### 2.7.12 Behaviours

Behaviours are nodes that makes changes to the scene graph in response to events, such as user input or the passing of time. In other words, behaviours are a general event handling mechanism for Java 3D. A behaviour indicates interest in a set of events called the behaviour's wakeup criterion. When an event occurs that matches the criterion, Java 3D calls the behaviour to process the event. A bounding region must be specified for all behaviours so that they are enabled. The bounding region is set using the following method of the `Behaviour` class:

- `void setSchedulingBounds(Bounds region)`
  Sets the bounding region for this `Behaviour` object to the specified value.

The remainder of this section discusses behaviours that respond to mouse event, behaviours that render an shape at different levels of detail, behaviours that cause a shape to always face the viewer and interpolators that change the property of a node over time.

#### 2.7.12.1 Mouse Behaviours

The location, orientation and scale of a single scene graph node or a group of scene graph nodes can be controlled using a `TransformGroup` object. The transformation associated with the `TransformGroup` is specified using a `Transform3D` object.

Alternatively a subclass of the abstract `MouseBehavior` class can be used to update the transformation associated with a `TransformGroup` object. The subclasses of `MouseBehavior` are:

- `MouseRotate` - lets the user control the rotational component of the transform associated with a `TransformGroup` object using the mouse.
  - Pressing the left mouse button causes the mouse events to be passed to the `MouseRotate` behaviour.
- `MouseTranslate` - lets the user control the translational component of the transform associated with a `TransformGroup` object using the mouse.
  - Pressing the right mouse button causes the mouse events to be passed to the `MouseTranslate` behaviour.
- `MouseWheelZoom` - lets the user control the scale component of the transform associated with a `TransformGroup` object using the mouse wheel.
  - Rotation of the wheel away from the users causes the scale to decrease and rotation of the wheel towards the user cases the scale to increase.
- `MouseZoom` - lets the user control the scale component of the transform associated with a `TransformGroup` object using the mouse.
  - Pressing the centre mouse button causes the mouse events to be passed to the `MouseZoom` behaviour.
  - **Note:** In cases where the mouse has only two buttons, pressing the left mouse button and the ALT key simultaneously causes the mouse events to be passed to the `MouseZoom` behaviour.

These classes are all defined in the `com.sun.j3d.utils.behaviors.mouse` package. All of the four mouse behaviours operate in the same way. The following discussion describes how the `MouseRotate` behaviour operates.

An new instance of a `MouseRotate` object can be created using one of the following constructors:

- `MouseRotate()`
  Create a default `MouseRotate` behaviour that captures mouse events from the `Canvas3D` object associated with the scene graph.

- `MouseRotate(Component c)`
  Creates a `MouseRotate` behaviour that captures mouse events from the specified `Component`.

- `MouseRotate(TransformGroup tg)`
  Creates a `MouseRotate` behaviour that captures mouse events from the `Canvas3D` object associated with the scene graph and updates the specified `TransformGroup` object.

If the `TransformGroup` object that is to be modified is not specified in the constructor for a `MouseBehavior` object then it can be specified using the following method:

- `void setTransformGroup(TransformGroup tg)`
  Sets the `TransformGroup` object updated by the `MouseRotate` behaviour to the specified `TransformGroup` object.

The `MouseRotate` behaviour is an environment node and must have a bound region associated with it so that it is enabled. A suitably constructed bounding region object can be associated with a `MouseRotate` behaviour using the following method:

- `void setSchedulingBounds(Bounds region)`
  Sets the bounding region of the `MouseRotate` behaviour to the specified region.

It is possible to set the rate of rotation caused by the mouse movements. This can be achieved using the following method:

- `void setFactor(double factor)`
  Sets the x-axis and y-axis movement multiplier to the specified value.

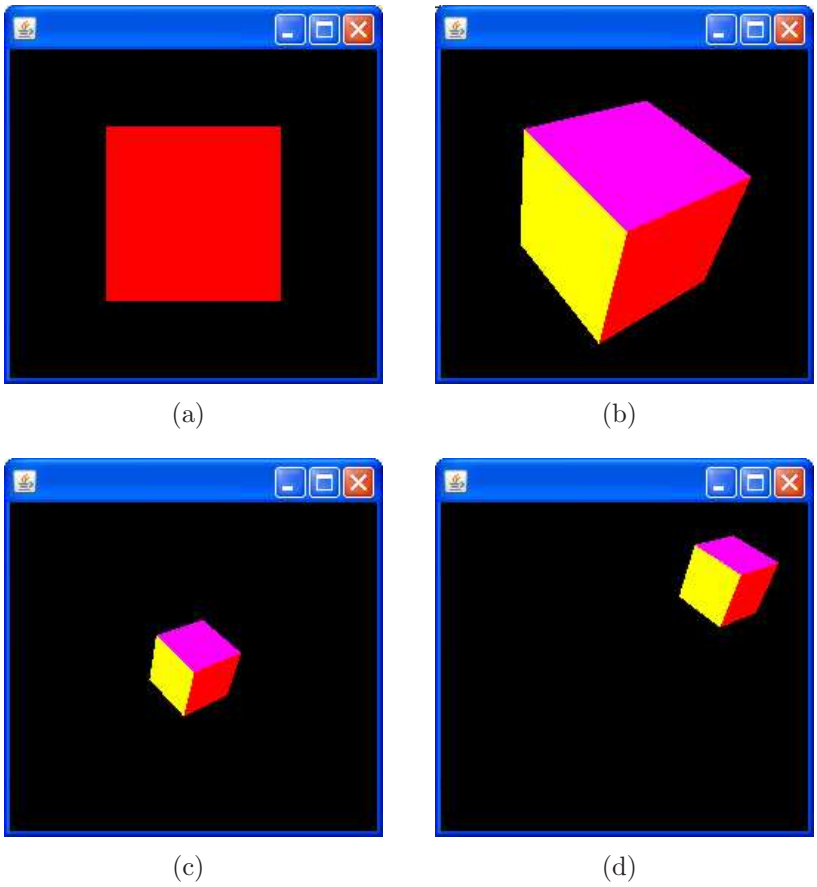
Once a `MouseRotate` behaviour has been constructed and configured it must be attached to the scene graph so that it is active. A mouse behaviour is usually attached to the `TransformGroup` that it updates. This is achieve by calling the `addChild()` method of the `TransformGroup` object.

The following example demonstrates how all four mouse behaviours can be used in conjunction with a single `TransformGroup` object.

```
0
import javax.media.j3d.*;
import javax.vecmath.Point3d;

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

public class MouseBehaviourExample extends BasicScene
{
  public static void main(String args[]){new MouseBehaviourExample();}
10
  public BranchGroup createContentBranch()
  {
    BranchGroup root = new BranchGroup();

15  // Create the TransformGroup that is to be updated
    TransformGroup tg = new TransformGroup();
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg);

20  // Approximate infinite bounds
    BoundingSphere infiniteBounds = new BoundingSphere(new Point3d(),
        Double.POSITIVE_INFINITY);

    // Create the MouseRotate behaviour
25  MouseRotate rotate = new MouseRotate(tg);
    rotate.setSchedulingBounds(infiniteBounds);
    tg.addChild(rotate);

    // Create the MouseZoom behaviour
30  MouseZoom zoom = new MouseZoom(tg);
    zoom.setSchedulingBounds(infiniteBounds);
    tg.addChild(zoom);

    // Create the MouseWheelZoom behaviour
35  MouseWheelZoom wheelZoom = new MouseWheelZoom(tg);
    wheelZoom.setSchedulingBounds(infiniteBounds);
    tg.addChild(wheelZoom);

    // Create the MouseTranslate behaviour
40  MouseTranslate translate = new MouseTranslate(tg);
    translate.setSchedulingBounds(infiniteBounds);
    tg.addChild(translate);

    ColorCube colorCube = new ColorCube(0.4);
45  tg.addChild(colorCube);

    return root;
  }
}
```

This example begins by creating a `TransformGroup` object that is attached to the

root of the scene graph and allows its associated transform to be written after the scene graph has gone live. Then the four mouse behaviours are constructed, configured and added to the `TransformGroup`. Finally, a `ColorCube` object with sides of 80 cm is added to the `TransformGroup`. The scale, orientation and locations of this `ColorCube` will be controlled by different mouse events. Examples of the various outputs of this program are illustrated in Figure 2.46



(a) (b)

(c) (d)

Figure 2.46: Examples of a shape (a) that was rotated (b), scaled (c) and translated (d) using different mouse behaviours.

#### 2.7.12.2 Level of Detail

A level of detail behaviour is used to control the level of detail of a shape based on its distance from the viewer. This allows a high resolution version of the shape to be used when the viewer is close to the shape. Then, as the viewer moves away from the shape lower resolution version of the shape can be used.

Level of detail control is provided by the `DistanceLOD` class which is a subclass of the abstract `LOD` class. Both of these classes are defined in the `javax.media.j3d` package. The `DistanceLOD` behaviour controls a `Switch` group to control which of its children are rendered based on the distance between the `DistanceLOD` node and the viewer.

An array of $n$ monotonically increasing distance values must be specified, such that $distances[0]$ is associated with the highest level of detail and $distances[n-1]$ is

associated with the lowest level of detail. The index of the child of the `Switch` node that is rendered is based on the distance between the `DistanceLOD` and the viewer $d$ is:

- $0, if d \leq distances[0]$
- $i, if distances[i-1] < d \leq distances[i]$
- $n, if d > distance[n-1]$

A `DistanceLOD` behaviour is created using one of the following constructors:

- `DistanceLOD(float[] distances)`
  Creates a `DistanceLOD` object with the specified list of distances that is positioned at the origin.
- `DistanceLOD(float[] distances, Point3f position)`
  Creates a `DistanceLOD` object with the specified list of distances and position.

