**1 检测设备是否支持OpenGL**

final ActivityManager activityManager=(ActivityManager)getSystemService(ACTIVITY\_SERVICE);

final ConfigurationInfo configurationInfo=activityManager.getDeviceConfigurationInfo();

boolean supportsEs2=configurationInfo.reqGlEsVersion>=0x2000;

boolean supportsEs2 = configurationInfo.reqGlEsVersion >= 0x2000;

boolean isEmulator = Build.VERSION.SDK\_INT > Build.VERSION\_CODES.ICE\_CREAM\_SANDWICH\_MR1

&& (Build.FINGERPRINT.startsWith("generic")

|| Build.FINGERPRINT.startsWith("unknown")

|| Build.MODEL.contains("google\_sdk")

|| Build.MODEL.contains("Emulator")

|| Build.MODEL.contains("Android SDK built for x86"));

supportsEs2 = supportsEs2 || isEmulator;

OpenGL还在绘制图形导致程序崩溃，因此我们还需要根据Activity的生命周期针对GLSurfaceView做一些处理：

@Override

protected void onPause() {

super.onPause();

if (glSurfaceView != null) {

glSurfaceView.onPause();

}

}

@Override

protected void onResume() {

super.onResume();

if (glSurfaceView != null) {

glSurfaceView.onResume();

}

}

Renderer是一个接口，它主要包含3个抽象函数：onSurfaceCreated、onDrawFrame、onSurfaceChanged;从名字就可以看出，分别是在SurfaceView创建时调用、在绘制图形时调用以及在视图大小发生改变时调用。

2.模型数据

我们定义一个三角形，需要9个数，如果我们有float类型表示一个数，那么定义一个三角形（三个点）如下：

private float[] mTriangleArray = {

0f, 1f, 0f,

-1f, -1f, 0f,

1f, -1f, 0f

};

此时，我们就有了一个三角形的3个点数据了。但是，OpenGL并不是对堆里面的数据进行操作，而是在直接内存中（Direct Memory），即操作的数据需要保存到NIO里面的Buffer对象中。而我们上面声明的float[]对象保存在堆中，因此，需要我们将float[]对象转为java.nio.Buffer对象。

我们可以选择在构造函数里面，将float[]对象转为java.nio.Buffer，如下所示：

private FloatBuffer mTriangleBuffer;

public GLRenderer() {

//先初始化buffer，数组的长度\*4，因为一个float占4个字节

ByteBuffer bb = ByteBuffer.allocateDirect(mTriangleArray.length \* 4);

//以本机字节顺序来修改此缓冲区的字节顺序

bb.order(ByteOrder.nativeOrder());

mTriangleBuffer = bb.asFloatBuffer();

//将给定float[]数据从当前位置开始，依次写入此缓冲区

mTriangleBuffer.put(mTriangleArray);

//设置此缓冲区的位置。如果标记已定义并且大于新的位置，则要丢弃该标记。

mTriangleBuffer.position(0);

}

# 3矩阵变换

## 模型变换和视图变换

而模型视图变换在OpenGL中对应标识为：GL10.GL\_MODELVIEW。通过glMatrixMode函数来声明：

gl.glMatrixMode(GL10.GL\_MODELVIEW);

gl.glLoadIdentity();

* 1

此时，当前变换矩阵为单位矩阵，后面才可以继续做变换，例如：

//绕（1,0,0）向量旋转30度

gl.glRotatef(30, 1, 0, 0);

//沿x轴方向移动1个单位

gl.glTranslatef(1, 0, 0);

//x，y，z方向放缩0.1倍

gl.glScalef(0.1f, 0.1f, 0.1f);

如果我们不想改变物体，而是改变观察点，可以使用如下函数

/\*\*

\* gl: GL10型变量

\* eyeX,eyeY,eyeZ: 观测点坐标（相机坐标）

\* centerX,centerY,centerZ：观察位置的坐标

\* upX,upY,upZ ：相机向上方向在世界坐标系中的方向（即保证看到的物体跟期望的不会颠倒）

\*/

GLU.gluLookAt(gl,eyeX,eyeY,eyeZ,centerX,centerY,centerZ,upX,upY,upZ);

