Android 3D 游戏开发教程– Part I-VI

这几篇 Android 3D 游戏开发的文章原文出自一位德国人 Martin 在 droidnova.com 写的文章，有 lixinso 翻译为中文。
第一部分首先介绍 OpenGL 相关的术语，并引导你开始 3D 开发的第一步。
这个关于 3D 游戏的系列的叫做 Vortex .
这个教程主要 focus 在 3D 编程上，其他的东西比如菜单和程序生命周期虽然是代码的一部分，但是在这里不会被提到。
首先开始介绍 OpenGL 的术语。

顶点 Vertex

顶点是 3D 空间中的一个点，也是许多对象的基础元素。在 OpenGL 中你可以生命少至二维坐标(X,Y)，多至四维(X,Y,Z,W). w 轴是可选的，默认的值是 1.0. Z 轴也是可选的，默认为 0. 在这个系列中，我们将要用到 3 个主要的坐标 X，Y，Z，因为 W 一般都是被用来作为占位符。vertex 的复数是 vertices（这对非英语母语的人来说比较重要，因为这容易产生歧义）。所有的对象都是用 vertices 作为它们的点，因为点就是 vertex。
三角形 Triangle
三角形需要三个点才能创建。因此在 OpenGL 中，我们使用 3 个顶点来创建一个三角形。
多边形 Polygon
多边形是至少有 3 个连接着的点组成的一个对象。三角形也是一个多边形。
图元 Primitives
一个 Primitive 是一个三维的对象，使用三角形或者多边形创建。形象的说，一个有 50000 个顶点的非常精细的模型是一个 Primitive，同样一个只有 500 个顶点的低模也叫做一个 Primitive。
现在我们可以开始变成了。
创建一个工程交 Vortex，activity 也是这个名字。我们的工程应该大概是这个样子的：

```
package com.droidnova.android.games.vortex;

import android.app.Activity;
import android.os.Bundle;

public class Vortex extends Activity {
    private static final String LOG_TAG = Vortex.class.getSimpleName();
    private VortexView _vortexView;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       _vortexView = new VortexView(this);
       setContentView(_vortexView);
    }
}
```

复制代码

如上图所示，我们已经添加了 View。让我们看一下 VortexView 类。

```
package com.droidnova.android.games.vortex;

import android.content.Context;
import android.opengl.GLSurfaceView;

public class VortexView extends GLSurfaceView {
    private static final String LOG_TAG = VortexView.class.getSimpleName();
    private VortexRenderer _renderer;

    public VortexView(Context context) {
       super(context);
       _renderer = new VortexRenderer();
       setRenderer(_renderer);
    }
}
```

复制代码

如上所示，我们继承了 GLSurfaceView 是因为它会帮助我们画 3D 图像。接下来看 VortexRenderer 类。一个 Renderer 包含画一帧所必需的所有东西。 引用自这儿 references 。Renderer 负责 OpenGL call 来 render 一个帧。

来看一下这个类:

```
package com.droidnova.android.games.vortex;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView;

public class VortexRenderer implements GLSurfaceView.Renderer {
private static final String LOG_TAG = VortexRenderer.class.getSimpleName();

private float _red = 0.9f;
private float _green = 0.2f;
private float _blue = 0.2f;

@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
// Do nothing special.
}

@Override
public void onSurfaceChanged(GL10 gl, int w, int h) {
gl.glViewport(0, 0, w, h);
}
```

```
@Override
public void onDrawFrame(GL10 gl) {
// define the color we want to be displayed as the "clipping wall"
gl.glClearColor(_red, _green, _blue, 1.0f);
// clear the color buffer to show the ClearColor we called above...
gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
}
}
```
复制代码

好，我们做了什么？

首先我们实现了 GLSurfaceView.Renderer 这个接口，主要是实现 3 个方法：onSurfaceCreated(), onSurfaceChanged() 和 onDrawFrame()。这些方法很容易理解，第一个在 surface 创建以后调用，第二个是在 surface 发生改变以后调用，例如从竖屏切换到横屏的时候，最后一个方法是当任何时候调用一个画图方法的时候。

从 11 行到 13 行，我们用浮点数来定义 RGB 颜色系统中的每一个颜色。

在 28 行，我们通过 glClearColor()方法为底色定义了颜色。底色是在我们能看到的所有东西的后面，所以所有在底色后面的东西都是不可见的。可以想象这种东西为浓雾，挡住了所有的东西。然后我们将要为之设置距离来 show 一下它怎么用的。那时候你就一定会明白它是怎么存在的了。

为了让颜色变化可见，我们必须调用 glClear()以及颜色缓冲的 Mask 来清空 buffer，然后为我们的底色使用新的底色。

为了能看到它在起作用，我们这里为 MotionEvent 创建一个 response，使用它来改变颜色。首先在 VortexRenderer 中来创建一个设置颜色的函数。

```
public void setColor(float r, float g, float b) {
    _red = r;
    _green = g;
    _blue = b;
}
```
复制代码

下面是 VortexView 类中创建的方法来处理 MotionEvent。

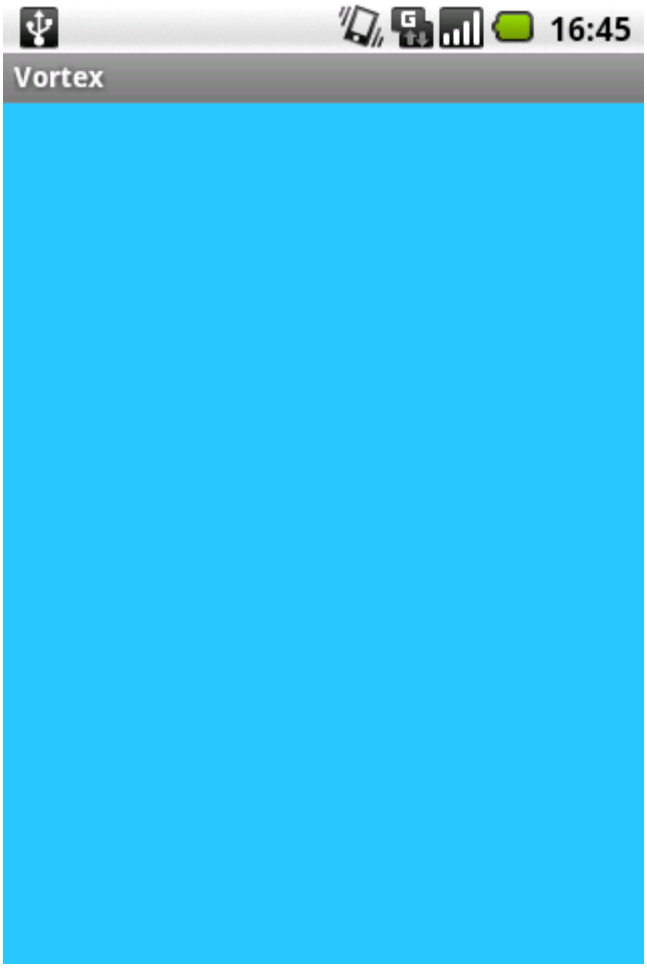
```
public boolean onTouchEvent(final MotionEvent event) {
queueEvent(new Runnable() {
public void run() {
_renderer.setColor(event.getX() / getWidth(), event.getY() / getHeight(), 1.0f);
}
});
return true;
}
```
复制代码

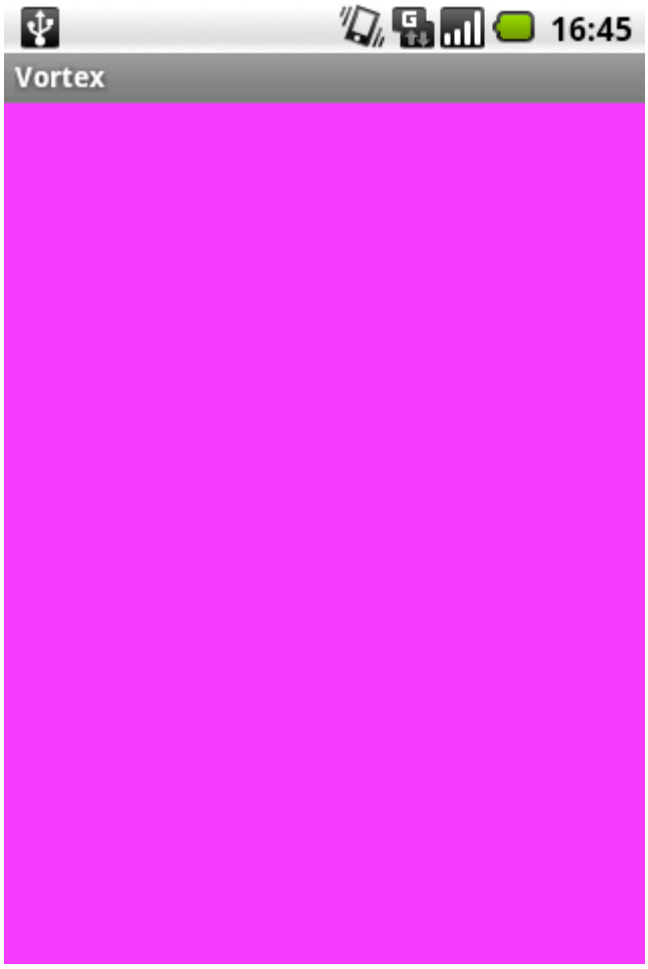
我们创建了一个匿名的 Runnable 对象，这里的 run()方法调用 renderer 中的 setColor 方法。这有会根据 MotionEvent 坐标做一些小的计算。

