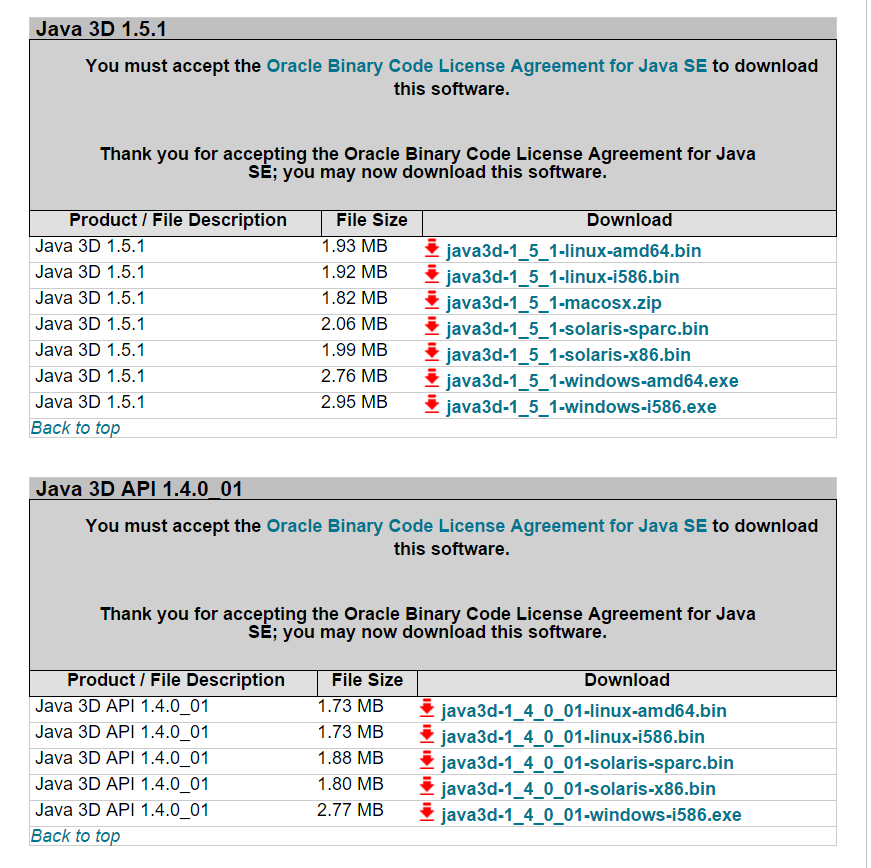
# Java 3D学习笔记

## 一、初识Java 3D

Java 3D是sun公司开发的一套适应于Java语言的3D编程的图形库。它在底层上的基础实际上就是OpenGL或者DirectX，所以Java 3D的版本也有两个，一个是OpenGL版的，一个是DirectX版的。

Java 3D的编程不像Java 2D那样简单，因为Java 2D的编程不需要导入外部jar包，Java 2D的图形标准库已经成为Java的标准库中的一员了。而Java 3D不一样，编写Java 3D的程序，需要导入外部jar包，借助sun公司的工具包才能开发。

### 1.1、Java 3D包的下载和安装：



我在写这个文档的时候，只有这两个版本下载，你们到时候可能会是更高的版本。这个下载页面就是：

|  |
| --- |
| http://www.oracle.com/technetwork/java/api-141528.html |

这是Oracle官网上的下载地址，但是下载的压缩包中没有源码，只有jar包。有jar包我们就能编程，只是没有有源码时方便，但是你可以在这个页面找到Java 3D的帮助文档，看帮助文档和看源码效果一样。

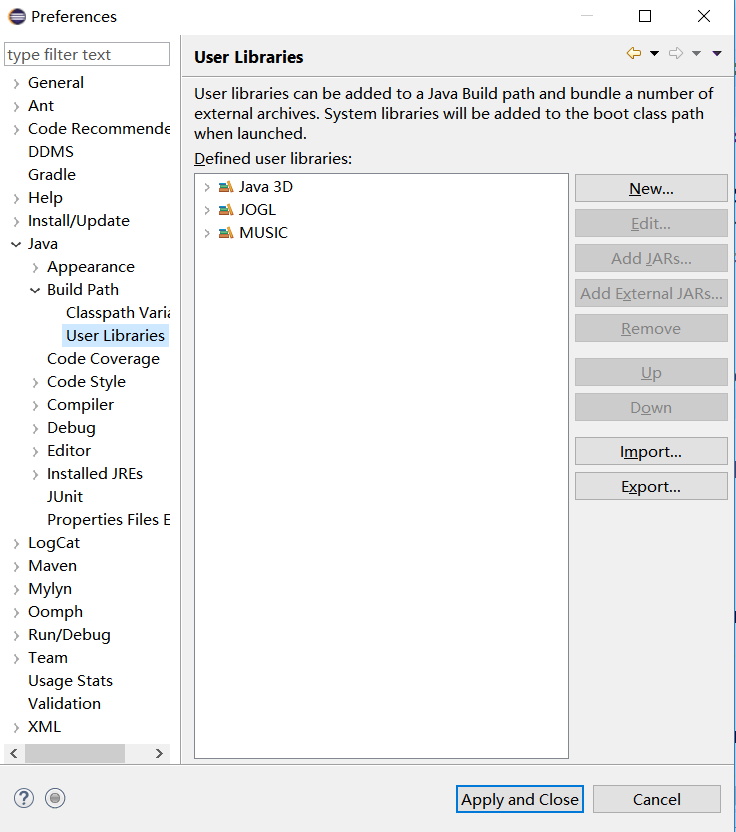
下载完压缩包后，解压缩安装，64位Windows系统的安装路径是：C:\Program Files\Java\Java3D；32位Windows系统的安装路径是：C:\Program Files (x86)\Java\Java3D

安装完成后进入这个文件夹，里面是以Java 3D版本命令的文件夹，点开，会看到三个jar文件，这三个jar文件就是我们所需要的。

### 1.2、Java 3D在eclipse中的配置

如果eclipse中自己配置User Library会的兄弟，你们可以跳过这节，自己将这三个jar文件做成一个User Library然后使用；如果不会配置的兄弟，跟着我继续往下走。

打开eclipse，按照这个顺序：Window -> preference -> Java -> Build Path -> User Library，找到User Library



这三个都是我自己设置的User Library。下面教你怎么设置：

·点击右边的New，设置user library name，也就是设置你的用户库名称，这就跟你给变量取名字一样，可以随便取，但是名字有实际意义更好；

·设置完名字后，鼠标点击自己刚才建立的User Library，会发现右边的一排按钮有些会变得可点击，那咱们就点击add External JARs，添加外部jar包。点击之后就让你选择你要添加的jar包的路径，这就很简单了，就是刚才上面所说的三个jar包所在的目录；

·三个jar包都要add进来，一个不漏

·然后Apply and Close

上面仅仅是建立了自己的库，要想将这个库用在程序上，还得有些步骤：

·新建一个项目，鼠标右键项目名称，点击Build Path，点击Add Library，选择User Library，点击next，然后选择你刚才配置的库（勾上），然后Finish就可以了。

至于怎么测试自己的配置是不是对的呢，这个下一节会讲一个程序，只要这个程序能运行，那就说明配置没问题。

## 二、Java 3D程序结构

多余话咱先别说，先看个程序，我们通过程序慢慢讲解Java 3D的程序结构，以后的学习都先给程序，通过程序学习。

**范例：**在窗口中央显示一个球体

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.Material;  import javax.vecmath.Color3f;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.geometry.Sphere;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();    Sphere sphere = new Sphere(0.5f);  Appearance ap = new Appearance();  ap.setMaterial(new Material());  sphere.setAppearance(ap);  bg.addChild(sphere);      DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere();  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |

### 2.1、场景图

Java 3D图形的绘制是根据一个叫做场景图的数据结构来的。这个数据结构是一棵树，这棵树的叶子节点就是程序需要绘制的对象，而父节点是用来组织叶节点的对象，根节点则是这棵树的核心，但是这个核心不需要我们去费力气编写。

就上面这个程序来说，根节点就是SimpleUniverse对象su，它在Java 3D中被称为虚拟世界，su有一个子节点是BranchGroup类型的对象bg，由su对象的addBranchGroup()方法来完成父子关系的设置。在BranchGroup对象bg下还有几个子节点，分别是Sphere对象和dl对象，sphere是一个球体的实例对象，dl是光照对象，它们都是由bg对象的addChild()方法完成父子关系的设置。总结上面的分析，我们可以得到一棵树：su节点有子节点bg，bg有子节点sphere和dl，这棵树就是最简单的场景图实例。

下面这些结构全部都是基于场景图的概念进行讲解的，例如：根节点，父节点，叶子节点这些都是树中的概念，而场景图就是一棵树。

### 2.2、超结构

超结构是场景图的根节点的组成部分。场景图的根节点是虚拟世界，虚拟世界中有多个局部坐标（Locale），超结构就是由虚拟世界和局部坐标构成的，这两者是Java 3D程序中必不可少的部分。

虚拟世界中的坐标十分精确，表示范围还十分大，如果直接在虚拟世界中表示坐标，将会是极大的浪费，这就是我们所说的杀鸡焉用宰牛刀。这里虚拟世界就是宰牛刀，所以我们在虚拟世界中放置一个局部坐标系，用局部坐标系这把杀鸡刀来杀坐标这只鸡，很合适。

那么为什么上面的程序中没有Locale对象的创建和使用呢？是不是我用宰牛刀来杀鸡了呢？并不是，上面的程序中，Locale的创建包含在SimpleUniverse类中了，我们在实例化su对象的时候，就自动帮我们创建了一个Locale对象，并且将这个Locale对象设置为su的子节点了。

Locale对象除了防止浪费，还有另一个作用，就是加载视图分支和内容分支。

先说内容分支，我们所声明的BranchGroup类型的bg对象，在这个程序中就是一个内容分支，内容分支就是我们需要绘制的东西，内容分支只有作为Locale对象的字节点后才能被绘制。那么为什么上面的程序中是将bg作为su的子节点还能绘制呢？其实不然，我在上一节说bg是su的子类，是因为我们还没学到这来，当学到这里时，就需要纠正了，我们用su对象调用addBranchGroup()方法，并不是将bg设置为su的子节点，而是将bg设置为su对象中包含的Locale对象的子类，只不过这个过程我们从程序上看不到（要是有Java 3D源码就能看到）。

视图分支先不需要了解，在第五章的时候会仔细讲到，前面学的知识并不需要对视图的了解。但是这个程序里面仍然是设置了视图的

|  |
| --- |
| su.getViewingPlatform().setNominalViewingTransform(); |

就是这句代码，你可以理解为设置为默认视图，先不需要纠结

### 2.3、组节点（Group的子类）

Group类是一个抽象类，这个类的子类对象在场景图中扮演的角色就是父节点。

常用的Group节点包括如下几类：  
 ·BranchGroup（附属组节点）：

·TransformGroup（几何变换组节点）：

·SharedGroup（共享组节点）：

其中最简单也是我们必须要理解的是BranchGroup节点，可以说如果一个Java 3D程序中没有这类节点就不可能运行。前面我们已经知道了场景图中超结构，超结构的下一级就是BranchGroup节点，这点从addBranchGroup()方法的方法名称也能看出来，Locale对象只能将BranchGroup节点作为其子节点，而Locale对象子节点的作用只有两个，一个是作为视图分支，一个是作为内容分支，这都必须由BranchGroup来完成。

TransformGroup节点是一个关于几何变换的组节点，TransformGroup对象中会引用一个Transform3D对象，Transform3D对象就是一个几何变换（关于几何变换第三章会讲到），而TransformGroup节点会将这个几何变换作用它的所用子节点。

SharedGroup节点是共享组节点，当需要用到这个节点的时候，直接用例子来讲会更清楚，这里不再赘述。

Group的子类远不止这些，但是其它的类平心而论，我从没用过，就这三个用的多。

### 2.4、叶子节点

如果让我像Group节点那样总结有哪些叶子节点是常用的，这还真没法总结。

叶子节点是最终绘制在窗口中的实际对象，就如同上面程序中的球和光照。

叶子节点都是Leaf类的子类，这点和Group是类似的。最具代表性的叶子节点是几何对象、光照、背景。我说不好总结主要是因为几何对象这块不好总结，这个随着学习的进行我们就能逐步理解了，到那个时候你应该也能体会到这是一种只可意会不可言传的东西。

