

Android OpenGL ES 开发教程

Android OpenGL ES 开发教程(1)

导言

Android ApiDemos 到目前为止，介绍完了出 View 以外的所有例子，在介绍 Graphics 示例时跳过了和 OpenGL ES 相关的例子，OpenGL ES 3D 图形开发需要专门的开发教程，因此从今天开始一边继续 Android ApiDemos Views 例子的解析，同时开始 Android OpenGL ES 开发教程。

在学习 Android OpenGL ES 开发之前，你必须具备 Java 语言开发经验和一些 Android 开发的基本知识，但并不需要有图形开发的经验，本教程也会涉及到一些基本的线性几何知识，如矢量，矩阵运算等。

对于 Android 开发的基本知识，可以参见 [Android 简明开发教程](#)，特别注意的是 [Android 简明开发教程二：安装开发环境](#)。本教程采用 Windows + Eclipse + Android SDK 作为开发的环境。

此外之前介绍的关于 Android OpenGL ES 开发的文章

[Android OpenGL ES 开发中的 Buffer 使用](#)

[Android OpenGL ES 简明开发教程](#)

也可以先看看，有助于学习 Android OpenGL ES 开发。

此外 Android SDK 中有关 OpenGL ES API 的开发文档

- android.opengl
- java.microedition.khronos.org
- java.microedition.khronos.org/opengles
- java.nio

注：上述 Android 文档基本为空，可以参见 [JSR239](#) 的文档，比较详细。

和 [OpenGL ES Specification](#) 都是学习时常用到的参考资料。

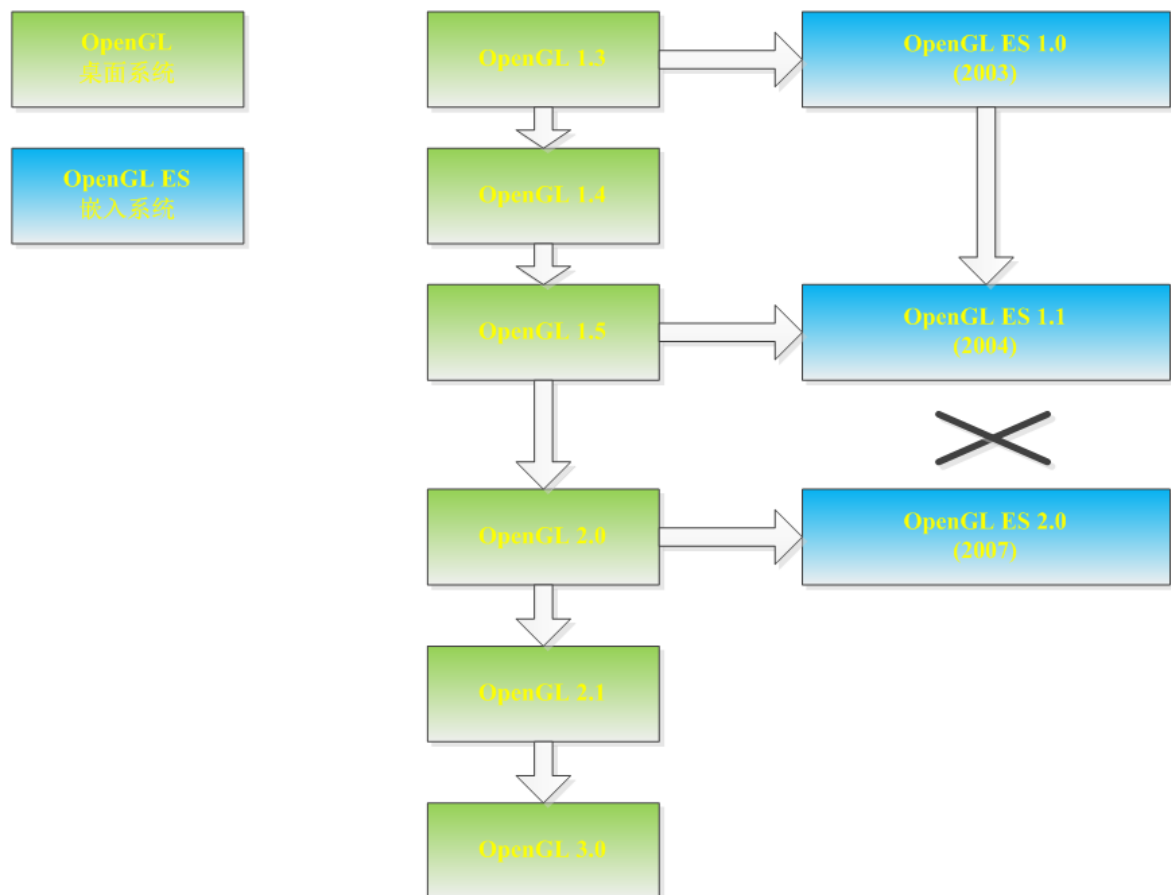
Android OpenGL ES 开发教程(2): 关于 OpenGL ES

什么是 OpenGL ES?

- OpenGL ES (为 OpenGL for Embedded System 的缩写) 为适用于嵌入式系统的一个免费二维和三维图形库。
- 为桌面版本 OpenGL 的一个子集。
- OpenGL ES 定义了一个在移动平台上能够支持 OpenGL 最基本功能的精简标准, 以适应如手机, PDA 或其它消费者移动终端的显示系统。
- Khronos Group 定义和管理了 OpenGL ES 标准。

OpenGL 与 OpenGL ES 的关系

OpenGL ES 是基于桌面版本 OpenGL 的, 下图显示了 OpenGL 和 OpenGL ES 之间的关系图



- OpenGL ES 1.0 基于 OpenGL 1.3 ， 在 2003 年发布
- OpenGL ES 1.1 基于 OpenGL 1.5 ， 在 2004 年发布
- OpenGL ES 2.0 基于 OpenGL 2.0, 在 2007 年发布
- OpenGL 2.0 向下兼容 OpenGL 1.5 而 OpenGL ES 2.0 和 OpenGL ES 1.x 不兼容，是两种完全不同的实现。

OpenGL ES Profiles

OpenGL ES 1.x 支持两种 Profile 以支持不同类型的嵌入设备。

1. The Common Profile 针对支持硬件浮点运算的设备，API 支持定点和浮点运算。
2. The Common Lite Profile 针对不支持硬件浮点运算的设备，API 只支持定点运算。

本教程主要针对 Common Profile 设备支持浮点运算。

Android OpenGL ES 开发教程(3)

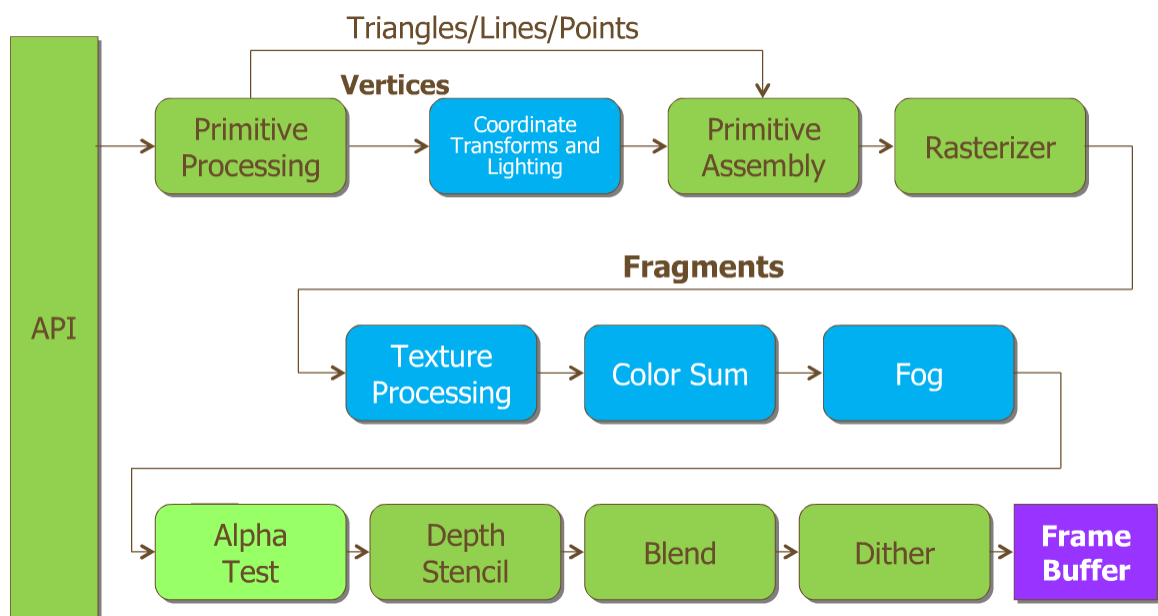
OpenGL ES 管道(Pipeline)

大部分图形系统都可以比作工厂中的装配线(Assemble line)或者称为管道(Pipeline)。前一道的输出作为下道工序的输入。主 CPU 发出一个绘图指令，然后可能由硬件部件完成坐标变换，裁剪，添加颜色或是材质，最后在屏幕上显示出来。

OpenGL ES 1.x 的工序是固定的，称为 **Fix-Function Pipeline**,可以想象一个带有很多控制开关的机器，尽管加工的工序是固定的，但是可以通过打开或关闭开关来设置参数或者打开关闭某些功能。

OpenGL ES 2.0 允许提供编程来控制一些重要的工序，一些“繁琐”的工序比如栅格化等仍然是固定的。

下图显示了 OpenGL ES 1.x 固定管道的结构图：



- 管道“工序”大致可以分为 Transformation Stage 和 Rasterization Stage 两大步。
- OpenGL ES 支持的基本图形为 点 Point, 线 Line, 和三角形 Triangle , 其它所有复制图形都是通过这几种基本几何图形组合而成。
- 在发出绘图指令后,会对顶点(Vertices)数组进行指定的坐标变换或光照处理。
- 顶点处理完成后,通过 Rasterizer 来生成像素信息,称为”Fragments“。
- 对于 Fragment 在经过 Texture Processing, Color Sum ,Fog 等处理并将最终处理结果存放在内存中(称为 FrameBuffer)。
- OpenGL 2.0 可以通过编程来修改蓝色的步骤,称为 Programmable Shader.

以上管道中工序可以通过设置来打开或关闭某些功能(比如无需雾化 Fog 处理),并可以为某个工序设置参数,比如设置 Vertex Array。

本教程主要介绍 OpenGL ES 1.1 编程,支持 OpenGL ES 2.0 的设备一般会同时支持 OpenGL ES 1.1。

Android OpenGL ES 开发教程(4)

OpenGL ES API 命名习惯

OpenGL ES 是个跨平台的 3D 图形开发包规范，最常见的实现是采用 C 语言实现的，Android OpenGL ES 实现上是使用 Java 语言对底层的 C 接口进行了封装，因此在 android.opengl

javax.microedition.khronos.egl ,javax.microedition.khronos.opengles 包中定义的 OpenGL 相关的类和方法带有很强的 C 语言色彩。

- 定义的常量都以 GL_为前缀。比如 GL10.GL_COLOR_BUFFER_BIT
- OpenGL ES 指令以 gl 开头，比如 gl.glClearColor
- 某些 OpenGL 指令以 3f 或 4f 结尾，3 和 4 代表参数的个数,f 代表参数类型为浮点数,如 gl.glColor4f，i,x 代表 int 如 gl.glColor4x
- 对应以 v 结尾的 OpenGL ES 指令，代表参数类型为一个矢量(Vector)，如 glTexEnvfv
- 所有 8-bit 整数对应到 byte 类型,16-bit 对应到 short 类型,32-bit 整数(包括 GLFixed)对应到 int 类型，而所有 32-bit 浮点数对应到 float 类型。
- GL_TRUE,GL_FALSE 对应到 boolean 类型
- C 字符串((char*)) 对应到 Java 的 UTF-8 字符串。

在前面 [Android OpenGL ES 开发中的 Buffer 使用](#) 说过 OpenGL ES 说过为了提高性能，通常将顶点，颜色等值存放在 java.nio 包中定义的 Buffer 类中。下表列出了 OpenGL ES 指令后缀，Java 类型，Java Buffer(java.nio)类型的对照表

Suffix	Java Types	Java Buffer Type
f	float	FloatBuffer
I	int	IntBuffer
x		

如下面代码

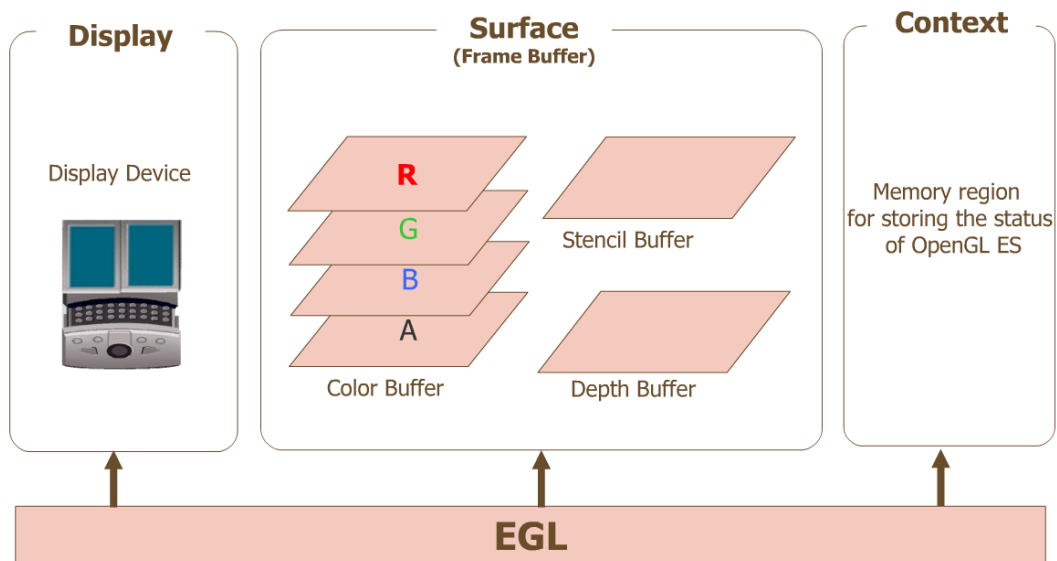
将为顶点指定 `color` 值，使用 `FloatBuffer` 来存放顶点的 `Color` 数组

```
1          // The colors mapped to the vertices.
2
3          float[] colors = {
4
5              1f, 0f, 0f, 1f, // vertex 0 red
6
7              0f, 1f, 0f, 1f, // vertex 1 green
8
9              0f, 0f, 1f, 1f, // vertex 2 blue
10
11             1f, 0f, 1f, 1f, // vertex 3 magenta
12
13             };
14
15             ...
16
17             // float has 4 bytes, colors (RGBA) * 4 bytes
18
19             ByteBuffer cbb
20
21             = ByteBuffer.allocateDirect(colors.length * 4);
22
23             cbb.order(ByteOrder.nativeOrder());
24
25             colorBuffer = cbb.asFloatBuffer();
26
27             colorBuffer.put(colors);
28
29             colorBuffer.position(0);
```

Android OpenGL ES 开发教程(5)

关于 EGL

OpenGL ES 的 `javax.microedition.khronos.opengles` 包定义了平台无关的 GL 绘图指令，`EGL(javax.microedition.khronos.egl)` 则定义了控制 `displays` ,`contexts` 以及 `surfaces` 的统一的平台接口。



- **Display**(`EGLDisplay`) 是对实际显示设备的抽象。
- **Surface** (`EGLSurface`) 是对用来存储图像的内存区域 `FrameBuffer` 的抽象，包括 `Color Buffer`, `Stencil Buffer` ,`Depth Buffer`.
- **Context** (`EGLContext`) 存储 `OpenGL ES` 绘图的一些状态信息。

使用 **EGL** 的绘图的一般步骤：

1. 获取 `EGLDisplay` 对象
2. 初始化与 `EGLDisplay` 之间的连接。
3. 获取 `EGLConfig` 对象
4. 创建 `EGLContext` 实例
5. 创建 `EGLSurface` 实例
6. 连接 `EGLContext` 和 `EGLSurface`.
7. 使用 `GL` 指令绘制图形
8. 断开并释放与 `EGLSurface` 关联的 `EGLContext` 对象
9. 删除 `EGLSurface` 对象

10. 删除 EGLContext 对象

11. 终止与 EGLDisplay 之间的连接。

一般来说在 Android 平台上开发 OpenGL ES 应用，无需直接使用 `javax.microedition.khronos.egl` 包中的类按照上述步骤来使用 OpenGL ES 绘制图形，在 Android 平台中提供了一个 `android.opengl` 包，类 `GLSurfaceView` 提供了对 `Display`, `Surface`, `Context` 的管理，大大简化了 OpenGL ES 的程序框架，对应大部分 OpenGL ES 开发，只需调用一个方法来设置 OpenGLView 用到的 `GLSurfaceView.Renderer`。可以参见

[Android OpenGL ES 简明开发教程二：构造 OpenGL ES View](#)

Android OpenGL ES 开发教程(6)

GLSurfaceView

Android OpenGL ES 相关的包主要定义在

- javax.microedition.khronos.opengles GL 绘图指令
- javax.microedition.khronos.egl EGL 管理 Display, surface 等
- android.opengl Android GL 辅助类, 连接 OpenGL 与 Android View, Activity
- javax.nio Buffer 类

其中 GLSurfaceView 为 android.opengl 包中核心类:

- 起到连接 OpenGL ES 与 Android 的 View 层次结构之间的桥梁作用。
- 使得 Open GL ES 库适应于 Android 系统的 Activity 生命周期。
- 使得选择合适的 Frame buffer 像素格式变得容易。
- 创建和管理单独绘图线程以达到平滑动画效果。
- 提供了方便使用的调试工具来跟踪 OpenGL ES 函数调用以帮助检查错误。

使用过 Java ME ,JSR 239 开发过 OpenGL ES 可以看到 Android 包 javax.microedition.khronos.egl ,javax.microedition.khronos.opengles 和 JSR239 基本一致, 因此理论上不使用 android.opengl 包中的类也可以开发 Android 上 OpenGL ES 应用, 但此时就需要自己使用 EGL 来管理 Display,Context, Surfaces 的创建, 释放, 捆绑, 可以参见 [Android OpenGL ES 开发教程\(5\): 关于 EGL](#) 。

使用 EGL 实现 GLSurfaceView 一个可能的实现如下:

```
1      class GLSurfaceView extends SurfaceView
2
3      implements SurfaceHolder.Callback, Runnable {
4
5      public GLSurfaceView(Context context) {
6
7          super(context);
8
9          mHolder = getHolder();
10
11         mHolder.addCallback(this);
```

```
7      mHolder.setType(SurfaceHolder.SURFACE_TYPE_GPU);
8  }
9      public void setRenderer(Renderer renderer) {
10         mRenderer = renderer;
11     }
12     public void surfaceCreated(SurfaceHolder holder) {
13     }
14     public void surfaceDestroyed(SurfaceHolder holder) {
15         running = false;
16         try {
17             thread.join();
18         } catch (InterruptedException e) {
19         }
20         thread = null;
21     }
22     public void surfaceChanged(SurfaceHolder holder,
23         int format, int w, int h) {
24         synchronized(this){
25             mWidth = w;
26             mHeight = h;
27             thread = new Thread(this);
```

```
28     thread.start();
29 }
30 }
31 public interface Renderer {
32     void EGLCreate(SurfaceHolder holder);
33     void EGLDestroy();
34     int Initialize(int width, int height);
35     void DrawScene(int width, int height);
36 }
37 public void run() {
38     synchronized(this) {
39         mRenderer.EGLCreate(mHolder);
40         mRenderer.Initialize(mWidth, mHeight);
41         running=true;
42         while (running) {
43             mRenderer.DrawScene(mWidth, mHeight);
44         }
45         mRenderer.EGLDestroy();
46     }
47 }
48 private SurfaceHolder mHolder;
```

```
49     private Thread thread;
50     private boolean running;
51     private Renderer mRenderer;
52     private int mWidth;
53     private int mHeight;
54 }
55 class GLRenderer implements GLSurfaceView.Renderer {
56     public GLRenderer() {
57     }
58     public int Initialize(int width, int height){
59         gl.glClearColor(1.0f, 0.0f, 0.0f, 0.0f);
60         return 1;
61     }
62     public void DrawScene(int width, int height){
63         gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
64         egl.eglSwapBuffers(eglDisplay, eglSurface);
65     }
66     public void EGLCreate(SurfaceHolder holder){
67         int[] num_config = new int[1];
68         EGLConfig[] configs = new EGLConfig[1];
69         int[] configSpec = {
```

```
70     EGL10.EGL_RED_SIZE, 8,
71     EGL10.EGL_GREEN_SIZE, 8,
72     EGL10.EGL_BLUE_SIZE, 8,
73     EGL10.EGL_SURFACE_TYPE,
74     EGL10.EGL_WINDOW_BIT,
75     EGL10.EGL_NONE
76 };
77
78 this.egl = (EGL10) EGLContext.getEGL();
79
80 eglDisplay =
81 this.egl.eglGetDisplay(EGL10.EGL_DEFAULT_DISPLAY);
82
83 this.egl.eglInitialize(eglDisplay, null);
84
85 this.egl.eglChooseConfig(eglDisplay, configSpec,
86 configs, 1, num_config);
87
88 eglConfig = configs[0];
89
90 eglContext = this.egl.eglCreateContext(eglDisplay, eglConfig,
91 EGL10.EGL_NO_CONTEXT, null);
92
93 eglSurface = this.egl.eglCreateWindowSurface(eglDisplay,
94 eglConfig, holder, null);
95
96 this.egl.eglMakeCurrent(eglDisplay, eglSurface,
97 eglSurface, eglContext);
98
99 gl = (GL10)eglContext.getGL();
100 }
```

```
91     public void EGLDestroy(){
92         if (eglSurface != null) {
93             egl.eglMakeCurrent(eglDisplay, EGL10.EGL_NO_SURFACE,
94             EGL10.EGL_NO_SURFACE, EGL10.EGL_NO_CONTEXT);
95             egl.eglDestroySurface(eglDisplay, eglSurface);
96             eglSurface = null;
97         }
98         if (eglContext != null) {
99             egl.eglDestroyContext(eglDisplay, eglContext);
100            eglContext = null;
101        }
102        if (eglDisplay != null) {
103            egl.eglTerminate(eglDisplay);
104            eglDisplay = null;
105        }
106    }
107    private EGL10 egl;
108    private GL10 gl;
109    private EGLDisplay eglDisplay;
110    private EGLConfig eglConfig;
111    private EGLContext eglContext;
```

```
112     private EGLSurface eglSurface;

113 }
```

可以看到需要派生 `SurfaceView` ,并手工创建,销毁 `Display,Context` ,工作繁琐。

使用 `GLSurfaceView` 内部提供了上面类似的实现,对于大部分应用只需调用一个方法来设置 `OpenGLView` 用到的 `GLSurfaceView.Renderer`.

```
1     public void setRenderer(GLSurfaceView.Renderer renderer)
```

`GLSurfaceView.Renderer` 定义了一个统一图形绘制的接口,它定义了如下三个接口函数:

```
1     // Called when the surface is created or recreated.

2     public void onSurfaceCreated(GL10 gl, EGLConfig config)

3     // Called to draw the current frame.

4     public void onDrawFrame(GL10 gl)

5     // Called when the surface changed size.

6     public void onSurfaceChanged(GL10 gl, int width, int height)
```

- `onSurfaceCreated`: 在这个方法中主要用来设置一些绘制时不常变化的参数,比如:背景色,是否打开 `z-buffer` 等。
- `onDrawFrame`: 定义实际的绘图操作。
- `onSurfaceChanged`: 如果设备支持屏幕横向和纵向切换,这个方法将发生在横向<->纵向互换时。此时可以重新设置绘制的纵横比率。

如果有需要,也可以通过函数来修改 `GLSurfaceView` 一些缺省设置:

- `setDebugFlags(int)` 设置 `Debug` 标志。
- `setEGLConfigChooser (boolean)` 选择一个 `Config` 接近 `16bitRGB` 颜色模式,可以打开或关闭深度(`Depth`)`Buffer` ,缺省为 `RGB_565` 并打开至少有 `16bit` 的 `depth Buffer`。
- `setEGLConfigChooser(EGLConfigChooser)` 选择自定义 `EGLConfigChooser`。

- `setEGLConfigChooser(int, int, int, int, int, int)` 指定 red ,green, blue, alpha, depth ,stencil 支持的位数，缺省为 RGB_565 ,16 bit depth buffer.

