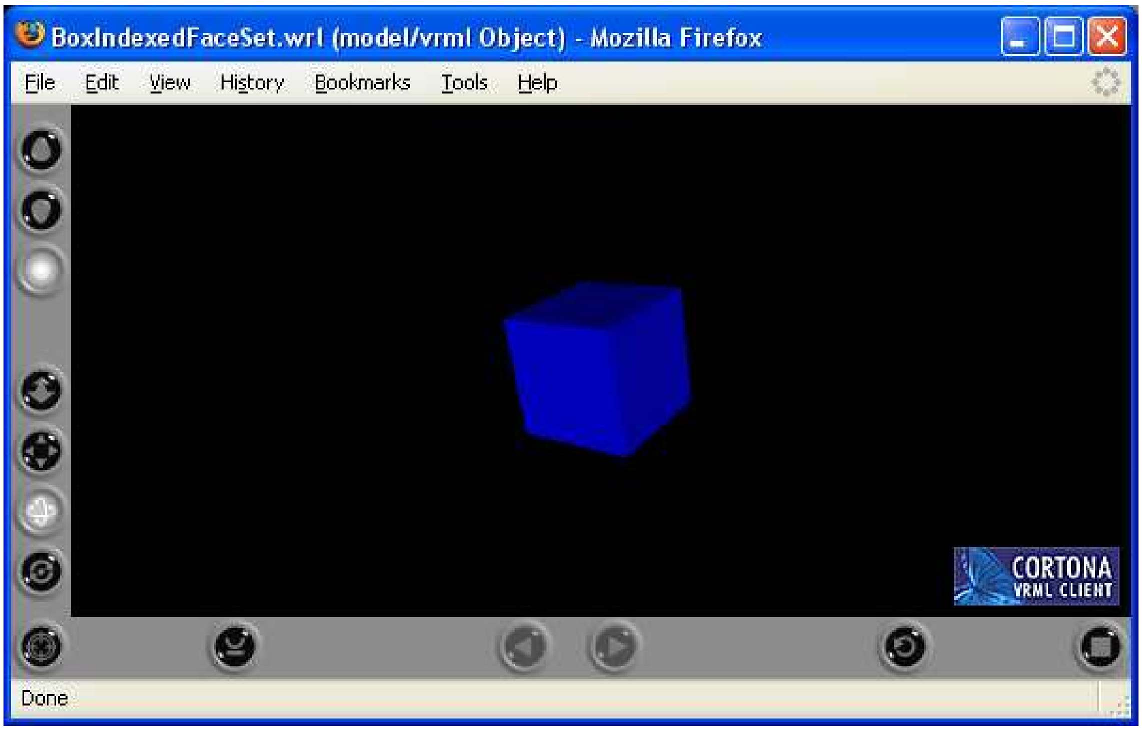

BoxIndexedFaceSet.wrl (model/vrml Object) - Mozilla Firefox
File Edit View History Bookmarks Tools Help
CORTONA VRML CLIENT
Done

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 introduces 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.