One of more `Switch` nodes can be associated with the `DistanceLOD` behaviour using the following method:

- `void addSwitch(Switch switch)`
  Appends the specified `Switch` node to the list of `Switch` nodes maintained by this `DistanceLOD` object.
- `void insertSwitch(Switch switch, int index)`
  Inserts the specified `Switch` node into the list of `Switch` nodes maintained by this `DistanceLOD` object at the specified index.
- `void setSwitch(Switch switch, int index)`
  Sets the `Switch` node in the list of `Switch` nodes maintained by this `DistanceLOD` object at the specified index to the specified value.

The following two points should be noted regarding the usage of a `DistanceLOD` behaviour:

1. The `ALLOW_SWITCH_WRITE` capability must be set for the `Switch` group. Otherwise the child mask cannot be updated after the scene graph has gone live.
2. A bounding region must be specified for the `DistanceLOD` behaviour. Otherwise it will be disabled and none of the children of the associated `Switch` node will be displayed.

The following program demonstrates how a `DistanceLOD` behaviour can be used to render different version of a shape based on the distance between the `DistanceLOD` node and the viewer.

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

  import com.sun.j3d.utils.behaviors.mouse.*;
5 import com.sun.j3d.utils.geometry.*;
```

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

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

    // Create the TransformGroup that is to be updated
    TransformGroup tg = new TransformGroup();
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg);

    // Approximate infinite bounds
    BoundingSphere infiniteBounds = new BoundingSphere(new Point3d(),
        Double.POSITIVE_INFINITY);

    // Create the MouseZoom behaviour
    MouseZoom zoom = new MouseZoom(tg);
    zoom.setSchedulingBounds(infiniteBounds);
    tg.addChild(zoom);

    // Create a Switch and enable write operations
    Switch sg = new Switch();
    sg.setCapability(Switch.ALLOW_SWITCH_WRITE);

    // Create a red sphere
    Appearance appRed = new Appearance();
    ColoringAttributes caRed = new ColoringAttributes(
        new Color3f(1.0f, 0.0f, 0.0f),
        ColoringAttributes.SHADE_FLAT);
    appRed.setColoringAttributes(caRed);
    Sphere sphereRed = new Sphere(0.4f, appRed);
    sg.addChild(sphereRed);

    // Create a green sphere
    Appearance appGreen = new Appearance();
    ColoringAttributes caGreen = new ColoringAttributes(
        new Color3f(0.0f, 1.0f, 0.0f),
        ColoringAttributes.SHADE_FLAT);
    appGreen.setColoringAttributes(caGreen);
    Sphere sphereGreen = new Sphere(0.4f, appGreen);
    sg.addChild(sphereGreen);

    // Create a blue sphere
    Appearance appBlue = new Appearance();
    ColoringAttributes caBlue = new ColoringAttributes(
        new Color3f(0.0f, 0.0f, 1.0f),
        ColoringAttributes.SHADE_FLAT);
    appBlue.setColoringAttributes(caBlue);
    Sphere sphereBlue = new Sphere(0.4f, appBlue);
    sg.addChild(sphereBlue);
```

```
60      // Create DistanceLOD behaviour
        float [] distances = {4.0f, 8.0f};
        DistanceLOD lod = new DistanceLOD(distances);
        lod.setSchedulingBounds(infiniteBounds);
        lod.addSwitch(sg);
65
        tg.addChild(sg);
        tg.addChild(lod);

        return root;
70    }
}
```

This program begins by creating a `TransformGroup` object that allows its associated transform to be written to after the scene graph has gone live. A `MouseZoom` behaviour is then attached to this `TransformGroup`. Three shapes are subsequently attached to a `Switch` node which is in turn attached to the `TransformGroup`, these are: red, green and blue spheres all 40 cm in diameter. A `DistanceLOD` behaviour is then created, associated with the `Switch` group and added to the `TransformGroup`. This causes:

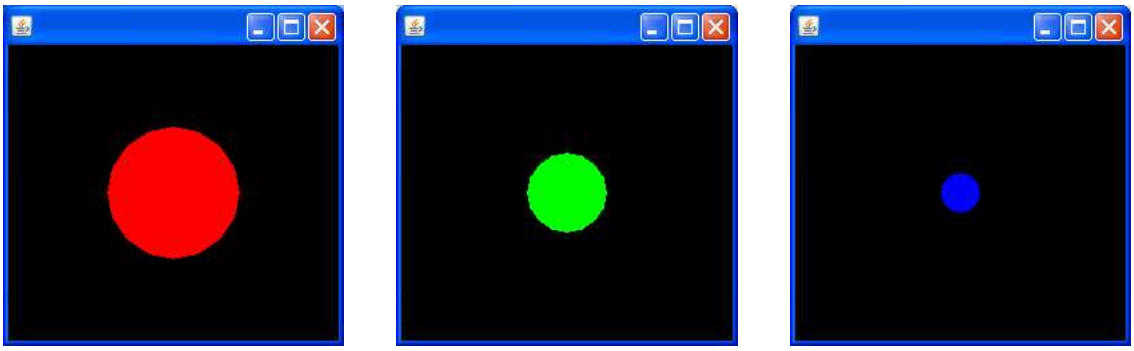
- This first child of the `Switch` group (i.e. the red sphere) to be displayed if the distance between the viewer and the `DistanceLOD` is $\leq$ 4 metres.
- The second child of the `Switch` group (i.e. the green sphere) to be displayed if the distance between the viewer and the `DistanceLOD` is $>$ 4 metres but $\leq$ 8 metres.
- The third child of the `Switch` group (i.e. the blue sphere) to be displayed if the distance between the viewer and the `DistanceLOD` is $>$ 8 metres.

It should be noted that in this example different coloured shapes were used rather than different resolution shapes. This done in order to highlight the operation of the `DistanceLOD` behaviour. Examples of the output generated by this program are illustrated in Figure 2.47.

#### 2.7.12.3 `Billboard`

The `Billboard` behaviour node operates on a `TransformGroup` node to cause the local +z axis of the `TransformGroup` to point at the viewer's eye position. This is done regardless of the transforms above the specified `TransformGroup` node in the scene graph. Two alignment modes are supported by the `Billboard` behaviour, these are:

- `ROTATE_ABOUT_AXIS`
  Causes the associated `TransformGroup` to rotate about the specified axis.
- `ROTATE_ABOUNT_POINT`
  Causes the associated `TransformGroup` to rotate about the specified point.



(a) (b) (c)

Figure 2.47: A `Switch` group rendering a different child based on the distance between a `DistanceLOD` behaviour and the viewer. A red sphere is rendered when the viewer is $\leq 4$ metres away (a). A green sphere is rendered when the viewer is $> 4$ metres away but $\leq 8$ metres away (b). Finally a blue sphere is rendered if the viewer is $> 8$ metres away.

`Billboard` behaviours are ideal for drawing screen aligned-text of for drawing roughly symmetrical objects. A typical use might consist of a quadrilateral that contains a tree structure. A `Billboard` behaviour can be created using one of the following constructors:

- `Billboard(TransformGroup tg, int mode, Point3f point)`
  Creates a `Billboard` behaviour with the specified rotation point and mode that operates on the specified `TransformGroup`.
- `Billboard(TransformGroup tg, int mode, Vector3f axis)`
  Creates a `Billboard` behaviour with the specified axis and mode that operates on the specified `TransformGroup`.

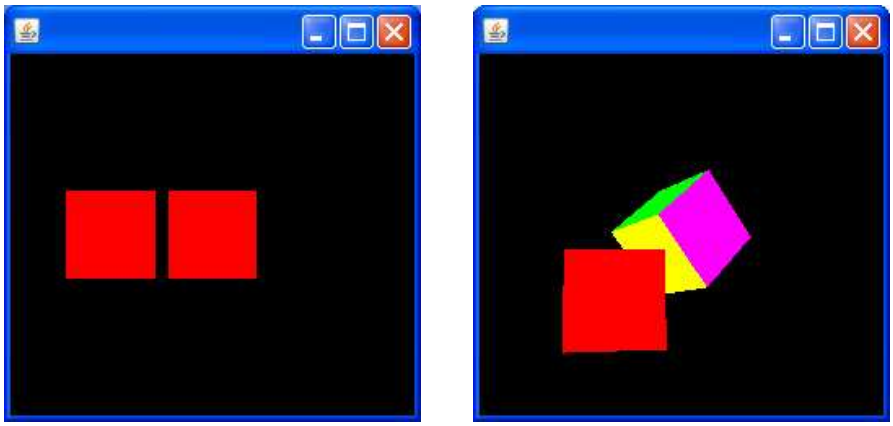
It should be noted that the `OrientatedShape3D` node provides the same kind of functionality as the `Billboard` behaviour, except that only a single `Shape3D` object is affected. `OrientatedShape3D` is generally faster than `Billboard` and should be used where possible.

The following example demonstrates how a `Billboard` behaviour can be used.