GLSurfaceView 缺省创建为 RGB_565 颜色格式的 Surface ,如果需要支持透明度，可以调用 `getHolder().setFormat(PixelFormat.TRANSLUCENT)`.

GLSurfaceView 的渲染模式有两种，一种是连续不断的更新屏幕，另一种为 on-demand ，只有在调用 `requestRender()` 在更新屏幕。 缺省为 `RENDERMODE_CONTINUOUSLY` 持续刷新屏幕。

Android OpenGL ES 开发教程(7)

创建实例应用 **OpenGLDemos** 程序框架

有了前面关于 Android OpenGL ES 的介绍，可以开始创建示例程序 OpenGLDemos。

使用 Eclipse 创建一个 Android 项目

- Project Name: OpenGLDemos
- Build Target: Android 1.6 (>1.5 即可)
- Application Name: Android OpenGL ES Demos
- Package Name: com.pstreets.opengl.demo
- Create Activity: AndroidOpenGLDemo

采用 Android ApiDemos 类似的方法, AndroidOpenGLDemo 为一 ListActivity , 可以使用 PackageManager 读取所有 Category 为 guidebee.intent.category.opengl.SAMPLE_CODE 的 Activity。

[Android ApiDemos 示例解析\(2\): SimpleAdapter, ListActivity, PackageManager](#)

创建一个 OpenGLRenderer 实现 GLSurfaceView.Renderer 接口:

```
1      public class OpenGLRenderer implements Renderer {  
2          private final IOpenGLDemo openGLDemo;  
3          public OpenGLRenderer(IGLOpenGLDemo demo){  
4              openGLDemo=demo;  
5          }  
6          public void onSurfaceCreated(GL10 gl, EGLConfig config) {  
7              // Set the background color to black ( rgba ).
```

```
8      gl.glClearColor(0.0f, 0.0f, 0.0f, 0.5f);

9      // Enable Smooth Shading, default not really needed.

10     gl.glShadeModel(GL10.GL_SMOOTH);

11     // Depth buffer setup.

12     gl.glClearDepthf(1.0f);

13     // Enables depth testing.

14     gl.glEnable(GL10.GL_DEPTH_TEST);

15     // The type of depth testing to do.

16     gl.glDepthFunc(GL10.GL_LEQUAL);

17     // Really nice perspective calculations.

18     gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,

19             GL10.GL_NICEST);

20 }

21 public void onDrawFrame(GL10 gl) {

22     if(openGLDemo!=null){

23         openGLDemo.DrawScene(gl);

24     }

25 }

26 public void onSurfaceChanged(GL10 gl, int width, int height) {
```

```
27      // Sets the current view port to the new size.

28      gl.glViewport(o, o, width, height);

29      // Select the projection matrix

30      gl.glMatrixMode(GL10.GL_PROJECTION);

31      // Reset the projection matrix

32      gl.glLoadIdentity();

33      // Calculate the aspect ratio of the window

34      GLU.gluPerspective(gl, 45.of,

35      (float) width / (float) height,

36      0.1f, 100.of);

37      // Select the modelview matrix

38      gl.glMatrixMode(GL10.GL_MODELVIEW);

39      // Reset the modelview matrix

40      gl.glLoadIdentity();

41      }

42      }

43

44

45
```

为简洁起见，为所有的示例定义了一个接口 `IOpenGLDemo`，

```
1      public interface IOpenGLDemo {
2          public void DrawScene(GL10 gl);
3      }
4
```

`DrawScene` 用于实际的 GL 绘图示例代码，其它的初始化工作基本就由 `GLSurfaceView` 和 `OpenGLRenderer` 完成,其中 `onSurfaceCreated` 和 `onSurfaceChanged` 中的代码含义现在无需了解，后面会有具体介绍，只要知道它们是用来初始化 `GLSurfaceView` 就可以了。

最后使用一个简单的例子“Hello World”结束本篇，“Hello World”使用红色背景刷新屏幕。

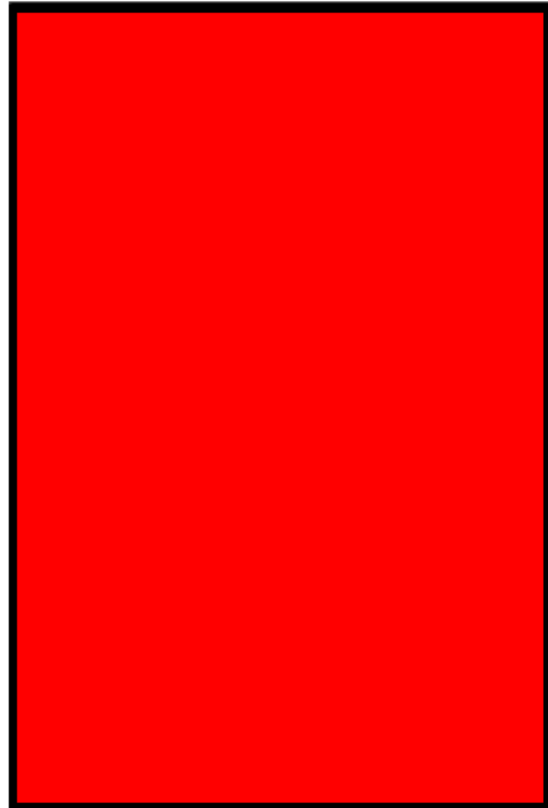
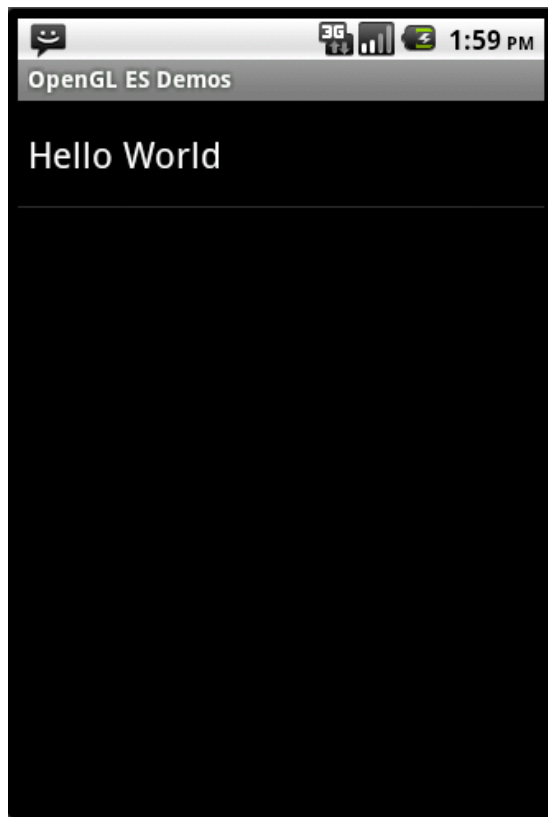
```
1      public class HelloWorld extends Activity
2          implements IOpenGLDemo{
3          /** Called when the activity is first created. */
4          @Override
5          public void onCreate(Bundle savedInstanceState) {
6              super.onCreate(savedInstanceState);
7              this.requestWindowFeature(Window.FEATURE_NO_TITLE)
8              ;
9              getWindow()
10             .setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
11             WindowManager.LayoutParams.FLAG_FULLSCREEN);
12             mGLSurfaceView = new GLSurfaceView(this);
13
```

```
14      mGLSurfaceView.setRenderer(new OpenGLRenderer(this));
15      setContentView(mGLSurfaceView);
16  }
17
18  public void DrawScene(GL10 gl) {
19      gl.glClearColor(1.0f, 0.0f, 0.0f, 0.0f);
20      // Clears the screen and depth buffer.
21      gl.glClear(GL10.GL_COLOR_BUFFER_BIT
22      | GL10.GL_DEPTH_BUFFER_BIT);
23  }
24
25  @Override
26  protected void onResume() {
27      // Ideally a game should implement
28      // onResume() and onPause()
29      // to take appropriate action when the
30      // activity loses focus
31      super.onResume();
32      mGLSurfaceView.onResume();
33  }
34
35  @Override
36  protected void onPause() {
37      // Ideally a game should implement onResume()
```

```
35    //and onPause()
36    // to take appropriate action when the
37    //activity loses focus
38    super.onPause();
39    mGLSurfaceView.onPause();
40    }
41    private GLSurfaceView mGLSurfaceView;
42    }
```

其对应应在 AndroidManifest.xml 中的定义如下：

```
<activity android:name=".HelloWorld"
android:label="@string/activity_helloworld">
<intent-filter>
< action android:name="android.intent.action.MAIN" />
<category
android:name="guidebee.intent.category.opengl.SAMPLE_CODE"
/>
< /intent-filter>
< /activity>
```



本例[下载](#)

Android OpenGL ES 开发教程(8)

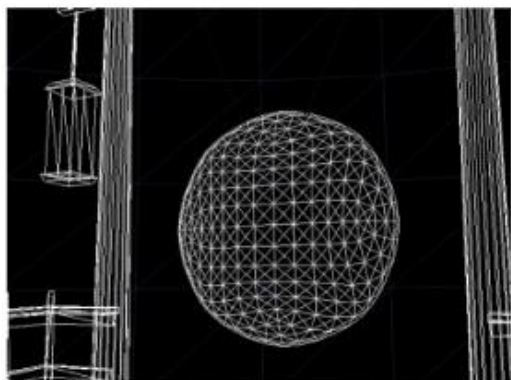
基本几何图形定义

在前面 [Android OpenGL ES 开发教程\(7\): 创建实例应用 OpenGL Demos 程序框架](#) 我们创建了示例程序的基本框架，并提供了一个“Hello World”示例，将屏幕显示为红色。

本例介绍 OpenGL ES 3D 图形库支持的几种基本几何图形，本篇部分内容与 [Android OpenGL ES 简明开发教程三: 3D 绘图基本概念](#) 类似。

通常二维图形库可以绘制点，线，多边形，圆弧，路径等等。OpenGL ES 支持绘制的基本几何图形分为三类：点，线段，三角形。也就是说 OpenGL ES 只能绘制这三种基本几何图形。任何复杂的 2D 或是 3D 图形都是通过这三种几何图形构造而成的。

比如下图复杂的 3D 图形，都有将其分割成细小的三角形面而构成的。然后通过上色(Color)，添加材质(Texture)，再添加光照 (lighting)，构造 3D 效果的图形：



点，线段，三角形都是通过顶点来定义的，也就是顶点数组来定义。对应平面上的一系列顶点，可以看出一个个孤立的点(Point)，也可以两个两个连接成线段(Line Segment)，也可以三个三个连成三角形(Triangle)。这些对一组顶点的不同解释就定义了 Android OpenGL ES 可以绘制的基本几何图形，下面定义了 OpenGL ES 定义的几种模式：

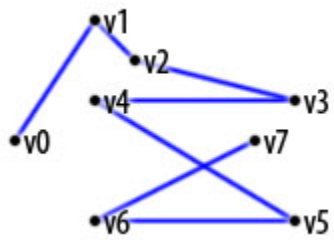
GL_POINTS

绘制独立的点。



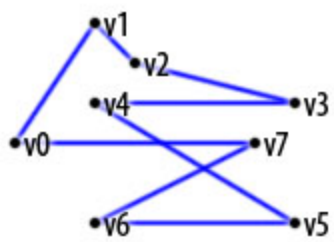
GL_LINE_STRIP

绘制一系列线段。



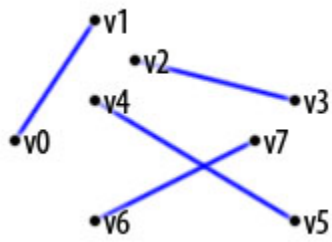
GL_LINE_LOOP

类同上，但是首尾相连，构成一个封闭曲线。



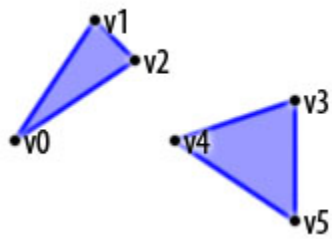
GL_LINES

顶点两两连接，为多条线段构成。



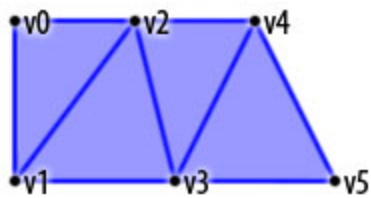
GL_TRIANGLES

每隔三个顶点构成一个三角形，为多个三角形组成。



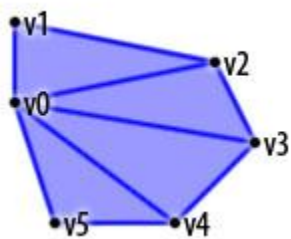
GL_TRIANGLE_STRIP

每相邻三个顶点组成一个三角形，为一系列相接三角形构成。



GL_TRIANGLE_FAN

以一个点为三角形公共顶点，组成一系列相邻的三角形。



以上模式对应到 Android 渲染方法：

OpenGL ES 提供了两类方法来绘制一个空间几何图形：

- `public abstract void glDrawArrays(int mode, int first, int count)` 使用 `VetexBuffer` 来绘制，顶点的顺序由 `vertexBuffer` 中的顺序指定。
- `public abstract void glDrawElements(int mode, int count, int type, Buffer indices)`，可以重新定义顶点的顺序，顶点的顺序由 `indices Buffer` 指定。

其中 `mode` 为上述解释顶点的模式。

前面说过顶点一般使用数组来定义，并使用 `Buffer` 来存储以提高绘图性能，参见 [Android OpenGL ES 开发中的 Buffer 使用](#)

如下面定义三个顶点坐标，并把它们存放在 `FloatBuffer` 中：

```
▪ float[] vertexArray = new float[]{  
1   ▪ -0.8f, -0.4f * 1.732f, 0.0f,  
2   ▪ 0.8f, -0.4f * 1.732f, 0.0f,  
3   ▪ 0.0f, 0.4f * 1.732f, 0.0f,  
4   ▪ };  
5   ▪ ByteBuffer vbb  
6   ▪ =  
7   ▪ ByteBuffer.allocateDirect(vertexArray.length*  
8   ▪ 4);  
9   ▪ vbb.order(ByteOrder.nativeOrder());  
10  ▪ FloatBuffer vertex = vbb.asFloatBuffer();  
11  ▪ vertex.put(vertexArray);  
12  ▪ vertex.position(0);
```

有了顶点的定义，下面就可以通过打开 [OpenGL ES 管道\(Pipeline\)](#)的相应开关将顶点参数传给 OpenGL 库：

打开顶点开关和关闭顶点开关的方法如下：

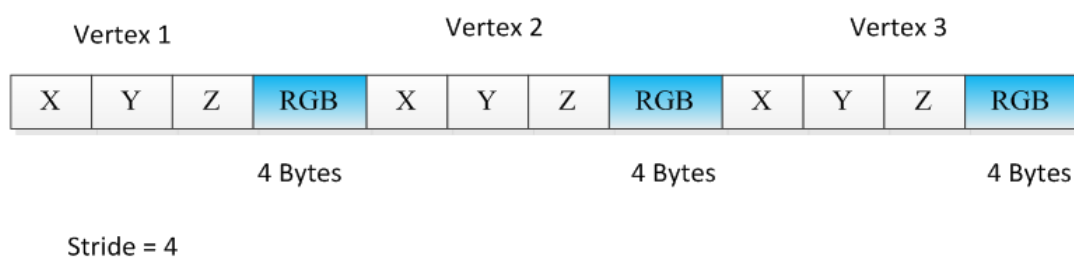
```
1  
  
2      gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);  
  
3      ...  
  
4      gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);  
  
5
```

在打开顶点开关后，将顶点坐标传给 OpenGL 管道的方法为：glVertexPointer：

```
public void glVertexPointer(int size,int type,int stride,Buffer pointer)
```

- **size**: 每个顶点坐标维数，可以为 2，3，4。
- **type**: 顶点的数据类型，可以为 GL_BYTE, GL_SHORT, GL_FIXED, 或 GL_FLOAT，缺省为浮点类型 GL_FLOAT。
- **stride**: 每个相邻顶点之间在数组中的间隔（字节数），缺省为 0，表示顶点存储之间无间隔。
- **pointer**: 存储顶点的数组。

应用用上可以般顶点的颜色值存放在对应顶点后面，如下图，RGB 采用 4 字节表示，此时相邻顶点就不是连续存放的，stride 值为 4



对应顶点除了可以为其定义坐标外，还可以指定颜色，材质，法线（用于光照处理）等。

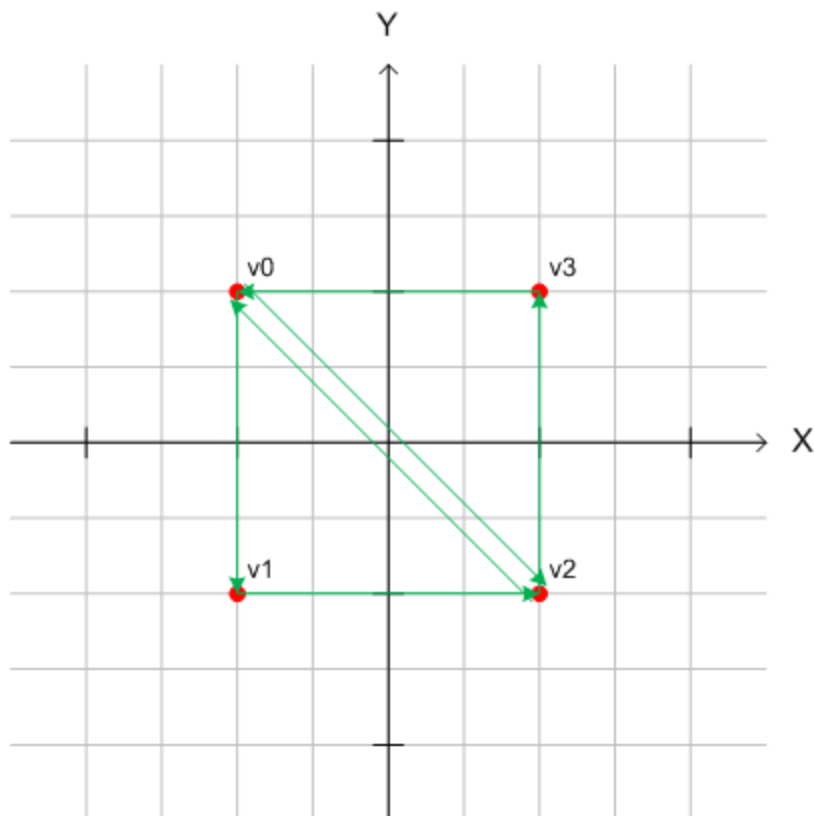
`glEnableClientState` 和 `glDisableClientState` 可以控制的 pipeline 开关可以有：
`GL_COLOR_ARRAY` (颜色) ,`GL_NORMAL_ARRAY` (法线),
`GL_TEXTURE_COORD_ARRAY` (材质), `GL_VERTEX_ARRAY`(顶点),
`GL_POINT_SIZE_ARRAY_OES` 等。

