eoe 特刊 第27期

eoe android 优亿开发

目录

前言

经过一个月征稿、编辑，新的一版特刊终于出炉了。

本次特刊的制作，比以往都更大胆。在只有一个主题的前提下，完全是通过社区的热心的网友，根据自己的想法，自行设计，自由发挥，来完成各自的作品。特刊中保留了网友作品的原汁原味。

在内容上，做到对知识的更加统一，项目的分析，以及对特别的类进行了说明，并涵盖了更完善的具有代表性的项目，可以作为我们的参考内容。

在这里要对参与本次特刊制作的网友（Sinoace，mfcai，szaren，1140112229）表示衷心感谢。别外对一些由于时间问题不能参与到特刊制作的网友，表示感谢。

特刊内容中如有不当之处,欢迎各位开发者纠正。如有疑问请发帖至特刊专区进行反馈:
http://www.eoeandroid.com/forum-39-1.html 。

eoeandroid 做最棒的 android 开发者社区
http://www.eoeandroid.com

Jiayouhe123
2012.12.15

作者简介

1140112229

论坛 ID：1140112229. 网名：我喜欢，1140112229 也是“我喜欢”的学号，来自广州的某个软件学院精英班的学生。正在读大二。身高不高，体重不重，通过照片就可以看出，喜欢折腾各种 IT 技术。因为兴趣的原因，接触 Android 差不多一年，都是自学的，喜欢研究源码，感觉特别有意思。经常在 eoeAndroid 论坛上潜水，喜欢帮忙解决问题，因为那样可以得到小小的成就感，而且在解决问题的同时也可以学到很多新的知识。“我喜欢”有个小目标：在以后的日子里，我会努力加强自己的技术，希望能够帮助到更多的学习 android 的朋友。同时他的座右铭：在成长中学习，在学习中成长。加油！让我们一起祝福他!

szaren

论坛 ID：n0tify，来自北边的人，也是一位在校学生，目前在一家软件公司做 Android 的应用开发和游戏开发，大学里学习的东西很杂，大部分都是自学的。经常混迹于 eoe 论坛的各大板块，但是发表的帖子不多，喜欢在一旁静静的观看，在同龄人中接触 Android 较早，在 09 年的时候 Android 还没有像现在这么普及，但是随之电信等运营商在高校的活动，使得这个小机器人红遍了大江南北，也是在那个时候开始学习 android 开发，在学习的过程中从 eoe 社区中获取到很多知识，尤其是喜欢一个类似 wiki 的板块，遇到不明白的问题都能找到答案。n0tify 马上就要毕业了，希望能沿着这条路走下去，同时 n0tify 也感谢曾经帮助过他，指点过他的人，愿大家一切顺利。也祝愿 n0tify 工作顺利。

mfcai

论坛 ID：mfcai，目前从事手机支付类应用开发。对 android 应用程序架构开发比较熟练。但从事 J2EE 企业级应用开发时间比 android 应用开发时间要长得多。网名：流星，人如其名，当制作特刊找到 mfcai 时，mfcai 欣然的接受，并很快的提交了任务。以前 mfcai 一直对技术比较热爱，最近才懂一些管理：带领一群人干比自己单干要有意思的多，现在 mfcai 正带领一个团队做项目。他坚信着：有更多的人，才能做更多的事。

Sinoace

论坛 ID：sinoace，sinoace 才 24 岁，就已经是 team leader，工作已经两年半，山东大汉，原本在济南工作，主要从事 java 和 Android，因为朋友在北京，所以推荐 sinoace 来北京发展，毕竟北京 Android 开发的公司比较多，发展较好。平时，在论坛一直是“潜水运动员“，在主要贡献”MQTT 的学习研究 【汇总贴】”的时候发现了 sinoace，并邀请他一起制作特刊，欣然接受。感谢 sinoace 能在百忙中制作特刊，祝愿 sinoace 的团队更好的发展。

一、OpenGLES 的介绍

1.1 简介

OpenGL ES （OpenGL for Embedded Systems）是 OpenGL 三维图形 API 的子集，针对手机、PDA 和游戏主机等嵌入式设备而设计。该 API 由 Khronos 集团定义推广，Khronos 是一个图形软硬件行业协会，该协会主要关注图形和多媒体方面的开放标准。

OpenGL ES 是从 OpenGL 裁剪定制而来的，去除了 glBegin/glEnd，四边形（GL_QUADS）、多边形（GL_POLYGONS）等复杂图元等许多非绝对必要的特性。经过多年发展，现在主要有两个版本，OpenGL ES 1.x 针对固定管线硬件的，OpenGL ES 2.x 针对可编程管线硬件。OpenGL ES 1.0 是以 OpenGL 1.3 规范为基础的，OpenGL ES 1.1 是以 OpenGL 1.5 规范为基础的，它们分别又支持 common 和 common lite 两种 profile。lite profile 只支持定点定点实数，而 common profile 既支持定点数又支持浮点数。 OpenGL ES 2.0 则是参照 OpenGL 2.0 规范定义的，common profile 发布于 2005-8，引入了对可编程管线的支持。OpenGL ES 还有一个 safety-critical profile。

1.2 基本操作

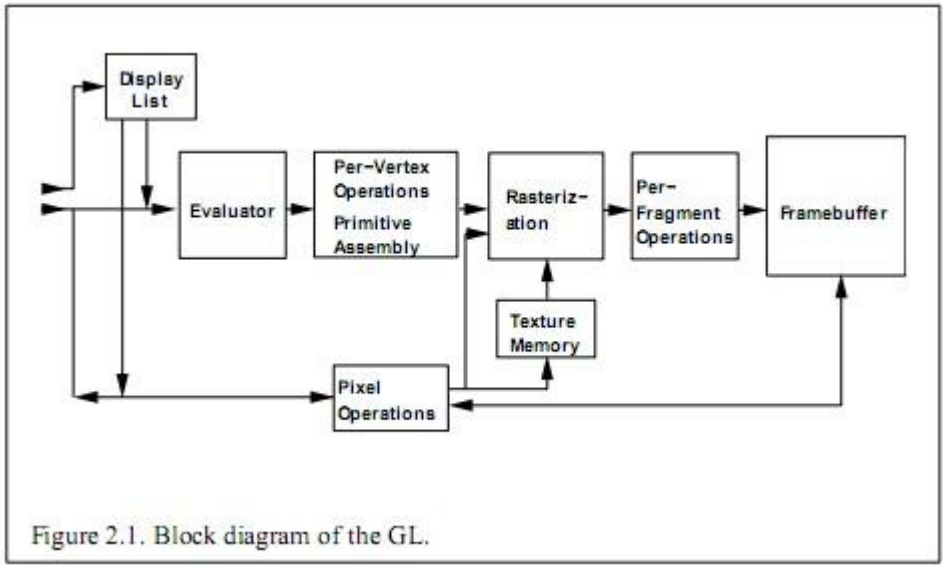


Figure 2.1. Block diagram of the GL.

openGL 的原理图如上图所示。命令从左边输入。一些命令用来绘制几何物体，而其它命令则控制不同阶段如何对物体进行处理。大多数指令可以在一个 DisplayList 中累积，以供 GL 稍后对其处理。否则,命令将被送入一个处理管道。

第一阶段通过计算输入值的多项式函数，提供了一种估计曲线和表面几何特征的有效方法。

第二阶段根据点、线和多边形三类基本类型的图元的顶点属性进行操作。在这个阶段，

对顶点经过了形变和光照处理,原始点被处理成许多卷积，为下一阶段——光栅化做准备。光栅化生产一系列的帧缓冲区。

产生的每个片断被送入下一阶段进行处理。此阶段中，每个片断在被绘制到帧缓冲区之前先被单独操作处理。这些操作包括根据送入的和以前存储的深度值（影响到深度缓冲）对帧缓冲进行更新，用存储的颜色渲染送入的片断，对片断值进行掩码以及其它逻辑运算的操作。

最后,有一个办法绕过管道的顶点处理部分。将一组片断直接进行单独片断处理，最终将一组像素写入到帧缓冲区中;值也可能从缓冲区中读取或者从缓冲区的一部分复制到另一部分。这些转移可能会包括一些数值类型的解码和编码。

以上顺序仅仅是作为描述 GL 的工具，并不是对 GL 如何工作的严格描述，因为我们仅仅是用它来呈现 GL 的多种操作。

OpenGL ES 是从 OpenGL 裁剪定制而来的，去除了 glBegin/glEnd，四边形（GL_QUADS）、多边形（GL_POLYGONS）等复杂图元等许多非绝对必要的特性。经过多年发展，现在主要有两个版本，OpenGL ES 1.x 针对固定管线硬件的，OpenGL ES 2.x 针对可编程管线硬件。OpenGL ES 1.0 是以 OpenGL 1.3 规范为基础的，OpenGL ES 1.1 是以 OpenGL 1.5 规范为基础的，它们分别又支持 common 和 common lite 两种 profile。lite profile 只支持定点定点实数，而 common profile 既支持定点数又支持浮点数。 OpenGL ES 2.0 则是参照 OpenGL 2.0 规范定义的，common profile 发布于 2005-8，引入了对可编程管线的支持。

1.3 不能使用以下这些 api：

1. glBegin/glEnd
2. glArrayElement
3. 显示列表
4. 求值器
5. 索引色模式
6. 自定义裁剪平面
7. glRect
8. 图像处理(这个一般显卡也没有，FireGL/Quadro 显卡有)
9. 反馈缓冲
10. 选择缓冲
11. 累积缓冲
12. 边界标志
13. glPolygonMode
14. GL_QUADS、GL_QUAD_STRIP、GL_POLYGON
15. glPushAttrib、PopAttrib、glPushClientAttrib、glPopClientAttrib
16. TEXTURE_1D、TEXTURE_3D、TEXTURE_RECT、TEXTURE_CUBE_MAP
17. GL_COMBINE
18. 自动纹理坐标生成
19. 纹理边界
20. GL_CLAMP、GL_CLAMP_TO_BORDER
21. 消失纹理代表

22. 纹理 LOD 限定
23. 纹理偏好限定
24. 纹理自动压缩、解压缩
25. glDrawPixels、glPixelTransfer、glPixelZoom
26. glReadBuffer、glDrawBuffer、glCopyPixels

