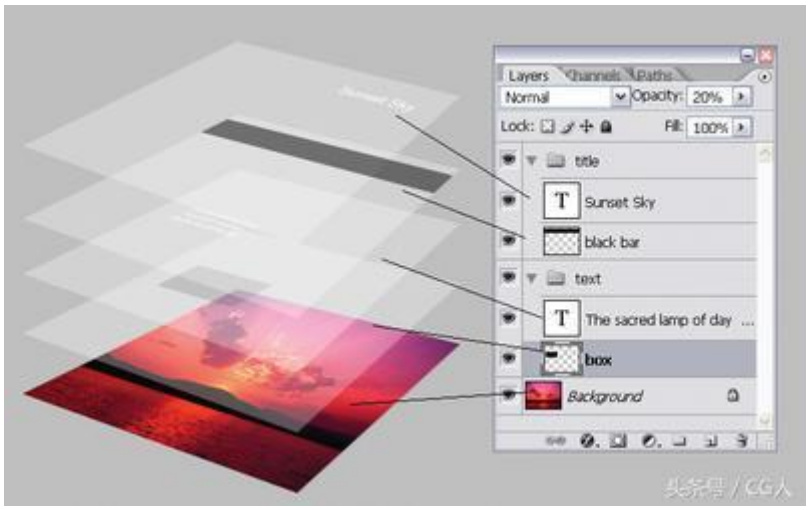
> **在Android平台上，虽说系统自带的GLSurfaceView这个控件中已经帮我们创建好EGL环境了，一般来说使用GLSurfaceView就可以满足我们的需求了。**



1.1 而当我们用不着GLSurfaceView的时候，那又如何自己创建EGL环境呢？因此就来探究下 EGL环境的创建。

> 在Java层，EGL封装了两套框架

位于javax.microedition.khronos.egl包下的EGL10。
位于android.opengl包下的EGL14。
区别如下：

> EGL14是在Android 4.2(API 17)引入的，所以API 17以下的版本不支持EGL14。
>
> EGL10不支持OpenGL ES 2.x，因此在EGL10中某些相关常量参数只能用手写硬编码代替，例如EGL14.EGL_CONTEXT_CLIENT_VERSION以及EGL14.EGL_OPENGL_ES2_BIT等等。

流程如下：

> EGLDisplay，获取默认的显示设备（渲染的目标），并初始化
> EGLConfig，配置参数来获取EGL支持的EGLConfig
> EGLContext，创建上下文环境
> EGLSurface，创建EGLSurface来连接EGL和设备的屏幕

一.EGL 介绍

通俗上讲，OpenGL是一个操作GPU的API，它通过驱动向GPU发送相关指令，控制图形渲染管线状态机的运行状态。但OpenGL需要本地视窗系统进行交互，这就需要一个中间控制层，最好与平台无关。

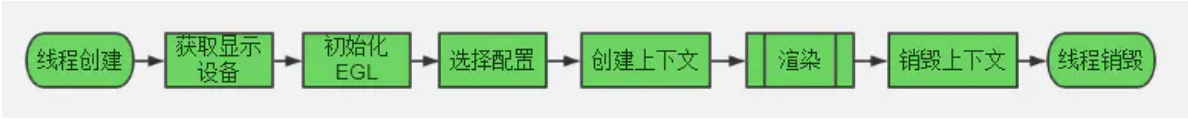
EGL——因此被独立的设计出来，它作为OpenGL ES和本地窗口的桥梁。

EGL 是 OpenGL ES（嵌入式）和底层 Native 平台视窗系统之间的接口。EGL API 是独立于OpenGL ES 各版本标准的独立API ，其主要作用是为OpenGL指令创建 Context 、绘制目标Surface 、配置Framebuffer属性、Swap提交绘制结果等。此外，EGL为GPU厂商和OS窗口系统之间提供了一个标准配置接口。

一般来说，OpenGL ES 图形管线的状态被存储于 EGL 管理的一个Context中。而Frame Buffers 和其他绘制 Surfaces 通过 EGL API进行创建、管理和销毁。 EGL 同时也控制和提供了对设备显示和可能的设备渲染配置的访问。

EGL标准是C的，在Android系统Java层封装了相关API。

动手开始挥泪斩情丝



OpenGL渲染一般流程

OpenGL的渲染是基于线程的，我们这里先创建一个GLRenderer类继承于HandlerThread：

二．EGL 数据类型与初始化（C标准、egl.h）

EGL 包含了自己的一组数据类型，同时也提供了对一组平台相关的本地数据类型的支持。这些 Native 数据类型定义在 EGL 系统的头文件中。EGL中的一些类型和define抽象了平台的窗口与显存类型，如果理解了这些类型，即可对EGL进行相关配置，有些复杂的类型不比太过纠结底层如何实现。

