Primitives - Box & Cylinder

- Box

  `Box()`

  Constructs a default box 2.0 metres in height, width, and depth, centred at the origin

  `Box(float xdim, float ydim, float zdim, Appearance app)`

  Constructs a box of a given dimension and appearance

- Cylinder

  `Cylinder()`

  Constructs a default cylinder of radius 1.0 metre and height 2.0 metres, aligned along the y-axis and centred at the origin

  `Cylinder(float radius, float height)`

  Constructs a cylinder of a given radius and height

  `Cylinder(float radius, float height, Appearance app)`

  Constructs a cylinder of a given radius, height and appearance

Primitives - Cone & Sphere

- Cone

  `Cone()`

  Constructs a default cone of radius 1.0 metre and height 2.0 metres, aligned along the y-axis and centred at the origin

  `Cone(float radius, float height)`

  Constructs a cone of a given radius and height

  `Cone(float radius, float height, Appearance app)`

  Constructs a cone of a given radius, height and appearance

- Sphere

  `Sphere()`

  Constructs a default sphere of radius of 1.0 metre, centred at the origin

  `Sphere(float radius)`

  Constructs a sphere of a given radius

  `Sphere(float radius, Appearance app)`

  Constructs a sphere of a given radius and appearance

Primitives - Surface Normals

The previous set of constructors create primitives without surface normals. In order to illuminate a geometric object surface normals must be provided. The following set of constructors will generate surface normals when creating the primitives -

```
Box (xdim, ydim, zdim,
     Box.GENERATE_NORMALS, appearance)
Cone (radius, height,
     Cone.GENERATE_NORMALS, appearance)
Cylinder (radius, height,
     Cylinder.GENERATE_NORMALS, appearance)
Sphere (radius, Sphere.GENERATE_NORMALS, appearance)
```

Primitives - Methods

`public void setNumVertices(int num)`

Sets total number of vertices in this primitive

`void setAppearance()`

Sets the main appearance of the primitive (all subparts) to a default white appearance

`void setAppearance(int partid, Appearance appearance)`

Sets the appearance of a subpart given a partid. Box, Cylinder and Cone objects are composed of more than one Shape3D object, each potentially with its own Appearance node component. The value used for partid specifies which of the Appearance node components to set

`Shape3D getShape(int partid)`

Gets one of the faces (Shape3D) from the primitive that contains the geometry and appearance. The value used for partid specifies which of the Geometry node components to get (it must be absent or 1 for a sphere)

Shape3D - Geometry

- Subclasses of Geometry fall into three broad categories
  - Non-indexed vertex-based geometry
    - Each time a visual object is rendered, its vertices may be used only once
  - Indexed vertex-based geometry
    - Each time a visual object is rendered, its vertices may be reused
  - Other visual objects
    - Raster, Text3D and CompressedGeometry

Geometry Arrays

`GeometryArray(int vertexCount, int vertexFormat)`

Constructs an empty GeometryArray object with the specified number of vertices, and vertex format. One or more individual flags are bitwise ORed together to describe the per-vertex data. The flag constants used for specifying the format are:

`COORDINATES:`	This vertex array contains co-ordinates
	This bit must be set
`NORMALS:`	This vertex array contains surface normals
`COLOR_3:`	This vertex array contains colours without transparency
`COLOR_4:`	This vertex array contains colours with transparency
`TEXTURE_COORDINATE_2:`	This vertex array contains 2D texture co-ords
`TEXTURE_COORDINATE_3:`	This vertex array contains 3D texture co-ords

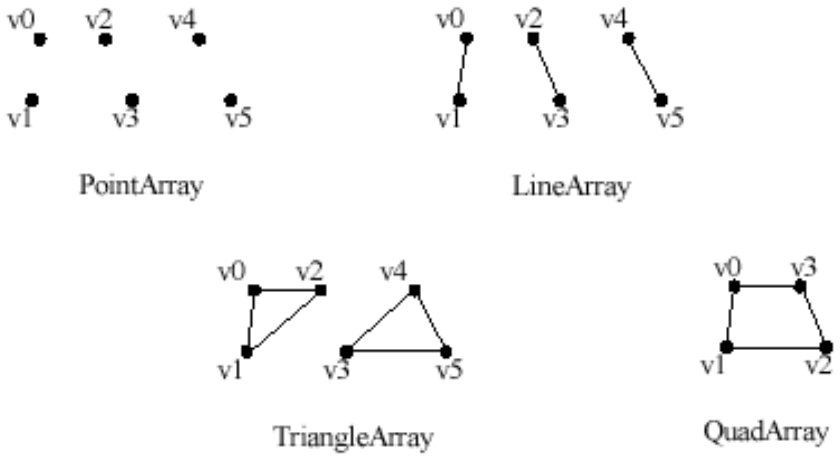
For each flag a corresponding array is created internal to the GeometryArray

