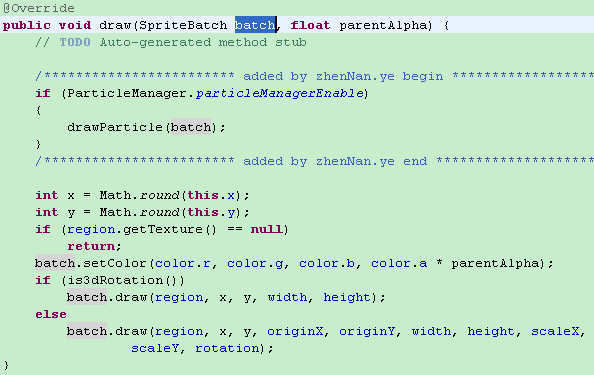
**Shader着色语言初窥**

1、由SpriteBatch说开去。

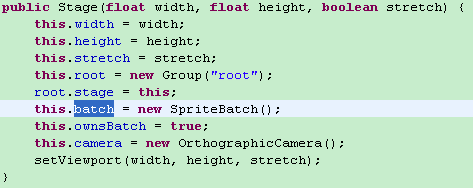
大家都知道我们Launcher的所有图形都是由基类View3D中的draw方法来完成的，或者由子类重写draw方法来完成。



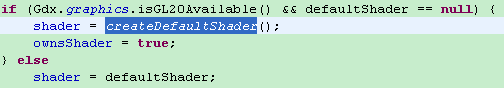
从截图中大家可以看到，第一个参数为SpriteBatch类型，最终的绘制由它来完成。那么OK，我们就来看看这个神秘的参数batch吧。跟踪流程，发现，我们的参数batch的根源，实际上是从Desktop3D的draw方法传下来的。



大家看截图中，蓝色高亮的部分，即是根源batch。然而这个全局变量是Desktop3D的父类Stage中的一个成员变量，在构造方法中创建。



OK，我们继续深挖它，解刨它，哈哈！最终通过对SpriteBatch构造方法的跟踪，发现构造方法中有这么一段程序：



继续深挖createDefaultShader这个方法，哈哈终于终于，我们发现了今天要讨论的主题“shader”。好吧，我承认，刚看到这一大串字符时，瞬间石化，有木有啊，表示不理解。而且，为何需要使用到这个呢？然后呢，这种百度，各种谷歌啊！搜索是个好东东！



2、shader究竟是个神马玩意儿呢？？

实际上，在图形硬件的可编程处理器上执行的一段程序代码，就叫做“着色器（shader）”，用来编写代码的编程语言，叫做“着色语言（Shader Language）”，其被用于对图形硬件进行编程，现在各种可编程图形硬件一般都提供着色器供开发人员使用，因此很有必要学习着色语言。

shader language被定位为高级语言，如，GLSL的全称是“High Level Shading Language”，Cg语言的全称为“C for Graphic”，并且这两种shader language的语法设计非常类似于C语言。不过高级语言的一个重要特性是“独立于硬件”，在这一方面shader language暂时还做不到，shader language完全依赖于GPU构架，这一特征在现阶段是非常明显的！任意一种shader language都必须基于图形硬件，所以GPU编程技术的发展本质上还是图形硬件的发展。在shader language存在之前，展示基于图形硬件的编程能力只能靠低级的汇编语言。

目前，shader language的发展方向是设计出在便捷性方面可以和C++\JAVA相比的高级语言，“赋予程序员灵活而方便的编程方式”，并“尽可能的控制渲染过程”同时“利用图形硬件的并行性，提高算法的效率”。Shader language目前主要有3种语言：基于OpenGL的GLSL，基于Direct3D的HLSL，还有NVIDIA公司的Cg 语言。

接下来回顾第一节的末尾的问题，为何我们的Launcher产品需要用到shader Language呢？因为我们的Launcher开发是基于开源游戏引擎Libgdx的，而Libgdx引擎底层采用OpenGL ES对图形进行渲染。也正是因为OpenGL ES的强力支撑，才能有更炫更酷的3D特效展现给用户。因此，我们这里谈论的shader language是“基于OpenGL的GLSL”。

