**Cocos2d-x  2.0 -- 从 点，线，面学起**

**本节所用Cocos2d-x版本：cocos2d-2.0-x-2.0.2**

         一个图形引擎，总是由构建点，线，面的绘制功能写起来的。点，线，面。构成了最初的图形基础。所以说，掌握点，线，面是掌握引擎的基础。

         在Cocos2d-x 1.0版本中，提供了使用OpenGL API来创建点，线，面的例子DrawPrimitivesTest。而在2.0中，同样的例子名称，而内部实现却差别巨大。我们知道，在Cocos2d-x 2.0版本，相较于1.0，增加了shader的支持，而DrawPrimitivesTest这个例子，就是学习基础Shader的最好教程。

**学前提示：**

        OpenGL着色语言（GLSL――OpenGL Shading Language）是用来在OpenGL中着色编程的语言，也即开发人员写的短小的自定义程序，他们是在[图形卡](http://baike.baidu.com/view/2449516.htm)的GPU （Graphic Processor Unit图形处理单元）上执行的，代替了固定的[渲染管线](http://baike.baidu.com/view/127356.htm)的一部分。比如：视图转换、投影转换等。GLSL（GL Shading Language）的着色器代码分成2个部分：Vertex Shader（[顶点着色](http://baike.baidu.com/view/415314.htm)器）和Fragment（片断着色器），有时还会有Geometry Shader（几何着色器）。负责运行顶点着色的是顶点着色器。它可以得到当前OpenGL 中的状态，GLSL内置[变量](http://baike.baidu.com/view/296689.htm)进行传递。

       打开TestCpp工程，找到Classes下的DrawPrimitivesTest目录。打开两个文件：

DrawPrimitivesTest.h/cpp

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. #ifndef \_DRAW\_PRIMITIVES\_TEST\_H\_
2. #define \_DRAW\_PRIMITIVES\_TEST\_H\_
3. //包含Cocos2d头文件
4. ////----#include "cocos2d.h"
5. //使用TestScene这个CCScene类
6. #include "../testBasic.h"
7. //定义派生于CCLayer的类DrawPrimitivesTest,重载draw用于进行手动渲染处理
8. **class** DrawPrimitivesTest : **public** CCLayer
9. {
10. **public**:
11. //构造
12. DrawPrimitivesTest();
13. //析构
14. **virtual** **void** draw();
15. };
16. //定义派生于TestScene的类DrawPrimitiveTestScene,做为TestCpp工程中TestController类集中管理的各个小功能例子的场景
17. **class** DrawPrimitivesTestScene : **public** TestScene
18. {
19. **public**:
20. //重载启动此功能例子的场景函数
21. **virtual** **void** runThisTest();
22. };
24. #endif

OK，头文件看完了，现在看CPP文件：

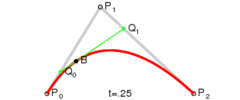
**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. #include "DrawPrimitivesTest.h"
2. //构造函数
3. DrawPrimitivesTest::DrawPrimitivesTest()
4. {
5. }
6. //手动处理的渲染函数
7. **void** DrawPrimitivesTest::draw()
8. {
9. //取得屏幕大小
10. CCSize s = CCDirector::sharedDirector()->getWinSize();
11. //检测是否有OpenGL错误发生，如果有则打印错误
12. CHECK\_GL\_ERROR\_DEBUG();
14. //平滑模式，即高洛德着色
15. //    glEnable(GL\_LINE\_SMOOTH);
16. //绘制一条件,参1为起点，参2为终点，ccp为生成CCPoint的宏
17. ccDrawLine( ccp(0, 0), ccp(s.width, s.height) );
19. //检测是否有OpenGL错误发生，如果有则打印错误
20. CHECK\_GL\_ERROR\_DEBUG();
22. //设置线宽
23. glLineWidth( 5.0f );
24. //设置后面要进行绘制时所用的色彩
25. ccDrawColor4B(255,0,0,255);
26. //绘制线条
27. ccDrawLine( ccp(0, s.height), ccp(s.width, 0) );
29. //检测是否有OpenGL错误发生，如果有则打印错误
30. CHECK\_GL\_ERROR\_DEBUG();
32. //设置点的大小
33. ccPointSize(64);
34. //设置后面要进行绘制时所用的色彩
35. ccDrawColor4B(0,0,255,128);
36. //绘制一个点
37. ccDrawPoint( ccp(s.width / 2, s.height / 2) );
38. //检测是否有OpenGL错误发生，如果有则打印错误
39. CHECK\_GL\_ERROR\_DEBUG();
41. // 绘制四个点
42. //这里创建位置点数组
43. CCPoint points[] = { ccp(60,60), ccp(70,70), ccp(60,70), ccp(70,60) };
44. ccPointSize(4);
45. //设置后面要进行绘制时所用的色彩
46. ccDrawColor4B(0,255,255,255);
47. //使用位置点数组做为四个顶点的位置进行绘制
48. ccDrawPoints( points, 4);
49. //检测是否有OpenGL错误发生，如果有则打印错误
50. CHECK\_GL\_ERROR\_DEBUG();
52. //绘制一个绿色圆
53. glLineWidth(16);
54. //设置后面要进行绘制时所用的色彩
55. ccDrawColor4B(0, 255, 0, 255);
56. //绘制圆函数，参1是中心点，参2为半径,参3为圆的逆时针旋转角度，参4为圆的平均切分段数，最后一个参数是指定是否从圆分段起止点位置向圆中心连线，这里不进行连线
57. ccDrawCircle( ccp(s.width/2,  s.height/2), 100, 0, 10, **false**);
58. //检测是否有OpenGL错误发生，如果有则打印错误
59. CHECK\_GL\_ERROR\_DEBUG();
61. //绘制一个蓝色圆,进行连线
62. glLineWidth(2);
63. //设置后面要进行绘制时所用的色彩
64. ccDrawColor4B(0, 255, 255, 255);
65. //这里使用了一个宏CC\_DEGREES\_TO\_RADIANS把角度值转为弧度。转动了90度，目的是为了让中心连线垂直显示。
66. ccDrawCircle( ccp(s.width/2, s.height/2), 50, CC\_DEGREES\_TO\_RADIANS(90), 50, **true**);
67. //继续检错
68. CHECK\_GL\_ERROR\_DEBUG();
70. // 绘制多边形线框。
71. ccDrawColor4B(255, 255, 0, 255);
72. glLineWidth(10);
73. CCPoint vertices[] = { ccp(0,0), ccp(50,50), ccp(100,50), ccp(100,100), ccp(50,100) };
74. //这里绘制多边形线框函数，使用上面的顶点数组做为多边形线框的顶点位置，第二个参数为顶点数量，第三个参数指定是否首尾自动连接形成封闭线框。
75. //注：其实这个函数拆成两个函数比较好,一个是去掉最后一个参数的ccDrawPoly，用于绘制默认封闭的多边形线框。另一个ccDrawLineList用于绘制线段列。
76. ccDrawPoly( vertices, 5, **false**);
77. //继续检错
78. CHECK\_GL\_ERROR\_DEBUG();
80. //绘制实体多边形
81. glLineWidth(1);
82. CCPoint filledVertices[] = { ccp(0,120), ccp(50,120), ccp(50,170), ccp(25,200), ccp(0,170) };
83. //这里绘制内部填充指定色彩的多边形
84. ccDrawSolidPoly(filledVertices, 5, ccc4f(0.5f, 0.5f, 1, 1 ) );
86. // 绘制封闭多边形线框，这里就是个三角形线框了。
87. ccDrawColor4B(255, 0, 255, 255);
88. glLineWidth(2);
89. CCPoint vertices2[] = { ccp(30,130), ccp(30,230), ccp(50,200) };
90. ccDrawPoly( vertices2, 3, **true**);
92. CHECK\_GL\_ERROR\_DEBUG();

