Objective:

1、了解cocos2dx渲染流程，学习如何在cocos2dx引擎下编写OpenGL代码

2、了解OpenGL的渲染管线，学习Vertex shader和Fragment shader的编写，学习shader与c++程序如何传做数据传递

作者：织法

在手游如雨后春笋般上线的今天，手游的同质化极其严重。不同游戏间差别可能只是一层皮，在这种情况下，一款游戏若想获得成功，就必须有所突破，图形就是一个很好的突破口。

在中国，Cocos2dx是当下最流行的2D手游开发引擎，其最大的优点之一就是开源。正因为这一特点，我们可以轻松的添加一些自己想要的效果，尤其在图形方面。

下面，我将讲解如何在cocos2dx下使用OpenGL。

目录

[一． Cocos2dx的渲染流程 3](#_Toc398284749)

[1.1 Node的Visite函数 3](#_Toc398284750)

[1.2 Sprite的draw函数 6](#_Toc398284751)

[1.3 render对象浅析 6](#_Toc398284752)

[1.4 AutoBatch原理讲解 10](#_Toc398284753)

[1.5 自定义RenderCommand 13](#_Toc398284754)

[二． OpenGL简介 14](#_Toc398284755)

[2.1 OpenGL 渲染管线 14](#_Toc398284756)

[2.1.1 Vertex Shading 15](#_Toc398284757)

[2.1.2 Primitive Assembly 15](#_Toc398284758)

[2.1.3 Clipping 16](#_Toc398284759)

[2.1.4 Rasterization 16](#_Toc398284760)

[2.1.5 fragment shading 17](#_Toc398284761)

[2.1.6 Pre-Fragment Operations 17](#_Toc398284762)

[2.2 一个简单的着色器程序 18](#_Toc398284763)

[2.3 VAO与VBO 21](#_Toc398284764)

[2.4 彩色的三角形 23](#_Toc398284765)

[2.5 shader语法简介 25](#_Toc398284766)

[2.5.1 变量的声明 25](#_Toc398284767)

[2.5.2 变量的转换 26](#_Toc398284768)

[2.5.3 Vector Component Accessors 27](#_Toc398284769)

[2.5.4 shader的结构 27](#_Toc398284770)

[2.5.5shader中的内置变量与内置函数 28](#_Toc398284771)

[2.5.6 cocos2dx提供的内置变量 28](#_Toc398284772)

[三 3D图元的绘制 30](#_Toc398284773)

[3.1 MVP 3D 矩阵变幻 30](#_Toc398284774)

[3.1.1 投影矩阵 31](#_Toc398284775)

# Cocos2dx的渲染流程

要想了解如何在cocos2dx引擎下编写OpenGL代码，首先要知道这些代码应当写在哪里。而cocos2dx这个引擎的渲染模块，正是基于OpenGL的API来编写，我们可以研究一下他的渲染流程。

## Node的Visite函数

打开cocos2dx引擎中的CCNode.cpp文件，翻到Visit处，可看到以下代码，如图1.1所示。

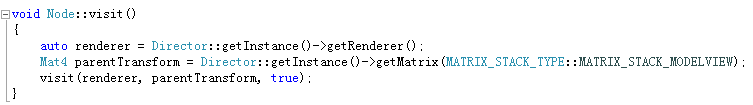


图1.1 Visit

Node的Visite先从导演类中取得渲染器Render对象，在从矩阵栈中获取当前的modelView矩阵。再把他们传入带3个参数的Visit中。下面我们看看这个函数都做了些什么，如图1.2,1.3所示

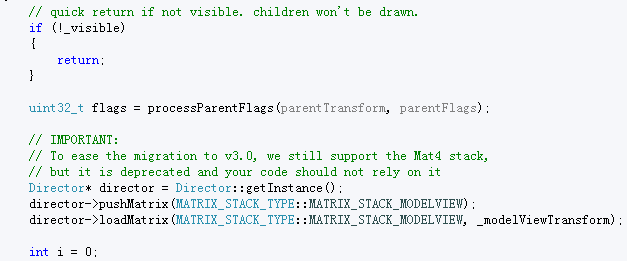


图1.2 visit1

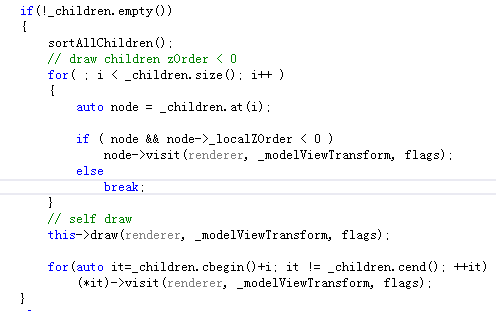


图1.3 visit2

我们可以看到，当visible为false时，函数直接被return掉。之后先向栈中推入一个矩阵，在把当前的modelView矩阵放入栈中。而后就是我们熟悉的遍历过程。先遍历Z-order小于0的，绘制，而后是大于0的。这也就揭示了zorder是如何控制渲染层级的，也揭示节点树的绘制顺序。

