**毕业论文**

**Android平台基于openGL的图形渲染与实现**

**吉林大学**

**李志启**

**计算机科学与技术学院**

**摘要**

目录

1. 绪论

1.1 课题背景及研究的目的和意义

1.2 课题相关国内外研究状况

1.2.1 Android 的国内外研究状况

1.2.2 OpenGL 的国内外研究状况

1. Android 体系结构

2.1 android 系统架构简介

2.2 android 四大基本组件

2.3 avtivity 生命周期

2.4 activity 启动模式

2.5 android 消息分发机制简介

1. OpenGL 与 计算机图形原理

3.1 基本几何图形绘制

3.2 立方体的三维建模

3.3 单张3D 旋转效果

3.4 多张复合三维效果

3.5 本章小结

1. 基于OpenGL 全景图像展示

4.1 基本原理

4.2 构建3D球体

4.3 加载图像纹理

4.4 交互设计

1. OpenGL 建模测试

5.1 测试背景及意义

5.2 如何利用工具测试

5.3 如何编写用例测试程序

1. 结论

参考文献

致谢

1. 绪论
2. Android体系结构

2.1 android系统架构简介

Android是一种基于[Linux](http://baike.baidu.com/view/1634.htm" \t "http://baike.baidu.com/_blank)的自由及开放源代码的[操作系统](http://baike.baidu.com/view/880.htm" \t "http://baike.baidu.com/_blank)，主要使用于[移动设备](http://baike.baidu.com/view/8323830.htm" \t "http://baike.baidu.com/_blank)，如[智能手机](http://baike.baidu.com/view/535.htm" \t "http://baike.baidu.com/_blank)和[平板电脑](http://baike.baidu.com/view/74538.htm" \t "http://baike.baidu.com/_blank)，由[Google](http://baike.baidu.com/view/105.htm" \t "http://baike.baidu.com/_blank)公司和[开放手机联盟](http://baike.baidu.com/view/1245202.htm" \t "http://baike.baidu.com/_blank)领导及开发。

Android的系统采用分层结构，从高到低分为四层，依次为应用程序层，应用框架层，系统运行库层和linux 内核层。

2.2 Android四大基本组件

说到四大组件，对于android 开发者来说再熟悉不过了，分别是Activity ,Service,BroadcastReceiver和ContentProvider。

Activity 是一种展示型组件，用于界面的展示以及人机交互。是android 开发中最重要的组件。

Service 是一种计算型组件，用于在后台执行一系列的计算任务。其分为两种：一种是非绑定的service;另一种是跟activity 绑定的，其生命周期跟相应的activity关联起来。

BroadcastReceiver 是一种消息型组件，用于不同组件甚至不用应用间的消息传递，在应用开发中可以使用它来解耦程序。BroadcastReceiver没有生命周期这一概念。

ContentProvider 是一种数据共享型组件，用于组件或者应用间的数据共享，其主要跟本地数据库相关联，也没有生命周期这一概念。

2.3 Activity生命周期

Activity是android开发最重要的组件，充分理解activity的生命周期是每个android开发者必须掌握的基本技能，也为以后开发打下坚实的基础。

如图所示，在一般正常情况下，activity 会经历一下生命周期过程：

1. OnCreate: 在activity 创建时会调用该方法

我们需要在activity 创建时初始化很多对象。例如在后面章节提到的Glserfaceview ，渲染器等

1. OnRestart:表示activity 正在重启
2. OnStart:表示activity正在启动
3. onResume:表示activity已经可见，并显示在前台。注意其跟onStart的区别。onResume和onStart都示avtivity 可见，但onStart的时候表示activity 还在后台，onResume的时候avtivity 已经显示到前台了。
4. onPause:表示avtivity 正在停止
5. onStop: 表示avtivity 即将停止

在activity 停止时我需要停止与该avtivity相关的所有内容，否者可能会出现异常，导致程序崩溃。

1. onDestroy: 表示activity即将被销毁

在activity被销毁的时候，我们要主动回收所有相关的内存，特别是在openGL 编程中，会使用直接内存，不受java 内存的回收机制的处理。这部分内存若不能及时手动回收，会导致内存泄漏甚至内存溢出。

2.4 activity 启动模式

在android 应用程序中，Activity 可以简单理解为对应每个页面，每次在启动activity时，会创建activity的实例，然后放入任务栈中，当我们点击back 回退键时，任务栈就会出栈，直到为空。开发者会根据不同的业务需求制定不同的启动模式。Activity 共有四种启动模式（launchMode）。

1. Standard: 标准模式，每次启动activity 都会创建实例，放入当前任务栈中。谁启动了这个activity，那么activity 就会运行在启动它的activity所在的任务栈中。
2. SingleTop:栈顶复用模式，如果新启动的activity 在任务栈栈顶，就不会重新创建新的实例，而是直接复用栈顶的实例。
3. SingleTask: 栈内复用模式，如果新启动的实例在栈中存在，则复用这个实例，并弹出该实例上面的所有实例，让该实例置于栈顶。否则创建新的实例。
4. SingleInstance:单实例模式，当启动新activity 会创建一个新的任务在，然后把activity 实例押入栈中。以后就一直复用这个实例，该栈中也一直只保存一个实例，直到被销毁。

