### 2.6.6 Strip Geometry

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

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

public class StripGeometryExample extends BasicSceneWithMouseControl{

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

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

    // Define the coordinates to be used
    float [] coordinates = {-0.5f, -0.5f,  0.0f,
                             0.2f, -0.4f,  0.0f,
                             0.3f,  0.3f,  0.0f,
                            -0.3f,  0.5f,  0.0f};

    // Specify a single strip that used all 4 coordinates
    int [] stripVertexCounts = {4};

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

    geometryArray.setCoordinates(0, coordinates);

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

    root.compile();

    return root;
  }
}
```

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

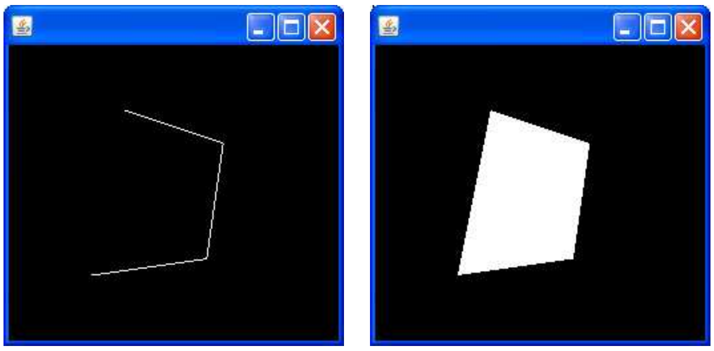
(a) (b)

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

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

### 2.6.7 Creating Complex Geometry

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

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

public class CylinderExample extends BasicSceneWithMouseControl{

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

  private Geometry createGeometry() {

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

    float angle = 0.0f;
    double angleIncrement = (2*Math.PI)/faces;

    // create an empty array of floats to hold the coordinates
    float coordinates[] = new float[faces*4*3];

    for(int f=0; f<faces; f++)
    {
      // Generate the four coordinates required for each face
      float x1 = (float)(radius*Math.cos(angle));
      float z1 = (float)(radius*Math.sin(angle));
      angle -= angleIncrement;
      float x2 = (float)(radius*Math.cos(angle));
      float z2 = (float)(radius*Math.sin(angle));

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

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

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

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

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

  }

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

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

    return objRoot;
  }
}
```

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

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

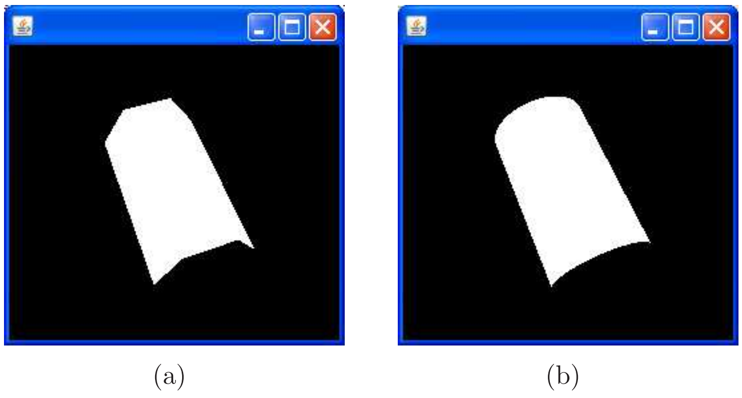
Figure 2.20: Examples of geometry representing a cylinder that was created using a for loop. (a) a cylinder with 6 faces and (b) a cylinder with 60 faces.

structure of the sphere geometry.

### 2.6.8 Indexed Geometry

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

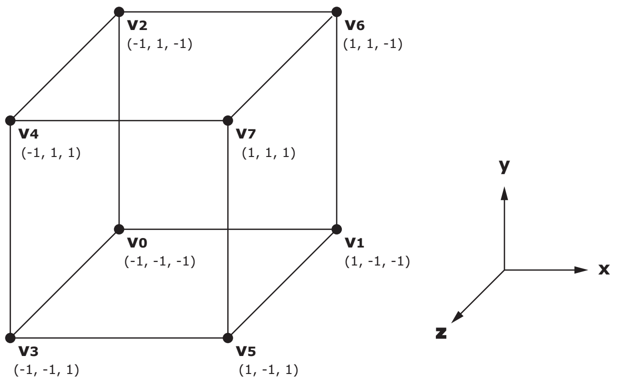

Figure 2.21: A simple cube consisting of six faces and eight vertices.