OpenGL ES 着色语言是一种高级的图形编程语言。其源自于应用广泛的C语言，同时具有RendeMan以及其他着色语言的一些优良特性，易于被开发人员掌握。 OpenGL ES 的着色语言主要包括以下特性：

1、 OpenGL ES 2.0着色语言是一种高级的过程语言（注意，不是面向对象的）。

2、对顶点着色器、片元着色器使用的是同样的语言。

3、基于C/C++的语法及流程控制。

4、完美支持向量与矩阵的各种操作。

5、通过类型限定符来管理输入与输出。

6、拥有大量的内置函数来提供丰富的功能。

3、着色语言基础

3.1 数据类型概述

1. 标量

标量也被称为“无向量”，其值只具有大小，并不具有方向。标量之间的运算遵循简单的代数法则，如质量、密度、体积、时间以及温度等都属于标量。OpenGL ES着色语言支持的标量类型有布尔型(bool)、整形(int)和浮点型(float)

2. 向量

OpenGL ES着色语言中，向量可以看做是用同样类型的标量组成，其基本类型也分为bool、int和float三种。每个向量可以由2个、3个、4个相同的标量组成，具体情况如下：

|  |  |  |  |
| --- | --- | --- | --- |
| **向量类型** | **说明** | **向量类型** | **说明** |
| vec2 | 包含了2个浮点数的向量 | ivec4 | 包含了4个整数的向量 |
| vec3 | 包含了3个浮点数的向量 | bvec2 | 包含了2个布尔数的向量 |
| vec4 | 包含了4个浮点数的向量 | bvec3 | 包含了3个布尔数的向量 |
| ivec2 | 包含了2个整数的向量 | bvec4 | 包含了4个布尔数的向量 |
| ivec3 | 包含了3个整数的向量 |  |  |

向量在着色器代码的开发中有着十分重要的作用，可以很方面的存储以及存储颜色、位置、纹理坐标等不仅包含一个组成部分的量。开发中，有时可能需奥单独访问向 量中的某个分量，基本的语法为“<向量名>.<分量名>”，根据目的的不同，主要有以下几种用法：

将一个向量看做颜色时，可以使用r,g,b,a四个分量名，分别代表红、绿、蓝、透明度4个色彩通道。具体用法如下：

3. 矩阵

有一些基础的开发人员都知道，3D场景中的移位、旋转、缩放等变换都是由矩阵的运算来实现的。因此3D场景的开发中会非常多的使用矩阵，矩阵按尺寸分为2x2矩阵、3x3矩阵、4x4矩阵，具体情况如下表所示：

|  |  |
| --- | --- |
| **矩阵类型** | **说明** |
| mat2 | 2x2浮点数矩阵 |
| mat3 | 3x3浮点数矩阵 |
| mat4 | 4x4浮点数矩阵 |

对于矩阵的访问，可以讲矩阵作为列向量的数组来访问。如matrix为一个mat4，可以使用matrix[2]取到该矩阵的第三列，其为一个vec4；也可以使用matix[2][2]取得第三列向量的第3个分量。

4. 采样器

采样器是着色语言中不同于C语言的一种特殊的基本数据类型，其专门用来进行纹理采样的相关操作。一般情况下，一个采样器变量代表一幅或一套纹理贴图，其具体情况如下：

|  |  |
| --- | --- |
| **采样器** | **说明** |
| sampler2D | 用于访问二维纹理 |
| smapler3D | 用于访问三维纹理 |
| samplerCube | 用于访问立方贴图纹理 |

需要注意的是，与前面介绍的几种变量不同，采样器变量不能在着色器中初始化。一般情况下采样器变量都用uniform限定符来修饰，从宿主语言(如java)接受传递进着色器的值。

5. 结构体

OpenGL ES着色语言还提供了类似C语言中的用户自定义结构体，同样也是使用struct关键字进行声明。其基本用法如下：

struct info{

vec3 color;

vec3 position;

vec2 textureCoor;

}

6. 数组

声明数组的方式主要有两种，

在声明数组的同时，指定数组的大小：

vec3 position[20];