`GeometryArray` Subclasses

- `GeometryArray` is an abstract class which acts as a base class for a number of classes which permit geometry to be specified as a list of co-ordinates

```
PointArray(int vertexCount, int vertexFormat)
LineArray(int vertexCount, int vertexFormat)
TriangleArray(int vertexCount, int vertexFormat)
QuadArray(int vertexCount, int vertexFormat)

GeometryStripArray()
```

Geometry Array Examples



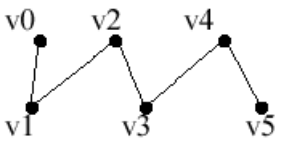
Geometry Strip Arrays

- `GeometryStripArray` is an abstract class from which strip primitives are derived

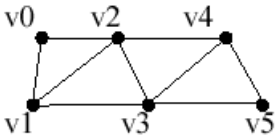
```
LineStripArray(int vertexCount, int vertexFormat,
int stripVertexCounts[])

TriangleStripArray(int vertexCount, int
vertexFormat, int stripVertexCounts[])

TriangleFanArray(int vertexCount, int
vertexFormat, int stripVertexCounts[])
```

- Strip primitives re-use vertices

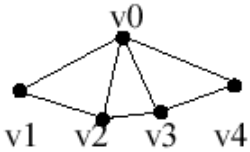
Geometry Strip Array Examples



LineStripArray



TriangleStripArray



TriangleFanArray

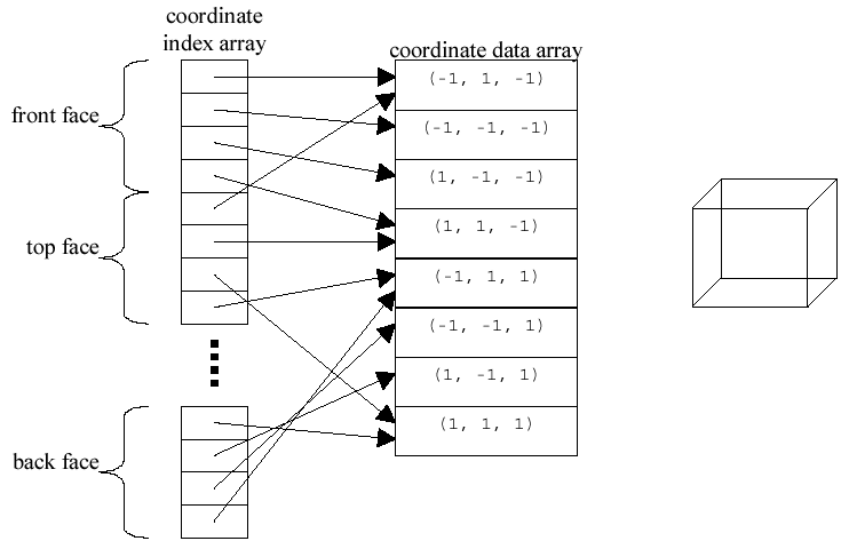
Insides and Outsides

- Right-hand rule
  - Place right hand over a polygon and curl fingers in direction in which vertices have been ordered
  - Thumb will point in outwards direction
- Triangle strip arrays cause a problem
  - Direction of thumb alternates for each successive triangle
- For `GeometryStripArray` objects Java does this alternation automatically

Indexed Geometry

- Indexed geometry provides a mechanism for eliminating all redundancy in vertices
  - Indexed geometry will take longer to render though
- A set of index arrays point into the four data arrays within a geometry array (co-ordinates, normals, colours, textures)
- The values in the index arrays provide the order in which the vertices are to be used during rendering so the order of the vertices themselves is no longer important

Co-ordinate Indexed Array



Indexed Geometry Arrays

- `IndexedGeometryArray` is an abstract class from which indexed geometry classes are derived

```
IndexedPointArray(int vertexCount, int vertexFormat,
  int indexCount)
IndexedLineArray(int vertexCount, int vertexFormat,
  int indexCount)
IndexedTriangleArray(int vertexCount, int vertexFormat,
  int indexCount)
IndexedQuadArray(int vertexCount, int vertexFormat,
  int indexCount)

IndexedGeometryStripArray()
```