在本论文中，我们可以通过设置mainActivity的luanchMode 为singleTask ,这样避免在测试中反复启动activity而重复创建实例，造成内存空间浪费，影响测试结果。

2.5 Activity 消息分发机制简介

本文第四章会涉及android 人机交互，尽管android官方就有像onclick() 等基本交互接口框架，但是在基于openGL的开发中，这些基本框架是满足不了所有需求的。令人可喜的是，android系统是基于事件分发设计的，人机交互事件也是如此。这里介绍一下android消息事件分发机制，理解透彻后，在以后的开发中即使没有现成的API ,我们也能轻松地处理各种自定义的事件交互。

2.3.1 handler

Handler 是android 中的一种处理线程中消息循环的机制。主要涉及thread、MessageQueue、Looper 。

如图所示：

Thread

Looper

MessageQueue

Handler

其中MessageQueue是消息队列，Looper 是消息循环，handler 可以理解为消息处理器，他们都是构建在thread 上。我们用一张图来更清楚地理解他们之间的关系。

Handler.send(Message)

handler.dispatchMessage

thread

Looper

Message

messageQueue

我们把android 中消息队列看成传送带，把每个消息看成是传送带上传送的一个货物，把Looper看成是传送带电机，把线程看成是电源。Handler 作为消息的处理者，他通过send 方法向消息队列不断发送消息，而在传送带的另一端，handler 又通过dispatch 方法将消息从消息队列中不断的取出，分发给其他组件。这样整个消息的分发机制就这样运作了起来。

那么问题来了：为什么不直接将消息给需要他的组件，而不是搞个消息队列这么大费周章的处理消息？

原因是这样的：①. android的消息的发生很难由开发者控制，比如点击事件的消息，对于开发者来说根本不能预先知道用户做出来了怎样的点击事件；②. 还有如果消息数量很多，并且是高并发，这时系统就没法处理所有消息了；③. 假如我们想要的消息有延迟，比如网络请求，我们不可能一直等待消息的到来，这是就需要单独开一个子线程来处理，我们就可以先做其他事情了。

以上3种情况我们都可以使用android的这种消息分发机制解决问题。他具有良好的可控制性，效率也非常高效。相当于把消息都加入到一个缓存队列里面，按需慢慢来处理消息。

1. OpenGL ES for android

3.1 初识OpenGL ES

正如本文第二章所述，在android系统架构层次中的系统运行库层中，专门用于图像处理的是系统运行库是OpenGL ES。OpenGL（全写Open Graphics Library）是个定义了一个跨编程语言、[跨平台](http://baike.baidu.com/view/469855.htm" \t "http://baike.baidu.com/_blank)的[编程接口](http://baike.baidu.com/view/897136.htm" \t "http://baike.baidu.com/_blank)的规格，它用于三维图象（二维的亦可）。OpenGL是个专业的图形程序接口，是一个功能强大，调用方便的开源的底层图形库。而OpenGL ES (OpenGL for Embedded Systems) 是 [OpenGL](http://baike.baidu.com/view/9222.htm" \t "http://baike.baidu.com/_blank)三维图形 API 的子集，针对手机、PDA和游戏主机等[嵌入式设备](http://baike.baidu.com/view/2778983.htm" \t "http://baike.baidu.com/_blank)而设计。

OpenGL ES 有诸多版本。版本1.0提供了一个不够灵活的、固定功能的管道。版本2.0 则推出了可编程的管道，可一解决任何图形学问题。版本3.0 在2.0 的基础上添加了一些新的特性，但是还未被广泛使用。本文全篇均以OpenGL 2.0 为准，编程与语言为java。

3.2 顶点与着色器

3.2.1 点、直线、三角形

首先介绍一下OpenGL中的基本概念。在OpenGL中，只能绘制点、直线、三角形。

三角形是最基本的几个图形，它有如此稳定的结构，并且在生活中随处可见。点和直线可以用于某些效果，但只有三角形才能用来构建复杂的对象和纹理场景。

在OpenGL中，我们把单独的点放在一个组里构建出一个三角形，再告诉OpenGL 怎样连接这些点。我们想要构建的所有东西都要使用点、直线、三角形来定义。如果需要构建更复杂的对象，就需要有足够的点来拟合对象中的曲线（在定义三角形的时候，约定以逆时针的顺序排列顶点）。

在openGL 开发中，图形的基本元素数据都必须使用直接内存。并且综合考虑图像的精度以及程序效率，我们均以浮点数组来存储顶点数据。

float[] vertexData = ByteBuffer  
 .*allocateDirect*(rawData.length \* *BYTES\_PER\_FLOAT*)  
 .order(ByteOrder.*nativeOrder*())  
 .asFloatBuffer();