我们需要知道的是，叶子节点最终的父节点一定要是作为Locale的子节点的BranchGroup对象，我知道这样说可能又有人不明白的了。拿上面的程序来说吧，如果我们要给这个球实施几何变换，那么我们就需要TransformGroup节点，并且将sphere对象作为TransformGroup节点的子节点，由于树中每个节点只能有一个父节点，所以此时，sphere不可能再作为bg的子节点了，但是我们可以让TransformGroup节点作为bg的子节点。所以叶子节点的最终父节点一定要是BranchGroup节点，并且这个BranchGroup节点必须是Locale的子节点。

### 2.5、节点组件

节点组件不是场景图的组成部分，它的作用修饰场景图中的叶子节点，每个节点组件对象可以被多个叶子节点引用。

常用到的节点组件包括：

·Appearance（外观）：

外观包括材质、颜色属性、几何属性、点的属性、线的属性、多边形属性

·Bounds（限制范围）

主要用来限制背景显示范围和灯光显示范围

·Font3D（3D字体）

主要用于绘制3D文本

从上面的程序中可以看出，Appearance对象是被sphere对象所引用了，Bounds对象是被dl对象所引用了。

### 2.6、画板

上面的讲解都是讲解场景图的组成，我们还有一个很重要的东西没讲，那就是Canvas3D对象，这个对象必须要有，因为这就是画板，我们所有的绘制都必须通过它才能显示出来。

Canvas3D是AWT中Canvas的子类，可以直接添加到Applet中，也可以添加到JFrame中（但是重量级组件还是不要用轻量级组件显示）。Java 3D中提供了一个MainFrame类来代替窗口，这个类接受一个Applet对象，后两个参数设置窗口大小。

实例化Canvas3D类必需要GraphicsConfiguration对象。实例化后需要将Canvas3D对象加入一个窗口进行显示（就如同JPanel需要加入JFrame一样）。这样的设置还不算完，我们必须要将Canvas3D对象与虚拟世界关联起来才行，这样才能使虚拟世界中绘制的东西能在窗口中显示。所以在实例化su对象的时候，我们将canvas对象作为参数了，这个内部有点复杂，先阶段的知识解释不了，等以后边学边解释。

但是需要记住的两点：必须把Canvas3D加入容器，必须把Canvas3D与虚拟世界关联。

## 三、几何图形

几何形状在Java3D中可以分为两类：

com.sun.j3d.utils.geometry.Primitive类族和Shape3D类族

### 3.1、Primitive（原始几何图形）

Primitive类定义了Java3D中的原始几何图形的标准，它有四个子类，分别是：

·Box：立方体类

·Cone：圆锥体类

·Cylinder：圆柱体类

·Sphere：球类

查阅文档了解其构造方法即可使用，其他方法都是Primitive类中继承而来的方法，最主要的有以下几个：

|  |  |
| --- | --- |
| 方法名称 | 方法作用 |
| public abstract void setAppearance(Appearance ap) | 给几何体设置外观 |
| public void setAppearance(int partid, Appearance ap) | 给几何体的哪一个面设置外观（例如立方体有六个面，参数一一般都由具体的子类来声明常量） |
| public abstract Shape3D getShape(int partid) | 由几何体得到对应的Shape3D类对象，参数一同样由具体的子类来声明常量 |
| public Appearance getAppearance() | 得到几何体的外观对象 |
| public abstract Appearance getAppearance(int partId) | 得到几何体的具体哪一面的外观对象，参数一同样也是由具体的子类来声明常量 |

常量可以从文档中找到。子类中还有一些自己的方法（例如球类中有获得球体半径的方法），就不多做介绍了，查询文档即可。

### 3.2、Shape3D（构造几何图形）

虽然在Java3D中并没有将Shape3D解释为构造几何类型，但是根据我的理解，这类对象实际上就是我们自己构造的，所以我将之称之为“构造几何图形”。

为什么说是自己构造的呢，因为这类几何体是由顶点组成，而这些顶点和顶点之间的连接顺序都是由我们自己设置的，相当于在构造一个几何体。我们设置的顶点和顶点之间的连接顺序称为几何属性。

Java3D中有两类族表示几何属性：GeometryArray、GeometryInfo。

当我们有了几何属性的对象（也就是GeometryArray对象或GeometryInfo对象）之后，Shape3D中有一个setGeometry()方法，将几何属性对象传进去就能构造几何体了。

#### 3.2.1、GeometryArray类族

这个类族中有关于构造点、构造线、构造三角形、构造四边形的类，下面分别从方法的角度分别介绍这四个类。

##### 3.2.1.1、最简单的四种构造类

**构造点：PointArray类**

**范例：**绘制四个点

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.GeometryArray;  import javax.media.j3d.Material;  import javax.media.j3d.PointArray;  import javax.media.j3d.PointAttributes;  import javax.media.j3d.Shape3D;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();    Appearance ap = new Appearance();  // ap.setMaterial(new Material());    PointArray pa = new PointArray(4, GeometryArray.COORDINATES | GeometryArray.COLOR\_3);  Point3d p[] = new Point3d[4];  Color3f c[] = new Color3f[4];  p[0] = new Point3d(-0.5, 0.5, 0);  p[1] = new Point3d(0.5, 0.5, 0);  p[2] = new Point3d(0.5, -0.5, 0);  p[3] = new Point3d(-0.5, -0.5, 0);  c[0] = new Color3f(1, 0, 0);  c[1] = new Color3f(1, 1, 0);  c[2] = new Color3f(1, 0, 1);  c[3] = new Color3f(1, 1, 1);    pa.setCoordinates(0, p);  pa.setColors(0, c);  Shape3D shape = new Shape3D();  shape.setGeometry(pa);    PointAttributes pAttr = new PointAttributes();  pAttr.setPointSize(10);  pAttr.setPointAntialiasingEnable(false);  ap.setPointAttributes(pAttr);  shape.setAppearance(ap);    bg.addChild(shape);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere();  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |

这个程序与第一个程序的不同之处就在getContent()方法之中。

这里先讲**PointArray的构造方法**

|  |
| --- |
| public PointArray(int vertexCount, int vertexFormat) |

第一个参数表示我们需要绘制的顶点的个数，范例中我们要绘制四个点，这个参数就设置为4；第二个参数表示顶点格式的掩码，主要用到的掩码有如下几个：

·COORDINATES：坐标

·COLOR\_3：不带alpha值的颜色

·COLOR\_4：带alpha值的颜色

·NORMALS：顶点法向量

·TEXTURE\_COORDINATES\_2：2D纹理坐标

·TEXTURE\_COORDINATES\_3：3D纹理坐标

·TEXTURE\_COORDINATES\_4：4维纹理坐标

最后三个掩码对构造点是没有用的，但是对构造其他图形有用

如果第二个参数中有这些掩码，则表示我们的顶点可以有这些属性，范例中给顶点设置了第一个和第二个掩码（是用逻辑或连接的），则表示我们可以给顶点设置坐标，给顶点设置颜色。如果范例中没有COLOR\_3的掩码，运行程序的时候会报错，因为没有设置该掩码，却给顶点设置了颜色。

**PointArray中的普通方法**

|  |  |
| --- | --- |
| 方法名称 | 方法描述 |
| public void setCoordinates(int index, Point3d[] coordinates) | 给PointArray对象中的所有顶点设置坐标，参数一基本上没什么作用，设置为0就行了（如果你想知道参数含义，可以自己设置为1或2尝试），参数二就是保存顶点坐标的数组，每一个顶点坐标由一个Point3d对象（3D坐标）确定。 |
| public void setColors(int index, Color3f[] colors) | 给PointArray对象中的所有顶点设置颜色，参数一与上面方法一样，参数二是保存颜色的数组，每一个顶点的颜色由一个Color3f对象确定 |

虽说PointArray还有其他设置顶点属性的方式，但是有这两种就能完全满足我们的要求。

程序中我们还用到了一个PointAttributes类，这个类是用来设置点的属性的，比如点的大小，点的形状，这里我们将点的大小设置为10，点的形状设置正方形了，用的这两个方法在程序中显而易见，不多做解释，设置形状的方法中，参数设置为false就是正方形，设置为true就是圆形。然后我们将一个外观对象（Appearance对象）引用这个PointAttributes对象，然后将外观设置为几何体的外观（注意：外观对象不能设置材质，不然顶点颜色无法显示）。

**构造线：LineArray**

**范例**：绘制三条线

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.GeometryArray;  import javax.media.j3d.LineArray;  import javax.media.j3d.LineAttributes;  import javax.media.j3d.Shape3D;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();    Appearance ap = new Appearance();  // ap.setMaterial(new Material());    LineArray la = new LineArray(6, GeometryArray.COORDINATES | GeometryArray.COLOR\_3);  Point3d p[] = new Point3d[6];  Color3f c[] = new Color3f[6];  p[0] = new Point3d(0.1, 0.1, 0);  p[1] = new Point3d(0.3, 0.3, 0);  p[2] = new Point3d(0.3, 0.1, 0);  p[3] = new Point3d(0.5, 0.3, 0);  p[4] = new Point3d(0.5, 0.3, 0);  p[5] = new Point3d(0.7, 0.1, 0);    c[0] = new Color3f(1, 0, 0);  c[1] = new Color3f(1, 1, 0);  c[2] = new Color3f(1, 0, 1);  c[3] = new Color3f(1, 1, 1);  c[4] = new Color3f(0, 1, 1);  c[5] = new Color3f(0, 0, 1);    la.setCoordinates(0, p);  la.setColors(0, c);  Shape3D shape = new Shape3D();  shape.setGeometry(la);    LineAttributes lAttr = new LineAttributes();  lAttr.setLineWidth(5);  lAttr.setLinePattern(LineAttributes.PATTERN\_DOT);  ap.setLineAttributes(lAttr);  shape.setAppearance(ap);    bg.addChild(shape);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere();  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |

这个程序不同之处也是getContent()方法，将构造点的部分改为构造线。

LineArray类每两个点连接成一条线，范例中是顶点0和1、2和3、4和5构成的三条线。

**LineArray的构造方法：**public LineArray(int vertexCount, int vertexFormat)；

参数和PointArray的含义相同，LineArray的两个设置顶点属性的方法也完全相同。同时也有一个对应的LineAttributes对象设置线的属性，比如线的宽度，线的格式（加粗、点式线）。

**构造三角形：TrianlgeArray**

**范例：**绘制一个由两个三角形组成的平行四边形

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.GeometryArray;  import javax.media.j3d.LineAttributes;  import javax.media.j3d.PolygonAttributes;  import javax.media.j3d.Shape3D;  import javax.media.j3d.TriangleArray;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();    Appearance ap = new Appearance();  // ap.setMaterial(new Material());    TriangleArray ta = new TriangleArray(6, GeometryArray.COORDINATES | GeometryArray.COLOR\_3);  Point3d p[] = new Point3d[6];  Color3f c[] = new Color3f[6];  p[0] = new Point3d(0, 0, 0);  p[1] = new Point3d(0.6, 0, 0);  p[2] = new Point3d(0.3, 0.3, 0);  p[3] = new Point3d(0.3, 0.3, 0);  p[4] = new Point3d(0.6, 0, 0);  p[5] = new Point3d(0.9, 0.3, 0);    c[0] = new Color3f(1, 0, 0);  c[1] = new Color3f(1, 1, 0);  c[2] = new Color3f(1, 0, 1);  c[3] = new Color3f(1, 1, 1);  c[4] = new Color3f(0, 1, 1);  c[5] = new Color3f(0, 0, 1);    ta.setCoordinates(0, p);  ta.setColors(0, c);  Shape3D shape = new Shape3D();  shape.setGeometry(ta);  PolygonAttributes pAttr = new PolygonAttributes();  pAttr.setPolygonMode(PolygonAttributes.POLYGON\_LINE);  ap.setPolygonAttributes(pAttr);    LineAttributes lAttr = new LineAttributes();  lAttr.setLineWidth(5);  ap.setLineAttributes(lAttr);  shape.setAppearance(ap);    bg.addChild(shape);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere();  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |

