**opengl的FBO**

http://blog.csdn.net/xiajun07061225/article/details/7283929/

**OpenGL默认的帧缓存包括：颜色缓存、深度缓存、模板缓存、累积缓存。**

**OpenGL扩展帧缓存（即FBO）包括：颜色缓存、深度缓存、模板缓存。**

在opengl渲染管线中，几何数据和纹理经过多次转化和测试，最后以二维像素的形式显示在屏幕上。opengl管线的最终渲染目的地被称为帧缓冲。帧缓冲是一些二维数组和opengl所使用的存储区的集合：颜色缓存、深度缓存、模板缓存和累积缓存。一般情况下，帧缓存完全由window系统生成和管理，由opengl使用，这个默认的帧缓存称为“window系统生成”（window-system-provided）的帧缓存。

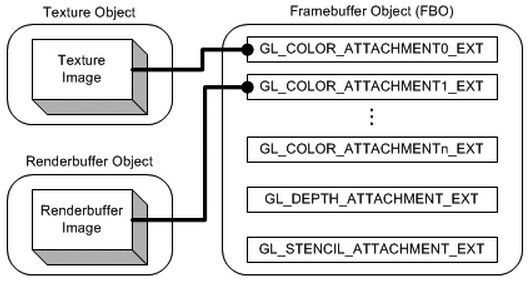
FBO最常见的应用就是：渲染到纹理（render to texture），通过这项技术可以实现发光效果、环境映射、阴影映射等很炫的效果，FBO被推荐用于把数据渲染到纹理对象，相对于其他同类技术，如数据拷贝或交换缓冲区等，使用FBO技术会更高效并且容易实现。

在opengl扩展中，GL\_EXT\_framebuffer\_object提供了一种创建额外的不能显示的帧缓存对象接口。为了和默认的“window系统生成”的帧缓存区别，这种缓存就称为应用程序帧缓存（application-createframebuffer）。通过使用帧缓存对象（FBO），opengl可以将显示输出到引用程序帧缓存对象，而不是传统的“window系统生成”帧缓存，而且，它完全受opengl控制。

类似于window系统提供的帧缓存，一个FBO也包含一些存储颜色、深度和模板数据的区域**（注意：没有累积缓存）**，我们把FBO中这些逻辑缓存称为“帧缓存关联图像”，它们是一些能够和一个帧缓存对象关联起来的二位数组像素。

有两种类型的“帧缓存关联图像“：纹理图像（texture images）和渲染缓存图像（renderbufer images）。如果纹理对象的图像数据关联到帧缓存，opengl执行的是”渲染到纹理“操作，如果渲染缓存的图像数据关联到帧缓存，opengl执行的是离线渲染（offscreen rendering）。

这里要注意的是，渲染缓存对象是在GL\_EXT\_framebuffer\_object扩展中定义的一种新的存储类型，在渲染过程中它被用作存储单幅二维图像。下面这幅图显示了帧缓存对象、纹理对象和渲染缓存对象之间的联系。多个纹理对象或者渲染缓存对象能够通过关联点关联到一个帧缓存对象上。



在一个帧缓存对象中有多个颜色关联点（GL\_COLOR\_ATTACHMENT0\_EXT... GL\_COLOR\_ATTACHMENTn\_EXT），一个深度关联点（GL\_DEPTH\_ATTACHMENT\_EXT）和一个模板关联点（GL\_STENCIL\_ATTACKMENT\_EXT）。每个FBO中至少有一个颜色关联点，其数目与实体显卡相关。可以通过GL\_MAX\_COLOR\_ATTACKMENTS\_EXT来查询颜色关联点的最大数目。FBO有多个颜色关联点的原因是这样可以同时将颜色而换成渲染到多个FBO关联去。这种“多渲染目标（multiple rendertargets，MRT）”可以通过GL\_ARC\_draw\_buffers扩展实现，需要注意的是：FBO本身并没有任何图像存储区，只有多个关联点。