Tetrahedron Example – Co-ordinates

Consider the following code fragments from the tetrahedron code supplied -

```
// Construct an indexed triangle array object to hold 12
// indices to the co-ordinates of the 4 vertices
   IndexedTriangleArray tetrahedronGeometry =
   new IndexedTriangleArray(4, GeometryArray.COORDINATES, 12);

// Generate an array to define the co-ordinates of 4 vertices
   Point3d vertices[] = new Point3d[4];
   vertices[0]= new Point3d(-1.0,-1.0,0.0);
   vertices[1]= new Point3d(1.0,-1.0,0.0);
   vertices[2]= new Point3d(0.0,1.0,0.0);
   vertices[3]= new Point3d(0.0,0.0,2.0);

// Set the co-ordinates of the 4 vertices starting with the
// vertex defined in element 0 of the vertices array
   tetrahedronGeometry.setCoordinates(0,vertices);
   Shape3D tetrahedron = new Shape3D(tetrahedronGeometry);
```

Tetrahedron Example - Indices

```
// Generate an index array which identifies the 3 vertices for
// each of the 4 triangular faces in the order required to
// define the inside and outside of the tetrahedron correctly

   int faceIndices[] = {
       0,2,1,  // Base – vertices[0], vertices[2], vertices[1]
       0,1,3,  // Side 1
       2,0,3,  // Side 2
       1,2,3   // Side 3
       };

// Set the 12 vertex indices, specified in the constructor,
// starting with element 0 of the faceIndices array

   TetrahedronGeometry.setCoordinateIndices(0,faceIndices);
   Shape3D tetrahedron = new Shape3D(tetrahedronGeometry);
```

Tetrahedron Example – Geometry and Colours

```
// Construct an indexed triangle array object to hold
// 12 indices to the co-ordinates of the 4 vertices
// and to 12 indices to colours for the vertices

IndexedTriangleArray TetrahedronGeometry =
   new IndexedTriangleArray(
4,GeometryArray.COORDINATES|GeometryArray.COLOR_3,12);

// Note that the colours will be used to gradient fill
// the 4 faces of the tetrahedron depending on which
// colours are linked to which vertices of each face

Shape3D tetrahedron = new Shape3D(tetrahedronGeometry);
```

Tetrahedron Example - Colours

```
// Generate an array to define 4 colours by their RGB values
    Color3f colours[] = new Color3f[4];
    colours[0]= new Color3f(1.0f,1.0f,1.0f); // White
    colours[1]= new Color3f(1.0f,0.0f,0.0f); // Red
    colours[2]= new Color3f(0.0f,1.0f,0.0f); // Green
    colours[3]= new Color3f(0.0f,0.0f,1.0f); // Blue
    TetrahedronGeometry.setColors(0,colours);

// Generate an index array which assigns one of the 4 colours
// to each vertex in the definition of each of the 4 faces
    int colourIndices[] = {
        0,2,1,  // Base – colours[0], colours[2], colours[1]
        0,1,3,  // Side 1
        2,0,3,  // Side 2
        1,2,3   // Side 3
       };
    TetrahedronGeometry.setColorIndices(0,colourIndices);
    Shape3D tetrahedron = new Shape3D(tetrahedronGeometry);
```

Tetrahedron Example – Polygon Attributes

```
// Set the appearance of the tetrahedron
PolygonAttributes tetrahedronAttributes =
     new PolygonAttributes();

tetrahedronAttributes.setPolygonMode(
   PolygonAttributes.POLYGON_LINE);

Appearance appear = new Appearance();
appear.setPolygonAttributes(tetrahedronAttributes);
tetrahedron.setAppearance(appear);

// Note that setting the appearance with PolygonMode
// in PolygonAttributes enforces rendering in
// wire-frame mode according to POLYGON_LINE
```

Indexed Geometry Strip Arrays

- `IndexedGeometryStripArray` is an abstract class from which indexed strip classes are derived

```
IndexedLineStripArray(int vertexCount,
  int vertexFormat, int indexCount,
  int stripVertexCounts[])
IndexedTriangleStripArray(int vertexCount,
  int vertexFormat, int indexCount,
  int stripVertexCounts[])
IndexedTriangleFanArray(int vertexCount,
  int vertexFormat, int indexCount,
  int stripVertexCounts[])
```

