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 resulting code can easily be related to the original scene graph diagram. An example of a scene graph is illustrated in Figure 2.1.

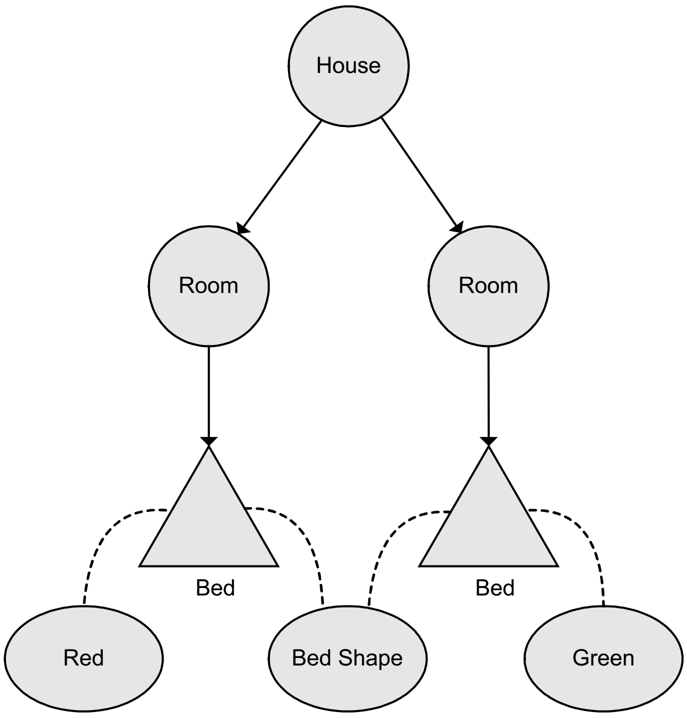

Figure 2.1: An example of a scene graph that represents a house. The house has two rooms and each room has a bed. Both of the beds have the same shape but different appearances.

This scene graph represents a house that has two rooms. There is a bed in each of the rooms. Both of the bed 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.

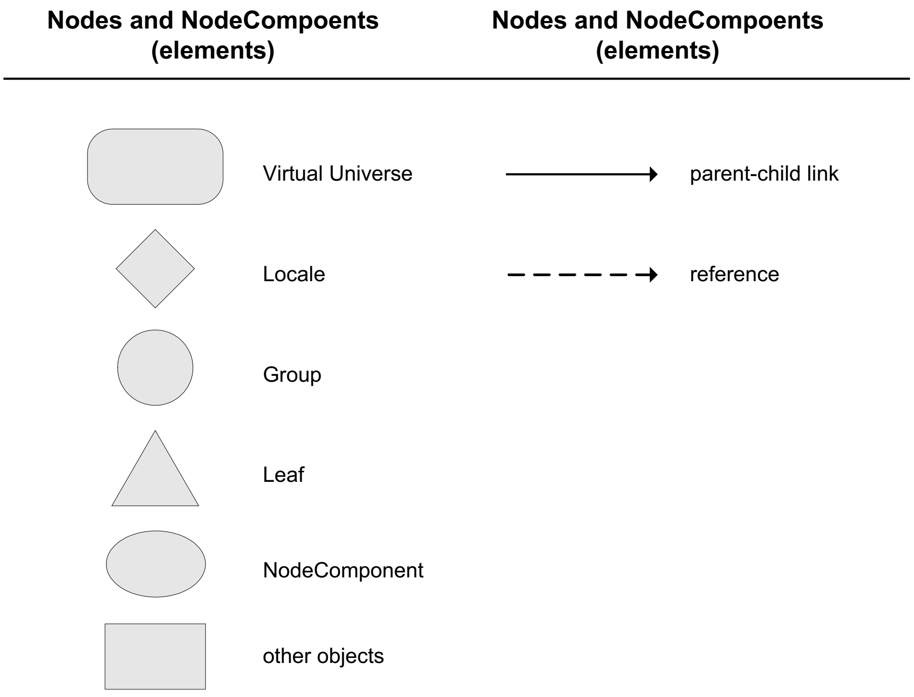

Figure 2.2: An overview of the range of symbols that are used to create Java 3D scene graphs.

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.