在声明数组时，也可以不指定数组的大小，但是必须符合下列两种情况之一。

引用数组之前，要再次使用第一种声明方式来生命该数组：

//声明了一个大小不定的vec3数组

vec3 position[];

//再次声明该数组，并且指定大小。

vec3 position[5];

代码中访问数组的下标都是编译时常量，这时编译器会自动创建适当大小的数组，使得数组尺寸足够存储编译器看到的最大索引值对应的元素。

//声明了一个大小不定的vec3数组

vec3 position[];

//position需要一个大小为4的数组

position[3] = vec3(3.0);

//position需要一个大小为21的数组

position[20] = vec3(6.0);

7. 空类型使用void表示，仅用来声明不返回任何值得函数。例如在顶点着色器以及片元着色器中必须存在的main函数就是一个返回值为空的函数，代码如下：

void main() {

}

3.2 数据类型的基本使用

1. 声明、作用域及初始化

变量的声明以及作用域与Java/C++语法类似，可以在任何需要的位置声明变量，同时其作用域也同样分为局部变量和全局变量：

int a,b; //声明了全局变量a和b

vec3 aPosition = vec3(1.0, 2.2, 3.3); //声明了全局变量aPosition并赋值

void myFunction() {

int c = 14; //声明了局部变量c并赋值

a = 4; //给全局变量a赋值

b = a \* c; //给全局变量b赋值

}

向量的初始化还有一些很灵活的技巧，巴友们体会一下下面的代码：

//声明浮点变量a并赋值

float a = 12.3;

//声明浮点变量b并赋值

float b = 11.4;

//声明2维向量va并赋值

vec2 va = vec2(2.3, 2.5);

//声明2维向量vb并赋值

vec2 vb = vec2(a, b);

//声明3维向量vc并赋值

vec3 vc = vec3(vb, 12.5);

//声明4维向量vd并赋值

vec4 vd = vec4(va, vb);

//声明4维向量ve并赋值, 相当于vec4(0.2 , 0.2 , 0.2, 0.2);

vec4 ve = vec4(0.2);

2. 运算符

与大多数编程语言类似，常见的运算符都可以在该语言中使用。下面按照优先级顺序列出了OpenGL ES着色语言中可以使用的运算符：

|  |  |  |  |
| --- | --- | --- | --- |
| **运算符** | **说明** | **运算符** | **说明** |
| [] | 用于索引 | . | 成员选择与混合 |
| ++ -- | 自加1与自减1后缀 | ++ -- | 自加1与自减1前缀 |
| - ! | 一元非与逻辑非 | \* / | 乘法与除法 |
| + - | 加法与减法 | < > <= >= | 关系运算符 |
| == != | 等于和不等于 | && | 逻辑与 |
| ^^ | 逻辑异或 | || | 逻辑或 |
| ?: | 选择 | = += -= \*= /= | 赋值运算符 |

3. 限定符

与其他的编程语言一样，着色器中对变量也有很多可选的限定符，主要如下：

|  |  |
| --- | --- |
| **限定符** | **说明** |
| attribute | 一般用于每个顶点都各不相同的量，如顶点位置、颜色等。 |
| uniform | 一般用于对同一组顶点组成的单个3D物体中所有顶点都相同的量，如当前光源的位置。 |
| varying | 用于从顶点着色器传递到片元着色器的量 |
| const | 用于声明常量 |

attribute限定符

顾名思义为属性限定符，其修饰的变量用来接收渲染管线传递进“顶点着色器”的当前待处理顶点的各种属性值。这些属性值每个顶点各自拥有独立的副本，用于描述顶点的各项特征，如顶点坐标、法向量、颜色、纹理坐标等。

用 attribute限定符修饰的变量，其值是由宿主程序批量出入渲染管线的，管线进行基本处理后再传递给“顶点着色器”。数据中有多少个顶点，管线就调用多少次“顶点着色器”，每次将一个顶点的各种属性数据传递给顶点着色器中对应atribute变量。因此，顶点着色器每次执行将完成对一个顶点各项属性数据的处理。