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

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

public class BillboardExample extends BasicScene
{
  public static void main(String args[]){new BillboardExample();}
10
  public BranchGroup createContentBranch()
  {
```

```
    BranchGroup root = new BranchGroup();

    // Create the TransformGroup for the MouseRotate
    TransformGroup tg1 = new TransformGroup();
    tg1.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg1);

    tg1.addChild(new ColorCube(0.2));

    // Approximate infinite bounds
    BoundingSphere infiniteBounds = new BoundingSphere(new Point3d(),
        Double.POSITIVE_INFINITY);

    // Create the MouseRotate behaviour
    MouseRotate rotate = new MouseRotate(tg1);
    rotate.setSchedulingBounds(infiniteBounds);
    tg1.addChild(rotate);

    // Create the TransformGroup for the translation
    Transform3D trans = new Transform3D();
    trans.setTranslation(new Vector3d(-0.5f, 0.0f, 0.0f));
    TransformGroup tg2 = new TransformGroup(trans);
    tg2.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg1.addChild(tg2);

    // Create the TransformGroup for use with the Billboard
    TransformGroup tg3 = new TransformGroup();
    tg3.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    tg2.addChild(tg3);

    tg3.addChild(new ColorCube(0.2));

    Billboard billboard = new Billboard(tg3,
        Billboard.ROTATE_ABOUT_POINT,
        new Point3f(0.0f, 0.0f, 0.0f));
    billboard.setSchedulingBounds(infiniteBounds);
    tg3.addChild(billboard);

    return root;
  }
}
```

The program creates a hierarchy that consists of three `TransformGroup` objects. The first `TransformGroup` is associated with a `MouseRotate` behaviour and has a `ColorCube` child and a `TransformGroup` child. The second `TransformGroup` is associated with a translation 50 cm left of the origin and has a `TransformGroup` child. This final `TransformGroup` has a `ColorCube` child and is associated with a `Billboard` behaviour so that its child is always orientated towards the viewer. The types of output generated when this program is executed are illustrated in Figure 2.48.



(a) (b)

Figure 2.48: The initial output of the `BillboardExample` program (a) and an example of the output after the scene has been rotated slightly. Note that the `ColorCube` that was originally on the left is still facing towards the viewer.

#### 2.7.12.4 Interpolators

Java 3D provides support for a range of behaviours that implement some type of interpolation. Interpolators are used to change an attribute of a node over time. The core interpolation functionality is defined in the abstract `Interpolator` base class. The types of interpolation that are defined by the subclasses of this class include:

- `ColorInterpolator` - This class defines a behaviour that modifies the ambient, emissive, diffuse, or specular colour of its target `Material` object by linearly interpolating between a pair of specified colours.
- `TransparencyInterpolator` - This class defines a behaviour that modifies the transparency of its target `TransparencyAttributes` object by linearly interpolating between a pair of specified transparency values.
- `SwitchValueInterpolator` - This class defines a behaviour that modifies the selected child of the target `Switch` node by linearly interpolating between a pair of specified child index values.
- `TransformInterpolator` - This is an abstract class that extends `Interpolator` to provide common methods used by various transform related interpolator subclasses.

`PositionInterpolator`

One example of a subclass of `TransformInterpolator` is `PositionInterpolator`. This class defines a behaviour that modifies the translation component of its target `TransformGroup` by linearly interpolating between a pair of specified positions. The interpolated position is used to generate a translation transform along the local X-axis of this interpolator. An instance of a `PositionInterpolator` can be created using one of the following constructors:

- `PositionInterpolator(Alpha alpha, TransformGroup target)`
  Constructs a position interpolator with a specified target, an axis-of-translation

set to the identity transformation, a start position of 0.0f and an end position of 1.0f.

- `PositionInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfTransform, float start, float finish)`
  Constructs a position interpolator with a specified target, a specified axis-of-translation, a specified start position and a specified end position.

Both of these constructors also require a `Alpha` object to be passed as an argument. An `Alpha` object is a node component that provides common methods for converting a time value into an alpha value (i.e. a value in the range 0 to 1). Some of the attributes defined by the `Alpha` alpha class are as follows:

- **Loop count -** This is the number of times to run this `Alpha`. A value of -1 indicates that the `Alpha` loops indefinitely.
- **Trigger time -** This is the time in milliseconds since the start time that this object first triggers. If the start time plus the trigger time is $\geq$ the current time, then the `Alpha` starts running.
- **Increasing alpha duration -** This is the period of time during which the `Alpha` object transitions from zero to one.

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

- `Alpha(int loopCount, long increasingAlphaDuration)`
  Creates an `Alpha` object that will loop for the specified number of durations where each loop lasts for the specified duration in milliseconds.

The attributes of the constructed `Alpha` object can then be access using the relevant accessor methods, for example the trigger time can be set or retrieved using the following methods:

- `void setTriggerTime(long time)`
  Sets the trigger time to the specified value, e.g. a value of 4000 would cause the `Alpha` to start iterating four seconds after the application was launched.
- `long getTriggerTime()`
  Retreves the trigger time in milliseconds associated with this alpha object.

The following example demonstrates how a `PositionInterpolator` can be used to move an object from one location to another, a specified number of times, over a specified period.

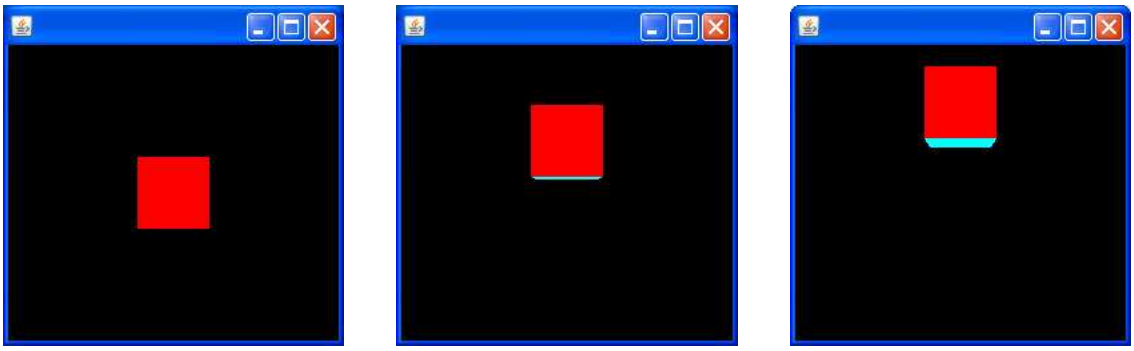
```
0
import javax.media.j3d.*;
import javax.vecmath.*;

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

public class PositionInterpolatorExample extends BasicScene
{
```

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

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

    // Create the TransformGroup associated with the PositionInterpolator
    TransformGroup tg = new TransformGroup();
    tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg);

    // Add a ColorCube child
    tg.addChild(new ColorCube(0.2));

    // Create an alpha that will start after two seconds, loop four times
    // where each loop lasts 1 second
    Alpha alpha = new Alpha(4, 1000);
    alpha.setTriggerTime(2000);

    // A Transform that rotates the x axis onto the y axis
    Transform3D transform = new Transform3D();
    transform.setRotation(new AxisAngle4f(0.0f, 0.0f, 1.0f,
        (float)(Math.PI/2.0)));

    // Create the positional interpolator to move the ColorCube
    // between 0.0 and 0.5 on the y-axis
    PositionInterpolator pi = new PositionInterpolator(alpha,
        tg, transform, 0.0f, 0.5f);
    pi.setSchedulingBounds(new BoundingSphere(new Point3d(),
        Double.POSITIVE_INFINITY));
    tg.addChild(pi);

    return root;
  }
}
```

The program begins by creating a `TransformGroup` object that will ultimately be updated by a `PositionInterpolator` behaviour. A `ColorCube` with sides 40 cm in length is added to the `TransformGroup`. An `Alpha` is then created with a loop count of 4, a loop duration of 1 second and a trigger time 2 seconds after the application starts. A `Transform3D` is then created that transforms the x-axis onto the y-axis. Finally, a `PositionInterpolator` is created that updates the `TransformGroup` and moves the `ColorCube` from 0.0 to 0.5 along the y-axis. Examples of the output renderings obtained from this program are illustrated in Figure 2.49.

### Spline Path Interpolator

A B-spline curve is a smooth path that is defined by a series of control points and blending functions. The origin of B-splines relates to industries such as ship building. where a designer was required to draw a life-size curves representing, for example, the cross-section through the hull of a ship.



(a) (b) (c)

Figure 2.49: An example of the output generated by the PositionInterpolator example. The `ColorCube` is initially positioned at the original. Its position is then interpolated along the positive y-axis (b) to y = 0.5 (c).

For small scale drawings draughtsmen would use French curves[2]. They would draw complete curves by putting together segments formed from different parts of different French curves. For full-scale plans this method was completely impractical and the draftsmen would employ long thin strips of metal. These were pushed into the required shape and secured using lead weights called ducks (see Figure 2.50). These ducks are analogous to the control points for a B-spline.

Figure 2.50: An example of how lead weights known as ducks can be used to generate a curved shape from a straight rod.

This is the physical basis for B-splines. The metal shape takes up a shape that minimises its internal strain. In addition, the effect of a duck is local and the shape of the curve is only altered in its vicinity.

A B-spline curve does not pass through its control points. It is a complete piecewise cubic polynomial consisting of any number of curve segments. The B-spline formulation is defined as follows:

[2]French curves are a set of small, flat preformed curve sections

$$Q_i(u) = UB_sP \tag{2.3}$$

Or alternatively in matrix notation:

$$Q_i(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \frac{1}{6} \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix} \begin{bmatrix} P_{i-3} \\ P_{i-2} \\ P_{i-1} \\ P_i \end{bmatrix} = \tag{2.4}$$

where $Q_i$ is the $i$th B-spline segment and $P_i$ is a set of four points in a sequence of control points. The value for u over a single curve segment is $0 \leq u \leq 1$. Using this notation $u$ represents a local parameter, locally varying over the parametric range 0 to 1 to define a single B-spline curve segment.

It is clear that a B-spline curve is a series of $m-2$ curve segments that are labeled $Q_3, Q_4, ..., Q_m$ defines or determined by $m+1$ control points $P_0, P_1, ..., P_m, m \geq 3$. Each curve segment is defined by four control points and each control point influences four and only four curve segments. An example of a B-spline curve is illustrated in Figure 2.51.

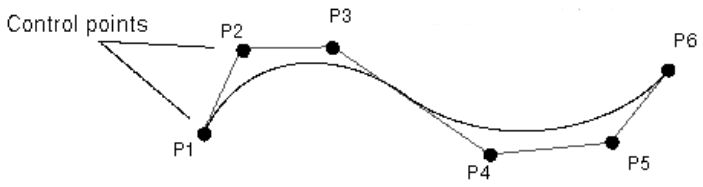



Figure 2.51: An example of a B-spline curve with 6 control points,

Java 3D provides support for a variation on B-splines known as Kochanek-Bartels cubic splines. These types of spline are also known as a TCB splines as they have configurable tension, continuity and bias characteristics. Unlike B-splines, TCB splines do go through the control points.