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

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

public class IndexedGeometryExample extends BasicSceneWithMouseControl{

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

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

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

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

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

    Shape3D cube = new Shape3D(quadArray);

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

    return root;
  }
}
```

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

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

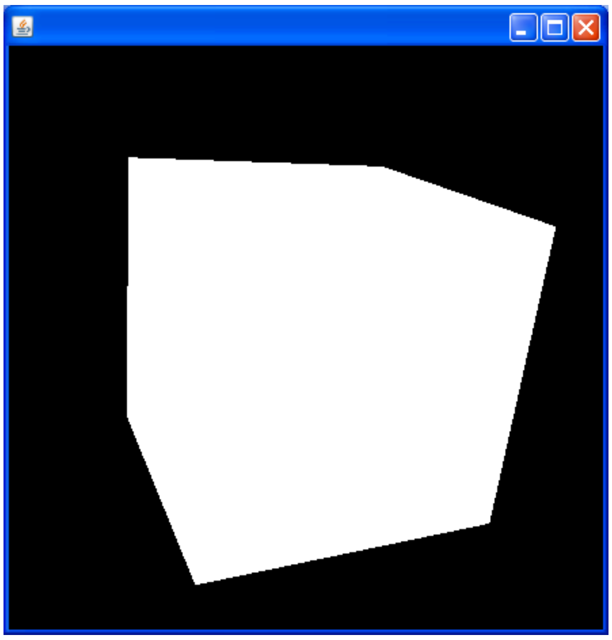

Figure 2.22: A cube created using a suitably constructed `IndexedQuadArray` geometry. The cube is rendered using the default appearance.

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

## 2.6.9 Loading Geometry from Files

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

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

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

- `ObjectFile(int flags)`

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

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

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

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

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

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

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

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

import com.sun.j3d.loaders.*;
import com.sun.j3d.loaders.objectfile.*;

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

  public BranchGroup createContentBranch()
  {
    // Resize the scene to fit the display
    int flags = ObjectFile.RESIZE;

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

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

    root.compile();

    return root;
  }
}
```