从上面的介绍中可以看出，attribute限定符只能用于顶点着色器中，不能在片元着色器中使用，且attribute限定符只能用来修饰浮点数标量、浮点向量以及矩阵变量，不能用来修饰其他类型的变量。下面的代码片段给出了在顶点着色器中正确使用attribute限定符的情况：

//顶点位置

attribute vec3 aPosition;

//顶点法向量

attribute vec3 aNormal;

前面已经提到，对于用attribute限定符修饰的变量的值是由宿主程序批量传入渲染管线的，相关代码如下：

// 声明顶点位置属性引用

int maPositionHandle;

// 获取顶点位置属性引用的值,

// mProgram为着色器程序ID，

// aPosition为着色器中对应属性的变量名称。

maPositionHandle = GLES20.glGetAttribLocation(mProgram, "aPosition");

// 将顶点位置数据传入渲染管线

// maPositionHandle：顶点位置属性引用

// 3：每顶点一组的数据个数(这里是X,Y,Z坐标，因此为3)

// GLES20.GL\_FLOAT：数据类型

// false：是否格式化

// 3 \* 4：每组数据的尺寸，这个魅族3个浮点数值(X,Y,Z坐标)，每个浮点数4个字节

// mVertexBuffer：存放了数据的缓冲

GLES20.glVertexAttribPointer(maPositionHandle, 3, GLES20.GL\_FLOAT, false, 3 \* 4, mVertexBuffer);

uniform限定符

uniform为一致变量限定符，一致变量指的是对于同一组顶点组成的单个3D物体中所有顶点都相同的量。Uniform变量可以用在顶点着色器或片元着色器中，其支持用来修饰所有的基本数据类型。与属性限定符类似，一致变量的值也是从宿主程序传入的。

下面的代码片给出了在顶点着色器或片元着色器中正确使用uniform限定符的情况:

//总变换矩阵

uniform mat4 uMVPMatrix;

//变换矩阵

uniform mat4 uMMatrix;

//光源位置

uniform vec3 uLightLocation;

//摄像机位置

uniform vec3 uCamera;

将一致变量的值由宿主程序传入渲染管线的代码如下：

//总变换矩阵一致变量引用

int muMVPMatrixHandle;

//获取着色器程序中总变换矩阵一致变量的引用

muMVPMatrixHandle = GLES20.glGetUniformLocation(mProgram, "uMVPMatrix");

//通过一致变量引用将一致变量值传入渲染管线

GLES20.glUniformMatrix4fv(muMVPMatrixHandle, 1, false, Triangle.getFianlMatrix(mMMatrix), 0);

需要注意的是，随一致性变量类型不同将值传入渲染管线的方法也有所不同，这些方法的名称都以glUniform开头，常用的如下所列：

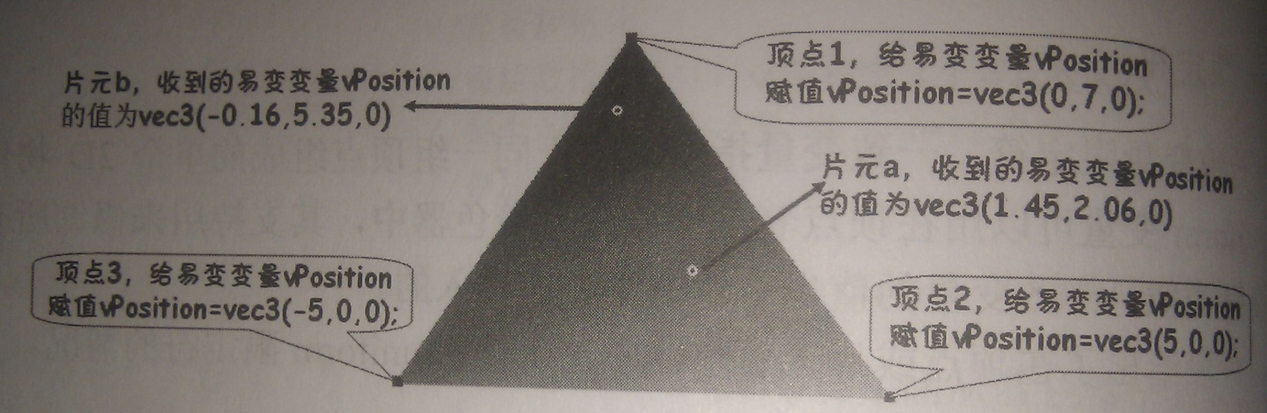
glUniform**N**f/glUniform**N**fv方法，将N个浮点数传入管线，以备管线传递给由N个浮点数组成的一致变量，N的取值为1,2,3,4。