Triangle类是每三个点组成一个三角形，范例中是顶点0和1和2、3和4和5构成的两个三角形。

构造方法和设置坐标颜色的普通方法就不再讲，这里讲PolygonAttributes类，这是一个设置多边形属性的类，我这里将多边形模式设置为线条模式，所以三角形会以线条的形式绘制（也就是只绘制三条边），而绘制线条的属性由LineAttributes对象控制，所以当多边形设置为线条绘制模式时，是由LineAttributes对象控制的，还可以设置为点绘制模式（POLYGON\_POINT），这时由PointAttributes对象控制（自行编写程序尝试）。

这里有点需要注意：多边形绘制请按顶点逆时针旋转的方向绘制，否则在窗口中会显示不出爱，这是因为多边形默认显示的是“正面”。至于正面是什么概念，百度即可知晓。

构造四边形我就不讲了，都是类似的，四边形和三角形都是多边形，都可以用PolygonAttributes对象控制显示样式。

##### 3.2.1.2、含有索引的四种构造类

通过上面的三个程序，我们了解到了怎么样自己构造几何图形，虽然我们只叫你怎么构造点、线、三角形和四边形，但是你可根据这四种图形组合成更复杂的平面图形，甚至立体图形。

但是在构造线和三角形的时候，我们会发现，在给顶点设置坐标的时候，由于在几何图形中有些顶点是重复的，所以我们需要将一个相同的坐标设置给不同的点，例如绘制线条的例子中，由顶点2和3、顶点4和5构成的两条线段中，顶点3和顶点4在几何图形中是同一个点，但是我们需要保存两份数据。所以这样是很浪费空间的，也给我们编写程序带来了很多不必要的操作。

接下来我们来解决这个问题。

由于构造点的时候不存在这个问题，直接跳过，讲构造线和构造三角形。

**构造线：IndexedLineArray**

**范例：**还是绘制原来的三条线

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.GeometryArray;  import javax.media.j3d.IndexedLineArray;  import javax.media.j3d.LineAttributes;  import javax.media.j3d.Shape3D;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();    Appearance ap = new Appearance();  // ap.setMaterial(new Material());    IndexedLineArray ila = new IndexedLineArray(5, GeometryArray.COORDINATES | GeometryArray.COLOR\_3, 6);  Point3d p[] = new Point3d[5];  Color3f c[] = new Color3f[5];  p[0] = new Point3d(0.1, 0.1, 0);  p[1] = new Point3d(0.3, 0.3, 0);  p[2] = new Point3d(0.3, 0.1, 0);  p[3] = new Point3d(0.5, 0.3, 0);  p[4] = new Point3d(0.7, 0.1, 0);    c[0] = new Color3f(1, 0, 0);  c[1] = new Color3f(1, 1, 0);  c[2] = new Color3f(1, 0, 1);  c[3] = new Color3f(0, 1, 1);  c[4] = new Color3f(0, 0, 1);    int index[] = {0,1,2,3,3,4};    ila.setCoordinates(0, p);  ila.setCoordinateIndices(0, index);  ila.setColors(0, c);  ila.setColorIndices(0, index);  Shape3D shape = new Shape3D();  shape.setGeometry(ila);    LineAttributes lAttr = new LineAttributes();  lAttr.setLineWidth(5);  lAttr.setLinePattern(LineAttributes.PATTERN\_DOT);  ap.setLineAttributes(lAttr);  shape.setAppearance(ap);    bg.addChild(shape);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere();  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |

**IndexedLineArray的构造方法：**public IndexedLineArray(int vertexCount, int vertexFormat, int indexCount)；第一个参数表示几何图形中实际需要的顶点的个数，范例中虽然需要绘制三条线，但实际上只需要5个顶点就行了，所以参数一设置为5；参数二的含义不变；参数三表示我们完成几何图形的绘制需要的顶点的个数，与参数一不一样，虽然我们绘制这三条线段只需要5个点，但是我们在绘制的时候实际还是绘制了6个点（需要的顶点个数和实际绘制的顶点个数并不相同）。

**IndexedLineArray新增的普通方法：**

|  |  |
| --- | --- |
| 方法名称 | 方法描述 |
| public void setCoordinateIndices(int index, int[] coordinateIndices) | 给IndexedLineArray对象中的顶点坐标设置绘制顺序，参数一设置为0即可，参数二是一个整形数组，按照顶点索引设置绘制顺序 |
| public void setColorIndices(int index, int[] colorIndices) | 给IndexedLineArray对象中的顶点颜色设置绘制顺序，参数二是一个整形数组，按照顶点索引设置绘制顺序 |

范例中我们创建了一个整型数组index[] = {0,1,2,3,3,4}，这个数组的意思是，按照这个数组的顺序，先取顶点0和顶点1构成一条线段，再取顶点2和顶点3构成一条线段，再取顶点3和顶点4构成一条线段。然后我们将这个整型数组作为参数传递给上面这两个方法了。

**构造三角形：IndexedTriangleArray**

**范例：**还是绘制原来的两个三角形

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.GeometryArray;  import javax.media.j3d.IndexedTriangleArray;  import javax.media.j3d.LineAttributes;  import javax.media.j3d.PolygonAttributes;  import javax.media.j3d.Shape3D;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();    Appearance ap = new Appearance();  // ap.setMaterial(new Material());    IndexedTriangleArray ita = new IndexedTriangleArray(4, GeometryArray.COORDINATES | GeometryArray.COLOR\_3, 6);  Point3d p[] = new Point3d[4];  Color3f c[] = new Color3f[4];  p[0] = new Point3d(0, 0, 0);  p[1] = new Point3d(0.6, 0, 0);  p[2] = new Point3d(0.3, 0.3, 0);  p[3] = new Point3d(0.9, 0.3, 0);    c[0] = new Color3f(1, 0, 0);  c[1] = new Color3f(1, 1, 0);  c[2] = new Color3f(1, 0, 1);  c[3] = new Color3f(1, 1, 1);    int index[] = {0,1,2,2,1,3};    ita.setCoordinates(0, p);  ita.setCoordinateIndices(0, index);  ita.setColors(0, c);  ita.setColorIndices(0, index);  Shape3D shape = new Shape3D();  shape.setGeometry(ita);  PolygonAttributes pAttr = new PolygonAttributes();  pAttr.setPolygonMode(PolygonAttributes.POLYGON\_LINE);  ap.setPolygonAttributes(pAttr);    LineAttributes lAttr = new LineAttributes();  lAttr.setLineWidth(5);  ap.setLineAttributes(lAttr);  shape.setAppearance(ap);    bg.addChild(shape);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere();  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |

和IndexedLineArray类似，IndexedTriangleArray的构造方法也多了一个参数，也需要创建一个整型数组来设置绘制顺序，由于三角形是每三个点绘制一个三角形，所以对我们创建的整型数组index = {0,1,2,2,1,3}，是先将顶点0、1、2三个点绘制为一个三角形，顶点2、1、3绘制为一个三角形（注意顶点顺序，2、1、3与2、3、1是不同的，正反面不同）。

##### 3.2.1.3、含有带的四种构造类

除了用含有索引的构造类来解决重复问题，还有另一种方式也能解决重复问题，那就是用含有带的构造类，带的英文是“Strip”。

同样的，我也只讲构造线和构造三角形

**构造线：LineStripArray**

**范例：**还是绘制三条线

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.GeometryArray;  import javax.media.j3d.IndexedTriangleArray;  import javax.media.j3d.LineAttributes;  import javax.media.j3d.LineStripArray;  import javax.media.j3d.PolygonAttributes;  import javax.media.j3d.Shape3D;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();    Appearance ap = new Appearance();  // ap.setMaterial(new Material());    int strip[] = {2,3};  LineStripArray lsa = new LineStripArray(5, GeometryArray.COORDINATES | GeometryArray.COLOR\_3, strip);  Point3d p[] = new Point3d[5];  Color3f c[] = new Color3f[5];  p[0] = new Point3d(0.1, 0.1, 0);  p[1] = new Point3d(0.3, 0.3, 0);  p[2] = new Point3d(0.3, 0.1, 0);  p[3] = new Point3d(0.5, 0.3, 0);  p[4] = new Point3d(0.7, 0.1, 0);    c[0] = new Color3f(1, 0, 0);  c[1] = new Color3f(1, 1, 0);  c[2] = new Color3f(1, 0, 1);  c[3] = new Color3f(0, 1, 1);  c[4] = new Color3f(0, 0, 1);    lsa.setCoordinates(0, p);  lsa.setColors(0, c);  Shape3D shape = new Shape3D();  shape.setGeometry(lsa);  LineAttributes lAttr = new LineAttributes();  lAttr.setLineWidth(5);  ap.setLineAttributes(lAttr);  shape.setAppearance(ap);    bg.addChild(shape);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere();  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |

LineStripArray类的构造方法：public LineStripArray(int vertexCount, int vertexFormat, int[] stripVertexCounts)；参数一表示几何图形实际需要的顶点个数，含义与IndexedLineArray的第一个参数相同；参数二也相同；参数三是一个整型数组，这个整型数组并不是设置顶点绘制顺序的，而是用来设置带，我们把几何图形的相连的部分称为一条带，范例中就有两条带，这应该很明显，并且一条带含有两个顶点，一条带含有三个顶点，所以我们的整型数组strip = {2,3}，整型数组的长度就表示该几何图形中含有的带的条数，而每一个元素则表示各条带所含有的顶点的个数。

由于我们设置了含有两条带，且分别含有两个顶点和三个顶点。所以，当我们将五个坐标和五个颜色分配给五个顶点的时候，LineStripArray对象会将前面两个点作为一条带，后面三个点作为一条带（这与strip数组中数字的顺序有关，{2,3}和{3,2}不一样）。三个点作为一条带，而我们又是绘制线段，所以每相邻的两个点连成线段，这样就达到了我们的需求。

**构造三角形①：TriangleStripArray**

实例上不用再举例子的，构造三角形和构造线段的不同之处在于，三角形需要三个点，而线段只需要两个点。所以我们在创建一个TriangleStripArray对象的时候，仍然需要一个strip的整型数组来设置带，如果还是绘制之前的两个三角形，那么就只有一条带，这条带中有四个顶点，而TriangleStripArray对象会将每相邻的三个顶点构成一个三角形，也就是说4个顶点会构成两个三角形

**构造三角形②：TriangleFanArray**

这个构造类与上面的区别在于带中三角形的构成，上面的是每相邻的三个点构成三角形，而这个类是，每相邻的两个点与第一个点构成三角形。

例如：五个点的分别是顶点0、1、2、3、4，按照上面的构造类会构造0、1、2和1、2、3和2、3、4三个三角形，而按照这个构造类则会构造0、1、2和0、2、3和0、3、4三个三角形。

##### 3.2.1.4、总结GeometryArray

实际上上面这几种构造类中三角形的构造类最常用，四边形有时会用。因为我们可以由三角形生成各种复杂图形，甚至空间图形（例如四面体），四面体就是由四个三角形构成。后面我们会讲利用多边形网格绘制曲面图像的方法。

#### 3.2.2、GeometryInfo类族

**范例：**构造一个四面体

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Alpha;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.Material;  import javax.media.j3d.RotationInterpolator;  import javax.media.j3d.Shape3D;  import javax.media.j3d.TransformGroup;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.geometry.GeometryInfo;  import com.sun.j3d.utils.geometry.NormalGenerator;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();  TransformGroup spin = new TransformGroup();  spin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);    // 设置旋转  RotationInterpolator rotator = new RotationInterpolator(new Alpha(-1, 4000), spin);  rotator.setSchedulingBounds(new BoundingSphere());  spin.addChild(rotator);    Appearance ap = new Appearance();  ap.setMaterial(new Material());    Point3d p[] = new Point3d[4];  p[0] = new Point3d(0, 1, 0);  p[1] = new Point3d(-Math.cos(Math.PI/6), -Math.sin(Math.PI/6), 0);  p[2] = new Point3d(Math.cos(Math.PI/6), -Math.sin(Math.PI/6), 0);  p[3] = new Point3d(0, 0, 1);    int index[] = {2,1,0, 0,1,3, 0,3,2, 1,2,3};  int strip[] = {3,3,3,3};    GeometryInfo gi = new GeometryInfo(GeometryInfo.POLYGON\_ARRAY);  gi.setCoordinates(p);  gi.setCoordinateIndices(index);    gi.setStripCounts(strip);  new NormalGenerator().generateNormals(gi);    Shape3D shape = new Shape3D();  shape.setGeometry(gi.getGeometryArray());  shape.setAppearance(ap);    spin.addChild(shape);  bg.addChild(spin);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere();  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |