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

## 2.7 Appearance

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

### 2.7.1 The `Appearance` Node Component

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

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

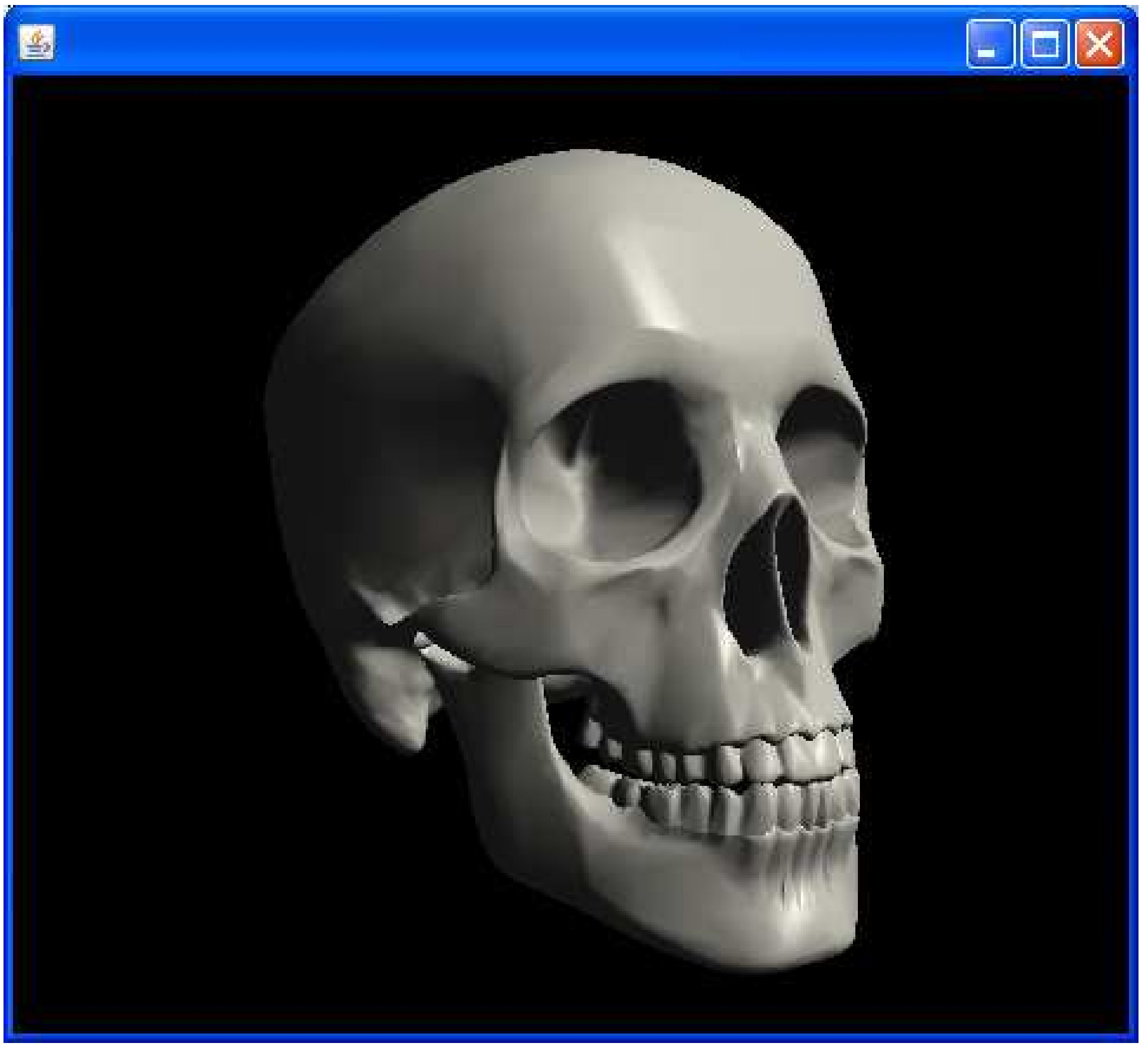
Figure 2.23: An example of the type of geometry that can be loaded into a Java 3D program using a custom `Loader`.

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

- **Material -** defines the appearance of an object under illumination, such as the ambient colour, diffuse colour, specular colour, emissive colour, and shininess. These attributes are defined using a `Material` object.
- **Texture -** defines the texture image and filtering parameters used when texture mapping is enabled. These attributes are defined using a `Texture` object.
- **Texture attributes -** defines the attributes that apply to texture mapping, such as the texture mode, texture transform, blend colour, and perspective correction mode. These attributes are defined using a `TextureAttributes` object.
- **Texture coordinate generation -** defines the attributes that apply to texture coordinate generation, such as whether coordinate generation is enabled, coordinate format (2D or 3D coordinates), coordinate generation mode (object linear, eye linear, or spherical reflection mapping), and the R, S and T coordinate plane equations. These attributes are defined using a `TexCoordGeneration` object.
- **Texture unit state -** an array that defines the texture state for each of *N* separate texture units. This allows multiple textures to be applied to a geometry. Each `TextureUnitState` object contains a `Texture` object, a `TextureAttributes` object and a `TexCoordGeneration` object.