## 1.2 Sprite的draw函数

下面，我们看一下精灵类draw到底做了些什么。打开引擎的CCSprite.cpp文件，翻到draw处，其代码如图1.4所示。

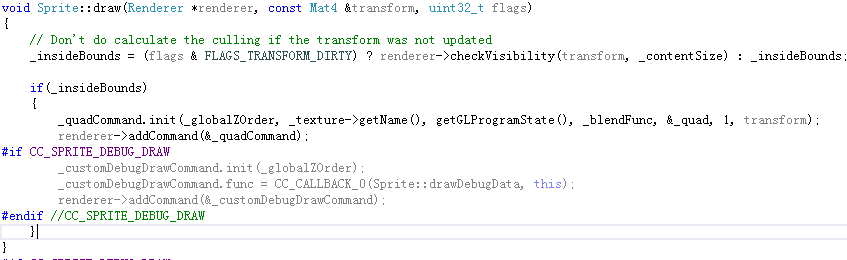


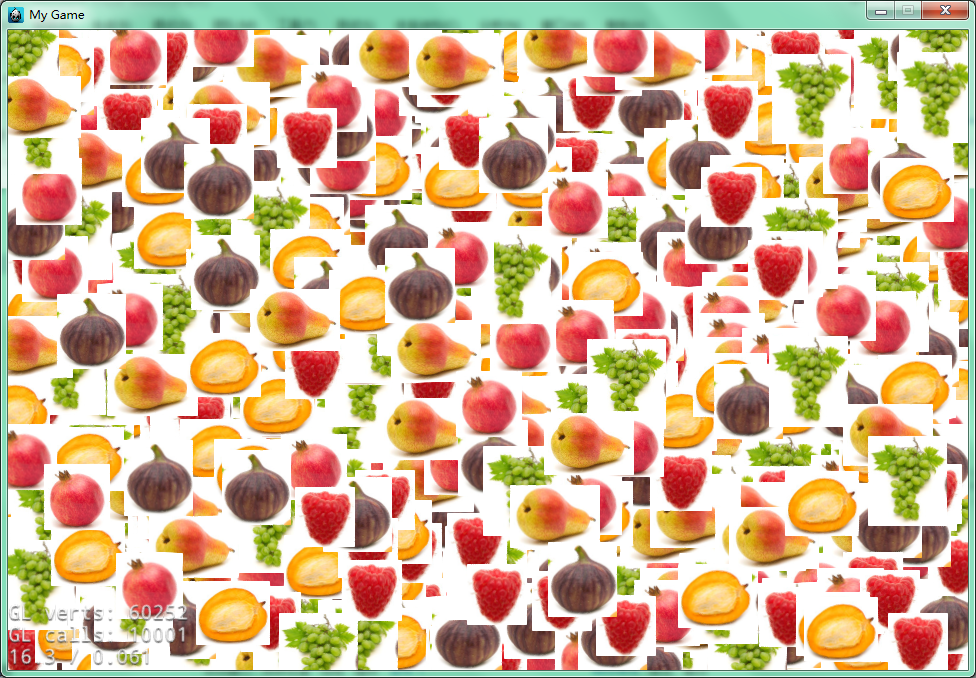
图1.4 Sprite::draw

我们可以看到，传说中的渲染剔除，就是在这里做的。而且就是一个AABB的判断。这函数的主要功能，就是把相关信息放入一个command中，传到了render对象中。

## 1.3 render对象浅析

下面，我们将详细分析下这个render对象。

首先，我们找6张可爱的水果图片，在helloworld中重复添加这些图片，运行效果如图1.5所示。



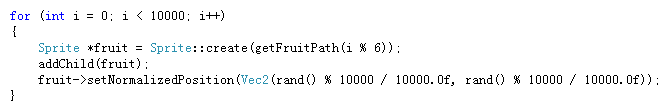


图1.5 小水果

在Renderer的render函数中打上断点，如图1.6所示。

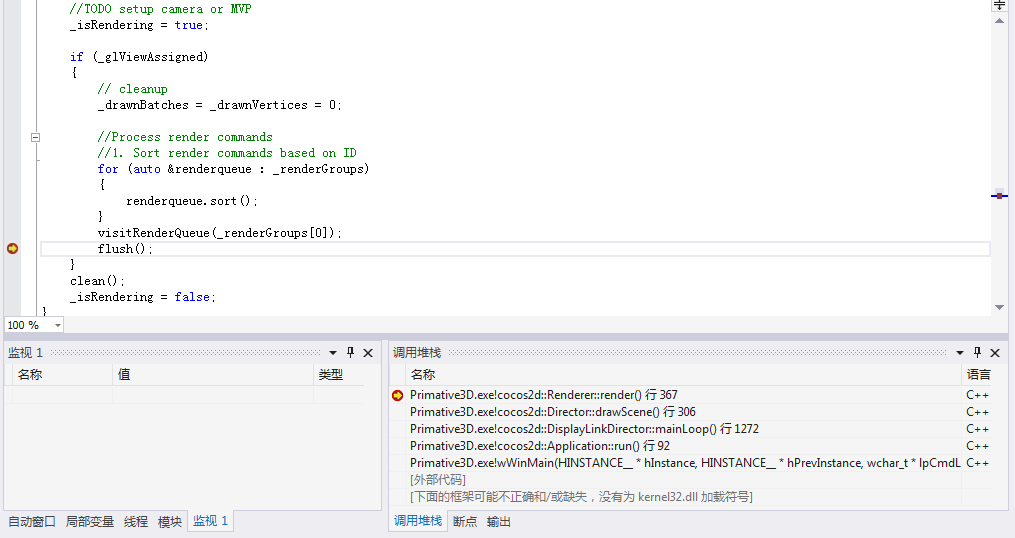


图1.6 render函数

在Renderer的render函数中打上断点，如图1.6所示。我们查看调用堆栈的上一级，可以看到我们的大导演在每帧的循环中做了这些事情。如图1.7,1.8,1.9所示。首先更新scheduler，而后让当前的场景visit（所有节点draw函数就在这里调用的），最后调用render，推出当前矩阵，交换背缓冲。

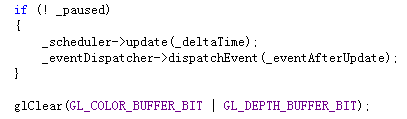


图1.7 大导演的drawScene (1)

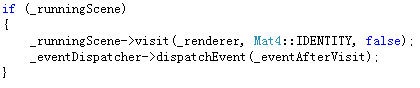


图1.8 大导演的drawScene (2)

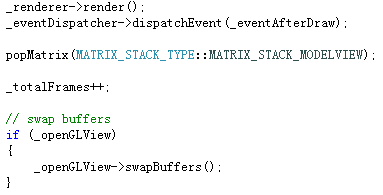


图1.6 大导演的drawScene (3)

而后我们再回到render的函数中，看一下他的visitRenderQueue函数，如图1.7，1.8所示。我们首先看最常用的Quard的渲染。如果当前缓存的Quard顶点数量还没达到VBO\_SIZE(10922)，就把这些顶点数据拷贝到quard数组中，再用command中保存modelView矩阵对这些顶点进行变换。而对于其他的命令，就是调用他的execute方法。对于自定义的execute，其代码如图1.9所示，就是调用一下他的绑定函数func，这通常在自定义节点的draw函数中绑定，这也是我绘制3D场景所用的方法。