## 投影变换

OpenGL支持主要两种投影变换：

* 透视投影
* 正投影

gl.glMatrixMode(GL10.GL\_PROJECTION);

gl.glLoadIdentity();

通过如下函数可将当前可视空间设置为透视投影空间：

gl.glFrustumf(left,right,bottom,top,near,far);

当然了，也可以通过另一个函数实现相同的效果：

GLU.gluPerspective(gl,fovy,aspect,near,far);

可以通过如下函数设置正投影：

gl.glOrthof(left,right,bottom,top,near,far);

## 视口变换

一般默认是在整个窗口中绘制，但是，如果你不希望在整个窗口中绘制，而是在窗口的某个小区域中绘制，你也可以自己定制：

@Override

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

gl.glViewport(0, 0, width, height);

}

每次窗口发生变化时，我们可以设置绘制区域，即在onSurfaceChanged函数中调用glViewport函数。

# 4 启用相关功能及配置

### 4.1 glClearColor()

设置清屏颜色，每次清屏时，使用该颜色填充整个屏幕。使用例子：

gl.glClearColor(1.0f, 1.0f, 1.0f, 0f);

里面参数分别代表RGBA，取值范围为[0,1]而不是[0,255]

### 4.2 glDepthFunc()

OpenGL中物体模型的每个像素都有一个深度缓存的值（在0到1之间，可以看成是距离）,可以通过glClearDepthf函数设置默认的“当前像素”z值。在绘制时，通过将待绘制的模型像素点的深度值与“当前像素”z值进行比较，将符合条件的像素绘制出来，不符合条件的不绘制。具体的“指定条件”可取以下值:

* GL10.GL\_NEVER：永不绘制
* GL10.GL\_LESS：只绘制模型中像素点的z值<当前像素z值的部分
* GL10.GL\_EQUAL：只绘制模型中像素点的z值=当前像素z值的部分
* GL10.GL\_LEQUAL：只绘制模型中像素点的z值<=当前像素z值的部分
* GL10.GL\_GREATER ：只绘制模型中像素点的z值>当前像素z值的部分
* GL10.GL\_NOTEQUAL：只绘制模型中像素点的z值!=当前像素z值的部分
* GL10.GL\_GEQUAL：只绘制模型中像素点的z值>=当前像素z值的部分
* GL10.GL\_ALWAYS：总是绘制

通过目标像素与当前像素在z方向上值大小的比较是否满足参数指定的条件，来决定在深度（z方向）上是否绘制该目标像素。

注意， 该函数只有启用“深度测试”时才有效，通过glEnable(GL\_DEPTH\_TEST)开启深度测试以及glDisable(GL\_DEPTH\_TEST)关闭深度测试。

例子：

gl.glDepthFunc(GL10.GL\_LEQUAL);

### 

### 4.3 glClearDepthf()

给深度缓存设定默认值。

缓存中的每个像素的深度值默认都是这个， 假设在 gl.glDepthFunc(GL10.GL\_LEQUAL);前提下：

* 如果指定“当前像素值”为1时，我们知道，一个模型深度值取值和范围为[0,1]。这个时候你往里面画一个物体， 由于物体的每个像素的深度值都小于等于1， 所以整个物体都被显示了出来。
* 如果指定“当前像素值”为0， 物体的每个像素的深度值都大于等于0， 所以整个物体都不可见。   
  如果指定“当前像素值”为0.5， 那么物体就只有深度小于等于0.5的那部分才是可见的

使用例子：

gl.glClearDepthf(1.0f);

### 

### 4.4 glEnable()，glDisable()

glEnable()启用相关功能，glDisable()关闭相关功能。

比如：

//启用深度测试

gl.glEnable(GL10.GL\_DEPTH\_TEST);

//关闭深度测试

gl.glDisable(GL10.GL\_DEPTH\_TEST)

//开启灯照效果

gl.glEnable(GL10.GL\_LIGHTING);

// 启用光源0

gl.glEnable(GL10.GL\_LIGHT0);

// 启用颜色追踪

gl.glEnable(GL10.GL\_COLOR\_MATERIAL);

4.5 glHint()

如果OpenGL在某些地方不能有效执行是，给他指定其他操作。

函数原型为：

void glHint(GLenum target,GLenum mod)