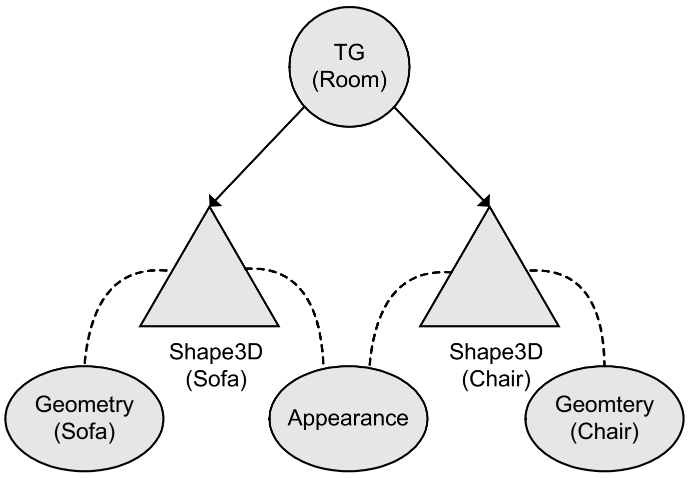

Figure 2.3: An example of how a single NodeComponent object can be shared by several scene graph nodes. In this example, both Shape3D objects share the same appearance. The relationship between the Shape3D nodes and the Appearance node component is represented by a dashed line to indicate a reference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

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

import java.awt.BorderLayout;
import java.awt.GraphicsConfiguration;

```

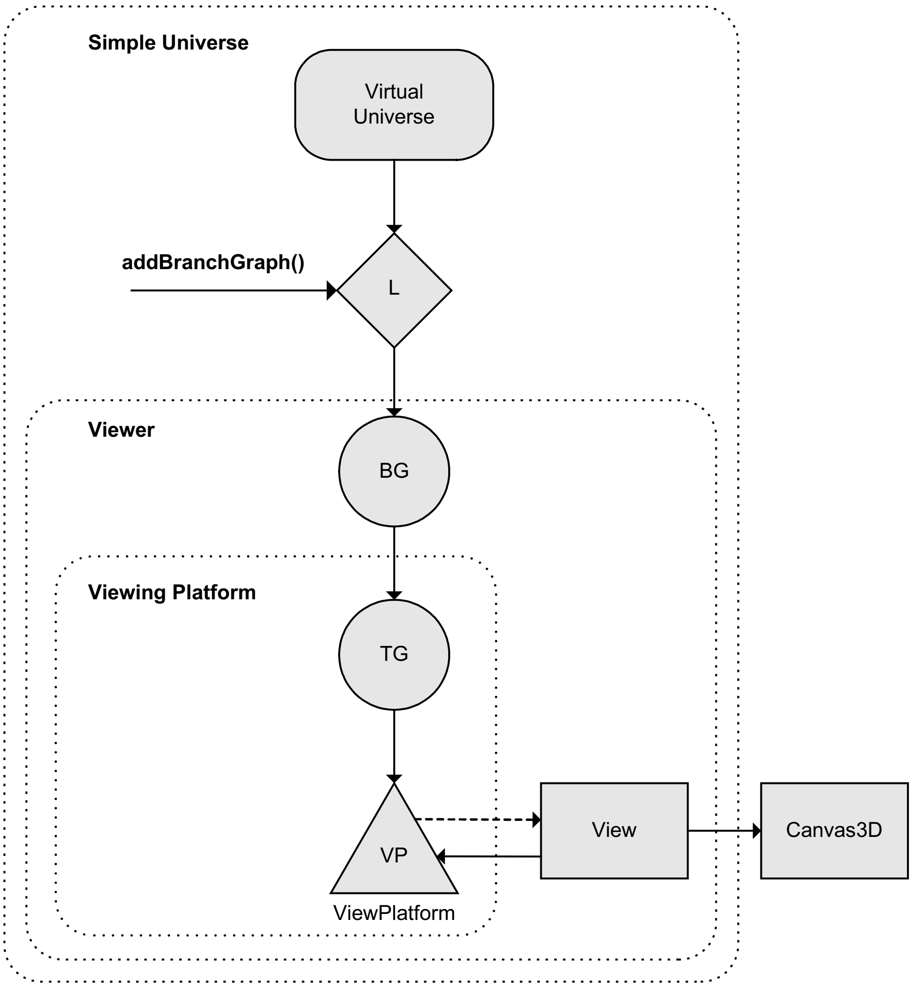

Figure 2.4: A simplified illustration of the scene graph diagram for the `SimpleUniverse` utility class. This scene graph represents the "view" branch graph. The "content" branch graph is added using the `addBranchGraph()` method.

