此示例展示了一个立方体的具体实现过程，与之前的纯 Opengl es 实现相比，它采用了 JPCT-AE 来实现，因为个人认为这个框架很方便，于是从今天开始通过其网站上的 Wiki 来介绍 JPCT-AE 的实现。通过这个示例能让你快速了解 JPCT-AE 的帮助文档，也就是入门。

(1)什么是 JPCT：一种封装了 OPENGL es 的 3D 游戏引擎，有 j2se 与 android 两个版本。

(2)如何获得其 jar 包及帮助文档：
http://download.csdn.net/user/Simdanfeg 处下载

第一个示例:同样的立方体，不同的实现

```
package com.threed.jpct.example;

import java.lang.reflect.Field;

import javax.microedition.khronos.egl.EGL10;
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.egl.EGLDisplay;
import javax.microedition.khronos.opengles.GL10;

import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.view.MotionEvent;

import com.threed.jpct.Camera;
import com.threed.jpct.FrameBuffer;
import com.threed.jpct.Light;
import com.threed.jpct.Logger;
import com.threed.jpct.Object3D;
import com.threed.jpct.Primitives;
import com.threed.jpct.RGBColor;
import com.threed.jpct.SimpleVector;
import com.threed.jpct.Texture;
import com.threed.jpct.TextureManager;
import com.threed.jpct.World;
import com.threed.jpct.util.BitmapHelper;
```

```
import com.threed.jpct.util.MemoryHelper;

/**
 * 一个简单的例子。比起展示如何写一个正确的 android 应用它更着重于展示如何使用 JPCT-AE 这个 3D 游戏框架。
 * 它包含了 Activity 类去处理 pause 和 resume 等方法
 *
 * @author EgonOlsen
 *
 */
public class HelloWorld extends Activity {

    // HelloWorld 对象用来处理 Activity 的 onPause 和 onResume 方法
    private static HelloWorld master = null;

    // GLSurfaceView 对象
    private GLSurfaceView mGLView;

    // 类 MyRenderer 对象
    private MyRenderer renderer = null;

    // 当 JPCT 渲染背景时 FrameBuffer 类提供了一个缓冲,它的结果本质上是一个能显示或者修改甚至能进行更多后处理的图片。
    private FrameBuffer fb = null;

    // World 类是 JPCT 时最重要的一个类，它好像胶水一样把事物"粘"起来。它包含的对象和光线定义了 JPCT 的场景
    private World world = null;

    // 类似 java.awt.*中的 Color 类
    private RGBColor back = new RGBColor(50, 50, 100);

    private float touchTurn = 0;
    private float touchTurnUp = 0;

    private float xpos = -1;
    private float ypos = -1;

    // Object3D 类是一个三维对象,千万不要傻呼呼的认为它与 java.lang.Object 类似。
    // 一个 Object3D 对象作为一个实例被添加到在渲染的 World 对象中。Object3D 在 World
    // 中一次添加一个实例 ，他们可能被联系起作为孩子/父母来在他们中建立一个制度.
```

```
    // 人体模型当然也能应用在以上的规则中。他们常常不加到一个 World 实
例中，而是
    // 绑定到其它对象中(人体模型或非人体模型)。有些方法 在这个类中需要
一个实例
    // 添加到一个 World 实例中(用 World.addObject()方法可以实现)。
    private Object3D cube = null;

    // 每秒帧数
    private int fps = 0;

    // 光照类
    private Light sun = null;

    protected void onCreate(Bundle savedInstanceState) {
        // Logger 类中 jPCT 中一个普通的用于打印和存储消息，错误和警告
的日志类。
        // 每一个 JPCT 生成的消息将被加入到这个类的队列中
        Logger.log("onCreate");
        // 如果本类对象不为 NULL,将从 Object 中所有属性装入该类
        if (master != null) {
            copy(master);
        }

        super.onCreate(savedInstanceState);

        // 实例化 GLSurfaceView
        mGLView = new GLSurfaceView(this);
        // 使用自己实现的 EGLConfigChooser,该实现必须在
setRenderer(renderer)之前
        // 如果没有 setEGLConfigChooser 方法被调用，则默认情况下，视图
将选择一个与当前 android.view.Surface 兼容至少 16 位深度缓冲深度
EGLConfig。
        mGLView.setEGLConfigChooser(new
GLSurfaceView.EGLConfigChooser() {
            public EGLConfig chooseConfig(EGL10 egl, EGLDisplay display)
{
                // Ensure that we get a 16bit framebuffer. Otherwise,
we'''''''''''''''''''''''''''''''''ll fall
                // back to Pixelflinger on some device (read: Samsung
I7500)
                int[] attributes = new int[] { EGL10.EGL_DEPTH_SIZE, 16,
                        EGL10.EGL_NONE };
                EGLConfig[] configs = new EGLConfig[1];
                int[] result = new int[1];
```

```
                egl.eglChooseConfig(display, attributes, configs, 1,
result);
                return configs[0];
            }
        });
        // 实例化 MyRenderer
        renderer = new MyRenderer();
        // 设置 View 的渲染器，同时启动线程调用渲染，以至启动渲染
        mGLView.setRenderer(renderer);
        // 设置一个明确的视图
        setContentView(mGLView);
    }




    // 重写 onPause()
    @Override
    protected void onPause() {
        super.onPause();
        mGLView.onPause();
    }

    // 重写 onResume()
    @Override
    protected void onResume() {
        super.onResume();