GeometryInfo比GeometryArray更好用，因为它能构造出边的数量任意多的多边形，而GeometryArray只能构造三角形和四边形，想要更多的边只能组合。

**GeometryInfo的构造方法**

|  |
| --- |
| public GeometryInfo(GeometryArray ga)； |
| public void GeometryInfo(int primitive)； |

上面的构造方法接收一个GeometryArray参数，这使得这两类族可以互相转化。但是我们常用的构造方法会是第二个，它接收一个表示样式的参数，所有的样式都在GeometryInfo中定义为常量了，我们用的最多的就是POLYGON\_ARRAY这个构造多边形的样式，如果你想构造其他的样式，建议选择GeometryArray，那更为方便。

在GeometryInfo构造方法中不必声明顶点所包含的属性（坐标、颜色等），可以直接给顶点设置坐标值，颜色值，甚至纹理坐标也可以。

从程序中我们可以看出，在GeometryInfo中包含有索引数组和带数组两个整型数组。我们拿上面的TriangleStripArray的范例为例，我们只有一个带数组，每个带中每相邻的三个点构成一个三角形，例如0和1和2、1和2和3，但是如果我现在需要是0和1和2、0和1和3这两个三角形，那么就做不到了，或者说想要达到这种需求，程序员就得调整每个顶点的顺序，一是给编程带来不便，二是阅读麻烦。

但是如果我们现在拥有的不仅仅只有带数组，还有索引数组，那么是不是就很方便，我们声明一个带数组来告诉程序我们需要构造多少个带，每个带中有多少顶点，至于每个带中的顶点是哪些顶点，这就由索引数组告诉程序了，索引数组按照带的顺序给带分配顶点。

范例中带数组中是{3,3,3,3}，所以索引数组会将每三个点分配给一个带，如果带数组是{5,2,6,4}，那么索引数组中的前五个顶点分配给第一个带，接下来的两个顶点分配给第二个带，再接下来的六的顶点分配给第三个带，再接下来的四个顶点分配给第四个带。

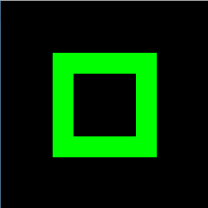
要是索引数组中的顶点数不够或者多了会怎么样？肯定会报错，不用想，赶紧检查程序。

GeometryInfo新增的方法：

·public void setStripCounts(int[] stripCounts)；

·public void setContourCounts(int[] contourCounts)；

第一个方法用来设置带数组；第二个方法用来设置多边形的轮廓数组，有些多边形内部有空洞，用这就能设置。下面这个绿色部分就是一个带有内部空洞的多边形，我们从外置内，由右上角的顶点开始逆时针给顶点编号为0,1,2,3,4,5,6,7，在构造这个多边形时，我们的索引数组为{0,1,2,3,4,5,6,7}，带数组为{4,4}，轮廓数组为{2}。前两个数组应该没什么问题吧，对轮廓数组来说，数组的长度代表几何图形中不相连的多边形的个数，这个和带不一样，带指的是线不相连，这里指的是多边形不相连，数组的每个元素表示这个多边形中轮廓的数量，下面这个图就有两个轮廓。我们这里只有一个多边形，所以轮廓数组只有一个元素，而这个多边形包含有一个外边界和一个内边界，共两个轮廓，所以数组元素值为2。



带有空洞的多边形只有在填充多边形的时候才有作用，如果多边形用线条绘制，那就没什么意义了。

GeometryInfo还有一点，就是有一个类能自动帮GeometryInfo对象中所有顶点生成法向量：NormalGenerator类，这个类中有一个方法：public void generateNormals(GeometryInfo gi)，接收一个GeometryInfo对象，然后自动生成法向量。

这里我没有给顶点设置颜色，实际上也是可以设置的，自己尝试即可。

#### 3.2.3、多边形网格构造法

范例：画出曲面y=2cos(x2)sin(z2)/e0.25(x\*x+z\*z)-1

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Alpha;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.Material;  import javax.media.j3d.RotationInterpolator;  import javax.media.j3d.Shape3D;  import javax.media.j3d.Transform3D;  import javax.media.j3d.TransformGroup;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.geometry.GeometryInfo;  import com.sun.j3d.utils.geometry.NormalGenerator;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();  Transform3D t = new Transform3D();  t.setScale(0.2);  TransformGroup spin = new TransformGroup(t);    Appearance ap = new Appearance();  ap.setMaterial(new Material());    int m=40, n=40;  Point3d p[] = new Point3d[m\*n];    // 生成多边形网格的所有顶点坐标  int idx = 0;  for(int i=0;i<m;i++) {  for(int j=0;j<n;j++) {  double x = (i-m/2)\*0.2;  double z = (j-n/2)\*0.2;  double y = 2\*Math.cos(x\*x)\*Math.sin(z\*z)/(Math.pow(Math.E, 0.25\*(x\*x+z\*z)))-1;  p[idx++] = new Point3d(x, y, z);  }  }    // 生成多边形网格的索引数组和带数组  int strip[] = new int[(n-1)\*(m-1)];  int index[] = new int[4\*(n-1)\*(m-1)];  idx = 0;  for(int i=0;i<m-1;i++) {  for(int j=0;j<n-1;j++) {  strip[idx++] = 4;  }  }    idx = 0;  for(int i=0;i<m-1;i++) {  for(int j=0;j<n-1;j++) {  index[idx++] = i\*n+j;  index[idx++] = (i+1)\*n+j;  index[idx++] = (i+1)\*n+j+1;  index[idx++] = i\*n+j+1;  }  }    GeometryInfo gi = new GeometryInfo(GeometryInfo.POLYGON\_ARRAY);  gi.setCoordinates(p);  gi.setCoordinateIndices(index);  gi.setStripCounts(strip);    new NormalGenerator().generateNormals(gi);    Shape3D shape = new Shape3D();  shape.setGeometry(gi.getGeometryArray());  shape.setAppearance(ap);    spin.addChild(shape);  bg.addChild(spin);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere(new Point3d(0,0,0), 10);  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |
|  |

这种多边形构造法是一定要掌握，不然你构造不了复杂的立体图形。我们来分析这个例子，首先我声明了一个m、n，这两个量将一个平面分成了m\*n个方格，仔细在脑海中想想，有一个坐标轴，然后有一个m行n列的网格对称的分布在坐标系中，我们就在这个网格中生成m\*n个点，作为我们需要的x坐标和z坐标，然后根据方程求出y坐标。这样我们就得到m\*n个点，虽然它们不在同一个平面，但是我们在生成索引数组的时候，把它当做在同一平面：

我们将**第i行第j列、第i+1行第j列、第i+1行第j+1列、第i行第j+1列作为一个带**。在脑海中想象一下这个场景。我就只能解释这么多了，理解的部分只能靠自己了。

### 3.3、绘制文字

如果学了一个图形库，不会在窗口中绘制文字，那是最烦的。

Java3D提供了两类文字对象：Text2D、Text3D

很明显Text2D是表示2D的文字，Text3D表示3D的文字。其实它们的区别并不仅仅是在此。

Text2D类是Shape3D的子类，也就是说它可以直接作为场景图的叶节点进行绘制。**Text2D类的构造方法：public Text2D(String text, Color3f color, String fontName, int fontSize, int fontStyle)**；参数一就是文字的内容；参数二是文字的显示的颜色；后三个参数很熟悉吧，就是Java中的Font类的三个参数，分别表示字体名称、字体大小、字体样式。

|  |
| --- |
| Text2D text = new Text2D(“Hello”, new Color3f(Color.BLUE), “宋体”, 20, Font.BOLD); |

这就是一句创建Text2D对象的语句，很简单吧。

Text3D类继承了Geometry类，意味着它是作为几何属性出现的，就像我们上面讲的GeometryArray和GeometryInfo一样，需要将这个几何属性交给一个Shape3D类对象。

创建一个Text3D对象，需要一个3D的字体对象和文字内容（字符串），后一个好解决，前面那个也好解决。3D的字体类是Font3D类，**Font3D的构造方法**：public Font3D(java.awt.Font font, FontExtrusion extrudePath)；第一个参数就是一个Font对象，第二个参数定义了字体的凸出量，这样才有立体的感觉。

|  |
| --- |
| Font3D f = new Font3D(new Font(“宋体”, Font.PLAIN, 20), new FontExtrusion());  Text3D t = new Text3D(f, “Hello”);  Shape3D s = new Shape3D();  s.setGeometry(t); |

第一句程序实例化了一个Font3D对象，第二句程序实例化了一个Text3D对象，接下来把这个对象t作为几何属性交给一个Shape3D，就能完成文字的绘制了。

### 3.4、外观和材质

这部分是我最不愿意讲的，因为讲起来很麻烦，而且一般也用不上，只有几个比较常用。

外观就是一个Appearance对象，我们给Appearance对象设置各种属性来完善外观，然后将外观应用于一个或多个几何体，这就是外观对象从实例化到应用的过程，其中最重要最复杂的是第二步，完善外观。

首先之前我已经讲过的PointAttributes、LineAttributes、PolygonAttributes这三个就属于外观的属性，用来控制多边形的绘制；

其次是着色模型，当既没有定义光照也没有定义顶点颜色的时候，ColoringAttributes对象可用于定义几何体的颜色。但是很不幸，我们一般会在构造几何体的时候给顶点定义颜色或者是在程序中定义了光照，所以这个东西一般不会用到。

透明度：TransparencyAttributes，可以控制可视对象的透明度绘制，透明度的值在0.0~1.0之间，0.0表示完全不透明，1.0表示完全透明。下面定义一个透明度为0.5的TransparencyAttributes对象，构造方法的第一个参数是TransparencyAttributes类中定义的常量，自行查阅文档了解，第二个参数就是定义的透明度。

|  |
| --- |
| TransparencyAttributes ta = new TransparencyAttributes(TransparencyAttributes.BLENDED, 0.5); |

**材质：**这是所有属性中最有用的，因为只有定义了这个属性，可视对象才会接受光照，这个在讲光照的时候再讲解。

## 四、几何变换

几何变换这个东西，如果是想仅仅会使用，那是非常好教的，如果想让你们理解，那就有点难了。所以我决定只让你们会用就可以了。

### 4.1、Transform3D类

这个类是Java3D中专门用来进行几何变换的类。我们经常用到的几何变换也就这几种：平移、旋转、放缩、反射、错位（用的并不多）。

Transform3D中的重要方法

|  |  |
| --- | --- |
| 方法名称 | 方法描述 |
| public final void setIdentity() | 清空之前设置的所有变换 |
| public final void setRotation(AxisAngle4d a1) | 将当前变换设置为旋转变换，具体怎么变换看参数 |
| public final void setScale(double scale) | 将当前变换设置为放缩变换，参数为放缩倍数 |
| public final void setTranslation(Vector3d trans) | 将当前变换设置为平移变换，具体怎么变换看参数 |
| public final void mul(Transform3D t1) | 将当前变换与参数所代表的变换合并 |

接下来讲AxisAngle4d怎么用，我们看看这个类的构造方法：

|  |
| --- |
| public AxisAngle4d(Vector3d axis, double angle)； |

第一个参数是一个向量，表示旋转变换的轴，第二个参数表示旋转的角度（弧度制），旋转的方向符合右手定则，就是用右手握住这个旋转轴，大拇指指向旋转轴向量的方向，四指环绕的方向就是旋转方向。

平移变换的参数就是一个向量，将这个向量的起点放在变换前的位置，则向量的终点就是变换后的位置，再通俗一点，new Vector(1, 1, 0)表示的是向x轴正向移动1，再向y轴正向移动1，z轴方向不变化。

而这个mul方法就厉害了，例如我现在有一个t1表示一个平移变换、一个t2表示旋转变换，如果t1.mul(t2)，则t1就变成了先旋转再平移的变换，t2不变，也就是它和t2合并了，记住是先旋转再平移；如果是t2.mul(t1)，则t2就会变成先平移再旋转的变换，t1不变。千万不要把合并之后的变换顺序搞错了，是从后往前的顺序。