`GeometryInfo` Class

- `GeometryInfo()` constructs a triangle array from a co-ordinate list defining a set of polygons
  - It also permits some of the polygons to define holes within other polygons
  - A separate `Triangulator()` can perform an additional random triangulation (deprecated!)
- `NormalGenerator()` generates normals for each vertex in the triangle array
- `Stripifier()` turns the triangle array into a triangle strip array for greater efficiency

`GeometryInfo` Types

- `GeometryInfo()` is invoked with a single argument which specifies the type of geometric shape which is to be created
  - `GeometryInfo.POLYGON_ARRAY`
  - `GeometryInfo.QUAD_ARRAY`
  - `GeometryInfo.TRIANGLE_ARRAY`
  - `GeometryInfo.TRIANGLE_FAN_ARRAY`
  - `GeometryInfo.TRIANGLE_STRIP_ARRAY`
- `GeometryInfo.POLYGON_ARRAY` is the most useful of these because the others can all be handled with constructors we have already met

`GeometryInfo` Details

- The vertices are specified as an ordered array of co-ordinates
  - With duplication of vertices as in geometry arrays
- A strip count array gathers the co-ordinates into an ordered list of polygonal faces
  - By stating how many vertices belong to each polygon starting from the beginning of the co-ordinate list
- A contour count array identifies containing polygons and the holes they contain
  - By forming groups of the polygons in the strip count array the first of which is a container, the rest holes

Gasket Example – Co-ordinates

Consider the following code fragments from the gasket code supplied –

```
GeometryInfo gasket = new GeometryInfo(GeometryInfo.POLYGON_ARRAY);
    // Generate the co-ordinate array for the 6 faces and 3 holes
    double[] gasketCoordinates = {
    // Bottom face of cube
    -1,-1,-1,     1,-1,-1,     1,-1, 1,    -1,-1, 1,
    // Hole in bottom face
    -0.5,-1,-0.5,     0.5,-1,-0.5,     0.5,-1, 0.5,    -0.5,-1, 0.5,
    // Front face of cube
     1, 1, 1,    -1, 1, 1,    -1,-1, 1,     1,-1, 1,
    // Hole in front face
     0.5, 0.5, 1,    -0.5, 0.5, 1,    -0.5,-0.5, 1,     0.5,-0.5, 1,
    // Right face of cube
     1, 1, 1,     1,-1, 1,     1,-1,-1,     1, 1,-1,
    // Hole in right face
     1, 0.5, 0.5,     1,-0.5, 0.5,     1,-0.5,-0.5,     1, 0.5,-0.5,
    // Left face of cube
     -1,-1,-1,     -1,-1, 1,     -1, 1, 1,     -1, 1,-1,
    // Top face of cube
      1, 1, 1,      1, 1,-1,     -1, 1,-1,     -1, 1, 1,
    // Back face of cube
     -1,-1,-1,     -1, 1,-1,      1, 1,-1,      1,-1,-1
    };
        gasket.setCoordinates(gasketCoordinates);
```

Gasket Example – Strip Counts

```
// Generate a strip count array which
// associates co-ordinates with polygons

int gasketStripCount[] = {
        4,4,    // Bottom face and its hole
        4,4,    // Front face and its hole
        4,4,    // Right face and its hole
        4,      // Left face
        4,      // Top face
        4       // Back face
        };
gasket.setStripCounts(gasketStripCount);
```

Gasket Example – Contour Counts

```
// Generate a contour count array which sets polygons
// in the strip count array to be containers and holes

int gasketContourCount[] = {
    2,  // First 2 elements are the bottom and its hole
    2,  // Next 2 are the front face and its hole
    2,  // Then the right face and its hole
    1,  // The left face has no hole
    1,  // Nor does the top face
    1,  // Finally the back face brings up the rear
    };

gasket.setContourCounts(gasketContourCount);
```

`GeometryInfo` Manipulations

To perform an additional random triangulation -

```
Triangulator triang = new Triangulator(1);
triang.triangulate(gasket);
```

To generate normals for each vertex in the triangle array -

```
NormalGenerator norma = new NormalGenerator();
norma.generateNormal(gasket);
```

To turn the triangle array into a triangle strip array -

```
Stripifier stripper = new Stripifier();
stripper.stripify(gasket);
```

Note that the geometry required by the Shape3D object is obtained from the geometry array in the gasket GeometryInfo object, not gasket itself -

```
Shape3D gasketShape = new Shape3D();
gasketShape.setGeometry(gasket.getGeometryArray());
```