        mGLView.onResume();
    }

    // 重写 onStop()
    @Override
    protected void onStop() {
        super.onStop();
    }



    private void copy(Object src) {
        try {
            // 打印日志
            Logger.log("Copying data from master Activity!");
            // 返回一个数组，其中包含目前这个类的的所有字段的 Filed 对
```

```
象
            Field[] fs = src.getClass().getDeclaredFields();
            // 遍历 fs 数组
            for (Field f : fs) {
                // 尝试设置无障碍标志的值。标志设置为 false 将使访问检
查，设置为 true，将其禁用。
                f.setAccessible(true);
                // 将取到的值全部装入当前类中
                f.set(this, f.get(src));
            }
        } catch (Exception e) {
            // 抛出运行时异常
            throw new RuntimeException(e);
        }
    }



    public boolean onTouchEvent(MotionEvent me) {

        // 按键开始
        if (me.getAction() == MotionEvent.ACTION_DOWN) {
            // 保存按下的初始 x,y 位置于 xpos,ypos 中
            xpos = me.getX();
            ypos = me.getY();
            return true;
        }
        // 按键结束
        if (me.getAction() == MotionEvent.ACTION_UP) {
            // 设置 x,y 及旋转角度为初始值
            xpos = -1;
            ypos = -1;
            touchTurn = 0;
            touchTurnUp = 0;
            return true;
        }

        if (me.getAction() == MotionEvent.ACTION_MOVE) {
            // 计算 x,y 偏移位置及 x,y 轴上的旋转角度
            float xd = me.getX() - xpos;
            float yd = me.getY() - ypos;
            // Logger.log("me.getX() - xpos---------->>"
            // + (me.getX() - xpos));
            xpos = me.getX();
```

```
            ypos = me.getY();
            Logger.log("xpos------------>>" + xpos);
            // Logger.log("ypos------------>>" + ypos);
            // 以 x 轴为例，鼠标从左向右拉为正，从右向左拉为负
            touchTurn = xd / -100f;
            touchTurnUp = yd / -100f;
            Logger.log("touchTurn------------>>" + touchTurn);
            // Logger.log("touchTurnUp------------>>" + touchTurnUp);
            return true;
        }

        // 每 Move 一下休眠毫秒
        try {
            Thread.sleep(15);
        } catch (Exception e) {
            // No need for this...
        }

        return super.onTouchEvent(me);
    }

    // MyRenderer 类实现 GLSurfaceView.Renderer 接口
    class MyRenderer implements GLSurfaceView.Renderer {
        // 当前系统的毫秒数
        private long time = System.currentTimeMillis();
        // 是否停止
        private boolean stop = false;

        // 停止
        public void stop() {
            stop = true;
        }

        // 当屏幕改变时
        public void onSurfaceChanged(GL10 gl, int w, int h) {
            // 如果 FrameBuffer 不为 NULL,释放 fb 所占资源
            if (fb != null) {
                fb.dispose();
            }
            // 创建一个宽度为 w,高为 h 的 FrameBuffer
            fb = new FrameBuffer(gl, w, h);
            Logger.log(master + "");
            // 如果 master 为空
            if (master == null) {
```

```
                // 实例化 World 对象
                world = new World();

                // 设置了环境光源强度。设置此值是负的整个场景会变暗，
而为正将照亮了一切。
                world.setAmbientLight(20, 20, 20);

                // 在 World 中创建一个新的光源
                sun = new Light(world);

                // 设置光照强度
                sun.setIntensity(250, 250, 250);

                // 创建一个纹理
                // 构造方法 Texture(Bitmap image)
                // static Bitmap rescale(Bitmap bitmap, int width, int
height)
                // static Bitmap convert(Drawable drawable)
                Texture texture = new Texture(BitmapHelper.rescale(
                        BitmapHelper.convert(getResources().getDrawab
le(
                                R.drawable.glass)), 64, 64));

                // TextureManager.getInstance()取得一个 Texturemanager
对象
                // addTexture("texture",texture)添加一个纹理
                TextureManager.getInstance().addTexture("texture",
texture);

                // Object3D 对象开始了:-)

                // Primitives 提供了一些基本的三维物体，假如你为了测试
而生成一些对象或为
                // 其它目的使用这些类将很明智，因为它即快速又简单，不
需要载入和编辑。
                // 调用 public static Object3D getCube(float scale)