1.4 常用的 api：

glClearColor(0.f, 0.f, 0.f, 1.f); // 设置模式窗口的背景颜色，颜色采用的是 RGBA 值
glViewport(0, 0, iScreenWidth, iScreenHeight);//设置视口的大小以及位置，视口：也就是图形最终显示到屏幕的区域，前两个参数是视口的位置，后两个参数是视口的宽和长。
glMatrixMode(GL_PROJECTION); // 设置矩阵模式为投影矩阵，之后的变换将影响投影矩阵。
OpenGL 属于状态管理机制，比如：设置当前矩阵为投影矩阵过后，在没有重新调用 glMatrixMode()之前，任何矩阵变换都将影响投影矩阵。

glFrustumf(-1.f, 1.f, -1.f, 1.f, 3.f, 1000.f); //该函数创建一个透视投影矩阵，其中的参数定义了视景体，可以理解为用相机的时候，眼睛的可视范围。就像一个三棱锥，参数 1、3、5 和 2、4、6 分别定义了近裁面和远裁面的左下和右上的（x、y、z）坐标。

OpenGL 投影有两种模式，一种是透视投影，也就是通过上述函数创建一个三棱锥视景体，这种模式下观看三维模型是近大远小。另外一种模式是正交模式，视景体是一个平行六面体，离相机的距离不会影响物体的大小。

glMatrixMode(GL_MODELVIEW); //设置当前矩阵为模式矩阵
glVertexpointer(3, GL_BYTE, 0, vertices); //指定从哪里存取空间坐标数据
glShadeModel(GL_FLAT); //设置阴影模式为 GL_FLAT，默认是 GL_SMOOTH
阴影模式一共有两种，GL_SMOOTH 和 GL_FLAT，在有关照的情况下会有不同的效果。

glClear(GL_COLOR_BUFFER_BIT); //清除颜色缓存
glLoadIdentity(); //设置当前矩阵为单位矩阵

OpenGL 里面的位置大小都是用矩阵来表示的，比如：glScanf()放大或缩小，其实就是用一个矩阵去乘当前的矩阵，为了使变换不受当前矩阵的影响，所以把当前矩阵设置为单位矩阵。

glTranslatex(0, 0, -100 << 16); //将坐标向 z 轴负方向移动 100
glColor4f(1.f, 0.f, 0.f, 1.f); 设置颜色为红色
glScalex(15 << 16, 15 << 16, 15 << 16); // 将物体沿 xyz 者分别放大 15 倍
glDrawElements(GL_TRIANGLES, 1 * 3, GL_UNSIGNED_BYTE, indices);
//绘制图形，GL_TRIANGLES 说明要绘制的图形是三角形，3 表示一共有三个定点，**GL_UNSIGNED_BYTE 表示 indices 存储的数据类型**

void glTranslatef(GLfloat x, GLfloat y, GLfloat z)
void glTranslatex(GLfixed x, GLfixed y, GLfixed z)
功能：沿 x、y、z 平移
void glScalef(GLfloat x, GLfloat y, GLfloat z)void glScalex(GLfixed x, GLfixed y, GLfixed z)
功能：在 x、y、z 轴进行缩放，参数 x、y、z 为你想要的大小。
void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)void glRotatex(GLfixed angle, GLfixed x, GLfixed y, GLfixed z)
功能：沿 x、y、z 轴进行旋转。Angle 表示将要旋转的角度。
void glClear(GLbitfield mask)
功能：用 mask 清除缓存可以有三种模式进行清除： GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, and GL_STENCIL_BUFFER_BIT.
void glClearDepthf(GLclampf depth)void glClearDepthx(GLclampx depth)
功能：设置深度缓存，参数为 0 到 1，使用 glClear 清除缓存。3D 场景 OpenGL 程序都使用深度缓存。它的排序决定那个物体先画。这样您就不会将一个圆形后面的正方形画到圆形上来。
void glClearColor(GLclampf red, GLclampf green, GLclampf blue, GLclampf alpha) void glClearColorx(GLclampx red, GLclampx green, GLclampx blue,GLclampx alpha)
功能：用指定的颜色值（RGBA）清除颜色缓存
void glColorPointer(GLint size, GLenum type, GLsizei stride,const GLvoid＊pointer) 功能：指定颜色的存储空间，size 在 OpenGL ES 默认为 4，表示（RGBA）；type 为 pointer 内容的类型；stride 为数据在 pointer 内存中的偏移量；pointer 为第一个元素的地址。
void glEnableClientState(GLenum array)void glDisableClientState(GLenum array)
功能：启用或者禁止 array，array 有：GL_COLOR_ARRAY, GL_MATRIX_INDEX_ARRAY_OES, GL_NORMAL_ARRAY, GL_POINT_SIZE_ARRAY_ARRAY_OES, GL_TEXTURE_COORD_ARRAY, GL_VERTEX_ARRAY, and GL_WEIGHT_ARRAY_OES。
void glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid * indices)
功能：按照参数给定的值绘制图形，mode 指定要绘制的类型：GL_POINTS, GL_LINE_STRIP, GL_LINE_LOOP, GL_LINES, GL_TRIANGLE_STRIP, GL_TRIANGLE_FAN, and GL_TRIANGLES；count 指定要绘制多少个；type 指定 indices 为数组首地址

二、Android Opengl ES 分析与实例

By mfcai

在 android 平台下用 opengGL 开发 3D 图形的过程进行详细的分析，并提供一个实例供大家参考

2.1 OpenGL ES 简介

OpenGL 是个专业的 3D 程序接口，是一个功能强大，调用方便的底层 3D 图形库。Android 3D 引擎采用的是 OpenGL ES.

OpenGL ES 与 OpenGL 的区别：

OpenGL ES 是专为内嵌和移动设备设计的一个 2D/3D 轻量级图形库，它基于 OpenGL API 设计，是 OpenGL 三维图形 API 的子集。android 3D 图形系统也分为 java 框架和本地代码两部分。本地代码主要实现的 OpenGL 接口的库，在 Java 框架层，javax.microedition.khronos.opengles 是 java 标准的 OpenGL 包，android.opengl 包提供了 OpenGL 系统和 Android GUI 系统之间的联系。

Android 支持 OpenGL 列表

1)GL

2)GL 10

3)GL 10 EXT

4)GL 11

5)GL 11 EXT

6)GL 11 ExtensionPack

Android 里有三个与 OpenGL 有关的包：

android.opengl

javax.microedition.khronos.egl

javax.microedition.khronos.opengles

在 Android 里有两个基类可以用来通过 OpenGL ES API 来创建和操作图形：GLSurfaceView 和 GLSurfaceView.Renderer。如果你想在 Android 应用中使用 OpenGL,你的首要目标便是学会如何在 activity 里实现这些类。

2.2 相关类

1）GLSurfaceView

这个类是一个视图，在此你可以通过 OpenGL API 的调用来绘制和操作对象，就如同 SurfaceView 一样。你可通过实例化该类并添加你自己的 Renderer 来使用它。

但是如果你想要捕捉触屏事件，你需要继承 GLSurfaceView 类来实现触摸监听器。

2）GLSurfaceView.Renderer

这个接口定义了在 OpenGL GLSurfaceView 上绘制图形的方法。你必须使用一个独立的类来实现该接口，并通过 GLSurfaceView.setRenderer()来将该类添加到你的 GLSurfaceView 实例上去。

GLSurfaceView.Renderer 需要你实现如下的函数：

onSurfaceCreated()：GLSurfaceView 创建时会调用一次该方法。使用该方法来实现只发生一次的动作，比如设置 OpenGL 的环境变量以及实例化 OpenGL 的图形对象。

onDrawFrame()：系统每次重绘 GLSurfaceView 时都会调用该方法。这个方法是每次绘制或重绘对象时首先被执行的函数。

onSurfaceChanged()：当 GLSurfaceView 的几何形状(包括图形大小以及设备方向)发生改变时调用，用这个函数来响应 GLSurfaceView 容器的改变。

2.3 Opengl es 开发步骤

Opengl es 开发主要分为三个步骤：建模、渲染、逻辑控制。

建模，指通过一些基本图元如点、线、三角形、多边形将物体画出来。具体来说，建模涉及模型的构建、贴纹理、制作动画等。

渲染，即使用 OpenGL 图形接口将模型在计算机上画出来。

逻辑控制，若要模型动起来，需要根据时间计算模型各个顶点的坐标，这个通过程序逻辑来控制。

2.3.1 建模

3D 模型由一些更小的元素（点，线，多边形）组成

2.3.1.1 顶点

在 Android 系统中可以使用一个浮点数数组来定义一个顶点，浮点数数组通常放在一个 Buffer（java.nio)中来提高性能

有了顶点的定义，下面一步就是如何将它们传给 OpenGL ES 库，OpenGL ES 提供一个成为”管道 Pipeline”的机制，这个管道定义了一些“开关”来控制 OpenGL ES 支持的某些功能，缺省情况这些功能是关闭的，如果需要使用 OpenGL ES 的这些功能，需要明确告知 OpenGL “管道”打开所需功能要注意的使用完某个功能之后，要关闭这个功能以免影响后续操作顶点数组实际上是多个数组，顶点坐标、纹理坐标、法线向量、顶点颜色等等，顶点的每一个属性都可以指定一个数组，然后用统一的序号来进行访问。

比如序号 3，就表示取得颜色数组的第 3 个元素作为颜色、取得纹理坐标数组的第 3 个元素作为纹理坐标、取得法线向量数组的第 3 个元素作为法线向量、取得顶点坐标数组的第 3 个元素作为顶点坐标。把所有的数据综合起来，最终得到一个顶点。

可以用 glEnableClientState/glDisableClientState 单独的开启和关闭每一种数组。

```
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glEnableClientState(GL_NORMAL_ARRAY);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
glEnableClientState(GL_INDEX_ARRAY);
glEnableClientState(GL_EDGE_FLAG_ARRAY);
```

GL_COLOR_ARRAY--启用一个存储每个顶点颜色信息的数组

GL_EDGE_FLAG_ARRAY--启动一个存储每个顶点边标识的数组
GL_INDEX_ARRAY--启动一个存储每个顶点调色板索引的数组
GL_NORMAL_ARRAY--启动一个存储每个顶点法线的数组
GL_TEXTURE_COORD_ARRAY--启动一个存储每个顶点纹理坐标的数组
GL_VERTEX_ARRAY--启动一个存储每个顶点的位置的数组

用以下的函数来指定数组的位置：

glColorPointer 函数
glEdgeFlagPointer 函数，边标识。特别是线框模式下。
glIndexPointer 函数，调色板显示模式下存储颜色索引。
glNormalPointer 函数，用于存储每一个顶点的法向量。
glTexCoordPointer 函数，存储每一个顶点的纹理坐标。
glVertexPointer 函数，存储每个顶点的位置数据。