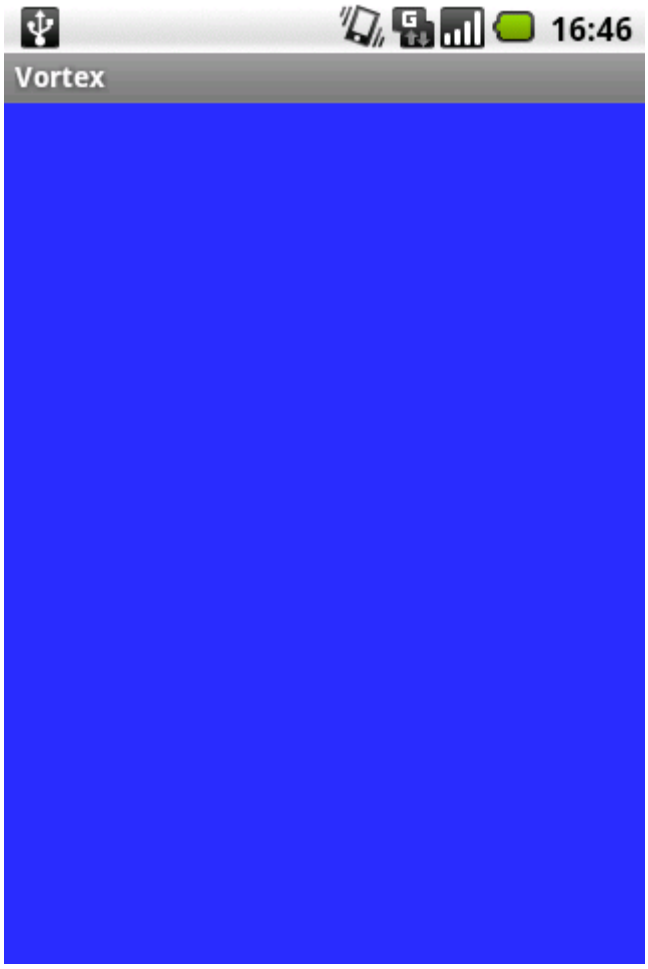
现在我们已经有了一个小小的程序来使用 OpenGl 来改变我们的背景色了。

在德语中我们叫这种小 case 为“Mit Kanonen auf Spatzen schießen”,翻译过来应该是“你在

车轮上打死了一只苍蝇”。这说的恰到好处，这只是一个最最最小的例子，要学习 OpenGL，你现在要准备更多更多的东西。
这部分最后提一下 OpenGL 的文档 documentation for OpenGL 。这个东西虽然可用想不高，但是它最少是一个文档。
Eclipse 工程源代码在这里下载（原地址）： Vortex Part I
这里是几个截图:




16:45
Vortex



这个系列的第二部分是关于如何添加一个三角形并可以旋转它。
第一件事情是初始化需要显示的三角形。我们来在 VortexRenderer 类中添加一个方法 initTriangle()。

```
// new object variables we need
// a raw buffer to hold indices
private ShortBuffer _indexBuffer;

// a raw buffer to hold the vertices
private FloatBuffer _vertexBuffer;

private short[] _indicesArray = {0, 1, 2};
private int _nrOfVertices = 3;

// code snipped
```

```
private void initTriangle() {
    // float has 4 bytes
    ByteBuffer vbb = ByteBuffer.allocateDirect(_nrOfVertices * 3 * 4);
    vbb.order(ByteOrder.nativeOrder());
    _vertexBuffer = vbb.asFloatBuffer();

    // short has 2 bytes
    ByteBuffer ibb = ByteBuffer.allocateDirect(_nrOfVertices * 2);
    ibb.order(ByteOrder.nativeOrder());
    _indexBuffer = ibb.asShortBuffer();

    float[] coords = {
       -0.5f, -0.5f, 0f, // (x1, y1, z1)
       0.5f, -0.5f, 0f, // (x2, y2, z2)
       0f, 0.5f, 0f // (x3, y3, z3)
    };

    _vertexBuffer.put(coords);
    _indexBuffer.put(_indicesArray);

    _vertexBuffer.position(0);
    _indexBuffer.position(0);
}
```

复制代码

让我们从新的对象变量开始. _vertexBuffer 为我们的三角形保存坐标._indexBuffer 保存索引. _nrOfVertices 变量定义需要多少个顶点.对于一个三角形来说,一共需要三个顶点.

这个方法首先为这里两个 buffer 分配必须的内存(14-22 行). 接下来我们定义一些坐标(24-28 行) 后面的注释对用途给予了说明.

在 30 行,我们将 coords 数组填充给_vertexBuffer . 同样在 31 行将 indices 数组填充给_indexBuffer 。最后将两个 buffer 都设置 position 为 0.

为了防止每次都对三角形进行初始化，我们仅仅在 onDrawFrame()之前的行数调用它一次。一个比较好的选择就是在 onSurfaceCreated()函数中.

```
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    // preparation
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    initTriangle();
}
```

复制代码

glEnableClientState() 设置 OpenGL 使用 vertex 数组来画。这是很重要的，因为如果不这么设置 OpenGL 不知道如何处理我们的数据。接下来我们就要初始化我们的三角形。

为什么我们不需使用不同的 buffer？ 在新的 onDrawFrame()方法中我们必须添加一些新的 OpenGL 调用。

```
@Override
```

```
public void onDrawFrame(GL10 gl) {
// define the color we want to be displayed as the "clipping wall"
gl.glClearColor(_red, _green, _blue, 1.0f);

// clear the color buffer to show the ClearColor we called above...
gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

// set the color of our element
gl.glColor4f(0.5f, 0f, 0f, 0.5f);

// define the vertices we want to draw
gl.glVertexPointer(3, GL10.GL_FLOAT, 0, _vertexBuffer);

// finally draw the vertices
gl.glDrawElements(GL10.GL_TRIANGLES, _nrOfVertices, GL10.GL_UNSIGNED_SHORT,
_indexBuffer);
}
```

复制代码

好，一步一步地看。

glClearColor() 和 glClear() 在教程 I 部分已经提到过。在第 10 行使用 glColor4f(red, green, blue, alpha)设置三角形为暗红色 .

在第 13 行，我们使用 glVertexPointer()初始化 Vertex Pointer. 第一个参数是大小，也是顶点的维数。我们使用的是 x,y,z 三维坐标。第二个参数，GL_FLOAT 定义 buffer 中使用的数据类型。第三个变量是 0，是因为我们的坐标是在数组中紧凑的排列的，没有使用 offset。最后哦胡第四个参数顶点缓冲。

最后，glDrawElements()将所有这些元素画出来。第一个参数定义了什么样的图元将被画出来。第二个参数定义有多少个元素，第三个是 indices 使用的数据类型。最后一个是绘制顶点使用的索引缓冲。

当最后测试这个应用的使用，你会看到一个在屏幕中间静止的三角形。当你点击屏幕的时候，屏幕的背景颜色还是会改变。

现在往里面添加对三角形的旋转。下面的代码是写在 VortexRenderer 类中的.

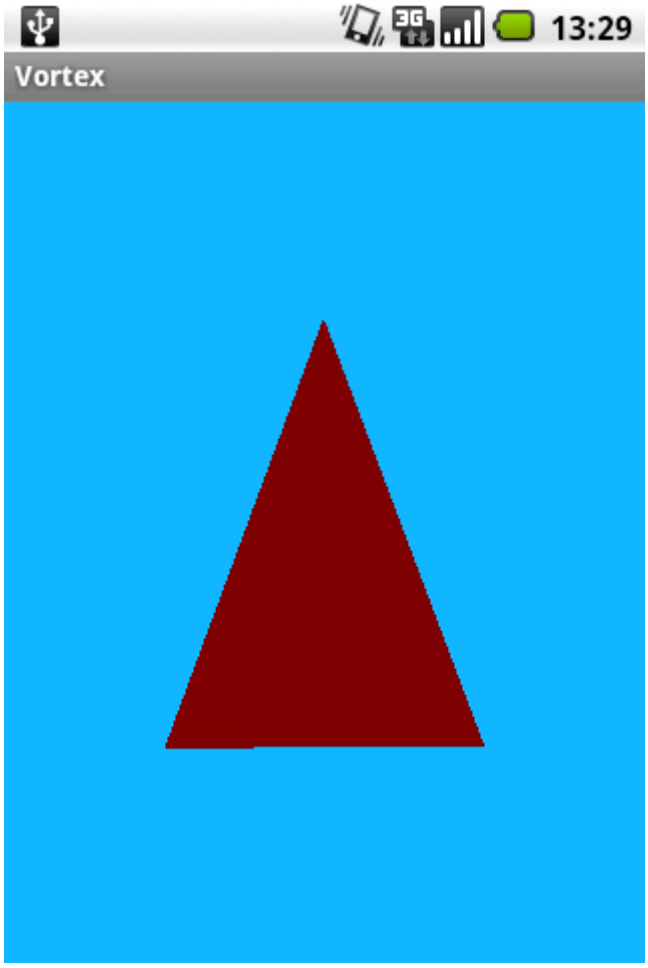
```
private float _angle;

public void setAngle(float angle) {
_angle = angle;
}
```