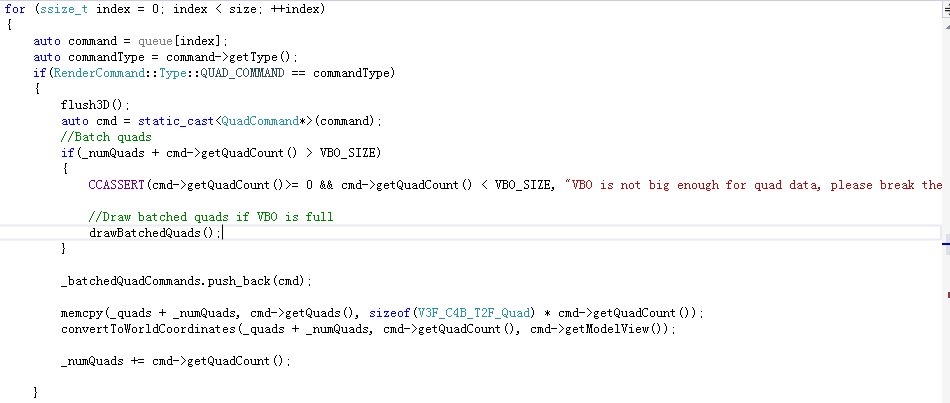


图1.7 visiteRendereQueue的QUAD\_COMMAND

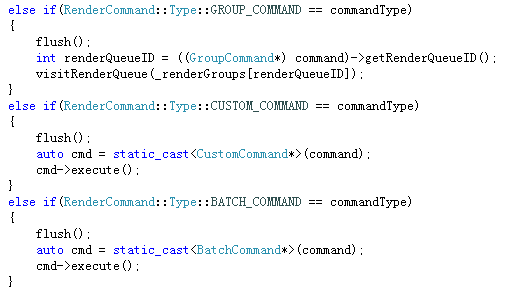


图1.8 visiteRendereQueue的其他命令类型



图1.9 自定义命令的

## 1.4 AutoBatch原理讲解

而后，我们再看看他的drawBatchQuard是怎么做的。如图1.10，1.11所示，这个函数首先判断是否有顶点，若没有则return掉。而后向显存传入数据，绑定VAO。之后的那一段(图1.11)就包含传说中的的autoBatch，drawCall合并(C:\Users\ADMINI~1\AppData\Local\Temp\ZX0P[3`D]FR55~CMA987U%B.jpg)。遍历数组，当当前命令的材质与上一个相同，则batch，否则绘制之前BUFFER，把这个quard给batch进去。

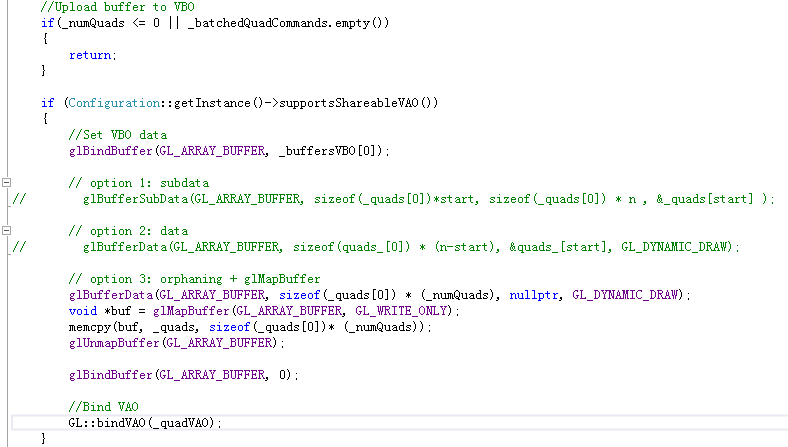


图1.10 drawBatchedQuads(1)

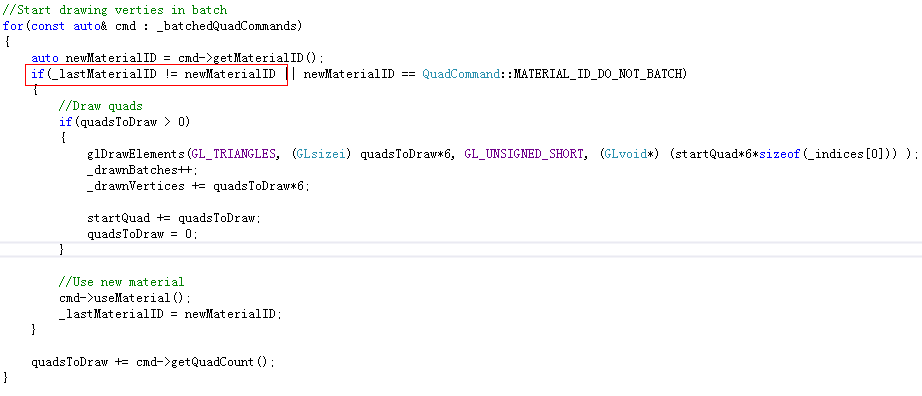


图1.11 drawBatchedQuads(2)

也就是说，如果我们相邻添加的Sprite的纹理不同，引擎是不会做batch的，正如图1.5所示，虽然我们只用了6张纹理，但drawCall仍然是10001。下面我们来做个试验，来说明这个问题。

首先，把之前的代码修改为如图1.12所示的样子，只让程序绘制一种水果。我们可以看到，引擎经行了Autobatch，绘制所有的水果的drawCall只有1次(另一次是glClear())。