2.3.1.2 多变形（Polygon）

四边行的四个顶点，次序也在数组 vertices 中排列好了，即第一个顶点的序号为 0({-1.0f, 1.0f, 0.0f}),第二个顶点序号为 1({-1.0f, -1.0f, 0.0f})等。

现在说如何画这个四边型，前面已经设定 Face 的旋转为逆时针，数组 vertices 组成的是一个四边型，四边型由两个 Face（也就是三角形）组成，按照逆时针画两个三角形，即组成一个四边行，第一个三角形的顶点顺序为 0、1、2，即 vertices 中的前三个顶点组成的三角形， 0、2、3 三个顶点按照逆时针的顺序组成第二个三角形。这个过程组成了一个顺序数组如下：

```
private short[] indices = { 0, 1, 2, 0, 2, 3 };
```

为了提高性能，将数组转换为位数组，代码如下：

```
// short is 2 bytes, therefore we multiply the number if vertices with 2.
ByteBuffer ibb = ByteBuffer.allocateDirect(indices.length * 2);
ibb.order(ByteOrder.nativeOrder());
ShortBuffer indexBuffer = bb.asShortBuffer();
indexBuffer.put(indices);
indexBuffer.position(0);
```

2.3.1.3 面（Face)

在 opengl 中，每个物体都由正面和背面组成设置 Face 的旋转方向是很重要的，因为选择不同的旋转方向意味这选择那一面为前面，那一面为后面。

因为在一个由完全不透明的表面组成的场景中，多边形的后面部分是绝不可能看到的，有时候为了提升性能，需要只描绘正面，而后面的图形就省略部描绘，可以通过为函数 glEnable(GL_CULL_FACE) 和 glDisable(GL_CULL_FACE)来决定是否省略描绘后面部分的图形。可以通过函数 glFrontFace 来改变方向从而决定多边形的那一面为正面。

glFrontFace： 常量参数 GL_CCW 表示逆时针，GL_CW 表示顺时针。默认情况下，逆时针的多边形坐标顺序是多边形是正面。例如对于三角形：

```
vertices[] ={
```

```
0, 1, 0,
-1, -1, 0,
1, -1, 0
}
```

如果添加 glEnable(GL_CULL_FACE)功能，逆时针可以正常显示图形，顺时针无法显示图形。

glCullFace： 函数决定绘制多边形的背面还是正面。

2.3.1.4 纹理

在 android 平台上，纹理图的长宽必须是 2 的幂，纹理图尺寸超过 512x512 时 fps 会显著降低。一般的做法是将所有纹理组合到成一张 256x256 或者 512x512 的大图。通过 glMatrixMode (glTexture) 对纹理坐标进行操作，代码如下：

```
glMatrixMode(GL_TEXTURE);
glPushMatrix();
glLoadIdentity();

// move the texture - this does not work
glTranslatef(offset, 0.0f, 0.0f);
glTranslatef(0.0f, offset, 0.0f);
glTranslatef(0.0f, 0.0f, offset);

glPopMatrix();
```

注意：

纹理坐标的偏移值范围为：-1~1

纹理坐标的原点在纹理图的左下角，若沿 Y 轴向下偏移，偏移量为负

glMatrixMode (GL_TEXTURE)往往与 glMatrixMode (GL_MODELVIEW)混用，注意保存好现场。

2.3.2 渲染（Render)

函数 glDrawArrays 和 glDrawElements 用来将上面的设置画在屏幕上。这两个函数的有一个相同的参数 mode，接受 GL_POINTS、GL_LINE_STRIP、GL_LINE_LOOP、GL_LINES、GL_TRIANGLES 等常量。

下面简单介绍一个常量的意思，

GL_POINTS：表示只画顶点；

GL_LINE_STRIP：从第一个顶点画到最后一个点；

GL_LINE_LOOP：和 GL_LINE_STRIP 相似，只是最后一点和第一个点链接，形成一个环路；

GL_LINES：每两个点画一条直线；

GL_TRIANGLES： 没三个点为一组画成一个个的三角形；

GL_TRIANGLE_STRIP：按一定的顺序画一系列三角形如按照顶点 v0, v1, v2, 然后 v2, v1, v3 (注意顺序), 最后 v2, v3, v4 ，确定这些三角形能正确的形成一个多边形；

GL_TRIANGLE_FAN：相似 GL_TRIANGLE_STRIP，但是顺序是 v0, v1, v2, 然后 v0, v2, v3,

最后 v0, v3, v4， 注意和 GL_TRIANGLE_STRIP 顺序区别；

public abstract void glDrawArrays(int mode, int first, int count)：按照前面 verticesBuffer 数组的顶点来画图形。

public abstract void glDrawElements(int mode, int count, int type, Buffer indices)：需要知道顶点的顺序，才能画图。

在第三部分关于多边形的内容中，指定了顶点的顺序。type 类型的含义是指明 indices 数组的类型，比如 GL10.GL_UNSIGNED_SHORT，表示 indices 是 short 类型的数组。

2.3.3 OpenGL 的坐标系统变换

OpenGL 的重要功能之一是将三维的世界坐标经过变换、投影等计算，最终算出它在显示设备上对应的位置。简单的讲，OpenGL 中从三维场景到屏幕图形要经历如下所示的变换过程：

模型坐标-〉世界坐标-〉观察坐标-〉投影坐标-〉设备坐标

OpenGL 的重要功能之一就是将三维的世界坐标经过变换、投影等计算，最终算出它在显示设备上对应的位置，这个位置就称为设备坐标。在屏幕、打印机等设备上的坐标是二维坐标。

1)几何变换：平移、旋转、缩放

```
glTranslatef (x, y, z)
glRotatef (alpha, x, y, z);
glScalef (x, y, z);
```

2)几何变换栈：为了维护几何变换， OpenGL 维护一个几何变换栈，每次几何变换，都用相应的矩阵乘对应的栈顶元素，并替换掉上次的栈顶。

对于几何变换栈，有两个操作可以使用：

glPushMatrix()： 保存当前坐标系，复制当前栈顶，并把复制的内容再次放到栈顶，能够保护栈上以前的内容；

glPopMatrix()： 恢复当前坐标系。

3)投影变换 ： 正交投影（正射投影、平行投影），透视投影

正交投影：将 3D 模型平行的映射到平面上，glOrtho(xleft, xright, ybottom, ytop, znear, zfar);

透视投影：将 3D 模型映射到相对某个观察点的平面上， gluPerspective(fovy, aspect, znear, zfar);

4)切换当前栈

切换当前操作的栈为投影变换栈：glMatrixMode(GL_PROJECTION);

切换当前操作的栈为几何变换栈：glMatrixMode(GL_MODELVIEW);

清除当前操作栈的内容：glLoadIdentity();

2.4 实例代码

实现功能：

1)绘制一个正方体

2)给正方体增加 3 轴旋转功能；

3)给正方体添加纹理；

模块划分：

Main 为入口 Activity

CMFSurfaceView 为窗口，处理传递到 view 的手势事件

CMFRender 为渲染器，绘制一个旋转的正方形，并对正方形形进行贴图

主要代码如下：

CMFRender 代码如下：

```
public class CMFRender    implements Renderer{

    private final float TOUCH_SCALE_FACTOR = 180.0f / 320;
    //转动时速度矢量
    private float rotateV = 0;
    //已旋转角度
    private float rotated = 0;

    private int texture = -1;
    private int one = 0x10000;

    private int[] quarter = {-one,-one,one,
                             one,-one,one,
                             one,one,one,
                             -one,one,one,

                             -one,-one,-one,
                             -one,one,-one,
                             one,one,-one,
                             one,-one,-one,

                             -one,one,-one,
                             -one,one,one,
                             one,one,one,
                             one,one,-one,

                             -one,-one,-one,
                             one,-one,-one,
                             one,-one,one,
                             -one,-one,one,

                             one,-one,-one,
                             one,one,-one,
                             one,one,one,
                             one,-one,one,

                             -one,-one,-one,
                             -one,-one,one,
```

```
                                -one,one,one,
                                -one,one,-one,};

    private int[] texCoords = {one,0,0,0,0,one,one,one,
            0,0,0,one,one,one,one,0,
                                one,one,one,0,0,0,0,one,
                                0,one,one,one,one,0,0,0,
                                0,0,0,one,one,one,one,0,
                                one,0,0,0,0,one,one,one,};


    //准备正方体顶点
    private IntBuffer quarterBuffer = BufferUtil.iBuffer(quarter);
    //纹理映射数据
    private IntBuffer texCoordsBuffer = BufferUtil.iBuffer(texCoords);
    ByteBuffer indicesBuffer = ByteBuffer.wrap(new byte[]{
            0,1,3,2,
            4,5,7,6,
            8,9,11,10,
            12,13,15,14,
            16,17,19,18,
            20,21,23,22,
    });

    private float rotateX; //用于正方体 x 轴的旋转；
    private float rotateY; //用于正方体 y 轴的旋转；
    private float rotateZ; //用于正方体 z 轴的旋转；


    private CMFSurfaceView surface = null;
    /**渲染器*/
    public CMFRender(CMFSurfaceView surface){
//          Log.i("tg","Renderer  构造。");
            this.surface = surface;

    }

    @Override
    public void onDrawFrame(GL10 gl) {
        // TODO Auto-generated method stub
        //清楚屏幕和深度缓存

        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);
```

```
        //重置当前的观察模型矩阵
        gl.glLoadIdentity();
        gl.glPushMatrix();

        //现将屏幕向里移动，用来画正方体
        gl.glTranslatef(0.0f, 0.0f, -4.0f);
        gl.glScalef(1.5f, 1.5f, 1.0f);
        //设置 3 个方向的旋转
        gl.glRotatef(rotateX+rotated, 1.0f, 0.0f, 0.0f);
        gl.glRotatef(rotateY+rotated, 0.0f, 1.0f, 0.0f);
//          gl.glRotatef(rotateZ, 0.0f, 0.0f, 1.0f);

        //通知 opnegl 将文理名字 texture 绑定到指定的纹理目标上 GL10.GL_TEXTURE_2D
        gl.glBindTexture(GL10.GL_TEXTURE_2D, texture);

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        //纹理的使用与开启颜色渲染一样，需要开启纹理功能
        gl.glEnableClientState(GL10.GL_TEXTURE_COORD_ARRAY);

        //设置正方体 各顶点
        gl.glVertexPointer(3, GL10.GL_FIXED, 0, quarterBuffer);
        gl.glTexCoordPointer(2, GL10.GL_FIXED, 0, texCoordsBuffer);

        //绘制
        gl.glDrawElements(GL10.GL_TRIANGLE_STRIP,  24,  GL10.GL_UNSIGNED_BYTE,
indicesBuffer);

        gl.glDisableClientState(GL10.GL_TEXTURE_COORD_ARRAY);
        gl.glDisableClientState(GL10.GL_VERTEX_ARRAY);
        gl.glPopMatrix();
        rotateX += 0.5f;
        rotateY += 0.6f;
        rotateZ += 0.3f;
    }


    //当窗口改变时，调用，至少在创建窗口时调用一次，这边设置下场景大小
    @Override
    public void onSurfaceChanged(GL10 gl, int width, int height) {
        // TODO Auto-generated method stub
        //设置 OpenGL 场景大小
        float ratio = (float) width / height;
        gl.glViewport(0, 0, width, height);
```

```
        gl.glMatrixMode(GL10.GL_PROJECTION);//设置为投影矩阵模式
        gl.glLoadIdentity();//重置
        gl.glFrustumf(-ratio, ratio, -1, 1, 1, 10);//设置视角
        gl.glMatrixMode(GL10.GL_MODELVIEW);
        gl.glLoadIdentity();
    }


    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        // TODO Auto-generated method stub
        //设置清除屏幕时所用的颜色，参数依次为红、绿、蓝、Alpha 值
        gl.glClearColor(0.0f, 0f, 1f, 0.5f);
        gl.glDisable(GL10.GL_CULL_FACE);
        //启用阴影平滑
        gl.glShadeModel(GL10.GL_SMOOTH);
        gl.glEnable(GL10.GL_DEPTH_TEST);//启用深度测试

        gl.glClearDepthf(1.0f);//设置深度缓存
        gl.glDepthFunc(GL10.GL_LEQUAL);//所做深度测试的类型

        //告诉系统对透视进行修正
        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT, GL10.GL_FASTEST);
        //允许 2D 贴图
        gl.glEnable(GL10.GL_TEXTURE_2D);

        IntBuffer intBuffer = IntBuffer.allocate(1);
        //创建纹理
        gl.glGenTextures(1, intBuffer);
        texture = intBuffer.get();
        //设置需要使用的纹理
        gl.glBindTexture(GL10.GL_TEXTURE_2D, texture);

        Bitmap bmp = GLImage.mBitmap;
        //生成纹理
        GLUtils.texImage2D(GL10.GL_TEXTURE_2D, 0, bmp, 0);
        //  线形滤波
        gl.glTexParameterx(GL10.GL_TEXTURE_2D,          GL10.GL_TEXTURE_MIN_FILTER,
GL10.GL_LINEAR);
        gl.glTexParameterx(GL10.GL_TEXTURE_2D,          GL10.GL_TEXTURE_MAG_FILTER,
GL10.GL_LINEAR);
    }

    /**
```

```
     * 响应触摸移动
     * @param dx
     * @param dy
     */
    public void onTouchMove(float dx,float dy){
        rotateV = Math.abs(dx) + Math.abs(dy);
//        Log.i("tg","GL rotateV/" + rotateV);
        rotated += rotateV*TOUCH_SCALE_FACTOR;

    }


}
```