使用该语句为顶点数组vertexData分配内存空间，其中rawData.length 为原始数据长度，BYTES\_PER\_FLOAT 为java中浮点数的字节长度。在分配直接内存中，均是先分配以字节为基本单位的空间，再进行处理。

vertexData.put(rawData);  
vertexData.position(0);

当空间分配完成后，使用以上语句往该内存块中写入数据，并让索引指向当前空间的开始位置。

3.2.2 着色器介绍

在把定义的对象画到屏幕上之前，需要找OpenGL的管道中传递，这里就需要使用着色器（shader），它会告示GPU 如何绘制数据。用两种shader:

1. 顶点着色器（vertex shader）生成每个顶点的最终位置
2. 片段着色器（fragment shader）为图像基本元素的每个片段生成最终的颜色，一个片段类似于计算机屏幕上的一个像素。

当最终的颜色生成后，OpenGL就会把他们写到帧缓冲区（frame buffer 内存的某部分）中，接着android就会把这个帧缓冲区显示在屏幕上。

基本流程如下：

读取顶点数据-->执行顶点着色器-->组装图元-->光栅化图元-->执行片段着色器-->写入帧缓冲区-->显示在屏幕上

1. GLSL (OpenGL Shading Language),用来在openGL 中着色器编程的语言。其在GPU中运行，代替固定的渲染管线的一部分，使渲染管线中不同层次中具有可编程性。在本文中，glsl作为编程辅助，只做简单简绍。

glsl 语法类似于c语言，有一个main() 函数，有uniform ,attribute等关键字，还有各种向量、矩阵等图形学类型，读者可以参考以下博客地址自行学习，这里不再赘述。http://my.oschina.net/sweetdark/blog/208024?fromerr=iCfFiZvJ

3.2.3 编译着色器

1. 加载着色器

我们将编写好的\*.glsl 着色器文件在资源文件的raw目录下，通过读取文件流的方式将文件内容读入字符串（shaderCode）中，具体实现可以查看本文项目源代码中的TexResReader.java 文件。

1. 创建着色器对象

final int shaderObjectId = *glCreateShader*(type);

其中type为shader类型：顶点着色器，片段着色器

1. 上传和编译着色器源代码

*glShaderSource*(shaderObjectId, shaderCode);  
 *glCompileShader*(shaderObjectId);

将着色器ID 和 对应的shader 代码关联起来，

然后调用API 进行编译，这样便得到了想要的shader

1. 验证编译状态并返回shader对象ID

3.2.4 链接进OpenGL 程序

1. OpenGL program 简介

这里指的是OpebGL 的一个组件，与一般的应用程序不同。一个OpenGL program就是把一个vertex shader 和一个fragment shader 链接在一起变成单个对象。他们两总是一起工作的。

1. 新建程序对象并附上着色器

final int programObjectId = *glCreateProgram*();

program 的创建与shader创建类似，返回对应的id

1. 链接程序

*glAttachShader*(programObjectId, vertexShaderId);  
 *glAttachShader*(programObjectId, fragmentShaderId);  
 *glLinkProgram*(programObjectId);

通过上述方法将创建好的顶点着色器和片段着色器与 对应的program关联起来。

以上过程实现可以查看本论文项目源代码ShaderHelper.java 文件

3.3 在屏幕上绘制

1. 使用创建的OpenGL 程序

glUseProgram(program);

1. 获取着色器相关属性

通过glGetUniformLocation(program,key),

glGetAttribLocation(program,key)

来获取 \*.glsl 中相关的属性（int 类型）。

1. 关联属性与顶点数据

我们到现在只是将相关属性获取了出来，进行进一步封装后，可以方便地操作对象的属性，进行各种变换。接下来便是给每个属性关联相应的实际数据。例如：

glVertexAttribPointer 关联顶点数组

glUniformMatrix4fv 关联矩阵数据...

诸如此类还有一下，读者可查看官方API,根据具体项目需求使用相关方法 。

1. 使能顶底数组

glEnableVertexAttribArray(...)

1. 绘制基本元素

glUniform4f(colorLocation, 1f, 1f, 1f, 1f)

调用该方法更新着色器中相应颜色的值，类似于android开发中设置画笔paint的颜色。

glDrawArrays(Gl\_TRIANGLES, 0, 6)

调用该方法画三角形，从关联的顶点数组的第0个元素，画6个顶点。其中参数可以根据需求更改：GL\_TRIANGLES\_FUN画三角扇形，GL\_TRIANGLES\_STRIP 画三角带，GL\_POINTS画点......