The `RotPosScaleTCBSplinePathInterpolator` class defines a behaviour that varies the rotational, translational and scale components of its target `TransformGroup` using Kochanek-Bartels cubic spline interpolation to interpolate among a series of key frames using the value generated by a specified `Alpha` object. An object of this class can be constructed as follows:

- `RotPosScaleTCBSplinePathInterpolator(Alpha alpha, TransformGroup target, Transform3D axisOfTransform, TCBKeyFrame[] keys)`
  Constructs a new `RotPosScaleTCBSplinePathInterpolator` object that varies the rotation, translation and scale of the transformation associated with the target `TransformGroup`.

The `TCBKeyFrame[]` argument defines a list of key points and their associated attributes. A `TCBKeyFrame` object can be created using the following constructor:

- `TCBKeyFrame(float k, int l, Point3f pos, Quat4f q, Point3f s, float t, float c, float b)`
  Creates a key frame with the specified attributes:

- `k` - Defines the knot value. The first knot must have a value of 0.0. The last knot must have a value of 1.0. An intermediate knot with index k must have a value strictly greater than any knot with index less than k.
- `l` - Indicate whether to use linear (1) or spline (0) interpolation.
- `pos` - The position of the key frame.
- `q` - The rotation at the key frame.
- `s` - The scale at the key frame.
- `t` - The tension at the key frame ($-1.0 < t < 1.0$).
- `c` - The continuity at the key frame ($-1.0 < c < 1.0$).
- `b` - The bias at the key frame ($-1.0 < b < 1.0$).

The follow example demonstrates how a `RotPosScaleTCBSplinePathInterpolator` behaviour can be used to generate a spline interpolated path along a series of control points.

```
0
import javax.media.j3d.*;
import javax.vecmath.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.behaviors.interpolators.*;
5
public class TCBSplineExample extends BasicScene
{
  public static void main(String args[]){new TCBSplineExample();}

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

    TransformGroup tg = new TransformGroup();
15  tg.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
    root.addChild(tg);

    ColorCube colorCube = new ColorCube(0.1);
    tg.addChild(colorCube);
20
    TCBKeyFrame[] keyFrame = new TCBKeyFrame[5];

    // Create the first key frame at (0.5, 0.5)
    keyFrame[0] = new TCBKeyFrame(0.0f,
25      0,
        new Point3f(0.5f,0.5f,0.0f),
        new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
        new Point3f(1.0f, 1.0f, 1.0f),
        0.0f, 0.0f, 0.0f);
30
    // Create the second key frame at (-0.5, 0.5)
    keyFrame[1] = new TCBKeyFrame(0.25f,
        0,
        new Point3f(-0.5f,0.5f,0.0f),
```

```
        new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
        new Point3f(1.0f, 1.0f, 1.0f),
        0.0f, 0.0f, 0.0f);


    // Create the thrid key frame at (-0.5, -0.5)
    keyFrame[2] = new TCBKeyFrame(0.50f,
        0,
        new Point3f(-0.5f,-0.5f,0.0f),
        new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
        new Point3f(1.0f, 1.0f, 1.0f),
        0.0f, 0.0f, 0.0f);


    // Create the fourth key frame at (0.5, -0.5)
    keyFrame[3] = new TCBKeyFrame(0.75f,
        0,
        new Point3f(0.5f,-0.5f,0.0f),
        new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
        new Point3f(1.0f, 1.0f, 1.0f),
        0.0f, 0.0f, 0.0f);


    // Create the final key frame at (0.0, 0.0)
    keyFrame[4] = new TCBKeyFrame(1.0f,
        0,
        new Point3f(0.0f,0.0f,0.0f),
        new Quat4f(0.0f, 0.0f, 0.0f, 0.0f),
        new Point3f(1.0f, 1.0f, 1.0f),
        0.0f, 0.0f, 0.0f);

    // Create an Alpha that loops indefinitely for
    // a duration of 4 seconds
    Alpha alpha = new Alpha(-1, 4000);

    // Create the TCB spline path interpolator
    RotPosScaleTCBSplinePathInterpolator i =
      new RotPosScaleTCBSplinePathInterpolator(alpha,
          tg, new Transform3D(),keyFrame);
    BoundingSphere s = new BoundingSphere(new Point3d(),
        Double.POSITIVE_INFINITY);
    i.setSchedulingBounds(s);
    tg.addChild(i);

    return root;
  }
}
```

The program begins by creating a `TransformGroup` whose associated transformation will ultimately be controlled by a TCB spline path interpolator. The child of the `TransformGroup` is a `ColorCube` with sides 20 cm in length.

A total of five key frames are defined. Each key frame has a unique position and knot value and uses spline interpolation. None of the key frames affect the rotation or scale of the transform group and the tension, continuity and bias are all set to 0.

A `RotPosScaleTCBSplinePathInterpolator` object is created using an `Alpha` object that loops infinitely with a period of 4 seconds, the `TransformGroup` that is to be modified, the identity transform and the array of key frames. The bounds for the interpolator are set to approximate infinite bounds and the interpolator is added to the scene graph. The path taken by the `ColorCube` under the control of the TCB spline path interpolator is illustrated in Figure 2.52.

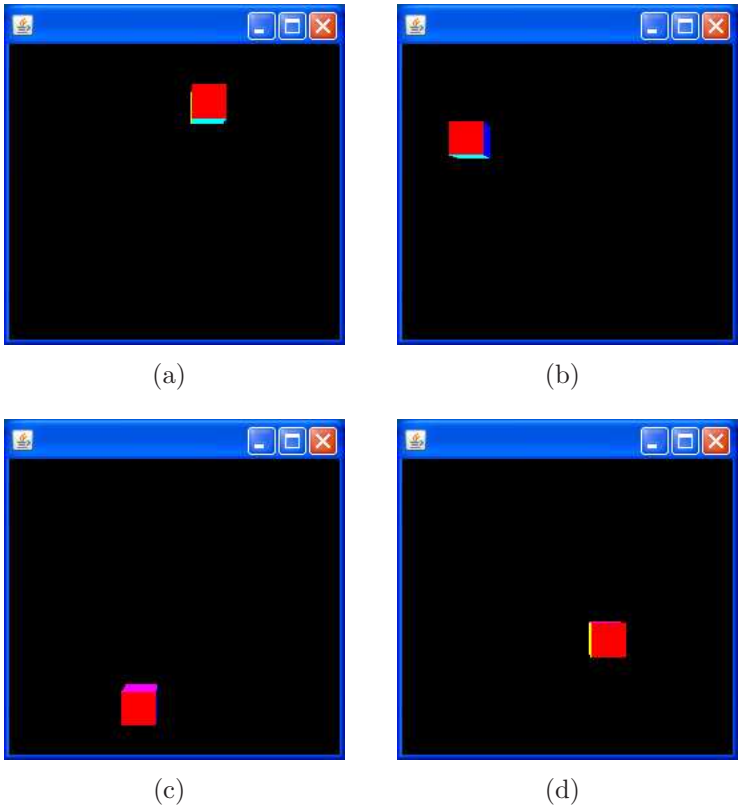



Figure 2.52: A illustration of the path taken by the `ColorCube` under control of the TCB spline path interpolator.

### 2.7.13 Picking

Picking is essentially the opposite operation to viewing. It enables the selection of a specific shape by projecting 2D screen coordinates into the virtual world and identifying the shape associated with the coordinates.

The `PickCanvas` class is used to turn the mouse coordinates into an area of space or a `PickShape`, that projects from the viewer through the mouse location into the virtual world. The `PickCanvas` class extends a more general `PickTool` class that defines basic picking operations.

When a pick is requested, Java 3D figures out the pickable shapes that intersect with the `PickShape`. These shapes are stored in a list of `PickResult` objects.

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

- `PickCanvas(Canvas3D canvas, BranchGroup b)`
  Creates a `PickCanvas` object that monitors the specified `Canvas3D` object for mouse events and uses these events to pick objects from the specified `BranchGroup` rooted scene graph.

A range of methods are available for configuring the attributes of a `PickCanvas` object, these include:

- `void setMode(int mode)`
  Sets the picking detail mode for this `PickCanvas` object.
- `void setTolerance(float tolerance)`
  Sets the picking tolerance. Objects within this distance in pixels to the mouse x,y location will be picked. The default tolerance is 2.0 pixels.

If a mouse click occurs on the canvas then the list of shapes that were picked can be obtained using the following method:

- `PickResult[] pickAll()`
  Results an array that represents all of the nodes that intersect with the pick location.