CMFSurfaceView 代码如下：

```
public class CMFSurfaceView extends GLSurfaceView {

    private CMFRender mRenderer = null;
    private float mPreviousX = 0;
    private float mPreviousY = 0;

    public CMFSurfaceView(Context context) {
        super(context);
        // 设置渲染器，
        mRenderer = new CMFRender(this);
        setRenderer(mRenderer);
        // 设置描绘方式，
        setAutoRender(true);
        this.requestRender();
    }

    @Override
    public boolean onTouchEvent(MotionEvent e) {
        float x = e.getX();
        float y = e.getY();
        //转换坐标方向；
        y = -y;

        switch (e.getAction()) {
        case MotionEvent.ACTION_MOVE:
            float dx = x - mPreviousX;
            float dy = y - mPreviousY;
            mRenderer.onTouchMove(dx, dy);
        case MotionEvent.ACTION_DOWN:
//            Log.i("tg","touch down/" + x + "/" + y);
```

```
                this.mPreviousX = x;
                this.mPreviousY = y;
                break;
            case MotionEvent.ACTION_UP:
//              Log.i("tg","touch up/" + x + "/" + y);
                this.mPreviousX = 0;
                this.mPreviousY = 0;
//              setAutoRender(true);
//              this.mRenderer.startRotate();
                break;
            }
            this.requestRender();
            return true;
        }
        /**
         * 设置是否自动连续渲染
         * @param auto
         */
        public void setAutoRender(boolean auto){
            // RENDERMODE_WHEN_DIRTY-有改变时重绘-需调用 requestRender()
            // RENDERMODE_CONTINUOUSLY -自动连续重绘（默认）
            if(auto){
                setRenderMode(GLSurfaceView.RENDERMODE_CONTINUOUSLY);
            }else{
                setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
            }
        }
}
```

三、Opengles 中两个基本的类:GLSurfaceView 和 GLSurfaceView.Renderer

By sinoace

3.1 OpenGL ES 简介

Android 3D 引擎采用的是 OpenGL ES。OpenGL ES 是一套为手持和嵌入式系统设计的 3D 引擎 API，由 Khronos 公司维护。在 PC 领域，一直有两种标准的 3D API 进行竞争，OpenGL 和 DirectX。一般主流的游戏和显卡都支持这两种渲染方式，DirectX 在 Windows 平台上有很大的优势，但是 OpenGL 具有更好的跨平台性。

由于嵌入式系统和 PC 相比，一般说来，CPU、内存等都比 PC 差很多，而且对能耗有着特殊的要求，许多嵌入式设备并没有浮点运算协处理器，针对嵌入式系统的以上特点，Khronos 对标准的 OpenGL 系统进行了维护和改动，以期望满足嵌入式设备对 3D 绘图的要求。

Android 系统使用 OpenGL 的标准接口来支持 3D 图形功能，android 3D 图形系统也分为 java 框架和本地代码两部分。本地代码主要实现的 OpenGL 接口的库，在 Java 框架层，javax.microedition.khronos.opengles 是 java 标准的 OpenGL 包，android.opengl 包提供了 OpenGL 系统和 Android GUI 系统之间的联系。

在 Android 框架中有两个基本的类使你可以通过 OpenGLES API 创建和操作图形系统：GLSurfaceView 和 GLSurfaceView.Renderer.

3.2 GLSurfaceView

这个类是一个 View，你可以用 OpenGLAPI 调用来绘制对象并管理它们.它与 SurfaceView 很相似. 你可以创建一个 GLSurfaceView 的实例然后把你的绘制操作添加给它. 然而，如果你想捕获触屏事件，你应扩展 GLSurfaceView 类来实现触屏事件监听器，就像在 SDK 的 OpenGL 例子 ES1.0, ES 2.0 和 TouchRotateActivity 中所示.

3.3 GLSurfaceView.Renderer

此接口定义了在一个 OpenGL GLSurfaceView 上作画所需的方法们. 你必须提供另一个类来实现这个接口然后把它附加到你的 GLSurfaceView 实例上，使用 GLSurfaceView.setRenderer().

GLSurfaceView.Renderer 接口需要你实现以下方法们：

onSurfaceCreated():当创建 GLSurfaceView 时被调用，只调用一次. 在这个方法中执行只发生一次的动作，比如设置 OpenGL 环境参数或初始化 OpenGL 图形对象.

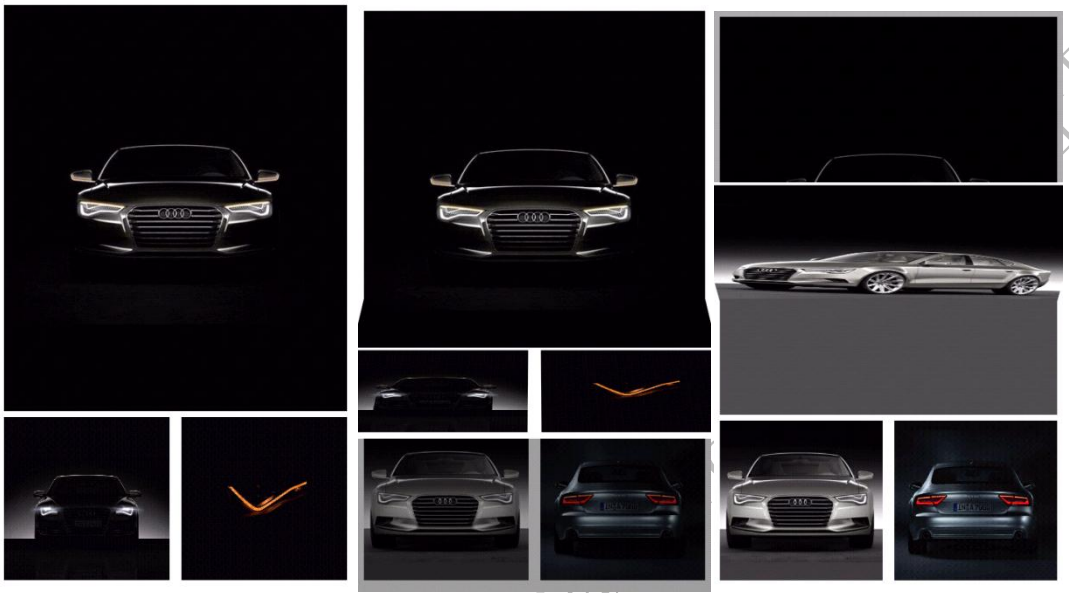
onDrawFrame():系统在每次重绘 GLSurfaceView 时调用此方法. 此方法是绘制图形对象

的主要的执行点.

onSurfaceChanged(): 当 GLSurfaceView 几何体改变时系统调用此方法，比如 GLSurfaceView 的大小改变或设备屏幕的方向改变. 使用此方法来响应 GLSurfaceView 容器的变化.

