**OpenGL ES基础知识以及shader滤镜的实现**

1. OpenGL基础知识
   1. OpenGL ES简介

OpenGL(Open Graphics Library)定义了一个跨编程语言、跨平台编程的专业图形程序接口。可用于二维或三维图像的处理和渲染，它是一个功能强大、调用方便的底层图形库。对于嵌入式设备，其提供了OpenGL ES(OpenGL for Embeddled Systems)版本，该版本是针对手机、Pad等嵌入式设备而设计的，是OpenGL的一个子集。

OpenGL主要是做图形图像处理的库，尤其是在移动设备上进行图形图像处理，它的性能优势更能体现出来。GLSL(OpenGL Shading Language)是OpenGL的着色器语言，开发人员利用这种语言编写程序运行在GPU（Graphic Processor Unit,图形图像处理单元，可以理解为是一种高并发的运算器）上以进行图像的处理和渲染。GLSL着色器代码分为两个部分，即Vertex Shader(顶点着色器)与Fragment Shade（片元着色器）两部分，分别完成各自在OpenGL渲染管线中的功能。

* 1. GLSL的基础知识

1.2.1 GLSL的修饰符和基本数据类型

##### 修饰符

具体如下：

* const:用于声明非可写的编译时常量变量。
* attribute:用于经常更改的信息，只能在顶点着色器中使用。
* uniform:用于不经常更改的信息，可用于顶点着色器和片元着色器。
* varying:用于修饰从顶点着色器向片元着色器传递的变量

##### 基本数据类型

int、float、bool,这些与C语言都是一致的，需要强调的一点就是，这里面的float是有一个修饰符的，即可以指定精度。三种修饰符的范围（范围一般视显卡而定）和应用情况具体如下。

* highp:32bit,一般用于顶点坐标（vertex Coordinate）。
* medium:16bit,一般用于纹理坐标（texure Coordinate）。
* lowp:8bit,一般用于颜色显示（color）。

##### 向量类型

向量类型是Shader中非常重要的一个数据类型，因为在做数据传递的时候需要经常传递多个参数，相较于写多个基本数据类型，使用向量类型是非常好的选择。列举一个最经典的例子，要将物体坐标和纹理坐标传递到Vertex Shader中，用的就是向量类型，每一个顶点就是一个四维向量，在Vertex Shader中利用这两个四维向量即可完成自己的纹理坐标映射操作。声明方式如下（GLSL代码）：

attribute vec4 position;

##### 矩阵类型

有一些效果器需要开发者传入矩阵类型的数据，比如后面会接触到的怀旧效果器，就需要传入一个矩阵来改变原始的像素数据。声明方式如下：

uniform lowp mat4 colorMatrix;

上面的代码表示了一个4x4的浮点矩阵，如果是mat2就是2x2的浮点矩阵，如果是mat3就是3x3的浮点矩阵。若要传递一个矩阵到实际的Shade中，则可以直接调用如下函数：

glUniformMarix4fv(mColorMatrixLocation,1,false,mColorMatrix);

##### 纹理类型

一般仅在Fragment Shader中使用这个类型，二维纹理的声明方式如下

uniform sample2D texSampler;

当客户端接收到这个句柄时，就可以为它绑定一个纹理，代码如下：

glActiveTexture(GL\_TEXTURE0);

glBindTexture(GL\_TEXTURE\_2D,texId);

glUniformli(mGLUniformTexture,0);

上述代码第一行激活的是哪一个纹理句柄，第三行代码中的第二个参数需要传递对应的Index,就像代码中激活的纹理句柄是GL\_TEXTURE0,对应的Index就是0，如果激活的纹理句柄是GL\_TEXTURE1,那么对应的Index就是1，在不同的平台上句柄的个数也是不一样，但是一般都会在32个以上。

##### varying

这个修饰符变量用于在Vertex Shader和Fragment Shader之间传递函数。首先在顶点着色器中声明这个类型的变量代表纹理的坐标点，并且对这个变量进行赋值，代码如下：

attribute vec2 texoord;

varying vec2 v\_texcoord;

void main(void)

{

// 计算顶点坐标

v\_texcoord = texcoord;

}

紧接着在Fragment Shader中也声明同名的变量，然后使用texture2D方法取出二维纹理中该纹理坐标点上的纹理像素值，代码如下

varying vec2 v\_texcoord;

vec4 texel = texture2D(texSampler,v\_texcoord);