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

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

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

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

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

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

### 2.7.2 `ColoringAttributes`

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

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

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

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

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

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

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

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

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

- `ColoringAttributes()`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    root.compile();

    return root;
  }
}
```

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

**Reference:**

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

### 2.7.3 `PointAttributes`

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

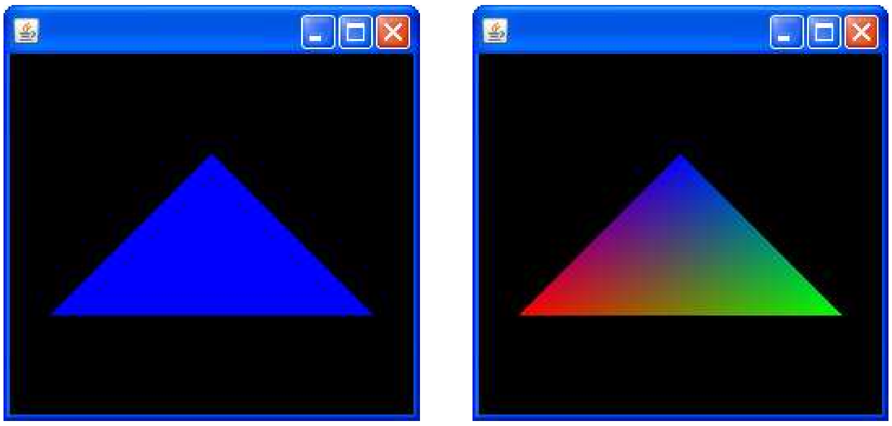
(a) (b)

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

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

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

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

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

```

import javax.media.j3d.*;

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

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

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

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

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

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

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

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

    root.compile();

    return root;
  }
}
```

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

### 2.7.4 `LineAttributes`

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

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

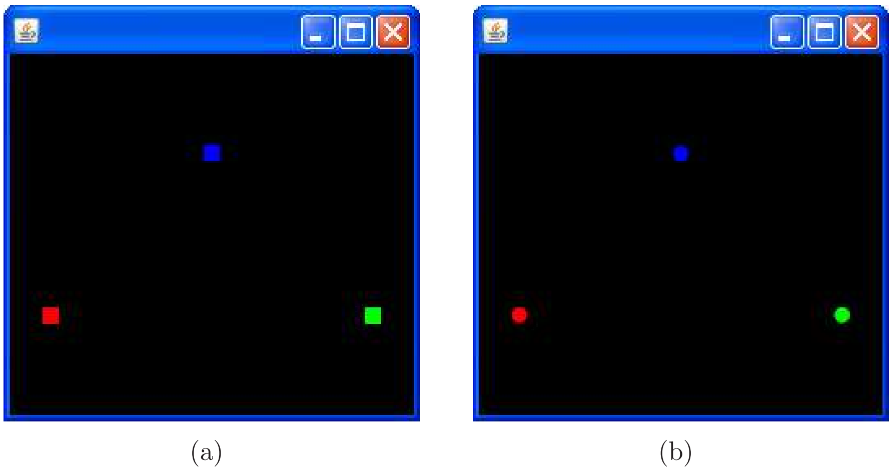
Figure 2.25: Examples of points 10 pixels in size the were rendered with (a) and without (b) antialiasing.

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

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

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

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

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

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

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

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

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

public class LineAttributesExample extends BasicSceneWithMouseControl
```

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

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

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

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

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

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

    // Define the line attributes appearance component
    Appearance appearance = new Appearance();
    LineAttributes la = new LineAttributes();
    la.setLinePattern(LineAttributes.PATTERN_USER_DEFINED);
    la.setPatternMask(0xFFF0);
    la.setLineAntialiasingEnable(true);
    la.setLineWidth(5);
    appearance.setLineAttributes(la);