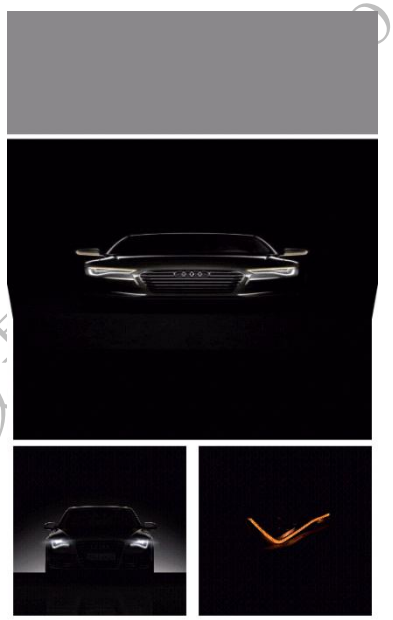
下面通过一个例子来介绍一下 OpenGL，利用 OpenGL 实现类似社交杂志 FlipBoard 的翻页效果。

首先介绍一下首页效果，首页为品字形页面见下图：



以中线为基础翻过一页

然后再翻第二页，一共三页都是类似 FlipBoard 效果以中心线为基础翻页，没有见过这种翻页效果的同学可以下载一个 FlipBoard 自己看一下，最后显示一下首页向前翻页效果，因为　前面没有页面所以翻到一半就不可以再翻，并露出灰色背景色。



最后一页翻页完成后

3.3.1 翻页效果：

好了下面直接上代码：

```
/**
 * 类似 FlipBoard 翻页效果首页
 * @author sinoace
 */
public class HomeActivity extends Activity {
    private FlipViewController flipView;

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        // 设置标题
        setTitle(R.string.activity_title);
        // 类似 FlipBoard 翻页效果视图
        flipView = new FlipViewController(this);
        // 设置适配器
        flipView.setAdapter(new MyBaseAdapter(this));
        // 设置显示翻页效果视图
        setContentView(flipView);
    }

    @Override
    protected void onResume() {
        super.onResume();
        // 重新启用翻页视图
```

```
        flipView.onResume();
    }

    @Override
    protected void onPause() {
        super.onPause();
        // 暂停使用翻页视图
        flipView.onPause();
    }

    /**
     * 类似 FlipBoard 翻页效果视图适配器
     * @author sinoace
     */
    private static class MyBaseAdapter extends BaseAdapter {
        // 显示内容描述列表 此处都为图片
        private static List<Data> IMG_DESCRIPTIONS = new ArrayList<Data>();

        static {
            IMG_DESCRIPTIONS.add(new Data("a7_H.jpg", "audi_l.jpg", "audi_r.jpg"));
            IMG_DESCRIPTIONS.add(new Data("a7_S.jpg", "a7_l.jpg", "a7_r.jpg"));
            IMG_DESCRIPTIONS.add(new Data("audi_a7_H.jpg", "audi_a7_l.jpg",
"audi_a7_r.jpg"));
        }
        // 布局填充器-大家都懂得
        private LayoutInflater inflater;

        private MyBaseAdapter(Context context) {
            inflater = LayoutInflater.from(context);
        }

        @Override
        public int getCount() {
            return IMG_DESCRIPTIONS.size();
        }

        @Override
        public Object getItem(int position) {
            return position;
        }

        @Override
        public long getItemId(int position) {
            return position;
```

```
        }

        @Override
        public View getView(final int position, View convertView, ViewGroup parent) {
            View layout = convertView;
            if (convertView == null)
                layout = inflater.inflate(R.layout.home, null);

            final Data data = IMG_DESCRIPTIONS.get(position);

            ImageView iv_pic_head = (ImageView) layout.findViewById(R.id.iv_pic_head);
            ImageView iv_pic_l = (ImageView) layout.findViewById(R.id.iv_pic_l);
            ImageView iv_pic_r = (ImageView) layout.findViewById(R.id.iv_pic_r);
            // 加载图片 图片位置在 assets 目录
            iv_pic_head.setImageBitmap(IO.readBitmap(inflater.getContext().getAssets(),
data.imageFilenameH));
            iv_pic_l.setImageBitmap(IO.readBitmap(inflater.getContext().getAssets(),
data.imageFilenameL));
            iv_pic_r.setImageBitmap(IO.readBitmap(inflater.getContext().getAssets(),
data.imageFilenameR));

            iv_pic_head.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    Toast.makeText(inflater.getContext(), "奥迪 A7  灵感天成", 0).show();
                    // 打开一个本地视频文件 - 演示效果
                    Intent intent = new Intent(inflater.getContext(),VideoActivity.class);
                    inflater.getContext().startActivity(intent);
                }
            });
            iv_pic_l.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    Toast.makeText(inflater.getContext(), "左边", 0).show();
                    // 打开 AudiA7 介绍页面 - 演示效果
                    Intent intent = new Intent(inflater.getContext(),AudiA7Activity.class);
                    inflater.getContext().startActivity(intent);

                }
            });
            iv_pic_r.setOnClickListener(new View.OnClickListener() {
                @Override
                public void onClick(View v) {
                    Toast.makeText(inflater.getContext(), "右边", 0).show();
```

```
                    // 打开 AudiA7 介绍页面 - 演示效果
                    Intent intent = new Intent(inflater.getContext(),AudiA7Activity.class);
                    inflater.getContext().startActivity(intent);
                }
            });

            return layout;
        }

        private static class Data {
            public String imageFilenameH;
            public String imageFilenameL;
            public String imageFilenameR;

            private Data(String imageFilenameH, String imageFilenameL, String
imageFilenameR) {
                this.imageFilenameH = imageFilenameH;
                this.imageFilenameL = imageFilenameL;
                this.imageFilenameR = imageFilenameR;
            }
        }
    }
}
```

下面说一下翻页效果的控制类 FlipViewController 这个类定义了翻页的基本设置

```
 public class FlipViewController extends AdapterView<Adapter> {
    private static final int MSG_SURFACE_CREATED = 1;
    // 主线程 handler
    private Handler handler = new Handler(new Handler.Callback() {
        @Override
        public boolean handleMessage(Message msg) {
            if (msg.what == MSG_SURFACE_CREATED) {
                contentWidth = 0;
                contentHeight = 0;
                requestLayout();
                return true;
            }
            return false;
        }
    });

    private GLSurfaceView surfaceView;
    private FlipRenderer renderer;
    private FlipCards cards;
```

```
private int contentWidth;
private int contentHeight;

private boolean enableFlipAnimation = true;

private boolean inFlipAnimation = false;

private Adapter adapter;
private DataSetObserver adapterDataObserver = new DataSetObserver() {
    // 内容观察者 当数据改变时调用
    @Override
    public void onChanged() {
        View v = bufferedViews.get(bufferIndex);
        if (v != null) {
            for (int i = 0; i < adapter.getCount(); i++) {
                if (v.equals(adapter.getItem(i))) {
                    adapterIndex = i;
                    break;
                }
            }
        }
        reloadAllViews();
    }
};
// 缓存视图列表
private final LinkedList<View> bufferedViews = new LinkedList<View>();
// 记录释放的相关视图索引
private final LinkedList<View> releasedViews = new LinkedList<View>();
private int bufferIndex = -1;
private int adapterIndex = -1;
private int sideBufferSize = 1;

private float touchSlop;
@SuppressWarnings("unused")
private float maxVelocity;

public FlipViewController(Context context) {
    super(context);
    ViewConfiguration configuration = ViewConfiguration.get(getContext());
    // 得到用户滑动触摸的距离
    touchSlop = configuration.getScaledTouchSlop();
    // 得到用户滑动触摸的最大速度
    maxVelocity = configuration.getScaledMaximumFlingVelocity();
    // 设置 SurfaceView
```

```
        setupSurfaceView();
    }

    @Override
    public void setAdapter(Adapter adapter) {
        setAdapter(adapter, 0);
    }
    // 设置适配器
    public void setAdapter(Adapter adapter, int initialPosition) {
        if (this.adapter != null)
            // 反注册数据观察者
            this.adapter.unregisterDataSetObserver(adapterDataObserver);

        this.adapter = adapter;

        if (this.adapter != null) {
            // 注册数据观察者
            this.adapter.registerDataSetObserver(adapterDataObserver);
            if (this.adapter.getCount() > 0)
                // 设置初始位置
                setSelection(initialPosition);
        }
    }

    ..... // 省略部分代码

    // 设置位置
    @Override
    public void setSelection(int position) {
        if (adapter == null)
            return;

        Assert.assertTrue("Invalid selection position", position >= 0 && position <
adapter.getCount());

        releaseViews();
        // position 位置的视图
        View selectedView = viewFromAdapter(position, true);
        bufferedViews.add(selectedView);

        for (int i = 1; i <= sideBufferSize; i++) {
            int previous = position - i;
            int next = position + i;
```

```
            if (previous >= 0)
                bufferedViews.addFirst(viewFromAdapter(previous, false));
            if (next < adapter.getCount())
                bufferedViews.addLast(viewFromAdapter(next, true));
        }

        bufferIndex = bufferedViews.indexOf(selectedView);
        adapterIndex = position;

        requestLayout();
        // 更新视图
        updateVisibleView(inFlipAnimation ? -1 : bufferIndex);
    }

    ..... // 省略部分代码

    // 设置 SurfaceView
    private void setupSurfaceView() {
        surfaceView = new GLSurfaceView(getContext());
        // 创建一个翻页卡的实例（每一页就是一个翻页卡）
        cards = new FlipCards(this);
        // 通过翻页卡创建一个渲染器实例
        renderer = new FlipRenderer(this, cards);
        // 配置选择器 选择一个配置
        surfaceView.setEGLConfigChooser(8, 8, 8, 8, 16, 0);
        // 设置为顶部视图
        surfaceView.setZOrderOnTop(true);
        // 设置渲染器
        surfaceView.setRenderer(renderer);
        // 设置透明样式
        surfaceView.getHolder().setFormat(PixelFormat.TRANSLUCENT);
        // 复用视图 使用 RENDERMODE_WHEN_DIRTY 模式可以提高电池寿命和总体系统
性能
        surfaceView.setRenderMode(GLSurfaceView.RENDERMODE_WHEN_DIRTY);
        // 添加要显示的布局文件列表
        addViewInLayout(surfaceView, -1, new
AbsListView.LayoutParams(LayoutParams.FILL_PARENT, LayoutParams.FILL_PARENT), false);
    }

    ..... // 省略部分代码

    // 实现视图翻页效果
    void flippedToView(int indexInAdapter) {
        debugBufferedViews();
```

```
            if (indexInAdapter >= 0 && indexInAdapter < adapter.getCount()) {
                // 向前一页
                if (indexInAdapter == adapterIndex + 1) {
                    if (adapterIndex < adapter.getCount() - 1) {
                        adapterIndex++;
                        View old = bufferedViews.get(bufferIndex);
                        if (bufferIndex > 0)
                            releaseView(bufferedViews.removeFirst());
                        if (adapterIndex + sideBufferSize < adapter.getCount())
                            bufferedViews.addLast(viewFromAdapter(adapterIndex +
sideBufferSize, true));
                        bufferIndex = bufferedViews.indexOf(old) + 1;
                        requestLayout();
                        // 更新视图
                        updateVisibleView(inFlipAnimation ? -1 : bufferIndex);
                    }
                // 向后翻页
                } else if (indexInAdapter == adapterIndex - 1) {
                    if (adapterIndex > 0) {
                        adapterIndex--;
                        View old = bufferedViews.get(bufferIndex);
                        if (bufferIndex < bufferedViews.size() - 1)
                            releaseView(bufferedViews.removeLast());
                        if (adapterIndex - sideBufferSize >= 0)
                            bufferedViews.addFirst(viewFromAdapter(adapterIndex -
sideBufferSize, false));
                        bufferIndex = bufferedViews.indexOf(old) - 1;
                        requestLayout();
                        // 更新视图
                        updateVisibleView(inFlipAnimation ? -1 : bufferIndex);
                    }
                } else
                    setSelection(indexInAdapter);
            } else
                Assert.assertTrue("Invalid indexInAdapter: " + indexInAdapter, false);

        }
        // 翻页完成后隐藏动画
        void postHideFlipAnimation() {
            handler.post(new Runnable() {
                @Override
                public void run() {
                    hideFlipAnimation();
```

```
                }
            });
        }
        // 显示翻页动画
        void showFlipAnimation() {
            if (!inFlipAnimation) {
                inFlipAnimation = true;

                cards.setVisible(true);
                surfaceView.requestRender();
                // 使用一个延迟消息以避免闪烁
                handler.postDelayed(new Runnable() {
                    public void run() {
                        if (inFlipAnimation)
                            updateVisibleView(-1);
                    }
                }, 100);
            }
        }
        // 隐藏翻页动画
        private void hideFlipAnimation() {
            if (inFlipAnimation) {
                inFlipAnimation = false;

                updateVisibleView(bufferIndex);

                handler.post(new Runnable() {
                    public void run() {
                        if (!inFlipAnimation)
                            cards.setVisible(false);
                    }
                });
            }
        }
}
```