        后面绘制贝塞尔曲线，要了解贝塞尔曲线，请在维基百科里学习一下

<http://zh.wikipedia.org/zh-cn/>貝茲曲線#.E4.BA.8C.E6.AC.A1.E6.96.B9.E8.B2.9D.E8.8C.B2.E6.9B.B2.E7.B7.9A

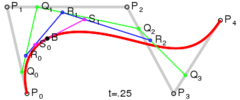
        下面是绘制二次贝塞尔曲线，类似下图，图是从维基百科上找来的，恕我没那个能力画图了暂拿来用讲原理



**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. //这个就是cocos2d-x2.0绘制二次贝塞尔曲线函数，三个参数分别如图中P0,P1,P2，不过在咱们这个例子中，正好与之上下镜像。最后一个是曲线构成所用的线段数,当然，线段数越多曲线越平滑。
2. ccDrawQuadBezier(ccp(0,s.height), ccp(s.width/2,s.height/2), ccp(s.width,s.height), 50);
3. //检错
4. CHECK\_GL\_ERROR\_DEBUG();

        然后是绘制高阶贝塞尔曲线，类似下图



**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. //前四个参数应该对应的是P0,P1,P3,P4,图上的P2可以省去。最后一个是曲线构成所用的线段数。
2. ccDrawCubicBezier(ccp(s.width/2, s.height/2), ccp(s.width/2+30,s.height/2+50), ccp(s.width/2+60,s.height/2-50),ccp(s.width, s.height/2),100);
4. CHECK\_GL\_ERROR\_DEBUG();

再继续

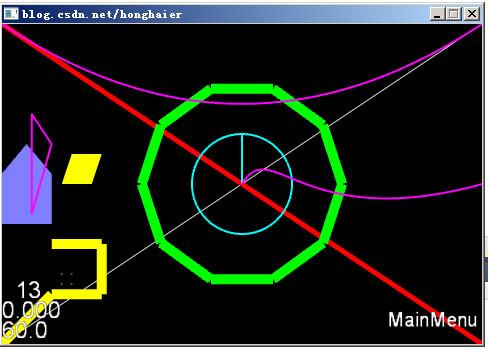
**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. //绘制黄色实心四边形色块。
2. CCPoint vertices3[] = {ccp(60,160), ccp(70,190), ccp(100,190), ccp(90,160)};
3. ccDrawSolidPoly( vertices3, 4, ccc4f(1,1,0,1) );
5. // 重置绘制状态
6. glLineWidth(1);
7. ccDrawColor4B(255,255,255,255);
8. ccPointSize(1);
10. CHECK\_GL\_ERROR\_DEBUG();
11. }

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. //启动场景
2. **void** DrawPrimitivesTestScene::runThisTest()
3. {
4. //用new 创建一个DrawPrimitivesTest实例对象做为场景中要显示的层。
5. CCLayer\* pLayer = **new** DrawPrimitivesTest();
6. addChild(pLayer);
7. pLayer->release();
8. //替换当前正在运行的基它实例场景
9. CCDirector::sharedDirector()->replaceScene(**this**);
10. }

好吧，运行一下，看图：



      一个一个来对照代码看一看，我们发现，其在调用ccPointSize进行点的大小设置时根本就不管用。 有点无语，可见这一版扣扣二弟放出来还是有点仓促~

      大家先看罢，吾上WC下，一会儿见。

      约过了五分钟………..

      现在大家看懂照截图看懂代码了吧。那我们更深入一步吧。

      既然画点有点问题咱就先看画点。在ccDrawPoint函数调用处加断点。F11进入CCDrawingPrimitives.cpp:

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. **void** ccDrawPoint( **const** CCPoint& point )
2. {
3. //Shader初始化函数，一会儿再解。
4. lazy\_init();
6. //定义顶点变量，填充位置
7. ccVertex2F p;
8. p.x = point.x;
9. p.y = point.y;
10. //设置OpenGL在后面的渲染中用到顶点位置属性。
11. ccGLEnableVertexAttribs( kCCVertexAttribFlag\_Position );
12. //这里对下面的渲染使用了Shader
13. s\_pShader->use();
14. //设置Shader的输入参数：最终矩阵结果值(世界x观察x投影矩阵)。
15. s\_pShader->setUniformForModelViewProjectionMatrix();
16. //设置Shader的输入参数:色彩。取s\_tColor.r地址做为float4参数值的地址传入。实际Shader使用的是s\_tColor的所有四个float值。
17. s\_pShader->setUniformLocationWith4fv(s\_nColorLocation, (GLfloat\*) &s\_tColor.r, 1);
18. //设置Shader的输入参数:点大小
19. s\_pShader->setUniformLocationWith1f(s\_nPointSizeLocation, s\_fPointSize);
20. //将p设置为使用的顶点位置参数
21. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, &p);
22. //绘制顶点数组，参1为要绘制的图形为点，参2为顶点起始索引，参3为顶点数量
23. glDrawArrays(GL\_POINTS, 0, 1);
24. //渲染批次（DrawCall）统计
25. CC\_INCREMENT\_GL\_DRAWS(1);
26. }

       回头看lazy\_init,作者命名lazy\_init的原因难道是说：我懒，初始化的相关处理就不每次写了，放函数里用多省心~。偷笑

      要看函数，先看文件最上部作者定义的静态变量。

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. **static** **bool** s\_bInitialized = **false**; //用于标记是否初始化
2. **static** CCGLProgram\* s\_pShader = NULL;  //Shader代码程式
3. **static** **int** s\_nColorLocation = -1;       //Shader输入色彩的变量位置索引
4. **static** ccColor4F s\_tColor = {1.0f,1.0f,1.0f,1.0f};  //色彩值
5. **static** **int** s\_nPointSizeLocation = -1;                   //Shader 输入的大小的变量位置索引
6. **static** GLfloat s\_fPointSize = 1.0f;                 //大小值

然后分析这个函数：

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. **static** **void** lazy\_init( **void** )
2. {
3. //如果尚未初始化，则进行初始化，已经初始化则略过
4. **if**( ! s\_bInitialized ) {
6. //通过字符串参数找到所对应Shader文件的Shader代码程式
7. s\_pShader = CCShaderCache::sharedShaderCache()->programForKey(kCCShader\_Position\_uColor);
8. //取得Shader对应变量"u\_color"在程式片段中的位置索引，返回给s\_nColorLocation。
9. s\_nColorLocation = glGetUniformLocation( s\_pShader->getProgram(), "u\_color");
10. //检错
11. CHECK\_GL\_ERROR\_DEBUG();
12. //取得Shader对应变量"u\_pointSize"在程式片段中的位置索引,返回给s\_nPointSizeLocation。
14. s\_nPointSizeLocation = glGetUniformLocation( s\_pShader->getProgram(), "u\_pointSize");
15. //检错
16. CHECK\_GL\_ERROR\_DEBUG();
17. //设置初始化完成
18. s\_bInitialized = **true**;
19. }
20. }

         lazy\_init函数指明了当前这些绘制点，线，面所用的Shader为字符串变量kCCShader\_Position\_uColor所对应的Shader。在kCCShader\_Position\_uColor上按F12进入CCGLProgram.h，这里定义了不少字符串变量:

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. #define kCCShader\_PositionTextureColor            "ShaderPositionTextureColor"
2. #define kCCShader\_PositionTextureColorAlphaTest    "ShaderPositionTextureColorAlphaTest"
3. #define kCCShader\_PositionColor                    "ShaderPositionColor"
4. #define kCCShader\_PositionTexture                "ShaderPositionTexture"
5. #define kCCShader\_PositionTexture\_uColor        "ShaderPositionTexture\_uColor"
6. #define kCCShader\_PositionTextureA8Color        "ShaderPositionTextureA8Color"
7. #define kCCShader\_Position\_uColor                "ShaderPosition\_uColor"

        Shader程序在哪呢？我们追查一下CCShaderCache::sharedShaderCache()->programForKey函数。进入到CCShaderCache.cpp文件，这里有一个CCShaderCache类，顾名思义，Shader缓冲。代码不多，详细解之：

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. #ifndef \_\_CCSHADERCACHE\_H\_\_
2. #define \_\_CCSHADERCACHE\_H\_\_
4. //使用到目录相关处理类
5. #include "cocoa/CCDictionary.h"
6. //使用Cocos2d命名空间
7. NS\_CC\_BEGIN
8. //声明使用类CCGLProgram
9. **class** CCGLProgram;
10. //由CCObject派生类CCShaderCache
11. **class** CC\_DLL CCShaderCache : **public** CCObject
12. {
13. **public**:
14. //构造函数
15. CCShaderCache();
16. //析构
17. **virtual** ~CCShaderCache();
18. //返回单件类实例指针
19. **static** CCShaderCache\* sharedShaderCache();
21. //释放所管理的所有Shader代码片段
22. **static** **void** purgeSharedShaderCache();
24. //从相关文件载入默认的一些Shader代码片段
25. **void** loadDefaultShaders();
27. //从相关文件重新载入默认的一些Shader代码片段
28. **void** reloadDefaultShaders();
30. //从容器中查询指定名称的Shader代码片段
31. CCGLProgram \* programForKey(**const** **char**\* key);
33. //将一个Shader代码片段指定名称放入容器
34. **void** addProgram(CCGLProgram\* program, **const** **char**\* key);
36. **private**:
37. //初始化
38. **bool** init();
39. //按照顶点格式类型载入相应的Shader代码片段
40. **void** loadDefaultShader(CCGLProgram \*program, **int** type);
41. //目录管理类
42. CCDictionary\* m\_pPrograms;
44. };
46. NS\_CC\_END
48. #endif /\* \_\_CCSHADERCACHE\_H\_\_ \*/

Cpp文件：

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. #include "CCShaderCache.h"
2. #include "CCGLProgram.h"
3. #include "ccMacros.h"
4. #include "ccShaders.h"
6. NS\_CC\_BEGIN
7. //枚举,Shader代码片段中的一些顶点格式类型
8. **enum** {
9. kCCShaderType\_PositionTextureColor, //顶点格式为位置+纹理UV+材质色
10. kCCShaderType\_PositionTextureColorAlphaTest,    //顶点格式为顶点格式为位置+纹理UV+材质色+用于AlphaTest的ALPHA值
12. kCCShaderType\_PositionColor,// 顶点格式为位置+材质色
14. kCCShaderType\_PositionTexture, //顶点格式为位置+纹理UV
16. kCCShaderType\_PositionTexture\_uColor, //顶点格式为位置+纹理UV+材质色
18. kCCShaderType\_PositionTextureA8Color, //顶点格式为位置+纹理UV+灰度
20. kCCShaderType\_Position\_uColor, //顶点格式为位置+材质色