glUniform**N**i/glUniform**N**iv方法，将N个整数传入管线，以备管线传递给由N个整数组成的一致变量，N的取值为1,2,3,4。

glUniformMatrix**N**fv方法，将N \* N的矩阵传入管线，以备管线传递给N \* N矩阵类型的一致变量，N的取值为2,3,4。

varying限定符

想要将顶点着色器中的信息传入到片元着色器中，必须使用varying限定符。varying限定符修饰的全局变量又称为易变变量，易变变量可以看成是顶点着色器和片元着色器之间的动态接口，方便顶点着色器与片元着色器之间信息的传递。下图给出了易变变量的工作原理：



从上图可以看出，首先顶点着色器在每个顶点中都对易变变量vPosition进行了赋值。接着在片元着色器中接受易变变量vPosition的值时得到的并不是某个顶点赋的特定值，而是根据片元所在的位置以及图元中各个顶点的位置进行差值计算产生的值。

如图中顶点1、2、3的vPosition值分别为vec3(0,7,0)、vec3(5,0,0)、vec3(-5,0,0),则插值后片元a的vPosition值为vec3(1.45, 2.06, 0)。

从上述介绍中可以看到，光栅化后产生了多少个片元，就会插值计算出多少套易变变量。同时渲染管线就会调用多少次片元着色器。可以看出，3D物体的渲染中，片元着色器执行的次数会大大超过顶点着色器。因此GPU硬件中配置的片元着色器硬件数量往往多于顶点着色器硬件数量以提高渲染速度。

const限定符

用const限定符修饰的变量其值是不可以变的，也就是常量，又称为编译时常量。编译时常量在声明的时候必须进行初始化。例如：

const int tempx = 1;

3.3 流程控制

OpenGL ES 着色语言提供了3种流程控制的方式，分别是if-else、while(do-while)循环、与for循环。这些控制语句的语法基本与Java一样，所以接受起来很简单。

3.4 内建变量

着色器代码的开发中会用到很多变量，其中大部分可能是由开发人员根据需求自定义的，但着色器中也提供了一些用来满足特定需求的内建变量。这些内建变量不需要声明就可以使用，一般用来实现渲染管线固定功能部分与自定义顶点或片元着色器之间的信息交互。

1. 顶点着色器中的内建变量

顶点着色器中的内建变量主要是输出变量，包括gl\_Position、gl\_PointSize等。在顶点着色器中应该根据需要给这些内建变量赋值，以便由渲染管线中的图元装配与光栅化等后续固定功能阶段进行进一步的操作。

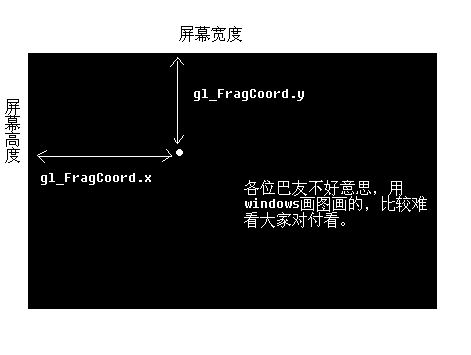
gl\_Position：顶点着色器从应用程序中获得原始的顶点位置的数据，这些原始的顶点数据在顶点着色器中经过平移、旋转、缩放等数学变换后，生成新的顶点位置。

gl\_PointSize：顶点着色器中可以计算一个点的大小(单位为像素)，并将其赋值给gl\_PointSize(标量，float类型)以传递给渲染管线，如果没有明确赋值的话，就是采用默认值1，gl\_PointSize的值一般只有在采用了点绘制方式之后才有意义。