3.4 OpenGL三维世界

3.4.1坐标变换

在之前的图形渲染中，我们都使用了绝对坐标，这样面临复杂的图形变换中，会很不方便，容易出错，不方便调试。而且在遇到屏幕旋转或屏幕比例变化时，图像会出现拉伸或压缩。为了解决这个问题，我们可以使用虚拟坐标空间，进行正交投影操作即可。

在顶点着色器中定义 u\_Matrix ,更新gl\_Position=u\_Matrix \* a\_Position ,这样图像的顶点就和投影矩阵关联起来。接着我们在代码中获取矩阵属性。

*orthoM*(projectMatrix, 0, -aspectRatio, aspectRatio, -1f, 1f, -1f, 1f);

调用正交投影方法

*glUniformMatrix4fv*(matrixLocation, 1, false, matrix, 0);

传递矩阵给着色器

3.4.3 透视投影

在绘画上，艺术家常常通过透视的技巧将图像绘制在一张平面上，给人的感觉却是由远及近的真实感，以此来欺骗观众的眼睛。而在计算机显示方面，我们可以使用同样的方法将图像渲染在屏幕上，达到三维立体的效果。

我们把每个点定义成4维数据：x,y,z,w。 其中x,y,z 表示点在三维空间的位置，而 w 分量用于计算透视投影（透视除法）。我们要定义投影矩阵，他最重要的任务就是为w产生正确的值。

3.4.4 矩阵变换

*perspectiveM*(projectMatrix, 0, 45f, (float) w / (float) h, 1f, 10f);

调用此方法（android API )处理透视矩阵projectMatrix

为了验证透视是否成功，我们可以通过旋转图形来观察。创建模型视图矩阵modelMatrix,接下来就是进行矩阵运算。

*setIdentityM*(modelMatrix, 0);  
 *translateM*(modelMatrix, 0, 0f, 0f, -2f);

先设置单位矩阵，再旋转该矩阵

然后我们需要将变换后的矩阵关联到图像数据上，

*multiplyMM*(clipMatrix, 0, projectMatrix, 0, modelMatrix, 0);

clipMatrix是裁剪矩阵

protected void bindMatrix(int matrixLocation, float[] matrix) {  
 *glUniformMatrix4fv*(matrixLocation, 1, false, matrix, 0);  
 }

通过该方法将着色器中的矩阵属性uMatrixLocation 与 clipMatrix 关联起来，这样矩阵变换就投射都真正的图像上了。

3.5 触控交互

1. 对activity增加触控交互

在activity中，我们创建了glSurfaceView。对于这样的视图，我们可以设置触摸监听器（setOnTouchListener）来获取触摸事件消息，android消息分发机制在本文第二章已经介绍过了。不过获取的触摸事件是最原始的，我们不能直接用，需要做预处理。主要是计算触摸点在屏幕上移动的距离：

final float deltaX = motionEvent.getX() - previousX;  
 final float deltaY = motionEvent.getY() - previousY;  
 glSurfaceView.queueEvent(new Runnable() {  
 public void run() {  
 mRender.handleTouchDrag(deltaX, deltaY);  
 }  
 });

在相应渲染器中定义handleTouchDrag(),通过该方法将触摸点变化坐标传入渲染器中处理。这是一个事件处理过程，可以看成是消息分发过程，所以需使用queueEvent()将该消息加入消息队列。

1. 渲染器处理交互事件

在渲染器中，我们之前定义了handleTouchFrag()方法，在该方法中我们处理交互事件。还记得我们使用模型矩阵（modelMatrix）来专门处理图像的移动，旋转，缩放等图形变换。因此在该方法中，只需要通过根据传递进来的事件数据改变modelMatrix 即可（基本矩阵计算，详情参考项目源代码）。

1. 拖拽移动物体

有时候，我需要在屏幕上拖拽三维立体图像，但是android手机屏幕都是平面，我们如何将二维交互转化到三维空间？我们当前获取的是被触摸点的x，y 坐标，那么在三维空间中该点到底离我们有多远？

要解决上述问题，我们把被触摸的点映射到三维空间中的一条直线上。由于该过程涉及到一些比较复杂的数学运算，在这里只做简单的概念性介绍，读者若想深入了解，可以阅读《OpenGL for Android: A quick-start Guide 》第9章 内容。

基本步骤如下：

创建反转矩阵，将触摸的二维的点通过该矩阵反转映射到三维空间的直线上。

使用向量计算拖拽方向和拖拽距离。

通过上述直线计算三维空间中直线到目标对象的距离，以此判断是否相交，相交则拖拽该物体，不相交则不拖拽。

1. **OpenGL ES应用实战**

本章主要介绍如何在android手机上展示360度无死角全景图片，通过构建球形视野，再贴上全景纹理，即可虚拟出全景视图。

4.1 基本原理

全景图通过广角的手段尽可能多的表现出周围的环境。全景图可以呈现360度的视野，然而人的视野和相机的视野始终是有限的，那么全景图就怎样拍出来的呢？

基本原理是通过相机拍摄每个有限的视野，然后将所有照片合成在一张平面图上，在合成的过程中对相片的边缘进行处理，还要考虑不同视角合成在一张平面上时视角扭曲对图像的影响，即要进行投影处理。最要会得到一张比较的扭曲的平面图。之所以扭曲是因为我们观察的视角不一样，当把合成图渲染到特定的3D视野上时（例如 球形视野，立方体视野），就是给人全方位的立体感受。当相机拍摄的视野照片越多时，全景图越清晰，立体感越强烈。