23. kCCShaderType\_MAX,  //枚举结束值
24. };
26. //静态Shader缓冲指针
27. **static** CCShaderCache \*\_sharedShaderCache = 0;
29. //取得Shader缓冲单件实例指针
30. CCShaderCache\* CCShaderCache::sharedShaderCache()
31. {
32. **if** (!\_sharedShaderCache) {
33. \_sharedShaderCache = **new** CCShaderCache();
34. **if** (!\_sharedShaderCache->init())
35. {
36. CC\_SAFE\_DELETE(\_sharedShaderCache);
37. }
38. }
39. **return** \_sharedShaderCache;
40. }
41. //释放
42. **void** CCShaderCache::purgeSharedShaderCache()
43. {
44. //安全释放并置空
45. CC\_SAFE\_RELEASE\_NULL(\_sharedShaderCache);
46. }
48. //构造
49. CCShaderCache::CCShaderCache()
50. : m\_pPrograms(0)
51. {
53. }
54. //析构
55. CCShaderCache::~CCShaderCache()
56. {
57. CCLOGINFO("cocos2d deallocing 0x%X", **this**);
58. m\_pPrograms->release();
59. }
60. //初始化
61. **bool** CCShaderCache::init()
62. {
63. //创建目录管理器
64. m\_pPrograms = **new** CCDictionary();
65. //从相关Shader文件载入默认的Shader代码片段
66. loadDefaultShaders();
67. **return** **true**;
68. }
69. //从相关Shader文件载入默认的Shader代码片段
70. **void** CCShaderCache::loadDefaultShaders()
71. {
72. //新建一个Shader代码片段
73. CCGLProgram \*p = **new** CCGLProgram();
74. //通过类型枚举值kCCShaderType\_PositionTextureColor加载相应的Shader文件到Shader代码片段中,这里是加载kCCShaderType\_PositionTexture\_uColor类型
75. loadDefaultShader(p, kCCShaderType\_PositionTextureColor);
76. //将Shader代码片段与代码片段字符串名称进行绑定存入容器
77. m\_pPrograms->setObject(p, kCCShader\_PositionTextureColor);
78. p->release();
80. //同上,创建Shader代码片段并加载其它类型
81. p = **new** CCGLProgram();
82. loadDefaultShader(p, kCCShaderType\_PositionTextureColorAlphaTest);
84. m\_pPrograms->setObject(p, kCCShader\_PositionTextureColorAlphaTest);
85. p->release();
87. //新建第三种Shader代码片段
88. p = **new** CCGLProgram();
89. loadDefaultShader(p, kCCShaderType\_PositionColor);
91. m\_pPrograms->setObject(p, kCCShader\_PositionColor);
92. p->release();
94. //新建第四种Shader代码片段
95. p = **new** CCGLProgram();
96. loadDefaultShader(p, kCCShaderType\_PositionTexture);
98. m\_pPrograms->setObject(p, kCCShader\_PositionTexture);
99. p->release();
101. //新建第五种Shader代码片段
102. p = **new** CCGLProgram();
103. loadDefaultShader(p, kCCShaderType\_PositionTexture\_uColor);
105. m\_pPrograms->setObject(p ,kCCShader\_PositionTexture\_uColor);
106. p->release();
108. //新建第六种Shader代码片段
109. p = **new** CCGLProgram();
110. loadDefaultShader(p, kCCShaderType\_PositionTextureA8Color);
112. m\_pPrograms->setObject(p, kCCShader\_PositionTextureA8Color);
113. p->release();
115. //新建第七种Shader代码片段
116. p = **new** CCGLProgram();
117. loadDefaultShader(p, kCCShaderType\_Position\_uColor);
119. m\_pPrograms->setObject(p, kCCShader\_Position\_uColor);
120. p->release();
121. }
123. //重新载入
124. **void** CCShaderCache::reloadDefaultShaders()
125. {
126. //通过字符串名称找到对应的Shader代码片段
127. //
128. CCGLProgram \*p = programForKey(kCCShader\_PositionTextureColor);
129. //重置
130. p->reset();
131. //重新载入
132. loadDefaultShader(p, kCCShaderType\_PositionTextureColor);
134. // 同上,重新载入其它类型的Shader代码片段
135. p = programForKey(kCCShader\_PositionTextureColorAlphaTest);
136. p->reset();
137. loadDefaultShader(p, kCCShaderType\_PositionTextureColorAlphaTest);
139. p = programForKey(kCCShader\_PositionColor);
140. p->reset();
141. loadDefaultShader(p, kCCShaderType\_PositionColor);
143. p = programForKey(kCCShader\_PositionTexture);
144. p->reset();
145. loadDefaultShader(p, kCCShaderType\_PositionTexture);
147. p = programForKey(kCCShader\_PositionTexture\_uColor);
148. p->reset();
149. loadDefaultShader(p, kCCShaderType\_PositionTexture\_uColor);
151. p = programForKey(kCCShader\_PositionTextureA8Color);
152. p->reset();
153. loadDefaultShader(p, kCCShaderType\_PositionTextureA8Color);
155. p = programForKey(kCCShader\_Position\_uColor);
156. p->reset();
157. loadDefaultShader(p, kCCShaderType\_Position\_uColor);
158. }
160. //按照顶点格式类型载入相应的Shader文件,组成代码片段
161. **void** CCShaderCache::loadDefaultShader(CCGLProgram \*p, **int** type)
162. {
163. **switch** (type) {
164. **case** kCCShaderType\_PositionTextureColor:
165. //顶点格式为位置+纹理UV+材质色
166. //从“ccShader\_PositionTextureColor\_vert.h”和“ccShader\_PositionTextureColor\_frag.h”中为Shader代码片段加载VS和PS
167. p->initWithVertexShaderByteArray(ccPositionTextureColor\_vert, ccPositionTextureColor\_frag);
168. //将Shader代码的输入位置参数名称与索引进行绑定
169. p->addAttribute(kCCAttributeNamePosition, kCCVertexAttrib\_Position);
170. //将Shader代码的输入色彩参数名称与索引进行绑定
171. p->addAttribute(kCCAttributeNameColor, kCCVertexAttrib\_Color);
172. //将Shader代码的输入纹理坐标参数名称与索引进行绑定
173. p->addAttribute(kCCAttributeNameTexCoord, kCCVertexAttrib\_TexCoords);
175. **break**;
176. **case** kCCShaderType\_PositionTextureColorAlphaTest:
177. //顶点格式为位置+纹理UV+材质色+用于AlphaTest的A通道
178. //从 p->initWithVertexShaderByteArray(ccPositionTextureColor\_vert, ccPositionTextureColorAlphaTest\_frag);
179. //将Shader代码的输入位置参数名称与索引进行绑定
180. p->addAttribute(kCCAttributeNamePosition, kCCVertexAttrib\_Position);
181. //将Shader代码的输入色彩参数名称与索引进行绑定
182. p->addAttribute(kCCAttributeNameColor, kCCVertexAttrib\_Color);
183. //将Shader代码的输入玟理坐标参数名称与索引进行绑定
184. p->addAttribute(kCCAttributeNameTexCoord, kCCVertexAttrib\_TexCoords);
186. **break**;
187. **case** kCCShaderType\_PositionColor:
188. //顶点格式为位置+材质色
189. p->initWithVertexShaderByteArray(ccPositionColor\_vert ,ccPositionColor\_frag);
191. //将Shader代码的输入位置参数名称与索引进行绑定 p->addAttribute(kCCAttributeNamePosition, kCCVertexAttrib\_Position);
192. //将Shader代码的输入色彩参数名称与索引进行绑定p->addAttribute(kCCAttributeNameColor, kCCVertexAttrib\_Color);
194. **break**;
195. **case** kCCShaderType\_PositionTexture:
196. //顶点格式为位置+纹理坐标p->initWithVertexShaderByteArray(ccPositionTexture\_vert ,ccPositionTexture\_frag);
197. //将Shader代码的输入位置参数名称与索引进行绑定
198. p->addAttribute(kCCAttributeNamePosition, kCCVertexAttrib\_Position);
199. //将Shader代码的输入色彩参数名称与索引进行绑定
200. p->addAttribute(kCCAttributeNameTexCoord, kCCVertexAttrib\_TexCoords);
202. **break**;
203. **case** kCCShaderType\_PositionTexture\_uColor:
204. //顶点格式为位置+纹理坐标p->initWithVertexShaderByteArray(ccPositionTexture\_uColor\_vert, ccPositionTexture\_uColor\_frag);
205. //将Shader代码的输入位置参数名称与索引进行绑定
206. p->addAttribute(kCCAttributeNamePosition, kCCVertexAttrib\_Position);
207. //将Shader代码的输入纹理坐标参数名称与索引进行绑定
208. p->addAttribute(kCCAttributeNameTexCoord, kCCVertexAttrib\_TexCoords);
210. **break**;
211. **case** kCCShaderType\_PositionTextureA8Color:
212. //顶点格式为位置+灰度值p->initWithVertexShaderByteArray(ccPositionTextureA8Color\_vert, ccPositionTextureA8Color\_frag);
213. //将Shader代码的输入位置参数名称与索引进行绑定
214. p->addAttribute(kCCAttributeNamePosition, kCCVertexAttrib\_Position);
215. //将Shader代码的输入色彩参数名称与索引进行绑定
216. p->addAttribute(kCCAttributeNameColor, kCCVertexAttrib\_Color);
217. //将Shader代码的输入纹理坐标参数名称与索引进行绑定
218. p->addAttribute(kCCAttributeNameTexCoord, kCCVertexAttrib\_TexCoords);
220. **break**;
221. **case** kCCShaderType\_Position\_uColor:
222. p->initWithVertexShaderByteArray(ccPosition\_uColor\_vert, ccPosition\_uColor\_frag);
223. //将Shader代码的输入位置参数名称与索引进行绑定
224. p->addAttribute("aVertex", kCCVertexAttrib\_Position);
225. **break**;
226. **default**:
227. //如果是其它打印错误
228. CCLOG("cocos2d: %s:%d, error shader type", \_\_FUNCTION\_\_, \_\_LINE\_\_);
229. **return**;
230. }
231. //将VS与PS进行连接
232. p->link();
233. //更新输入参数
234. p->updateUniforms();
235. //检错
236. CHECK\_GL\_ERROR\_DEBUG();
237. }
238. //通过字符串键值查询出相应的Shader代码片段。
239. CCGLProgram\* CCShaderCache::programForKey(**const** **char**\* key)
240. {
241. **return** (CCGLProgram\*)m\_pPrograms->objectForKey(key);
242. }
243. //将一个新的Shader代码片段设置字符串键值存储到容器中
244. **void** CCShaderCache::addProgram(CCGLProgram\* program, **const** **char**\* key)
245. {
246. m\_pPrograms->setObject(program, key);
247. }
249. NS\_CC\_END