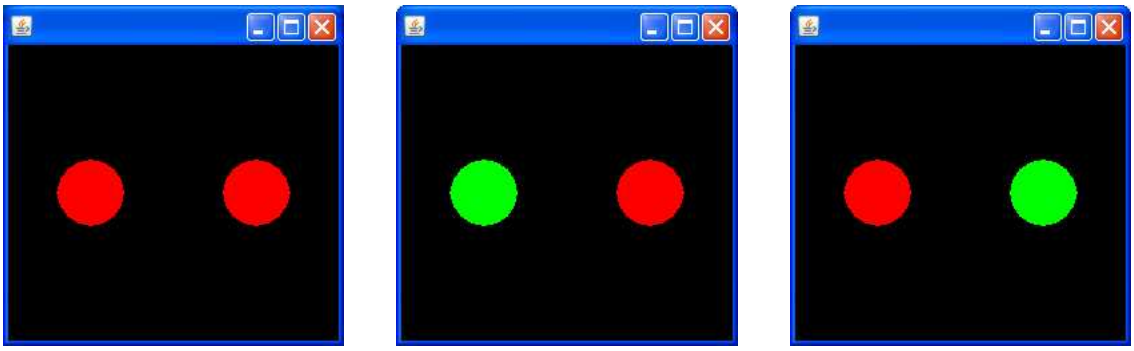
The following examples demonstrates how the `PickCanvas` and its associated classes can be used to select objects from a 3D scene using the mouse.

```
0
import java.awt.event.*;
import javax.media.j3d.*;

import com.sun.j3d.utils.picking.*;
5 import com.sun.j3d.utils.geometry.*;
import javax.vecmath.*;

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

   PickCanvas pickCanvas;
   Appearance redAppearance;
   Appearance greenAppearance;
15   Shape3D rightShape;
   Shape3D leftShape;

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

     // Create and configure the PickCanvas
     pickCanvas = new PickCanvas(canvas, root);
     pickCanvas.setMode(PickTool.GEOMETRY);
```

```
    pickCanvas.setTolerance(4.0f);

    // Create a red appearance
    redAppearance = new Appearance();
    ColoringAttributes red = new ColoringAttributes(
        new Color3f(1.0f, 0.0f, 0.0f),
        ColoringAttributes.SHADE_FLAT);
    redAppearance.setColoringAttributes(red);

    // Create a green appearance
    greenAppearance = new Appearance();
    ColoringAttributes green = new ColoringAttributes(
        new Color3f(0.0f, 1.0f, 0.0f),
        ColoringAttributes.SHADE_FLAT);
    greenAppearance.setColoringAttributes(green);

    // Create a TransformGroup whose children will be located
    // 50 cm to the left of the origin
    Transform3D left = new Transform3D();
    left.setTranslation(new Vector3f(-0.5f, 0.0f, 0.0f));
    TransformGroup leftGroup = new TransformGroup(left);
    root.addChild(leftGroup);

    // Add a red sphere to the left TransformGroup
    Sphere leftSphere = new Sphere(0.2f, redAppearance);
    leftShape = leftSphere.getShape();
    leftShape.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);
    leftGroup.addChild(leftSphere);

    // Create a TransformGroup whose children will be located
    // 50 cm to the right of the origin
    Transform3D right = new Transform3D();
    right.setTranslation(new Vector3f(0.5f, 0.0f, 0.0f));
    TransformGroup rightGroup = new TransformGroup(right);
    root.addChild(rightGroup);

    // Add a red sphere to the right TransformGroup
    Sphere rightSphere = new Sphere(0.2f, redAppearance);
    rightShape = rightSphere.getShape();
    rightShape.setCapability(Shape3D.ALLOW_APPEARANCE_WRITE);
    rightGroup.addChild(rightSphere);

    return root;
  }

  public void mouseClicked(MouseEvent me)
  {
    // Set the colour of both spheres to red
    leftShape.setAppearance(redAppearance);
    rightShape.setAppearance(redAppearance);

    // Get the picked nodes
    pickCanvas.setShapeLocation(me);
```

```
    PickResult[] results = pickCanvas.pickAll();

    if(results != null)
      for(int r=0; r<results.length; r++)
      {
        // Set the colour of the picked shape to green
        PickResult result = results[r];
        Shape3D sphere = (Shape3D)result.getObject();
        sphere.setAppearance(greenAppearance);
      }
  }
}
```

This program begins by associating a `PickCanvas` with the main `Canvas3D` of the application. Then two red spheres with radius 20cm are created and positioned 50 cm left and right of the origin. When the mouse is clicked the colour of both of the spheres is set to red. The list of picked nodes is obtained and the colour of each picked shape is set to green. Examples of the renderings obtained when this program is executed are illustrated in Figure 2.53



(a) (b) (c)

Figure 2.53: The output generated by the picking example. Intially both of the spheres are coloured red (a). When the user clicks on the left sphere its colour changes to green (b). Then when the user clicks on the right sphere its colour changes to green and the colour of the left sphere changes back to red (c).

## 2.8 Summary

This chapter has provided a detailed discussion of the Java 3D API. The concept of scene graphs forms the basis for the organisation of content in a 3D scene. The scene graph can contain group nodes, leaf nodes and node components. Relationships and references can be created to link the various scene graph elements in order to create a coherent structure.

The group nodes in the scene graph allow other nodes to be grouped together. In some cases the group nodes implemented some kind of functionality, for example conditional rendering of the groups children, or implementing a specified transformation that is applied to all the children of the group.

The main visible content that can be contained in a scene graph is represented by a `Shape3D` object. A `Shape3D` object is essentially a container for an appearance and one or possibly more geometries.

The `Appearance` node component defines a series of attributes that indicate how a shape appears in the rendered scene. These attributes define things like how the shape appears when there is no light, how different types of light affect the appearance of the shape, whether the shape is to be rendered using points, lines or polygons and whether texture mapping is to be used in conjunction with the shape.

The `Geometry` node component essentially defines the structure of the shape. At the most basic level the geometry can be a set of point, lines or polygons. Several different approaches to polygon definition are also available to optimise the way that geometry can be defined. These include fan and strip arrays of triangles. Indexed arrays of vertices can also be used to reduce the amount of repetition that occurs when defining adjoining polygons.

A series of environment nodes are also defined. These nodes are used to determine different aspects of the environment. For example, they can be used to define a background image or fog in order to add realism to a scene. A series of behaviours are also defined. These enable the scene to react to various situations, for example mouse events or the passage of time.

It is evident from the material discussed in this chapter that Java 3D is a comprehensive, fully featured, 3D graphics API that allow the programmer to develop 3D content at a high level, i.e. it allows the developer to focus on what to render and not how it is rendered.

# Chapter 3

# Surface Extraction

Modern medical imaging modalities typically generate 3D volumetric data that represents the characteristics at each point in the scanned region. Example of medical imaging modalities that generate volumetric data include:

- Computed Tomography (CT)
- Magnetic Resonance Imaging (MRI)
- Single Photon Emission Computed Tomography (SPECT)
- Positron Emission Tomography (PET)

Examples of images obtained from an abdominal CT study of a patient are illustrated in Figure 3.1.

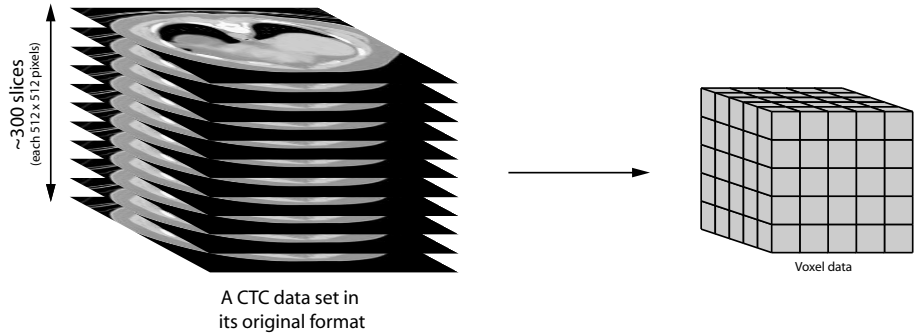



Figure 3.1: A series of slices obtained from an abdominal CT study of a patient. Although the data is obtained as a series of slices it essentially represents a volumetric data set consisting of voxels.

The images in this figure are represented as 2D slices. However, they actually represent thin volumetric regions with a thickness of approximately 1.5 mm. Consequently, the series of images represents a volumetric data set consisting of individual voxels. In the case of CT, the value of a voxel represents the density at a particular point in the scanned region.

If the volumetric data contains continuous isosurfaces, then these surfaces can be explicitly extracted and converted into a polygonal mesh. The resulting mesh can be rendered using the methods outlined in the previous chapter. The techniques used to extract an isosurface from a volumetric data sets is known as the marching cubes algorithm and was originally reported by Lorenson and Cline in 1987.

- Lorensen, W. E. & Cline, H. E. (1987), 'Marching cubes: A high resolution 3D surface construction algorithm', *Computer Graphics* **41**(4), 163-169.

This chapter will provide an overview of the standard marching cubes algorithm. A series of enhancements are then introduced that enhance the operation of the standard algorithm. The marching cubes algorithm is described in relation to a specific area of medical imaging known as virtual endoscopy (specifically virtual colonoscopy).

## 3.1 The Standard Marching Cubes Algorithm

The marching cubes algorithm (MCA) is used to extract a surface represented by an isosurface value ($d_{iso}$) from a volumetric data set. The value of $d_{iso}$ is dependent on the surface being extracted and in the case of the colon surface $d_{iso}$ has a value of -800 HU[1].

The MCA begins by thresholding the data set, assigning a 1 to voxels $\geq d_{iso}$ (inside the isosurface) and a 0 to voxels $< d_{iso}$ (outside the isosurface). The isosurface associated with the colonic mucosa cannot be uniquely identified using a simple threshold operation due to the the number of gas/soft tissue interfaces that are present in a CTC data set (i.e. those due to the lung bases, the small intestine, the stomach and the exterior of the patient). In this case, segmentation information is used in conjunction with the isosurface value to identify the region associated with the colon surface.

A cubic mask of size $2 \times 2 \times 2$ is then passed through the volume and at each mask location the configuration of the eight underlying voxels is examined and the relevant surface patches are generated. This process is illustrated in Figure 3.2.

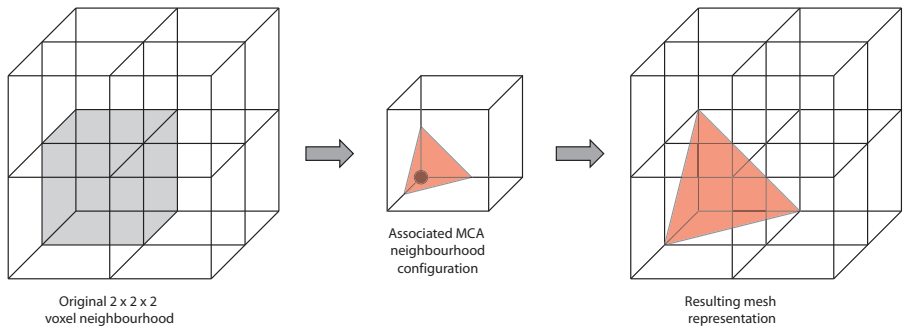



Figure 3.2: An illustration of the marching cubes isosurface generation process. Each $2 \times 2 \times 2$ voxel neighbourhood is examined and the surface patch associated with the neighbourhood configuration is generated and added to the output mesh. In this example, the original eight voxels in the input volume are replaced by a single triangle in the output mesh. Note that the corner sphere in the central image indicates the presence of a voxel located inside the isosurface i.e. the shaded voxel in the original $2 \times 2 \times 2$ volume.