2. 片元着色器中的内建变量

片元着色器中的内部按输入变量主要有gl\_FragCoord以及gl\_FrontFacing，这两个内建变量都是只读的，由渲染管线中片元着色器之前的阶段生成。其具体含义如下：

gl\_FragCoord：vec4类型，其中含有当前片元相对于窗口位置的坐标值X,Y,Z。如下图所示：

[](http://www.apkbus.com/data/attachment/album/201208/20/2323115lodtkdud005f5db.png)

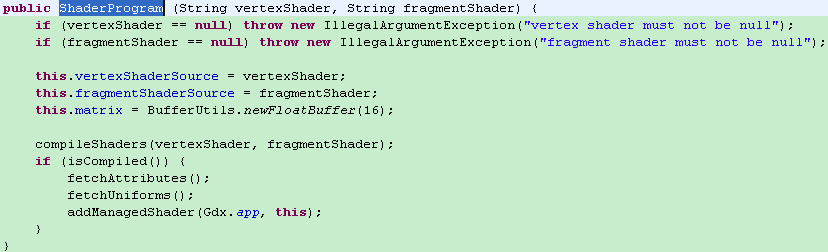
gl\_FrontFacing：布尔型内建变量，通过读取内建变量的值可以判断正在处理的片元是否属于在光栅化阶段生成此片元的对应图元的正面。如果属于正面，那么该值为true，否则为false。

OK，经过了漫长、枯燥以及乏味的基础理论知识后，我们再次回到第一节的末尾，再重新看createDefaultShader方法截图时，是不是已经对那一系列的字符符号不那么的陌生了。它们实际上就是用shader language写的一段程序代码，那么接下来我们一起来学习学习，Libgdx究竟是怎样来利用着色语言来编写着色器，对图形进行渲染的？

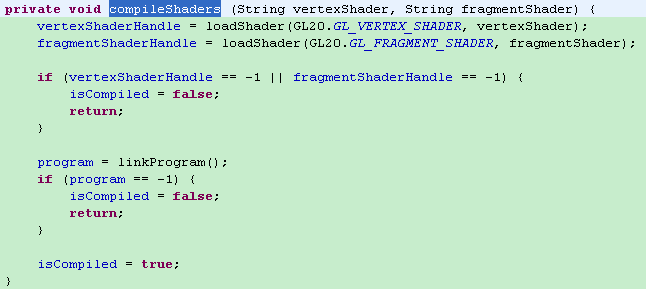
3、Libgdx中着色语言的使用

方法createDefaultShader中，SpriteBatch已经定义好了“顶点着色器（vertexShader）”和“片元着色器（fragmentShader）”。接下来我们看看，这两个着色器程序是怎样被上传到GPU，并参与后续整个渲染工作的。

首先，两个着色器被作为参数传入方法构造器ShaderProgram中，



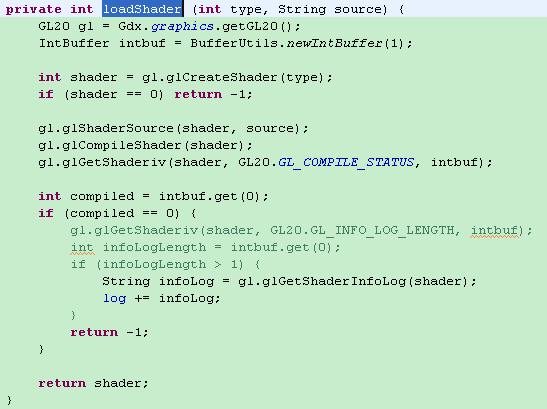
接着，通过方法compileShaders将着色器加载到GPU中并进行编译，最后将顶点着色器和片元着色器程序链接为一个整体的着色器程序。