         好了，从这个文件我们可以知道。Shader缓冲类是对游戏中用到的Shader文件和代码片段进行统一的管理。

        看明白了，自然找一下相应的Shader文件。

        可以在ccShaders.cpp中的相应VS,PS的名称上右键弹出菜单第一项打开相应的Shader文件，也可以到Cocos2d-x解压目录下的cocos2dx\shaders目录下找到这些h文件，文件不多，那既然画点，线，面用到的Shader是"ShaderPosition\_uColor",在CCShaderCache::loadDefaultShader函数中我们得知其对应的VS文件是：ccShader\_Position\_uColor\_vert.h,PS文件是：ccShader\_Position\_uColor\_frag.h

        注：作者用.h来做为扩展名，其实这个用什么做扩展名都无所谓，总之是文本形式的文件就OK！

我们找来看一下：ccShader\_Position\_uColor\_vert.h:

**[html]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

2. attribute vec4 a\_position;          //定义vec4接口变量a\_position
3. uniform    mat4 u\_MVPMatrix;       //传入mat4参数u\_MVPMatrix，World矩阵乘View矩阵乘Proj矩阵的累积矩阵
4. uniform    vec4 u\_color;           //传入vec4参数u\_color,用于色彩
5. uniform float u\_pointSize;         //传入float 参数u\_pointSize,用于点的大小
6. //如果使用的是OpenGL的ES版本，使用低精度的vec4变量 v\_fragmentColor
7. #ifdef GL\_ES
8. varying lowp vec4 v\_fragmentColor;
9. #else
10. //如果使用的是OpenGL，使用高精度的vec4变量 v\_fragmentColor
11. varying vec4 v\_fragmentColor;
12. #endif
13. //VS的入口函数
14. void main()
15. {
16. //顶点x矩阵u\_MVPMatrix，将结果设置为VS返回屏幕最终显示位置
17. gl\_Position = u\_MVPMatrix \* a\_position;
18. //使用变量v\_fragmentColor设置为VS输出的顶点大小
19. gl\_PointSize = u\_pointSize;
20. //设置色彩变量值
21. v\_fragmentColor = u\_color;
22. }
23. ";

ccShader\_Position\_uColor\_frag.h:

**[html]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

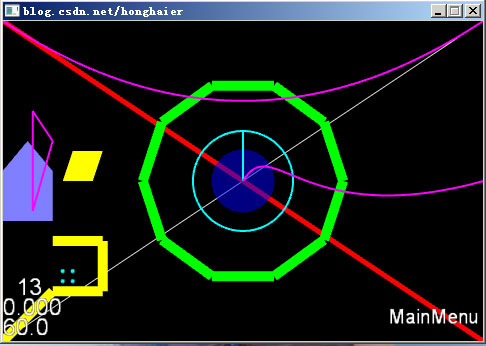
2. //如果使用的是OpenGL的ES版本，float精度设为低精度
3. #ifdef GL\_ES
4. precision lowp float;
5. #endif
6. //这里vec4变量 v\_fragmentColor，跟据OpenGL版本选择使用精度。
7. varying vec4 v\_fragmentColor;
8. //PS的入口函数
9. void main()
10. {
11. //使用变量v\_fragmentColor设为Opengl的PS输出色彩值
12. gl\_FragColor = v\_fragmentColor;
13. }
14. ";

        前边我们说过，此例程有个BUG。虽然传入了点的大小，但是实际却没有设置成功。作者应该是忘了在渲染前打开启用VS顶点大小功能，回到DrawPrimitivesTest.cpp，把开启和闭闭VS顶点大小的代码加上：

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. // 设置点大小64
2. ccPointSize(64);
3. //半透明的的蓝色
4. ccDrawColor4B(0,0,255,128);
5. //glEnable(GL\_POINT\_SPRITE\_ARB);加上此句为正方形,否则为圆形
6. //此处启用VS顶点大小功能
7. glEnable(GL\_VERTEX\_PROGRAM\_POINT\_SIZE);
8. //绘制点
9. ccDrawPoint( ccp(s.width / 2, s.height / 2) );
10. //检错
11. CHECK\_GL\_ERROR\_DEBUG();
12. // 绘制四个点
13. CCPoint points[] = { ccp(60,60), ccp(70,70), ccp(60,70), ccp(70,60) };
14. //点大小为4
15. ccPointSize(4);
16. //纯青色
17. ccDrawColor4B(0,255,255,255);
18. //绘制四个点
19. ccDrawPoints( points, 4);
20. //此处关闭VS顶点大小功能
21. glDisable(GL\_VERTEX\_PROGRAM\_POINT\_SIZE);
22. //glDisable(GL\_POINT\_SPRITE\_ARB);
23. CHECK\_GL\_ERROR\_DEBUG();