[1]The units of CT attenuation are named after the inventor of the CT technique, G. N. Hounsfield. The CT attenuation of a material is related to its density.

There are 256 ($2^8$) possible configurations of eight binary voxels and although possible, the task of manually specifying the surface patches associated with each configuration is both tedious and prone to error. Lorensen & Cline observed that this task could be greatly simplified by considering all rotations and complementary cases for each configuration.

Using this approach, the number of possible configurations is reduced from 256 down to just 15, as illustrated in Figure 3.3. This significant reduction in the number of configurations makes the task of manually specifying surface patches associated with particular voxel configurations much more manageable and a lot less prone to error.

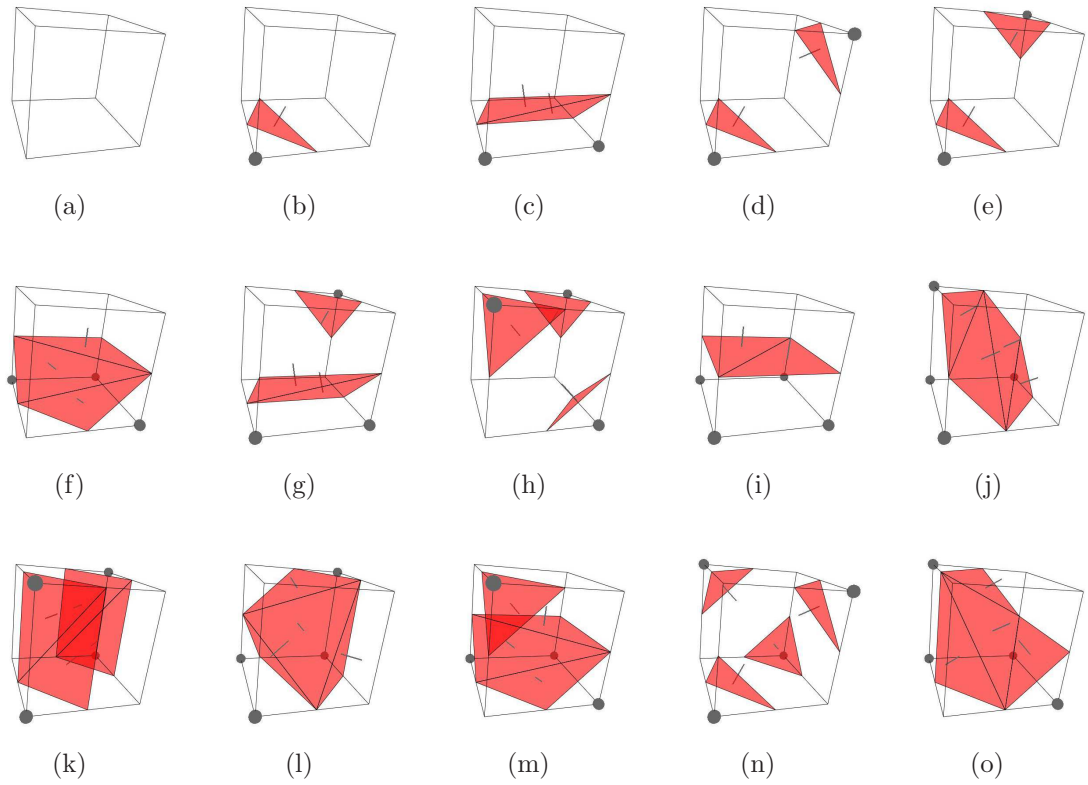
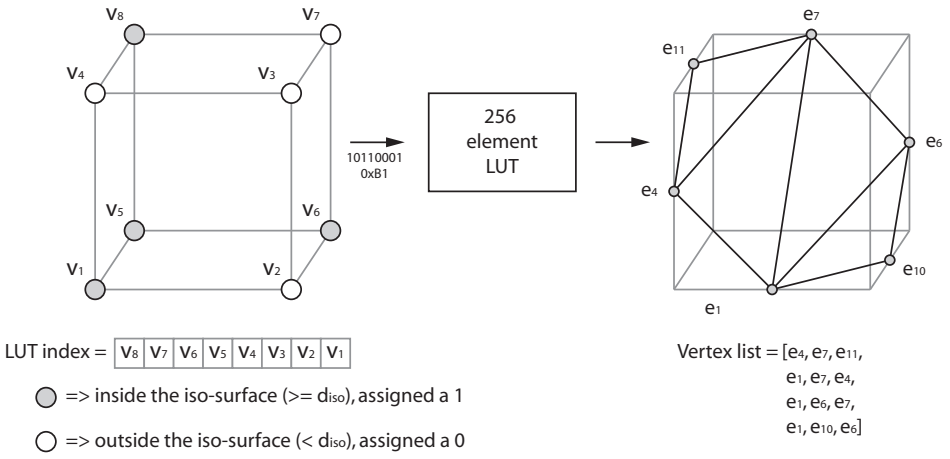



Figure 3.3: The 15 possible configurations of eight binary voxels arranged in a cubic formation. Voxels located inside the isosurface are indicated by cube corners with spheres and voxels outside the isosurface are indicated by cube corners without spheres. The relevant surface patches (highlighted using red) have been inserted as documented in the original marching cubes algorithm specification.

Once the relevant surface patches have been specified for the 15 base configurations, the original 256 configurations are regenerated by applying a series of rotations to the original 15 configurations and their complements. In each case the associated list of vertices (i.e. surface patches) and their complements are also rotated.

The resulting information is used to populate a 256 element lookup table associating voxel configurations with edge lists (i.e. lists of vertices that are positioned relative to the local origin of the cube). The lookup table index is generated based on the voxel configuration. This process is illustrated in Figure 3.4.

By default a vertex will lie midway between the two complementary valued voxels




(a)

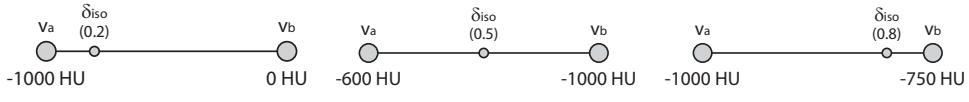
Figure 3.4: A sample voxel configuration where $v_1$, $v_5$, $v_6$ & $v_8$ are inside the isosurface and all other voxels are outside. A LUT index of 0xB1 is generated which results in the specified edge list. Note that the order of the edges is important as it defines the front face for the associated triangle in Java 3D.

($v_a$ & $v_b$) that led to its creation. This is demonstrated in general by all of the vertices associated with each of the 15 cases illustrated in Figure 3.3 and in particular, by the vertex resulting from $v_1$ & $v_2$ in Figure 3.4.

In order to fit the extracted surface more accurately to the actual isosurface identified by $d_{iso}$, each vertex must be interpolated between $v_a$ & $v_b$ based on the relationship between their densities $d_a$ & $d_b$ and the isosurface density $d_{iso}$. This is achieved by calculating the normalised distance ($\delta_{iso}$) between the voxel that is closest to the origin (either $v_a$ or $v_b$) and the isosurface (see Equation 3.1).

$$\delta_{iso} = \frac{d_{iso} - d_a}{d_b - d_a} \tag{3.1}$$

The value for $\delta_{iso}$ represents the intersection location relative to the reference voxel in terms of the intervoxel spacing (see Figure 3.5).




(a) (b) (c)

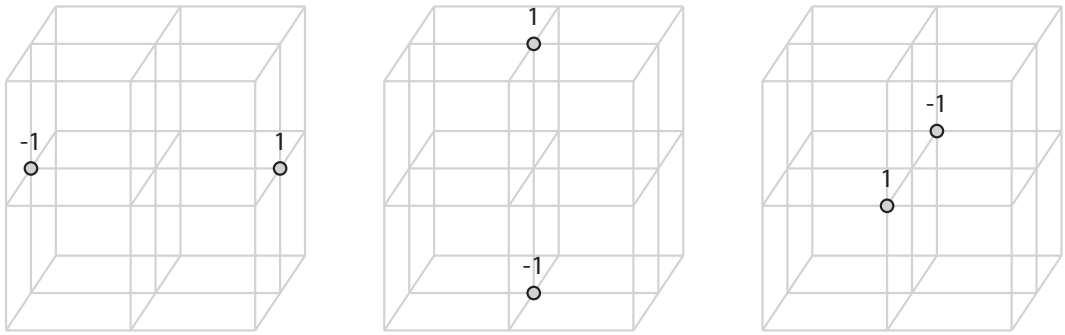
Figure 3.5: The calculation of $\delta_{iso}$ at three sample boundaries. In each case, the value of $\delta_{iso}$ is calculated using linear interpolation and the density at the point represented by $d_{iso}$ is -800 HU, i.e. the isosurface density $d_{iso}$.

The closest voxel to the origin is used as the reference point $v_a$ to ensure $\delta_{iso}$ has a positive value and to standardise the interpolation process throughout the surface extraction algorithm. The $\delta_{iso}$ value can then be used to calculate an interpolated vertex location that is more representative of the actual isosurface. Assuming that $v_a$ is the closest voxel to the origin, the interpolated vertex location $(x_i, y_i, z_i)$ located between $v_a$ and $v_b$ is calculated using:

$$
\begin{aligned}
x_i &= x_a + \delta_{iso}(x_b - x_a) \\
y_i &= y_a + \delta_{iso}(y_b - y_a) \\
z_i &= z_a + \delta_{iso}(z_b - z_a)
\end{aligned} \tag{3.2}
$$

where $(x_a, y_a, z_a)$ and $(x_b, y_b, z_b)$ are the locations of $v_a$ and $v_b$ respectively.