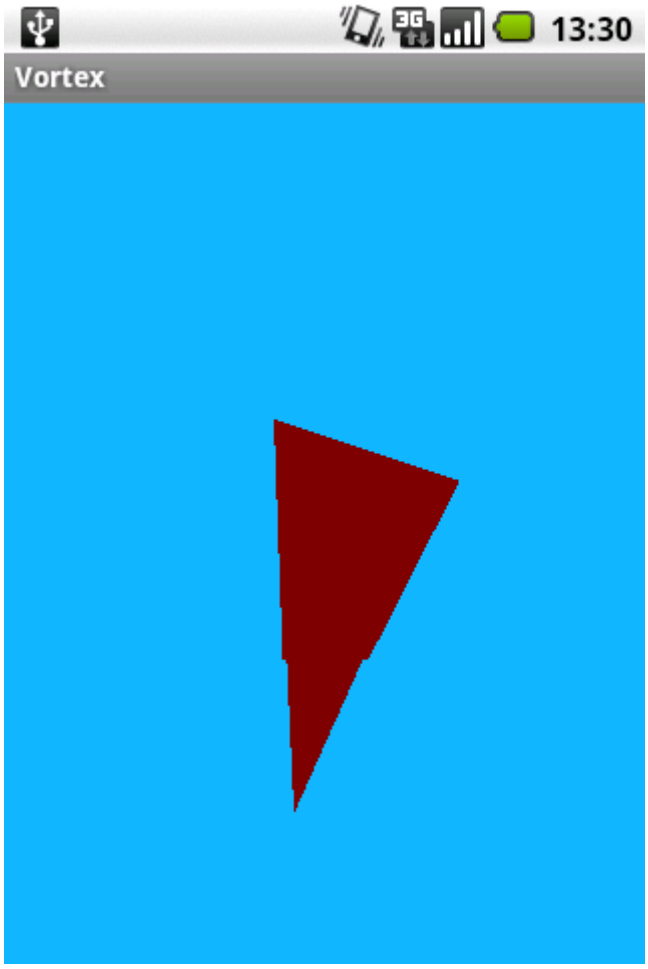
复制代码

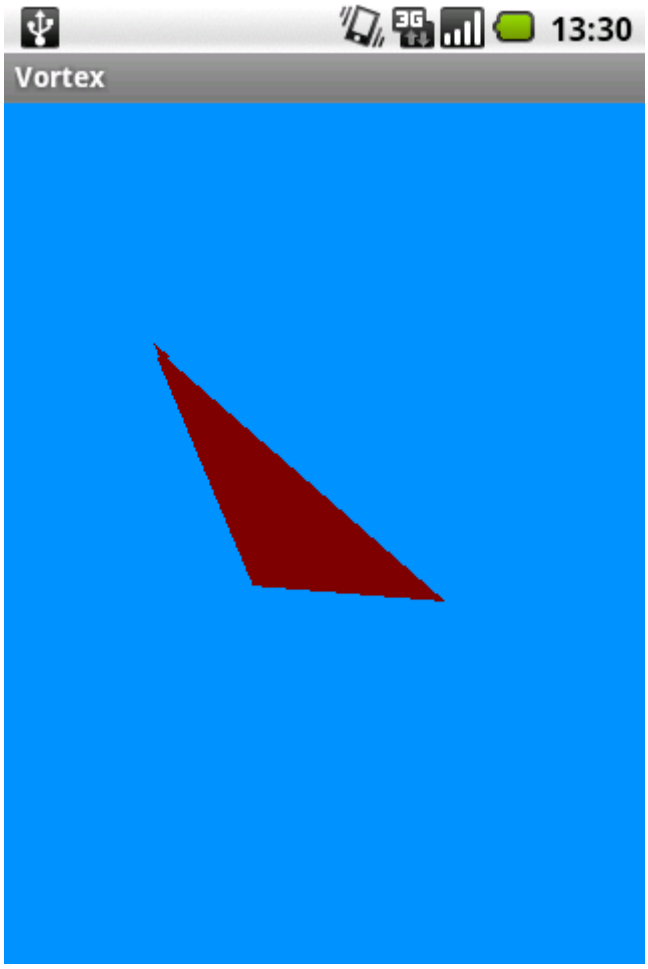
glRotatef()方法在 glColor4f()之前被 onDrawFrame()调用.

```
@Override
public void onDrawFrame(GL10 gl) {
// set rotation
gl.glRotatef(_angle, 0f, 1f, 0f);

gl.glColor4f(0.5f, 0f, 0f, 0.5f);
```

```
// code snipped
}
```
复制代码
这时候我们可以绕 y 轴旋转。如果需要改变只需要改变 glRotate()方法中的 0f。这个参数中的值表示一个向量，标志三角形绕着旋转的坐标轴。
要让它可用，我们必须在 VortexView 中的 onTouchEvent()中添加一个调用。
```
public boolean onTouchEvent(final MotionEvent event) {
    queueEvent(new Runnable() {
      public void run() {
          _renderer.setColor(event.getX() / getWidth(), event.getY() / getHeight(), 1.0f);
          _renderer.setAngle(event.getX() / 10);
      }
    });
    return true;
}
```
复制代码
上面代码中除以 10 是为了减小角度变换的速度。
现在编译运行这个程序。如果你在屏幕的最左边点击，你会看到三角形轻微旋转。如果你将手指移到右边，旋转的速度就会变得很快。
Eclipse 工程的源代码在这里下载（原链接）: Vortex Part II


13:29
Vortex


13:30
Vortex



在这个系列的第三部分给你 show 一下如何停止三角形的转动，并告诉你原来的旋转其实只是在三角形上进行的旋转，而不是在摄像机“camera”上进行的旋转。
我们希望能对旋转进行更多的控制。为此，在每次调用 onDrawFrame()方法的时候都会重置这个矩阵。这会重设三角形的角度以便其总是可以旋转到给定的角度。

```
@Override
public void onDrawFrame(GL10 gl) {
    // define the color we want to be displayed as the "clipping wall"
    gl.glClearColor(_red, _green, _blue, 1.0f);

    // reset the matrix - good to fix the rotation to a static angle
    gl.glLoadIdentity();

    // clear the color buffer and the depth buffer to show the ClearColor
    // we called above...
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

    // code snipped
```