可以看到中心的半透明蓝色圆点和左下方四个小青圆点。



       看完ccDrawPoint函数，我们重新进入 CCDrawingPrimitives.cpp看其它的绘制图形函数：

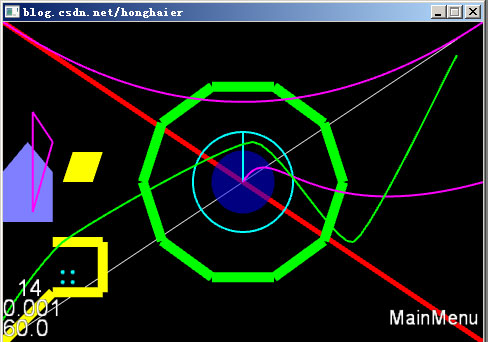
**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. //绘制多个点
2. //参1为顶点位置数组
3. //参2为顶点数量
4. **void** ccDrawPoints( **const** CCPoint \*points, unsigned **int** numberOfPoints )
5. {
6. //初始化
7. lazy\_init();
8. //Shader使用顶点位置属性
9. ccGLEnableVertexAttribs( kCCVertexAttribFlag\_Position );
10. //后面的渲染应用Shader
11. s\_pShader->use();
12. //设置Shader中的最终结果矩阵参数
13. s\_pShader->setUniformForModelViewProjectionMatrix();
14. //设置Shader中的色彩参数，取s\_tColor.r地址做为float4参数值的地址传入。实际Shader使用的是s\_tColor的所有四个float值。
15. s\_pShader->setUniformLocationWith4fv(s\_nColorLocation, (GLfloat\*) &s\_tColor.r, 1);
16. //设置Shader中的大小参数
17. s\_pShader->setUniformLocationWith1f(s\_nPointSizeLocation, s\_fPointSize);
19. //这里创建顶点位置数组，用于存储指定数量的顶点位置。这段代码放在这个位置有问题。看下一句分析。
20. ccVertex2F\* newPoints = **new** ccVertex2F[numberOfPoints];
22. //如果是32位的系统（Iphone和32位Windows）,这里似乎应该使用一个全局变量
23. **if**( **sizeof**(CCPoint) == **sizeof**(ccVertex2F) )
24. {
25. //将参数指向的顶点位置数据与位置属性进行绑定
26. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, points);
27. }
28. **else**
29. {
30. //ccVertex2F\* newPoints = new ccVertex2F[numberOfPoints]应该放在这里更好。因为如果是32位系统，则根本没有必要创建顶点数组。同样把后面的CC\_SAFE\_DELETE\_ARRAY(newPoints);放在这个else的大括号内。
32. // 如果是64位的系统,则遍历填充数据
33. **for**( unsigned **int** i=0; i<numberOfPoints;i++) {
34. newPoints[i].x = points[i].x;
35. newPoints[i].y = points[i].y;
36. }
37. //将新创建的顶点位置数组与位置属性进行绑定
38. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, newPoints);
39. }
40. //绘制顶点数组，参1为要绘制的图形为点，参2为顶点起始索引，参3为顶点数量
41. glDrawArrays(GL\_POINTS, 0, (GLsizei) numberOfPoints);
42. //释放申请的顶点数组占用内存。各位看官，这里可以研讨一下：我个人对在渲染时new这样一个数组后又delete的做法是感觉比较不妥的，可能产生碎片。如果没有多线程渲染，倒是不如直接在开始时就创建好一个固定长的数组，这里直接使用。缺点当然是定长可能会有浪费，但是所有渲染函数可以共用，避免了申请内存的时间开销，效率非常高，而且存的数据不多，占用不了太多的内存。
44. CC\_SAFE\_DELETE\_ARRAY(newPoints);
45. //统计渲染调用次数
46. CC\_INCREMENT\_GL\_DRAWS(1);
47. }
49. //画线函数
50. //参1:起点位置
51. //参2:终点位置
52. **void** ccDrawLine( **const** CCPoint& origin, **const** CCPoint& destination )
53. {
54. //初始化
55. lazy\_init();
56. //定义2个顶点的位置数组
57. ccVertex2F vertices[2] = {
58. {origin.x, origin.y},
59. {destination.x, destination.y}
60. };
61. //设定后面的渲染使用Shader
62. s\_pShader->use();
63. //检错
64. CHECK\_GL\_ERROR\_DEBUG();
65. //设置Shader中的最终结果矩阵参数
66. s\_pShader->setUniformForModelViewProjectionMatrix();
67. CHECK\_GL\_ERROR\_DEBUG();
68. //设置Shader中的色彩参数，取s\_tColor.r地址做为float4参数值的地址传入。实际Shader使用的是s\_tColor的所有四个float值。
69. s\_pShader->setUniformLocationWith4fv(s\_nColorLocation, (GLfloat\*) &s\_tColor.r, 1);
70. CHECK\_GL\_ERROR\_DEBUG();
71. //设置Shader中的大小参数
72. s\_pShader->setUniformLocationWith1f(s\_nPointSizeLocation, s\_fPointSize);
73. CHECK\_GL\_ERROR\_DEBUG();
74. //Shader使用顶点位置属性
75. ccGLEnableVertexAttribs( kCCVertexAttribFlag\_Position );
76. CHECK\_GL\_ERROR\_DEBUG();
77. //将新创建的顶点位置数组与位置属性进行绑定
78. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, vertices);
79. CHECK\_GL\_ERROR\_DEBUG();
80. //绘制顶点数组，参1为要绘制的图形为线段，参2为顶点起始索引，参3为顶点数量
81. glDrawArrays(GL\_LINES, 0, 2);
82. //统计渲染调用次数
83. CC\_INCREMENT\_GL\_DRAWS(1);
84. }
85. //绘制矩形线框
86. //参1:左下角位置点
87. //参2:右上角位置点
88. **void** ccDrawRect( CCPoint origin, CCPoint destination )
89. {
90. //绘制四条边，这样写倒是简单，不过效率不高，因为需要设多次渲染状态并渲染调用多次。不如在这里创建顶点数组调ccDrawPoly 效率高，设一次渲染状态，调用一次渲染处理就OK了。
92. ccDrawLine(CCPointMake(origin.x, origin.y), CCPointMake(destination.x, origin.y));
93. ccDrawLine(CCPointMake(destination.x, origin.y), CCPointMake(destination.x, destination.y));
94. ccDrawLine(CCPointMake(destination.x, destination.y), CCPointMake(origin.x, destination.y));
95. ccDrawLine(CCPointMake(origin.x, destination.y), CCPointMake(origin.x, origin.y));
96. }
97. //绘制实心矩形
98. //参1:左下角位置点
99. //参2:右上角位置点
100. //参3:填充色彩
101. **void** ccDrawSolidRect( CCPoint origin, CCPoint destination, ccColor4F color )
102. {
103. //创建四个顶点的位置数组
104. CCPoint vertices[] = {
105. origin,
106. ccp(destination.x, origin.y),
107. destination,
108. ccp(origin.x, destination.y)
109. };
110. //使用顶点位置数组做为参数调用绘制填充多边形
111. ccDrawSolidPoly(vertices, 4, color );
112. }
113. //绘制多线形线框
114. //参1:顶点位置数组
115. //参2:顶点数组中的顶点数量
116. //参3:是否封闭，即是否首尾相连
117. **void** ccDrawPoly( **const** CCPoint \*poli, unsigned **int** numberOfPoints, **bool** closePolygon )
118. {
119. //初始化
120. lazy\_init();
121. //设定后面的渲染使用Shader
122. s\_pShader->use();
123. //设置Shader中的最终结果矩阵参数
124. s\_pShader->setUniformForModelViewProjectionMatrix();
125. //设置Shader中的顶点色彩参数
126. s\_pShader->setUniformLocationWith4fv(s\_nColorLocation, (GLfloat\*) &s\_tColor.r, 1);
127. //Shader使用顶点位置属性
128. ccGLEnableVertexAttribs( kCCVertexAttribFlag\_Position );
130. //无奈，又把new放判断外边了。
131. ccVertex2F\* newPoli = **new** ccVertex2F[numberOfPoints];
133. //32位系统，实际用不到newPoli ,将参数poli与顶点位置属性绑定即可
134. **if**( **sizeof**(CCPoint) == **sizeof**(ccVertex2F) )
135. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, poli);
137. **else**
138. {
139. //64位系统，填充一下数据
140. **for**( unsigned **int** i=0; i<numberOfPoints;i++) {
141. newPoli[i].x = poli[i].x;
142. newPoli[i].y = poli[i].y;
143. }
144. //将newPoli与顶点位置属性绑定
145. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, newPoli);
146. }
147. //是否首尾相连
148. **if**( closePolygon )
149. glDrawArrays(GL\_LINE\_LOOP, 0, (GLsizei) numberOfPoints);
150. **else**
151. glDrawArrays(GL\_LINE\_STRIP, 0, (GLsizei) numberOfPoints);
152. //释放申请的顶点位置数组
153. CC\_SAFE\_DELETE\_ARRAY(newPoli);
154. //统计渲染调用次数
155. CC\_INCREMENT\_GL\_DRAWS(1);
156. }
157. //绘制填充色彩的多边形
158. //参1:顶点数组
159. //参2:顶点数量
160. //参3:色彩值
161. **void** ccDrawSolidPoly( **const** CCPoint \*poli, unsigned **int** numberOfPoints, ccColor4F color )
162. {
163. //初始化
164. lazy\_init();
165. //初始化
166. lazy\_init();
167. //设定后面的渲染使用Shader
168. s\_pShader->use();
169. //设置Shader中的最终结果矩阵参数
170. s\_pShader->setUniformForModelViewProjectionMatrix();
171. //设置Shader中的顶点色彩参数