3.3.2 用代码说一下渲染器 GLSurfaceView.Renderer 的使用

```
/**
* 翻页渲染器
*/
public class FlipRenderer implements GLSurfaceView.Renderer {
```

```
private FlipViewController flipViewController;

private FlipCards cards;

private boolean created = false;

private final LinkedList<Texture> postDestroyTextures = new LinkedList<Texture>();

public FlipRenderer(FlipViewController flipViewController, FlipCards cards) {
    this.flipViewController = flipViewController;
    this.cards = cards;
}
// 当 Surface 创建时调用此方法
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    // 设置背景色
    gl.glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    // 阴影平滑
    gl.glShadeModel(GL_SMOOTH);
    // 设置深度值
    gl.glClearDepthf(1.0f);
    // 深度测试 在涉及到消隐等情况（可能遮挡）都要开启深度测试
    // glEnable(GL_DEPTH_TEST),硬件上打开了深度缓存区，当有新的同样 XY 坐标的片断到来时，比较两者的深度
    // 并且在初始化时打开深度，绘制每一帧前要 gl_clear(gl-depth-bit)
    // 这根 clear_buffer_bit 类似，而且同样要设置 clear_deppth_bitd 值，
    // 用 glClearDepth(GLclampd depth)，一般设为 1，这将背景设为最深，这是默认的，通常不用写
    gl.glEnable(GL_DEPTH_TEST);

    // 深度检测设置
    gl.glDepthFunc(GL_LEQUAL);

    // 函数原型: void glHint(GLenum target,GLenum mod)
    // 参数说明：

    // target：指定所控制行为的符号常量，可以是以下值
    // 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。
        gl.glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

        created = true;
        // 不使用纹理
        cards.invalidateTexture();
        // 重新加载纹理
        flipViewController.reloadTexture();

    }

    public static float[] light0Position = {0, 0, 100f, 0f};

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

        // glViewport 它负责把视景体截取的图像按照怎样的高和宽显示到屏幕上
        gl.glViewport(0, 0, width, height);
```

// 指定哪一个矩阵是当前矩阵，就是对接下来要对什么进行操作做一下说明，他有三个参数选择

// GL_MODELVIEW,对模型视景矩阵堆栈应用随后的矩阵操作。

// 这个是对模型视景的操作，接下来的语句描绘一个以模型为基础的场景，这样来设置参数，接下来用到的就是像 gluLookAt()这样的函数；

// GL_PROJECTION,对投影矩阵应用随后的矩阵操作.

// 这个是投影的意思，就是要对投影相关进行操作，也就是把物体投影到一个平面上，就像我们照相一样，把 3 维物体投到 2 维的平面上。

// 这样，接下来的语句可以是跟透视相关的函数，比如 glFrustum()或 gluPerspective()；

// GL_TEXTURE,对纹理矩阵堆栈应用随后的矩阵操作.

```
        gl.glMatrixMode(GL_PROJECTION);

        // 恢复初始坐标系 重置当前指定的矩阵为单位矩阵
        gl.glLoadIdentity();
```

```
        float fovy = 20f;
        float eyeZ = height / 2f / (float) Math.tan(TextureUtils.d2r(fovy / 2));
        // 建立一个透视投影矩阵
        GLU.gluPerspective(gl, fovy, (float) width / (float) height, 0.5f, Math.max(2500.0f, eyeZ));

        gl.glMatrixMode(GL_MODELVIEW);
        // 恢复初始坐标系 重置当前指定的矩阵为单位矩阵
        gl.glLoadIdentity();

        // 建立一个透视投影
        GLU.gluLookAt(gl,
                width / 2.0f, height / 2f, eyeZ,
                width / 2.0f, height / 2.0f, 0.0f,
                0.0f, 1.0f, 0.0f
        );
        // 启动总光源开关
        gl.glEnable(GL_LIGHTING);
        //允许光源 0 的使用
        gl.glEnable(GL_LIGHT0);

        float lightAmbient[] = new float[]{3.5f, 3.5f, 3.5f, 1f};
        // 光照设置 指定光源 0 的环境光参数
        gl.glLightfv(GL_LIGHT0, GL_AMBIENT, lightAmbient, 0);

        light0Position = new float[]{0, 0, eyeZ, 0f};
        //指定光源 0 的位置
        gl.glLightfv(GL_LIGHT0, GL_POSITION, light0Position, 0);

    }
    // 绘制框架
    @Override
    public void onDrawFrame(GL10 gl) {
        gl.glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

        synchronized (postDestroyTextures) {
            for (Texture texture : postDestroyTextures)
                texture.destroy(gl);
            postDestroyTextures.clear();
        }

        cards.draw(this, gl);
    }
    // 销毁纹理后的工作
```

```
    public void postDestroyTexture(Texture texture) {
        synchronized (postDestroyTextures) {
            postDestroyTextures.add(texture);
        }
    }
    // 更新纹理
    public void updateTexture(int frontIndex, View frontView, int backIndex, View backView) {
        if (created) {
            cards.reloadTexture(frontIndex, frontView, backIndex, backView);
            flipViewController.getSurfaceView().requestRender();
        }
    }
    // 检查错误
    public static void checkError(GL10 gl) {
        int error = gl.glGetError();
        if (error != 0)
            throw new RuntimeException(GLU.gluErrorString(error));
    }
}
```

3.3.3 最后简单说明一下纹理类

```
/*
   纹理类
 */
public class Texture {
    private FlipRenderer renderer;

    private int[] id = {0};

    private int width, height;
    private int contentWidth, contentHeight;

    private boolean destroyed = false;

    private Texture() {
    }

    public static Texture createTexture(Bitmap bitmap, FlipRenderer renderer, GL10 gl) {
        Texture t = new Texture();
        t.renderer = renderer;
```

```
int w = Integer.highestOneBit(bitmap.getWidth() - 1) << 1;
int h = Integer.highestOneBit(bitmap.getHeight() - 1) << 1;

t.contentWidth = bitmap.getWidth();
t.contentHeight = bitmap.getHeight();
t.width = w;
t.height = h;

/*
glGenTextures(GLsizei n, GLuint *textures)函数说明
n：用来生成纹理的数量
textures：存储纹理索引的
glGenTextures 函数根据纹理参数返回 n 个纹理索引。
纹理名称集合不必是一个连续的整数集合。
（glGenTextures 就是用来产生你要操作的纹理对象的索引的，
比如你告诉 OpenGL，我需要 5 个纹理对象，它会从没有用到的整数里返回 5 个给你）
glBindTexture 实际上是改变了 OpenGL 的这个状态，它告诉 OpenGL 下面对纹理的任何操作都是对它所绑定的纹理对象的，
比如 glBindTexture(GL_TEXTURE_2D,1)告诉 OpenGL 下面代码中对 2D 纹理的任何设置都是针对索引为 1 的纹理的。
产生纹理函数假定目标纹理的面积是由 glBindTexture 函数限制的。
先前调用 glGenTextures 产生的纹理索引集不会由后面调用的 glGenTextures 得到，
除非他们首先被 glDeleteTextures 删除。你不可以在显示列表中包含 glGenTextures。
*/
gl.glGenTextures(1, t.id, 0);

// 允许建立一个绑定到目标纹理的有名称的纹理。
// 例如，一幅具有真实感的图像或者照片作为纹理贴到一个矩形上，就可以在定义纹理对象生成纹理对象数组后，
// 通过使用 glBindTexture 选择纹理对象，来完成该纹理对象的定义。
gl.glBindTexture(GL_TEXTURE_2D, t.id[0]);

// GL_TEXTURE_MIN_FILTER 设置最小过滤，第三个参数决定过滤形式，此处为线性过滤
gl.glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
// GL_TEXTURE_MAG_FILTER 设置最大过滤，第三个参数决定过滤形式，此处为线性过滤
gl.glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
/*
生成纹理 将载入的位图文件(*.bmp)转换成纹理贴图。
glTexImage2D()的用法举例

glTexImage2D(GL_TEXTURE_2D,     //此纹理是一个 2D 纹理
```

```
            0,                                  //代表图像的详细程度，默认为 0 即可
            3,                                  //颜色成分 R(红色分量)、G(绿色分量)、B(蓝色分量)三部分，若为 4 则是 R(红色分量)、G(绿色分量)、B(蓝色分量)、Alpha
            TextureImage[0]->sizeX,             //纹理的宽度
            TextureImage[0]->sizeY,             //纹理的高度
            0,                                  //边框的值
            GL_RGB,                             //告诉 OpenGL 图像数据由红、绿、蓝三色数据组成
            GL_UNSIGNED_BYTE,                   //组成图像的数据是无符号字节类型
            TextureImage[0]->data);             //告诉 OpenGL 纹理数据的来源,此例中指向存放在 TextureImage[0]记录中的数据
             */
            gl.glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA, GL_UNSIGNED_BYTE, null);

            GLUtils.texSubImage2D(GL_TEXTURE_2D, 0, 0, 0, bitmap);

            return t;
        }

        ..... // 省略部分代码

        public int[] getId() {
            return id;
        }

        public int getWidth() {
            return width;
        }

        public int getHeight() {
            return height;
        }

        public int getContentWidth() {
            return contentWidth;
        }

        public int getContentHeight() {
            return contentHeight;
        }
}
```