}
复制代码
在 VortexView 类中，你应该删除“除以 10”以便其可以旋转范围更大一些。
_renderer.setAngle(event.getX());
复制代码
如果尝试了这些，你将会看到旋转只会根据触摸的到的位置来旋转。如果没有触摸屏幕，旋转不会发生改变。
下一件事情：我们旋转的是三角形本身，还是旋转的 view/camera？
为了验证它，最简单的办法是创建第二个不旋转的三角形进行对照。
最快也是最笨的办法是 copy&paste initTriangle()方法为一个新的方法 initStaticTriangle()，copy&paste 其中的两个 buffer，copy&paste 并修改 onDrawFrame()方法中的最后四行。
不要忘记了改变第二个三角形的颜色以及改变第二个三角形的坐标，这样方便我们能看到两个三角形。我将每个地方的 0.5f 都改成了 0.4f.
这里是整个的类：

```
package com.droidnova.android.games.vortex;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.nio.ShortBuffer;

import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView;

public class VortexRenderer implements GLSurfaceView.Renderer {
    private static final String LOG_TAG = VortexRenderer.class.getSimpleName();

    private float _red = 0f;
    private float _green = 0f;
    private float _blue = 0f;

    // a raw buffer to hold indices allowing a reuse of points.
    private ShortBuffer _indexBuffer;
    private ShortBuffer _indexBufferStatic;

    // a raw buffer to hold the vertices
    private FloatBuffer _vertexBuffer;
    private FloatBuffer _vertexBufferStatic;

    private short[] _indicesArray = {0, 1, 2};
    private int _nrOfVertices = 3;
```

```
    private float _angle;

    @Override
    public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        // preparation
        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
        initTriangle();
        initStaticTriangle();
    }

    @Override
    public void onSurfaceChanged(GL10 gl, int w, int h) {
        gl.glViewport(0, 0, w, h);
    }

    public void setAngle(float angle) {
        _angle = angle;
    }

    @Override
    public void onDrawFrame(GL10 gl) {
        // define the color we want to be displayed as the "clipping wall"
        gl.glClearColor(_red, _green, _blue, 1.0f);

        // reset the matrix - good to fix the rotation to a static angle
        gl.glLoadIdentity();

        // clear the color buffer to show the ClearColor we called above...
        gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

        // draw the static triangle
        gl.glColor4f(0f, 0.5f, 0f, 0.5f);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, _vertexBufferStatic);
                                gl.glDrawElements(GL10.GL_TRIANGLES,        _nrOfVertices,
GL10.GL_UNSIGNED_SHORT, _indexBufferStatic);

        // set rotation for the non-static triangle
        gl.glRotatef(_angle, 0f, 1f, 0f);

        gl.glColor4f(0.5f, 0f, 0f, 0.5f);
        gl.glVertexPointer(3, GL10.GL_FLOAT, 0, _vertexBuffer);
                                gl.glDrawElements(GL10.GL_TRIANGLES,        _nrOfVertices,
GL10.GL_UNSIGNED_SHORT, _indexBuffer);
```

```
}

private void initTriangle() {
    // float has 4 bytes
    ByteBuffer vbb = ByteBuffer.allocateDirect(_nrOfVertices * 3 * 4);
    vbb.order(ByteOrder.nativeOrder());
    _vertexBuffer = vbb.asFloatBuffer();

    // short has 4 bytes
    ByteBuffer ibb = ByteBuffer.allocateDirect(_nrOfVertices * 2);
    ibb.order(ByteOrder.nativeOrder());
    _indexBuffer = ibb.asShortBuffer();

    float[] coords = {
        -0.5f, -0.5f, 0f, // (x1, y1, z1)
        0.5f, -0.5f, 0f, // (x2, y2, z2)
        0f, 0.5f, 0f // (x3, y3, z3)
    };

    _vertexBuffer.put(coords);

    _indexBuffer.put(_indicesArray);

    _vertexBuffer.position(0);
    _indexBuffer.position(0);
}

private void initStaticTriangle() {
    // float has 4 bytes
    ByteBuffer vbb = ByteBuffer.allocateDirect(_nrOfVertices * 3 * 4);
    vbb.order(ByteOrder.nativeOrder());
    _vertexBufferStatic = vbb.asFloatBuffer();

    // short has 4 bytes
    ByteBuffer ibb = ByteBuffer.allocateDirect(_nrOfVertices * 2);
    ibb.order(ByteOrder.nativeOrder());
    _indexBufferStatic = ibb.asShortBuffer();

    float[] coords = {
        -0.4f, -0.4f, 0f, // (x1, y1, z1)
        0.4f, -0.4f, 0f, // (x2, y2, z2)
        0f, 0.4f, 0f // (x3, y3, z3)
    };
```

```
        _vertexBufferStatic.put(coords);

        _indexBufferStatic.put(_indicesArray);

        _vertexBufferStatic.position(0);
        _indexBufferStatic.position(0);
    }

    public void setColor(float r, float g, float b) {
        _red = r;
        _green = g;
        _blue = b;
    }
}
```

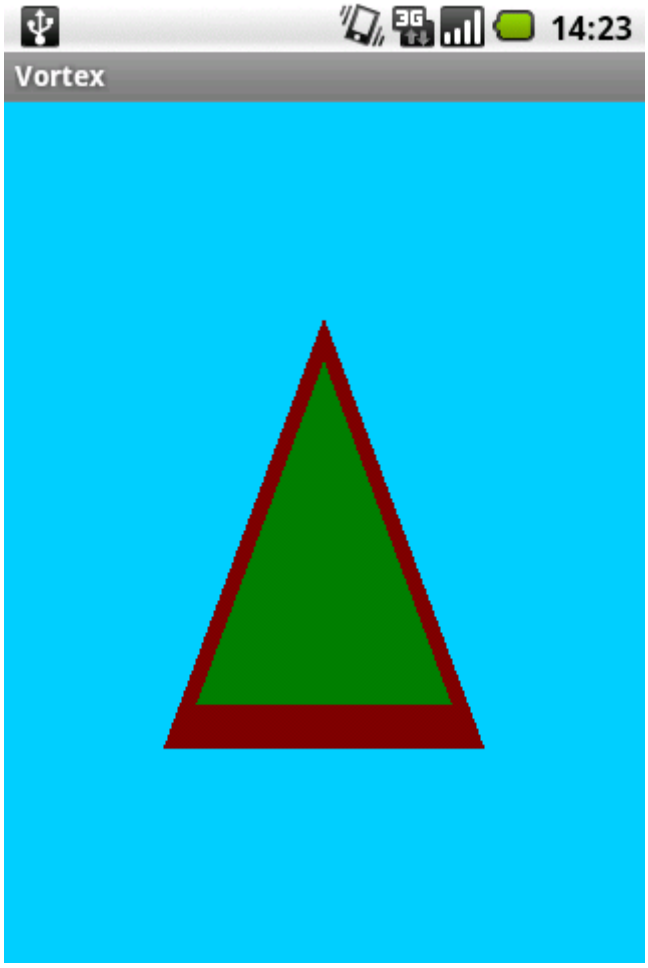
复制代码

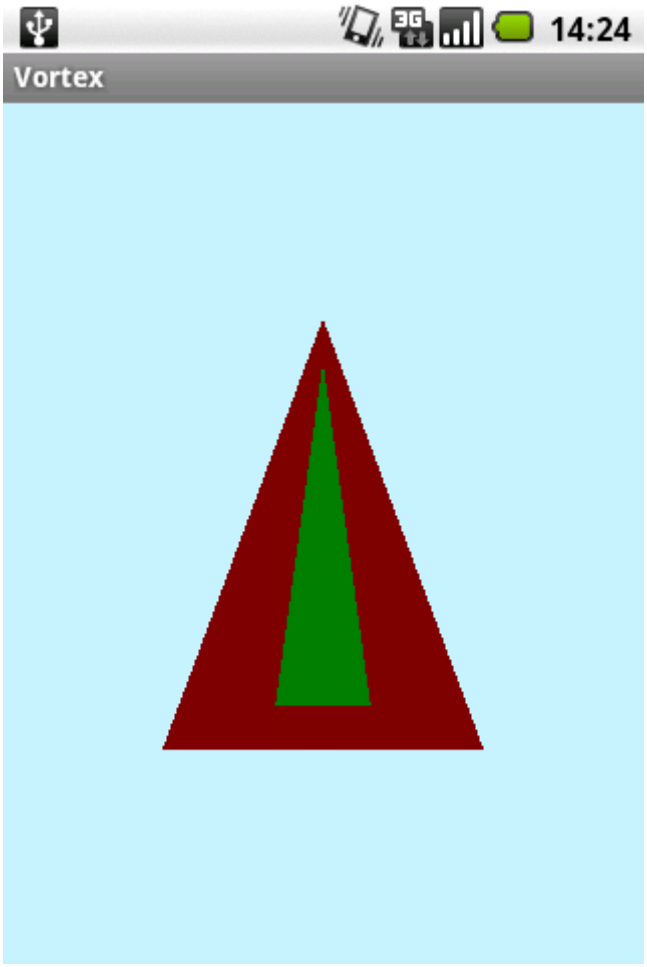
如果作了以上这些，你会看到只有一个三角形可以旋转。如果你想两个都可以旋转，只需要在“draw the static triangle”这个注释的旁边也给它加上一行代码就可以了。

编译并运行这个程序，你可以看到绿色的三角形在旋转，同时红色的三角形还是呆在原来的地方。

这也充分验证了我们的答案，我们旋转的只是三角形而不是整个场景。

Eclipse 工程源代码在这里下载: Vortex Part III


14:23
Vortex



这个系列的第四部分讲如何给三角形添加颜色。
在上一部分我们创建了第二个静态的三角形来验证我们旋转的是三角形而不是整个场景。这里我们将这个静态的三角形删除掉。删除掉 initStaticTriangle()函数，删除两个 buffer，_indexBufferStatic 和_vertexBufferStatic。同时也要删除原来初始静止三角形时用到的 onDrawFrame()中的最后四行。
新的 onDrawFrame()方法如下：

```
@Override
public void onDrawFrame(GL10 gl) {
    // define the color we want to be displayed as the "clipping wall"
    gl.glClearColor(_red, _green, _blue, 1.0f);

    // reset the matrix - good to fix the rotation to a static angle
    gl.glLoadIdentity();

    // clear the color buffer to show the ClearColor we called above...
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);
```

```
    // set rotation for the non-static triangle
    gl.glRotatef(_angle, 0f, 1f, 0f);

    gl.glColor4f(0.5f, 0f, 0f, 0.5f);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, _vertexBuffer);
                        gl.glDrawElements(GL10.GL_TRIANGLES,  _nrOfVertices,
GL10.GL_UNSIGNED_SHORT, _indexBuffer);
}
```