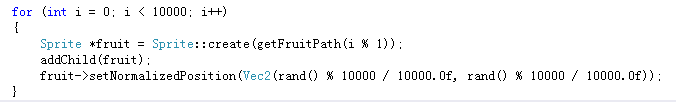


图1.12 代码1



图1.13 autobatch效果

再次修改代码，如图1.14所示，交替添加两种水果的sprite。drawCall又变回10001。再次修改代码，如图1.16所示。

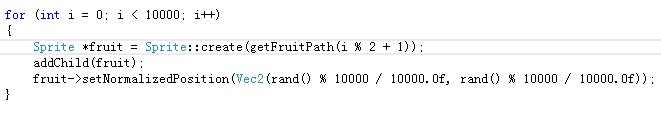


图1.14 代码2



图1.15 autobatch效果2

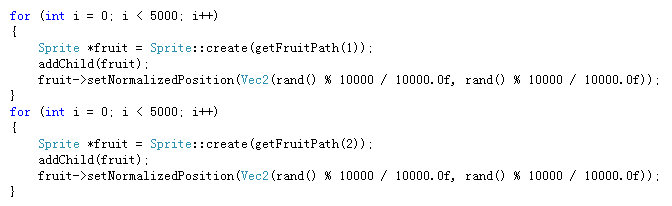


图1.16代码3



图1.17 autobatch效果3

修改代码后，芒果遮住了所有的葡萄（葡萄还发生了绘制），此时drawCall变成了3。这也正应正了前面的猜想。那么，我没修改Z-order是否会影响drawCall呢？答案是肯定的。修改代码如图1.18所示。我们可以看到，修改了Zorder，并没有使drawCall暴增，因为在绘制前，引擎会根据zOrder进行排序，所以此时drawCall为5。再对比一下帧数，我们可以发现，经过drawCall合并的渲染，效率提升了至少20个百分点。

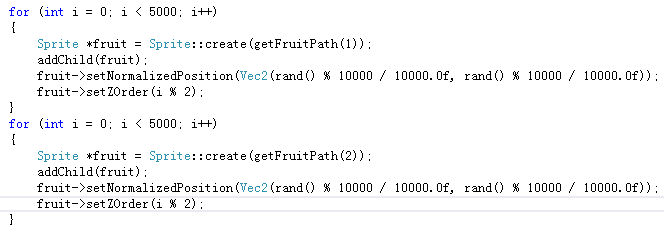


图1.18代码4



图1.19 autobatch效果4

## 1.5 自定义RenderCommand

下面，我们开始进入正题，探究一下如何在2dx下写openGL代码。首先，我们依照惯例写个类，CustomRenderCommandTest：

class CustomRenderCommandTest : public cocos2d::Node

{

public:

CustomRenderCommandTest();

~CustomRenderCommandTest();

static CustomRenderCommandTest\* create();

void draw(cocos2d::Renderer \*renderer, const cocos2d::Mat4& transform, uint32\_t flags)override;

void onDraw(const cocos2d::Mat4& transform, uint32\_t flags);

protected:

virtual bool init()override;

cocos2d::CustomCommand m\_renderCommand;

};

void CustomRenderCommandTest::draw(cocos2d::Renderer \*renderer, const cocos2d::Mat4& transform, uint32\_t flags)

{

m\_renderCommand.init(\_globalZOrder);

m\_renderCommand.func = CC\_CALLBACK\_0(CustomRenderCommandTest::onDraw, this, transform, \_globalZOrder);

renderer->addCommand(&m\_renderCommand);

}

void CustomRenderCommandTest::onDraw(const cocos2d::Mat4& transform, uint32\_t flags)

{

glClearColor(0.1f, 0.1f, 0.5f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

}

其实逻辑很简单，定义一个customRenderCommand，在draw初始化他，使用CC\_CALLBACK\_0绑定一下onDraw函数(CC\_CALLBACK\_1,2,3,4指的是固定值参数，其宏定义的内容就是std::bind)。在onDraw里，我们简单的调用glClear函数，使屏幕变为蓝色。其效果如图1.20所示。

# OpenGL简介

在上一章，我们已经了解了cocos2dx引擎的渲染流程，了解了如何在2dx下编写OpenGL代码。但问题又来了，到底该如何编写OpenGL代码么？且听在下分解。

## 2.1 OpenGL 渲染管线

OpenGL implements what’s commonly called a rendering pipeline, which is a sequence of processing stages for converting the data your application provides to OpenGL into a final rendered image.

咳咳，每天装个B，生活好甜蜜。上面那一段是OpenGL红宝书第八版上对OpenGL管线的描述。说管线就是把从程序输入的数据变为最终图像的一系列不同的程序阶段，也就是我们常说的渲染流水线。整个管线的流程，如图2.1所示。

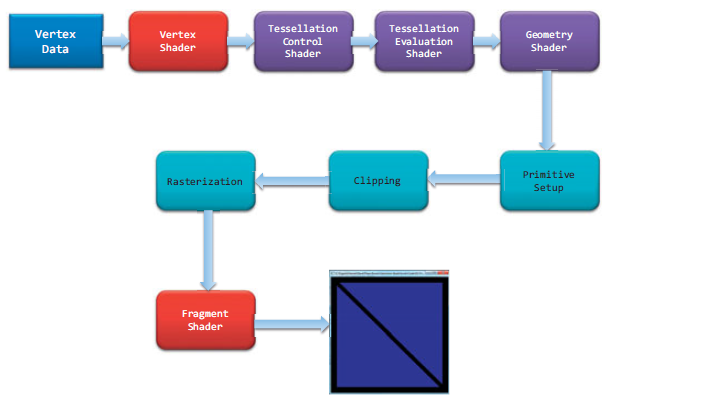


图2.1 openGL 渲染管线

### 2.1.1 Vertex Shading

当我们向显卡发送渲染命令时，显卡会把显存中的顶点传入这个阶段。针对每一个顶点，程序员可以编写代码控制他的各项属性，如位置，颜色，纹理坐标，以及其他自定义属性。在可编程管线时代，把3D空间的顶点坐标转换为2D的平面的顶点坐标，也是在这一阶段实现的。在这一阶段之后，GPU会把各个顶点传入Tessellation Shading 和 Geometry Shading。这两个阶段我们现在不用关心，在以后的深入学习中，再去探讨。

### 2.1.2 Primitive Assembly

经过前面的一系列操作，显卡就会将前面的一系列顶点组装成图元，在ES系统下，通常是组装成三角形。传入下面一个阶段。

### 2.1.3 Clipping

组装好的图元将会被送进这个阶段。在这个阶段中，完全超出视图区域的将被完全丢弃，一部分超出剪裁区域的将被裁切成新的图元。

### 2.1.4 Rasterization

经过剪裁的图元将被送入这个阶段，进行光栅化操作。也就是把三角形进行填充，把顶点信息进行插值，这也是为什么给顶点设置不同的颜色，可以画出彩色方块的原因。如图2.2所示。