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

    root.compile();

    return root;
  }
}
```

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

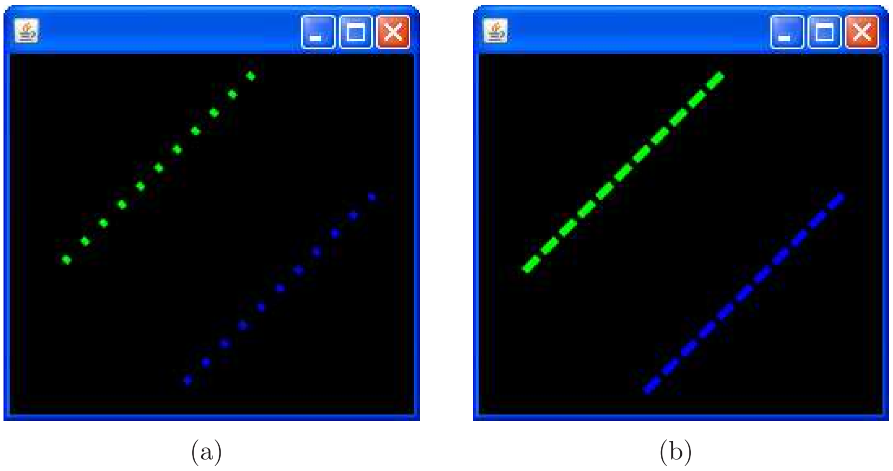

Figure 2.26: Examples of lines rendered using different patterns. (a) A line with 4 pixel segments followed by a 12 pixel gap. (b) A line with 12 pixel segments followed by a 4 pixel gap. In both cases the lines are 5 pixels wide and rendered with antialiasing.

### 2.7.5 `PolygonAttributes`

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

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

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

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

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

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

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

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

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

    Appearance appearance = new Appearance();

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

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

    root.compile();

    return root;
  }
}
```

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

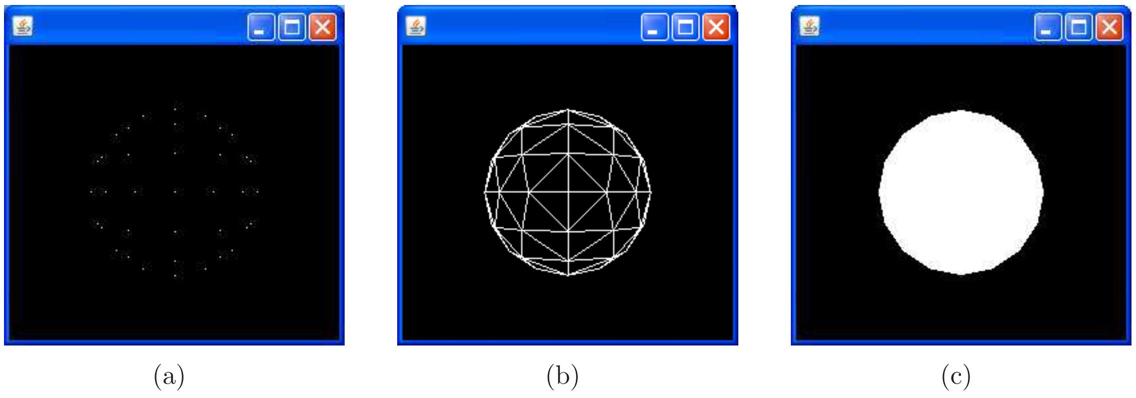

Figure 2.27: A `Sphere` primitive with 15 subdivisions rendered as (a) a set of points, (b) a wire frame and (c) filled polygons. Back facing polygons in all three renderings are culled.

### 2.7.6 `RenderingAttributes`

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

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

### 2.7.7 `TransparencyAttributes`

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

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

  **Note:** In the Microsoft Windows XP implementation of Java 3D version `FASTEST` and `NICEST` modes both result in `BLENDED` being used.

- Transparency value, the amount of transparency to be applied to this appearance component object. The transparency values are in the range [0.0, 0.1], with 0.0 being fully opaque and 1.0 being fully transparent.

- Blend function - used in blended transparency and antialiasing operations. The source function specifies the factor that is multiplied by the source colour. This value is added to the product of the destination blend function and destination colour.