复制代码

现在我们为保存颜色信息创建一个新的 buffer。这个_colorBuffer 是一个对象变量，但是我们需要在 initTriangle()方法中定义颜色并填充这个 buffer.

```
// code snipped

// a raw buffer to hold the colors
private FloatBuffer _colorBuffer;

// code snipped
private void initTriangle() {
    // float has 4 bytes
    ByteBuffer vbb = ByteBuffer.allocateDirect(_nrOfVertices * 3 * 4);
    vbb.order(ByteOrder.nativeOrder());
    _vertexBuffer = vbb.asFloatBuffer();

    // short has 4 bytes
    ByteBuffer ibb = ByteBuffer.allocateDirect(_nrOfVertices * 2);
    ibb.order(ByteOrder.nativeOrder());
    _indexBuffer = ibb.asShortBuffer();

    // float has 4 bytes, 4 colors (RGBA) * number of vertices * 4 bytes
    ByteBuffer cbb = ByteBuffer.allocateDirect(4 * _nrOfVertices * 4);
    cbb.order(ByteOrder.nativeOrder());
    _colorBuffer = cbb.asFloatBuffer();

    float[] coords = {
       -0.5f, -0.5f, 0f, // (x1, y1, z1)
       0.5f, -0.5f, 0f, // (x2, y2, z2)
       0.5f, 0.5f, 0f // (x3, y3, z3)
    };

    float[] colors = {
       1f, 0f, 0f, 1f, // point 1
       0f, 1f, 0f, 1f, // point 2
       0f, 0f, 1f, 1f, // point 3
    };
```

```
    _vertexBuffer.put(coords);
    _indexBuffer.put(_indicesArray);
    _colorBuffer.put(colors);

    _vertexBuffer.position(0);
    _indexBuffer.position(0);
    _colorBuffer.position(0);
}
```
复制代码

我们创建了一个 FloatBuffer 类型的对象变量_colorBuffer（第四行）。在 initTriangle()方法中我们为新的颜色 buffer 分配了足够多的内存(19-21 行)。接下来我们创建了一个 float 数组（23-27 行），每个顶点有 4 个值。 这个结构是 RGBA(Red，Green，Blue，alpha）的，所以第一个顶点是红颜色，第二个颜色是绿色，第三个颜色是蓝色。最后两部和_vertexBuffer 相同。我们将颜色数组放到 buffer 里面，将 buffer 的 position 设置为 0.

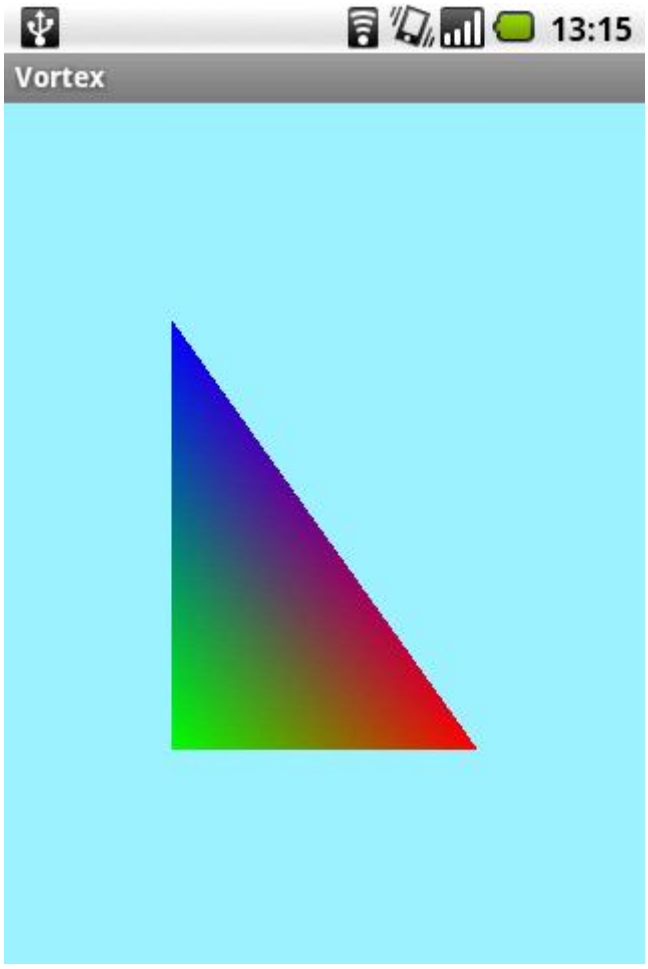
当这些准备工作都做完了以后，我们开始告诉 OpenGL ES 使用我们的颜色数组。这通过 glEnableClientState()，以及 glColorPointer()来完成，和 vertexBuffer 类似。

```
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    // preparation
    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_COLOR_ARRAY);
    initTriangle();
}

// code snipped

@Override
public void onDrawFrame(GL10 gl) {
    // code snipped

    // gl.glColor4f(0.5f, 0f, 0f, 0.5f);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, _vertexBuffer);
    gl.glColorPointer(4, GL10.GL_FLOAT, 0, _colorBuffer);
                        gl.glDrawElements(GL10.GL_TRIANGLES,  _nrOfVertices,
GL10.GL_UNSIGNED_SHORT, _indexBuffer);
}
```
复制代码

第五行我们 enable 了 color mode。在 17 行我们设置了颜色 pointer。参数 4 表示 RGBA(RGBA 刚好是四个值），其余的几个参数大家都比较熟悉了。

也许你也看到了，我们注释掉了 15 行，因为我们使用的是 color mode,所以不再需要 glColor4f。它会覆盖，所以我们可以注注释掉或者删除掉他。

Eclipse 工程源代码参考: Vortex Part IV


13:15
Vortex

系列的第五部分讲如果创建你的第一个完整的 3D 对象。这个 case 中是一个 4 面的金字塔。
为了让我们接下来的开发更容易，这里需要做一些准备。
我们必须将计算 buffer 以及创建数组时的大变得更加动态。

```
private int _nrOfVertices = 0;

private void initTriangle() {
    float[] coords = {
            // coodinates
    };
    _nrOfVertices = coords.length;

    float[] colors = {
            // colors
    };

    short[] indices = new short[] {
            // indices
```

```
    };

    // float has 4 bytes, coordinate * 4 bytes
    ByteBuffer vbb = ByteBuffer.allocateDirect(coords.length * 4);
    vbb.order(ByteOrder.nativeOrder());
    _vertexBuffer = vbb.asFloatBuffer();

    // short has 2 bytes, indices * 2 bytes
    ByteBuffer ibb = ByteBuffer.allocateDirect(indices.length * 2);
    ibb.order(ByteOrder.nativeOrder());
    _indexBuffer = ibb.asShortBuffer();

    // float has 4 bytes, colors (RGBA) * 4 bytes
    ByteBuffer cbb = ByteBuffer.allocateDirect(colors.length * 4);
    cbb.order(ByteOrder.nativeOrder());
    _colorBuffer = cbb.asFloatBuffer();

    _vertexBuffer.put(coords);
    _indexBuffer.put(indices);
    _colorBuffer.put(colors);

    _vertexBuffer.position(0);
    _indexBuffer.position(0);
    _colorBuffer.position(0);
}
```

复制代码

为了更加的动态，我们必须改变一些变量以便我们接下来的工作。让我们来细看一下：

在第一行你可以看到，我们初始化_nrOfVertices 为 0，因为我们可以在第七行那里通过坐标的大小来确定它。

我们同时也将_indecesArray 改为局部变量 indices，并在 13 行进行了初始化。

这个 buffer 创建过程被放在了坐标、颜色以及顶点数组的下边，因为 buffer 大小取决于数组。所以请看 17-18 行，22-23 行，27-28 行。在注释里面我解释了计算方法。

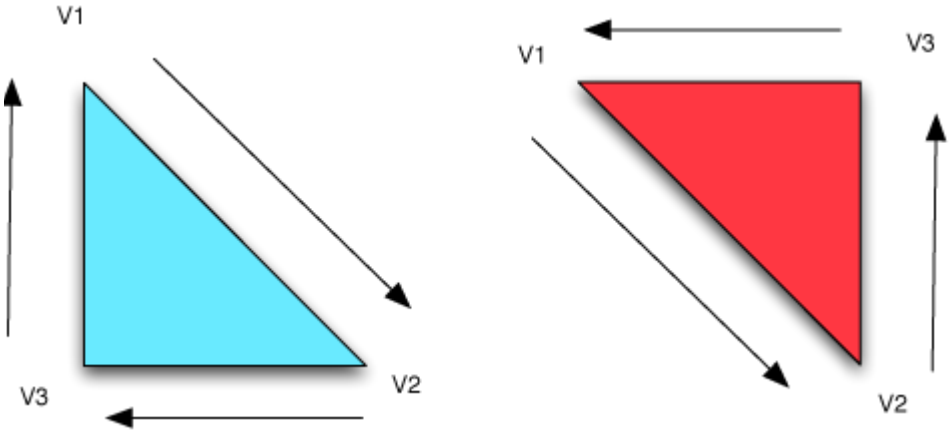
主要的好处是，我们可以创建更多的 vertices，而不用手动重新计算有多少个 vertices，以及数组和 buffer 的大小。

下一步：你需要明白 OpenGL 如何绘制并决定我们看到的东西。

相对于 OpenGL 来说 OpenGL ES 一个很大的缺点就是除了三角形以外没有别的图元类型。我们没有其它多边形，所以我们想要创建的所有的对象都必须由三角形构成。我引用一个 blog 的帖子来说明这个问题： IPhone developer ，同时也推荐他的这些文章 OpenGL ES series.