标准 EGL 数据类型如下所示：

```
EGLBoolean ——EGL_TRUE =1, EGL_FALSE=0
EGLint ——int 数据类型
EGLDisplay ——系统显示 ID 或句柄，可以理解为一个前端的显示窗口
EGLConfig ——Surface的EGL配置，可以理解为绘制目标framebuffer的配置属性
EGLSurface ——系统窗口或 frame buffer 句柄，可以理解为一个后端的渲染目标窗口。
EGLContext ——OpenGL ES 图形上下文，它代表了OpenGL状态机；如果没有它，OpenGL指令就没有执行的环境。
```

下面几个类型比较复杂，通过例子可以更深入的理解。这里要说明的是这几个类型在不同平台其实现是不同的，EGL只提供抽象标准。

> NativeDisplayType——Native 系统显示类型，标识你所开发设备的物理屏幕
> NativeWindowType ——Native 系统窗口缓存类型，标识系统窗口
> NativePixmapType ——Native 系统 frame buffer，可以作为 Framebuffer 的系统图像（内存）数据类型，该类型只用于离屏渲染.

EGL Displays

EGLDisplay 是一个关联系统物理屏幕的通用数据类型，表示显示设备句柄，也可以认为是一个前端显示窗。为了使用系统的显示设备， EGL 提供了 EGLDisplay 数据类型，以及一组操作设备显示的 API 。
下面的函数原型用于获取 Native Display：

> EGLDisplay eglGetDisplay (NativeDisplayType display);

其 中 display 参数是 native 系统的窗口显示 ID 值。如果你只是想得到一个系统默认的 Display ，你可以使用 EGL_DEFAULT_DISPLAY 参数。如果系统中没有一个可用的 native display ID 与给定的 display 参数匹配，函数将返回 EGL_NO_DISPLAY ， 而没有任何 Error 状态被设置。

由于设置无效的 display 值不会有任何错误状态，在你继续操作前请检测返回值。下面是一个使用 EGL API 获取系统 Display 的例子：

```
m_eglDisplay = eglGetDisplay( system.display);
if (m_eglDisplay == EGL_NO_DISPLAY || eglGetError() != EGL_SUCCESS))
throw error_egl_display;
```

Initialization 初始化

每个 EGLDisplay 在使用前都需要初始化。初始化 EGLDisplay 的同时，你可以得到系统中 EGL 的实现版本号。了解当前的版本号在向后兼容性方面是非常有价值的。在移动设备上，通过动态查询 EGL 版本号，你可以为新旧版本的 EGL 附加额外的特性或运行环境。基于平台配置，软件开发可用清楚知道哪些 API 可用访问，这将会为你的代码提供最大限度的可移植性。

下面是初始化 EGL 的函数原型：

> EGLBoolean eglInitialize (EGLDisplay dpy, EGLint *major, EGLint *minor);

其中 dpy 应该是一个有效的 EGLDisplay 。函数返回时， major 和 minor 将被赋予当前 EGL 版本号。比如 EGL1.0 ， major 返回 1 ， minor 则返回 0 。给 major 和 minor 传 NULL 是有效的，如果你不关心版本号。
eglQueryString() 函数是另外一个获取版本信息和其他信息的途径。通过 eglQueryString() 获取版本信息需要解析版本字符串，所以通过传递一个指针给 eglInitializ() 函数比较容易获得这个信息。注意在调用 eglQueryString() 必须先使用 eglInitialize() 初始化 EGLDisplay ，否则将得到 EGL_NOT_INITIALIZED 错误信息。

1.2 获取并初始化EGLDisplay
所谓显示设备，也就是我们OpenGL渲染的目标，也可以说就是我们的屏幕，而EGLDisplay是一个封装了物理屏幕的数据类型。

一般来说我们都需要对获取的结果进行验证，如果没获取到抛出异常即可。

```
    //获取默认的显示设备，渲染的目标
    mEglDisplay = EGL14.eglGetDisplay(EGL14.EGL_DEFAULT_DISPLAY);
    if (mEglDisplay == EGL14.EGL_NO_DISPLAY) {
        throw new RuntimeException("egl no display!");
    }
    //初始化显示设备，主次版本号
    int[] version = new int[2];
    if (!EGL14.eglInitialize(mEglDisplay, version, 0, version, 1)){
        mEglDisplay = null;
        throw new RuntimeException("eglInitialize error");
    }
```