```java
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 JFrame
{
  public static void main(String args[])
  {
    new BasicScene();
  }

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

    GraphicsConfiguration config =
      SimpleUniverse.getPreferredConfiguration();

    // Create a Canvas3D object and add it to the frame
```

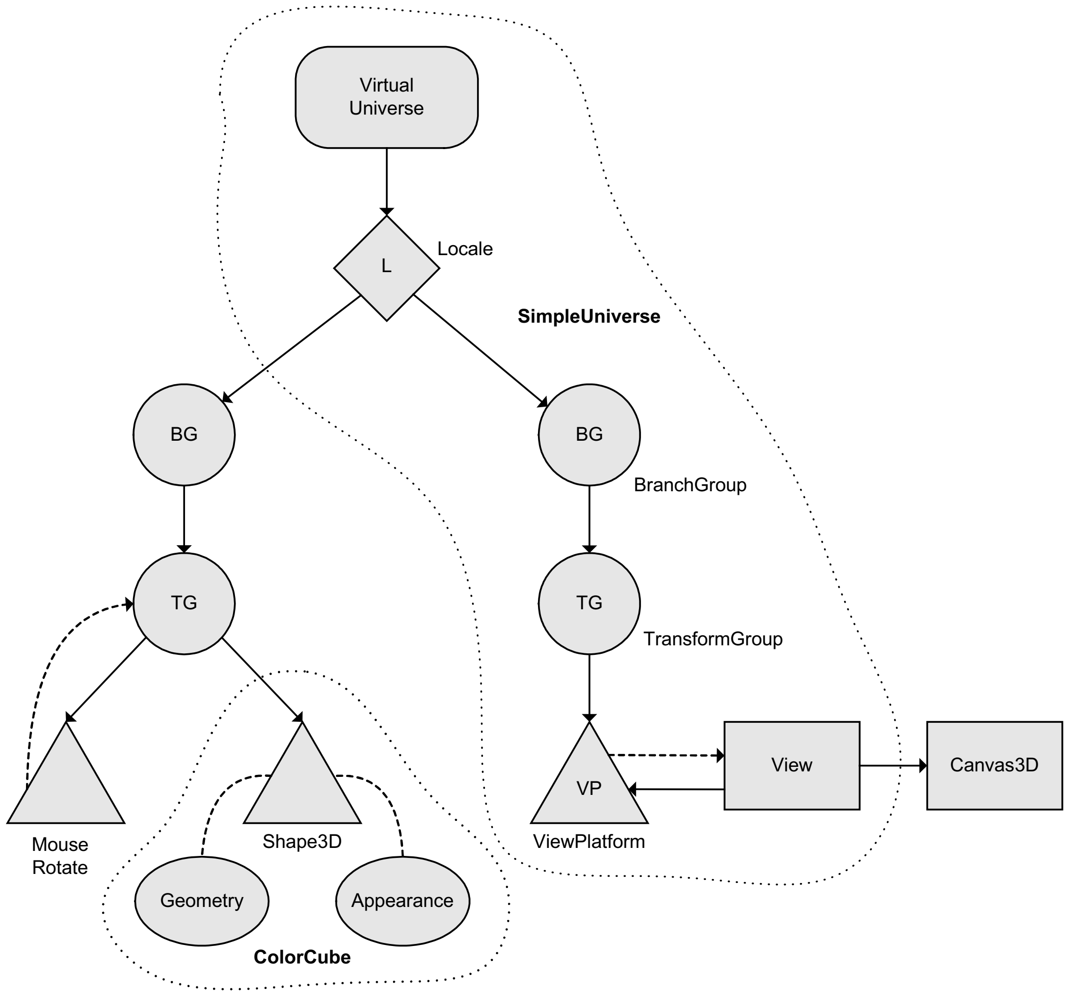

Figure 2.5: The scene graph used to generate `BasicScene.java`. The "content" branch graph consists of a root represented by a `BranchGroup` object, a `TransformGroup` that is modified by a `MouseRotate` behaviour and a `ColorCube` object that connected to the `TransformGroup` object.

```
    Canvas3D canvas = new Canvas3D(config);
    getContentPane().add(canvas, BorderLayout.CENTER);

    // Create a SimpleUniverse object to mange the "view" branch
    SimpleUniverse u = new SimpleUniverse(canvas);
    u.getViewingPlatform().setNominalViewingTransform();

    // Add the "content" branch to the SimpleUniverse
    BranchGroup scene = createContentBranch();
    u.addBranchGraph(scene);

    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_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 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:

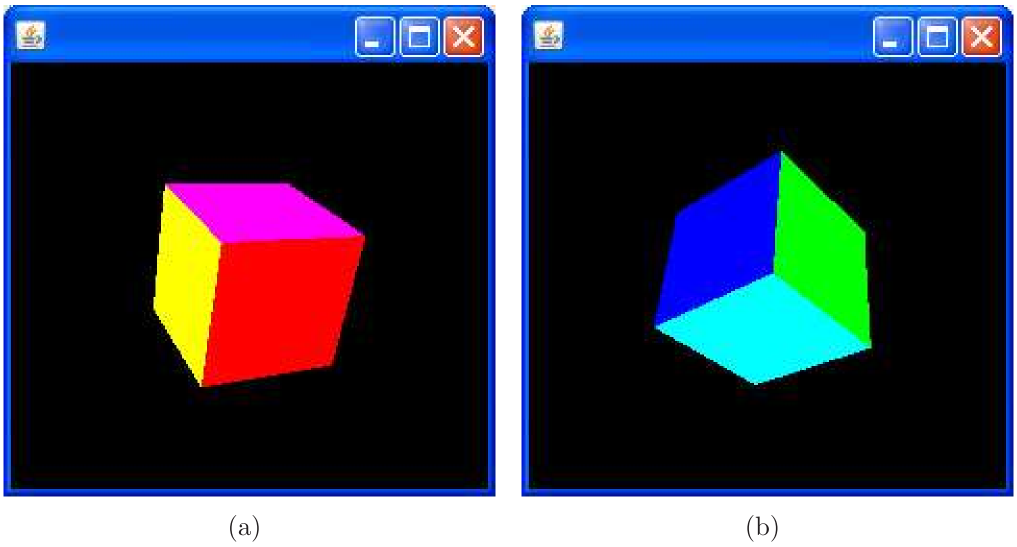

Figure 2.6: A `ColorCube` consists of six faces with different colours: red, green, blue, yellow, purple and cyan. This illustration shows two different renderings of a `ColorCube` object created using the `BasicScene` example.

- `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:

```
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();}
```

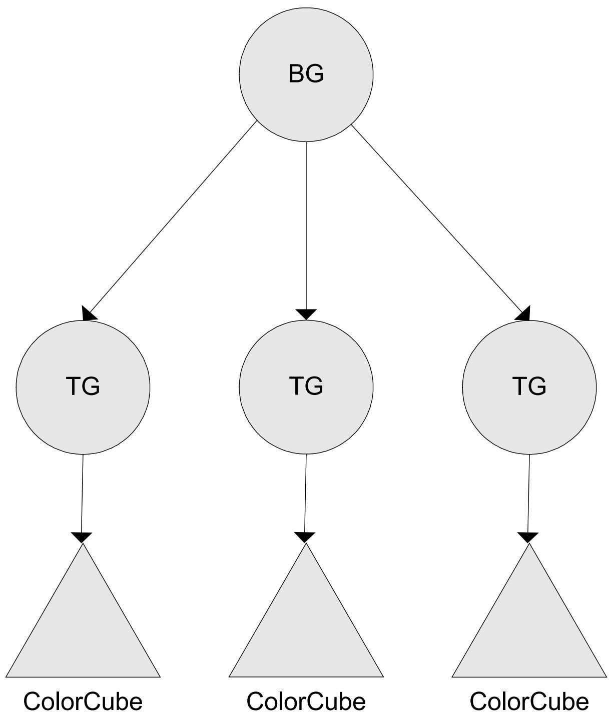

Figure 2.7: The scene graph for the `TransformGroup` example. This scene graph consists of a `BranchGroup` that represents the root, the root `BranchGroup` has three children. These children are `TransformGroup` objects which each have one `ColorCube` child. The location of the `ColorCube` nodes in the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

```
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.

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.

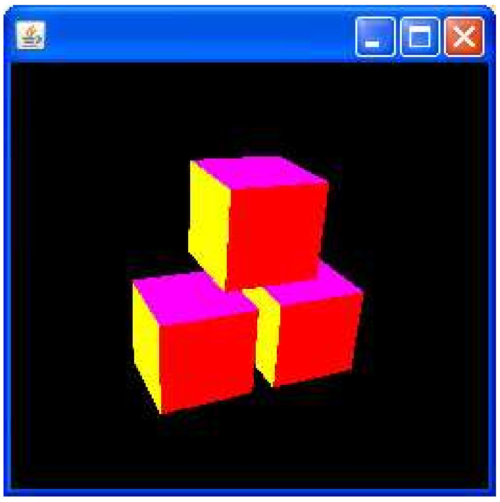

Figure 2.8: A rendering of the scene graph illustated in Figure 2.7. The three 40 cm `ColourCubes` appear in a triangular formation with a 10 cm gap between them

- `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 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 BisSet 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:

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

public class SwitchExample extends BasicSceneWithMouseControl{
```