此方法用于将两个着色器链接为一个整体的着色器程序。

此方法用于对着色器进行加载和编译

加载着色器并编译。

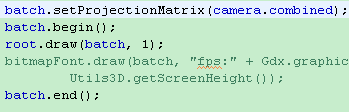


将顶点着色器和片元着色器链接为一个整体着色器程序。

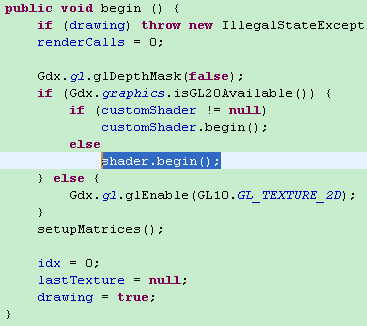


此刻，大家应该还会很疑惑，究竟我们Launcher的图形都是怎样在着色器的帮助下渲染到屏幕上的呢？接下来，我们继续一起来探索下。

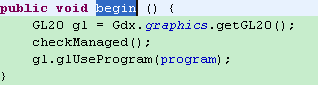
我们先回到Desktop3D中的draw方法：



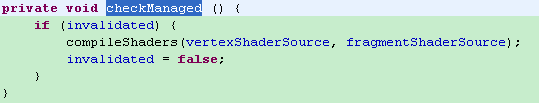
batch先设置了投影矩阵，然后看看begin方法都做了什么吧。



注意蓝色高亮部分，shader.begin();这里，我们继续看看都做了什么：

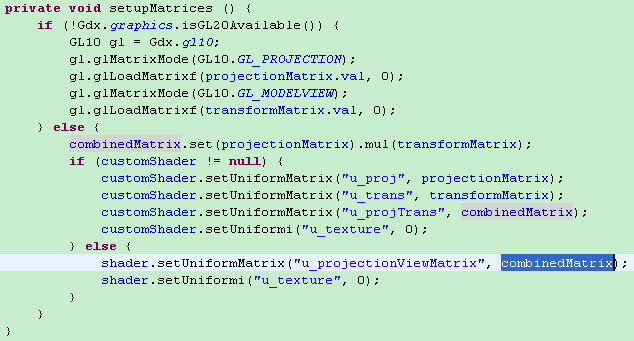


先由方法checkManaged检查着色器是否已经创建成功。

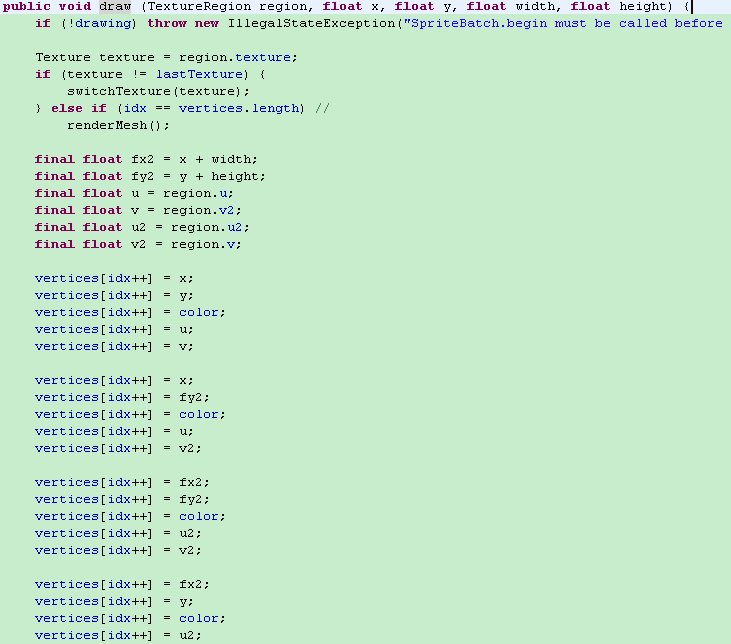


然后通过OpenGL ES方法gl.glUseProgram使用着色器。

最后在方法setupMatrices中，对投影矩阵和变换矩阵进行运算，将最终的矩阵combinedMatrix传入shader中。



当一切初始化工作好了以后，经由方法开始遍历绘制所有的图形，最终调用到SpriteBatch中的draw方法，draw方法中，会依次将图形的位置坐标，颜色值，纹理坐标记录到vertices数组中。