对应的传入颜色，顶点，材质，法线的方法如下：

```
glColorPointer(int size,int type,int stride,Buffer pointer)
glVertexPointer(int size, int type, int stride, Buffer pointer)
glTexCoordPointer(int size, int type, int stride, Buffer pointer)
glNormalPointer(int type,int stride, Buffer pointer)
```

如果需要使用三角形来构造复杂图形，可以使用 `GL_TRIANGLE_STRIP` 或 `GL_TRIANGLE_FAN` 模式，另外一种是通过定义顶点序列：

如下图定义了一个正方形：



对应的顶点和 `buffer` 定义代码：

```
1      private short[] indices = { 0, 1, 2, 0, 2, 3 };  
2      //To gain some performance we also put this ones in a byte  
3      buffer.  
4      // short is 2 bytes, therefore we multiply the number of vertices  
5      with 2.  
6      ByteBuffer ibb = ByteBuffer.allocateDirect(indices.length * 2);  
7      ibb.order(ByteOrder.nativeOrder());  
8      ShortBuffer indexBuffer = ibb.asShortBuffer();  
9      indexBuffer.put(indices);  
10     indexBuffer.position(0);
```

定义三角形的顶点的顺序很重要 在拼接曲面的时候，用来定义面的顶点的顺序非常重要，因为顶点的顺序定义了面的朝向（前向或是后向），为了获取绘制的高性能，一般情况不会绘制面的前面和后面，只绘制面的“前面”。虽然“前面”“后面”的定义可以应人而易，但一般为所有的“前面”定义统一的顶点顺序(顺时针或是逆时针方向)。

下面代码设置逆时针方法为面的“前面”：

```
1      gl.glFrontFace(GL10.GL_CCW);
```

打开 忽略“后面”设置：

```
1      gl.glEnable(GL10.GL_CULL_FACE);
```

明确指明“忽略”哪个面的代码如下：

```
1      gl.glCullFace(GL10.GL_BACK);
```

Android OpenGL ES 开发教程(9)

绘制点 Point

上一篇介绍了 OpenGL ES 能够绘制的几种基本几何图形：点，线，三角形。将分别介绍这几种基本几何图形的例子。为方便起见，暂时在同一平面上绘制这些几何图形，在后面介绍完 OpenGL ES 的坐标系统和坐标变换后，再介绍真正的 3D 图形绘制方法。

在 [Android OpenGL ES 开发教程\(7\): 创建实例应用 OpenGLDemos 程序框架](#) 创建了示例应用的程序框架，并提供了一个“Hello World”示例。

为避免一些重复代码，这里定义一个所有示例代码的基类 `OpenGLESActivity`，其定义如下：

```
1      public class OpenGLESActivity extends Activity
2
3          implements IOpenGLDemo{
4
5              /** Called when the activity is first created. */
6
7              @Override
8
9              public void onCreate(Bundle savedInstanceState) {
10
11                  super.onCreate(savedInstanceState);
12
13                  this.requestWindowFeature(Window.FEATURE_NO_TITLE);
14
15                  getWindow().setFlags(
16
17                      WindowManager.LayoutParams.FLAG_FULLSCREEN,
18
19                      WindowManager.LayoutParams.FLAG_FULLSCREEN);
20
21                  mGLSurfaceView = new GLSurfaceView(this);
```



```
12      mGLSurfaceView.setRenderer(new OpenGLRenderer(this));

13      setContentView(mGLSurfaceView);

14  }

15  public void DrawScene(GL10 gl) {

16      gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);

17      // Clears the screen and depth buffer.

18      gl.glClear(GL10.GL_COLOR_BUFFER_BIT

19      | GL10.GL_DEPTH_BUFFER_BIT);

20  }

21  @Override

22  protected void onResume() {

23      // Ideally a game should implement onResume() and
24      onPause()

25      // to take appropriate action when the activity loses focus

26      super.onResume();

27      mGLSurfaceView.onResume();

28  }

29  @Override

30  protected void onPause() {

    // Ideally a game should implement onResume() and
```

```

31     onPause()

32     // to take appropriate action when the activity loses focus

33     super.onPause();

34     mGLSurfaceView.onPause();

35 }

36     protected GLSurfaceView mGLSurfaceView;

37 }

```

- 在 onCreate 方法中创建一个 GLSurfaceView mGLSurfaceView,并将屏幕设置为全屏。并为 mGLSurfaceView 设置 Render.
- onResume ,onPause 处理 GLSurfaceView 的暂停和恢复。
- DrawScene 使用黑色清空屏幕。

OpenGL ES 内部存放图形数据的 Buffer 有 COLOR ,DEPTH (深度信息) 等,在绘制图形之前一般需要清空 COLOR 和 DEPTH Buffer。

本例在屏幕上使用红色绘制 3 个点。创建一个 DrawPoint 作为 OpenGL ESActivity 的子类,并定义 3 个顶点的坐标:

```

1     public class DrawPoint extends OpenGL ESActivity

2     implements IOpenGLDemo{

3     float[] vertexArray = new float[]{

4     -0.8f , -0.4f * 1.732f , 0.0f ,

5     0.8f , -0.4f * 1.732f , 0.0f ,

6     0.0f , 0.4f * 1.732f , 0.0f ,

```

```
7      };

8      /** Called when the activity is first created. */

9      @Override

10     public void onCreate(Bundle savedInstanceState) {

11         super.onCreate(savedInstanceState);

12     }

13     ...

14 }
```

下面为 DrawPoint 的 DrawScene 的实现：

```
1     public void DrawScene(GL10 gl) {

2         super.DrawScene(gl);

3         ByteBuffer vbb

4         = ByteBuffer.allocateDirect(vertexArray.length*4);

5         vbb.order(ByteOrder.nativeOrder());

6         FloatBuffer vertex = vbb.asFloatBuffer();

7         vertex.put(vertexArray);

8         vertex.position(0);
```

```
9      gl.glColor4f(1.0f, 0.0f, 0.0f, 1.0f);

10     gl.glPointSize(8f);

11     gl.glLoadIdentity();

12     gl.glTranslatef(0, 0, -4);

13     gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);

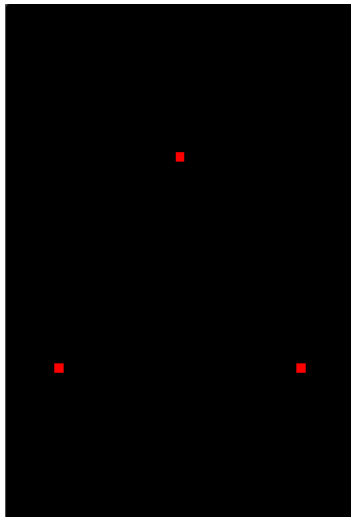
14     gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertex);

15     gl.glDrawArrays(GL10.GL_POINTS, 0, 3);

16     gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);

17 }
```

- 首先是使用 `FloatBuffer` 存放三个顶点坐标。
- 使用 `glColor4f(float red, float green, float blue, float alpha)` 将当前颜色设为红色。
- `glPointSize(float size)` 可以用来设置绘制点的大小。
- 使用 `glEnableClientState` 打开 Pipeline 的 Vertex 顶点“开关”
- 使用 `glVertexPointer` 通知 OpenGL ES 图形库顶点坐标。
- 使用 `GL_POINTS` 模式使用 `glDrawArrays` 绘制 3 个顶点。



本例[下载](#)

Android OpenGL ES 开发教程(10)

绘制线段 Line Segment

创建一个 DrawLine Activity，定义四个顶点：

```
▪ 1      ▪ float vertexArray[] = {  
▪ 2      ▪ -0.8f, -0.4f * 1.732f, 0.0f,  
▪ 3      ▪ -0.4f, 0.4f * 1.732f, 0.0f,  
▪ 4      ▪ 0.0f, -0.4f * 1.732f, 0.0f,  
▪ 5      ▪ 0.4f, 0.4f * 1.732f, 0.0f,  
▪ 6      ▪ };
```

分别以三种模式 GL_LINES, GL_LINE_STRIP, GL_LINE_LOOP 来绘制直线：

```
▪ 1      ▪ public void DrawScene(GL10 gl) {  
▪ 2      ▪ super.DrawScene(gl);  
▪ 3      ▪ ByteBuffer vbb  
▪ 4      ▪ =  
▪ 5      ▪ ByteBuffer.allocateDirect(vertexArray.length * 4  
▪ 6      ▪ );  
▪ 7      ▪ vbb.order(ByteOrder.nativeOrder());  
▪ 8      ▪ FloatBuffer vertex = vbb.asFloatBuffer();  
▪ 9      ▪ vertex.put(vertexArray);  
▪ 10     ▪ vertex.position(0);  
▪ 11     ▪ gl.glLoadIdentity();  
▪ 12     ▪ gl.glTranslatef(0, 0, -4);  
▪ 13     ▪ gl.glEnableClientState(GL10.GL_VERTEX_AR  
▪ 14     ▪ RAY);  
▪ 15     ▪ gl.glVertexPointer(3, GL10.GL_FLOAT, 0,  
▪ 16     ▪ vertex);  
▪ 17     ▪ index++;  
▪ 18     ▪ index%=10;  
▪ 19     ▪ switch(index){  
▪ 20     ▪ case 0:  
▪ 21     ▪ case 1:  
▪ 22     ▪ case 2:
```

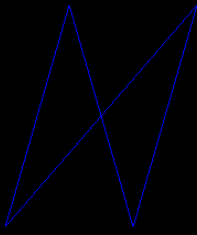
```

▪ 23      ▪ gl.glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
▪ 24      ▪ gl.glDrawArrays(GL10.GL_LINES, 0, 4);
▪ 25      ▪ break;
▪ 26      ▪ case 3:
▪ 27      ▪ case 4:
▪ 28      ▪ case 5:
▪ 29      ▪ gl.glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
▪ 30      ▪ gl.glDrawArrays(GL10.GL_LINE_STRIP, 0,
▪ 31      4);
▪ 32      ▪ break;
▪ 33      ▪ case 6:
▪ 34      ▪ case 7:
▪ 35      ▪ case 8:
▪ 36      ▪ case 9:
▪ 37      ▪ gl.glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
▪ 38      ▪ gl.glDrawArrays(GL10.GL_LINE_LOOP, 0, 4);
▪ 39      ▪ break;
▪ 40      ▪ }
▪ 41      ▪ gl.glDisableClientState(GL10.GL_VERTEX_A
▪ 42      RRAY);
▪ 43      ▪ }

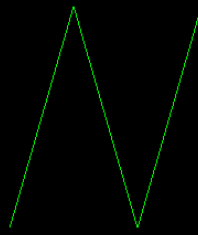
```

这里 `index` 的目的是为了延迟一下显示（更好的做法是使用固定时间间隔）。前面说过 `GLSurfaceView` 的渲染模式有两种，一种是连续不断的更新屏幕，另一种为 `on-demand`，只有在调用 `requestRender()` 在更新屏幕。缺省为 `RENDERMODE_CONTINUOUSLY` 持续刷新屏幕。

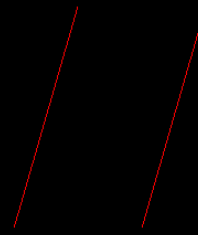
`OpenGLDemos` 使用的是缺省的 `RENDERMODE_CONTINUOUSLY` 持续刷新屏幕，因此 `Activity` 的 `drawScene` 会不断的执行。本例中屏幕上顺序以红，绿，蓝色显示 `LINES`, `LINE_STRIP`, `LINE_LOOP`。



LINE LOOP



LINE STRIP



Lines

Android OpenGL ES 开发教程(11)

绘制三角形 Triangle

三角形为 OpenGL ES 支持的面，同样创建一个 DrawTriangle Activity，定义 6 个顶点使用三种不同模式来绘制三角形：

```
1      float vertexArray[] = {
2          -0.8f, -0.4f * 1.732f, 0.0f,
3          0.0f, -0.4f * 1.732f, 0.0f,
4          -0.4f, 0.4f * 1.732f, 0.0f,
5          0.0f, -0.0f * 1.732f, 0.0f,
6          0.8f, -0.0f * 1.732f, 0.0f,
7          0.4f, 0.4f * 1.732f, 0.0f,
8      };
9
```

本例绘制

```
1      public void DrawScene(GL10 gl) {
2          super.DrawScene(gl);
3          ByteBuffer vbb
4              = ByteBuffer.allocateDirect(vertexArray.length*4);
5          vbb.order(ByteOrder.nativeOrder());
6          FloatBuffer vertex = vbb.asFloatBuffer();
7          vertex.put(vertexArray);
8          vertex.position(0);
```



```
9      gl.glLoadIdentity();
10     gl.glTranslatef(0, 0, -4);
11     gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
12     gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertex);
13     index++;
14     index%=10;
15     switch(index){
16     case 0:
17     case 1:
18     case 2:
19         gl.glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
20         gl.glDrawArrays(GL10.GL_TRIANGLES, 0, 6);
21         break;
22     case 3:
23     case 4:
24     case 5:
25         gl.glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
26         gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, 0, 6);
27         break;
28     case 6:
29     case 7:
```

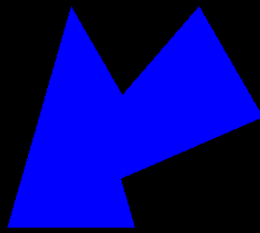
```
30     case 8:
31     case 9:
32         gl.glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
33         gl.glDrawArrays(GL10.GL_TRIANGLE_FAN, 0, 6);
34         break;
35     }
36     gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
37 }
38
39
40
41
42
43
```

这里 `index` 的目的是为了延迟一下显示（更好的做法是使用固定时间间隔）。前面说过 `GLSurfaceView` 的渲染模式有两种，一种是连续不断的更新屏幕，另一种为 `on-demand`，只有在调用 `requestRender()` 在更新屏幕。缺省为 `RENDERMODE_CONTINUOUSLY` 持续刷新屏幕。

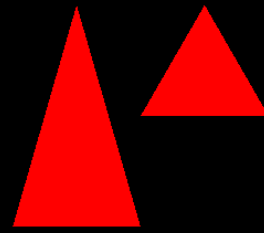
`OpenGLDemos` 使用的是缺省的 `RENDERMODE_CONTINUOUSLY` 持续刷新屏幕，因此 `Activity` 的 `drawScene` 会不断的执行。本例中屏幕上顺序以红，绿，蓝色显示 `TRIANGLES`, `TRIANGLE_STRIP`, `TRIANGLE_FAN`。



TRIANGLE STRIP



TRIANGLE FAN



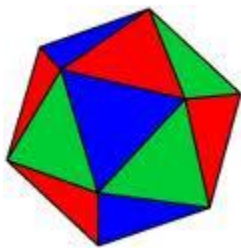
TRIANGLES

Android OpenGL ES 开发教程(12)

绘制一个 20 面体

前面介绍了 OpenGL ES 所有能够绘制的基本图形，点，线段和三角形。其它所有复杂的 2D 或 3D 图形都是由这些基本图形构成。

本例介绍如何使用三角形构造一个正 20 面体。一个正 20 面体，有 12 个顶点，20 个面，30 条边构成：



创建一个 DrawIcosahedron Activity，定义 20 面体的 12 个顶点，和 20 个面如下：

```
1      static final float X=.525731112119133606f;
2      static final float Z=.850650808352039932f;
3      static float vertices[] = new float[]{
4          -X, 0.of, Z, X, 0.of, Z, -X, 0.of, -Z, X, 0.of, -Z,
5          0.of, Z, X, 0.of, Z, -X, 0.of, -Z, X, 0.of, -Z, -X,
6          Z, X, 0.of, -Z, X, 0.of, Z, -X, 0.of, -Z, -X, 0.of
7      };
8      static short indices[] = new short[]{
9          0,4,1, 0,9,4, 9,5,4, 4,5,8, 4,8,1,
10         8,10,1, 8,3,10, 5,3,8, 5,2,3, 2,7,3,
11         7,10,3, 7,6,10, 7,11,6, 11,0,6, 0,1,6,
12         6,1,10, 9,0,11, 9,11,2, 9,2,5, 7,2,11 };
```

OpenGL ES 缺省使用同一种颜色来绘制图形，为了能够更好的显示 3D 效果，我们为每个顶点随机定义一些颜色如下：

```
1      float[] colors = {  
2          of, of, of, 1f,  
3          of, of, 1f, 1f,  
4          of, 1f, of, 1f,  
5          of, 1f, 1f, 1f,  
6          1f, of, of, 1f,  
7          1f, of, 1f, 1f,  
8          1f, 1f, of, 1f,  
9          1f, 1f, 1f, 1f,  
10         1f, of, of, 1f,  
11         of, 1f, of, 1f,  
12         of, of, 1f, 1f,  
13         1f, of, 1f, 1f  
14     };  
15
```

添加颜色可以参见 [Android OpenGL ES 简明开发教程五：添加颜色](#) 以后也会有详细说明。

将顶点，面，颜色存放到 **Buffer** 中以提高性能：

```
1      private FloatBuffer vertexBuffer;

2      private FloatBuffer colorBuffer;

3      private ShortBuffer indexBuffer;

4      ByteBuffer vbb

5      = ByteBuffer.allocateDirect(vertices.length * 4);

6      vbb.order(ByteOrder.nativeOrder());

7      vertexBuffer = vbb.asFloatBuffer();

8      vertexBuffer.put(vertices);

9      vertexBuffer.position(0);

10     ByteBuffer cbb

11     = ByteBuffer.allocateDirect(colors.length * 4);

12     cbb.order(ByteOrder.nativeOrder());

13     colorBuffer = cbb.asFloatBuffer();

14     colorBuffer.put(colors);

15     colorBuffer.position(0);

16     ByteBuffer ibb

17     = ByteBuffer.allocateDirect(indices.length * 2);

18     ibb.order(ByteOrder.nativeOrder());

19     indexBuffer = ibb.asShortBuffer();

20     indexBuffer.put(indices);

21     indexBuffer.position(0);
```

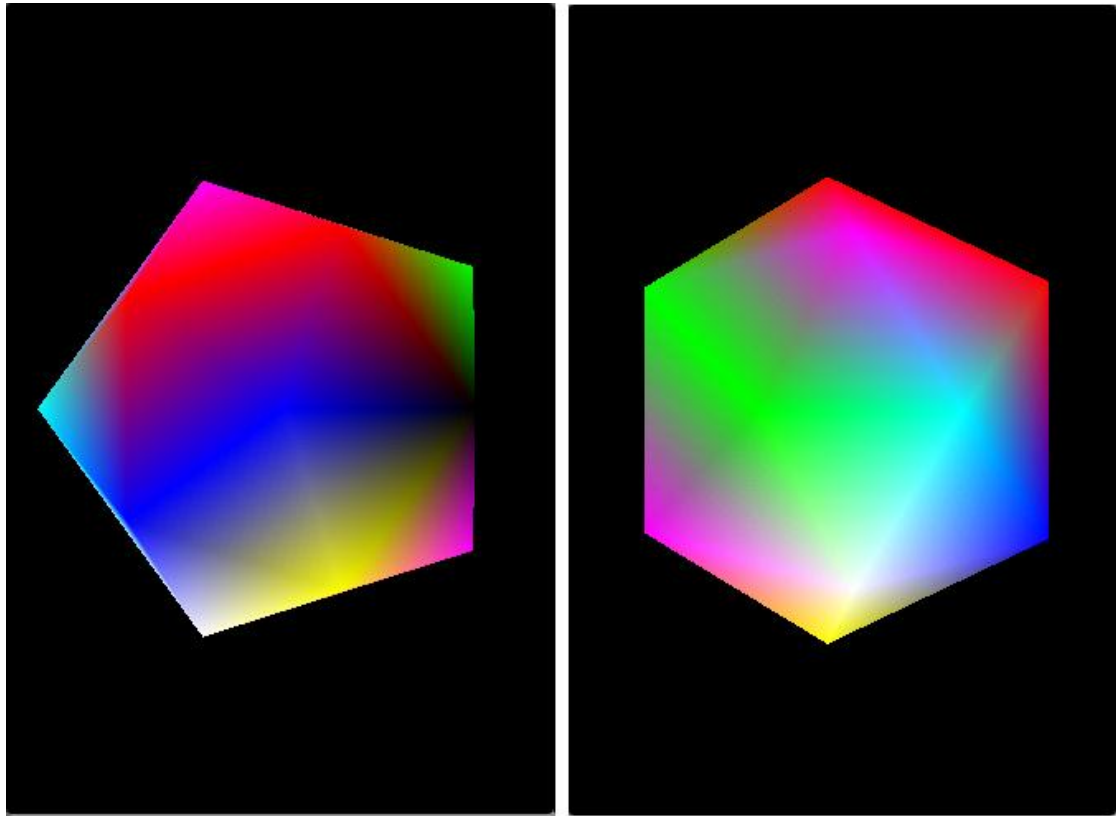
看一看 DrawScene 代码：

```
1
2     public void DrawScene(GL10 gl) {
3         super.DrawScene(gl);
4         gl.glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
5         gl.glLoadIdentity();
6         gl.glTranslatef(0, 0, -4);
7         gl.glRotatef(angle, 0, 1, 0);
8         gl.glFrontFace(GL10.GL_CCW);
9         gl.glEnable(GL10.GL_CULL_FACE);
10        gl.glCullFace(GL10.GL_BACK);
11        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
12        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, vertexBuffer);
13        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
14        gl.glColorPointer(4, GL10.GL_FLOAT, 0, colorBuffer);
15        gl.glDrawElements(GL10.GL_TRIANGLES, indices.length,
16        GL10.GL_UNSIGNED_SHORT, indexBuffer);
17        gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
18        gl.glDisable(GL10.GL_CULL_FACE);
19        angle++;
20    }
21
```