The default blend equation has the following format:

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

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

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

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

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

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

The possible values for the destination blend function are:

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

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

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

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

public class TransparencyAttributesExample extends BasicSceneWithMouseControl
{

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

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

    Appearance appearance1 = new Appearance();

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

    // Create the transparency appearance compoennt that represents
    // a 50% blended transparency
    TransparencyAttributes ta = new TransparencyAttributes();
    ta.setTransparencyMode(TransparencyAttributes.BLENDED);
    ta.setTransparency(0.5f);
    appearance1.setTransparencyAttributes(ta);

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

    Appearance appearance2 = new Appearance();

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

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

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

    return root;
  }
}
```

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

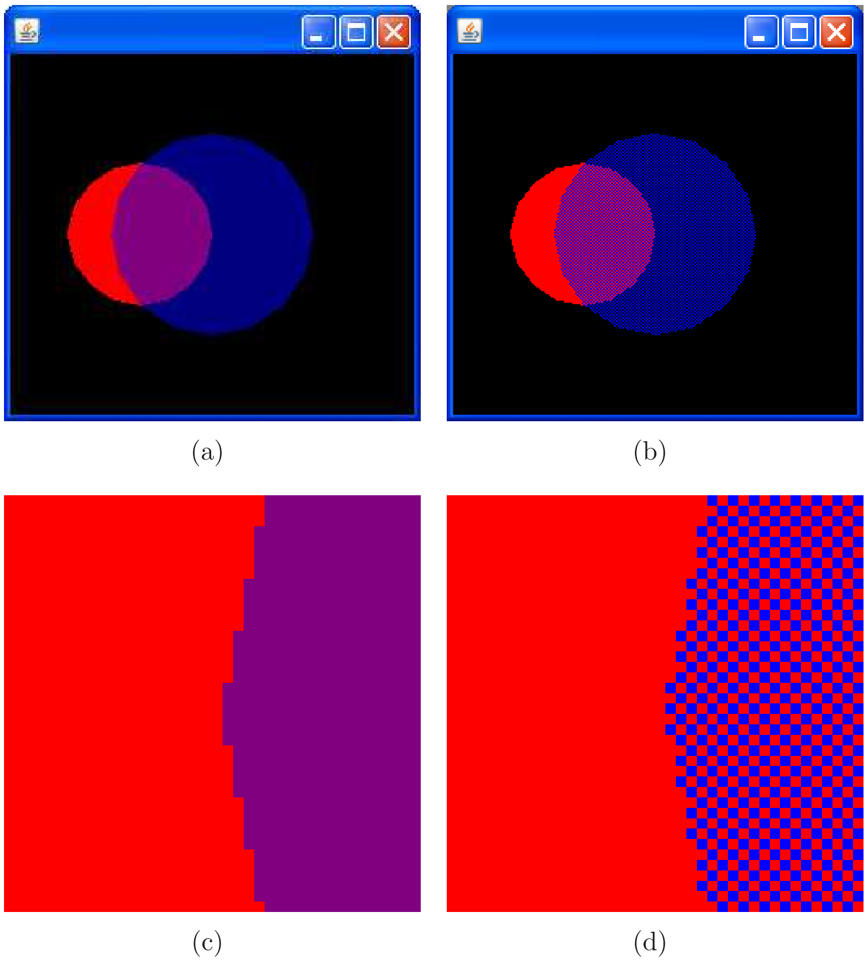

Figure 2.28: An scene containing a opaque red sphere located behind a semitransparent blue sphere. The `BLENDED` transparency mode is used in (a) and (c) and the `SCREEN_DOOR` mode is used in (b) and (d).

### 2.7.8 `Material`

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

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

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

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

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

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

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

#### 2.7.8.1 Ambient Colour

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

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

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

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

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

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

- `void getAmbientColor(Color3f colour)`

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

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

#### 2.7.8.2 Emissive Colour

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

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

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

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

- `void getEmissiveColor(Color3f colour)`

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

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

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

#### 2.7.8.3 Diffuse Colour

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

The diffuse colour of a `Material` can be updated or retrieved using the similar methods as those described for ambient and emissive colours. The required capabilities must also be set of the diffuse colour needs to be updated or retrieved after the scene graph has gone live.

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

#### 2.7.8.4 Specular Colour

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

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

#### 2.7.8.5 Shininess

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

**Note:** The default shininess value is 64.

#### 2.7.8.6 Vertex Colours

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

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

The possible values for the `colourTarget` argument are:

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

### 2.7.9 Lighting

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

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

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

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

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

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

### 2.7.9.1 `DirectionalLight`

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

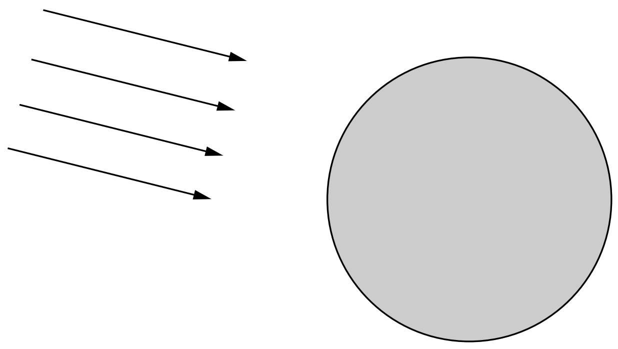

Figure 2.29: A `DirectionalLight` represents light from a distant source a can be thought of as a set of parallel rays originating from a specific direction.

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

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

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

- `void setDirection(Vector3f direction)`
  Sets the direction of this `DirectionalLight` to the specified value.

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

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

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

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

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

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

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

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

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

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

    Appearance appearance = new Appearance();

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

    Sphere sphere = new Sphere(0.5f, appearance);

    root.addChild(sphere);

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

    root.compile();

    return root;
  }
}
```