scale:角度
                // 返回一个立方体
                cube = Primitives.getCube(10);

                // 以纹理的方式给对象所有面"包装"上纹理
                cube.calcTextureWrapSpherical();
```

```
                // 给对象设置纹理
                cube.setTexture("texture");

                // 除非你想在事后再用 PolygonManager 修改,否则释放那些
不再需要数据的内存
                cube.strip();

                // 初始化一些基本的对象是几乎所有进一步处理所需的过程。
                // 如果对象是"准备渲染"(装载，纹理分配，安置，渲染模式
设置，
                // 动画和顶点控制器分配),那么 build()必须被调用，
                cube.build();




                // 将 Object3D 对象添加到 world 集合
                world.addObject(cube);

                // 该Camera代表了Camera/viewer在当前场景的位置和方向，
它也包含了当前视野的有关信息
                // 你应该记住Camera的旋转矩阵实际上是应用在World中的
对象的一个旋转矩阵。
                // 这一点很重要，当选择了 Camera 的旋转角度，一个
Camera(虚拟)围绕 w 旋转和通过围绕 World 围绕 w 旋转、
                // 将起到相同的效果，因此，考虑到旋转角度，World 围绕
camera 时，camera 的视角是静态的。假如你不喜欢
                // 这种习惯，你可以使用 rotateCamera()方法
                Camera cam = world.getCamera();

                // 以 50 有速度向后移动 Camera（相对于目前的方向）
                cam.moveCamera(Camera.CAMERA_MOVEOUT, 50);

                // cub.getTransformedCenter()返回对象的中心
                // cam.lookAt(SimpleVector lookAt))
                // 旋转这样 camera 以至于它看起来是在给定的 world-space
的位置
                cam.lookAt(cube.getTransformedCenter());

                // SimpleVector 是一个代表三维矢量的基础类，几乎每一个
矢量都
                // 是用 SimpleVector 或者至少是一个 SimpleVector 变体构
成的(有时由于
```

```
                // 某些原因比如性能可能会用(float x,float y,float z)
之类)。
                SimpleVector sv = new SimpleVector();

                // 将当前 SimpleVector 的 x,y,z 值设为给定的
SimpleVector(cube.getTransformedCenter())的值
                sv.set(cube.getTransformedCenter());

                // Y 方向上减去 100
                sv.y -= 100;

                // Z 方向上减去 100
                sv.z -= 100;

                // 设置光源位置
                sun.setPosition(sv);

                // 强制 GC 和 finalization 工作来试图去释放一些内存，同
时将当时的内存写入日志，
                // 这样可以避免动画不连贯的情况，然而，它仅仅是减少这
种情况发生的机率
                MemoryHelper.compact();

                // 如果 master 为空,使用日志记录且设 master 为
HelloWorld 本身
                if (master == null) {
                    Logger.log("Saving master Activity!");
                    master = HelloWorld.this;
                }
            }
        }

        // 需实现的 onSurfaceCreated(GL10 gl, EGLConfig config)
        public void onSurfaceCreated(GL10 gl, EGLConfig config) {
        }

        // 绘制到当前屏幕哦:-D
        public void onDrawFrame(GL10 gl) {

            try {
                // 如果 stop 为 true
                if (!stop) {
                    // 如果 touchTurn 不为 0,向 Y 轴旋转 touchTure 角度
                    if (touchTurn != 0) {
```

```
                    // 旋转物体的旋转绕 Y 由给定矩阵 W 轴角（弧度顺时针方向为正值）,应用到对象下一次渲染时。
                    cube.rotateY(touchTurn);
                    // 将 touchTurn 置 0
                    touchTurn = 0;
                }

                if (touchTurnUp != 0) {
                    // 旋转物体的旋转围绕 x 由给定角度宽（弧度，逆时针为正值）轴矩阵,应用到对象下一次渲染时。
                    cube.rotateX(touchTurnUp);
                    // 将 touchTureUp 置 0
                    touchTurnUp = 0;
                }

                // 用给定的颜色(back)清除 FrameBuffer
                fb.clear(back);
                // 变换和灯光所有多边形
                world.renderScene(fb);
                // 绘制
                world.draw(fb);
                // 渲染图像显示
                fb.display();

                // 记录 FPS
                if (System.currentTimeMillis() - time >= 1000) {
                    // Logger.log(fps + "fps");
                    fps = 0;
                    time = System.currentTimeMillis();
                }
                fps++;

                // 如果 stop 为 false,释放 FrameBuffer
            } else {
                if (fb != null) {
                    fb.dispose();
                    fb = null;
                }
            }
            // 当出现异常，打印异常信息
        } catch (Exception e) {
            Logger.log(e, Logger.MESSAGE);
        }
    }
```

```
    }
}
```