* 1

其中，target：指定所控制行为的符号常量，可以是以下值（引自[【OpenGL函数思考-glHint 】](http://blog.csdn.net/shuaihj/article/details/7230867)）：

* GL\_FOG\_HINT：指定雾化计算的精度。如果OpenGL实现不能有效的支持每个像素的雾化计算，则GL\_DONT\_CARE和GL\_FASTEST雾化效果中每个定点的计算。
* GL\_LINE\_SMOOTH\_HINT：指定反走样线段的采样质量。如果应用较大的滤波函数，GL\_NICEST在光栅化期间可以生成更多的像素段。
* GL\_PERSPECTIVE\_CORRECTION\_HINT：指定颜色和纹理坐标的差值质量。如果OpenGL不能有效的支持透视修正参数差值，那么GL\_DONT\_CARE 和 GL\_FASTEST可以执行颜色、纹理坐标的简单线性差值计算。
* GL\_POINT\_SMOOTH\_HINT：指定反走样点的采样质量，如果应用较大的滤波函数，GL\_NICEST在光栅化期间可以生成更多的像素段。
* GL\_POLYGON\_SMOOTH\_HINT：指定反走样多边形的采样质量，如果应用较大的滤波函数，GL\_NICEST在光栅化期间可以生成更多的像素段。

mod：指定所采取行为的符号常量，可以是以下值:

* GL\_FASTEST：选择速度最快选项。
* GL\_NICEST：选择最高质量选项。
* GL\_DONT\_CARE：对选项不做考虑。

例子：

gl.glHint(GL10.GL\_PERSPECTIVE\_CORRECTION\_HINT, GL10.GL\_NICEST);

### 4.6 glEnableClientState()

当我们需要启用顶点数组（保存每个顶点的坐标数据）、顶点颜色数组（保存每个顶点的颜色）等等，就要通过glEnableClientState()函数来开启：

//以下两步为绘制颜色与顶点前必做操作

// 允许设置顶点

//GL10.GL\_VERTEX\_ARRAY顶点数组

gl.glEnableClientState(GL10.GL\_VERTEX\_ARRAY);

// 允许设置颜色

//GL10.GL\_COLOR\_ARRAY颜色数组

gl.glEnableClientState(GL10.GL\_COLOR\_ARRAY);

### 

### 4.7 glShadeModel()

设置着色器模式，有如下两个选择：

* GL10.GL\_FLAT
* GL10.GL\_SMOOTH（默认）

如果为每个顶点指定了顶点的颜色，此时：

* GL\_SMOOTH：根据顶点的不同颜色，最终以渐变的形式填充图形。
* GL\_FLAT：假设有n个三角片，则取最后n个顶点的颜色填充着n个三角片。

使用例子：

gl.glShadeModel(GL10.GL\_SMOOTH);

# 5 几个重要的函数

### 5.1 glVertexPointer()

其实就是设置一个指针，这个指针指向顶点数组，后面绘制三角形（或矩形）根据这里指定的顶点数组来读取数据。   
函数原型如下：

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

其中：

* size: 每个顶点有几个数值描述。必须是2，3 ，4 之一。
* type: 数组中每个顶点的坐标类型。取值：GL\_BYTE,GL\_SHORT, GL\_FIXED, GL\_FLOAT。
* stride：数组中每个顶点间的间隔，步长（字节位移）。取值若为0，表示数组是连续的
* pointer：即存储顶点的Buffer

### 5.2 glColorPointer()

跟上面类似，只是设定指向颜色数组的指针。   
函数原型：

void glColorPointer(

int size,

int type,

int stride,

java.nio.Buffer pointer

);

* size: 每种颜色组件的数量。 值必须为 3 或 4。
* type: 颜色数组中的每个颜色分量的数据类型。 使用下列常量指定可接受的数据类型：GL\_BYTEGL\_UNSIGNED\_BYTE，GL\_SHORT GL\_UNSIGNED\_SHORT，GL\_INT GL\_UNSIGNED\_INT，GL\_FLOAT，或GL\_DOUBLE。
* stride：连续颜色之间的字节偏移量。 当偏移量为0时，表示数据是连续的。
* pointer：即颜色的Buffer