The final stage of the MCA involves generating unit normals for each vertex in the extracted isosurface. The normals are used to facilitate surface shading (Gouraud shading in the case of Java 3D). Normals are calculated at each voxel in the original data set using a 3-D gradient operator. The three masks for the gradient operator proposed by Lorensen and Cline are illustrated in Figure 3.6. The resulting gradient magnitude in each direction is divided by the overall gradient magnitude to yield the unit normal.




(a) (b) (c)

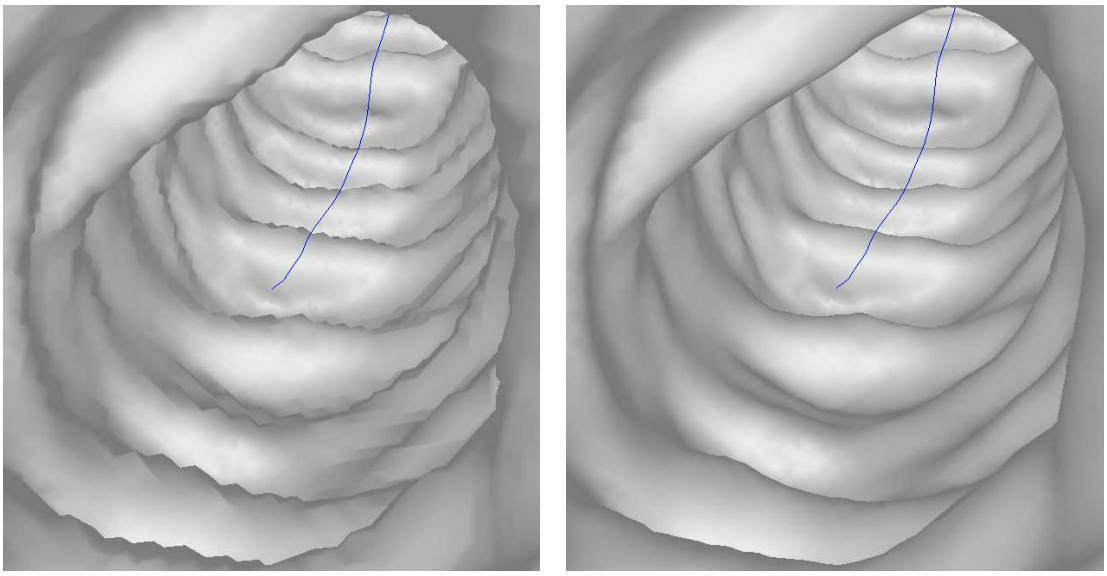
Figure 3.6: The three masks (a, b & c) that are used by (lorensen1987a) to calculate the edge magnitudes in the $x$, $y$ & $z$ directions respectively. Note that the scaling factors are omitted as the implementation discussed here is intended for use with isotropic data.

The normals of the two voxels (e.g. $v_a$ & $v_b$ as above) associated with a particular vertex must be interpolated in order to give an approximation of the normal value at the vertex location $(a_i, b_i, c_i)$. As with the vertex locations, the normals are interpolated using $\delta_{iso}$ from Equation 3.1 as follows:

$$
\begin{aligned}
a_i &= (1 - \delta_{iso})a_a + \delta_{iso}a_b \\
b_i &= (1 - \delta_{iso})b_a + \delta_{iso}b_b \\
c_i &= (1 - \delta_{iso})c_a + \delta_{iso}c_b
\end{aligned} \tag{3.3}
$$

Where $(a_a, b_a, c_a)$ and $(a_b, b_b, c_b)$ are the normals associated with voxels $v_a$ and $v_b$ respectively. The effect of vertex and normal interpolation on the quality of the extracted surface is illustrated in Figure 3.7.

This completes the basic description of the standard MCA. The vertices and their associated normals can now be rendered using conventional 3-D graphics techniques.



(a) (b)

Figure 3.7: An illustration of an isosurface model for the colonic mucosa before (a) and after (b) the use of vertex and normal interpolation.

The standard MCA is only suitable for display purposes. In order to use the MCA in virtual colonoscopy, a number of modifications and enhancements are required. These modifications, which are summarised below, are dealt with in the remainder of this chapter.

1. The standard MCA does not generate airtight surfaces. In certain cases holes may be inadvertently introduced into the generated mesh. The standard MCA must be updated so that such surface discontinuities do not occur.

2. The standard MCA uses a very basic method to generate normals. A more accurate normal generation technique is proposed as normals will be used for surface analysis as well as surface visualisation.

3. The mesh generated by the standard MCA is wasteful of memory as it contains a vast amount of repeated information. A more streamlined alternative is used to reduce the amount of memory required to store vertex coordinates and associated information (e.g. normals, colours, etc.).

4. In the streamlined mesh, mentioned in 3. above, a vertex is no longer represented by an $(x, y, z)$ coordinate. Instead, it is represented by a unique index into a list of common coordinates. A neighbour list is generated for each vertex index that identifies all of the directly connected neighbouring vertices. This is extremely useful for region growing in a triangular mesh and crucial in identifying the surface of the mesh with polyp-like properties.

## 3.2 Topology Errors (Holes)

Upon visual inspection of the output generated by the standard MCA, it is clear that topology errors (or holes) are present in the generated surface, see Figure 3.10 (a). These holes are due to ambiguous cases resulting from mismatches between the

surface patches of adjoining cubes. An example of an ambiguous case is illustrated in Figure 3.8.

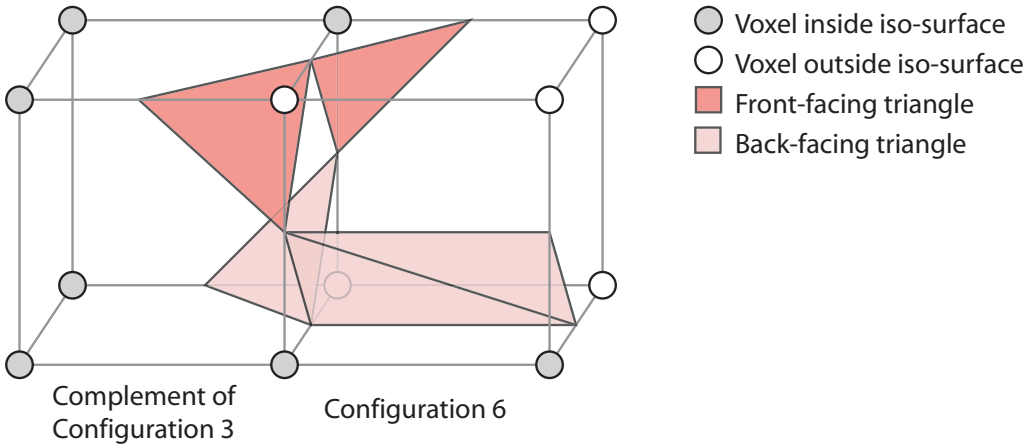



Figure 3.8: An example of the ambiguous case that results when the complement of configuration 3 (Figure 3.3 (d)) occurs next to configuration 6 (Figure 3.3 (g)). A hole is evident at the interface between these two cubes.

The ambiguous cases that result in unwanted holes are a direct result of the use of complementary cases in the standard MCA to reduce the number of core cube configurations that must be specified. By disregarding complementary cases and using only rotation to identify equivalent cube configurations the number of core configurations increases from 15 to 23. The eight additional cases and their associated surface patches are illustrated in Figure 3.9.

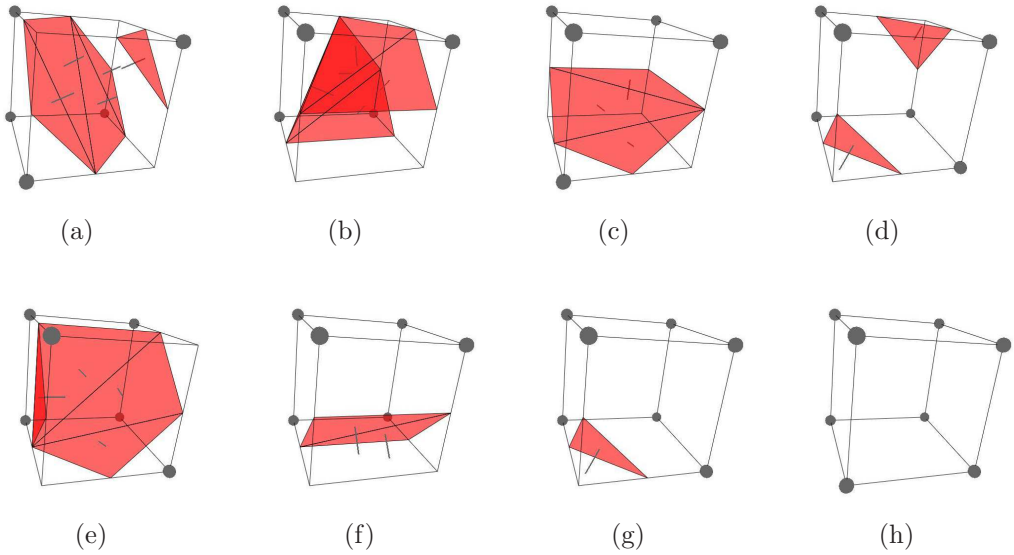
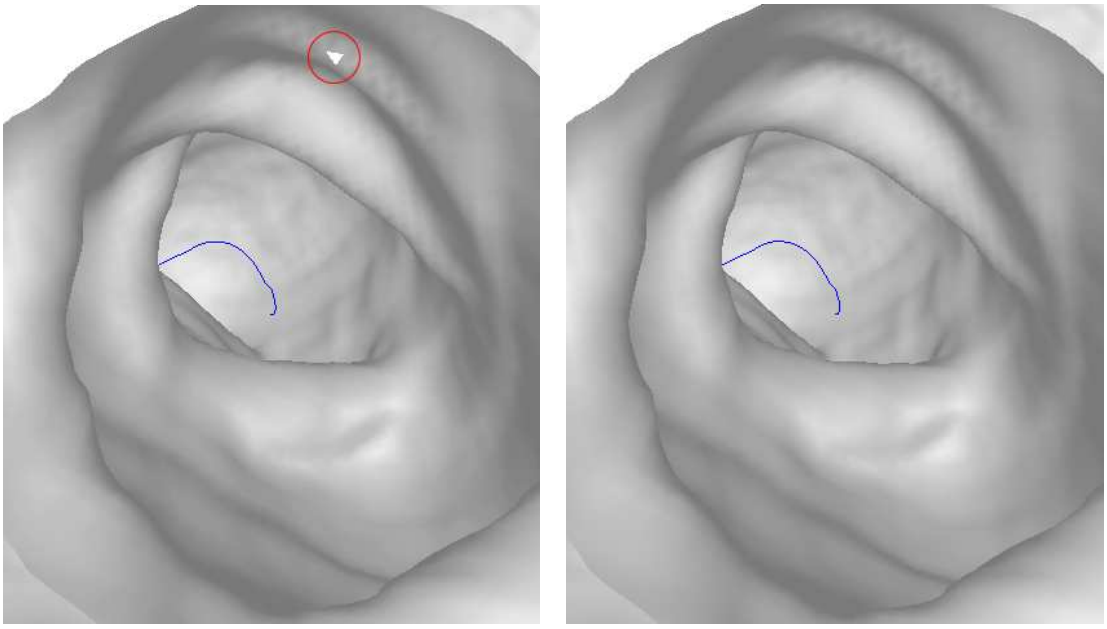