172. s\_pShader->setUniformLocationWith4fv(s\_nColorLocation,(GLfloat\*) &s\_tColor.r, 1);
173. //Shader使用顶点位置属性
174. ccGLEnableVertexAttribs( kCCVertexAttribFlag\_Position );
176. //不说了，你懂的
177. ccVertex2F\* newPoli = **new** ccVertex2F[numberOfPoints];
178. **if**( **sizeof**(CCPoint) == **sizeof**(ccVertex2F) )
179. {
180. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, poli);
181. }
182. **else**
183. {
184. // Mac on 64-bit
185. **for**( unsigned **int** i=0; i<numberOfPoints;i++)
186. {
187. newPoli[i] = vertex2( poli[i].x, poli[i].y );
188. }
189. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, newPoli);
190. }
191. //这里参数1改为GL\_TRIANGLE\_FAN，即按扇面顺序方式绘制三角形。
192. glDrawArrays(GL\_TRIANGLE\_FAN, 0, (GLsizei) numberOfPoints);
193. //你懂的
194. CC\_SAFE\_DELETE\_ARRAY(newPoli);
195. CC\_INCREMENT\_GL\_DRAWS(1);
196. }
197. //绘制圆,在画圆算法中，圆其实是由多个小线段连线构成的封闭多边形线框。段数直多，就越像圆。
198. //参1:圆心位置
199. //参2:半径
200. //参3:圆在逆时针方向的转动角度
201. //参4:段数
202. //参5:是否在段起止点处向圆心连线
203. **void** ccDrawCircle( **const** CCPoint& center, **float** radius, **float** angle, unsigned **int** segments, **bool** drawLineToCenter)
204. {
205. //初始化
206. lazy\_init();
207. //这里设变量additionalSegment为增加段的数量，默认为1。即用来做最后尾部与首部相连以形成封闭线框的线段。
208. **int** additionalSegment = 1;
209. //如果在段起止点处向圆心连线，就再加一条线段。
210. **if** (drawLineToCenter)
211. additionalSegment++;
213. //通过圆周的弧度除以段数计算每个段所跨的弧度
214. **const** **float** coef = 2.0f \* (**float**)M\_PI/segments;
215. //申请相应的顶点数组，纳闷了这里为什么不用ccVertex2F而使用GLfloat来存顶点的位置，因为一个顶点的位置是二维点，有x,y两个float值才能表示。所以这里在计算内存大小时乘2,段数值它这里用了最大可能情况值(segments + 2),也可以改为segments + additionalSegment，这样如果不在段起止点处向圆心连线，内存申请会小一点，但后面储存圆心也需要做改动处理。
216. GLfloat \*vertices = (GLfloat\*)calloc( **sizeof**(GLfloat)\*2\*(segments+2), 1);
217. **if**( ! vertices )
218. **return**;
219. //for循环每一个段设置顶点位置
220. **for**(unsigned **int** i = 0;i <= segments; i++) {
221. **float** rads = i\*coef;
222. GLfloat j = radius \* cosf(rads + angle) + center.x;
223. GLfloat k = radius \* sinf(rads + angle) + center.y;
225. vertices[i\*2] = j;
226. vertices[i\*2+1] = k;
227. }
228. //存储圆心,这里也可以改动一下效率更好，判断一下additionalSegment是否大于1才处理。
229. vertices[(segments+1)\*2] = center.x;
230. vertices[(segments+1)\*2+1] = center.y;
231. //你懂的，亲~
232. s\_pShader->use();
233. s\_pShader->setUniformForModelViewProjectionMatrix();
234. s\_pShader->setUniformLocationWith4fv(s\_nColorLocation, (GLfloat\*) &s\_tColor.r, 1);
235. ccGLEnableVertexAttribs( kCCVertexAttribFlag\_Position );
236. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, vertices);
237. //这里参数1指定按GL\_LINE\_STRIP顺序方式绘制线段。
238. glDrawArrays(GL\_LINE\_STRIP, 0, (GLsizei) segments+additionalSegment);
239. //你懂的~
240. free( vertices );
241. CC\_INCREMENT\_GL\_DRAWS(1);
242. }
243. //绘制二次贝塞尔曲线
244. //参1:起点
245. //参2:控制点
246. //参3:结束点
247. //参4:构成曲线的线段数
248. **void** ccDrawQuadBezier(**const** CCPoint& origin, **const** CCPoint& control, **const** CCPoint& destination, unsigned **int** segments)
249. {
250. //初始化
251. lazy\_init();
252. //创建顶点数组
253. ccVertex2F\* vertices = **new** ccVertex2F[segments + 1];
254. //遍历每一段，用计算公式计算点的位置
255. **float** t = 0.0f;
256. **for**(unsigned **int** i = 0; i < segments; i++)
257. {
258. vertices[i].x = powf(1 - t, 2) \* origin.x + 2.0f \* (1 - t) \* t \* control.x + t \* t \* destination.x;
259. vertices[i].y = powf(1 - t, 2) \* origin.y + 2.0f \* (1 - t) \* t \* control.y + t \* t \* destination.y;
260. t += 1.0f / segments;
261. }
262. //目标点
263. vertices[segments].x = destination.x;
264. vertices[segments].y = destination.y;
265. //使用Shader进行渲染
266. s\_pShader->use();
267. s\_pShader->setUniformForModelViewProjectionMatrix();
268. s\_pShader->setUniformLocationWith4fv(s\_nColorLocation, (GLfloat\*) &s\_tColor.r, 1);
269. ccGLEnableVertexAttribs( kCCVertexAttribFlag\_Position );
270. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, vertices);
271. //这里参数1指定按GL\_LINE\_STRIP顺序方式绘制线段。
272. glDrawArrays(GL\_LINE\_STRIP, 0, (GLsizei) segments + 1);
273. CC\_SAFE\_DELETE\_ARRAY(vertices);
274. CC\_INCREMENT\_GL\_DRAWS(1);
275. }
276. //绘制默认曲率的基数样条。关于基数样条可以参看：http://technet.microsoft.com/zh-cn/4cf6we5y(v=vs.85)
277. //基数样条是一连串单独的曲线，这些曲线连接起来形成一条较大的曲线。 样条由点的数组和张力参数指定。 基数样条平滑地经过数组中的每个点；曲线的陡度上没有尖角和突然的变化。 下面的插图显示了一组点和经过这一组点中每一点的基数样条。
278. //参1:顶点位置数组
279. //参2:构成曲线的线段数
280. **void** ccDrawCatmullRom( CCPointArray \*points, unsigned **int** segments )
281. {
282. ccDrawCardinalSpline( points, 0.5f, segments );
283. }
284. //绘制可指定曲率的基数样条
285. //参1:顶点位置数组
286. //参2:曲率
287. //参3:构成曲线的线段数
288. **void** ccDrawCardinalSpline( CCPointArray \*config, **float** tension,  unsigned **int** segments )
289. {
290. lazy\_init();
292. ccVertex2F\* vertices = **new** ccVertex2F[segments + 1];
294. unsigned **int** p;
295. **float** lt;
296. **float** deltaT = 1.0f / config->count();
298. **for**( unsigned **int** i=0; i < segments+1;i++) {
300. **float** dt = (**float**)i / segments;
302. // border
303. **if**( dt == 1 ) {
304. p = config->count() - 1;
305. lt = 1;
306. } **else** {
307. p = dt / deltaT;
308. lt = (dt - deltaT \* (**float**)p) / deltaT;
309. }
311. // Interpolate
312. CCPoint pp0 = config->getControlPointAtIndex(p-1);
313. CCPoint pp1 = config->getControlPointAtIndex(p+0);
314. CCPoint pp2 = config->getControlPointAtIndex(p+1);
315. CCPoint pp3 = config->getControlPointAtIndex(p+2);
317. CCPoint newPos = ccCardinalSplineAt( pp0, pp1, pp2, pp3, tension, lt);
318. vertices[i].x = newPos.x;
319. vertices[i].y = newPos.y;
320. }
322. s\_pShader->use();
323. s\_pShader->setUniformForModelViewProjectionMatrix();
324. s\_pShader->setUniformLocationWith4fv(s\_nColorLocation, (GLfloat\*)&s\_tColor.r, 1);
326. ccGLEnableVertexAttribs( kCCVertexAttribFlag\_Position );
328. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, vertices);
329. glDrawArrays(GL\_LINE\_STRIP, 0, (GLsizei) segments + 1);
331. CC\_SAFE\_DELETE\_ARRAY(vertices);
332. CC\_INCREMENT\_GL\_DRAWS(1);
333. }