1.3获取EGLConfig

当获取到EGLDisplay后，我们需要指定一些配置项，例如色彩格式、像素格式、RGBA的表示以及SurfaceType等，实际上也就是指FrameBuffer的配置参数。

EGL的参数配置在数组中以 id，value 依次存放，对于个别的属性可以只有 id 没有 value ，结尾需要用EGL_NONE标识。

```
eglChooseConfig(EGLDisplay dpy, const EGLint *attrib_list, EGLConfig *configs,
EGLint config_size,EGLint *num_config)
```

> attrib_list：配置参数列表
> configs：返回输出的EGLConfigs数据，可能有多个；
> config_size：表示最多需要输出多少个EGLConfig；
> num_config：表示满足配置参数的EGLConfig的个数。

```
//配置输出的格式 也就是制定FrameBuffer的配置参数
       int[] attrList = {
```

```
                EGL14.EGL_RED_SIZE,8,//颜色缓冲区中红色的位数
                EGL14.EGL_GREEN_SIZE,8,
                EGL14.EGL_BLUE_SIZE,8,
                EGL14.EGL_ALPHA_SIZE,8,
                EGL14.EGL_RENDERABLE_TYPE,EGL14.EGL_OPENGL_ES2_BIT,//渲染窗口支持的
布局组成
                EGL14.EGL_SURFACE_TYPE,EGL14.EGL_PBUFFER_BIT, //egl支持的窗口类型
                EGL14.EGL_NONE
        };
        EGLConfig[] configs = new EGLConfig[1];
        int[] numConfigs = new int[1];
        if (!EGL14.eglChooseConfig(mEglDisplay, attrList, 0, configs, 0,
configs.length, numConfigs, 0)) {
        throw new RuntimeException("unable to find RGB888 ES2 EGL config");
        }
```

1.4 创建EGLContext

一个线程对应一个EGLContext，如果需要共享资源，则可在创建的时候设置即可。

```
int[] attriList = {
                EGL14.EGL_CONTEXT_CLIENT_VERSION, 2,
                EGL14.EGL_NONE
        };
        //第三个参数即为需要共享的上下文对象，资源包括纹理、FrameBuffer以及其他的Buffer等资
源。
        mEglContext = EGL14.eglCreateContext(mEglDisplay, configs[0],
EGL14.EGL_NO_CONTEXT, attriList, 0);
        checkEglError("eglCreateContext");
        if (mEglContext == null) {
            throw new RuntimeException("context is null");
        }
```

1.5 创建EGLSurface

最后则需要创建一个GLSurface，它的作用就是把我们的EGL和我们渲染的目标连接起来。

而我们的 EGLSurface也就是要给FrameBuffer，可以通过如下俩种方式创建。

> EGL14.eglCreatePbufferSurface：创建离屏的Surface
> EGL14.eglCreateWindowSurface：创建可以实际显示的Surface

```
 //创建surface 离屏
    int[] surfaceAttribs = {
            EGL14.EGL_WIDTH, mWidth,
            EGL14.EGL_HEIGHT, mHeight,
            EGL14.EGL_NONE
    };
    mEglSurface = EGL14.eglCreatePbufferSurface(mEglDisplay, configs[0],
surfaceAttribs, 0);
    checkEglError("eglCreatePbufferSurface");
    if (mEglSurface == null) {
        throw new RuntimeException("surface is null");
    }
```

```
    //非离屏 需要surface承载
    mEglSurfaceEncoder = EGL14.eglCreateWindowSurface(mEglDisplay,
configEncoder,surface,new int[]{EGL14.EGL_NONE}, 0);
    checkEglError("eglCreatePbufferSurface");
    if (mEglSurface == null) {
        throw new RuntimeException("surface is null");
    }
```

注意：
EGL在初始化时默认设置的是双缓冲模式，也就是两份FrameBuffer;
一个用于绘制图像，一个用于显示图像，每次绘制完一帧都需要交换一次缓冲。
因此我们需要在OpenGL ES绘制完毕后，调用如下进行交换。

EGL14.eglSwapBuffers(mEglDisplay, mEglSurfaceEncoder);