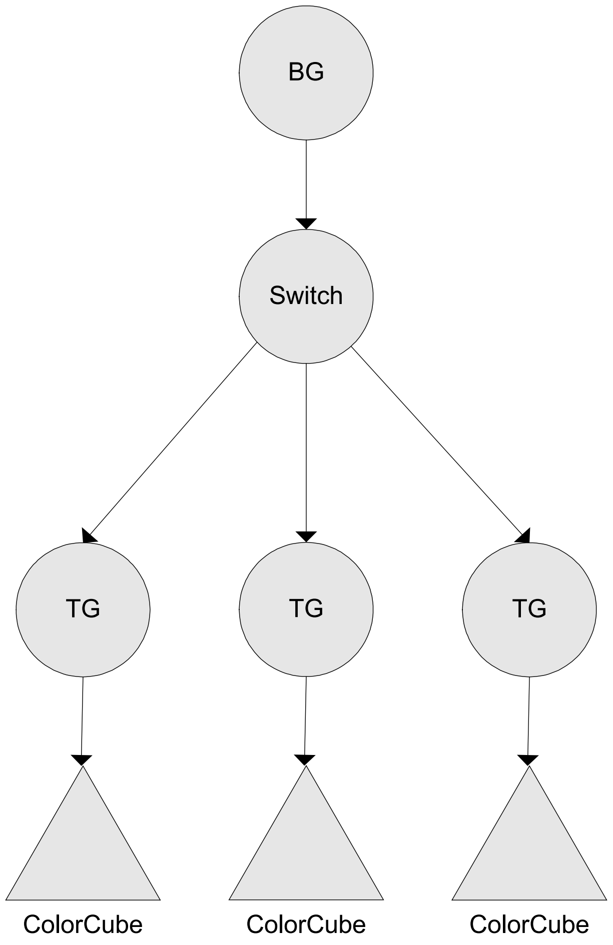

Figure 2.9: The scene graph for the `Switch` example program. This is similar to the scene graph from the `TransformGroup` example except that `Switch` node has been placed between the `BranchGroup` and the three `TransformGroup` nodes. The `Switch` node can ultimately be used to control which of the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

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

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);

    // 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 `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.

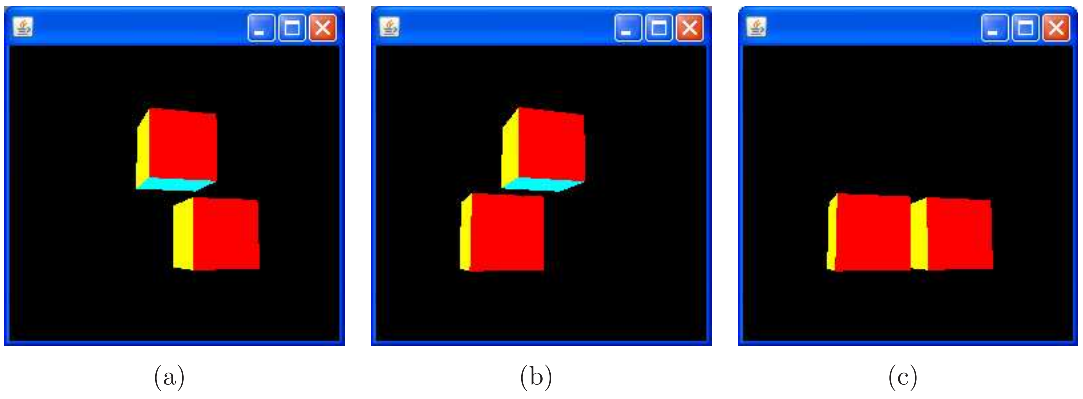
Figure 2.10: Three renderings of the scene graph illustrated in Figure 2.9. In each case the BitSet object that controls which children are to be rendered has a difference configuration: (a) 011, (b) 101 and (c) 110.

- A 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:

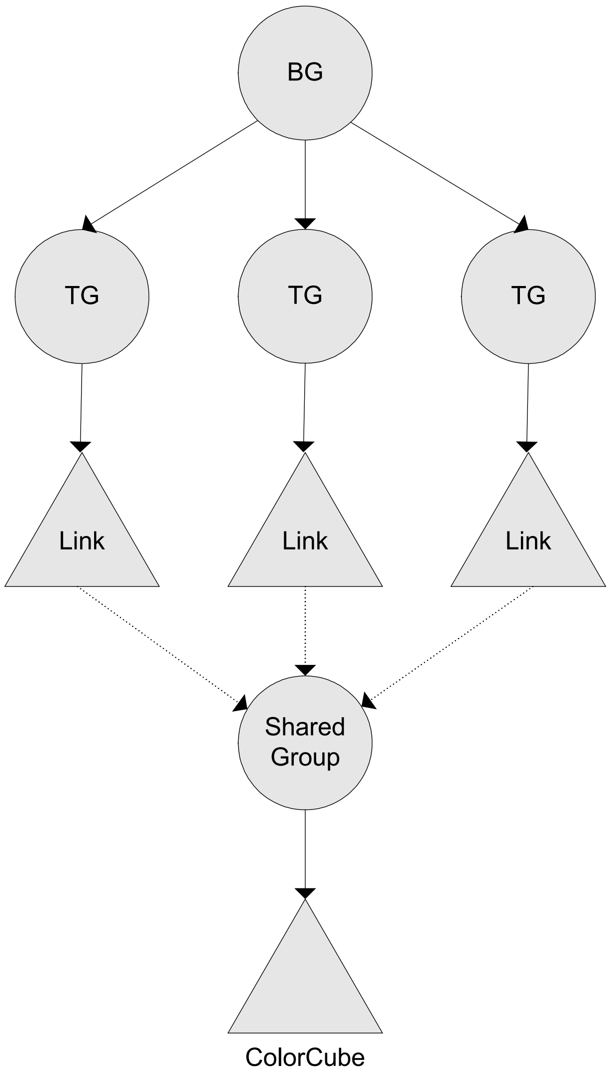

Figure 2.11: The scene graph for the `SharedGroup` example. This builds on the previous `TransformGroup` example, however, instead of having three separate instance of a `ColorCube` object, there is only one. This single instance is referenced three times via a `SharedGroup` object using three `Link`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

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

public class SharedGroupExample extends BasicSceneWithMouseControl{

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

  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:

- `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.

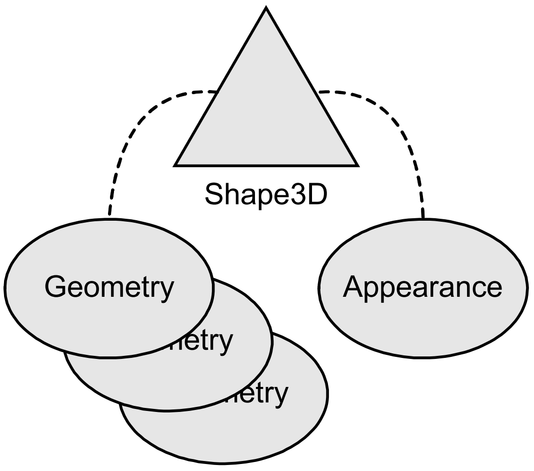

Figure 2.12: An illustration of a Shape3D object. This object maintains a single reference to an `Appearance` object and can maintain multiple references to `Geometry` objects.

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 re-ordering. 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);


    root.addChild(tg);