用来绘制 2D 面体使用了 `gl.glDrawElements`:

`public abstract void glDrawElements(int mode, int count, int type, Buffer indices)` , 可以重新定义顶点的顺序, 顶点的顺序由 `indices Buffer` 指定。

本例会显示一个绕 Y 轴不断旋转的 2D 面体, 如何旋转模型将在后面的坐标系统和坐标变换中介绍。



`DrawLine`, `DrawTriangle`,及本例[下载](#)

Android OpenGL ES 开发教程(13)

阶段小结

之前介绍了什么是 OpenGL ES，OpenGL ES 管道的概念，什么是 EGL，Android 中 OpenGL ES 的开发包以及 GLSurfaceView，OpenGL ES 所支持的基本几何图形：点，线，面，已及如何使用这些基本几何通过构成较复杂的图像（2D 面体）。

- [Android OpenGL ES 开发教程\(1\): 引言](#)
- [Android OpenGL ES 开发教程\(2\): 关于 OpenGL ES](#)
- [Android OpenGL ES 开发教程\(3\): OpenGL ES 管道\(Pipeline\)](#)
- [Android OpenGL ES 开发教程\(4\): OpenGL ES API 命名习惯](#)
- [Android OpenGL ES 开发教程\(5\): 关于 EGL](#)
- [Android OpenGL ES 开发教程\(6\): GLSurfaceView](#)
- [Android OpenGL ES 开发教程\(7\): 创建实例应用 OpenGLDemos 程序框架](#)
- [Android OpenGL ES 开发教程\(8\): 基本几何图形定义](#)
- [Android OpenGL ES 开发教程\(9\): 绘制点 Point](#)
- [Android OpenGL ES 开发教程\(10\): 绘制线段 Line Segment](#)
- [Android OpenGL ES 开发教程\(11\): 绘制三角形 Triangle](#)
- [Android OpenGL ES 开发教程\(12\): 绘制一个 2D 面体](#)

但到目前为止还只是绘制 2D 图形，而忽略了一些重要的概念，3D 坐标系，和 3D 坐标变换，在介绍 OpenGL ES Demo 程序框架时，创建一个 OpenGLRenderer 实现 GLSurfaceView.Renderer 接口

```
1      public void onSurfaceChanged(GL10 gl, int width, int height) {  
2          // Sets the current view port to the new size.  
3          gl.glViewport(0, 0, width, height);  
4          // Select the projection matrix  
5          gl.glMatrixMode(GL10.GL_PROJECTION);  
6          // Reset the projection matrix  
7          gl.glLoadIdentity();
```

```
8      // Calculate the aspect ratio of the window
9      GLU.gluPerspective(gl, 45.0f,
10     (float) width / (float) height,
11     0.1f, 100.0f);
12     // Select the modelview matrix
13     gl.glMatrixMode(GL10.GL_MODELVIEW);
14     // Reset the modelview matrix
15     gl.glLoadIdentity();
16 }
17 }
```

我们忽略了对 `glViewport`, `glMatrixMode`, `gluPerspective` 以及 20 面体时 `glLoadIdentity`, `glTranslatef`, `glRotatef` 等方法的介绍, 这些都涉及到如何来为 3D 图形构建模型, 在下面的几篇将详细介绍 OpenGL ES 的坐标系和坐标变换已经如何进行 3D 建模。

Android OpenGL ES 开发教程(14)

三维坐标系及坐标变换初步

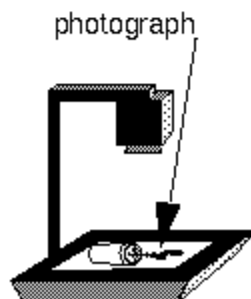
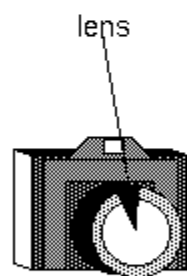
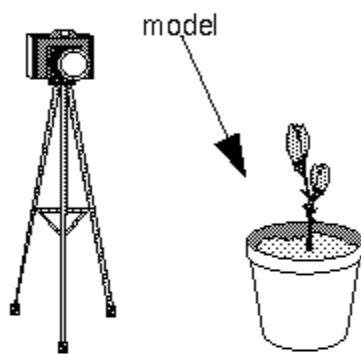
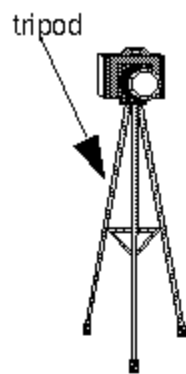
OpenGL ES 图形库最终的结果是在二维平面上显示 3D 物体(常称作模型 Model)这是因为目前的打部分显示器还只能显示二维图形。但我们在构造 3D 模型时必须要有空间现象能力，所有对模型的描述还是使用三维坐标。也就是使用 3D 建模，而有 OpenGL ES 库来完成从 3D 模型到二维屏幕上的显示。

这个过程可以分成三个部分：

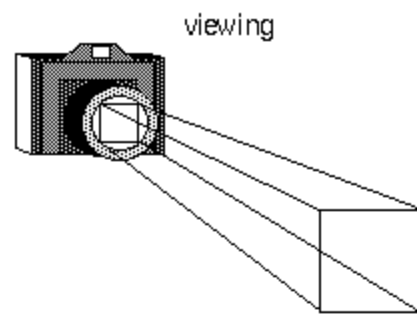
- 坐标变换，坐标变换通过使用变换矩阵来描述，因此学习 3D 绘图需要了解一些空间几何，矩阵运算的知识。三维坐标通常使用[齐次坐标](#)来定义。变换矩阵操作可以分为视角（Viewing），模型（Modeling）和投影（Projection）操作，这些操作可以有选择，平移，缩放，正侧投影，透视投影等。
- 由于最终的 3D 模型需要在一个矩形窗口中显示，因此在这个窗口之外的部分需要裁剪掉以提高绘图效率，对应 3D 图形，裁剪是将处在剪切面之外的部分扔掉。
- 在最终绘制到显示器（2D 屏幕），需要建立起变换后的坐标和屏幕像素之间的对应关系，这通常称为“视窗”坐标变换(Viewport) transformation.

如果我们使用照相机拍照的过程做类比，可以更好的理解 3D 坐标变换的过程。

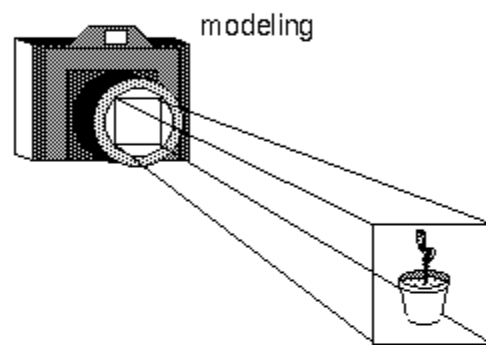
With a Camera



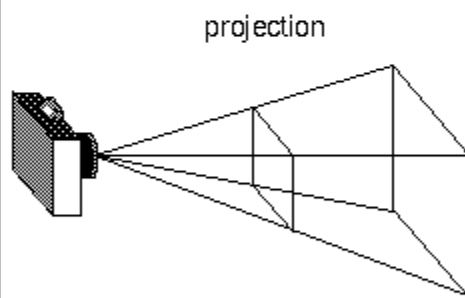
With a Computer



positioning the viewing volume
in the world



positioning the models
in the world



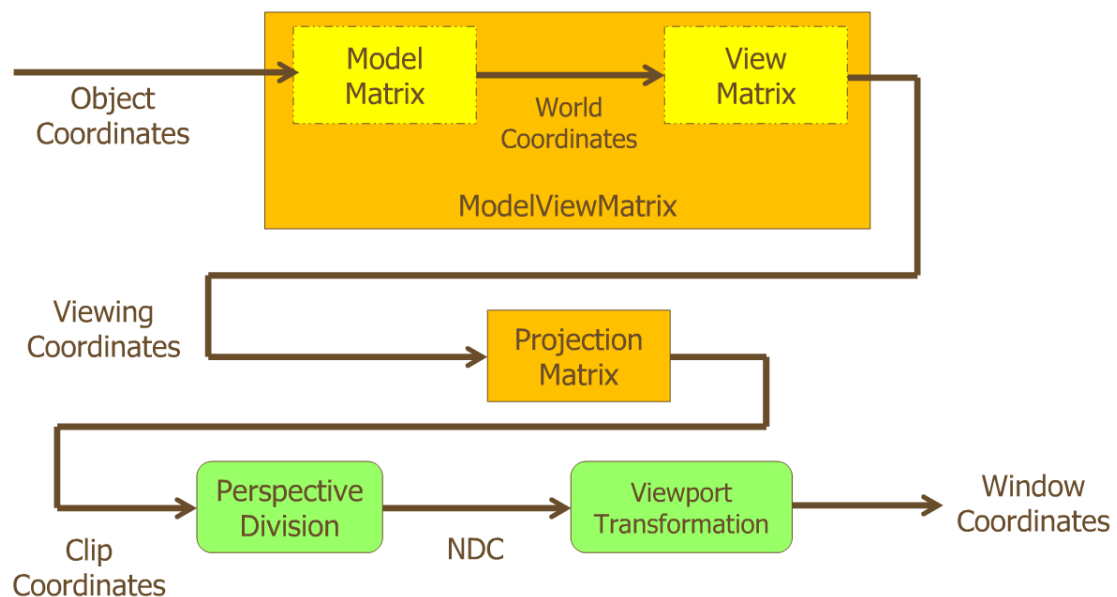
determining shape of viewing volume



viewport

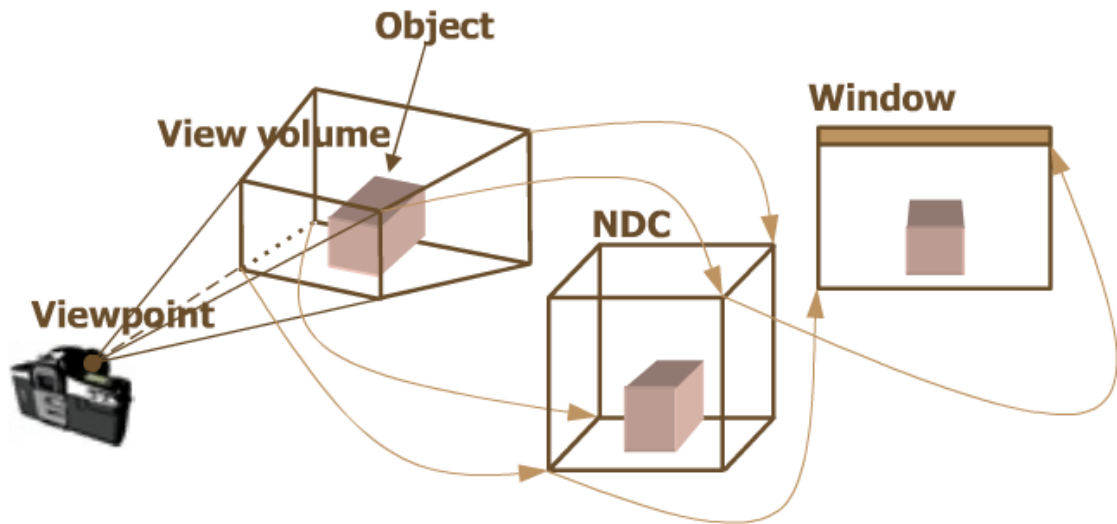
1. 拍照时第一步是架起三角架并把相机的镜头指向需要拍摄的场景，对应到 3D 变换为 **viewing transformation** （平移或是选择 **Camera** ）
2. 然后摄影师可能需要调整被拍场景中某个物体的角度，位置，比如摄影师给架好三角架后给你拍照时，可以要让你调整站立姿势或是位置。对应到 3D 绘制就是 **Modeling transformation** （调整所绘模型的位置，角度或是缩放比例）。
3. 之后摄影师可以需要调整镜头取景（拉近或是拍摄远景），相机取景框所能拍摄的场景会随镜头的伸缩而变换，对应到 3D 绘图则为 **Projection transformation**(裁剪投影场景)。
4. 按下快门后，对于数码相机可以直接在屏幕上显示当前拍摄的照片，一般可以充满整个屏幕（相当于将坐标做规范化处理 **NDC**），此时你可以使用缩放放大功能显示照片的部分。对应到 3D 绘图相当于 **viewport transformation** （可以对最终的图像缩放显示等）

下图为 Android OpenGL ES 坐标变换的过程：



- **Object Coordinate System:** 也称作 **Local coordinate System**, 用来定义一个模型本身的坐标系。
- **World Coordinate System:** 3d 虚拟世界中的绝对坐标系，定义好这个坐标系的原点就可以用来描述模型的实现的位置，**Camera** 的位置，光源的位置。
- **View Coordinate System:** 一般使用用来计算光照效果。
- **Clip Coordinate System:** 对 3D 场景使用投影变换裁剪视锥。
- **Normalized device coordinate System (NDC):** 规范后坐标系。

- **Windows Coordinate System:** 最后屏幕显示的 2D 坐标系统，一般原点定义在屏幕左上角。



对于 Viewing transformation (平移, 选择相机) 和 Modeling transformation (平移, 选择模型) 可以合并起来看, 只是应为向左移动相机, 和相机不同将模型右移的效果是等效的。

所以在 OpenGL ES 中,

- 使用 `GL10.GL_MODELVIEW` 来同时指定 viewing matrix 和 modeling matrix.
- 使用 `GL10.GL_PROJECTION` 指定投影变换, OpenGL 支持透视投影和正侧投影 (一般用于工程制图)。
- 使用 `glViewport` 指定 Viewport 变换。

此时再看看下面的代码, 就不是很难理解了, 后面就逐步介绍各种坐标变换。

- | | |
|-----|---|
| ▪ 1 | ▪ <code>public void onSurfaceChanged(GL10 gl, int width, int</code> |
| ▪ 2 | ▪ <code>height) {</code> |
| ▪ 3 | ▪ <code>// Sets the current view port to the new size.</code> |
| ▪ 4 | ▪ <code>gl.glViewport(0, 0, width, height);</code> |
| ▪ 5 | ▪ <code>// Select the projection matrix</code> |
| ▪ 6 | ▪ <code>gl.glMatrixMode(GL10.GL_PROJECTION);</code> |
| ▪ 7 | ▪ <code>// Reset the projection matrix</code> |
| ▪ 8 | ▪ <code>gl.glLoadIdentity();</code> |

▪ 9	▪ // Calculate the aspect ratio of the window
▪ 10	▪ GLU.gluPerspective(gl, 45.0f, (float) width / (float)
▪ 11	height, 0.1f, 100.0f);
▪ 12	▪ // Select the modelview matrix
▪ 13	▪ gl.glMatrixMode(GL10.GL_MODELVIEW);
▪ 14	▪ // Reset the modelview matrix
	▪ gl.glLoadIdentity();
	▪ }

Android OpenGL ES 开发教程(15)

通用的矩阵变换指令

Android OpenGL ES 对于不同坐标系下坐标变换，大都使用矩阵运算的方法来定义和实现的。这里介绍对应指定的坐标系（比如 `viewmodel`, `projection` 或是 `viewport`）Android OpenGL ES 支持的一些矩阵运算及操作。

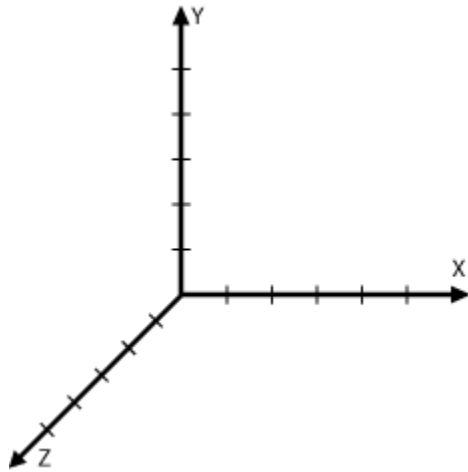
OpenGL ES 中使用四个分量(x,y,z,w)来定义空间一个点，使用 4 个分量来描述 3D 坐标称为[齐次坐标](#)：所谓齐次坐标就是将一个原本是 n 维的向量用一个 $n+1$ 维向量来表示。它有什么优点呢？许多图形应用涉及到几何变换，主要包括平移、旋转、缩放。以矩阵表达式来计算这些变换时，平移是矩阵相加，旋转和缩放则是矩阵相乘，综合起来可以表示为 $p' = m1 * p + m2$ ($m1$ 旋转缩放矩阵， $m2$ 为平移矩阵， p 为原向量， p' 为变换后的向量)。引入齐次坐标的目的主要是合并矩阵运算中的乘法和加法，表示为 $p' = M * p$ 的形式。即它提供了用矩阵运算把二维、三维甚至高维空间中的一个点集从一个坐标系变换到另一个坐标系的有效方法。它可以表示无穷远的点。 $n+1$ 维的齐次坐标中如果 $h=0$ ，实际上就表示了 n 维空间的一个无穷远点。对于齐次坐标 $[a,b,h]$ ，保持 a,b 不变， $|V| = (x1 * x1, y1 * y1, z1 * z1)^{1/2}$ 的过程就表示了标准坐标系中的一个点沿直线 $ax+by=0$ 逐渐走向无穷远处的过程。

为了实现 `viewing`, `modeling`, `projection` 坐标变换，需要构造一个 4×4 的矩阵 M ，对应空间中任意一个顶点 `vertex v`，经过坐标变换后的坐标 $v' = Mv$

矩阵本身可以支持加减乘除，对角线全为 1 的 4×4 矩阵成为单位矩阵 `Identity Matrix`。

- 将当前矩阵设为单位矩阵的指令 为 `glLoadIdentity()`.
- 矩阵相乘的指令 `glMultMatrix*()` 允许指定任意矩阵和当前矩阵相乘。
- 选择当前矩阵种类 `glMatrixMode()`. OpenGL ES 可以运行指定 `GL_PROJECTION`, `GL_MODELVIEW` 等坐标系，后续的矩阵操作将针对选定的坐标。
- 将当前矩阵设置成任意指定矩阵 `glLoadMatrix*()`
- 在栈中保存当前矩阵和从栈中恢复所存矩阵，可以使用 `glPushMatrix()` 和 `glPopMatrix()`
- 特定的矩阵变换平移 `glTranslatef()`, 旋转 `glRotatef()` 和缩放 `glScalef()`

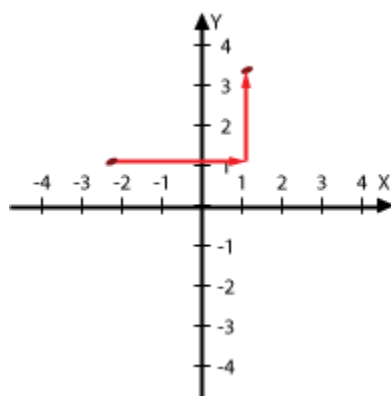
OpenGL 使用了右手坐标系，右手坐标系判断方法：在空间直角坐标系中，让右手拇指指向 x 轴的正方向，食指指向 y 轴的正方向，如果中指能指向 z 轴的正方向，则称这个坐标系为右手直角坐标系。



Translate 平移变换

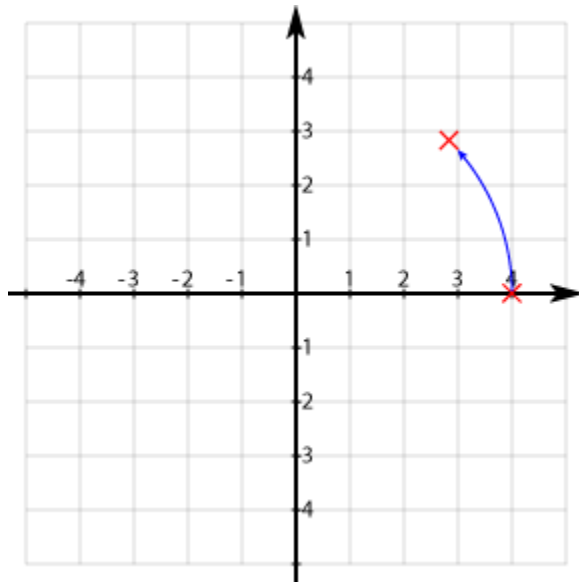
方法 `public abstract void glTranslatef(float x, float y, float z)` 用于坐标平移变换。

在上个例子中我们把需要显示的正方形后移了 4 个单位，就是使用的坐标的平移变换，可以进行多次平移变换，其结果为多个平移矩阵的累计结果，矩阵的顺序不重要，可以互换。