这里有更多的关于三角形的东西你需要知道。在 OpenGL 中，有一个概念叫做弯曲（winding），意思是 vertices 绘制时的顺序。与现实世界中的对象不同，OpenGL 中的多边形一般没有两个面。他们只有一个面，一般是正面，一个三角形只有当其正面面对观察者的时候才可以被看到。可以配置 OpenGL 将一个多边形作为两面的，但是默认情况下三角

形只有一个可见的面。知道了那边是多边形的正面以后，OpenGL 就可以少做一半的计算量。如果设置两面都可视，则需要多的计算。
虽然有时候一个多边形会独立地显示，但是你或许不是非常需要它的背面显示，经常一个三角形是一个更大的对象的一部分，多边形的一面将在这个物体的内部，所以永远也不会被看到。这个没有被显示的一面被称作背面，OpenGl 通过绘制的顺序来确定那个面是正面哪个是背面。顶点按照逆时针绘制的是正面（默认是这样，但是可以被改变）。因为 OpenGL 能很容易地确定哪些三角形对用户是可视的，它就可以通过使用 Backface Culling 来避免为那些不显示在前面的多边形做无用功。我们将在下一篇文章里讨论视角的问题，但是你现在可以想象它为一个虚拟摄像机，或者通过一个虚拟的窗口来观察 OpenGL 的世界。



在上面的示意图中，左边青绿色的的三角形是背面，将不会被绘制，因为它相对于观察者来说是顺时针的。而在右边的这个三角形是正面，将会被绘制，因为绘制顶点的顺序相对于观察者来说是逆时针的。
因为我们想做的是创建一个漂亮的金字塔，我们首先 disable 这个 glClearColor()。我们可以删除掉变量_red, _green, _blue,还有方法 setColor(). 我们也想改变视角，所以我们将旋转分的 x 和 y 轴上。

```
public class VortexRenderer implements GLSurfaceView.Renderer {
private static final String LOG_TAG = VortexRenderer.class.getSimpleName();

// a raw buffer to hold indices allowing a reuse of points.
private ShortBuffer _indexBuffer;

// a raw buffer to hold the vertices
private FloatBuffer _vertexBuffer;

// a raw buffer to hold the colors
private FloatBuffer _colorBuffer;

private int _nrOfVertices = 0;

private float _xAngle;
```

```
private float _yAngle;

@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
// code snipped
}

@Override
public void onSurfaceChanged(GL10 gl, int w, int h) {
gl.glViewport(0, 0, w, h);
}

public void setXAngle(float angle) {
_xAngle = angle;
}

public float getXAngle() {
return _xAngle;
}

public void setYAngle(float angle) {
_yAngle = angle;
}

public float getYAngle() {
return _yAngle;
}
// code snipped
```

复制代码

为了确保你那里有了和这边相同的对象变量，我也将这些都贴到这个类的上边了。你可以看到我们现在有两个 float 变量，_xAngle 和_yAngle(15-16 行)还有他们的 setter 和 getter 方法(28-42 行)

现在让我们实现根据触摸屏来计算角度的逻辑，为了做这个，需要先稍稍改变一下 VortexView 类。

```
// code snipped

private float _x = 0;
private float _y = 0;

// code snipped

public boolean onTouchEvent(final MotionEvent event) {
    if (event.getAction() == MotionEvent.ACTION_DOWN) {
        _x = event.getX();
```

```
        _y = event.getY();
    }
    if (event.getAction() == MotionEvent.ACTION_MOVE) {
        final float xdiff = (_x - event.getX());
        final float ydiff = (_y - event.getY());
        queueEvent(new Runnable() {
            public void run() {
                _renderer.setXAngle(_renderer.getXAngle() + ydiff);
                _renderer.setYAngle(_renderer.getYAngle() + xdiff);
            }
        });
        _x = event.getX();
        _y = event.getY();
    }
    return true;
}
```

复制代码

在第 3 和 4 行我们有两个变量给我们的 x 和 y 值使用。当移动时，我们在 ACTION_DOWN 事件中设置他们的值，我们根据 MotionEvent 来计算当前值和旧的值的差。计算他们的差并加到已经应用的角度上。不要被 ydiff 添加到 x-angle 上或者 xdiff 添加到 y-angle 上而迷惑。你可以想象，如果你想 x 轴的值不变，而进行旋转，只有在 y 轴上旋转。对于 y 轴也一样。如果我们向左或者向上移动手指，xdiff/ydiff 的值就会变成负的，旋转就会向后旋转。所以在这两个轴上旋转都比较容易。

现在到了非常有意思的部分：金字塔。

像上面我们所引用的一样，winding 需要一些设置。有的也许是默认的设置，但是我们还是定义一下，为了安全起见。

```
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    // preparation
    // enable the differentiation of which side may be visible
    gl.glEnable(GL10.GL_CULL_FACE);
    // which is the front? the one which is drawn counter clockwise
    gl.glFrontFace(GL10.GL_CCW);
    // which one should NOT be drawn
    gl.glCullFace(GL10.GL_BACK);

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

    initTriangle();
}
```

复制代码

在第 5 行，我们 enable 了 culling 面，以保证只有一面。在第 7 行我们定义了那种顺序是前面。GL_CCW 表示逆时针。在第 9 行，我们最终定义了那个面作为 culling 面。 沃恩设置

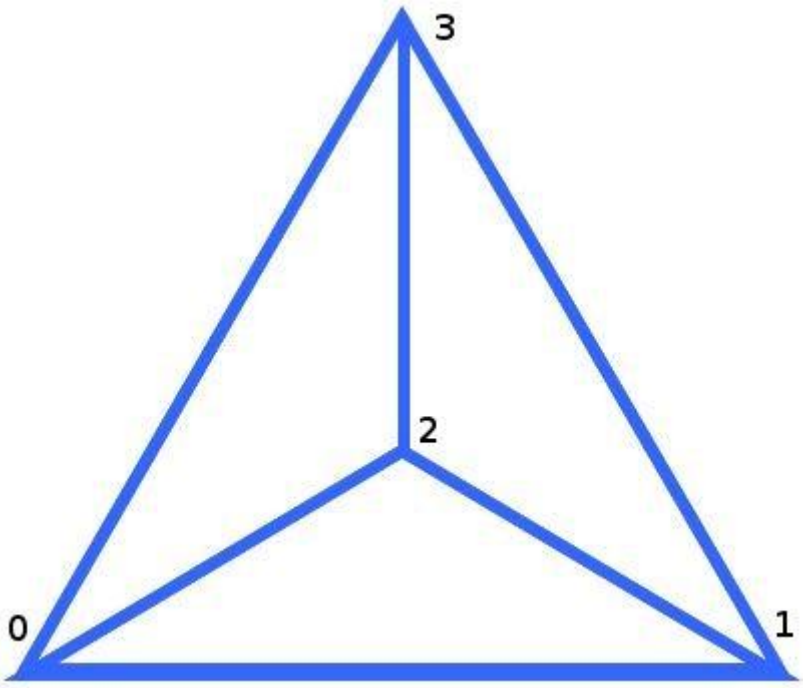
其为 GL_BACK 以保证只显示正面。这或许有点迷糊偶，你可以看看如果用 GL_FRONT_AND_BACK 会发生什么……你将什么也看不到。

```
@Override
public void onDrawFrame(GL10 gl) {
// define the color we want to be displayed as the "clipping wall"
gl.glClearColor(0f, 0f, 0f, 1.0f);

// reset the matrix - good to fix the rotation to a static angle
gl.glLoadIdentity();

// clear the color buffer to show the ClearColor we called above...
gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

// set rotation
gl.glRotatef(_xAngle, 1f, 0f, 0f);
gl.glRotatef(_yAngle, 0f, 1f, 0f);

//gl.glColor4f(0.5f, 0f, 0f, 0.5f);
gl.glVertexPointer(3, GL10.GL_FLOAT, 0, _vertexBuffer);
gl.glColorPointer(4, GL10.GL_FLOAT, 0, _colorBuffer);
gl.glDrawElements(GL10.GL_TRIANGLES, _nrOfVertices, GL10.GL_UNSIGNED_SHORT,
_indexBuffer);
}
```