取出了该坐标点上的像素值之后，就可以进行像素变化操作了，比如说提高对比度，最终将改变的像素值复制给gl\_FragColor。

1.2.2 GLSL的内置函数与内置变量

首先来看内置变量，最常见的是两个Shader的输出变量。  
先来看Vertex Shader的内置变量：

vec4 gl\_position;

上述代码用来设置顶点转换到屏幕坐标的位置，Vertex Shader一定要去更新这个数值。另外还有一个内置变量，

float gl\_pointSize;

在粒子效果的场景下，需要为粒子设置大小，改变该内置变量的值就是为了设置每一个粒子矩形的大小。

其次是Fragment Shader的内置变量，代码如下

vec4 gl\_FragColor;

上述代码用于指定当前纹理坐标所代表的像素点的最终颜色值。然后是内置函数，具体的函数可以去官方文档中查询，这里仅介绍几个常用的函数。

* abs(genType x):绝对值函数。
* floor(genType x):向下取整函数。
* ceil(genType x):向上取整函数。
* mod(genType x,genType y):取模函数。
* min(genType x,genType y):取的最小值函数。
* max(genType x,genType y)：取得最大值函数。
* clamp(genType x,genType y,genType z):取得中间值函数。
* step(genType x,genType y):如果x<edge,则返回0.0，否则返回1.0。
* smoothstep(genType edge0,genType edge1,genType x):如果x<=edge0,则返回0.0；如果x>=edge1,则返回1.0；如果edge0<x<edge1,则执行0-1的平衡插值。
* mix(genType x,genType y,genType a):返回线性混合的x和y,用公式表示为x(1-a)+ya,这个函数在mix两个纹理图像的时候非常有用。

GLSL的控制流与C语言非常类似，既可以使用for、while、以及do-while实现循环，也可以使用if和if-else进行条件分支的操作。

### 3.创建显卡执行程序

如何将Shader传递给OpenGL的渲染管线。

##### 1）创建shader的过程

第一步是调用glCreateShader方法创建一个对象，作为shade的容器，该函数会返回一个容器的句柄，函数的原型如下：

GLuint glCreateShader(GLenum shaderType);

函数原型中的参数shaderType有两种类型，当要创建VertexShader时，开发者应该传入类型GL\_VERTEX\_SHADER;当要创建FragmentShader时，开发者应该传入GL\_FRAGMENT\_SHADER类型。

下一步就是为创建的这个shader添加源代码，源代码就是根据GLSL语法和内嵌函数编写的两个着色器程序（Shader）,其为字符串类型。函数原型如下：

void glShaderSource(GLuint shader,int numOfStrings,const char \*\*strings,int \*lenOfStrings)

上述函数的作用就是把开发者编写的着色器程序加载到着色器句柄所关联的内存中。

最后一步就是编译该Shader,编译Shader的函数原型如下：

void glCompileShader(GLuint shader);

待编译完成之后，还需要验证该Shader是否编译成功了。那么，应该如何验证呢？使用下面的函数即可进行验证：

void glCetShaderiv(GLuint shader,GLenum pname,GLint \*params);

其中第一个参数就是需要验证的Shader句柄；第二个参数值是需要验证的Shader状态值，这里一般是验证编译是否成功，该状态值一般是选取GL\_COMPILE\_STATUS;第三个参数是返回值。当返回1时，则说明该Shader是编译成功的；如果为0，则说明该Shader没有被编译成功，此时获取的是改Shader的另外一个状态，该状态值应该选取GL\_INFO\_LOG\_LENGTH,返回值返回的则是错误原因字符串的长度，我们可以利用这个长度分配出一个buffer,然后调用获取Shader的InfoLog函数，函数原型如下：

void glGetShaderInfoLog(GLuint object,int maxLen,int \*len,char \*log);

之后可以把InfoLog打印出来，以帮助我们调试实际Shader中的错误。

##### 2）如何通过两个Shader来创建显卡可执行程序

首先创建一个对象，作为程序的容器，此函数将返回容器的句柄。函数原型如下：

GLuint glCreateProgram(void);

第二步是把前文编译的Shader附加到刚刚创建的程序中，调用的函数名称如下：

void glAttachShader(GLuint program,GLuint shader);