    return root;
  }
}
```

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

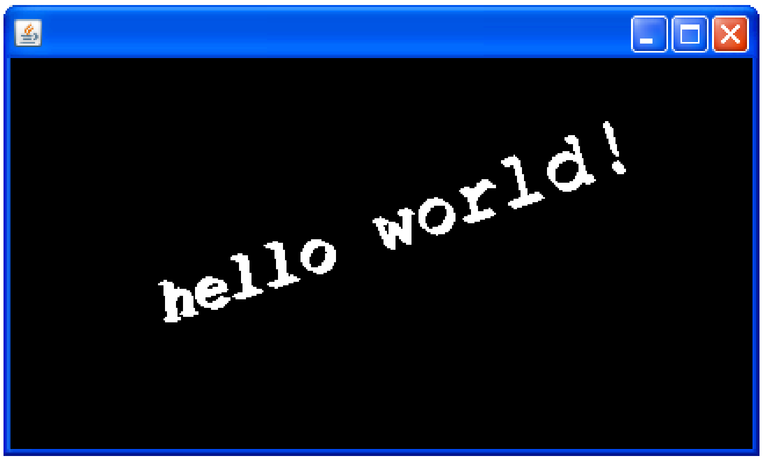

Figure 2.13: The output generated by the `Text3D` example. The string is centre aligned and rendered from left to right.

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:

- **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.

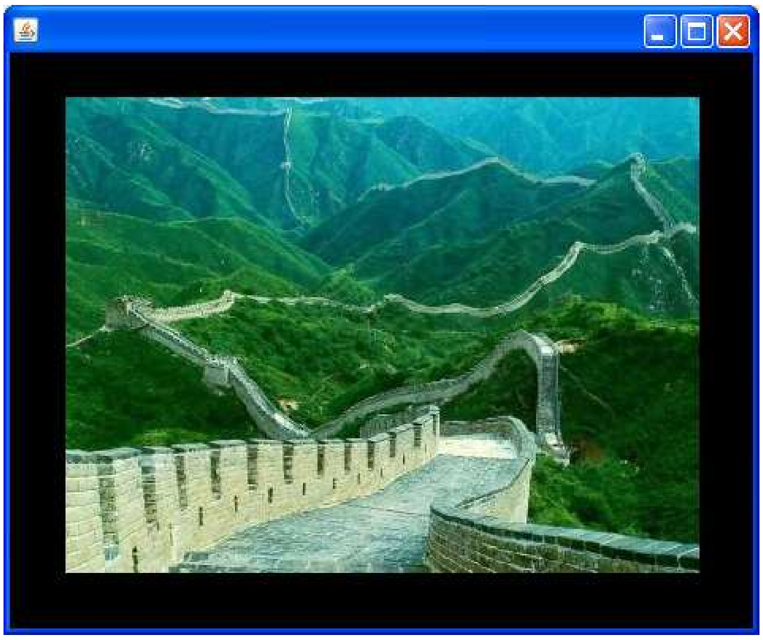

Figure 2.14: The output generated by the `Raster` example. The position of the image has been selected so that it is located in the centre of the frame.

- **By 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.

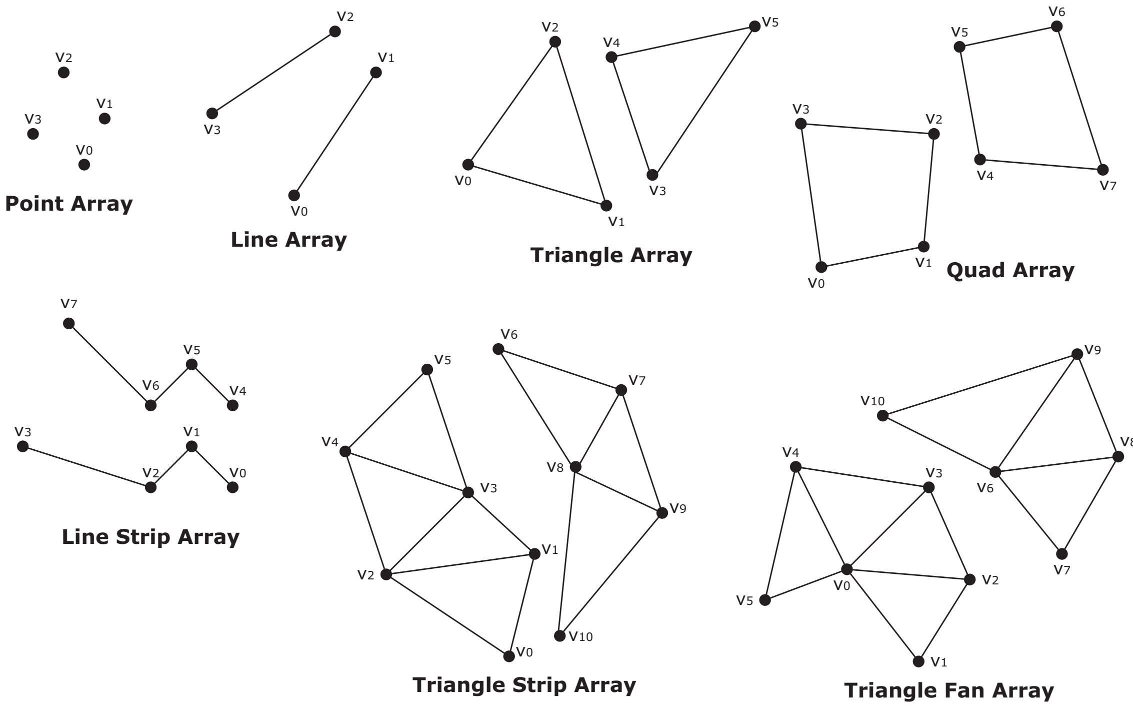

Figure 2.15: An illustration of the seven basic types of geometry supported by the Java 3D API

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 orientation where the vertices of the polygon form an anticlockwise loop. The two possible orientations of a polygon are illustrated in Figure 2.16.

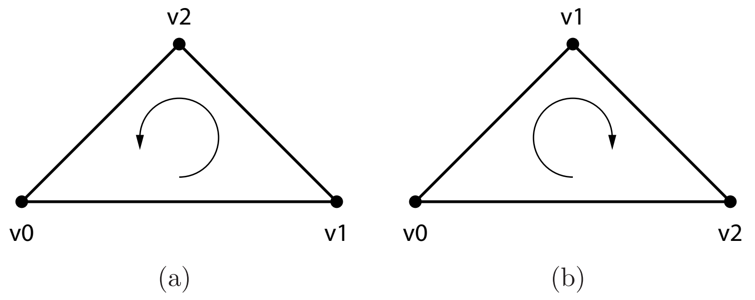

Figure 2.16: Examples of front facing (a) and rear facing (b) triangles.

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.

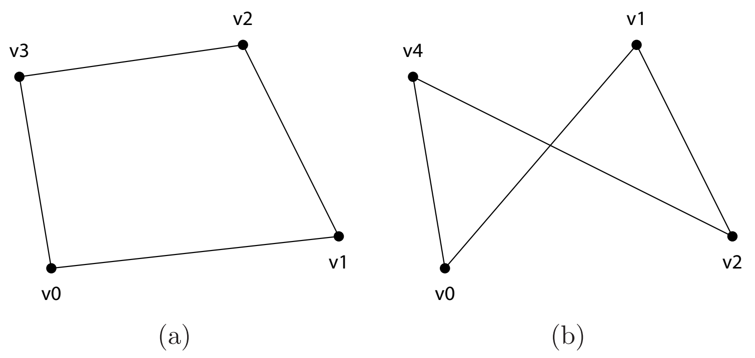

Figure 2.17: Examples of convex (a) and non-convex (concave) (b) quadrilaterals.

2.7 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.