**范例：**

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.GraphicsConfiguration;  import java.awt.event.KeyEvent;  import java.awt.event.KeyListener;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.Material;  import javax.media.j3d.Transform3D;  import javax.media.j3d.TransformGroup;  import javax.vecmath.AxisAngle4d;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3d;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.geometry.Sphere;  import com.sun.j3d.utils.universe.SimpleUniverse;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    private TransformGroup trans = new TransformGroup();  private Transform3D t = new Transform3D();  private int flag = 0;    public DocDemo() {  this.trans.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);    this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);  canvas.addKeyListener(new KeyListener() {  @Override  public void keyTyped(KeyEvent e) {    }  @Override  public void keyReleased(KeyEvent e) {    }  @Override  public void keyPressed(KeyEvent e) {  if(e.getKeyCode() == KeyEvent.VK\_ENTER) {  flag++;  if(flag == 3) {  flag = 0;  }  return;  }  char ch = e.getKeyChar();  Transform3D tf = new Transform3D();  trans.getTransform(tf);  t.setIdentity();  if(flag == 0) {  if(ch == 'a') {  t.setTranslation(new Vector3d(-0.1, 0, 0));  } else if(ch == 'd') {  t.setTranslation(new Vector3d(0.1, 0, 0));  } else if(ch == 'w') {  t.setTranslation(new Vector3d(0, 0.1, 0));  } else if(ch == 's') {  t.setTranslation(new Vector3d(0, -0.1, 0));  }  } else if(flag == 1) {  if(ch == 'a') {  t.setRotation(new AxisAngle4d(new Vector3d(0, 1, 0), Math.PI/90));;  } else if(ch == 'd') {  t.setRotation(new AxisAngle4d(new Vector3d(0, 1, 0), -Math.PI/90));;  }  } else if(flag == 2) {  if(ch == 'w') {  t.setScale(0.5);  } else {  t.setScale(2);  }  }  tf.mul(t);  trans.setTransform(tf);  }  });    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();  Transform3D t = new Transform3D();  t.setScale(0.2);  TransformGroup spin = new TransformGroup(t);    Appearance ap = new Appearance();  ap.setMaterial(new Material());    Sphere sphere = new Sphere(0.2f, ap);    this.trans.addChild(sphere);  spin.addChild(trans);  bg.addChild(spin);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere(new Point3d(0,0,0), 10);  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }  } |

程序中我创建了一个可视对象Sphere对象，然后添加了一个键盘监听事件，定义了一个flag变量，每次按enter键，flag变化一次，而根据flag的不同，我们在按WASD时会给这个球设置不同的几何变换。在你按wasd的时候，可能你按d之后，球会往左边移动，这个学过Java2D或OpenGL的都会明白，这就留给你们自己去找答案了。

### 4.2、利用变换构造几何体

一个点，绕着另一个点旋转一周可以得到一个圆；圆绕着自己的直径旋转半圈可以得到一个球；一个矩形通过拉伸可以得到一个立方体。我们可以通过已有的形状，加上几何变换得到新的几何图形。

**范例：**构造一个圆环

|  |
| --- |
| package fuhao.shape;  import javax.media.j3d.Shape3D;  import javax.media.j3d.Transform3D;  import javax.vecmath.AxisAngle4d;  import javax.vecmath.Point3d;  import javax.vecmath.Vector3d;  import com.sun.j3d.utils.geometry.GeometryInfo;  import com.sun.j3d.utils.geometry.NormalGenerator;  public class Torus extends Shape3D {  public Torus(double r1, double r2) {  int m = 20;  int n = 40;    Point3d[] p = new Point3d[m\*n];  double theta1 = 2\*Math.PI/m; // 生成圆的旋转角度  double theta2 = 2\*Math.PI/n; // 生成圆环的旋转角度    p[0] = new Point3d(0, r2, 0); // 生成圆的第一个点  Transform3D circle = new Transform3D(); // 总变换  circle.setIdentity();    Transform3D circle1 = new Transform3D();  circle1.setTranslation(new Vector3d(0, (r2-r1)/2+r1, 0)); // 逆平移变换  circle.mul(circle1);  circle1.setIdentity();  circle1.setRotation(new AxisAngle4d(new Vector3d(1, 0, 0), theta1)); // 旋转变换  circle.mul(circle1);    circle1.setIdentity();  circle1.setTranslation(new Vector3d(0, -(r2-r1)/2-r1, 0)); // 平移变换  circle.mul(circle1);    // 生成圆的过程  for(int i=0;i<m-1;i++) {  p[i+1] = new Point3d();  circle.transform(p[i], p[i+1]);  }    Transform3D torus = new Transform3D();  torus.setRotation(new AxisAngle4d(new Vector3d(0, 0, -1), theta2));    // 生成圆环的过程  for(int i=0;i<n-1;i++) {  for(int j=0;j<m;j++) {  p[(i+1)\*m+j] = new Point3d();  torus.transform(p[i\*m+j], p[(i+1)\*m+j]);  }  }    // 设置带的数量  int idx=0;  int strip[] = new int[m\*n];  for(int i=0;i<m;i++) {  for(int j=0;j<n;j++) {  strip[idx++] = 4;  }  }    // 生成索引数组  int index[] = new int[4\*m\*n];  idx = 0;  for(int i=0;i<n;i++) {  for(int j=0;j<m;j++) {  // 第一个  index[idx++] = i\*m+j;  // 第二个  if(j==m-1) {  index[idx++] = i\*m;  } else {  index[idx++] = i\*m+j+1;  }  // 第三个  if(i==n-1) {  if(j==m-1) {  index[idx++] = 0;  } else {  index[idx++] = j+1;  }  } else {  if(j==m-1) {  index[idx++] = (i+1)\*m;  } else {  index[idx++] = (i+1)\*m+j+1;  }  }  // 第四个  if(i==n-1) {  index[idx++] = j;  } else {  index[idx++] = (i+1)\*m+j;  }  }  }  GeometryInfo gi = new GeometryInfo(GeometryInfo.POLYGON\_ARRAY);  gi.setCoordinates(p);  gi.setCoordinateIndices(index);  gi.setStripCounts(strip);    new NormalGenerator().generateNormals(gi);  this.setGeometry(gi.getGeometryArray());  }  } |

我们从一个点出发生成了一个圆，然后绕圆外的一直线旋转一周，得到一个圆环，这其中最难的部分在于计算带数组和索引数组，一定要一句一句看代码，读一句思考一下，把这种构造几何体的思想学会。

## 五、视图

视图的操作可以分为两类：一类是投影操作，一类是拾取操作。投影操作是将3D空间中的物体投影到2D平面上显示；拾取操作是将2D平面上的点对应到3D空间中的图形。

### 5.1、投影操作

对视图的定义分为两种模式，一种是Java3D模式，一种是OpenGL传统模式。

**范例：**定义观察模型：观察点在（0,0,1）处，观察目标在（0,0,0）处，上方向为y轴正向，定义可视空间：视域为60°，宽高比为1，最近可视距离为0.001，最远可视距离为2。

|  |
| --- |
| private BranchGroup getView(Canvas3D canvas,  Point3d eye, Point3d center, Vector3d up,  double fovy, double aspect, double zNear, double zFar) {  BranchGroup bgView = new BranchGroup();    View view = new View();  view.setProjectionPolicy(View.PERSPECTIVE\_PROJECTION);  view.setPhysicalBody(new PhysicalBody());  view.setPhysicalEnvironment(new PhysicalEnvironment());  view.addCanvas3D(canvas);  view.setCompatibilityModeEnable(true);    ViewPlatform vp = new ViewPlatform();  view.attachViewPlatform(vp);    Transform3D viewMode = new Transform3D();  viewMode.lookAt(eye, center, up);  view.setVpcToEc(viewMode);    Transform3D projection = new Transform3D();  projection.perspective(fovy, aspect, zNear, zFar);  view.setLeftProjection(projection);    bgView.addChild(vp);  return bgView;  } |
| VirtualUniverse vu = new VirtualUniverse();  Locale loc = new Locale(vu);  Point3d eye = new Point3d(0, 0, 1);  Point3d center = new Point3d(0, 0, 0);  Vector3d up = new Vector3d(0, 1, 0);  double fovy = Math.PI/3;  double aspect = 1;  double zNear = 0.25;  double zFar = 2;  BranchGroup bgView = this.getView(canvas, eye, center, up, fovy, aspect, zNear, zFar);  loc.addBranchGraph(bgView); |

这是OpenGL的传统模式来创建视图，基本上所有的操作都是由View类对象完成的。首先，view对象必须调用setCompatibilityModeEnable()方法启动兼容模式，这个兼容模式是Java3D对类似OpenGL中视图规范的传统照相机模型的支持，之后我们需要两个Transform3D对象分别定义观察模型和可视空间，定义观察模型调用lookAt()方法，定义可视空间调用perspective()方法，然后将观察模型和可视空间都用相应的方法传递给view对象，即可完成视图的创建。

|  |
| --- |
| private BranchGroup getView(Canvas3D canvas,  Point3d eye, Point3d center, Vector3d up,  double fovy, double zNear, double zFar) {  BranchGroup bgView = new BranchGroup();    View view = new View();  view.setProjectionPolicy(View.PERSPECTIVE\_PROJECTION);  view.setPhysicalBody(new PhysicalBody());  view.setPhysicalEnvironment(new PhysicalEnvironment());  view.addCanvas3D(canvas);  ViewPlatform vp = new ViewPlatform();  view.attachViewPlatform(vp);    Transform3D trans = new Transform3D();  trans.lookAt(eye, center, up);  trans.invert();  TransformGroup tg = new TransformGroup(trans);    view.setFieldOfView(fovy);  view.setFrontClipDistance(zNear);  view.setBackClipDistance(zFar);    tg.addChild(vp);  bgView.addChild(tg);  return bgView;  } |
| VirtualUniverse vu = new VirtualUniverse();  Locale loc = new Locale(vu);  Point3d eye = new Point3d(0, 0, 1);  Point3d center = new Point3d(0, 0, 0);  Vector3d up = new Vector3d(0, 1, 0);  double fovy = Math.PI/3;  double zNear = 0.25;  double zFar = 2;  BranchGroup bgView = this.getView(canvas, eye, center, up, fovy, zNear, zFar);  loc.addBranchGraph(bgView); |

这是Java3D创建视图的模式，这里我们没有启动兼容模式，利用Transform3D类和TransformGroup类完成观察模型的定义（这里要注意的是，trans调用了invert()方法，这是一个求逆矩阵的方法，如果在几何变换中你们自行了解了几何变换的原理，就会明白的），然后用view对象完成了可视空间的部分定义，没有定义宽高比的方法，当然这个是无关紧要的。

我们从程序中还能看到，视图分支只能附属于一个Locale对象，如果我们创建的不是SimpleUniverse对象，那么就必须自己实例化一个Locale对象。

### 5.2、拾取操作

举个很简单的例子，我们在Canvas3D中绘制了一个Sphere对象后，我们希望当我们点击这个球的时候，会有一些响应，例如球的颜色改变等等，这就需要拾取操作的支持。