第一个参数就是传入上一步返回的程序容器的句柄，第二个参数就是编译的Shader容器的句柄，当然要为每一个Shader都调用一次这个方法才能把两个Shader都关联到Program中去。  
最后一步就是链接程序了，链接函数原型如下：

void glLinkProgram(GLuint program);

传入参数就是程序容器的句柄，那么这个程序有没有链接成功呢？OpenGL提供了一个函数来检查该程序的状态，函数原型如下：

void glGetProgramiv（GLuint program,GLenum pname,GLint \*params）;

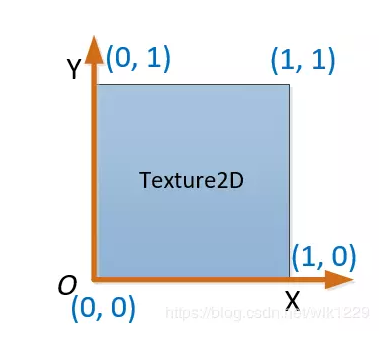
第一个参数就是传入程序容器的句柄，第二个参数代表需要检查该程序的哪一个状态，这里传入的是GL\_LINK\_STATUS,最后一个参数就是返回值。返回值为1则代表链接成功，如果返回值为0则代表链接失败。如果想获取具体的错误信息，第二个参数要传递GL\_INFO\_LOG\_LENGTH,代表获取该程序的InfoLog的长度，获取到长度之后我们分配出一个char \*的内存空间以获取InfoLog,函数原型如下：

void glGetProgramInfoLog(GLuint object,int maxLen,int \*len,char \*log);

该函数返回InfoLog之后可以将其打印出来。

1. OpenGL ES中的纹理

OpenGL中的纹理可以用来表示图像、照片、视频画面等数据，在视频渲染中只需要处理二维的纹理，每个二维纹理都由很多小的纹理元素组成，它们都是小块数据，类似于前面章节所说的像素点。要使用纹理，最常用的是直接从一个图像文件加载数据。  
 为了访问到每一个纹理元素，每个二维纹理都有自己的坐标空间，其范围是从左下角的（0，0）到右上角的（1，1）。



我们所熟知的不论是计算机还是手机的屏幕坐标系，x轴从左到右都是从0到1，y轴从上到下是从0到1，与图片的存储恰好是一致的，假设图片的存储是把所有的像素点都存储到一个大数组中，图片存储的第一个像素点是左上角的像素点（即第一排第一列的像素点），然后第二个像素点（第一排第二列）存储在数组的第二个元素中，那么，这里的坐标和OpenGL中的纹理坐标正好做了一个180度的旋转。

下面再来看一下如何加载一张图片作为OpenGL中的纹理，首先要在显卡中创建一个纹理对象，OpenGL ES提供的方法原型如下：

void glGenTextures(GLsizei n,GLuint \*textures)

这个方法传递进去的第一个参数是需要创建几个纹理对象，并且把创建好的纹理对象的句柄放到第二个参数中去，所以第二个参数是一个数组（指针）的形式。如果只需要创建一个纹理对象的话，则只需要声明一个GLuint类型的texId,然后针对该纹理ID取地址，并将其作为第二个参数，就可以创建出这个纹理对象了，代码如下：

glGenTextures(1,&texId);

执行完这行代码之后，就会在显卡中创建一个纹理对象，并且把该纹理对象的返回给texId变量。紧接着开发者要操作该纹理对象，但是在OpenGL ES的操作过程中必须告诉OpenGL ES具体操作的是哪一个纹理对象，所以必须调用OpenGL ES提供的一个绑定纹理对象的方法，调用代码如下：

glBindTexture(GL\_TEXTURE\_2D,texId);

执行完上面这行代码之后，下面的操作就都是针对于texId这个纹理对象的了，最终对该纹理的对象操作完毕之后，我们可以调用一次解绑定的代码：

glBindTexture(GL\_TEXTURE\_2D,0);

这行代码执行完毕之后，代表开发者不会对texId纹理做任何操作了，所以上面这行代码只在最后的时候才调用。

接下来就是最关键的部分，即如何将本地磁盘中的一个PNG的图片上传到显卡中的这个纹理对象上。在将图片上传到这个纹理上之前，首先应该要对这个纹理对象设置一些参数，具体参数有哪些？其实就是纹理的过滤方式，当纹理对象（可以理解为一张图片）被渲染到物体表面上的时候（实际上OpenGL绘制管线将纹理的元素映射到OpenGL生成的片段上的时候），有可能要被放大或者缩小，而当其放大或者缩小的时候，具体应该如何确定每个像素是如何被填充的，就由开发者配置的纹理对象的纹理过滤器来指明。  
magnification(放大)：

glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MAG\_FILTER,GL\_LINEAR);

minification(缩小)

glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_MIN\_FILTER,GL\_LINEAR);