FBO提供了一种高效的切换机制，将前面的帧缓存关联图像从FBO分离，然后把新的帧缓存关联图像关联到FBO。在帧缓存关联图像之间切换比在FBO之间切换要快得多，FBO提供了glFramebufferTexture2DEXT来切换2D纹理对象和glFramebufferRenderbufferEXT来切换渲染缓存对象。

1）glGenFramebuffers(GLsizei n, GLuint \*ids)

该函数需要两个参数：第一个是要创建的帧缓存的数目，第二个是指向存储一个或者多个ID的变量或数组的指针。它返回未使用的FBO的ID。ID为0表示默认帧缓存，即window系统提供的帧缓存

2）glBindFramebuffer(GLenum target, GLuint id)

一旦一个FBO被创建，在使用它之前必须绑定

第一个参数target应该是GL\_FRAMEBUFFER\_EXT，第二个参数是FBO的ID号，一旦FBO被绑定，之后的所有opengl操作都会对当前所绑定的FBO造成影响，ID号为0表示缺省帧缓存，即默认的window提供的帧缓存，因此，在glBindFramebufferEXT中将ID号设置为0可以解绑定当前的FBO

构建一个完整的额帧缓冲必须满足以下条件：

1）我们必须往里面加入至少一个附件（颜色、深度、模板缓冲）

2）其中至少有一个是颜色附件

3）所有的附件都应该是已经完全做好的（已经存储在内存中）

4）每个缓冲都应该有同样数目的样本

**渲染缓存对象（Renderbuffer Object）**

另外，渲染缓存是为了离线渲染而新引进的，它允许将一个场景渲染到一个渲染缓存对象中，而不是渲染到纹理对象中。渲染缓存对象是用于存储单幅图像的数据存储区域。该图像按照一种可渲染的内部格式存储，它用于存储没有相关纹理格式的opengl逻辑缓存，比如模板缓存或者深度缓存。

渲染缓冲对象的一大优点是，它以opengl原生渲染格式存储它的数据，因此在离屏渲染到帧缓冲的时候，这些数据就相当于被优化过了的。

渲染缓冲对象将所有渲染数据直接存储到它们的缓冲中，而不会进行针对特定纹理格式的任何转换，这样它们就变成了一种快速可写的存储介质了。和纹理图像一样，渲染缓冲对象也是一个缓冲，它可以是一堆字节、整数、像素或者其他东西。

由于我们只采样颜色缓冲，并不采样其他缓冲，我们可以创建一个渲染缓冲对象来达到这个目的。记住：当你不打算从指定缓冲采样的时候，它们是一个不错的选择。

1）glGenRenderbufferEXT

一旦一个渲染缓存对象被创建，它返回一个非零的正整数，ID为0是opengl的保留值

2）glBindRenderbufferEXT

和opengl中其他对象一样，在引用渲染缓存之前必须绑定当前渲染缓存对象，它的target参数应该是GL\_RENDERBUFFER\_EXT

3）glRenderbufferStorageEXT

当一个渲染缓存被创建，它没有任何数据存储区域，所以我们还要为它分配空间。第一个参数必须是GL\_RENDERBUFFER\_EXT，第二个参数可以是用于颜色的（GL\_RGB、GL\_RGBA等），用于深度（GL\_DEPTH\_COMPONENT）或者是用于模板的格式（GL\_STENCIL\_INDEX），width和height是渲染缓存图像的像素维度。

4）glGetRenderbufferParameterivEXT(GLenum target，GLenum param，GLint \*value)

我们可以得到当前绑定的渲染缓存对象的一些参数。Target应该是GL\_RENDERBUFFER\_EXT，第二个参数是所要得到的参数名字，最后一个是指向存储返回值的整型量的指针。渲染缓存的变量名有如下：