**范例**：用线条样式绘制四种基本几何图形（球、圆锥、圆柱、立方体），当点击对应的图形时，给它们设置上外观材质，使其接收光照

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.Dimension;  import java.awt.GraphicsConfiguration;  import java.awt.event.MouseEvent;  import java.awt.event.MouseListener;  import java.util.Enumeration;  import javax.media.j3d.Alpha;  import javax.media.j3d.AmbientLight;  import javax.media.j3d.Appearance;  import javax.media.j3d.Background;  import javax.media.j3d.Behavior;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.Locale;  import javax.media.j3d.Material;  import javax.media.j3d.Node;  import javax.media.j3d.PhysicalBody;  import javax.media.j3d.PhysicalEnvironment;  import javax.media.j3d.PointLight;  import javax.media.j3d.PolygonAttributes;  import javax.media.j3d.RotationInterpolator;  import javax.media.j3d.Shape3D;  import javax.media.j3d.Transform3D;  import javax.media.j3d.TransformGroup;  import javax.media.j3d.View;  import javax.media.j3d.ViewPlatform;  import javax.media.j3d.VirtualUniverse;  import javax.media.j3d.WakeupOnElapsedTime;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.Point3f;  import javax.vecmath.Vector3d;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.geometry.Box;  import com.sun.j3d.utils.geometry.Cone;  import com.sun.j3d.utils.geometry.Cylinder;  import com.sun.j3d.utils.geometry.Primitive;  import com.sun.j3d.utils.geometry.Sphere;  import com.sun.j3d.utils.picking.PickCanvas;  import com.sun.j3d.utils.picking.PickResult;  import com.sun.j3d.utils.picking.PickTool;  import com.sun.j3d.utils.universe.SimpleUniverse;  class MyBehavior extends Behavior {  private TransformGroup tg = null;  public MyBehavior(TransformGroup tg) {  this.tg = tg;  }  @Override  public void initialize() {  this.wakeupOn(new WakeupOnElapsedTime(1000));  }  @Override  public void processStimulus(Enumeration e) {  Transform3D t = new Transform3D();  t.setTranslation(new Vector3d(0.01, 0, 0));    Transform3D t1 = new Transform3D();  this.tg.getTransform(t1);  t.mul(t1);    this.tg.setTransform(t);  this.wakeupOn(new WakeupOnElapsedTime(10));  }  }  @SuppressWarnings("serial")  public class Demo extends Applet {  **private PickCanvas pc;**  private Canvas3D canvas;  public Demo() {  this.setPreferredSize(new Dimension(700, 700));  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  this.canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);  canvas.addMouseListener(new MouseListener() {  @Override  public void mouseReleased(MouseEvent e) {  }  @Override  public void mousePressed(MouseEvent e) {  }  @Override  public void mouseExited(MouseEvent e) {  }  @Override  public void mouseEntered(MouseEvent e) {  }  @Override  public void mouseClicked(MouseEvent e) {  **pc.setShapeLocation(e);**  **PickResult[] res = pc.pickAll();**  **if(res != null) {**  **for(int i=0;i<res.length;i++) {**  **Node node = res[i].getObject();**  **if(node instanceof Shape3D) {**  **Appearance ap = createMaterialAppearance();**  **((Shape3D) node).setAppearance(ap);**  **}**  **}**  **}**  }  });    VirtualUniverse vu = new VirtualUniverse();  Locale loc = new Locale(vu);    Point3d eye = new Point3d(0, 0, 1);  Point3d center = new Point3d(0, 0, 0);  Vector3d up = new Vector3d(0, 1, 0);  double fovy = Math.PI/3;  double zNear = 0.1;  double zFar = 2;    BranchGroup bgView = this.getView(canvas, eye, center, up, fovy, zNear, zFar);  loc.addBranchGraph(bgView);    BranchGroup bgContent = this.getContent();  loc.addBranchGraph(bgContent);    **this.pc = new PickCanvas(canvas, bgContent);**  **this.pc.setMode(PickTool.GEOMETRY);**  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();  TransformGroup spin = new TransformGroup();  bg.addChild(spin);  spin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);    BoundingSphere bound = new BoundingSphere();  Appearance ap = this.createLineAppearance();  // 设置旋转  RotationInterpolator ratator = new RotationInterpolator(new Alpha(-1, 4000), spin);  ratator.setSchedulingBounds(bound);  spin.addChild(ratator);    Sphere sphere = new Sphere(0.2f, Primitive.ENABLE\_APPEARANCE\_MODIFY | **Primitive.ENABLE\_GEOMETRY\_PICKING** | Primitive.GENERATE\_NORMALS, ap);  Cone cone = new Cone(0.1f, 0.1f, Primitive.ENABLE\_APPEARANCE\_MODIFY | **Primitive.ENABLE\_GEOMETRY\_PICKING** | Primitive.GENERATE\_NORMALS, ap);  Cylinder cylinder = new Cylinder(0.1f, 0.1f, Primitive.ENABLE\_APPEARANCE\_MODIFY | **Primitive.ENABLE\_GEOMETRY\_PICKING** | Primitive.GENERATE\_NORMALS, ap);  Box box = new Box(0.1f, 0.1f, 0.1f, Primitive.ENABLE\_APPEARANCE\_MODIFY | **Primitive.ENABLE\_GEOMETRY\_PICKING** | Primitive.GENERATE\_NORMALS, ap);    Transform3D t = new Transform3D();    t.setTranslation(new Vector3d(0, -0.3, 0));  TransformGroup sphereTrans = new TransformGroup(t);  sphereTrans.addChild(sphere);    t.setTranslation(new Vector3d(-0.5, 0, 0));  TransformGroup coneTrans = new TransformGroup(t);  coneTrans.addChild(cone);    t.setTranslation(new Vector3d(0.5, 0, 0));  TransformGroup cylinderTrans = new TransformGroup(t);  cylinderTrans.addChild(cylinder);    t.setTranslation(new Vector3d(0, 0.3, 0));  TransformGroup boxTrans = new TransformGroup(t);  boxTrans.addChild(box);    spin.addChild(sphereTrans);  spin.addChild(coneTrans);  spin.addChild(cylinderTrans);  spin.addChild(boxTrans);    // 设置光源  PointLight light1 = new PointLight(new Color3f(Color.GREEN), new Point3f(-5, 5, 5), new Point3f(1, -1, -1));  light1.setInfluencingBounds(bound);  bg.addChild(light1);    PointLight light2 = new PointLight(new Color3f(Color.BLUE), new Point3f(5, 5, 5), new Point3f(-1, -1, -1));  light2.setInfluencingBounds(bound);  bg.addChild(light2);    AmbientLight light3 = new AmbientLight(true, new Color3f(Color.orange));  light3.setInfluencingBounds(bound);  bg.addChild(light3);    // 设置背景  Background back = new Background(new Color3f(Color.WHITE));  back.setApplicationBounds(bound);  // bg.addChild(b);    return bg;  }    private Appearance createLineAppearance() {  Appearance ap = new Appearance();  PolygonAttributes pa = new PolygonAttributes();  pa.setPolygonMode(PolygonAttributes.POLYGON\_LINE);  ap.setPolygonAttributes(pa);  return ap;  }    private Appearance createMaterialAppearance() {  Appearance ap = new Appearance();  ap.setMaterial(new Material());  return ap;  }    public static void main(String[] args) {  new MainFrame(new Demo(), 700, 700);  }    private BranchGroup getView(Canvas3D canvas,  Point3d eye, Point3d center, Vector3d up,  double fovy, double zNear, double zFar) {  BranchGroup bgView = new BranchGroup();    View view = new View();  view.setProjectionPolicy(View.PERSPECTIVE\_PROJECTION);  view.setPhysicalBody(new PhysicalBody());  view.setPhysicalEnvironment(new PhysicalEnvironment());  view.addCanvas3D(canvas);  ViewPlatform vp = new ViewPlatform();  view.attachViewPlatform(vp);    Transform3D trans = new Transform3D();  trans.lookAt(eye, center, up);  trans.invert();  TransformGroup tg = new TransformGroup(trans);    view.setFieldOfView(fovy);  view.setFrontClipDistance(zNear);  view.setBackClipDistance(zFar);    tg.addChild(vp);  bgView.addChild(tg);  return bgView;  }  } |

这段程序中除了拾取操作，其他的我们都很熟悉。这里我们用到了三个类：PickCanvas、PickTool、PickResult。

PickCanvas会将一个内容分支与画布相关联，从而能从画布上拾取到相关内容分支中的可视物体。当我们点击画布的时候，会产生一个点击事件（MouseEvent），我们将这个点击事件传递给PickCanvas对象，它会从点击的位置来拾取相关的3D可视对象，拾取的结果可由pickAll()方法返回，由PickResult接收，由于2D平面中的一个点对应于3D空间中可能会是多个可视对象，所以要用数组接收。

## 六、光照与纹理

光照模型和纹理的使用使得场景的绘制更加真实，也更加方便。

### 6.1、光源

Java3D的光照模型中，光源有四类：方向性光源、环境光源、点光源、聚光光源。这些光源类都是Light类的子类

**Light类**

|  |  |  |
| --- | --- | --- |
| 方法类型 | 方法名称 | 方法描述 |
| 普通方法 | public void setInfluencingBounds(Bounds region) | 设置光源的影响范围 |
| 普通方法 | public void setInfluencingBoundingLeaf(BoundingLeaf region) | 设置光源的影响范围 |

**方向性光源：DirectionalLight类**

|  |  |  |
| --- | --- | --- |
| 方法类型 | 方法名称 | 方法描述 |
| 构造方法 | public DirectionalLight(boolean lightOn, Color3f color, Vector3f direction)； | 实例化一个方向性光源，确认光源的颜色，方向，以及是否打开光源 |
| 构造方法 | public DirectionalLight(Color3f color, Vector3f direction)； | 实例化一个方向性光源，确认光源的颜色，方向，并打开光源 |

**环境光源：AmbientLight类**

|  |  |  |
| --- | --- | --- |
| 方法类型 | 方法名称 | 方法描述 |
| 构造方法 | public AmbientLight(boolean lightOn, Color3f color) | 实例化一个环境光源，确认光源颜色，以及是否打开光源 |
| 构造方法 | public AmbientLight(Color3f color) | 实例化一个环境光源，确认光源颜色，并打开光源 |

**点光源：PointLight**

|  |  |  |
| --- | --- | --- |
| 方法类型 | 方法名称 | 方法描述 |
| 构造方法 | public PointLight(boolean lightOn, Color3f color, Point3f position, Point3f attenuation) | 实例化一个点光源，确认光源颜色，光源位置，光源的衰弱系数，以及是否打开光源 |
| 构造方法 | public PointLight(Color3f color, Point3f position, Point3f attenuation) | 实例化一个点光源，确认光源颜色，光源位置，光源的衰弱系数，并打开光源 |

**聚光光源：SpotLight类**

|  |  |  |
| --- | --- | --- |
| 方法类型 | 方法名称 | 方法描述 |
| 构造方法 | public SpotLight(boolean lightOn, Color3f color, Point3f position, Point3f attenuation, Vector3f direction, float spreadAngle, float concentration) | 实例化一个聚光光源，确认光源的颜色，位置，光源的衰弱系数，光源的方向，光线能覆盖到的角度，聚光指数，以及是否打开光源。 |
| 构造方法 | public SpotLight(Color3f color, Point3f position, Point3f attenuation, Vector3f direction, float spreadAngle, float concentration) | 实例化一个聚光光源，确认光源的颜色，位置，光源的衰弱系数，光源的方向，光线能覆盖到的角度，聚光指数，并打开光源。 |

方向性光源和环境光源很简单，就不多做解释。点光源是一个能向四面八方发射光线的光源，但是光线强度会随着传播距离的增大而衰减，衰减函数是A(d) = 1/(a0+a1d+a2d2)。这个a0、a1、a2就是衰减系数，由一个Point3f对象的三个坐标值分别指定。聚光光源是一种类似手电筒的光源，朝着指定圆锥形区域发射光线，这个就比较复杂了，它除了随传播距离的衰减（随传播距离的衰减函数与点光源一样）外，还有一个偏离中心轴方向的衰减，这个衰减函数A(θ) = cosnθ，θ为偏离中心轴的角度，n为聚光指数（由构造方法中最后一个参数确定）。

当实例化光源对象后，必须为其指定影响范围，就是用Light类中的两个方法设置的。

### 6.2、材质属性

可视对象必须定义材质属性后才能接收光源的照射，所以这个材质属性中必然有影响光照效果的参数。

我们把参数分成四类：漫反射类、镜面反射类、环境类、自发光类。

·漫反射类定义的是当可视对象接收光照后发生漫反射的反射光线的情况；

·镜面反射类定义的是当可视对象接收光照发生镜面反射的反射光线的情况；

·环境类定义的是当光线经过无数次反射后的反射光线的情况；

·自发光类定义的是可视对象自己发出的光线（只是微弱的）的情况。

在Material类中有这些方法

|  |  |
| --- | --- |
| 方法名称 | 方法描述 |
| public Material(Color3f ambientColor, Color3f emissiveColor, Color3f diffuseColor, Color3f specularColor, float shininess) | 构造方法 |
| public void setDiffuseColor(Color3f color) | 设置漫反射的反射光的颜色 |
| public void setSpecularColor(Color3f color) | 设置镜面反射的反射光的颜色 |
| public void setShininess(float shininess) | 设置可视对象的镜面指数（范围0~128），该值越小表示可视对象表面越粗糙 |
| public void setEmissiveColor(Color3f color) | 设置可视对象的自发光颜色 |
| public void setAmbientColor(Color3f color) | 设置经过可视对象无数次反射后的反射光颜色 |