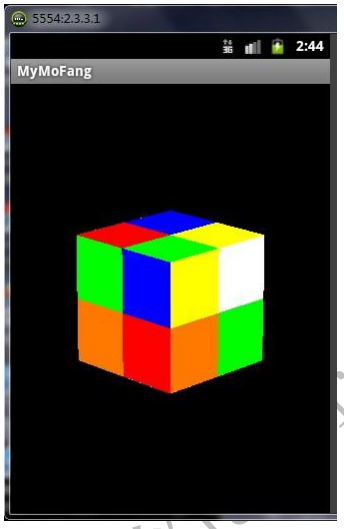
以上几大类就是实现类似 flipboard 翻页效果的基本类，当然还有很多类因为时间关系没

有罗列出来，因为本人对 OpenGL 也只是懂一点皮毛而已，在这里给大家提供一个参考的小例子，其实 OpenGL 的很多实用功能还是要靠大家挖掘的。

四、opengles 之二阶魔方

开发这个是源于在网上很难找到魔方的教程，很多教程都只是文字，而没有实际的代码。往往让初学者很苦恼。又因为一般的魔方涉及的代码量很多，并且对于刚接触 OpenGL ES 的人来说真的是无从下手。还有就是设计一个魔方需要计算的数字也很多，一个不小心就会出错。所以，基于以上的原因，我就讲解一下二阶魔方的制作。（因为代码量的问题，很多需要结合源码一起学习）

先看看效果图吧（其实蛮丑的，因为没有用到纹理，直接使用颜色 RGBA）



还是以一些核心代码来讲解吧!

4.1 首先是立方体的绘制：

基本思路是绘制 8 大小相等的立方体，然后利用 gl.glPushMatrix()和 gl.glPopMatrix()方法以原点为中心进行魔方的绘制。主要代码如下：

Java 代码：

```
public void DrawCube1(GL10 gl){
gl.glPushMatrix();//保护变换矩阵现场
gl.glTranslatef(0.48f, 0.48f, 0.48f); //在坐标为(0.48f, 0.48f, 0.48f)点上绘制第一个立方体
gl.glColor4f(color[0], color[1], color[2], 1.0f);
DrawSquare(gl,new float[]{//z 轴面
-0.48f,-0.48f,0.48f,
-0.48f,0.48f,0.48f,
0.48f,-0.48f,0.48f,
0.48f,0.48f,0.48f
});
……
```

```
gl.glColor4f(0,0,0, 1f);
DrawCube(gl,0.45f,0.45f,0.45f);
gl.glPopMatrix();//恢复变换矩阵现场
}
```

代码分析：

这段代码的意思是以点(0.48f, 0.48f, 0.48f)为中心，绘制一个长=0.96，宽为 0.96，高为 0.96 的立方体，并且在此立方体的 Z，Y，X 面绘制正方形，绘制不同的颜色，2-8 立方体以此类推。只要改变一下 gl.glTranslatef(0.48f, 0.48f, 0.48f); 这里的值就可以了。DrawCube(gl,0.45f,0.45f,0.45f);这个函数由于太多了，就不在这里写出来了，它的作用就是绘制一个立方体，参数分别为：长，宽，高。（不懂的话，看看源码）由于以上的代码还涉及到颜色的绘制，那接下来就讲解一下颜色的绘制。

4.2 颜色的绘制：

其实，就相当于在已建立的立方体的表面上贴”瓷砖”，所以，涉及到面的绘制，先来看看这段代码：（取自上面）

Java 代码：

```
gl.glColor4f(color[0], color[1], color[2], 1.0f);
DrawSquare(gl,new float[]{//z轴面
-0.48f,-0.48f,0.48f,
-0.48f,0.48f,0.48f,
0.48f,-0.48f,0.48f,
0.48f,0.48f,0.48f
});
```

代码分析：

这段代码的作用就是在立方体 Z 轴面绘制一个正方形并且设置颜色值为黄色。gl.glColor4f 的作用就是为图形设置颜色，而这里是给一个立方体的一个面绘制颜色，它的参数取自 float color[] 颜色数组：

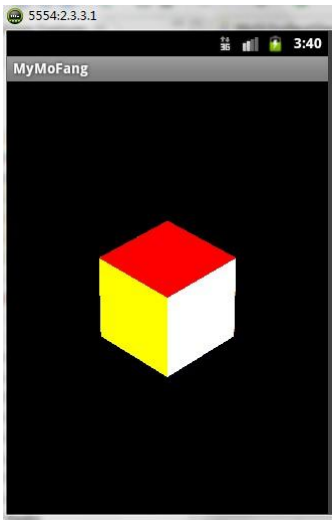
Java代码：

```
public float color[]={
1.0f,1.0f,0.0f , 1.0f,1.0f,0.0f , 1.0f,1.0f,0.0f , 1.0f,1.0f,0.0f , //    黄
0.0f,0.0f,1.0f , 0.0f,0.0f,1.0f , 0.0f,0.0f,1.0f , 0.0f,0.0f,1.0f,//    蓝
1.0f,0.0f,0.0f,   1.0f,0.0f,0.0f , 1.0f,0.0f,0.0f , 1.0f,0.0f,0.0f,//   红
1.0f,1.0f,1.0f , 1.0f,1.0f,1.0f , 1.0f,1.0f,1.0f , 1.0f,1.0f,1.0f ,//    白
0.0f,1.0f,0.0f , 0.0f,1.0f,0.0f , 0.0f,1.0f,0.0f , 0.0f,1.0f,0.0f ,//    绿
1.0f,0.48f,0.0f , 1.0f,0.48f,0.0f , 1.0f,0.48f,0.0f , 1.0f,0.48f,0.0f , //   橘黄
};
```

里面的值为 3 个一组，分别为 R，G，B。每一种颜色需要 4 组（二阶魔方 6 个面，每个面由四个立方体的其中一个面构成）。DrawSquare(GL10 gl,float myfloat[])的作用为绘制一个面，因为是以 GL_TRIANGLE_STRIP 来绘制的，所以参数 myfloat[] 为面的位置。(注意，这里一定要计算清楚是在哪个立方体的哪个面绘制正方形的)。

其余的各个面也是以此类推。好，到这一步的话呢，如果操作正确，你应该看到的是这

样的立体图：



4.3 动画的绘制：

动画的绘制分为两种：第一种就是旋转，第二种就是颜色的变化。

4.3.1 首先先来讲解旋转，先看如下代码：

代码 1：

```
switch (move){
case 1:                          //前面随z轴旋转
DrawCube5(gl); DrawCube6(gl); DrawCube7(gl); DrawCube8(gl);
gl.glPushMatrix();//保护变换矩阵现场
gl.glRotatef(-angle,0.0f,0.0f,1.0f);              //z轴(0,0,1)
DrawCube1(gl); DrawCube2(gl); DrawCube3(gl); DrawCube4(gl);
gl.glPopMatrix();//恢复变换矩阵现场
break;
……
```

代码2：

```
private void DrawAnimation(){
.....
    myrenderer.angle=0;                          //初始化角度为0
    while(myrenderer.angle<85){   //绘制旋转效果
    myrenderer.angle=myrenderer.angle+5;
    try{
    Thread.sleep(20);     //20毫秒绘制一次，一次5度
    }
```

```
        catch(InterruptedException e){
        }
        requestRender();       //绘制
        }
        myrenderer.angle=0;
.....
        }
```

代码分析：

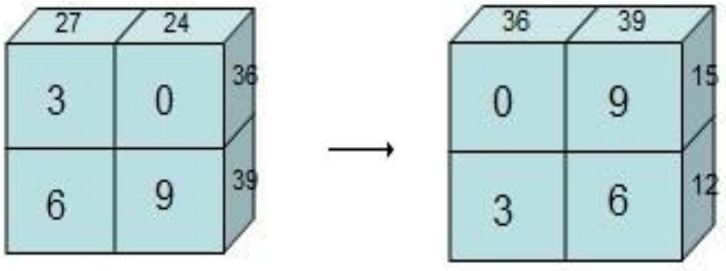
这段代码的作用是先绘制 5,6,7,8,立方体，然后随着 Z 轴坐标旋转-angle 度，然后再绘制 1,2,3,4,立方体，由于 angle 是从 0 递增到 90 度的。所以 1,2,3,4 也就旋转了-90 度，并且旋转时间为(90/5)*20ms。这样旋转的效果就出来了。其余的 5 个面也是以此类推。

4.3.2 颜色的变化：

```
public void colorChange(){
switch(myrenderer.move){     move的值表示哪个面
case 1:                  //前面      这里的顺序与MyRenderer.DrawRotation()   1里面的角度有关
shitf(0,3,6,9);
shitf(24,48,12,39);
shitf(36,27,51,15);
break;
.....}
}.....
//颜色的变换顺序：a->d->c->b->a
public void shitf(int a,int b,int c,int d){
float save1 = myrenderer.color[a];        //储存a的颜色R
float save2 = myrenderer.color[a+1];      //G
float save3 = myrenderer.color[a+2];      //B
change(a,b);          //b的颜色覆盖掉a的颜色
change(b,c);          //c的颜色覆盖掉a的颜色
change(c,d);         //d的颜色覆盖掉c的颜色
 myrenderer.color[d]=save1;        //将原本a的颜色赋值给d
 myrenderer.color[d+1]=save2;
 myrenderer.color[d+2]=save3;
}
//将e的颜色改为f的颜色
public void change(int e,int f){
myrenderer.color[e]=myrenderer.color[f];
myrenderer.color[e+1]=myrenderer.color[f+1];
myrenderer.color[e+2]=myrenderer.color[f+2];
}
```