拍摄全景图像需要特殊的相机以及特殊的图片拼接合成算法，如何制作全景图片不在本文讨论之内，本文只对如何将平面的全景图展示成立体的效果感兴趣。

下图为一张合成好的全景平面图：

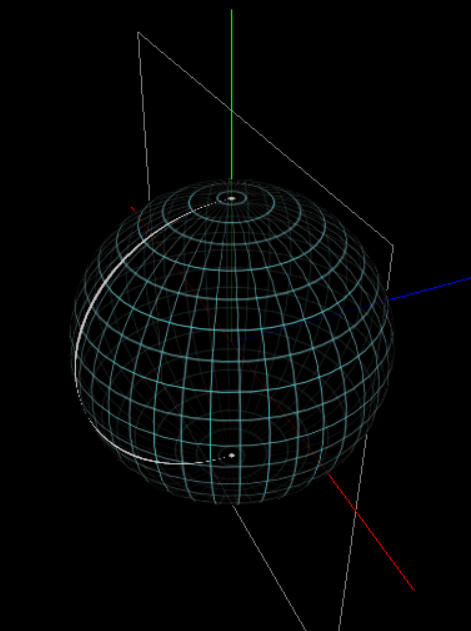


将图像纹理贴在3D球体上，就成的如下的效果：

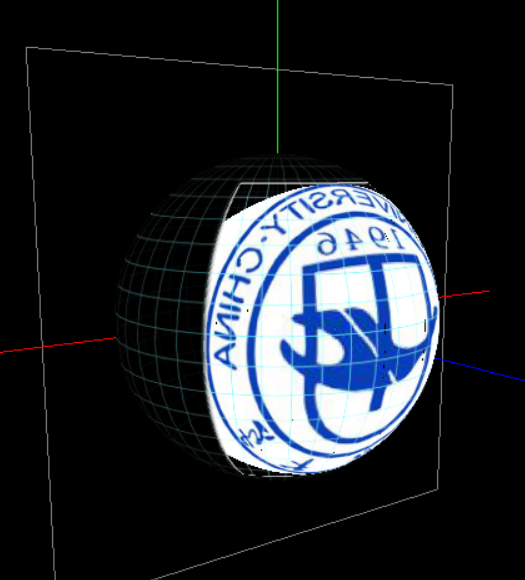


我们可以使用工具Hugin来演示一下：

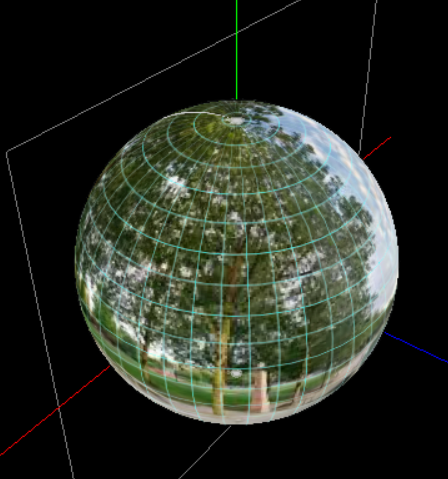
下图为没有贴图的球形视野：



接下来是贴上一张正常平面图的效果：



最后就是贴上整张全景的效果：



当转动球体时，我们就看到了立体的全景效果。

通过了解原理后，我们可以分为一下几个方面来逐步构架出我们想要的效果：构建3D球体，加载图像纹理并贴在球体上面，增加旋转等基本交互操作。

4.2 构建3D球体

在编写程序之前，我们要思考整个程序该如何组织才能更简洁、方便开发，甚至优雅，让人有赏心悦目的感觉。作为一名程序员，维护好程序的简单易用是一种基本的职业素养。对程度有一定的洁癖，把程序设计当做一种艺术来看，更是每个程序员走向职业巅峰所应该具备的优秀品质。本文内容基于OpenGL API,像这种底层API 面向范围非常广泛，在进行像本文这样范围较小的项目开发时，多数API基本无用，有用的少数API也无章法可寻，我需要根据业务需求在这些少数的API上构建出一套通用的开发设计模式，使程序底耦合，高内聚，并高度复用，减少冗余代码。

在这里，我们可以借鉴android 开发 MVC 设计框架，如下图所示：

Controller

Model

View

Model: 模型，用于关联数据，存储数据。

View: 视图，专门用于数据展示。

Controller: 控制器，具有数据计算，处理用户交互等控制，是Model和View之间的沟通桥梁。

在本章中，笔者将基本程序结构分为 Model,Program,Render。

Model : 其作为3D图形的基本数据结构，存储了图形的顶点数据、纹理顶点数据以及构建图形的数据结构。

Program : 在OpenGL 中，每个要绘制的图形必须对应一个着色器程序。我们对此进行封装，Program 包含了通过OpenGL API 将图形数据关联到OpenGL 上来,主要为着色器的处理。

Render :GLsurfaceView 每秒会绘制约60帧的图像，并不断地显示在屏幕上。通过给 GLsurfaceView 增加 Render 来跟踪视图生命周期，可以很方便地处理问题，渲染器作为视图的展示，以及图形的基本交互变换最为合适。其兼并 MVC 模式中View和Controller的作用。