GL\_RENDERBUFFER\_WIDTH\_EXT

GL\_RENDERBUFFER\_HEIGHT\_EXT

GL\_RENDERBUFFER\_INTERNAL\_FORMAT\_EXT

GL\_RENDERBUFFER\_RED\_SIZE\_EXT

GL\_RENDERBUFFER\_GREEN\_SIZE\_EXT

GL\_RENDERBUFFER\_BLUE\_SIZE\_EXT

GL\_RENDERBUFFER\_ALPHA\_SIZE\_EXT

GL\_RENDERBUFFER\_DEPTH\_SIZE\_EXT

GL\_RENDERBUFFER\_STENCIL\_SIZE\_EXT

**将图像和FBO关联**

FBO本身没有图像缓冲区，我们必须将帧缓存关联图像（纹理或渲染对象）关联到FBO，这种机制允许FBO快速地切换（分离和关联）帧缓存关联图像。切换帧缓存关联图像比在FBO之间切换要快得多，而且，这节省了不必要的数据拷贝和内存消耗。比如一个纹理可以摆关联到多个FBO上，乳香存储区可以被多个FBO共享。

**把2D纹理图像关联到FBO**

glFramebufferTexrure2DEXT(GLeunm target, GLenumattachmentPoint, GLenum textureTarget, GLuint textureId, GLint level)

glFramebufferTexrure2DEXT将一幅纹理图像关联到FBO，第一个参数一定是GL\_FRAMEBUFFER\_EXT，第二个参数是关联纹理图像的关联点，第三个参数textureTarget在多数情况下是GL\_TEXTURE\_2D，第四个参数是纹理对象的ID号。最后一个参数是要被关联额纹理的mipmap等级。

如果参数textureId被设置为0，那么纹理图像将会被从FBO分离。如果纹理对象在关联的FBO上时删除，那么纹理对象将会自动从当前的FBO上分离，然而，如果它被关联到多个FBO上然后被删除，那么它将只被从绑定的FBO上分离，而不会从其他非绑定的FBO上分离。

**把渲染缓存对象关联到FBO**

void glFramebufferRenderbufferEXT(GLenum target, GLenum attachmentPoint, GLenum renderbufferTarget, GLuint renderbufferId)

通过调用glFramebufferRenderbufferEXT可以关联渲染缓存图像，前两个参数和glFramebufferTexrure2DEXT一样，第三个参数只能是GL\_RENDERBUFFER\_EXT，最后一个参数是渲染缓存对象的ID号。

如果参数renderbufferId被设置为0，那么纹理图像将会被从FBO分离。如果纹理对象在关联的FBO上时删除，那么纹理对象将会自动从当前的FBO上分离，然而，如果它被关联到多个FBO上然后被删除，那么它将只被从绑定的FBO上分离，而不会从其他非绑定的FBO上分离。

**检查FBO状态**

一旦关联图像（纹理和渲染缓存）被关联到FBO，在执行FBO操作之前，必须检查FBO的状态，这可以通过调用glCheckFramebufferStatusEXT实现，如果这个FBO不完整，那么任何绘制和读取命令都会失败。

FBO完整性的准则有：

1）帧缓存关联图像的宽度和高度必须非零

2）如果一幅图像被关联到一个颜色关联点，那么这幅图像必须有颜色可渲染的内部格式（GL\_RGBA、GL\_DEPTH\_COMPONENT、GL\_LUMINANCE等）

3）如果一幅被图像关联到GL\_DEPTH\_ATTACHMENT\_EXT，那么这堵图像必须有深度可渲染的内部格式（GL\_DEPTH\_COMPONENT、GL\_DEPTH\_COMPONENT24\_EXT等）

4）如果一幅图像被关联到GL\_STENCIL\_ATTACHMENT\_EXT，那么这幅图像必须有模板可渲染的内部格式（GL\_STENCIL\_INDEX、GL\_STENCIL\_INDEX8\_EXT等）

5）FBO至少有一幅图像关联

6）被关联到FBO的缩影图像必须有相同的宽度和高度

7）被关联到颜色关联点上的所有图像必须有相同的内部格式

有时候，你需要产生动态纹理。比较常见的是产生镜面反射效果，动态环境贴图和阴影等效果。动态纹理可以通过把场景渲染到纹理来实现。渲染到纹理的一种传统方式是将场景绘制到普通的帧缓存上，然后调用glCopyTexSubImage2D拷贝帧缓存图像到纹理。（见下面的摘抄）