代码分析：

colorChange()函数是改变颜色，因为每一次转动，一个立方体需要改变的是三个面的颜色，所以里面的每一个 case 都调用了 3 个 shitf()函数。Shitf 的参数对应的是颜色数组 color 的下标，并且是 3 的倍数。看图：



改变前　　改变后

4.4 触摸效果

这里使用了 GestureDetector 类，也就是手势识别类。用到了 onfiling()方法识别来识别用户移动的方向。因为要监听触摸屏的触摸事件和手势时间，所以在 MyGLSurfaceView 类需要实现 OnTouchListener 和 OnGestureListener 两个接口，并重写其中的方法。

核心代码：

```
public boolean onFling(MotionEvent e1, MotionEvent e2, float velocityX,float velocityY) {
//当用户 down 和 up 时的点的坐标的范围在屏幕坐标 y>170 和 y<305（也就是用户触摸到魔方，ratex 和 ratey 为不同屏幕的分辨率的比值）。
if(e1.getY()>170*ratey&&e2.getY()<305*ratey&&e2.getY()>170*ratey&&e2.getY()<305*ratey){
    //70-111   当 down 时 x 坐标在屏幕坐标轴 70-111 之间,先定位在最左边
if(e1.getX()>70*ratex&&e1.getX()<111*ratex){
    //这个 if 是根据 up 时 x 点的坐标来判断是竖滑还是横滑
if(e2.getX()>70*ratex&&e2.getX()<111*ratex){   //竖滑
    //这里是判断是向上滑还是向下滑
if(e2.getY()-e1.getY()>0){       //向上(滑)左面逆时针
 myrenderer.action=2;
 myrenderer.move=10;
DrawAnimation();
            }
    if(e2.getY()-e1.getY()<0){         //向下(滑)左面顺时针
 myrenderer.action=2;
 myrenderer.move=4;
 DrawAnimation();
 }
}else{        //横滑
    //判断是上层滑还是下层滑，这里都是往右滑动
```

```
  if(e1.getY()<255*ratey&&e2.getX()>111*ratex){ //上面 逆时针转
     myrenderer.action=2;
     myrenderer.move=5;
  DrawAnimation();
 }
  if(e1.getY()>255*ratey&&e2.getX()>111*ratex){//下面 逆时针转
     myrenderer.action=2;
     myrenderer.move=6;
     DrawAnimation();
                          ……..
```

（以此类推，自己水平问题，总感觉这里挺模糊的，写的也不好，不懂的可以和我交流。）
代码分析：（分析基本在上面了）

这里是根据屏幕坐标取点定位来判断相应动作，然后调用 DrawAnimation()函数，进行绘制效果，由于代码量太大，详细的源码上有详细的解释，所以在这就不解释了。
到此为止，二阶魔方基本的实现就制作好了。还有很多功能都没做，比如说：触摸还有待改进，使用纹理….想做的朋友就可以逐步的去实现。这里只是给初学者讲解一下最基本的实现。

五、基于 Rajawali 的框架的 3D 壁纸模型

By n0tify

承蒙 eoe 社区的厚爱，邀请我在这一期特刊中给大家介绍一个 3D 壁纸的模型。刚好趁着周末，就做一下。本来是说叫我做关于 OpenGL 的一个模型介绍给大家，但是向来比较懒的我就直接拿一个开源的框架做了。

其实关于 android 的 3D 框架有很多，也不必拘泥于一种，多了解一下也是比较好。好了废话也不多说，下面就是效果图。

图 1：

3Ddemos 就是我们要展示的壁纸。

图 2：

一个简单的地球模型。

可能有大家对于这个框架还不是太了解，不过没关系，更多内容都在作者官网上，而且作者也发布了关于这个框架的所有课程。所以学起来也不是很难。最大的感触就是对于 3D 的制作变得非常简单。像我这种第一次接触 3D 引擎的人也很快的上手了。

不过国内好像都没人介绍这框架的文章。对于这款框架的学习，基本来自作者的 bolg 和 Demos。

下面先介绍一下 Rajawali 的设计流程。

1.导入工程所需的 lib 和 so 文件。

2.在项目中创建一个类。继承 Rajawali 的 Renderer 父类。

3.下面就开始构建模型了。关键的几点就是要有一个模型类。导入 obj 文件。然后要设置视角和光线，把光线加入模型中。ok 基本完成了。

这样我们呈现的就是一个静态的 3D 实景图,这是基本的设计思路。

今天就教大家如何开发属于自己的动态壁纸，那么大家跟着我一起做就可以了。

首先是创建工程，然后把 Rajawalilib 包里所有文件复制到工程的 libs 包里面就可以了。接下来创建一个名为 Renderer 的类

```
public class Renderer extends RajawaliRenderer
```

默认要实现构造方法。然后给里面添加一个 setFrameRate 方法，用来设置刷新频率。

```
public Renderer(Context context) {
        super(context);
        setFrameRate(30);
```

```
        }
```

下面要重写 onSurfaceChanged 的方法，该方法里面就是我们要显示内容，也就是我们构造的一个 3D 环境。

```
public void onSurfaceChanged(GL10 gl, int width, int height) {
//首先要创建一个光源，并且设置它的强度。
DirectionalLight mLight = new DirectionalLight(0.1f, 0.2f, 1.0f);
    mLight.setPower(1.5f);
//然后设置得到一张图片。用于模型的表面的显示。
    Bitmap bg = BitmapFactory.decodeResource(mContext.getResources(),
                        R.drawable.earthtruecolor_nasa_big);
    //接下来是创建一个球体。
            Sphere mSphere = new Sphere(1, 12, 12);
    //创建一个纹理。
            DiffuseMaterial material = new DiffuseMaterial();
/*在这里我要说明一下，如果对于3D模型比较熟悉就不难理解上面为什么要创建的3个对象。因为一个3D模型的创建，包含3个要素，模型，纹理，和贴图。只有满足这三个要素才能组成一个完整的3D模型,这里详细的解释下，模型就是一个实体，它可以绘制，也可以用外部导入到obj文件，当然，大多数情况下都是用的外部的模型，然后说纹理，也可以理解成为材质，不同纹理能体现出不同的效果，这个可以参照维基百科纹理的解释。贴图就更好理解了，可以当作外表。它是覆盖在模型上的。如果你还不理解，可以自己修改这个demo，来体味一下。*/

//下面是给这个球体加纹理。
            mSphere.setMaterial(material);
    //下面是给这个球体加光源。
    mSphere.addLight(mLight);
    //下面是给这个球体加贴图。
    mSphere.addTexture(mTextureManager.addTexture(bg));
    //然后添加这个模型，只有添加才能显示出来。其实就是设置显示的意思。
    addChild(mSphere);
    //然后设置摄像头，也就是我们视角。
            mCamera.setZ(-4.2f);
            super.onSurfaceChanged(gl, width, height);
        }
```

现在一个完整的地球模型就建立好了，但是注意，这是一个静态的模型，我们要让它转起来。

重写 onDrawFrame 方法。

```
    public void onDrawFrame(GL10 glUnused) {
            super.onDrawFrame(glUnused);
        //这里设置地球自转。
    mSphere.setRotY(mSphere.getRotY() + 1);
        }
```

回过头来，我们在创建一个 Main 类继承 Wallpaper。

```
public class Main extends Wallpaper
//创建一个 Renderer 的对象。
private Renderer mRenderer;
//然后重写 onCreateEngine 的方法。
mRenderer = new Renderer(this);
//返回 WallpaperEngine 的方法。
Return new
WallpaperEngine(this.getSharedPreferences(SHARED_PREFS_NAME,
Context.MODE_PRIVATE), getBaseContext(), mRenderer, false);
```

至此代码的部分已经结束了。

然后我们还有两个东西没做，一个是设置壁纸启动和设置壁纸的画面，一个是设置 Manifest 来启动壁纸服务。

下面我们先来设置设置壁纸的界面，效果就是图 1 显示的。

新建一个 xml 包，然后在新建一个 wallpaper 的 xml 文件。

内容如下：

```
<wallpaper xmlns:android="http://schemas.android.com/apk/res/android"
    android:author="@string/app_name"
    android:description="@string/hello_world"
android:thumbnail="@drawable/ic_launcher" />
```

三个参数依次是作者名，描述，和显示的图标。

最后我们打开 Manifest。

添加壁纸的权限和 OpenGL 的声明。

```
<uses-feature android:glEsVersion="0x00020000" />
<uses-feature android:name="android.software.live_wallpaper" />
```

然后就是设置启动 server。

```
<service
    android:name=".Main"
    android:label="@string/app_name"
    android:permission="android.permission.BIND_WALLPAPER" >
    <intent-filter>
 <action android:name="android.service.wallpaper.WallpaperService" />
    </intent-filter>
     <meta-data
                android:name="android.service.wallpaper"
                android:resource="@xml/wallpaper"/>
</service>
```

Ok，所有的项目都已经完成，可以运行测试一下，

我们可以看一下测试的结果。

下面说一下开发中关于 Rajawali 的几点注意事项：

1.在加载 obj 文件时候有两种方法。在加载 3dMax 导出的 obj 会出现问题，模型必须由

blender 导出才能使用。

2.Rajawali 建立的模型坐标系可能和你导出 obj 的模型不一样。

3.Rajawali 在做优化时候会导出 ser 流文件。相当于用这个流文件代替 obj，大大节省了内存，但是在生成 ser 文件的时候，obj 里只能有一个模型，两个以上将会出错。解决的方法：一是按照原则一个 obj 里只带一个模型。二是在 Rajawali 的讨论区中有大牛给出了代码解决方案。有兴趣的可以去看看。

4.在一些实际测试时候会出现一些莫名其妙的问题，这极有可能是你手机内存或者模拟器的原因。

其实大家有没有发现，这一个简单的小壁纸代码量很少，这是因为 Rajawali 封装了大量的 openGL 的相关函数，在国外已经有很多开发者尝试使用 Rajawali 开发，这个框架尤其适合 3D 壁纸的开发，相信以后 3D 壁纸也会渐渐成为主流。

相关资源已经打包成 demo，大家可以去 eoe 上面下载。

感谢 eoe 能给我这次机会，还有感谢小朱同学借我手机测试，特别感谢张翼教会我很多知识。希望大家多去 eoe 上逛逛，最后祝大家学习开心，钞票多多。O(∩_∩)o

eoeAndroid开发者社区

责任编辑：jiayouhe123

美术支持：徐士勇　　技术支持：郝留有