这五个方法的参数对应于构造方法的五个参数。有了材质对象，用一个外观对象引用这个材质，然后将外观设置给可视对象即可让可视对象加入光照模型。

### 6.3、大气衰减和景深效果（雾化）

现实生活中，有雾的时候，看远处的物体将会很模糊，这就是大气衰减造成的效果。计算机图形学中可以通过一个相对高效的方法，来模拟大气衰减效果，这就是景深效果处理。景深效果是将对象与背景颜色相混合，混合之后的颜色中背景颜色所占的比重是一个关于距离的递增函数。

Java3D中的Fog类用来支持景深效果处理，Fog的两个子类分别指定了两种不同类型的雾：LinearFog、ExponentialFog。

LinearFog：线性雾，意思是这个递增函数是一个线性函数：

f(z) = (back-z)/(back-front)

其中z是到观察点的距离，front和back表示雾的范围的常数。LinearFog的构造方法：

|- public LinearFog()；

|- public LinearFog(Color3f color)；

|- public LinearFog(Color3f color, double frontDistance, double backDistance)；

构造方法中可以指定雾的颜色，以及雾的作用范围。

ExponentialFog：指数雾，意思是这个递增函数是一个指数函数：

f(z) = e-density\*z

dennsity用于控制f的衰减速度，也就是雾的密度。所以指数雾是没有作用范围这个概念的。ExponentialFog的构造方法：

|- public ExponentialFog()；

|- public ExponentialFog(Color3f color)；

|- public ExponentialFog(Color3f color, float density)；

### 6.4、纹理映射机制

纹理映射机制使我们能够不费大力气的完成可视对象的许多细节的绘制。纹理映射就是利用外部的图片附着在几何体表面使几何体看起来更逼真的技术。

2D纹理映射是一种将3D图像映射到3D对象表面的方法。在纹理映射过程中，我们将纹理图像上的点称为纹元，将3D对象上的点称为像素。纹理图像在Java3D中有自己的坐标，以图像左下角为原点，向右为x轴，向上为y轴，而且各顶点坐标分别为左下角坐标(0,0)、左上角坐标(0,1)、右上角坐标(1,1)、右下角坐标(1,0)，也就是说纹理图像中最大的坐标值就是1。

**范例：**将世界地图分为四个全等的三角形作为纹理贴到四面体的四个面上

|  |
| --- |
| package fuhao.demo;  import java.applet.Applet;  import java.awt.BorderLayout;  import java.awt.Color;  import java.awt.Component;  import java.awt.GraphicsConfiguration;  import javax.media.j3d.Alpha;  import javax.media.j3d.Appearance;  import javax.media.j3d.BoundingSphere;  import javax.media.j3d.BranchGroup;  import javax.media.j3d.Canvas3D;  import javax.media.j3d.DirectionalLight;  import javax.media.j3d.Material;  import javax.media.j3d.RotationInterpolator;  import javax.media.j3d.Shape3D;  import javax.media.j3d.Texture;  import javax.media.j3d.TransformGroup;  import javax.vecmath.Color3f;  import javax.vecmath.Point3d;  import javax.vecmath.TexCoord2f;  import javax.vecmath.Vector3f;  import com.sun.j3d.utils.applet.MainFrame;  import com.sun.j3d.utils.geometry.GeometryInfo;  import com.sun.j3d.utils.geometry.NormalGenerator;  import com.sun.j3d.utils.geometry.Primitive;  import com.sun.j3d.utils.image.TextureLoader;  import com.sun.j3d.utils.universe.SimpleUniverse;  import fuhao.shape.Shape3;  public class DocDemo extends Applet {  public static void main(String[] args) {  new MainFrame(new DocDemo(), 700, 700);  }    public DocDemo() {  this.setLayout(new BorderLayout());    GraphicsConfiguration gc = SimpleUniverse.getPreferredConfiguration();  Canvas3D canvas = new Canvas3D(gc);  this.add(canvas, BorderLayout.CENTER);    BranchGroup bg = this.getContent();  SimpleUniverse su = new SimpleUniverse(canvas);  su.getViewingPlatform().setNominalViewingTransform();    su.addBranchGraph(bg);  }    private BranchGroup getContent() {  BranchGroup bg = new BranchGroup();  TransformGroup spin = new TransformGroup();  spin.setCapability(TransformGroup.ALLOW\_TRANSFORM\_WRITE);  bg.addChild(spin);    RotationInterpolator r = new RotationInterpolator(new Alpha(-1, 4000), spin);  r.setSchedulingBounds(new BoundingSphere());  spin.addChild(r);    Appearance ap = new Appearance();  ap.setMaterial(new Material());    Shape3D shape = this.createShape(ap);  spin.addChild(shape);    DirectionalLight dl = new DirectionalLight(new Color3f(Color.GREEN), new Vector3f(0, 0, -2));  BoundingSphere bound = new BoundingSphere(new Point3d(0,0,0), 10);  dl.setInfluencingBounds(bound);  bg.addChild(dl);    return bg;  }    private Shape3D createShape(Appearance ap) {  GeometryInfo triangle = new GeometryInfo(GeometryInfo.POLYGON\_ARRAY);  Shape3D shape = new Shape3D();    Point3d p[] = new Point3d[4];  p[0] = new Point3d(0, 1, 0);  p[1] = new Point3d(-Math.cos(Math.PI/6), -Math.sin(Math.PI/6), 0);  p[2] = new Point3d(Math.cos(Math.PI/6), -Math.sin(Math.PI/6), 0);  p[3] = new Point3d(0, 0, 1);  int index[] = {2,1,0, 0,1,3, 0,3,2, 1,2,3};  int strip[] = {3,3,3,3};  triangle.setCoordinates(p);  triangle.setCoordinateIndices(index);  triangle.setStripCounts(strip);    /\*\*  \* 第一个参数，指定纹理坐标集的个数  \* 第二个参数，指定纹理坐标的维度  \*/  triangle.setTextureCoordinateParams(2, 2);    /\*\*  \* 参数：指定几何对象所使用的纹理坐标集在所有纹理坐标集中的索引  \* eg：如果上面的方法中设置纹理坐标集的个数为2，而你想用第二个纹理坐标集，则下标设置为1  \*/  triangle.setTexCoordSetMap(new int[] {0});    TexCoord2f t1[] = new TexCoord2f[5];  t1[0] = new TexCoord2f(0, 0);  t1[1] = new TexCoord2f(0, 1);  t1[2] = new TexCoord2f(1, 1);  t1[3] = new TexCoord2f(1, 0);  t1[4] = new TexCoord2f(0.5f, 0.5f);  int textureIndex[] = {0,4,1, 0,3,4, 2,1,4, 2,4,3};  /\*\*  \* 第一个参数：纹理坐标集的索引值  \* 第二个参数：纹理坐标集  \*/  triangle.setTextureCoordinates(0, t1);  triangle.setTextureCoordinateIndices(0, textureIndex);    TexCoord2f[] t2 = new TexCoord2f[5];  t2[0] = new TexCoord2f(0, 0);  t2[1] = new TexCoord2f(0, 0);  t2[2] = new TexCoord2f(0, 0);  t2[3] = new TexCoord2f(0, 0);  t2[4] = new TexCoord2f(0, 0);    triangle.setTextureCoordinates(1, t2);  triangle.setTextureCoordinateIndices(1, textureIndex);    new NormalGenerator().generateNormals(triangle);    shape.setGeometry(triangle.getGeometryArray());  Texture texture = new TextureLoader("C:\\Users\\fuhao\\Desktop\\世界地图.jpg", new Component(){}).getTexture();  ap.setTexture(texture);    shape.setAppearance(ap);  return shape;  }  } |

利用GeometryInfo构造四面体，设置顶点坐标，带数组和索引数组都不是问题，我们都能完成，但是由特殊颜色标注的代码是用来设置纹理，以及纹理的映射方式。虽然有些代码我写了注释，但是你们看起来可能还是会不明白，所以这里我一句一句分析。

最前面的绿色的代码，第一句使用了setTextureCoordinateParams()方法，从方法名就可以看出，这是设置纹理映射时的参数的方法，设置什么参数呢，注释告诉你了，设置纹理的维度和纹理坐标集的个数。纹理的维度：包括2D纹理，3D纹理，四维纹理，但是后两个都不讲，所以我们用的是2D纹理，第一个参数设置为2。纹理坐标集的个数是什么？我们在映射纹理的时候，是将图片上的左边与几何体上点的坐标对应起来的，所以除了几何体需要设置顶点坐标，纹理（也就是图片）也需要设置坐标，我蓝色部分和红色部分的代码就是给纹理设置了坐标，明显蓝色部分和红色部分设置的坐标不一样，这就是不同的纹理坐标集。给一幅图象以不同的方式设置纹理坐标的时候，得到的就是不同的纹理坐标集。那为什么需要不同的纹理坐标集呢？很简单，我们在纹理映射的时候，如果根据用户的输入我们需要改变映射方式的时候，就可以使用其他的纹理坐标集了。紫色部分的代码是从图片加载纹理并将纹理交给外观对象的过程。从图片中加载纹理有多种方式，用TextureLoader类是最简单的一种，其他的方式就不多讲了，这种够用了。

前面我们讲到过纹理（纹元）的大小可能和几何体的表面（像素）大小不一样，有时候需要一个纹元对应多个像素，有时需要多个纹元对应一个像素（术语称为插值函数）。在2D纹理中有这样的方法来帮助我们完成这个功能：

|- texture.setMagFilter(Texture.BASE\_LEVEL\_LINEAR);（放大滤波器，单纹元对多像素）

|- texture.setMinFilter(Texture.BASE\_LEVEL\_POINT);（缩小滤波器，多纹元对单像素）

参数是Texture类中定义的常量，共三个（可能不只）BASE\_LEVEL、BASE\_LEVEL\_LINEAR、BASE\_LEVEL\_POINT，一般使用第二个，线性插值。

## 七、行为和交互

这章所讲的东西，我相信会是你们最想学的。没错，基本上就属于动画编程了。

前面我们讲到的技术都属于绘制静态图像的技术，可能你会说有的程序中我们不是设置了旋转了吗。没错，这个旋转正是我们这章所讲的“行为”。

计算机图形中由两种常见的动态变化：交互和动画。交互是根据用户的输入改变图形场景；动画是根据时间变化来改变图形场景。改变图形场景的常见形式有改变可视对象的几何属性，外观属性，几何变换等等。

Java3D用“行为”这个概念来实现场景中的动画、交互以及其他动态机制。一个行为（Behavior）对象是场景图中的叶节点，它定义了当行为被激活的时候所要执行的动作（可能是改变几何属性或外观属性等等动作），行为是由一类特殊的对象触发的，这类对象称为唤醒条件。也就是说当唤醒条件满足时，行为就触发激活了，然后执行行为所定义的动作。当唤醒条件与移动鼠标之类的用户动作相关时，则该行为定义为一种交互形式，如果行为是由与时间相关的唤醒条件触发，则该行为定义的就是一种动画形式。

这一章中我们主要介绍行为的第一种常见形式：交互。我们还会讲到拾取功能在交互方面的应用。动画作为另一类行为将在下一章讨论。

### 7.1、Behavior类的唤醒条件

在Java3D中行为的操作都由Behavior类和它的子类完成。Behavior类是一个抽象类，它是Leaf类的子类，是场景图中的叶子节点。

Behavior类作为抽象类，类中有两个抽象方法：

|- public abstract void initialize()；

|- public abstract void processStimulus(java.util.Enumeration criteria)；

当实例化一个行为的时候，会自动调用它的initialize()（下面简称init方法）方法，所以行为的初始唤醒条件一般在这个方法中设置；在init方法中设置的唤醒条件满足时，就会执行processStimulus()方法，这个方法中我们一般都会写上具体的操作，例如满足唤醒条件后就改变可视对象的外观，这个改变外观的操作就可以写在这个方法中，一般在执行完这些操作后，我们还可以再次为行为设置唤醒条件，这样就能使行为无限的执行。如果在processStimulus()方法中没有再次设置唤醒条件，那么行为在满足上一次设置的唤醒条件后也不会被唤醒，也就是说，唤醒条件是一次性用品，用过一次就失效了。

Behavior类有许多子类，这些子类都是根据特定的唤醒条件来编写的，所以与其学习这些子类，不如先学习有哪些唤醒条件。前面讲过，唤醒条件是一类特殊的对象，既然是对象，那必然存在这样的一个类或者说一些类。