Rotate 旋转

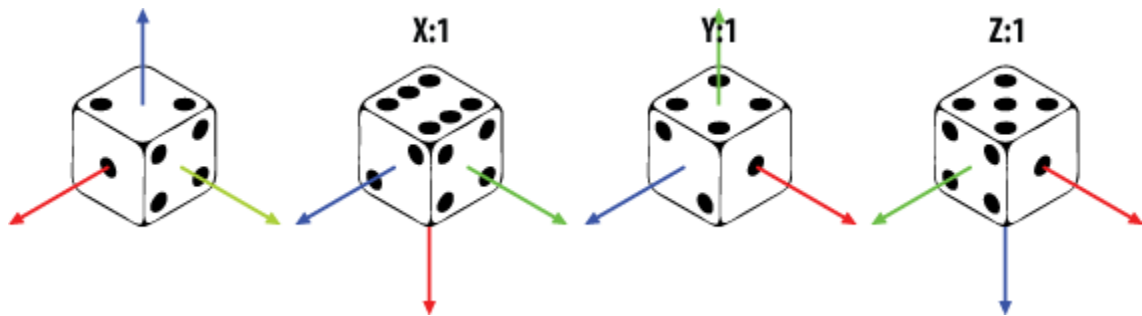
方法 `public abstract void glRotatef(float angle, float x, float y, float z)` 用来实现选择坐标变换，单位为角度。 (x,y,z) 定义旋转的参照矢量方向。多次旋转的顺序非常重要。



比如你选择一个骰子，首先按下列顺序

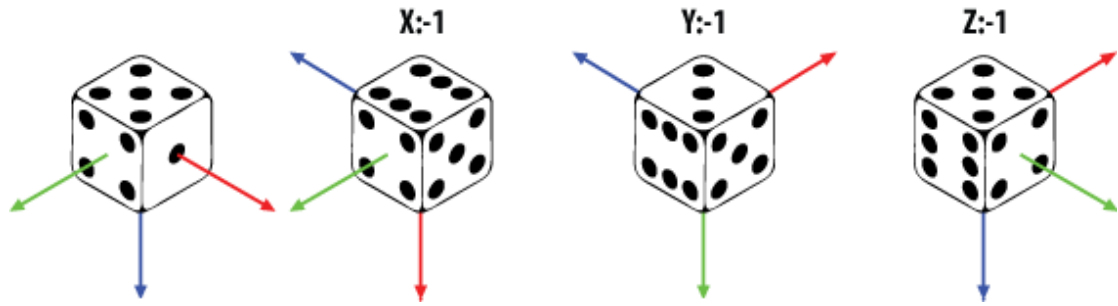
选择 3 次：

- 1 `gl.glRotatef(90f, 1.0f, 0.0f, 0.0f);`
- 2 `gl.glRotatef(90f, 0.0f, 1.0f, 0.0f);`
- 3 `gl.glRotatef(90f, 0.0f, 0.0f, 1.0f);`



然后打算逆向旋转回原先的初始状态，需要有如下旋转：

- 1 `gl.glRotatef(90f, -1.0f, 0.0f, 0.0f);`
- 2 `gl.glRotatef(90f, 0.0f, -1.0f, 0.0f);`
- 3 `gl.glRotatef(90f, 0.0f, 0.0f, -1.0f);`



或者如下旋转：

```
1      gl.glRotatef(90f, 0.0f, 0.0f, -1.0f);

2      gl.glRotatef(90f, 0.0f, -1.0f, 0.0f);

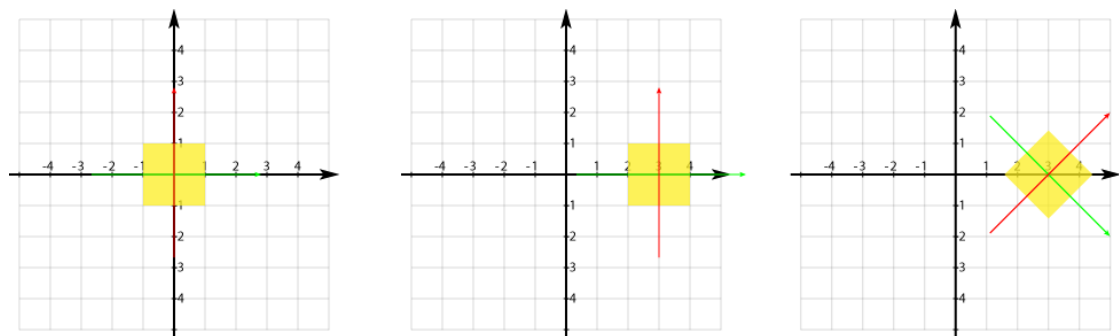
3      gl.glRotatef(90f, -1.0f, 0.0f, 0.0f);
```

旋转变换 $\text{glRotatef}(\text{angle}, -x, -y, -z)$ 和 $\text{glRotatef}(-\text{angle}, x, y, z)$ 是等价的，但选择变换的顺序直接影响最终坐标变换的结果。 角度为正时表示逆时针方向。

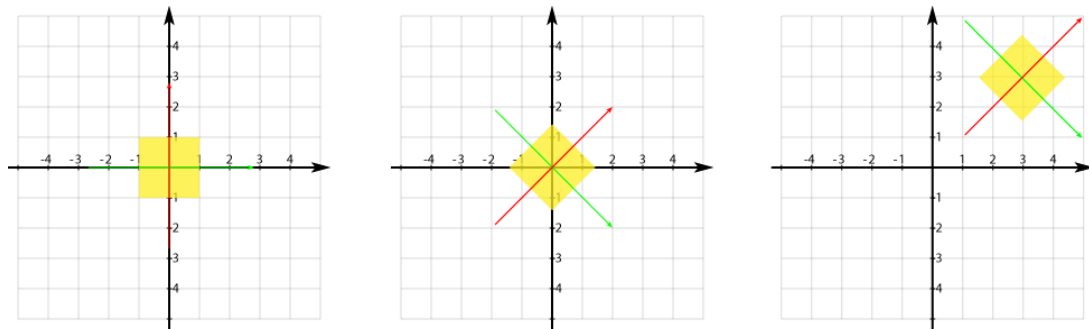
Translate & Rotate （平移和旋转组合变换）

在对 Mesh（网格，构成三维形体的基本单位）同时进行平移和选择变换时，坐标变换的顺序也直接影响最终的结果。

比如：先平移后旋转， 旋转的中心为平移后的坐标。



先选择后平移： 平移在则相对于旋转后的坐标系：

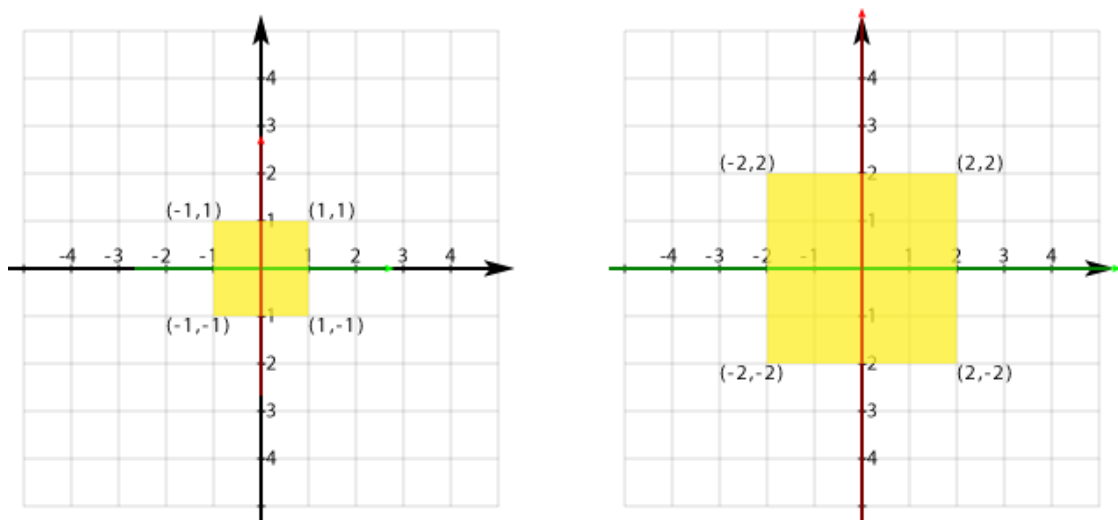


一个基本原则是，坐标变换都是相对于变换的 Mesh 本身的坐标系而进行的。

Scale（缩放）

方法 `public abstract void glScalef (float x, float y, float z)` 用于缩放变换。

下图为使用 `gl.glScalef(2f, 2f, 2f)` 变换后的基本，相当于把每个坐标值都乘以 2。

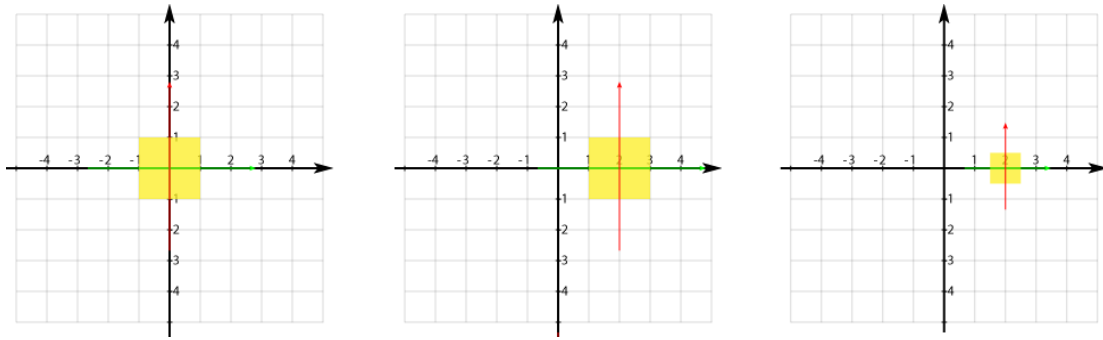


Translate & Scale（平移和缩放组合变换）

同样当需要平移和缩放时，变换的顺序也会影响最终结果。

比如先平移后缩放：

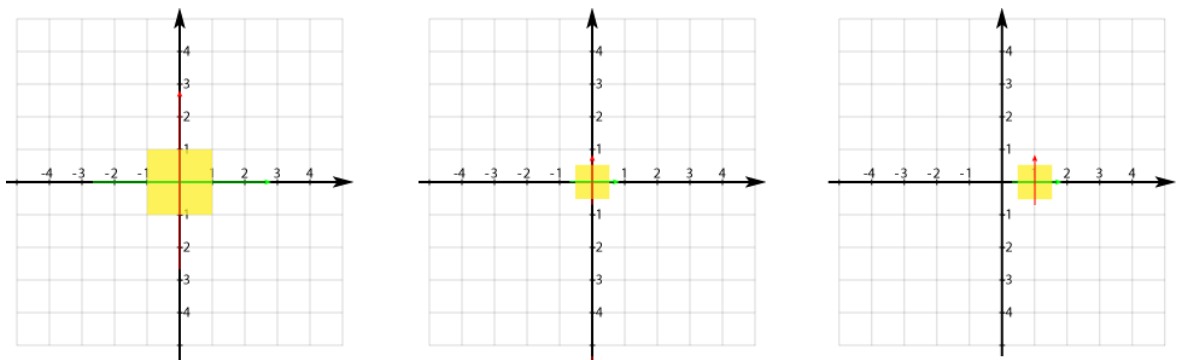
- 1 `gl.glTranslatef(2, 0, 0);`
- 2 `gl.glScalef(0.5f, 0.5f, 0.5f);`



如果调换一下顺序：

- 1 `gl.glScalef(0.5f, 0.5f, 0.5f);`
- 2 `gl.glTranslatef(2, 0, 0);`

结果就有所不同：



矩阵操作，单位矩阵

在进行平移，旋转，缩放变换时，所有的变换都是针对当前的矩阵（与当前矩阵相乘），如果需要将当前矩阵回复最初的无变换的矩阵，可以使用单位矩阵（无平移，缩放，旋转）。

`public abstract void glLoadIdentity\(\)。`

在栈中保存当前矩阵和从栈中恢复所存矩阵，可以使用

`public abstract void glPushMatrix\(\)和 public abstract void glPopMatrix\(\)。`

在进行坐标变换的一个好习惯是在变换前使用 `glPushMatrix` 保存当前矩阵，完成坐标变换操作后，再调用 `glPopMatrix` 恢复原先的矩阵设置。

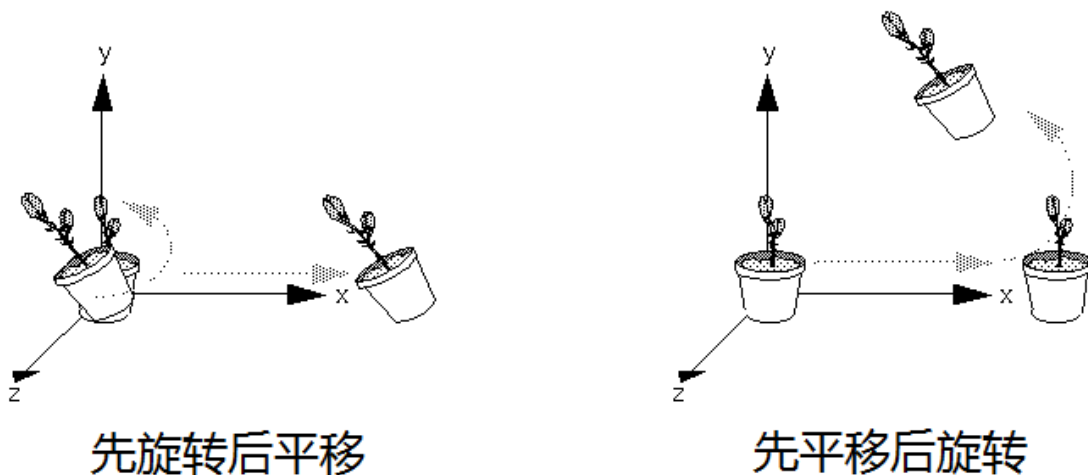
Android OpenGL ES 开发教程(16)

Viewing 和 Modeling(MODELVIEW) 变换

Viewing 和 Modeling 变换关系紧密，对应到相机拍照为放置三角架和调整被拍物体位置及角度，通常将这两个变换使用一个 `modelview` 变换矩阵来定义。对于同一个坐标变换，可以使用不同的方法来想象这个变换，比如将相机向某个方向平移一段距离，效果等同于将被拍摄的模型(model)向相反的方向平移同样的距离（相对运动）。两个不同的空间想象方法对于理解坐标变换各有其优缺点。你可以使用适合自己理解能力的方法来想象空间坐标变换。

下面我们使用一个由两个坐标变换组成的简单例子开始介绍 MODELVIEW 变换：一个变换为逆时针绕 Z 轴旋转 45 度，另一个变换为沿 X 轴平移。假定需要绘制的物体的尺寸和平移的距离相比要小的多从而你可以跟容易的看到平移效果。这个物体初始位置在坐标系的原点。

如果先旋转物体然后再平移，旋转后的物体的位置在 X 轴上面，但如果先平移后绕原点旋转物体，最终物体会出现在 $y=x$ 的直线上：



可以看到坐标变换的次序直接影响到最终的变换结果。所有的 Viewing 和 Modeling 变换操作都可以使用一个 4×4 的矩阵来表示，所有后续的 `glMultMatrix*()` 或其它坐标变换指令 会使用一个新的变换矩阵 M 于当前 `modelview` 矩阵 C 相乘得到一个新的变换矩阵 CM 。然后所有顶点坐标 v 都会和这个新的变换矩阵相乘。这个过程意味着最后发出的坐标变换指令实际上是第一个应用到顶点上的： CMv 。因此一种来理解坐标变换次序的方法是：使用逆序来指定坐标变换。

比如下面代码：

```
1      gl.glMatrixMode(GL_MODELVIEW);

2      gl.glLoadIdentity();

3      //apply transformation N

4      gl.glMultMatrixf(N);

5      //apply transformation M

6      gl.glMultMatrixf(M);

7      //apply transformation L

8      gl.glMultMatrixf(L);

9      //draw vertex

10     ...
```

上面代码,modelview 矩阵依次为 I （单位矩阵）, N , NM 最终为 NML ，最终坐标变换的结果为 $NMLv$ ，也就是 $N(M(Lv))$ ， v 首先与 L 相乘，结果 Lv 再和 M 相乘，所得结果 MLv 再和 N 相乘。可以看到坐标变换的次序和指令指定的次序正好相反。而实际代码运行时，坐标无需进行三次变换运算，顶点 v 只需和计算出的最终变换矩阵 NML 相乘一次就可以了。

因此如果你采用世界坐标系（原点和 X ， Y ， Z 轴方向固定）来思考坐标变换，代码中坐标变换指令的次序和 顶点和矩阵相乘的次序相反。比如，还是上面的例子，如果你想最终的物体出现在 X 轴上，此时必须是先旋转后平移，可以使用如下代码（ R 代表选择矩阵， T 代表平移矩阵）

```
1      gl.glMatrixMode(GL_MODELVIEW);

2      gl.glLoadIdentity();
```

```
3      //translation

4      gl.glMultMatrixf(T);

5      //rotation

6      gl.glMultMatrixf(R);

7      draw_the_object()

8
```

另外一种想象坐标变换的方法是忘记这种固定的坐标系，而是使用物体本身的局部坐标系，这种局部坐标系和物体的相对位置是固定的。所有的坐标变换操作都是针对物体的局部坐标系。使用这种方法，代码中矩阵相乘的次序和相对局部坐标系坐标变换的次序是一致的。（不管使用哪种方法来想象坐标变换，最终同样变换结果代码都是一样的，只是理解的方法不同）。还是使用上面旋转平移的例子。想象一个和物体连接一起的局部坐标系，如下图红色的坐标系，想象所有的坐标变换都是相对这个局部坐标系的，要使物体最终出现在 $y=x$ 上，可以想象先转动物体及其局部坐标系(**R**)，然后再平移物体及其局部坐标系 (**T**),这时代码的顺序和相对于物体局部坐标系的次序是相同的。

```
1      gl.glMatrixMode(GL_MODELVIEW);

2      gl.glLoadIdentity();

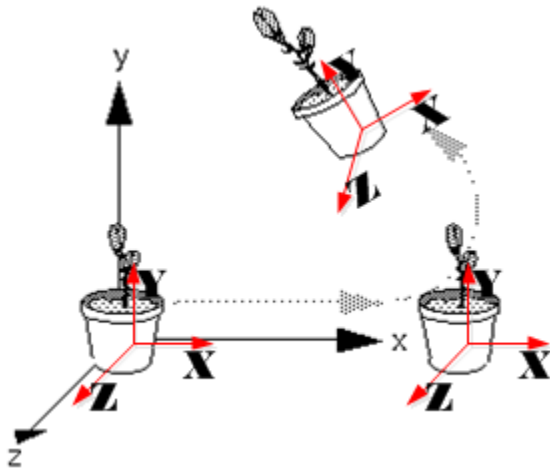
3      //rotation

4      gl.glMultMatrixf(R);

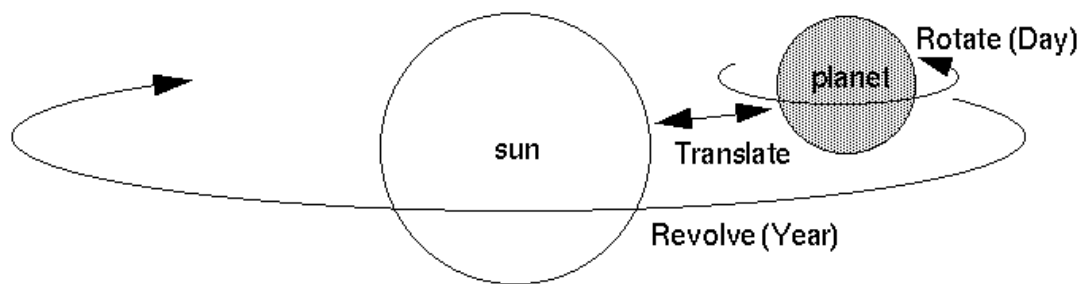
5      //translation

6      gl.glMultMatrixf(T);

7      draw_the_object()
```

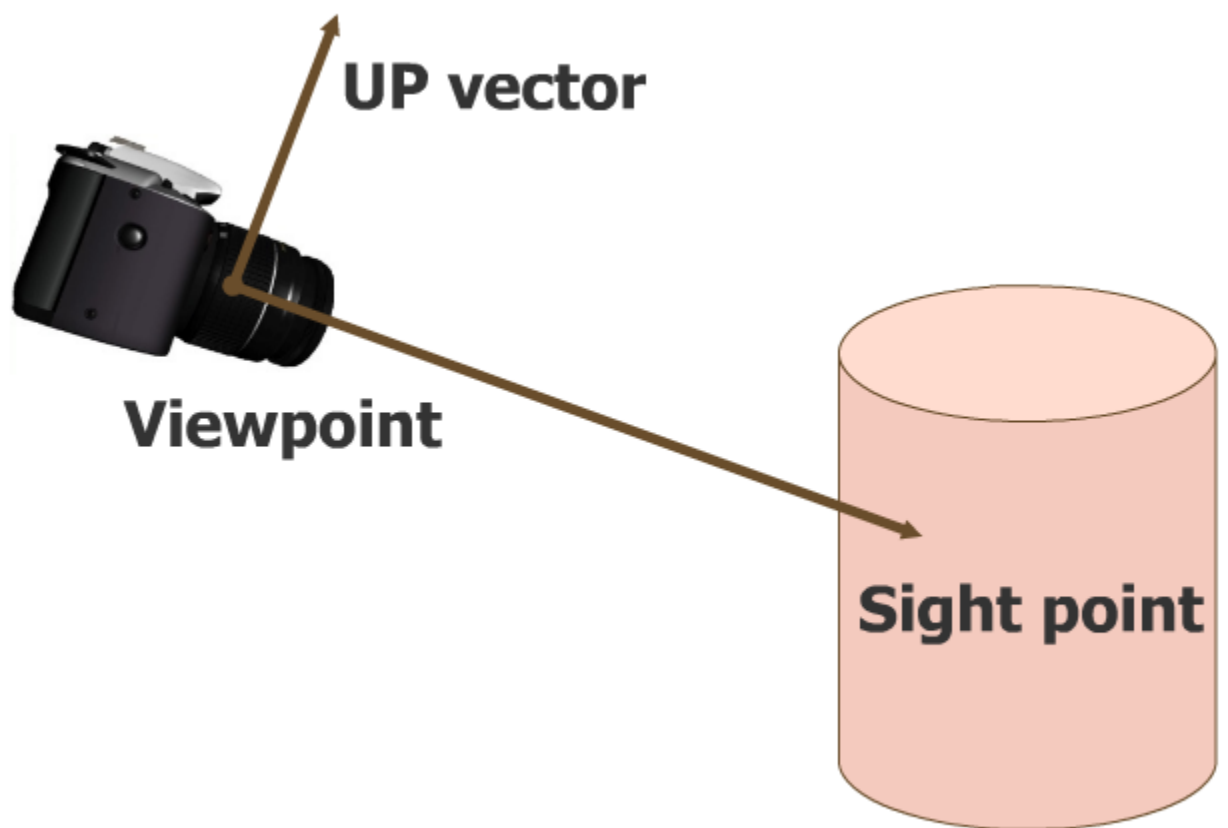



使用物体局部坐标系，可以更好的了解理解如机械手和太阳系之类的图形系统。



Android OpenGL ES 的 GLU 包有一个辅助函数 `gluLookAt` 提供一个更直观的方法来设置 `modelview` 变换矩阵：

```
void gluLookAt(GL10 gl, float eyeX, float eyeY, float eyeZ, float centerX, float
centerY, float centerZ, float upX, float upY, float upZ)
```