```

import javax.media.j3d.*;

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

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


    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
                           };

    // Create a geometry array from the specified coordinates
    GeometryArray geometryArray = new TriangleArray(3,
        GeometryArray.COORDINATES
        );

    geometryArray.setCoordinates(0, coordinates);
```

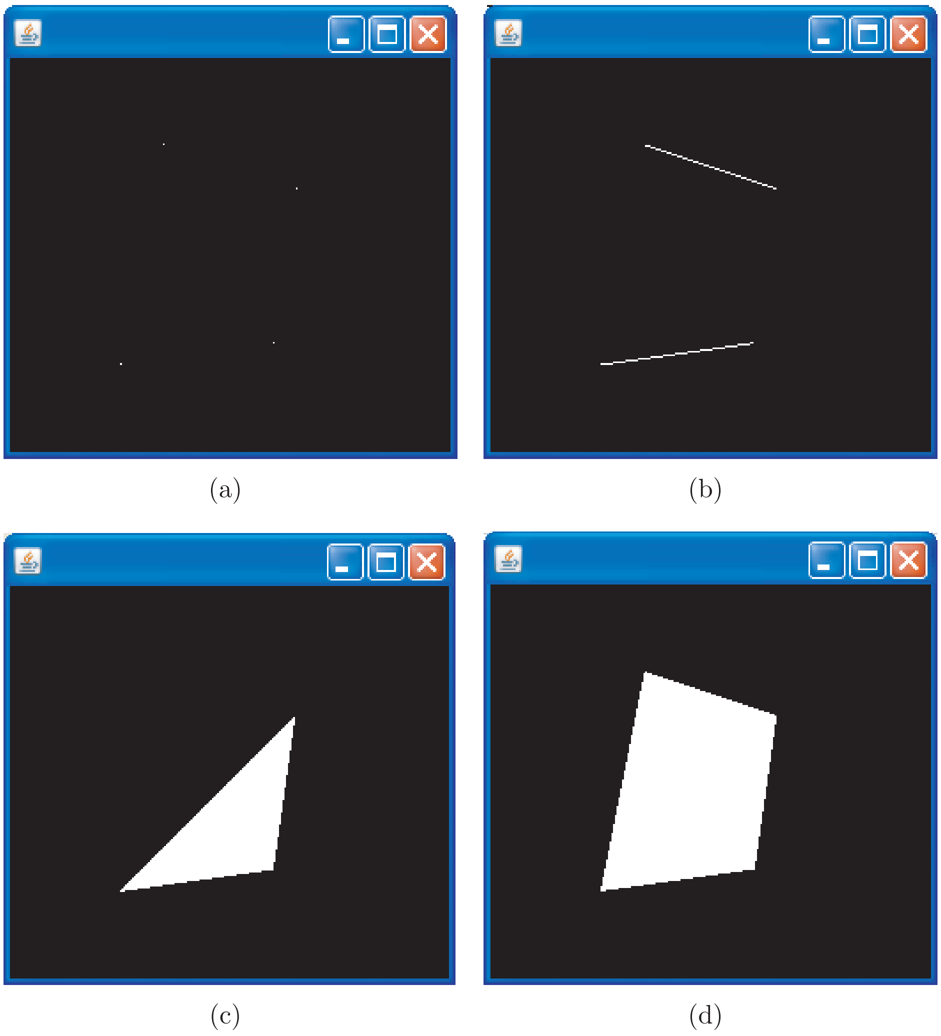

Figure 2.18: Examples of the geometry that can be created using the four simple geometry classes provided by Java 3D (a) A set of points defined using a `PointArray` object (b) A pair of lines defined using a `LineArray` object (c) A triangle defined using a `TriangleArray` object (d) A quadrilateral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

```

    // 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.