开发基本流程：

在render的onSurfaceCreated 方法中 new 出model，program 对象,并通过TextureHelper加载纹理。

在render的onSurfaceChanged 方法中 初始化个矩阵，包括透视投影，模型矩阵，视图矩阵，初识化方向，尺寸等参数。

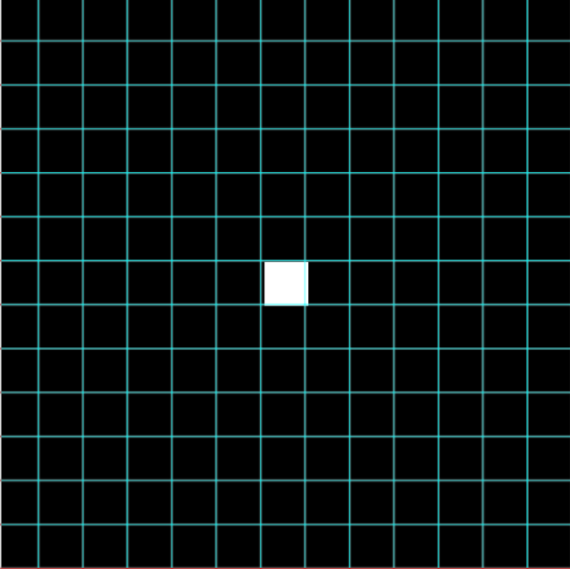
在render的onDrawFrame 方法中绘制每一帧图像，包括使能、设置数据、绑定着色器程序、调用绘制方法 ，为交互设置矩阵参数等，如下代码所示：

ballProgram.useProgram();  
ballProgram.setData(modelViewProjectMatrix, textureID);  
mBall.bindProgram(ballProgram);  
mBall.draw();

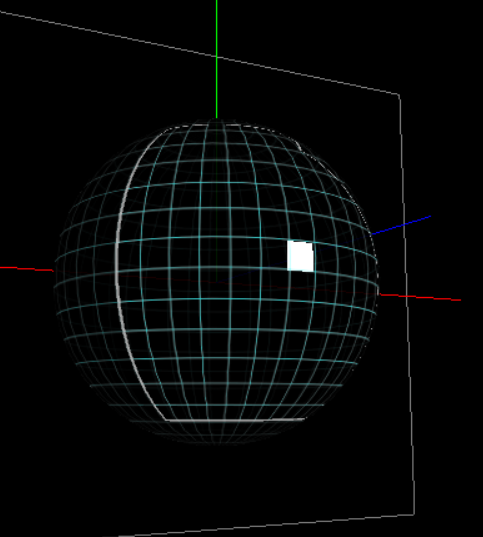
下面讲解构建3D球体的具体过程：

这里笔者用地球仪的一些概念来阐述一些基本参数。将整个球体（半径为1）经纬度分割成36等份，每等份的角度为 Θ=2Π/36 ,即 Θ=Π/18。

在BallModel 中，声明了两个浮点数组：纹理坐标数组，顶点坐标数组。将纹理经纬方向各等分成36份，共36\*36个小矩形，纹理数组记录了每个矩形的纹理坐标。如下图纹理被等分：



同样的，在球面上，经纬方向被分别等角度分割成36份，每角度即Θ，球面被分割为36\*36个小块，顶点坐标数组记录了每个小块四个顶点的坐标 。顶点坐标可以根据角度通过三角变换计算出来。如下图球面等角度分：



事实上每个小块面我们绘制的是平面，纹理渲染也是在平面上。最后绘制的球体并完全是球，当被分割份数越多时，越接近球体。我们用这种方式来拟合曲面。

这里要注意在openGL 中是不能直接绘制矩形的，我们将矩形分为两个三角形来绘制，在构造相关的浮点数组时应该注意这一点。

4.3 着色器程序构建

着色器程序program需要先加载解析glsl着色器代码:

String res = TextResReader.*readTextFileFromRes*(context, id);

然后对代码进行编译：

public static int compileShader(int type, String shaderCode) {  
 final int shaderObjectId = *glCreateShader*(type);  
 *glShaderSource*(shaderObjectId, shaderCode);  
 *glCompileShader*(shaderObjectId);  
 return shaderObjectId;  
 }

最后将顶点着色器和片段着色器进行链接即可获得program：

public static int linkProgram(int vertexShaderId, int fragmentShaderId) {  
 final int programObjectId = *glCreateProgram*();  
 *glAttachShader*(programObjectId, vertexShaderId);  
 *glAttachShader*(programObjectId, fragmentShaderId);  
 *glLinkProgram*(programObjectId);  
 return programObjectId;  
}

过程十分繁琐，但幸运的是，这些过程都是可以设计成高度复用的，笔者只需将其写好在ShaderHelper工具类中，以后便可以直接调用，不必关心其实现细节，让开发者将更多的精力放在如何处理业务逻辑上。这里就体现了一个结构优秀的程序给开发者带来很大的好处，并且显著提交项目开发效率。在计算机行业中，特别是互联网行业，瞬息万变，工作效率影响非常巨大。