Figure 3.9: The eight additional configurations of eight binary voxels arranged in a cubic formation. These extra configurations remove the need to generate complementary cases and as a result, solve the topology problem associated with the original MCA.

Altering the standard MCA to include these eight extra cases removes the necessity to generate complementary cases and thus results in the generation of airtight surfaces that do not contain unwanted holes. The result of using the revised algorithm is presented in Figure 3.10 (b) where the solitary hole that is evident at the top of Figure 3.10 (a) is no longer present.



(a) (b)

Figure 3.10: An isosurface extracted using the 15 core neighbourhood configurations of the standard marching cubes algorithm (a) and the extended 23 neighbourhood configurations of the modified algorithm (b). Note that the standard approach creates an unwanted hole, indicated by a red circle, whereas the extended approach results in an airtight isosurface.

## 3.3 Improved Normal Calculation

The edge detector that is used for normal generation by the standard MCA (see Figure 3.6) is very basic and provides only a rough estimation of 3-D edge direction. Vertex normals are usually only used for visualisation purposes i.e. to enable shading so that surfaces generate a more realistic response to lighting. In the enhanced MCA a more accurate estimation of 3-D edge direction is required. The Zucker-Hummel edge operator was selected for this task due to its inherent smoothing effect.

- Zucker, S. W. & Hummel, R. A. (1981), 'A three-dimensional edge operator', *IEEE Transactions on Pattern Analysis and Machine Intelligence* **3**(3), 324-331.

The use of this edge operator gives a more global indication of the normal at each vertex location. The three masks for the 3-D Zucker-Hummel operator are illustrated in Figure 3.11.

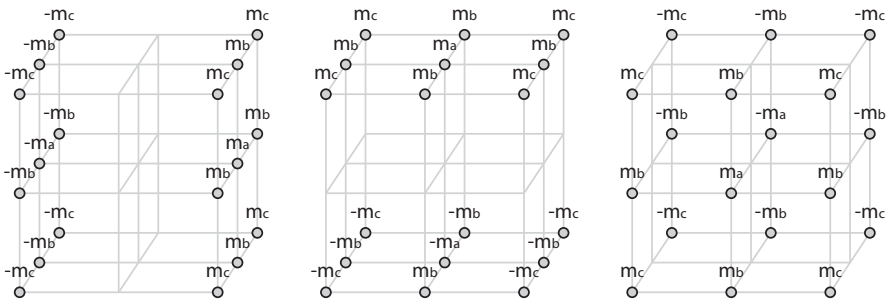



Figure 3.11: The three 26-neighbour masks representing the Zucker-Hummel edge operator where $m_a = 1.0$, $m_b = \frac{\sqrt{2}}{2}$ & $m_c = \frac{\sqrt{3}}{3}$.

## 3.4 Mesh Representation

The mesh generated by the standard MCA consists of a disjoined set of triangular patches where there is a high degree of vertex repetition. This representation, although suitable for visualisation purposes, is not ideal for analysis. It is also extremely wasteful of memory due to the high degree of vertex repetition.

An alternative mesh representation involves storing each vertex in an array structure where a particular vertex can be present only once. Using this approach, triangles that represent the mesh are specified as indices into the vertex array and not as actual vertex locations. An array of unit normals is generated and populated in the same manner.

This approach to mesh storage has the potential to significantly reduce the amount of memory required to store a fully characterised isosurface representation of the colonic mucosa. A simple example illustrating the difference between the standard mesh and the indexed mesh is presented in Figure 3.12.

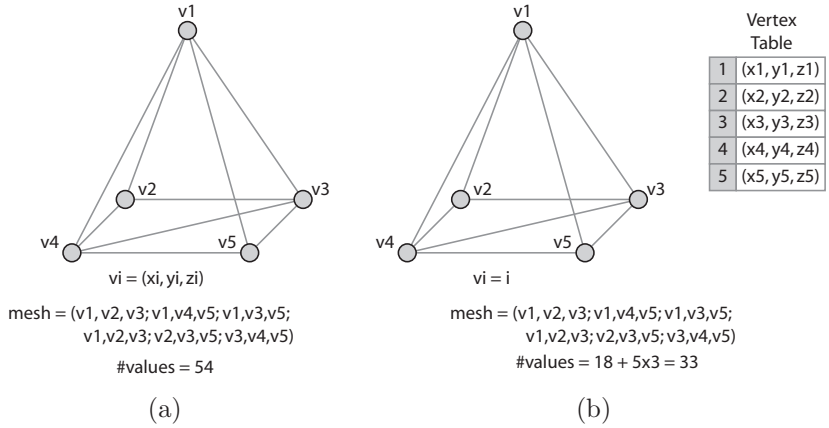



Figure 3.12: An illustration of how indexing can be used to reduce the amount of data required to specify a mesh structure. A pyramid consists of five vertices and six triangles. The unoptimised representation yields 54 values (a) and the optimised alternative yields only 33 (b). Note that in Java the data type for vertex (float) and index (int) both require four bytes of storage space.

The indexing process requires a modification to the standard MCA. As each vertex is encountered it is assigned an index and stored in the vertex list. This index is then used to represent the relevant vertex coordinates (i.e. an Integer primitive (four bytes) is used to reference three floating point primitives (12 bytes)).

If a vertex that was already assigned an index is encountered then that index is used. Conversely, a new index is generated if a new vertex is encountered. Only vertices associated with the same slice or two adjacent slices can be shared. As a result, only two slices need to be resident in memory at any one time. Noting that in the context of the marching cubes algorithm a slice is actually two voxels thick.

## 3.5 Neighbour Identification

The final modification to the standard MCA involves identifying all of the directly connected neighbouring vertices within the mesh. This step is required to facilitate the automatic detection of polyps from the isosurface representation of the colon surafce.
As each triangle, consisting of vertices $e_0$, $e_1$ and $e_2$, is identified, its vertex relationships are added to an array structure where: $e_0$ is associated with $e_1$ & $e_2$ (i.e. two vertex pairs $(e_0, e_1)$ and $(e_0, e_2)$), $e_1$ is associated with $e_0$ & $e_2$ and $e_2$ is associated with $e_0$ & $e_1$. A vertex pair is only added to the array structure if this vertex pair is not already present.

An example illustrating the neighbour identification process is illustrated in Figure 3.13. Representing the vertex neighbours in this way reduces the task of neighbourhood identification from an extensive mesh search to a simple table lookup i.e. by specifying the index of one vertex the indices of all of the neighbouring vertices are returned.

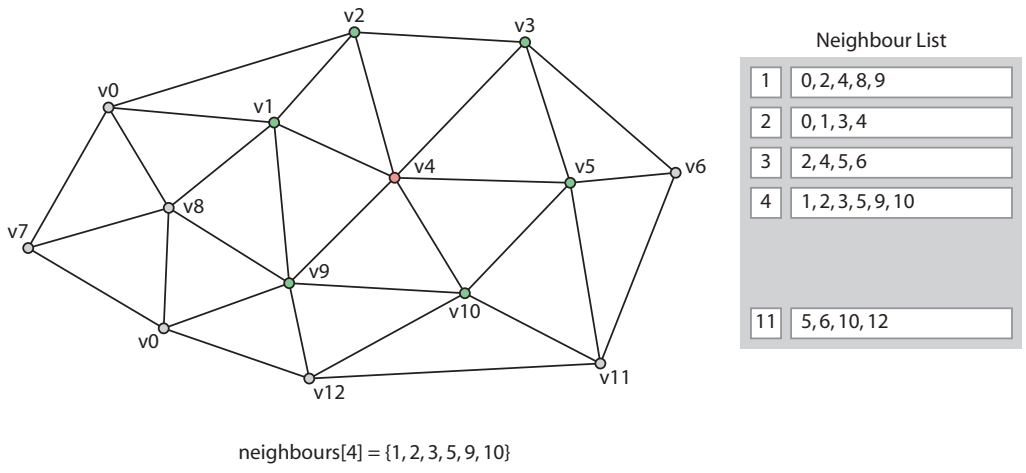



Figure 3.13: The neighbour indexing process: The neighbouring vertices for each mesh vertex are stored in a list to streamline the process of searching for vertex neighbours.

## 3.6 Summary

This chapter provided a complete description of the marching cubes algorithm. The use of this approach to extract an isosurface enables the indirect rendering of volumetric data using conventional surface rendering techniques, i.e. those described in the previous chapter. The implementation of the marching cubes algorithm described in this chapter deals with some of the issues associated with the standard marching cubes algorithm e.g. fixing topology errors and improving the normal calculation stage. It has also been demonstrated that the use of indexed geometry can be used to reduce the amount of memory required to store the extracted polygonal mesh structure.