### 5.3 glDrawArrays()

绘制数组里面所有点构成的各个三角片。

函数原型:

void glDrawArrays(

int mode,

int first,

int count

);

mode：有三种取值 

* + GL\_TRIANGLES：每三个顶之间绘制三角形，之间不连接
  + GL\_TRIANGLE\_FAN：以V0 V1 V2,V0 V2 V3,V0 V3 V4，……的形式绘制三角形
  + GL\_TRIANGLE\_STRIP：顺序在每三个顶点之间均绘制三角形。这个方法可以保证从相同的方向上所有三角形均被绘制。以V0 V1 V2 ,V1 V2 V3,V2 V3 V4,……的形式绘制三角形
* first：从数组缓存中的哪一位开始绘制，一般都定义为0
* count：顶点的数量

# 光照效果

因为我们所做的立体效果是根据真实世界原理来计算的，所以很有必要去了解在现实世界中，我们所看到的一个物体有哪些光。

## 1.1 真实世界中的光照

我们知道，在黑暗中，当我们将手电筒对准某个物体时，我们所看到的该物体的“亮度”有3种：

* 物体表面发生镜面反射部分（Specular），一般是白色。
* 物体表面发生漫反射部分（Diffuse），一般是物体表面的颜色。
* 物体表面没有照射到光的部分，即通过环境光（Ambient）照射，在黑暗中环境光是黑色。

## 1.2 Android OpenGL相关API

### 1.2.1 光源 GL10.GL\_LIGHT0

0号光源，该光源的默认颜色为白色，即RGBA为（1.0,1.0,1.0,1.0），漫反射和镜面反射也为白色。类似的，还有其他光源如GL10.GL\_LIGHT1，系统提供了0~7共8种光源，其他的光源默认为黑色，即RGBA为（0.0,0.0,0.0,1.0）.

开启光源也非常简单：

//启用光照功能

gl.glEnable(GL10.GL\_LIGHTING);

//开启0号灯

gl.glEnable(GL10.GL\_LIGHT0);

### 1.2.2 设置各种反射光颜色

一旦开启了光照功能，就可以通过glLightfv函数来指定各种反射光的颜色了，glLightfv函数如下：

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)

其中，

* 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(用于设置环境光颜色)
  + GL\_DIFFUSE(用于设置漫反射光颜色)
  + GL\_SPECULAR(用于设置镜面反射光颜色)
  + GL\_SPOT\_DIRECTION
  + GL\_POSITION（用于设置光源位置）
* params: 参数的值（数组或是Buffer类型），数组里面含有4个值分别表示R,G,B,A。

指定光源的位置的参数为GL\_POSITION,位置的值为(x,y,z,w)，如果是平行光则将w 设为0，此时，(x,y,z)为平行光的方向：

# 材料属性

材料的属性设置和光源的设置有些类似，用到的函数

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)

其中，

* 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，这些参数用在光照方程。可以取如下值：   
  + GL\_AMBIENT
  + GL\_DIFFUSE
  + GL\_SPECULAR
  + GL\_EMISSION
  + GL\_SHININESS。
* param：指定反射的颜色。

跟设置光照类似，设置材料属性首先需要定义各种反射光的颜色：

float[] materialAmb = {0.4f, 0.4f, 1.0f, 1.0f};

float[] materialDiff = {0.0f, 0.0f, 1.0f, 1.0f};//漫反射设置蓝色

float[] materialSpec = {1.0f, 0.5f, 0.0f, 1.0f};

然后就是将这些颜色通过glMaterialfv函数设置进去：

public void enableMaterial(GL10 gl) {

//材料对环境光的反射情况

gl.glMaterialfv(GL10.GL\_FRONT\_AND\_BACK, GL10.GL\_AMBIENT, Util.floatToBuffer(materialAmb));

//散射光的反射情况

gl.glMaterialfv(GL10.GL\_FRONT\_AND\_BACK, GL10.GL\_DIFFUSE, Util.floatToBuffer(materialDiff));

//镜面光的反射情况

gl.glMaterialfv(GL10.GL\_FRONT\_AND\_BACK, GL10.GL\_SPECULAR, Util.floatToBuffer(materialSpec));

}