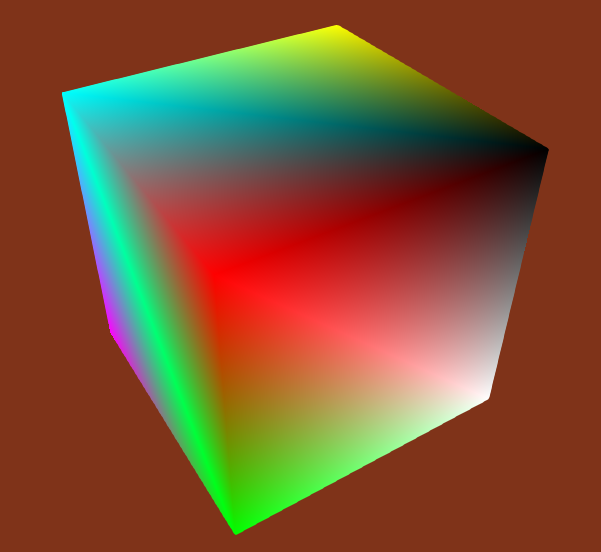


图2.2 彩色立方体

### 2.1.5 fragment shading

这一阶段，就是我们所熟知的片源着色器阶段。在这一阶段，我们使用自定义的程序，对每一个片源进行处理。在可编程管线时代，纹理的采样，光照的处理，都是在这一阶段执行。

### 2.1.6 Pre-Fragment Operations

在执行完上一阶段后，管线还会做一些额外的操作，如图2.3所示。

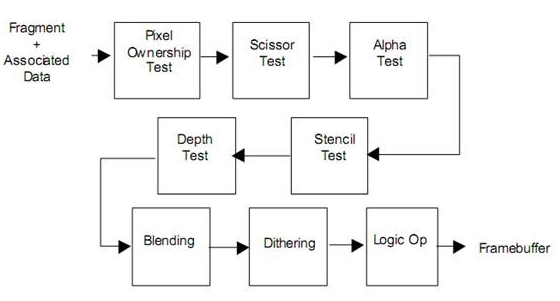


图2.3 Pre-Fragment Operations

在这里，我们常用用到的是这几个，剪裁测试，Alpha测试，深度测试，蒙版测试，颜色混合。

剪裁测试，是通过调用glScissorXXX相关函数，设定一个矩形区域，当片源不在矩形区域中，管线将自动阻止他进入下一阶段。

Alpha测试，我们可以通过调用glAlphaFunc，设定只有alpha满足一定条件后，才能通过这个测试。

深度测试，当将要绘制的片源Z值大于原先帧缓冲片源的Z值时，才会通过这一阶段。

蒙版测试，我们可以调用glStencil 相关函数，设置蒙版区域，只有蒙版区域被标记为某些值得时候，才能通过测试。

通常，深度缓冲和蒙版缓冲是放在一起的，深度缓冲占24位，蒙版缓冲占8位。

另外纠正一点，我们平常爱说的，给某一个层添加一个蒙版，添加了一个半透明的层，蒙到场景上面。准确的说，这种方法叫Blending，而不是蒙版。是管线根据alpha值，与之前的帧缓冲经行叠加。