WakeupCriterion类是单个唤醒条件类的超类，后面我们还会讲组合唤醒条件。WakeupCriterion的子类及构造方法：

·**WakeupOnElapsedTime类**：

public WakeupOnElapsedTime(long milliseconds)；

|- 经历给定的时间后唤醒行为（时间单位毫秒）

·**WakeupOnAWTEvent类**：

public WakeupOnAWTEvent(int AWTId)；

|- 在指定的AWT事件发生时，唤醒行为。这个指定的AWT事件由参数决定，这个int型的参数是用Event类中的常量来表示的，例如：Event.MOUSE\_DOWN表示该事件是一个鼠标点击事件，Event.MOUSE\_ENTER表示鼠标进入窗口事件，AWT事件除了鼠标事件外，还有键盘按下等等事件，查java.awt.Event类文档即可。

public WakeupOnAWTEvent(long eventMask)；

|- 如果所指定的AWT事件不止一个的时候，就用这个构造方法，这个方法的参数是由多个事件的掩码的逻辑或操作合并而成的。例如：我们需要在鼠标事件（包括enter、move、click等）或键盘事件满足的时候唤醒行为，那么构造方法的参数可以这么写“AWTEvent.MOUSE\_EVENT\_MASK | AWTEvent.KEY\_EVENT\_MASK”。这里将java.awt.Event换成了java.awt.AWTEvent。

**·WakeupOnTransformChange类**：

public WakeupOnTransformChange(TransformGroup node)；

|- 当指定的TransformGroup节点所引用的变换发生改变的时候，唤醒行为。

**·WakeOnCollisionEntry类**：

public WakeupOnCollisionEntry(Bounds armingBounds)；

|- 当有其它对象进入所指定的范围时，唤醒行为。

public WakeupOnCollisionEntry(Node armingNode)；

|- 当其它对象与所指定的节点发生碰撞时，唤醒行为。

**·WakeupOnCollisionMovement类**：

public WakeupOnCollisionMovement(Bounds armingBounds)；

|- 当其他对象进入指定范围并移动时，唤醒行为。

public WakeupOnCollisionMovement(Node armingNode)；

|- 当其它对象与所指定的节点发生碰撞并移动时，唤醒行为。

**·WakeupOnCollisionExit类**：

public WakeupOnCollisionExit(Bounds armingBounds)；

|- 当其它对象离开指定范围时，唤醒行为。

public WakeupOnCollisionExit(Node armingNode)；

|- 当其它对象离开指定对象的时候（离开指定对象所在的范围），唤醒行为。

**·WakeupOnBehaviorPost类**：

public WakeupOnBehaviorPost(Behavior behavior, int postId)；

|- 当指定的Behavior对象（也可是子类对象）发送指定的ID时，唤醒行为。在Behavior类中有一个postId(int id)方法，使得Behavior对象都可以发送一个带有postID的事件，而这个事件会被设置了这样的唤醒条件的行为所监视。例如：行为A设置了这样的一个唤醒条件a，而行为B发送了这样一个事件，那么唤醒条件a会捕捉到这个事件，并唤醒行为A。在这个构造方法中，如果第一个参数为空，那么所发送的postID事件可以来自于任何一个Behavior对象，当第二个参数为0的时候，任何ID都能满足这个唤醒条件。注意：这里的postId参数不是已经定义好了的常量

单个的唤醒条件基本上讲完了，那当我们需要多个唤醒条件联合使用的时候怎么办呢？有这样四个类，从四个逻辑角度定义了四种联合方式：WakeupAnd类、WakeupOr类、WakeupOrOfAnd类、WakeupAndOfOr类。

前两个很好理解，And这个类是在所有的唤醒条件都满足的情况才唤醒行为，Or这个类是在满足了所有的唤醒条件中的一个的时候就唤醒行为。那么后面两个呢？

我们从构造方法上去观察。

**WakeupAnd的构造方法：public WakeupAnd(WakeupCriterion[] conditions)；**

**WakeupOr的构造方法：public WakeupOr(WakeupCriterion[] conditions)；**

**WakeupOrOfAnd的构造方法：public WakeupOrOfAnds(WakeupAnd[] conditions)；**

**WakeupAndOfOr的构造方法：public WakeupAndOfOrs(WakeupOr[] conditions)；**

前两个构造方法只需要前面我们讲到的单个的唤醒条件所组成的数组即可。第三个构造方法需要的是WakeupAnd类型的对象数组，而这个类的名称又为WakeupOrOfAnd。如果我们把And理解为交，Or理解为并，那么这个WakeupOrOfAnd就可以理解为（A交B）并（C交D），这里A交B是一个WakeupAnd类对象，C交D也是一个WakeupAnd类对象，如果学过数理逻辑的同学就会知道这是一种主析取范式。同样的道理WakeupAndOfOr类构造方法接收WakeupOr类对象数组，内部实际上是一个主合取范式。

不管是单个唤醒条件还是组合唤醒条件，他们都是WakeupCondition的子类，而Behavior对象设置唤醒条件的参数类型就是WakeupCondition类：

protected void wakeupOn(WakeupCondition criteria)；

实例大家自己随便想一想，实践实践，体验一下行为类的用法，用处，好处，这是Java动画编程的必经之路，必须要熟悉这几个唤醒条件。而最好的方法就是多编程。

### 7.2、交互

交互实际上就是由用户的输入信息触发的动态行为，包括鼠标行为、键盘行为。这两个行为实际上是Behavior的子类，它们将鼠标唤醒条件和键盘唤醒条件进行了封装，所以我们可以直接使用。

**在这里我先说明一下：鼠标行为和键盘行为都是行为，都必须设置范围，都必须作为叶子节点加入场景图，不然无效**

#### 7.2.1、鼠标行为（MouseBehavior）

MouseBehavior类定义了与鼠标动作关联的行为，这些行为都是在一个TransformGroup对象上进行的操作，用户可以通过鼠标调整TransformGroup节点所引用的几何变换。对封装的鼠标行为，我们主要用MouseBehavior的三个子类：

·MouseRotate：鼠标左键按下时拖动鼠标，相关联的TransformGroup节点的几何变换会添加相应的旋转变换；

构造方法：

|- MouseRotate(TransformGroup tg)；

|- MouseRotate(Component c, TransformGroup tg)；

|- MouseRotate(Component c)；

·MouseTranslate：鼠标右键按下时拖动鼠标，相关联的TransformGroup节点的几何变换会添加相应的平移变换；

构造方法：

|- MouseTranslate(TransformGroup tg)；

|- MouseTranslate(Component c, TransformGroup tg)；

|- MouseTranslate(Component c)；

·MouseZoom：鼠标中键按下时拖动鼠标，相关联的TransformGroup节点的几何变换会添加相应的放缩变换。

构造方法：

|- MouseZoom(TransformGroup tg)；

|- MouseZoom(Component c, TransformGroup tg)；

|- MouseZoom(Component c)；

如果是第一个构造方法，那么这个鼠标行为所关联的鼠标事件是Canvas3D上的鼠标事件。如果参数列表中有Component参数，那么可以指定另一个AWT组件来监听鼠标事件，当另一个AWT（假如是一个窗口）发生鼠标事件后，可以用相应的规则去修改这个鼠标行为所关联的TransformGroup对象。

你们可以利用一个球体对象来对这个行为进行测试。

#### 7.2.2、键盘行为（KeyNavigatorBehavior）

键盘行为也是在相关联的TransformGroup对象上进行操作，由这样的构造方法：

|- KeyNavigatorBehavior(TransformGroup tg)；

|- KeyNavigatorBehavior(Component c, TransformGroup tg)；

键盘行为没有鼠标行为那样有许多子类，在键盘行为中，内建的控制键定义如下：

· ←/→：左/右旋转；

· ↑/↓：前/后平移；

· Alt+←/Alt+→：左/右平移；

· PgUp/PgDn：上/下转动；

· Alt+PgUp/Alt+PgDn：上/下平移；

· -：减少裁剪距离；

· +：增加裁剪距离；

· =：复位场景。

这个也可以通过一个球体来进行测试。

#### 7.2.3、视图平台行为（ViewPlatformBehavior）

OrbitBehavior类是ViewPlatformBehavior类的一个子类，它允许用户通过鼠标的操作来改变观察平台。与MouseBehavior类有三个支持不同变换的子类不同的是，OrbitBehavior是由这个类独自来支持三种变换：平移、旋转、放缩。

OrbitBehavior的构造方法：

|- OrbitBehavior(Canvas3D cv)；

将一个OrbitBehavior对象设置为当前观察平台的视图平台行为：

ViewPlatform类中有一个setViewPlatformBehavior(ViewPlatformBehavior vb)方法，可以将一个OrbitBehavior行为加入场景。

所以视图平台行为不用我们编程者写代码将其加入场景图，只需要将其交给ViewPlatform对象处理即可，但仍然需要我们为其设置范围。

还有一点需要注意：视图平台的移动与可视对象的移动是相反的，千万不要以为出错了。

#### 7.2.4、行为和拾取

PickRotateBehavior、PickTranslateBehavior、PickZoomBehavior是PickMouseBehavior的子类将鼠标控制的旋转、平移、放缩等行为与拾取结合起来，将这些变换仅仅应用到被选中的物体。创建PickMouseBehavior类对象涉及到Canvas3D对象和BranchGroup对象，在Canvas3D画板上的BranchGroup分支上进行拾取。

PickRotateBehavior类构造方法：

|- PickRotateBehavior(Canvas3D cv, BranchGroup bg, Bounds bound)；

另外两个类是类似的。

同样的这三个类都是Behavior的子类，需要作为叶子节点添加到视图分支上才有效，构造方法中最后一个参数是用来设置行为的作用范围。

可以利用拾取操作时的四个可视对象来练习编写这三个类的例子。

## 八、动画

动画也是程序中的一种动态效果，这种动态效果是与时间相关的，并且是与帧相关的，帧反应了变化过程的中间结果，它是动画的主体。前一章讲了Behavior类能够实现交互功能，这一章将会将到Behavior类实现动画的能力。

Behavior类中的插值器（interpolator）封装了生成一般动画的基本操作。Alpha类实现了驱动插值器的时间函数，它能够通过插值器简单的生成动画，用户要做的只是创建带Alpha值得合适的插值器对象，并把它链接到目标对象上。

### 8.1、Alpha类

一个Alpha类对象实现一个时间函数（一般是周期性的），生成0.0~1.0之间的值。使用Alpha类驱动动画，比用实时时钟更方便。

Java3D中Alpha类是节点组件类的子类，用来修饰某些叶节点。

Alpha对象的功能由下面几个参数决定：

·LoopCount：周期的变化次数，-1表示无限循环；

·startTime：初始化参考点的绝对时间；

·triggerTime：从startTime到激活Alpha对象的时间，一般在微秒级；

·phaseDelayDuration：从triggerTime到初始化延时的时间，一般在微秒级；

·alphaAtZeroDuration：alpha对象值保持在0.0的时间，一般在微秒级；

·alphaAtOneDuration：alpha对象值保持在1.0的时间，一般在微秒级；

·increasingAlphaDuration：alpha对象值从0.0增长到1.0的时间，一般在微秒级；

·decreasingAlphaDuration：alpha对象值从1.0减少到0.0的时间，一般在微秒级；

·increasingAlphaRampDuration：alpha对象值在增长阶段累加的时间，一般在微秒级；

·decreasingAlphaRampDuration：alpha对象值在减少阶段累加的时间，一般在微秒级。

Alpha的构造方法：

|-public Alpha(int loopCount,

int mode,

long triggerTime,

long phaseDelayDuration,

long increasingAlphaDuration,

long increasingAlphaRampDuration,

long alphaAtOneDuration,

long decreasingAlphaDuration,

long decreasingAlphaRampDuration,

long alphaAtZeroDuration)；

|- public Alpha(int loopCount,

long triggerTime,

long phaseDelayDuration,

long increasingAlphaDuration,

long increasingAlphaRampDuration,

long alphaAtOneDuration)；

|- public Alpha(int loopCount, long increasingAlphaDuration)；

在第一个构造方法中第二个参数是一个int型的参数，这个参数我们在设置的时候，用到的是Alpha类中的常量，有两个：

·public static final int DECREASING\_ENABLE

·public static final int INCREASING\_ENABLE

**范例**：我们将Alpha对象的各个时间点的值在窗口中绘制出来

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |

|  |
| --- |
|  |