2 坐标系

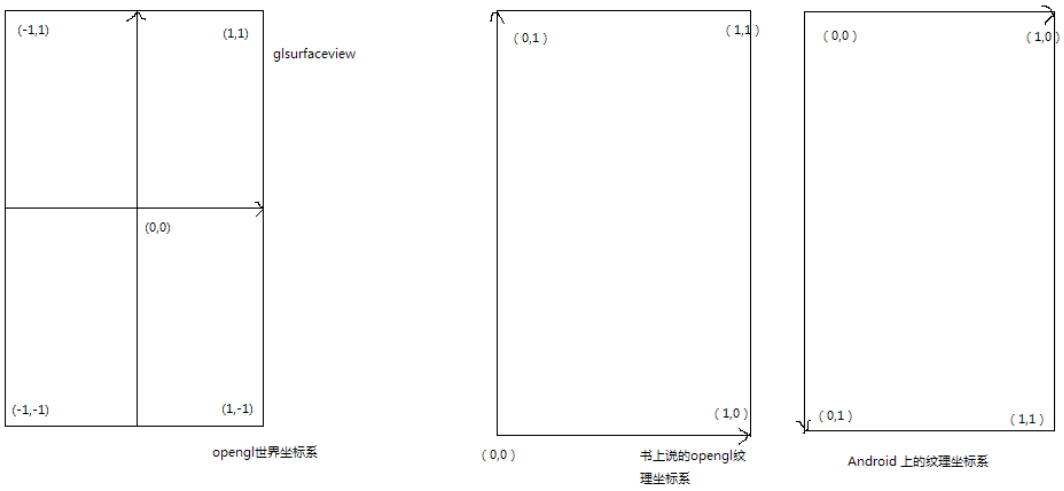
初学opengl ES,每一个教你在屏幕上贴图的opengl版hello world都有这么两数组:

```
static final float COORD[] = {
        -1.0f, -1.0f,
        1.0f, -1.0f,
        -1.0f, 1.0f,
        1.0f, 1.0f,
};

static final float TEXTURE_COORD[] = {
        0.0f, 1.0f,
        1.0f, 1.0f,
        0.0f, 0.0f,
        1.0f, 0.0f,
};
但是几乎都不解释，所以我学的时候都不明白这些点为什么要这么写，前后顺序有没有什么规律。于是各种查资料试验，终于搞懂了。
```


1.坐标系
PS：本人学opengl es主要是为了2D贴图，所以不涉及Z轴

如图



，图一是　　opengl的世界坐标系，这个基本没啥问题，主要是很多教程说纹理坐标是左下原点。实践得出在Android上应该是最右边的图那样，以左上为原点。
个人猜测纹理吧其实就是一组颜色点组成的数组，Android由于UI坐标是以左上为原点，所以把数组里颜色点的存储顺序改了一下，于是坐标系就不一样了。

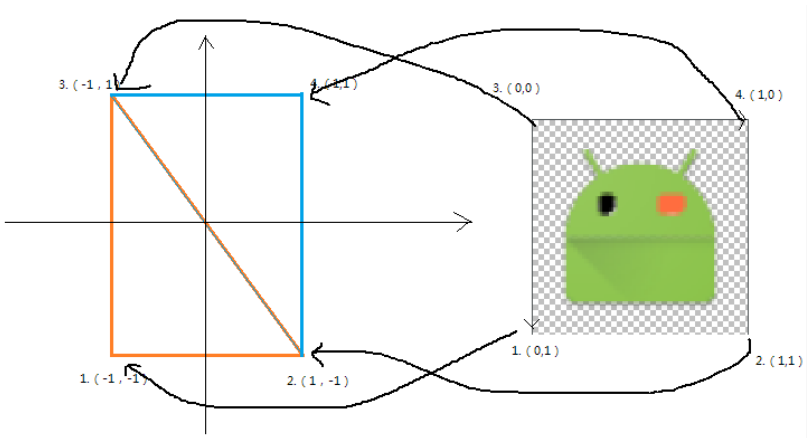
2.1.纹理顶点顺序

纹理的点和世界坐标的点之间是对应的：

```
static final float COORD1[] = {
        -1.0f, -1.0f,
        1.0f, -1.0f,
        -1.0f, 1.0f,
        1.0f, 1.0f,
};

static final float TEXTURE_COORD1[] = {
        0.0f, 1.0f,
        1.0f, 1.0f,
        0.0f, 0.0f,
        1.0f, 0.0f,
};
```

显示结果如图：



如图中箭头，opengl会把纹理中颜色顶点绘到对应的世界坐标顶点上，中间的点则按一定的规律取个平均值什么的，所以可见实际显示的图被上下拉伸了，因为原图是1:1，而在该程序里
GLES20.glViewport(0, 0, width, height);
赋予的显示区域是高大于宽的（这里涉及到opengl世界坐标和屏幕坐标的映射，和本文主旨关系不大就不多说了）。

其实也就是只要世界坐标和纹理坐标数组里的点能够对的上，顺序不是问题