## 2.2 一个简单的着色器程序

了解了OpenGL管线，小伙伴们是不是已经按捺不住，想要做点什么呢。下面，我们将写第一个应用shader程序，如图2.4,2.5,2.6,2.7所示。

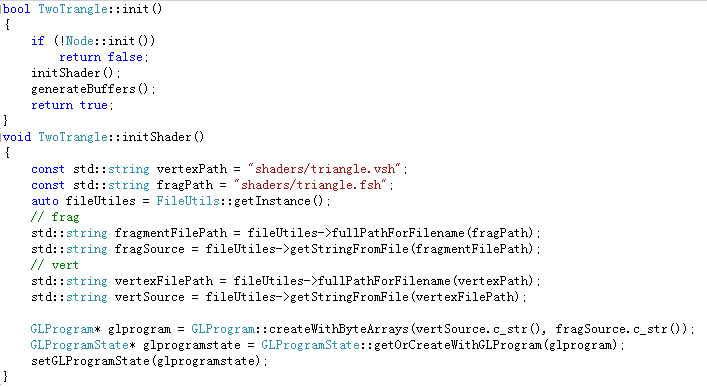


图2.4 第一个三角形代码-1

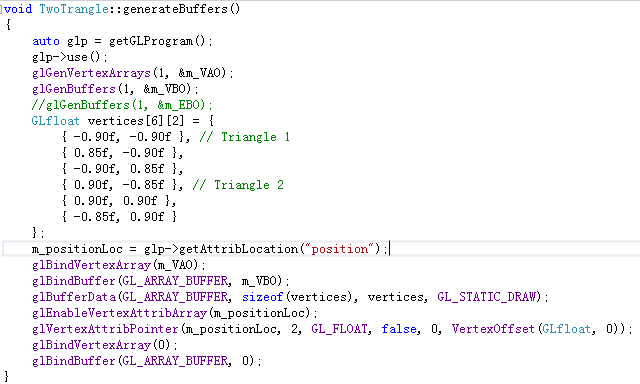


图2.5 第一个三角形代码-2

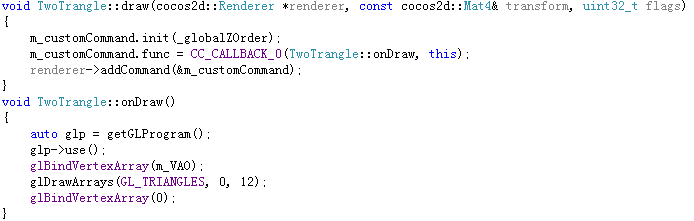


图2.6 第一个三角形代码-3

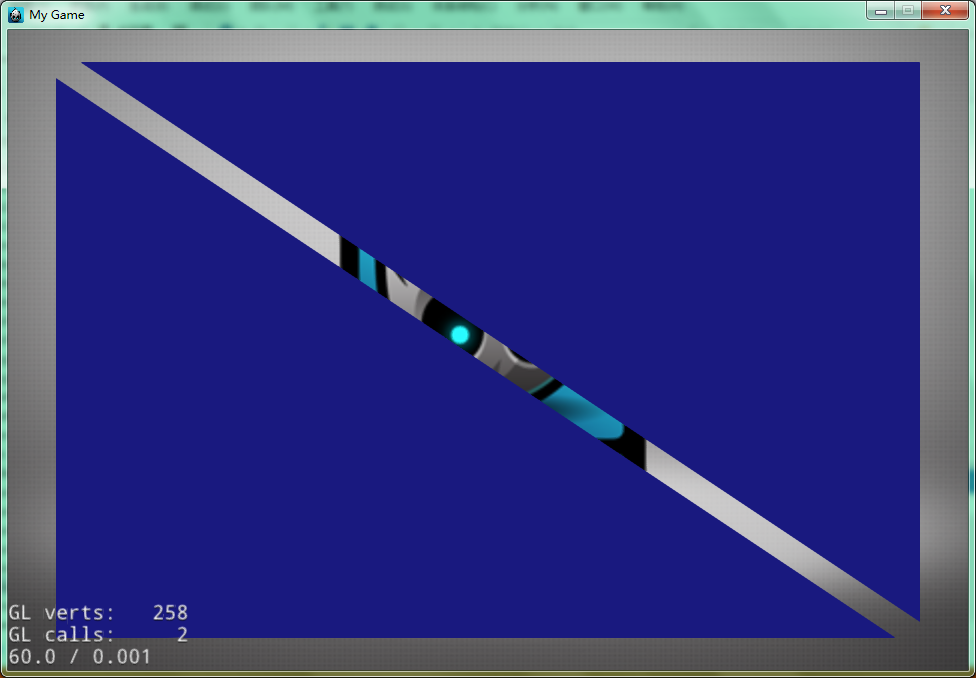


图2.7 第一个三角形

Vertex Shader:

attribute vec2 position;

void main()

{

gl\_Position = vec4(position,0.0,1.0);

}

Fragment Shader:

void main()

{

gl\_FragColor = vec4(0.1,0.1,0.5,1.0);

}

下面，我将来逐一讲解这些程序。

首先是初始化shader，这些都是固定的代码，以后可以封装到基类中，以供统一调用。在这里，我不得不吐槽一下新版本的GLProgramState。这个类是对于GLProgram的一个封装，提供了对uniform和attribute的统一管理。但当我查看他的源码的时候发现，他的实现方法用了共用体，用了std::function的callback，令人很难驾驭，而且在apply函数中，没有检测是否需要更新。总之，还没有小编以前写的UniformManager好用。所以建议大家不要用他。

## 2.3 VAO与VBO

下面就是创建顶点，并把顶点传入显存中。

有些同学看到这里就要问了，为什么要把数据传入显存中，内存中一份数据，显存中一份数据，这不是浪费存储么？但问题在于，CPU和显卡的GPU并不是放一起的，在硬件中是两个独立的原件。CPU更善于随机访问的逻辑运算，GPU善于做批量处理的数学运算。我们的手机、电脑之所以要装显卡，是为了把CPU从繁重的图形学运算中解脱出来，专职处理逻辑运算。而让更善于做并行批量数学运算的显卡去做图形学计算。这样一来，CPU和GPU就成了一个CS架构。在硬件高度发展的今天，图形学程序的效率损失，往往会发生在CPU和GPU之间的数据传输以及同步上，如图2.8所示。

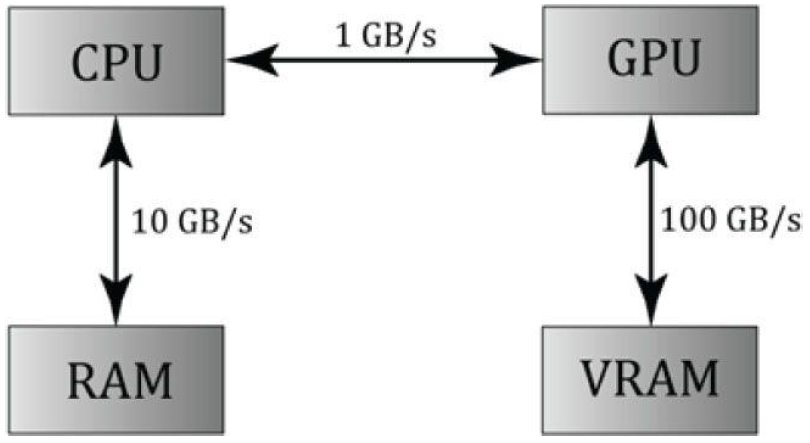


图2.8 GPU和CPU的数据传递

正因如此，OpenGL提供了一个叫Vertex Buffer Object的解决方案，也就是我们常说的VBO。我们使用glGenBuffer申请BUFFER的标识（有GL管理），使用glBindBuffer指定当前操作哪一个VBO，使用glBufferData向显存传输数据。关于这些函数的用法，我们可以在<http://www.khronos.org/opengles/sdk/docs/man/> 查到。

显卡有了数据，但还不知到该如何解释这些数据，所以我们要告诉他。这样Vertex Array Object就派上了用场。我们使用GLprogram对象的getAttribLocation方法，获取顶点数组在shader中的位置，使用glEnableVertexAttribute来激活这个位置，使用glAttributePointer来制定对数据的解析。

glAttributePointer的几个参数比较有讲究，我单独拿出来讲讲。第一个参数index是指用getAttribLocation获取的那个位置，第二个参数size,是一个顶点由几维，只能写1,2,3,4，如我们程序传的都是二维顶点，写2就好了。第三个参数type是数据类型，我们通常用GLFloat。第四个参数normalize指是否归一化，我们通常写false。第5个参数stride，指的是一个顶点数据的字节数，如果我们定点数据只有一种，如位置，这里写0就可以。第6个参数，offest，指的是第一个该数据在传入VBO中的偏移量。我们的这个程序里只传入了顶点位置，写0就好了。

这些工作做完后，我们不要忘了把VBO和VAO绑定回0，以免影响其他图元的绘制。

最后，在ondraw函数中，绑定前面的VAO，调用drawArray命令，完成三角形的绘制。

在这个程序里，我们的shader写的很简单，就是把传过来的顶点负值，把颜色设为蓝色。

## 2.4 彩色的三角形

有的小伙伴看到这里，不禁要吐槽了。”擦，老子裤子都脱了，你就给我看这个”。别急，下面我们让三角形带上点颜色。我们对之前的代码做如下修改，如图2.9,2.10,2.11所示，运行效果如2.12所示。