一般在视频的渲染与处理的时候使用GL\_LINEAR这种过滤方式，该过滤方式称为双线性过滤，可使用双线性插值平滑像素之间的过渡，OpenGL会使用四个临近的纹理元素，并在它们之间用一个线性插值算法做插值，该过滤方式是最主要的过滤方式，当然OpenGL中还提供了另外几种过滤方式。常见的有GL\_NEAREST,称为最临近过滤，该方式将为每个片段选择最近的纹理元素，但是当其放大的时候会有很严重的锯齿效果（因为相当于将原始的直接放大，其实就是降采样），而当其缩小的时候，因为没有足够的片段来绘制所有的纹理单元，（这个是真正的降采样），许多细节都会丢失；其实OpenGL还提供了另外一种技术，称为MIP贴图，但是这种技术会占用更多的内存，其优点是渲染也会更快。当缩小和放大到一定程度之后效果也比双线性过滤的方式更好，但是其对纹理的尺寸及内存的占用是有一定限制的，不过，在视频的处理以及渲染的时候不需要放大或者缩小这么多倍，所以在进行视频的处理以及渲染的场景下，MIP贴图并不适用。

紧接着来看一下对于纹理对象的另外一个设置，也就是在纹理坐标系的s轴和t轴的纹理映射过程中用到的重复映射或者简约映射的规则，代码如下：

glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_WRAP\_S,GL\_CLAMP\_TO\_EDGE);

glTexParameteri(GL\_TEXTURE\_2D,GL\_TEXTURE\_WRAP\_T,GL\_CLAMP\_TO\_EDGE);

上述代码所表示的含义是，将该纹理的s轴和t轴的坐标设置为GL\_CLAMP\_TO\_EDGE类型，因为纹理坐标可以超出（0，1）的范围，而按照上述设置规则，所有大于1的纹理都要设置为1，所有小于0的纹理都要置为0。

接下来，就是将PNG素材的内容放到该纹理对象上，OpenGL的大部分纹理一般都只接受RGBA类型的数据（否则还得去转化），所以我们需要对PNG这种压缩格式进行解码操作，如果想要采用一种更通用的方式，那么可以引用libpng库来进行解码操作，当然也可以使用各自平台的API进行解码，最终可以得到RGBA数据。待得到RGBA数据之后，记为uint8\_t数组类型的pixels,然后执行如下操作：

glTexImage2D(GL\_TEXTURE\_2D,0,GL\_RGBA,width,height,0,GL\_RGBA,GL\_UNSIGNED\_BYTE,pixels);

这样就可以将该RGBA的数组表示的像素内容上传到显卡里面texId所代表的纹理对象中去了，以后只要使用该纹理对象，其实表示的就是这个PNG图片。

OpenGL中的纹理表示如何为物体增加细节，现在我们已经准备好了该纹理，那么如何把这张图片（或者说这个纹理）绘制到屏幕上呢？首先来看一下OpenGL中的物体坐标系，物体坐标系中x轴从左到右是从-1到1变化的，y轴从下到上是从-1到1变化的，物体的中心点恰好是（0，0）的位置。

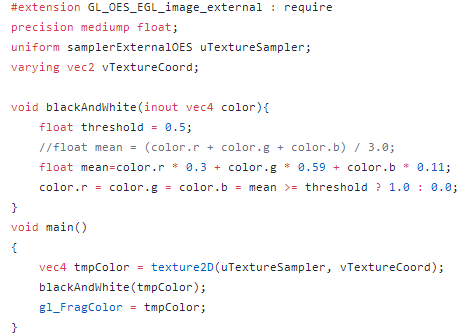
1. 实践与应用

根据所学的知识，在Android平台实现了三种滤镜，分别为二值化、边缘提取以及九分屏的实现。

Android平台实现以上滤镜仅仅需要改变相关的着色代码，接口是不需要变动的，其相关着色代码为：

二值化代码：



边缘提取代码：



九分屏代码：