4.4 加载图像纹理

图像纹理作为一种数据资源，我们完全可以单独作为一个模块来处理，最终目的就是为其他模块提供数据，这样做可操作性强大，即插即用。同样，我们只用实现一次，便可以永久复用该加载模块，因此将其实现在TextureHelper 工具类中，

public static int loadBallMap(Context context, int drawableId)// textureId{  
 int[] textures = new int[1];  
 *glGenTextures*(1, textures, 0);  
 *glBindTexture*(*GL\_TEXTURE\_2D*, textureId);  
 *glTexParameterf*(*GL\_TEXTURE\_2D*, *GL\_TEXTURE\_MIN\_FILTER*, *GL\_NEAREST*);  
 。

。

。

InputStream is = context.getResources().openRawResource(drawableId);  
 Bitmap bitmapTmp = BitmapFactory.*decodeStream*(is);  
 GLUtils.*texImage2D*(*GL\_TEXTURE\_2D*, 0, bitmapTmp, 0);  
 bitmapTmp.recycle();  
 return textureId;  
}

先创建Texture对象，然后设置相关参数，接着资源解析成位图，方便texture加载，这里是直接拷贝进texture中。在最后，当texture已经加载完位图数据，我们要手动回收该位图。在手机这种内存如金的设备上，要时刻关注内存使用情况，特别是在遇到像图片这样的大对象时，稍不注意就会引发内存泄漏，内存溢出，导致程序崩溃。虽然java有自动回收机制，但像这种情况，即便是java GC 也吃不消，及时回收内存是一个良好的习惯。

到现在，我们可以进行模块关联了。按照本章第2节的流程，先创建ballModel,ballProgram,ballTexture 对象，然后再在BallRender的onDrawFrame 方法中绘制。调用ballProgram.useProgram()使能；

调用ballProgam.setData(matrix,ballTexture) 为着色程序绑定矩阵和纹理数据；接下来调用ballModel.bindProgram(...)给模型绑定着色器程序；最后调用ballModel.draw() 进行真正的绘图。

至此，绘图部分已经完成。

4.5 交互设计

上一节完成了图像的绘制，但这样还不能得到们想要的效果，我们需要考虑球体在屏幕上显示也是一个平面图，如何让其看起来有立体效果，就需要用到前一章介绍的投影技术。定义projectMatrix 为投影矩阵。

投影后看起来会有立体效果，这些都是通过技术加数学运算得到的虚拟的图像。假如现实生活中有一个物体，我们站在一点看到的是一个立体的效果，对于同样一个物体，我们换一个地点看到的是不同的立体效果，这即是视角转变引发的视觉效果转变的问题。我们同样可以用数学运算加技术的手段来解决这样的问题，定义viewMatirx 为视图矩阵。

在全景图交互中， 我们需要对图像进行旋转，以观察完整全景图。定义modelMatrix 为模型矩阵，专门用于图像的旋转、缩放、平移等基本变换。

最终需要将所有变换合并，在数学上，只需按顺序相乘矩阵。如下所示：

**resultMatirx=projectMatrix \* viewMatrix \* modelMatrix**

得到的结果矩阵通过上一节所述ballProgam.setData(resultMatirx,

ballTexture) 语句与着色器程序中对应的4维矩阵的位置绑定起来。

接下来在ballRender的onSurfaceChanged方法中调用

*frustumM*(projectMatrix, 0, -ratio, ratio, -1, 1, 1, 20);

进行锥视图投影。当然，用什么投影效果可以根据需要自行设置。这里使用该投影方面全景图查看。

在本项目中，可以不用设置视角即可实现效果，所以讲viewMatrix设置成单位矩阵即可，调用: *setIdentityM*(viewMatrix, 0);

此时运行程序可以看到一个静态画面，他是平面全景图的一部分，但好像发生了一点扭曲：



我们距离成功已经很近了，此时全景图已经完整地绘制了出来， 现在要做的就是旋转视图查看全景图其他部分。

在onDrawFrame方法开头中添加：

*setIdentityM*(modelMatrix, 0);  
 *rotateM*(modelMatrix, 0, -yRotation, 1f, 0f, 0f);  
 *rotateM*(modelMatrix, 0, -xRotation, 0f, 1f, 0f);

旋转是连续性操作，并不是一次就能绘制好，所以需要在onDrawFrame 中进行。对于这样连续性操作都必须到该方法中处理，这样理解读者就会更容易明白render 三个实现方法的职能。类似的操作还包括缩放，平移等。

在调用rotateM( )方法时，我们对modelMatrix 设置了两个可变参数：yRotation,xRotation。分别表示图像在坐标系中 Y 轴方向 和 X 轴方向 旋转的角度。通过改变其值来达到旋转的目的。