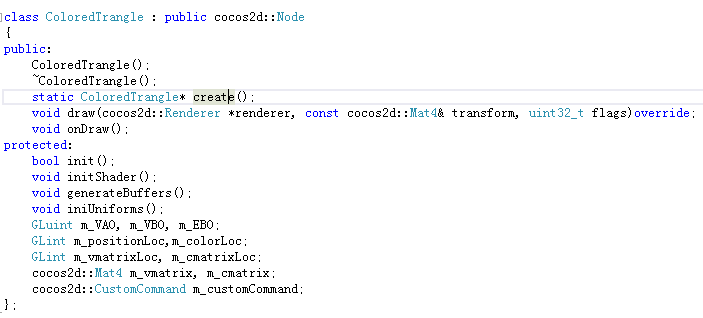


图2.9 彩色三角形代码修改-1

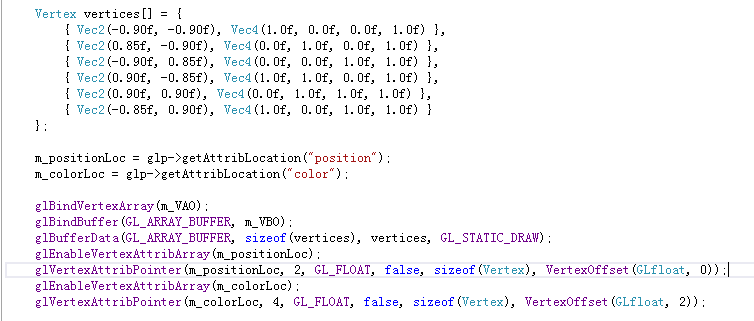


图2.10 彩色三角形代码修改-2

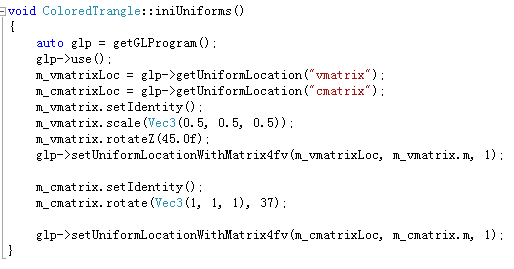


图2.11 彩色三角形代码修改-3

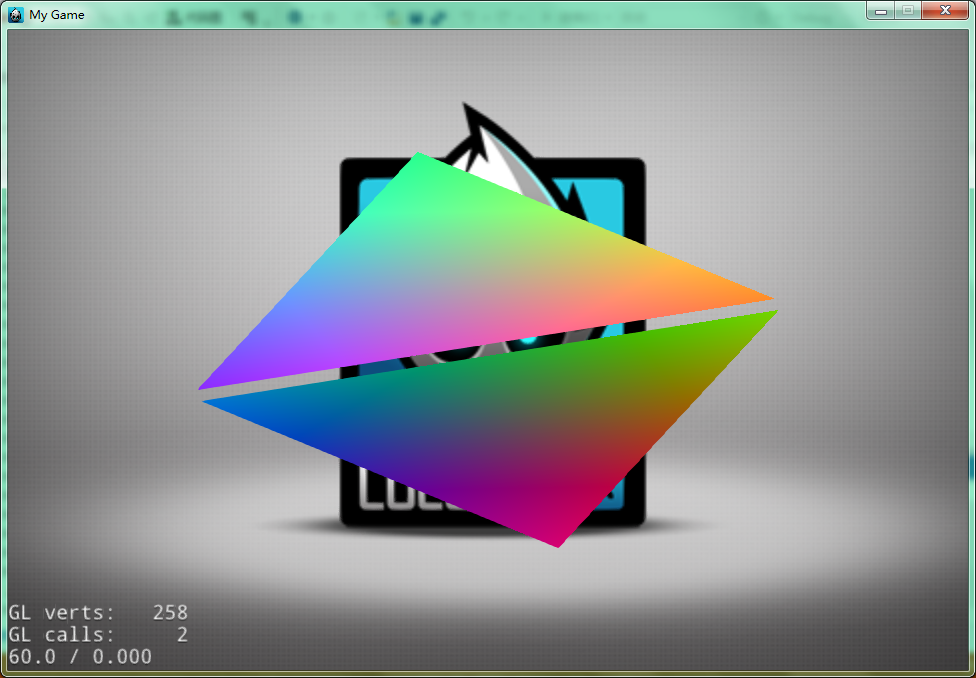


图2.12 彩色三角形运行效果

Vertex Shader:

attribute vec2 position;

attribute vec4 color;

varying vec4 v\_color;

uniform mat4 vmatrix;

void main()

{

gl\_Position = vmatrix \* vec4(position,0.0,1.0);

v\_color = color;

}

Fragment Shader:

varying vec4 v\_color;

uniform mat4 cmatrix;

void main()

{

gl\_FragColor = cmatrix \* v\_color;

}

在这个程序中，我们对每个顶点添加了颜色，为顶点的变幻和颜色的变幻添加了一个控制矩阵。在shader中，我们让矩阵乘以这些值。

## 2.5 shader语法简介

Shader的整体语法与C++类似，但也有些许不同，现在我们来逐一分析。

### 2.5.1 变量的声明

Shader中的变量与C语言类似，有float，double, int, uint, bool。这些类型的详细解释如图2.13所示。但要注意的是，不能用unsigned做修饰，bool型变量目前不能用作unifrom，因为引擎没有提供传值得基础。