复制代码

在第四行你可以看到背景色是黑色，因为我们已经删除了以前设置的动态颜色。在 13 和 14 行你可以看到在每个轴的旋转角度。其它的都和以前讲过的一样。

最后一件事情你需要做的是改变 initTriangle()函数中的颜色数组，坐标和索引。我们的金子谈应该显示成这样子的。

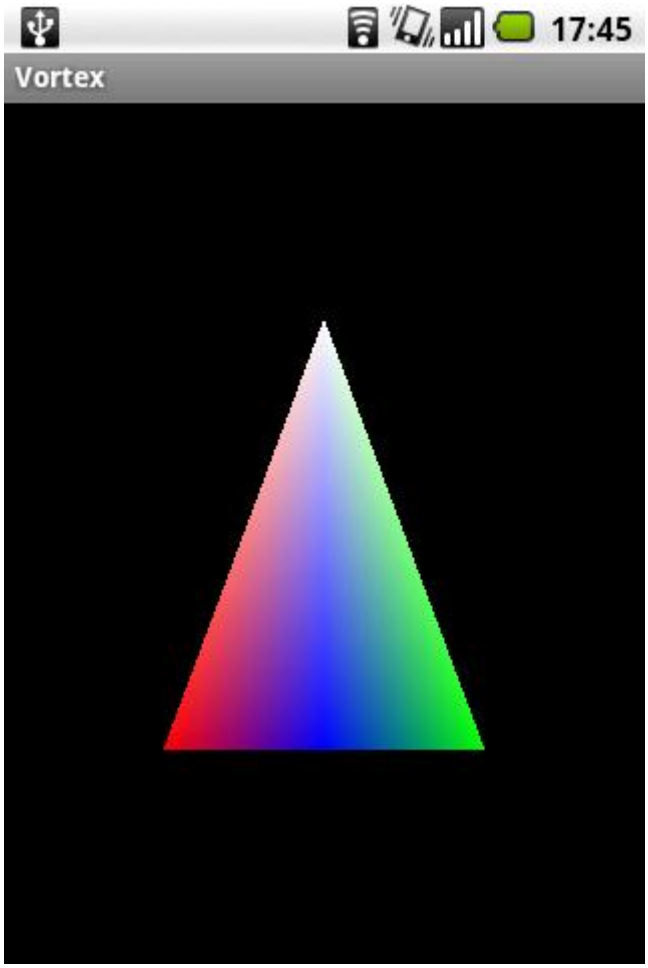


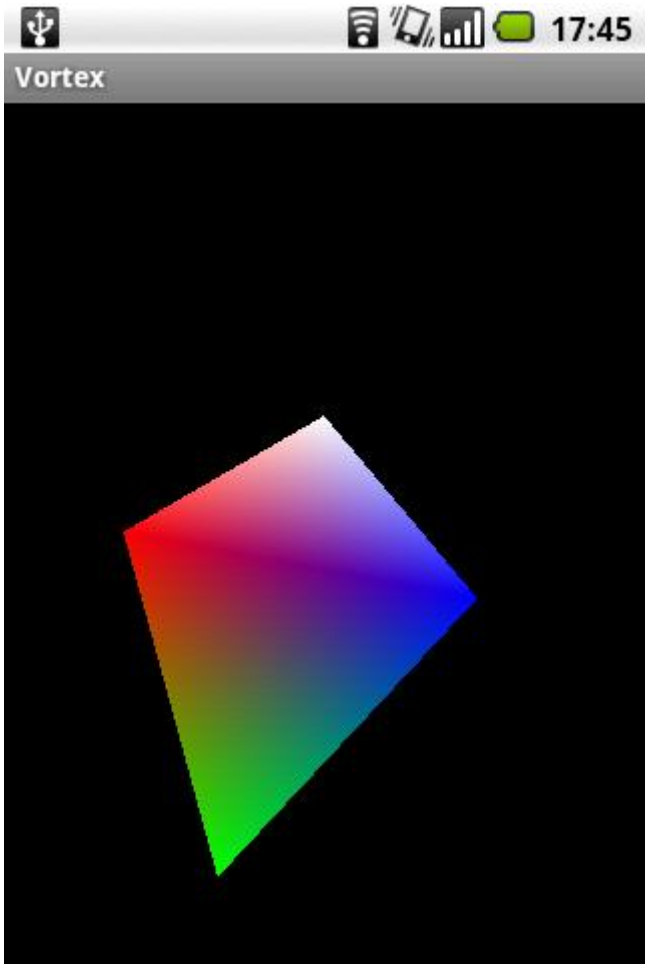

```
private void initTriangle() {
    float[] coords = {
            -0.5f, -0.5f, 0.5f, // 0
            0.5f, -0.5f, 0.5f, // 1
            0f, -0.5f, -0.5f, // 2
            0f, 0.5f, 0f, // 3
    };
    _nrOfVertices = coords.length;

    float[] colors = {
            1f, 0f, 0f, 1f, // point 0 red
            0f, 1f, 0f, 1f, // point 1 green
            0f, 0f, 1f, 1f, // point 2 blue
            1f, 1f, 1f, 1f, // point 3 white
    };

    short[] indices = new short[] {
            0, 1, 3, // rwg
            0, 2, 1, // rbg
            0, 3, 2, // rbw
            1, 2, 3, // bwg
    };

    //code snipped
}
```

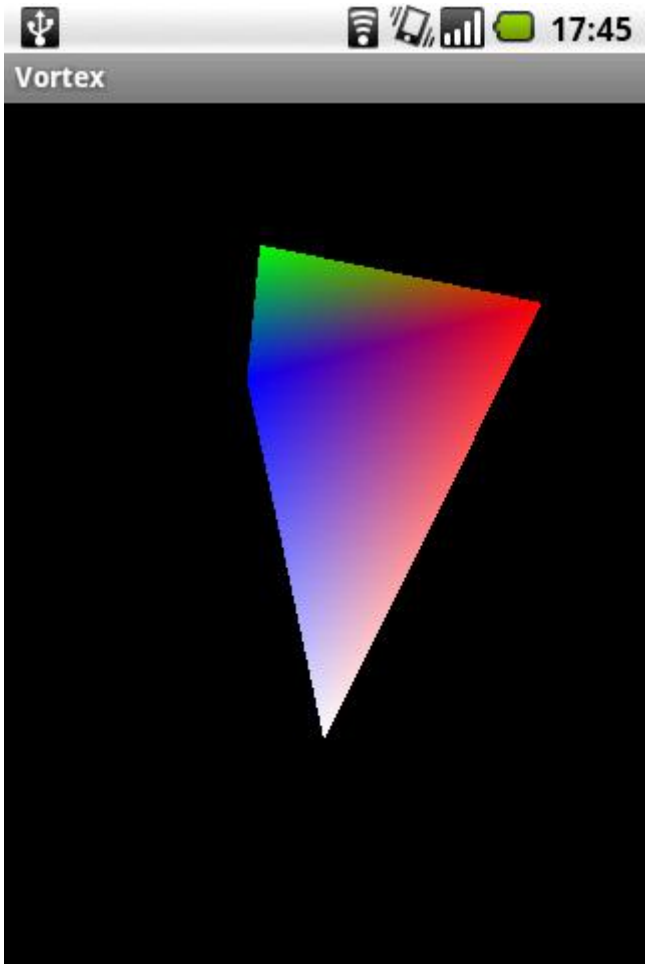
复制代码

正如你在图上看到的一样，我们的金字塔有 4 个角。每个角都有自己的坐标，所以我们有 4

个顶点需要定义。如 2-7 行。
每一个顶点有自己的颜色，在 10-15 行定义。
在中间部分给出了定义三角形所需要的索引。时刻记着 winding，三角形 0，1，3 和 0，3，1 是不一样的。
每一个索引点指向 coords 数组中定义的顶点。
编译运行程序，看一下如果你改变了 indices 的顺序会发生什么，或者看一下如果你将 GL_CCW 改成 GL_CW 以后将会看到什么。
Eclipse 工程的源代码下载: Vortex Part V




17:45
Vortex



这个系列的第六部分主要是关于如何创建正确的视角，因为如果没有正确的视角，3D 就没有任何意义。
在开始之前我们需要先讨论一下 OpenGL 提供的这两种 view：正交和投影。
正交 Orthographic (无消失点投影)
正交视图无法看到一个物体是远离自己还是正在我们面前。为什么？因为它不会根据距离收缩。所以如果你如果你画一个固定大小的物体在视点前面，同时画一个同样大小的物体在第一个物体的远后方，你无法说那个物体是第一个。因为两个都是一样的大小，根距离无关。他们不会随着距离而收缩。
透视 Perspective (有消失点投影)
透视视图和我们从眼睛看到的视图是一样的。例如，一个高个子的人站在你面前，你看上去是很高的。如果这个人站在 100 米以外，他甚至还没有你的拇指大。他看上去会随着距离而缩小，但是我们实际上都知道，它依然是个高个子。这种效果叫做透视。上面例子中提到的两个物体，第二个物体将会显示地更小，所以我们可以区分哪个是离我们近的物体，那个是离我们远的物体。
因为我的例子或许让你会迷惑。我再次推荐这个 blog 帖子：iPhone development: OpenGL ES From the Ground Up, Part 3: Viewports in Perspective，这里面使用或者轨道作为例子。
我们想创建的第一个 view 是哟娜 orthographic。这个创建过程只需要一次，就是在每次 surface