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

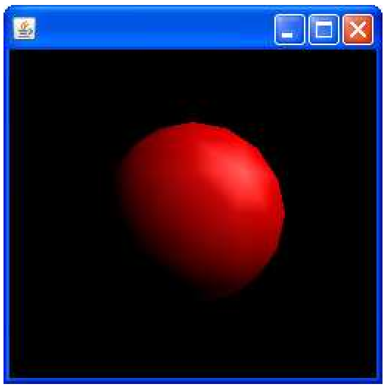

Figure 2.30: An example of a red sphere illuminated by white directional light shining in the direction (-1.0f, -1.0f, -1.0f).

#### 2.7.9.2 `PointLight`

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

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

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

- Constant attenuation
- Linear attenuation
- Quadratic attenuation

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

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

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

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

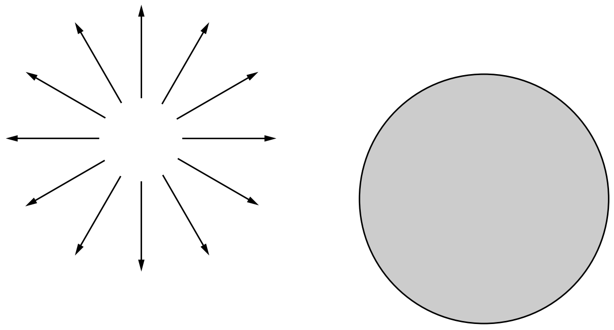

Figure 2.31: A `PointLight` represents light from a specific location. The light rays associated with the point light source all emanate from this point.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    Sphere sphere2 = new Sphere(0.4f, appearance2);
    tg2.addChild(sphere2);

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

    root.compile();

    return root;
  }
}
```

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

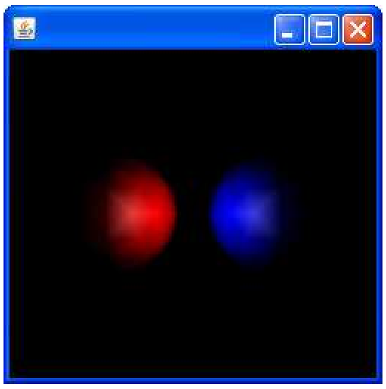

Figure 2.32: An examples of two spheres located left and right of the origin illuminated by a `PointLight` that is located at the origin.