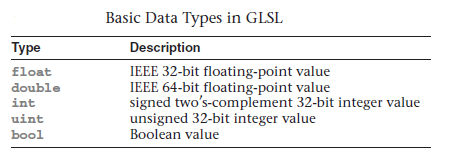


图2.13 shader中的基本类型

除了上面的基础类型，在shader中还有一些和向量矩阵有关的拓展类型，如图2.14所示。

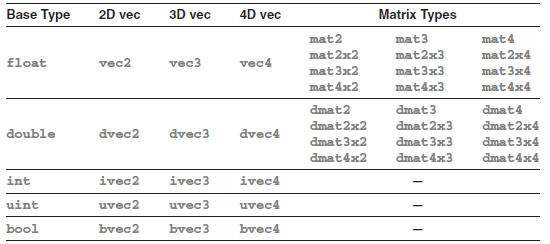


图2.14 shader中的拓展类型

脑筋灵活的同学，又该发出疑问，在shader中数组和指针该如何定义呢。额，告诉大家一个不幸的好消息，shader是脚本语言，指针这种杀伤脑细胞的东西是木有滴，对于数组的声明，我们可以像既可以c语言那样在变量后面加中括号(float a[3];)，也可以像C#那样在类型名后加中括号(float[] a;)

### 2.5.2 变量的转换

在shader的语法中，对类型的隐式转换比C更加严格，比如float a= false这样的代码是会报错的。只有一下类型的隐式转换是被允许的，如图2.15所示。

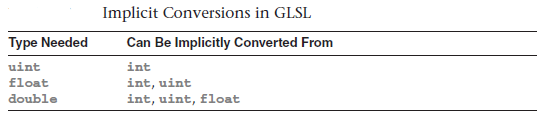


图2.15 shader中的隐式类型转换

大多数情况下，我们使用显示类型转换，类型名()的形式，如float(a);vec4(a\_vec2.xy,0.0,1.0);

### 2.5.3 Vector Component Accessors

vec4(a\_vec2.xy,0.0,1.0)？xy是什么？这是为了方便书写，GLSL对vec中的变量提供了访问别名，如下图所示。

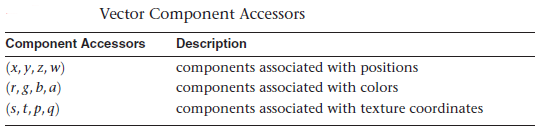


图2.16 vector的别名

### 2.5.4 shader的结构

我们知道，我们常用的shader有两个，一个是顶点着色器，一个是片元着色器。在顶点着色器中，传入的变量用attribute修饰，通常放在程序顶部。需要在顶点着色器与片元着色器间传递的变量，用varying修饰，通常这种变量在顶点着色器中赋值，在片元着色器中读取（读取到的内容是经过插值的）。Uniform型变量同时可以在顶点着色器和片元着色器中被访问，但不能被修改。

每一个shader程序，虽然有着main函数，但实质上却是一个有着输入输出的函数。正如咱前面ColoredTriangle的代码，顶点着色器中position,color是输入，gl\_Position，v\_color是输出；片元着色器中，v\_color是输入，gl\_fragColor是输出，通常片元着色器中有且仅有这一个输出。

### 2.5.5shader中的内置变量与内置函数

细心的同学可能要问，gl\_Position, gl\_fragColor是什么，好像程序中完全没有定义过啊。其实，这些是GLSL的内置变量，如图2.17,1.18所示。

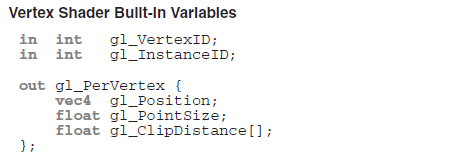


图2.17顶点着色器的内置变量

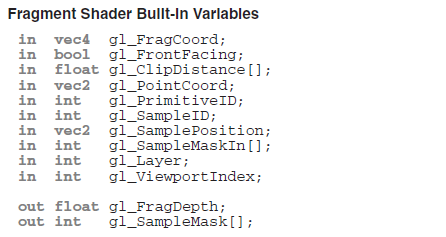


图2.18片元着色器的内置变量

这些内置变量的详细解释，请在红宝书第8版的附录C中查询。除了内置变量，还有一些内置函数，内置常数，在这也就不一一列举，有兴趣的同学可以翻阅红宝书进行深入学习。

### 2.5.6 cocos2dx提供的内置变量

除了上面提到的GLSL内置变量，我们还可以在官方demo中看到一些CC打头的变量，那是Cocos2dx在编译shader时加进去的。

打开引擎cocos/render下的CCGLProgram.h文件，我们可以看到这些静态变量，如图2.19所示。

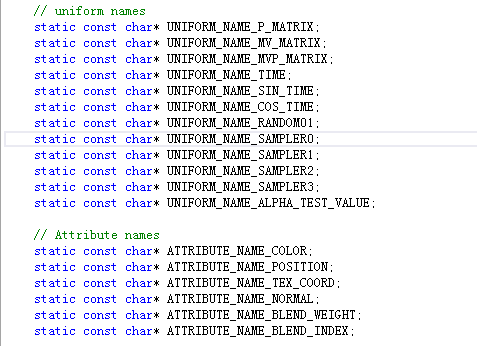


图2.19 cocos2dx的shader内置变量1

转到CPP文件，我们可以看到如下定义。如图2.20所示。在updateUniforms函数中，把这些变量定义到shader中，新版本的createWithByteArrays函数中会被自动调用。在每帧调用setUniformsForBuiltins函数可以更新诸如MV矩阵时间控制等变量。

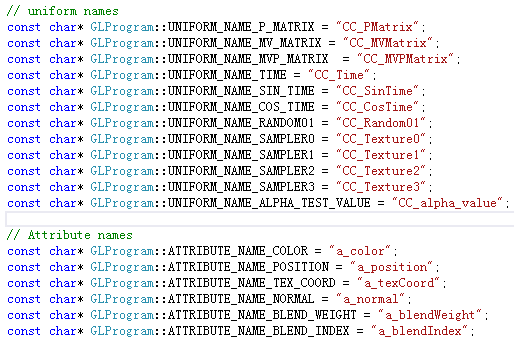


图2.20 cocos2dx的shader内置变量2

# 三 3D图元的绘制

通过前面的学习，大家是否已经对cocos2dx下的openGL编程有了初步的了解呢。下面，我们来进入正题，绘制3D图元。

## 3.1 MVP 3D 矩阵变幻

在学习3D的时候，我们常常会听到MVP的概念。下面我分析下这些矩阵的意义。

我们知道，电脑的屏幕都是二维的，显卡的光栅化，也都是处理的二维图元。那么，我们如何把3D的坐标，变化为屏幕的二维坐标呢。

### 3.1.1 投影矩阵