- $eyex, eyey, eyez$ 指定观测点的空间坐标。
- $tarx, tary, tarz$, 指定被观测物体的参考点的坐标。
- upx, upy, upz 指定观测点方向为“上”的向量。

注意：这些坐标都采用世界坐标系。

Android OpenGL ES 开发教程(17)

投影变换 **Projection**

前面 **ModelView** 变换相当于拍照时放置相机和调整被拍物体的位置和角度。投影变换则对应于调整相机镜头远近来取景。

下面代码设置当前 **Matrix** 模式为 **Projection** 投影矩阵：

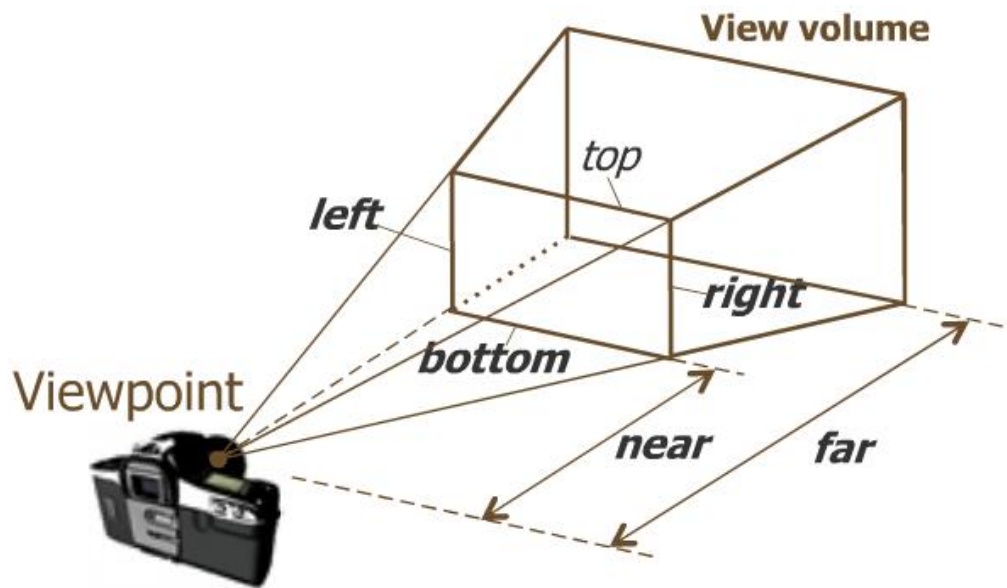
```
1      gl.glMatrixMode(GL_PROJECTION);  
  
2      gl.glLoadIdentity();
```

后续的坐标变换则针对投影矩阵。投影变换的目的是定义视锥(**viewing volume**)，视锥一方面定义了物体如何投影到屏幕（如透视投影或是正侧投影），另一方面视锥也定义了裁剪场景的区域大小。

OpenGL ES 可以使用两种不同的投影变换：透视投影（**Perspective Projection**）和正侧投影（**Orthographic Projection**）。

透视投影(**Perspective Projection**)

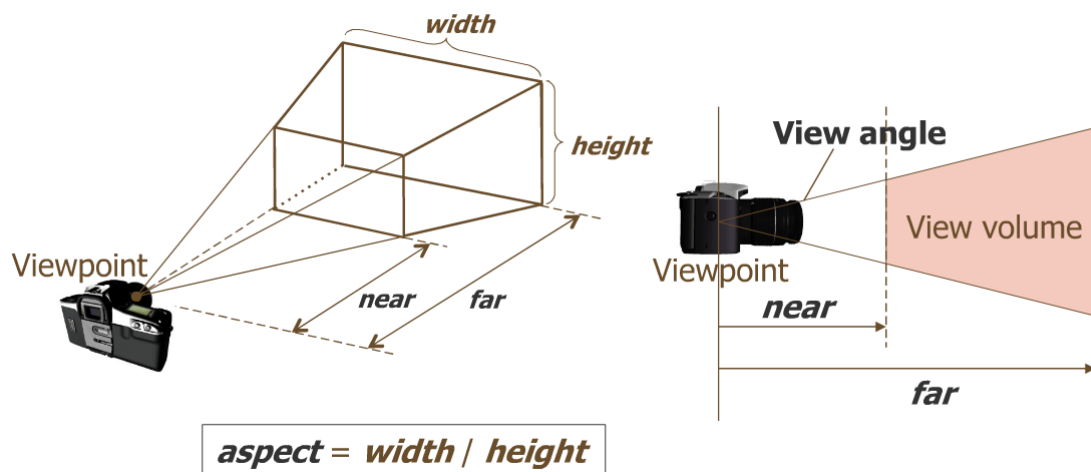
透视投影的特点是“近大远小”，也就是我们眼睛日常看到的世界。**OpenGL ES** 定义透视投影的函数为 **glFrustum()**：



```
public void glFrustumf(float left,float right,float bottom,float top,float
near,float far)
```

视锥由(left,bottom,-near) 和(right,top,-near) 定义了靠近观测点的裁剪面,near 和 far 定义了观测点和两个创建面直接的近距离和远距离。

在实际写代码时, Android OpenGL ES 提供了一个辅助方法 **gluPerspective()** 可以更简单的来定义一个透视投影变换:



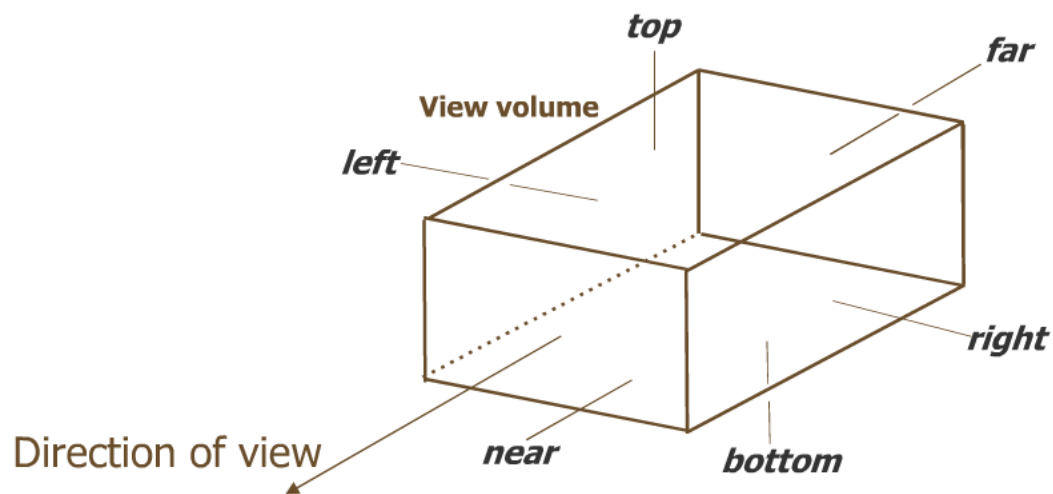
```
public static void gluPerspective(GL10 gl, float fovy, float aspect, float zNear,
float zFar)
```

- fovy: 定义视锥的 view angle.

- **aspect**: 定义视锥的宽高比。
- **zNear**: 定义裁剪面的近距离。
- **zFar**: 定义创建面的远距离。

正侧投影(Orthographic Projection)

正侧投影, 它的视锥为一长方体, 特点是物体的大小不随到观测点的距离而变化, 投影后可以保持物体之间的距离和夹角。它主要用在工程制图上。



定义正侧投影（也称作平移投影）的函数为：

```
public void glOrthof(float left, float right, float bottom, float top, float near, float far)
```

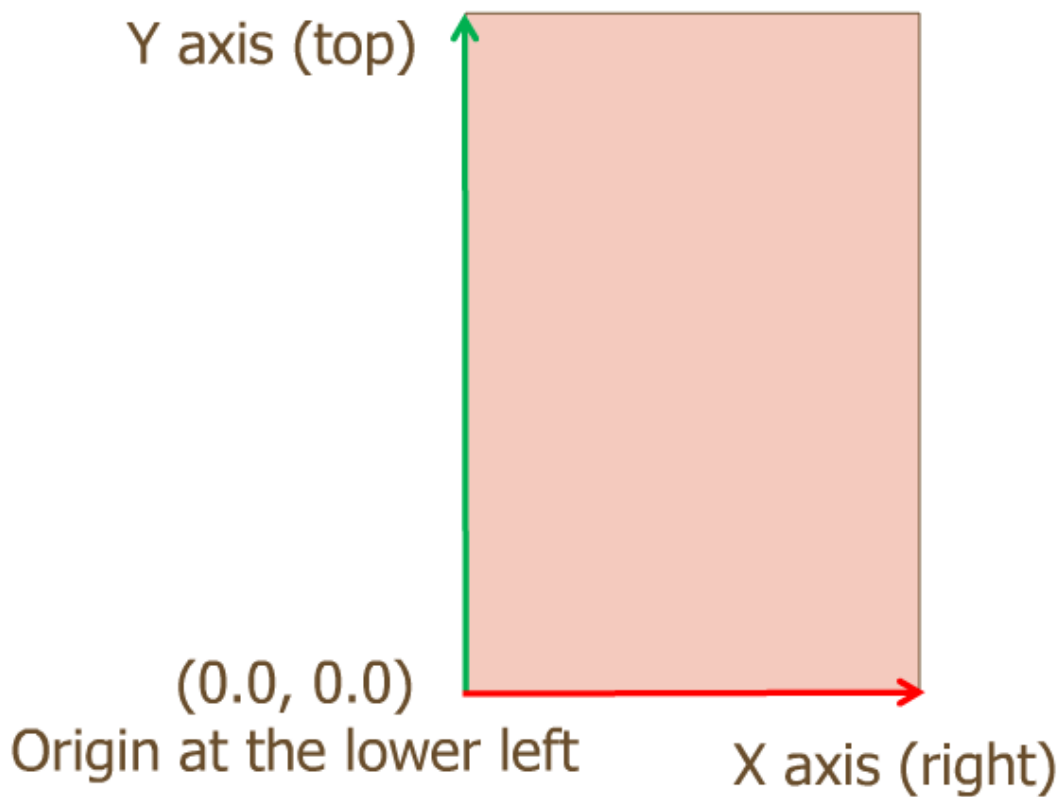
裁剪

场景中的图形的顶点经过 **modelview** 和 **projection** 坐标变换后, 所有处在 **Viewing volumn** 之外的顶点都会被裁剪掉, 透视投影和正侧投影都有 6 个裁剪面。所有处在裁剪面外部的顶点都需剪裁掉以提高绘图性能。

Android OpenGL ES 开发教程(18)

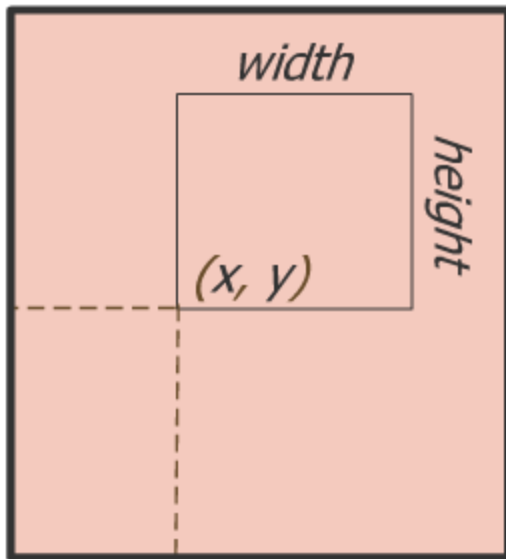
Viewport 变换

摄影师调整好相机和被拍摄物体的位置角度（modelview），对好焦距（projection）后，就可以按下快门拍照了，拍好的照片可以在计算机上使用照片浏览器查看照片，放大，缩小，拉伸，并可以将照片显示窗口在屏幕上任意拖放。对应到 3D 绘制就是 Viewport 变换，目前的显示器大多还是 2D 的，viewport（显示区域）为一个长方形区域，并且使用屏幕坐标系来定义：



OpenGL ES 中使用 **glViewport()** 来定义显示视窗的大小和位置：

glViewport(int x, int y, int width, int height)



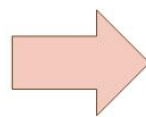
Android 缺省将 viewport 设置成和显示屏幕大小一致。

如果投影变换的宽度/高度比 (aspect) 和最后的 Viewport 的 width/height 比不一致的话，最后显示的图形就可能需要拉伸以适应 Viewport,从而可能造成图像变形。比如：现在的电视的显示模式有 4:3 和 16:9 或是其它模式，如果使用 16:9 的模式来显示原始宽高比为 4: 3 的视频，图像就有变形。、

Aspect ratio of view volume



Aspect ratio of viewport



Z 坐标变换

前面提到的 modelview, projection 变换 同样应用于 Z 轴坐标，但和屏幕坐标系中 x,y 坐标不同的时，在屏幕坐标系下 ,Android OpenGL ES 将 z 坐标重新编码，它的值总会在 0.0 到 1.0 之间。作为深度 depth 测试的依据。

我们在示例代码的 `OpenGLRenderer` 的 `onSurfaceChanged` 使用 and 屏幕一样大小的区域作为 `Viewport`，你也可以通过 `glViewport` 将视窗设成屏幕的局部某个区域。

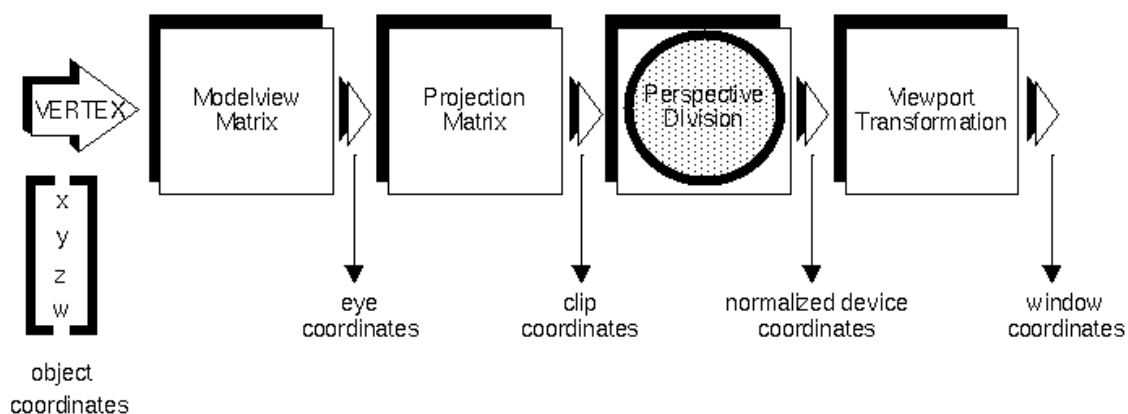
并且可以看到透视投影的 `aspect` 为 `width/height`，因此最后的图形不会有变形：

```
1      public void onSurfaceChanged(GL10 gl, int width, int height) {  
2          // Sets the current view port to the new size.  
3          gl.glViewport(0, 0, width, height);  
4          // Select the projection matrix  
5          gl.glMatrixMode(GL10.GL_PROJECTION);  
6          // Reset the projection matrix  
7          gl.glLoadIdentity();  
8          // Calculate the aspect ratio of the window  
9          GLU.gluPerspective(gl, 45.0f,  
10         (float) width / (float) height,  
11         0.1f, 100.0f);  
12         // Select the modelview matrix  
13         gl.glMatrixMode(GL10.GL_MODELVIEW);  
14         // Reset the modelview matrix  
15         gl.glLoadIdentity();  
16     }
```


Android OpenGL ES 开发教程(19)

绘制迷你太阳系

前面介绍了 3D 坐标系统和 3D 坐标变换以及在 OpenGL ES 中坐标变换的过程，并与相机拍照片的过程做类比，以便更好的理解这 OpenGL 中构造 3D 模型的一部步骤：



本例提供绘制一个迷你太阳系系统作为前面知识的总结，这个迷你太阳系，有一个红色的太阳，一个蓝色的地球和一个白色的月亮构成：

- 太阳居中，逆时针自转。
- 地球绕太阳顺时针公转，本身不自转。
- 月亮绕地球顺时针公转，自身逆时针自转。

为简单起见，使用一个 2D 五角星做为天体而没有使用球体（绘制球体在后面有介绍），构造一个 **Star** 类：

```
1      public class Star {  
2          // Our vertices.  
3          protected float vertices[];  
4          // Our vertex buffer.
```

```

5      protected FloatBuffer vertexBuffer;

6      public Star() {

7          float a=(float)(1.0f/(2.0f-2f*Math.cos(72f*Math.PI/180.f)));

8          float bx=(float)(a*Math.cos(18*Math.PI/180.0f));

9          float by=(float)(a*Math.sin(18*Math.PI/180f));

10         float cy=(float)(-a * Math.cos(18*Math.PI/180f));

11         vertices=new float[]{

12             0,a,0.5f,cy,-bx,by,bx,by,-0.5f,cy

13         };

14         ByteBuffer vbb

15             = ByteBuffer.allocateDirect(vertices.length * 4);

16         vbb.order(ByteOrder.nativeOrder());

17         vertexBuffer = vbb.asFloatBuffer();

18         vertexBuffer.put(vertices);

19         vertexBuffer.position(0);

20     }

21     /**

22     * This function draws our star on screen.

23     * @param gl

```

```
24    */
25    public void draw(GL10 gl) {
26        // Counter-clockwise winding.
27        gl.glFrontFace(GL10.GL_CCW);
28        // Enable face culling.
29        gl.glEnable(GL10.GL_CULL_FACE);
30        // What faces to remove with the face culling.
31        gl.glCullFace(GL10.GL_BACK);
32        // Enabled the vertices buffer for writing
33        //and to be used during
34        // rendering.
35        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
36        // Specifies the location and data format of
37        //an array of vertex
38        // coordinates to use when rendering.
39        gl.glVertexPointer(2, GL10.GL_FLOAT, 0,
40        vertexBuffer);
41        gl.glDrawArrays(GL10.GL_LINE_LOOP, 0,5);
42        // Disable the vertices buffer.
```

```
43      gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);

44      // Disable face culling.

45      gl.glDisable(GL10.GL_CULL_FACE);

46  }

47  }

48
```

Star 定义了五角星的五个顶点，并使用 `glDrawArrays` 来绘制五角星，因此 `vertices` 顶点的顺序比较重要。

然后定义一个 `DrawSolarSystem` 来绘制这个迷你太阳系：

```
1      public class DrawSolarSystem extends OpenGLActivity

2      implements IOpenGLDemo{

3      private Star sun=new Star();

4      private Star earth=new Star();

5      private Star moon=new Star();

6      private int angle=0;

7      /** Called when the activity is first created. */

8      @Override

9      public void onCreate(Bundle savedInstanceState) {

10     super.onCreate(savedInstanceState);
```

```
11     }

12     public void DrawScene(GL10 gl) {

13         super.DrawScene(gl);

14         gl.glLoadIdentity();

15         GLU.gluLookAt(gl,0.of, 0.of, 15.of,

16         0.of, 0.of, 0.of,

17         0.of, 1.of, 0.of);

18         // Star A

19         // Save the current matrix.

20         gl.glPushMatrix();

21         // Rotate Star A counter-clockwise.

22         gl.glRotatef(angle, 0, 0, 1);

23         gl.glColor4f(1.of, 0.of, 0.of, 1.of);

24         // Draw Star A.

25         sun.draw(gl);

26         // Restore the last matrix.

27         gl.glPopMatrix();

28         // Star B

29         // Save the current matrix
```

```
30     gl.glPushMatrix();

31     // Rotate Star B before moving it,

32     //making it rotate around A.

33     gl.glRotatef(-angle, 0, 0, 1);

34     // Move Star B.

35     gl.glTranslatef(3, 0, 0);

36     // Scale it to 50% of Star A

37     gl.glScalef(.5f, .5f, .5f);

38     gl.glColor4f(0.0f, 0.0f, 1.0f, 1.0f);

39     // Draw Star B.

40     earth.draw(gl);

41     // Star C

42     // Save the current matrix

43     gl.glPushMatrix();

44     // Make the rotation around B

45     gl.glRotatef(-angle, 0, 0, 1);

46     gl.glTranslatef(2, 0, 0);

47     // Scale it to 50% of Star B

48     gl.glScalef(.5f, .5f, .5f);
```

```
49      // Rotate around it's own center.

50      gl.glRotatef(angle*10, 0, 0, 1);

51      gl.glColor4f(1.0f, 1.0f, 1.0f, 1.0f);

52      // Draw Star C.

53      moon.draw(gl);

54      // Restore to the matrix as it was before C.

55      gl.glPopMatrix();

56      // Restore to the matrix as it was before B.

57      gl.glPopMatrix();

58      // Increse the angle.

59      angle++;

60  }

61  }

62

63

64

65

66
```