**关联交互事件：**在avtivity中对glSurfaceView 调用setOnTouch()获取触摸事件，对于触摸事件本文在第二章已经简单介绍过了。我被要对基本交互事件做一层封装，以满足业务需求。

* 拖拽滑动事件：

声明两个浮点数，用于记录前一次按下时的位置：preX,preY。在滑动时，计算当前触摸点位置坐标与上一次位置的坐标的变化量(deltaX,

deltaY)，该变化量即是滑动的距离，调用render.handleTouchDrag(

deltaX,deltaY) 将滑动变化量传递给渲染器进行处理。最后更新preX,preY 为当前触摸点坐标位置，这样就是可以不断的将滑动事件传递下去。

* 双点触摸滑动事件：

该事件主要为以后通过滑动改变两个触摸点之间的距离来改变图像的scale而设计的。

通过event.getPointerCount() 方法来获取屏幕上触摸点的个数。其中event.getX(0) 为第0个触摸点的X 坐标，以此类推。当触摸点个数为两个或者两个以上,并且在ACTION\_MOVE 状态时，通过勾股定理计算当前两点间的距离：

currentDistance = (float) Math.*sqrt*(offsetX \* offsetX + offsetY \* offsetY);

接下来判断当前距离与上次距离的差值，这里当差值大于5时判断为扩大操作（反之则为缩小操作），然后调用render.upScale() 来处理放大事件。最好要将lastDistance 更新为currentDistance的值，这样才会在滑动过程中不断的触发该事件。

if (currentDistance - lastDistance > 5) {  
 surfaceView.queueEvent(new Runnable() {  
 public void run() {  
 render.upScale();  
 }  
 });  
 lastDistance = currentDistance;  
 }

为什么当前距离要和上次距离差值为5 ? 在手机上，手机屏幕是很敏感的，非常微小的距离都可以被认为是滑动，但是对人本身来时，这种滑动是感觉不到的。为了适应人的习惯，将差值设为5，减小其灵敏度，这样用户体验会更好。当然设置成其他值也可以，但不要太离谱。

至于调用queueEvent 方法，是为了消息分发，在第二章已经介绍过了，这里不再赘述。

* 双点击缩放事件：

前面所述交互事件在本文项目中足够满足需求，但从用户的角度来讲，我希望用户体验更好一点，增加一个双点击事件来一次性放大或者缩小图像到一个固定尺寸。程序员在开发过程中，多从产品的角度去看的问题，有益于扩宽思维视野，并能提高开发产品质量。

为捕获该事件，可以增加一个整型变量来记录点击次数，增加两个long 变量分别记录两次点击的时间。然后比较时间间隔，当间隔事件小于某一个值时，可以看做是一个双点击事件。最后记得将前面的三个变量全部清理零，这样才能循环使用。

要注意该事件在实现的过程中可能会与前面所述事件有一定的冲突，要解决这些冲突笔者花了一些功夫，主要是在各个地方进行必要的清零。过程比较复杂，加上本事件是拓展内容，这里就只做个简单的介绍了。

**处理交互事件：**前一小节我们将交互事件关联起来，render作为控制中心，在接收了交互事件请求后，接下来就是处理这些交互事件，做出相应的控制。

* 处理拖拽旋转事件：

public void handleTouchDrag(float deltaX, float deltaY) {  
 yAngle += deltaX \* 0.3f;// 设置填充椭圆绕y轴旋转的角度  
 xAngle += deltaY \* 0.3f;// 设置填充椭圆绕x轴旋转的角度  
}

该方法为连续调用，变换角度会即时地反映在每一帧上。

* 处理缩放事件：

public void upScale() {  
 scale \*= (1 + *SCALE\_STEP*);  
 if (scale >= *SCALE\_MAX*) {  
 scale = *SCALE\_MAX*;  
 }  
 }

该方法为放大事件，是连续调用，放大尺寸会即时地反映在每一帧上。

* 处理双点击缩放事件：

当两次点击完后，手指离开了屏幕，系统不再获取触摸事件。假如我们要做一个放大操作，但发现该事件不是连续的，直接调用upscale()只会调用一次，并不是连续调用，没法得到想要的现象。

为解决这个问题，我们可以使用一个while（） 循环来连续调用upscale方法。当scale到达某一特定值时，停止调用。这是一个可行的方法，运行程序，然后执行相应的操作，图像确实被放大，但总感觉很怪：图像是被瞬间放大的，体验很糟糕！作为一个追求高质产品的技术开发人员，肯定不会想要这样的体验，我期望在变换的过程是渐变的，柔和的。

那么要如何实现这种效果？While（）在循环次数不足够大的情况下是瞬间完成的，在调用upscale方法次数一定的情况，笔者希望while()能慢下来。至此进入了一个思维误区--其实我们是希望while里面想要执行的内容慢下来，而不是while（）本身慢下来。

有了上思想的正确指导，笔者通过控制两次调用upscale 方法的时间间隔来实现渐变效果。声明一个long 变量记录上次调用的时间，当当前时间与上次调用时间时间间隔大于某一值时，调用upscale，然后更新上次调用时间为当前时间。运行程序，实验效果非常理想，优雅。

while (scale < 3) {  
 if (System.*currentTimeMillis*() - intervalStart > 5) {  
 upScale();  
 intervalStart = System.*currentTimeMillis*();  
 }  
 }