被 created 的时候。所以我们需要改一下代码。一些在 onDrawFrame()中的方法将要被转移到 onSurfaceCreated()方法中。这样他们只会当程序启动或者旋转的时候被执行。

```
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    // preparation
    // define the color we want to be displayed as the "clipping wall"
    gl.glClearColor(0f, 0f, 0f, 1.0f);

    // enable the differentiation of which side may be visible
    gl.glEnable(GL10.GL_CULL_FACE);
    // which is the front? the one which is drawn counter clockwise
    gl.glFrontFace(GL10.GL_CCW);
    // which one should NOT be drawn
    gl.glCullFace(GL10.GL_BACK);

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

    initTriangle();
}

// code snipped

@Override
public void onDrawFrame(GL10 gl) {
    gl.glLoadIdentity();

    // clear the color buffer and the depth buffer to show the ClearColor
    // we called above...
    gl.glClear(GL10.GL_COLOR_BUFFER_BIT);

    // set rotation
    gl.glRotatef(_xAngle, 1f, 0f, 0f);
    gl.glRotatef(_yAngle, 0f, 1f, 0f);

    //gl.glColor4f(0.5f, 0f, 0f, 0.5f);
    gl.glVertexPointer(3, GL10.GL_FLOAT, 0, _vertexBuffer);
    gl.glColorPointer(4, GL10.GL_FLOAT, 0, _colorBuffer);
                        gl.glDrawElements(GL10.GL_TRIANGLES,  _nrOfVertices,
GL10.GL_UNSIGNED_SHORT, _indexBuffer);
}
```

复制代码

你可以看到我们没有将 glClear() 和 glLoadIdentity() 方法从 onDrawFrame() 移动到 onSurfaceCreate 中。原因很简单：他们会在每一帧都被绘制。

因为我们需要使用屏幕大小来计算屏幕的比例，所以我们引入了两个对象变量：_width 和 _height。我们需要在 onSurfaceChanged()方法中设置，在每次旋转改变的时候调用。

```
private float _width = 320f;
private float _height = 480f;

@Override
public void onSurfaceChanged(GL10 gl, int w, int h) {
_width = w;
_height = h;
gl.glViewport(0, 0, w, h);
}
```

复制代码

现在我们已经有了启动一个视点所需要的所有东西。我们需要改变一下 onSurfaceChanged() 方法。

```
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
gl.glMatrixMode(GL10.GL_PROJECTION);
float ratio = _width / _height;
// orthographic:
gl.glOrthof(-1, 1, -1 / ratio, 1 / ratio, 0.01f, 100.0f);
gl.glViewport(0, 0, (int) _width, (int) _height);
gl.glMatrixMode(GL10.GL_MODELVIEW);
gl.glEnable(GL10.GL_DEPTH_TEST);

// define the color we want to be displayed as the "clipping wall"
gl.glClearColor(0f, 0f, 0f, 1.0f);

// enable the differentiation of which side may be visible
gl.glEnable(GL10.GL_CULL_FACE);
// which is the front? the one which is drawn counter clockwise
gl.glFrontFace(GL10.GL_CCW);
// which one should NOT be drawn
gl.glCullFace(GL10.GL_BACK);

gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

initTriangle();
}
```

复制代码

Wow,现在有了好多的新代码，别怕，我们一步一步来。

在第三行，我们可以看到 glMatrixMode()，使用 GL10.GL_RPOJECTION 作为参数，在第 8 行我们可以再次看到这个方法，但是是使用 GL10.GL_MODELVIEW 作为变量。这样使用的原因是 3-8 行中间的代码。在这几行中，4-7 行设置了我们的视点，所以我们设置了我们

的投影。在 9-17 行，我们设置了我们的模型环境。在这中上下文中，两种调用使用不同的参数应该是可理解的。Tips：经常可以试着去删除掉一些代码行来看一下结果。这样比较容易明白哪些代码行是起什么作用的。
第四行计算下一样需要的屏幕比率。在这行中（6 行）我们设置我们的视点来做 orthographic view。这些参数是为边界设定,顺序是这样的：left, right, bottom, top, zNear, zFar。
在第 7 行我们设置了视点。我们知道这个方法的用法因为我们已经在 onSurfaceChanged()中使用过了。
在第 8 行我们切换了 MatrixMode 到 GL10.GL_MODELVIEW，设置 OpenGL 接受关于改变 model 绘制方式的调用。
第 9 行我们调用了 glEnable()并使用参数 GL10.GL_DEPTH_TEST.这使 OpenGL ES 检查对象的 z-order。如果我们没有 enable 它，我们将看到最后被绘制的对象一直显示在最前面。这意味着，及时即使这个物体本来应该被更近更大的物体遮盖，我们依然可以看到它。
其它代码行我们已经在前面的几篇中已经介绍过了。
透视视图与之相同，只是不同的地方是不是使用 glOrthof()而是使用 glFrustumf(). glFrustumf()函数的参数和 glOrthof()的参数略有不同。因为我们没有缩小物体，但是我们定义的锥体将被漏斗状切开。看一下这个图片来了解一下 glOrthof()和 glFrustumf()的区别。

正交 Orthographic:

透视 Perspective:

回到我们的代码:

```
@Override
public void onSurfaceCreated(GL10 gl, EGLConfig config) {
    Log.i(LOG_TAG, "onSurfaceCreated()");
    gl.glMatrixMode(GL10.GL_PROJECTION);
    float size = .01f * (float) Math.tan(Math.toRadians(45.0) / 2);
    float ratio = _width / _height;
    // perspective:
    gl.glFrustumf(-size, size, -size / ratio, size / ratio, 0.01f, 100.0f);
    // orthographic:
    //gl.glOrthof(-1, 1, -1 / ratio, 1 / ratio, 0.01f, 100.0f);
    gl.glViewport(0, 0, (int) _width, (int) _height);
    gl.glMatrixMode(GL10.GL_MODELVIEW);
    gl.glEnable(GL10.GL_DEPTH_TEST);

    // define the color we want to be displayed as the "clipping wall"
    gl.glClearColor(0f, 0f, 0f, 1.0f);

    // enable the differentiation of which side may be visible
    gl.glEnable(GL10.GL_CULL_FACE);
    // which is the front? the one which is drawn counter clockwise
    gl.glFrontFace(GL10.GL_CCW);
    // which one should NOT be drawn
```

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

    gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);
    gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

    initTriangle();
}
```
复制代码

Information: 我们要记住计算变量大小（5 行），我们将会看到当我们讨论矩阵的时候它为什么可以用。

在第 8 行我们使用 glFrustumf()替代 glOrthof().这是我们在 orthographic view 和 perspective view 之前切换时需要做的所有的变化。

但是，hey，还有最后一项我们就能看到结果了。OK，让我们改一下 onDrawFrame()方法吧。

```
@Override
public void onDrawFrame(GL10 gl) {
// clear the color buffer and the depth buffer
gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);

gl.glVertexPointer(3, GL10.GL_FLOAT, 0, _vertexBuffer);
gl.glColorPointer(4, GL10.GL_FLOAT, 0, _colorBuffer);

for (int i = 1; i <= 10; i++) {
gl.glLoadIdentity();
gl.glTranslatef(0.0f, -1f, -1.0f + -1.5f * i);
// set rotation
gl.glRotatef(_xAngle, 1f, 0f, 0f);
gl.glRotatef(_yAngle, 0f, 1f, 0f);
gl.glDrawElements(GL10.GL_TRIANGLES, _nrOfVertices, GL10.GL_UNSIGNED_SHORT,
_indexBuffer);
}
}
```
复制代码

OK，我们改变了什么？

第三行，我们修改了参数确保 depth buffer 也会被清除。

在第 9 行，我们开始一个循环来创建 10 个物体。

在第 10 行，我们看到 glLoadIdentity(). 现在它在这儿重设矩阵。 这是必须做的因为我们要使用 glRotatef()和 glTranslatef()来修改我们的物体。但是为了确保我们只修改了当前循环到的物体，我们调用 glLoadIndentify()。所以我们重设每一个对之前物提的 glTranslatef()和 glRotatef()调用。

在 11 行我们看到新的 glTranslatef()方法，将我们的物体移动到另外一个位置。在这个例子中，我们不修改 x 轴上的位置。但是我们修改 y 轴上的-1.0f，这表示它接近我们的屏幕低端。最后的计算你可以看到，只是修改 z 轴的位置，表示物体的神对。第一个物体在-2.5f，第二个在-4.0f 等等。所以我们将 10 个物体摆放在了屏幕的中央。

如果你使用 glFrustumf()调用，你看到的是这种结果：

如果你切换 glFrustumf()和 glOrthof() ，你将会看到这种结果:

Hey 等一下，为什么只看到一个物体？因为在正交情况下我们没有任何的投影。所以每一个独体的物体都有相同的大小所以没有缩放，所以没有任何消失的点，所以实际上所有的物体都在第一个物体的后面。