在本例程中，没有展示基数样条，我稍做了下改动,在绘制图形的函数加入了:

**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. //这里创建5个顶点
2. CCPointArray\*  tpSplinePtArray = CCPointArray::arrayWithCapacity(5);
3. //填充顶点
4. tpSplinePtArray->addControlPoint(ccp(0, 0));
5. tpSplinePtArray->addControlPoint(ccp(60,100));
6. tpSplinePtArray->addControlPoint(ccp(250,200));
7. tpSplinePtArray->addControlPoint(ccp(350,100));
8. tpSplinePtArray->addControlPoint(ccp(450,280));
9. //用默认曲率绘制段数为50的绿色基数样条
10. ccDrawColor4B(0,255,0,255);
11. ccDrawCatmullRom(tpSplinePtArray,50);
13. //重置绘制参数值
14. glLineWidth(1);
15. ccDrawColor4B(255,255,255,255);
16. ccPointSize(1);
17. 运行后可以看到多了一条绿色基数样条。

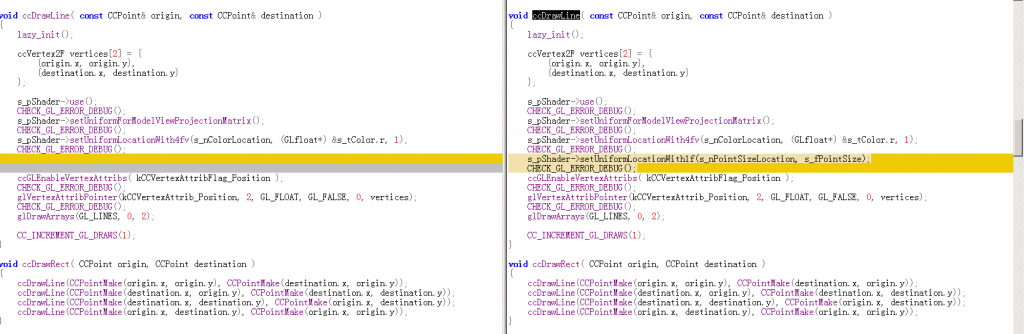
然后继续：

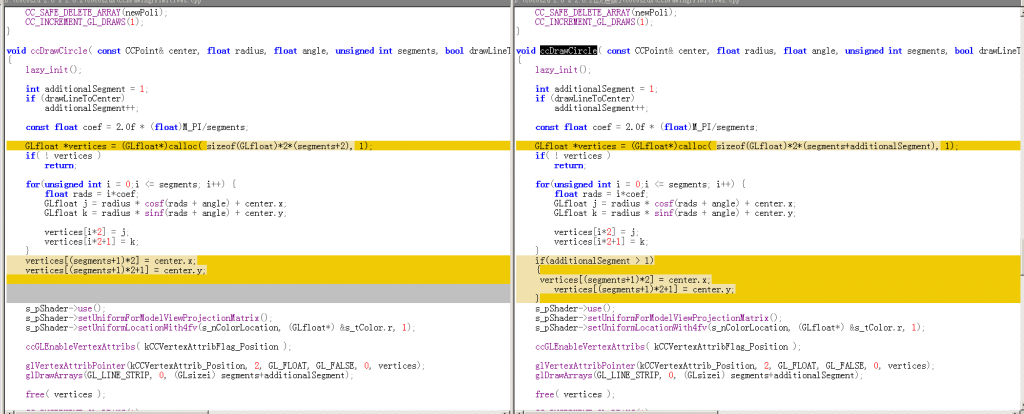
**[cpp]** [view plaincopy](http://blog.csdn.net/honghaier/article/details/7973078)

1. //绘制高阶贝赛尔曲线
2. //参1:起点
3. //参2:控制点1
4. //参3:控制点2
5. //参4:结束度
6. //参5:构成曲线的线段数
7. **void** ccDrawCubicBezier(**const** CCPoint& origin, **const** CCPoint& control1, **const** CCPoint& control2, **const** CCPoint& destination, unsigned **int** segments)
8. {
9. lazy\_init();
11. ccVertex2F\* vertices = **new** ccVertex2F[segments + 1];
13. **float** t = 0;
14. **for**(unsigned **int** i = 0; i < segments; i++)
15. {
16. vertices[i].x = powf(1 - t, 3) \* origin.x + 3.0f \* powf(1 - t, 2) \* t \* control1.x + 3.0f \* (1 - t) \* t \* t \* control2.x + t \* t \* t \* destination.x;
17. vertices[i].y = powf(1 - t, 3) \* origin.y + 3.0f \* powf(1 - t, 2) \* t \* control1.y + 3.0f \* (1 - t) \* t \* t \* control2.y + t \* t \* t \* destination.y;
18. t += 1.0f / segments;
19. }
20. vertices[segments].x = destination.x;
21. vertices[segments].y = destination.y;
23. s\_pShader->use();
24. s\_pShader->setUniformForModelViewProjectionMatrix();
25. s\_pShader->setUniformLocationWith4fv(s\_nColorLocation, (GLfloat\*) &s\_tColor.r, 1);
27. ccGLEnableVertexAttribs( kCCVertexAttribFlag\_Position );
29. glVertexAttribPointer(kCCVertexAttrib\_Position, 2, GL\_FLOAT, GL\_FALSE, 0, vertices);
30. glDrawArrays(GL\_LINE\_STRIP, 0, (GLsizei) segments + 1);
31. CC\_SAFE\_DELETE\_ARRAY(vertices);
33. CC\_INCREMENT\_GL\_DRAWS(1);
34. }
35. //设置静态色彩变量值s\_tColor，因为很多绘制图形函数都用到这个值做为色彩参数传入Shader,故要更改色彩，在渲染前调用此函数进行设置即可。这里参数分别代表r,g,b,a,用0.0~1.0来表示0~255的值
36. **void** ccDrawColor4F( GLfloat r, GLfloat g, GLfloat b, GLfloat a )
37. {
38. s\_tColor.r = r;
39. s\_tColor.g = g;
40. s\_tColor.b = b;
41. s\_tColor.a = a;
42. }
43. //设置代表顶点大小的静态变量值，用于渲染点时传入Shader
44. **void** ccPointSize( GLfloat pointSize )
45. {
46. s\_fPointSize = pointSize \* CC\_CONTENT\_SCALE\_FACTOR();
48. }
49. //设置静态色彩变量值s\_tColor，这里参数用0~255。
50. **void** ccDrawColor4B( GLubyte r, GLubyte g, GLubyte b, GLubyte a )
51. {
52. s\_tColor.r = r/255.0f;
53. s\_tColor.g = g/255.0f;
54. s\_tColor.b = b/255.0f;
55. s\_tColor.a = a/255.0f;
56. }

另外，改动两个Cocos2d-x的底层代码：在cocos2dx目录的CCDrawingPrimitives.cpp中

增加线宽：左边是原代码，右边是改后代码

  
 优化画圆



           好啦，啰啰嗦嗦总算是把这一章讲完了，希望大家学有所用。下次再见！