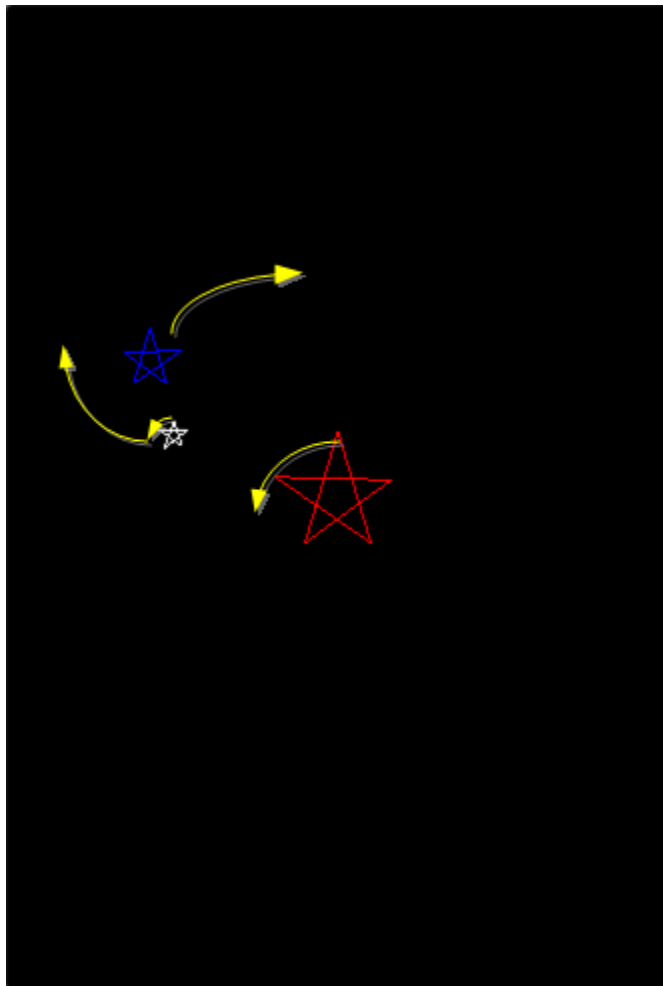
使用 GLU 的 `gluLookAt` 来定义 `modelview Matrix`，把相机放在正对太阳中心 $(0,0,0)$ ，距离 15 $(0,0,15)$ 。

使用 `glPushMatrix` 和 `glPopMatrix` 来将当前 `Matrix` 入栈或是出栈。

首先将当前 `matrix` 入栈，以红色绘制太阳，并逆向转动，将当前 `matrix` 入栈的目的是在能够在绘制地球时恢复当前栈。

然后绘制地球，使用局部坐标系来想象地球和太阳之间的相对运动，地球离开一距离绕太阳公转，相当于先旋转地球的局部坐标系，然后再平移地球的局部坐标系。对应到代码为先 `glRotatef` ,然后 `glTranslate` .

最后是绘制月亮，使用类似的空间想象方法。



Android OpenGL ES 开发教程(21)

定义 3D 模型的前面和后面

OpenGL ES 使用也只能使用三角形来定义一个面(Face)，为了获取绘制的高性能，一般情况不会同时绘制面的前面和后面，只绘制面的“前面”。虽然“前面”“后面”的定义可以应人而易，但一般为所有的“前面”定义统一的顶点顺序(顺时针或是逆时针方向)。只绘制“前面”的过程称为”Culling”。

下面代码设置逆时针方法为面的“前面”：

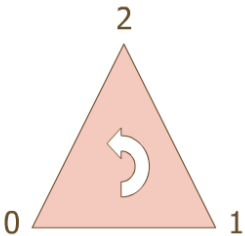
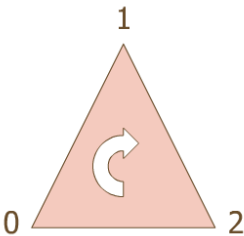
```
1      gl.glFrontFace(GL10.GL_CCW);
```

打开 忽略“后面”设置：

```
1      gl.glEnable(GL10.GL_CULL_FACE);
```

明确指明“忽略“哪个面的代码如下：

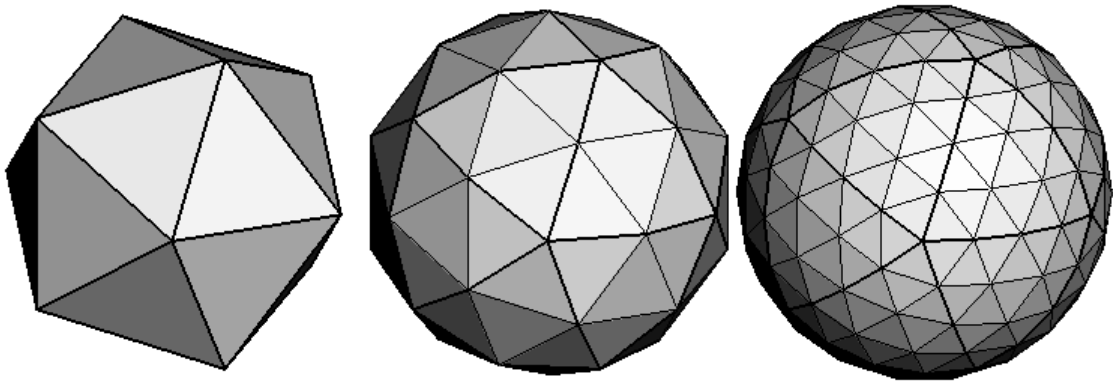
```
1      gl.glCullFace(GL10.GL_BACK);
```

		
GL_CCW mode	Front face	Back face
GL_CW mode	Back face	Front face

Android OpenGL ES 开发教程(22)

绘制一个球体

OpenGL ES 只能通过绘制三角形来构造几何图形，比如前面绘制的 20 面体 [Android OpenGL ES 开发教程\(12\): 绘制一个 20 面体](#)，通过增加正多面体的边数，就可以构造出一个球体：



在项目中创建一个 Sphere 类，它的 Draw 方法，通过绘制三角形来构造球体，并且为其添加法线，法线主要用于光照效果，将在后面介绍。

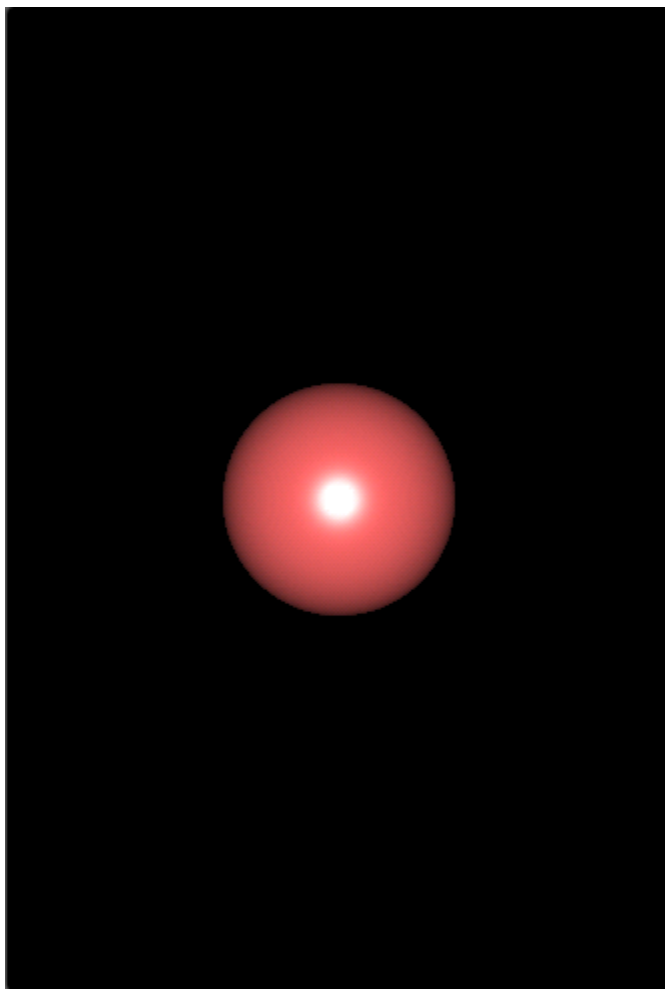
```
1      public void draw(GL10 gl) {  
2          float theta, pai;  
3          float co, si;  
4          float r1, r2;  
5          float h1, h2;  
6          float step = 2.0f;  
7          float[][] v = new float[32][3];  
8          ByteBuffer vbb;  
9          FloatBuffer vBuf;  
10         vbb = ByteBuffer.allocateDirect(v.length * v[0].length * 4);  
11         vbb.order(ByteOrder.nativeOrder());
```

```
12     vBuf = vbb.asFloatBuffer();
13     gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
14     gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);
15     for (pai = -90.0f; pai < 90.0f; pai += step) {
16         int n = 0;
17         r1 = (float)Math.cos(pai * Math.PI / 180.0);
18         r2 = (float)Math.cos((pai + step) * Math.PI / 180.0);
19         h1 = (float)Math.sin(pai * Math.PI / 180.0);
20         h2 = (float)Math.sin((pai + step) * Math.PI / 180.0);
21         for (theta = 0.0f; theta <= 360.0f; theta += step) {
22             co = (float)Math.cos(theta * Math.PI / 180.0);
23             si = -(float)Math.sin(theta * Math.PI / 180.0);
24             v[n][0] = (r2 * co);
25             v[n][1] = (h2);
26             v[n][2] = (r2 * si);
27             v[n + 1][0] = (r1 * co);
28             v[n + 1][1] = (h1);
29             v[n + 1][2] = (r1 * si);
30             vBuf.put(v[n]);
31             vBuf.put(v[n + 1]);
32             n += 2;
```

```
33     if(n>31){
34         vBuf.position(o);
35         gl.glVertexPointer(3, GL10.GL_FLOAT, o, vBuf);
36         gl.glNormalPointer(GL10.GL_FLOAT, o, vBuf);
37         gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, o, n);
38         n = o;
39         theta -= step;
40     }
41 }
42 vBuf.position(o);
43 gl.glVertexPointer(3, GL10.GL_FLOAT, o, vBuf);
44 gl.glNormalPointer(GL10.GL_FLOAT, o, vBuf);
45 gl.glDrawArrays(GL10.GL_TRIANGLE_STRIP, o, n);
46 }
47 gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
48 gl.glDisableClientState(GL10.GL_NORMAL_ARRAY);
49 }
50
51
52
53
```

有了 Sphere 类， 创建一个 DrawSphere Activity 来绘制球体，为了能看出 3D 效果，给场景中添加光源（后面介绍）

```
1      public void DrawScene(GL10 gl) {  
2          super.DrawScene(gl);  
3          initScene(gl);  
4          sphere.draw(gl);  
5      }
```



本例[下载](#)

Android OpenGL ES 开发教程(23)

Framebuffer

OpenGL ES 中的 FrameBuffer 指的是存储像素的内存空间。对应一个二维图像，如果屏幕分辨率为 1280X1024，如果屏幕支持 24 位真彩色 (RGB)，则存储这个屏幕区域的内存至少需要 1024X1280X3 个字节。此外如果需要支持透明度 (Alpha)，则一个像素需要 4 个字节。

对应 3D 图像来说，上面存储显示颜色的 Buffer 称为 Color Buffer，除 Color Buffer 之外，还需要存储每个像素和 View Point 之间的距离，OpenGL ES 中使用 Depth Buffer 存储像素与眼睛 (eye 或是 view point) 的距离，Depth Buffer 也可称为 z Buffer。

此外 OpenGL ES 还定义了一个称为遮罩(Stencil) Buffer, 可以将屏幕显示局限在某个由 Stencil Buffer 定义的区域，在日常生活中常见的 Stencil Buffer 示例时使用纸质模板在墙上或是 T 恤上印刷文字或是图像：

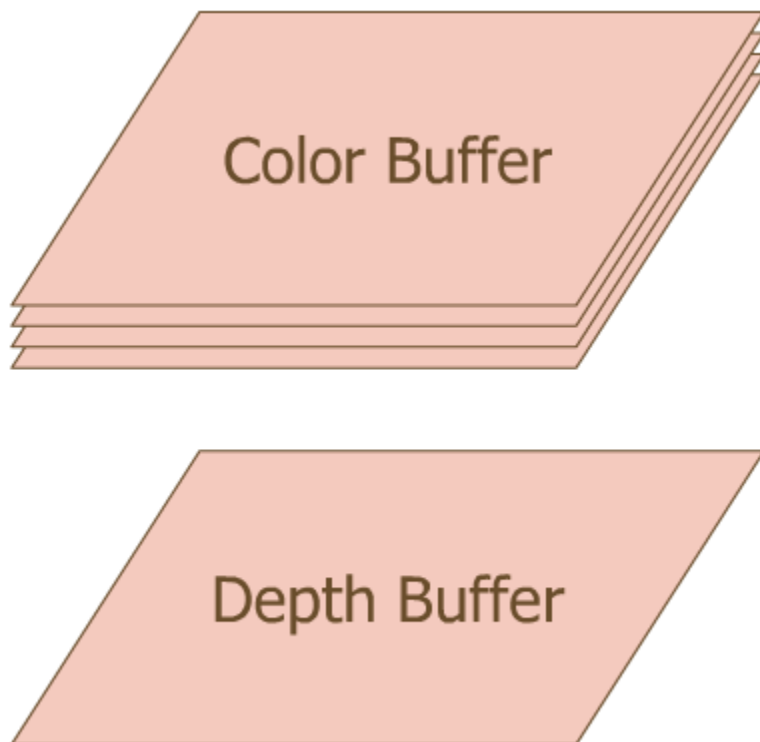


在 OpenGL ES 允许配置 Color Buffer 中 R,G,B,A 的颜色位数，Depth Buffer 的位数，以及 Stencil Buffer 的位数：

参数	含义
GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS	Number of bits per R, G, B, or A component in the color buffers

GL_DEPTH_BITS	Number of bits per pixel in the depth buffer
GL_STENCIL_BITS	Number of bits per pixel in the stencil buffer

在最终 OpenGL ES 写入这些 Buffer 时，OpenGL ES 提供一些 Mask 函数可以控制 Color Buffer 中 RGBA 通道，是否允许写入 Depth Buffer 等，这些 Mask 函数可以打开或是关闭某个通道，只有通道打开后，对应的分量才会写入指定 Buffer，比如你可以关闭红色通道，这样最后写道 Color Buffer 中就不含有红色。这些函数有 glColorMask, glDepthMask, glStencilMask。这些在后面有更详细的介绍。

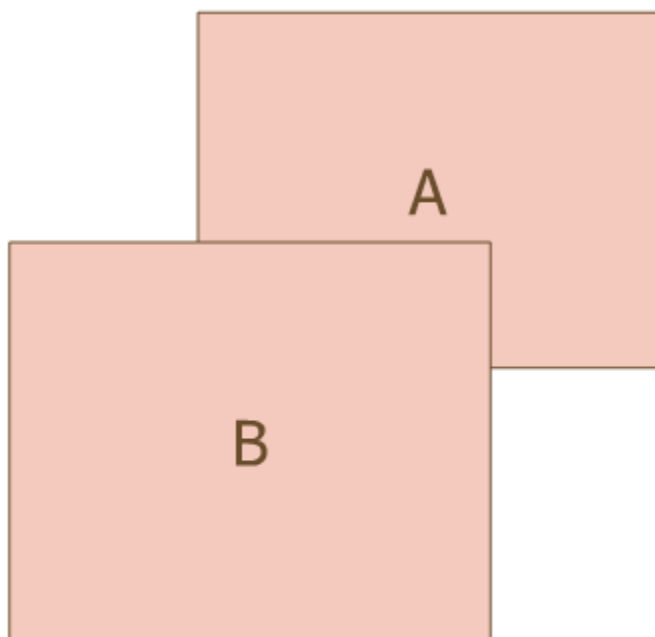


Android OpenGL ES 开发教程(24)

Depth Buffer

OpenGL ES 中 Depth Buffer 保存了像素与观测点之间的距离信息，在绘制 3D 图形时，将只绘制可见的面而不去绘制隐藏的面，这个过程叫”Hidden surface removal”，采用的算法为”The depth buffer algorithm”。

一般来说，填充的物体的顺序和其顺序是一致的，而要准确的显示绘制物体在 Z 轴的前后关系，就需要先绘制距离观测点(ViewPoint)最远的物体，再绘制离观测点较远的物体，最后绘制离观测点最近的物体，因此需要对应所绘制物体进行排序。OpenGL ES 中使用 Depth Buffer 存放需绘制物体的相对距离。



A绘制完成后，再绘制B

The depth buffer

algorithm 在 OpenGL ES 3D 绘制过程中这个算法是自动被采用的，但是了解这个算法有助于理解 OpenGL ES 部分 API 的使用。

这个算法的基本步骤如下：

1. 将 Depth Buffer 中的值使用最大值清空整个 Depth Buffer，这个最大值缺省为 1.0，为距离 viewPoint 最远的裁剪的距离。最小值为 0，表示距离

viewPoint 最近的裁剪面的距离。距离大小为相对值而非实际距离，这个值越大表示与 Viewpoint 之间的距离越大。因此将初值这设为 1.0 相当于清空 Depth Buffer。

2. 当 OpenGL 栅格化所绘制基本图形(Primitive)，将计算该 Primitive 与 viewpoint 之间的距离，保存在 Depth Buffer 中。
3. 然后比较所要绘制的图形的距离和当前 Depth Buffer 中的值，如果这个距离比 Depth Buffer 中的值小，表示这个物体离 viewPoint 较近，Open GL 则更像相应的 Color Buffer 并使用这个距离更新 Depth Buffer,否则，表示当前要绘制的图形在已绘制的部分物体后面，则无需绘制该图形(删除)。

这个过程也称为”Depth Test”(深度测试)。

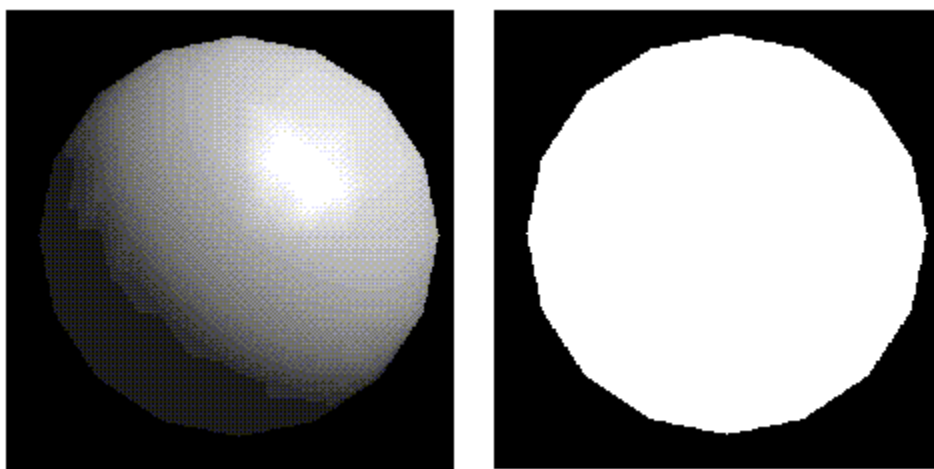
下面给出了 OpenGL ES 中与 Depth Buffer 相关的几个方法：

- gl.Clear(GL10.GL_DEPTH_BUFFER_BIT) 清空 Depth Buffer (赋值为 1.0)通常清空 Depth Buffer 和 Color Buffer 同时进行。
- gl.glClearDepthf(float depth) 指定清空 Depth Buffer 是使用的值，缺省为 1.0，通常无需改变这个值，
- gl.glEnable(GL10.GL_DEPTH_TEST) 打开 depth Test
- gl.glDisable(GL10.GL_DEPTH_TEST) 关闭 depth Test

Android OpenGL ES 开发教程(25)

OpenGL 光照模型

前面绘制球体时 [Android OpenGL ES 开发教程\(22\): 绘制一个球体](#)，为了能看出 3D 效果，给场景中添加光源。如果没有光照，绘出的球看上去和一个二维平面上圆没什么差别，如下图，左边为有光照效果的球体，右边为同一个球体但没有设置光源，看上去就没有立体效果，因此 OpenGL 光照效果对显示 3D 效果非常明显。