最终经由方法renderMesh进行绘制。

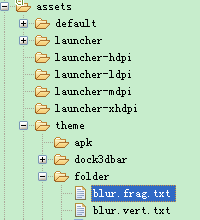


4、着色语言使用示例

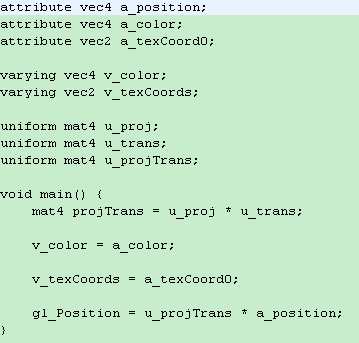
利用着色语言做文件夹模糊效果，最终效果截图如下。



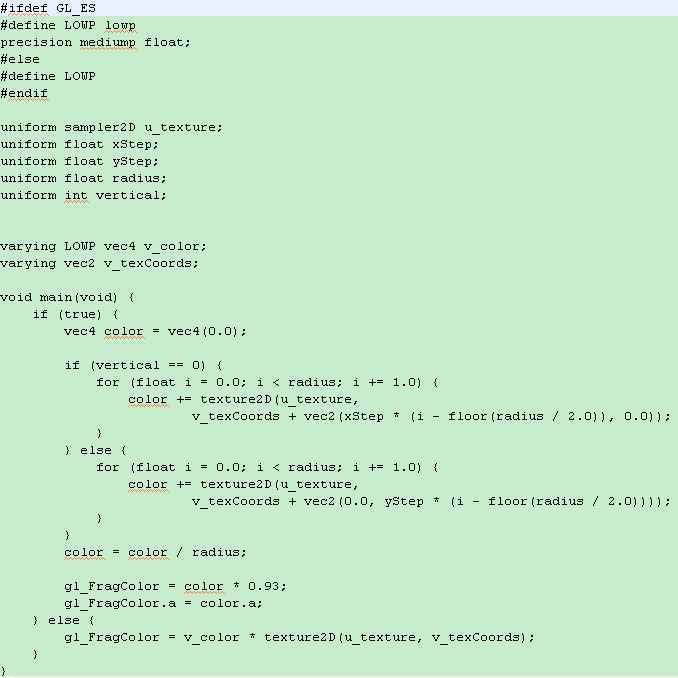
这里，我们将用着色语言开发的顶点着色器和片元着色器作为脚本文件放入工程的assets目录下。



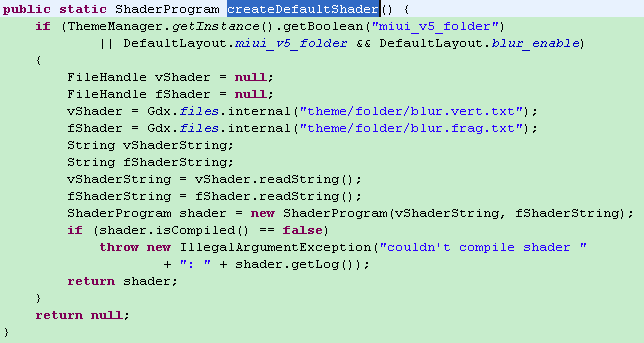
blur.vert.txt文件即是“顶点着色器”，程序截图如下：



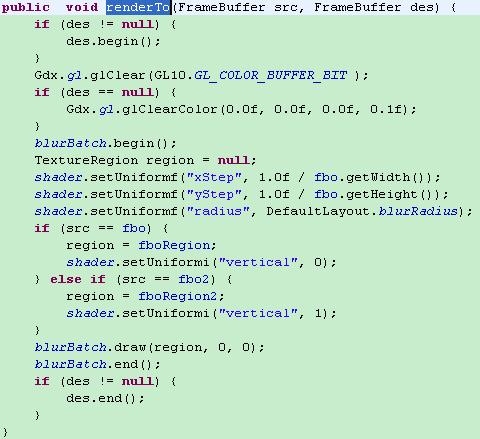
blur.frag.txt文件即是“片元着色器”，程序截图如下：



接着我们来创建着色器：



绘制



至此，对于着色语言的初步介绍已经告一段落。