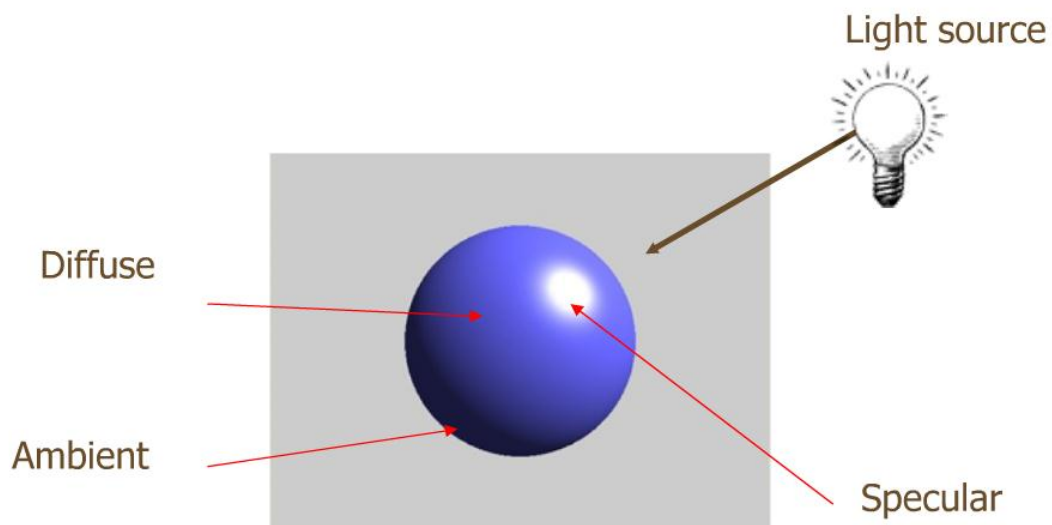


在

OpenGL 光照模型中光源和光照效果可以细分为红，绿，蓝三个部分，光源由红，绿，蓝强度来定义，而物体表面材料由其反射红，绿，蓝的程度和方向来定义。OpenGL 光照模型使用的计算公式是对于现实世界光照的一个近似但效果非常好并适合快速计算。

OpenGL 光照模型中定义的光源可以分别控制，打开或关闭，OpenGL ES 支持最多八个光源。

OpenGL 光照模型中最终的光照效果可以分为四个组成部分：**Emitted**(光源)，**ambient**(环境光)，**diffuse**(漫射光)和 **specular**（镜面反射光），最终结果由这四种光叠加而成。



Emitted : 一般只发光物体或者光源, 这种光不受其它光源的影响。

ambient: 指光线经过多次反射后已经无法得知其方向(可以看作来自所有方向), 可以成为环境光, 该光源如果射到某个平面, 其反射方向为所有方向。**Ambient 不依赖于光源的方向。**

diffuse: 当一束平行的入射光线射到粗糙的表面时, 因面上凹凸不平, 所以入射线虽然互相平行, 由于各点的法线方向不一致, 造成反射光线向不同的方向无规则地反射, 这种反射称之为“漫反射”或“漫射”。这个反射的光则称为漫射光。漫射光射到某个平面时, 其反射方向也为所有方向。**diffuse 只依赖于光源的方向和法线的方向。**

specular : 一般指物体被光源直射的高亮区域, 也可以成为镜面反射区, 如金属。**specular 依赖于光源的方向, 法线的方向和视角的方向。**

尽管光源可能只发送某一频率的光线, 但 **ambient**, **diffuse** 和 **specular** 可能不同。比如使用白光照射一堵红墙, 散射的光线可能为红色。**OpenGL** 允许为光源分别设置红, 绿, 蓝三个元素的值。

最终决定所看到物体的颜色除了光源的颜色和方向外, 还取决于物体本身的颜色, 比如红色的光照在红色的物体和蓝色的物体, 最终看到的物体一个还是红色, 一个为黑色。**OpenGL** 中对物体材料(**Material**)的颜色是通过其反射红, 绿, 蓝的比例来定义的。和光源一样, 物体的颜色也可以有不同的 **ambient**, **diffuse** 和 **specular**, 表现为反射这些光的比例。**ambient**, **diffuse** 反射通常为同样的颜色,

而 **specular** 常常表现为白色或灰色光，如使用白光照射一个红色的球，球的大部分区域显示为红色，而高亮区域为白色。

Android OpenGL ES 开发教程(26)

设置光照效果 Set Lighting

上一篇简单介绍了 OpenGL 中使用的光照模型，本篇结合 OpenGL ES API 说明如何使用光照效果：

- 设置光源
- 定义法线
- 设置物体材料光学属性

光源

OpenGL ES 中可以最多同时使用八个光源，分别使用 0 到 7 表示。

OpenGL ES 光源可以分为

- 平行光源(Parallel light source)，代表由位于无限远处均匀发光体，太阳可以近似控制平行光源。
- 点光源(Spot light source) 如灯泡就是一个点光源，发出的光可以指向 360 度，可以为点光源设置光衰减属性 (attenuation)或者让点光源只能射向某个方向（如射灯）。
- 可以为图形的不同部分设置不同的光源。

下面方法可以打开某个光源，使用光源首先要开光源的总开关：

```
1      gl.glEnable(GL10.GL_LIGHTING);
```

然后可以再打开某个光源如 0 号光源：

```
1      gl.glEnable(GL10.GL_LIGHT0);
```

设置光源方法如下：

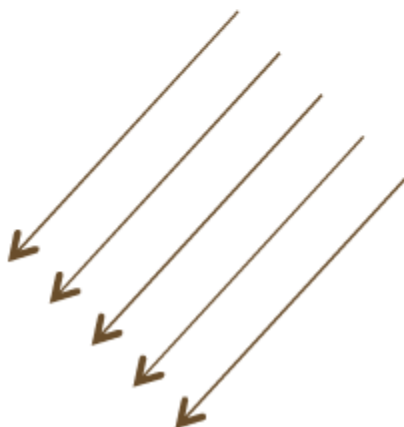
- `public void glLightfv(int light,int pname, FloatBuffer params)`
- `public void glLightfv(int light,int pname,float[] params,int offset)`
- `public void glLightf(int light,int pname,float param)`

- `light` 指光源的序号，OpenGL ES 可以设置从 0 到 7 共八个光源。
- `pname`: 光源参数名称，可以有如下：`GL_SPOT_EXPONENT`,
`GL_SPOT_CUTOFF`,`GL_CONSTANT_ATTENUATION`,
`GL_LINEAR_ATTENUATION`,`GL_QUADRATIC_ATTENUATION`,
`GL_AMBIENT`,`GL_DIFFUSE`,`GL_SPECULAR`,
`GL_SPOT_DIRECTION`,`GL_POSITION`
- `params` 参数的值（数组或是 Buffer 类型）。

其中为光源设置颜色的参数类型为 `GL_AMBIENT`,
`GL_DIFFUSE`,`GL_SPECULAR`, 可以分别指定 R,G,B,A 的值。

指定光源的位置的参数为 `GL_POSITION`, 值为(x,y,z,w):

平行光将 `w` 设为 0.0, (x,y,z)为平行光的方向:



`GL_POSITION`

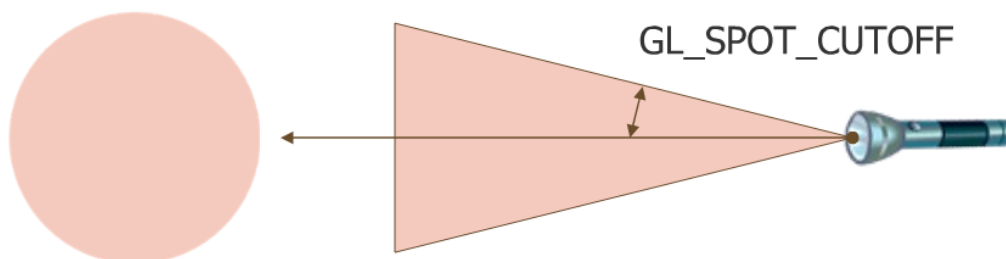
对于点光源，将 `w` 设成非 0 值，通常设为 1.0. (x,y,z)为点光源的坐标位置。



将点光源设置成聚光灯，需要同时设置 `GL_SPOT_DIRECTION`, `GL_SPOT_CUTOFF` 等 参数, `GL_POSITION` 的设置和点光源类似：将 `w` 设成非 0 值，通常设为 1.0. (x,y,z) 为点光源的坐标位置。而对于 `GL_SPOT_DIRECTION` 参数，设置聚光的方向 (x,y,z)

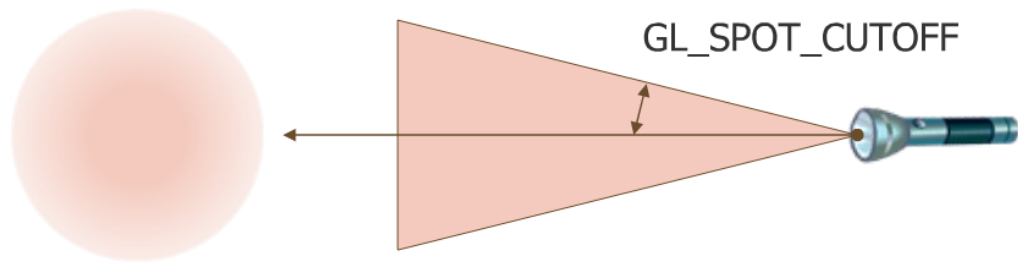


`GL_SPOT_CUTOFF` 参数设置聚光等发散角度（0 到 90 度）



`GL_SPOT_EXPONENT` 给出了聚灯光源汇聚光的程度，值越大，则聚光区域越小（聚光能力更强）。

GL_SPOT_EXPONENT



对应点光源（包括聚光灯），其它几个参数 GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION 为点光源设置光线衰减参数，公式有如下形式，一般无需详细了解：

$$\text{Attenuation coefficient} = \frac{1}{k_c + k_l d + k_q d^2}$$

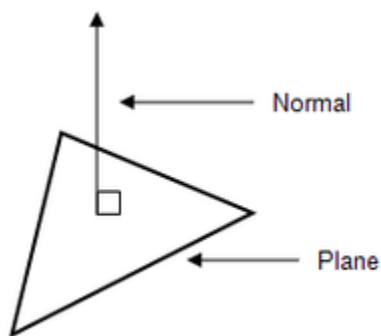
d : Distance between light source and vertex positions

k_c : GL_CONSTANT_ATTENUATION

k_l : GL_LINEAR_ATTENUATION

k_q : GL_QUADRATIC_ATTENUATION

在场景中设置好光源后，下一步要为所绘制的图形设置法线(Normal)，只有设置了法线，光源才能在所绘物体上出现光照效果。三维平面的法线是垂直于该平面的三维向量。曲面在某点 P 处的法线为垂直于该点切平面的向量



和设置颜色类似，有两个方法可以为平面设置法线，一是

```
public void glNormal3f(float nx,float ny,float nz)
```


这个方法为后续所有平面设置同样的方向，直到重新设置新的法线为止。

为某个顶点设置法线：

```
public void glNormalPointer(int type,int stride, Buffer pointer)
```

- type 为 Buffer 的类型，可以为 GL_BYTE, GL_SHORT, GL_FIXED, 或 GL_FLOAT
- stride: 每个参数之间的间隔，通常为 0.
- pointer: 法线值。

打开法线数组

```
1      gl.glEnableClientState(GL10.GL_NORMAL_ARRAY);
```

用法和 Color, Vertex 类似。参见 [Android OpenGL ES 开发教程\(8\): 基本几何图形定义](#)。

规范化法向量，比如使用坐标变换（缩放），如果三个方向缩放比例不同的话，顶点或是平面的法线可能就有变好，此时需要打开规范化法线设置：

```
1      gl.glEnable\(GL10.GL\_NORMALIZE\);
```

经过规范化后法向量为单位向量（长度为 1）。同时可以打开缩放法线设置

```
1      gl.glEnable\(GL10.GL\_RESCALE\_NORMAL\);
```

设置好法线后，需要设置物体表面材料(Material)的反光属性（颜色和材质）。

将在下篇介绍设置物体表面材料(Material)的反光属性（颜色和材质）并给出一个光照的示例。

Android OpenGL ES 开发教程(27)

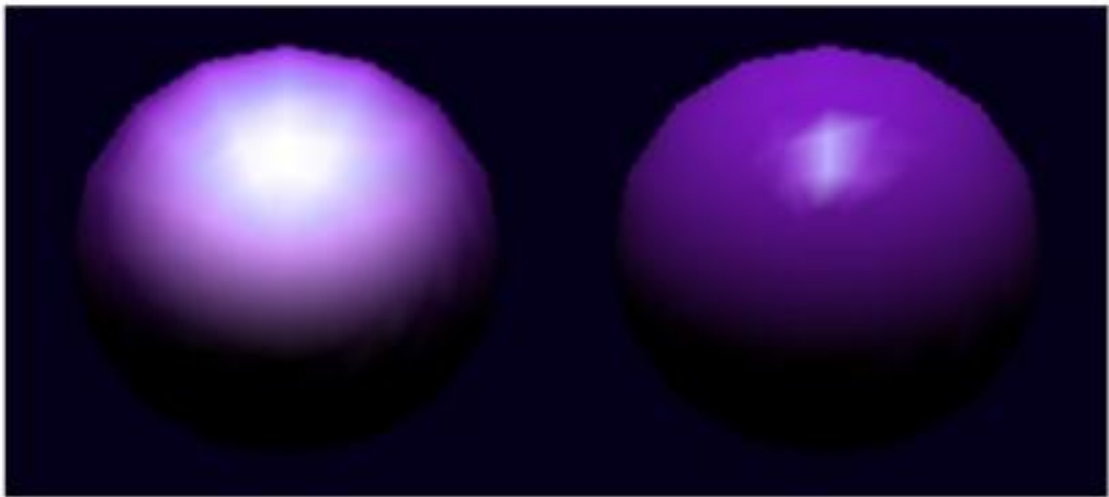
材质及光照示例

设置物体表面材料(Material)的反光属性（颜色和材质）的方法如下：

```
public void glMaterialf(int face,int pname,float param)
public void glMaterialfv(int face,int pname,float[] params,int offset)
public void glMaterialfv(int face,int pname,FloatBuffer params)
```

- face：在 OpenGL ES 中只能使用 GL_FRONT_AND_BACK, 表示修改物体的前面和后面的材质光线属性。
- pname: 参数类型，可以有 GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR, GL_EMISSION, GL_SHININESS。这些参数用在光照方程。
- param: 参数的值。

其中 GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR , GL_EMISSION 为颜色 RGBA 值，GL_SHININESS 值可以从 0 到 128，值越大，光的散射越小：



此外，方法 glLightModel* 给出了光照模型的参数

```
public void glLightModelf(int pname,float param)
public void glLightModelfv(int pname,float[] params,int offset)
public void glLightModelfv(int pname,FloatBuffer params)
```

- pname: 参数类型，可以为 GL_LIGHT_MODEL_AMBIENT 和 GL_LIGHT_MODEL_TWO_SIDE
- params: 参数的值。

最终顶点的颜色由这些参数（光源，材质光学属性，光照模型）综合决定（光照方程计算出）。

下面例子在场景中设置一个白色光源：

```

1      public void initScene(GL10 gl){
2      float[] amb = { 1.0f, 1.0f, 1.0f, 1.0f, };
3      float[] diff = { 1.0f, 1.0f, 1.0f, 1.0f, };
4      float[] spec = { 1.0f, 1.0f, 1.0f, 1.0f, };
5      float[] pos = { 0.0f, 5.0f, 5.0f, 1.0f, };
6      float[] spot_dir = { 0.0f, -1.0f, 0.0f, };
7      gl.glEnable(GL10.GL_DEPTH_TEST);
8      gl.glEnable(GL10.GL_CULL_FACE);
9      gl.glEnable(GL10.GL_LIGHTING);
10     gl.glEnable(GL10.GL_LIGHT0);
11     ByteBuffer abb
12     = ByteBuffer.allocateDirect(amb.length*4);
13     abb.order(ByteOrder.nativeOrder());
14     FloatBuffer ambBuf = abb.asFloatBuffer();
15     ambBuf.put(amb);
16     ambBuf.position(0);
17     ByteBuffer dbb
18     = ByteBuffer.allocateDirect(diff.length*4);
19     dbb.order(ByteOrder.nativeOrder());
20     FloatBuffer diffBuf = dbb.asFloatBuffer();
21     diffBuf.put(diff);
22     diffBuf.position(0);
23     ByteBuffer sbb
24     = ByteBuffer.allocateDirect(spec.length*4);
25     sbb.order(ByteOrder.nativeOrder());
26     FloatBuffer specBuf = sbb.asFloatBuffer();
27     specBuf.put(spec);
28     specBuf.position(0);
29     ByteBuffer pbb
30     = ByteBuffer.allocateDirect(pos.length*4);

```

▪ 31	▪ pbb.order(ByteOrder.nativeOrder());
▪ 32	▪ FloatBuffer posBuf = pbb.asFloatBuffer();
▪ 33	▪ posBuf.put(pos);
▪ 34	▪ posBuf.position(0);
▪ 35	▪ ByteBuffer spbb
▪ 36	▪ =
▪ 37	ByteBuffer.allocateDirect(spot_dir.length*4);
▪ 38	▪ spbb.order(ByteOrder.nativeOrder());
▪ 39	▪ FloatBuffer spot_dirBuf =
▪ 40	spbb.asFloatBuffer();
▪ 41	▪ spot_dirBuf.put(spot_dir);
▪ 42	▪ spot_dirBuf.position(0);
▪ 43	▪ gl.glLightfv(GL10.GL_LIGHT0,
▪ 44	GL10.GL_AMBIENT, ambBuf);
▪ 45	▪ gl.glLightfv(GL10.GL_LIGHT0,
▪ 46	GL10.GL_DIFFUSE, diffBuf);
▪ 47	▪ gl.glLightfv(GL10.GL_LIGHT0,
▪ 48	GL10.GL_SPECULAR, specBuf);
▪ 49	▪ gl.glLightfv(GL10.GL_LIGHT0,
▪ 50	GL10.GL_POSITION, posBuf);
▪ 51	▪ gl.glLightfv(GL10.GL_LIGHT0,
▪ 52	GL10.GL_SPOT_DIRECTION,
▪ 53	spot_dirBuf);
▪ 54	▪ gl.glLightf(GL10.GL_LIGHT0,
▪ 55	GL10.GL_SPOT_EXPONENT, 0.0f);
▪ 56	▪ gl.glLightf(GL10.GL_LIGHT0,
▪ 57	GL10.GL_SPOT_CUTOFF, 45.0f);
▪ 58	▪ gl.glLoadIdentity();
▪ 59	▪ GLU.gluLookAt(gl,0.0f, 4.0f, 4.0f, 0.0f, 0.0f,
▪ 60	0.0f,
	▪ 0.0f, 1.0f, 0.0f);
	▪ }

绘制一个球，并使用蓝色材质：

```
▪ 1      ▪ public void drawScene(GL10 gl) {
▪ 2      ▪ super.drawScene(gl);
▪ 3      ▪ float[] mat_amb = {0.2f * 0.4f, 0.2f * 0.4f,
▪ 4      ▪ 0.2f * 1.0f, 1.0f,};
▪ 5      ▪ float[] mat_diff = {0.4f, 0.4f, 1.0f, 1.0f,};
▪ 6      ▪ float[] mat_spec = {1.0f, 1.0f, 1.0f, 1.0f,};
▪ 7      ▪ ByteBuffer mabb
▪ 8      ▪ =
▪ 9      ▪ ByteBuffer.allocateDirect(mat_amb.length*4);
▪ 10     ▪ mabb.order(ByteOrder.nativeOrder());
▪ 11     ▪ FloatBuffer mat_ambBuf =
▪ 12     ▪ mabb.asFloatBuffer();
▪ 13     ▪ mat_ambBuf.put(mat_amb);
▪ 14     ▪ mat_ambBuf.position(0);
▪ 15     ▪ ByteBuffer mdbb
▪ 16     ▪ =
▪ 17     ▪ ByteBuffer.allocateDirect(mat_diff.length*4);
▪ 18     ▪ mdbb.order(ByteOrder.nativeOrder());
▪ 19     ▪ FloatBuffer mat_diffBuf =
▪ 20     ▪ mdbb.asFloatBuffer();
▪ 21     ▪ mat_diffBuf.put(mat_diff);
▪ 22     ▪ mat_diffBuf.position(0);
▪ 23     ▪ ByteBuffer msbb
▪ 24     ▪ =
▪ 25     ▪ ByteBuffer.allocateDirect(mat_spec.length*4);
▪ 26     ▪ msbb.order(ByteOrder.nativeOrder());
▪ 27     ▪ FloatBuffer mat_specBuf =
▪ 28     ▪ msbb.asFloatBuffer();
▪ 29     ▪ mat_specBuf.put(mat_spec);
▪ 30     ▪ mat_specBuf.position(0);
▪ 31     ▪ gl.glMaterialfv(GL10.GL_FRONT_AND_BAC
▪ 32     ▪ K,
▪ 33     ▪ GL10.GL_AMBIENT, mat_ambBuf);
▪ 34     ▪ gl.glMaterialfv(GL10.GL_FRONT_AND_BAC
▪ 35     ▪ K,
▪ 36     ▪ GL10.GL_DIFFUSE, mat_diffBuf);
▪ 37     ▪ gl.glMaterialfv(GL10.GL_FRONT_AND_BAC
```

```
▪ 38      K,  
▪ 39      ▪ GL10.GL_SPECULAR, mat_specBuf);  
▪ 40      ▪ gl.glMaterialf(GL10.GL_FRONT_AND_BACK,  
▪ 41      ▪ GL10.GL_SHININESS, 64.0f);  
      ▪ sphere.draw(gl);  
      ▪ }
```