使用FBO，我们能够将场景直接渲染到纹理，所以我们不必使用window系统提供的帧缓存，并且，我们能够去除额外的数据拷贝（从帧缓存到纹理）

这面这个demo实现了使用FBO和不使用FBO两种情况渲染到纹理的操作，并且比较了性能差异，除了能够获得性能上的提升，使用FBO还有另外一个优点，在传统的渲染到纹理的模式中（不使用FBO），如果纹理分辨率比渲染窗口的尺寸大，超出窗口区域的部分将会裁剪掉，然而，使用FBO就不会有这个纹理，你可以产生比现实窗口大的帧缓存对象。

// 这里如果设置为屏幕大小（以左下角为原点）

static const GLuint TEXTURE\_WIDTH = 600;

static const GLuint TEXTURE\_HEIGHT = 600;

void FBO::setRC()

{

cout << "fbo setup" << endl;

glGenFramebuffers(1, &fboHandler); // 创建fbo

glBindFramebuffer(GL\_FRAMEBUFFER, fboHandler); // 绑定帧缓冲

// 创建帧缓冲纹理

glGenTextures(1, &fboTexture);

glBindTexture(GL\_TEXTURE\_2D, fboTexture);

// 只分配空间，而不指定数据

glTexImage2D(GL\_TEXTURE\_2D, 0, GL\_RGB, TEXTURE\_WIDTH, TEXTURE\_HEIGHT, 0, GL\_RGB, GL\_UNSIGNED\_BYTE, nullptr);

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

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

// 将纹理附加到帧缓冲的颜色附件上

glFramebufferTexture2D(GL\_FRAMEBUFFER, GL\_COLOR\_ATTACHMENT0, GL\_TEXTURE\_2D, fboTexture, 0);

// 创建渲染缓冲对象

GLuint rbo;

glGenRenderbuffers(1, &rbo);

glBindRenderbuffer(GL\_RENDERBUFFER, rbo);

glRenderbufferStorage(GL\_RENDERBUFFER, GL\_DEPTH24\_STENCIL8, TEXTURE\_WIDTH, TEXTURE\_HEIGHT);

glBindRenderbuffer(GL\_RENDERBUFFER, 0);

// 将渲染缓冲对象附加到帧缓冲的深度和模板附件上

glFramebufferRenderbuffer(GL\_FRAMEBUFFER, GL\_DEPTH\_STENCIL\_ATTACHMENT, GL\_RENDERBUFFER, rbo);

if (glCheckFramebufferStatus(GL\_FRAMEBUFFER) == GL\_FRAMEBUFFER\_COMPLETE)

{

cout << "frame buffer create success." << endl;

}

else

{

cout << "frame buffer create failed." << endl;

}

modelCube.initProgram("shaderObject.vert", "shaderObject.frag");

modelCube.bindAttrib(1, "a\_position");

modelCube.bindAttrib(2, "a\_color");

modelCube.bindAttrib(3, "a\_normal");

modelCube.bindAttrib(4, "a\_textureCoord");

modelCube.link();

modelCube.initTexturePath("..\\front.jpg");

modelCube.initData(verticesCube, indicesCube, sizeof(verticesCube) / sizeof(GL\_FLOAT), sizeof(indicesCube) / sizeof(GL\_UNSIGNED\_INT));

modelCube.begin();

modelCube.setTranslate(0.0f, 0.0f, -50.0f);

modelCube.showData();

modelCube2.initProgram("shaderObject.vert", "shaderObject.frag");

modelCube2.bindAttrib(1, "a\_position");

modelCube2.bindAttrib(2, "a\_color");

modelCube2.bindAttrib(3, "a\_normal");

modelCube2.bindAttrib(4, "a\_textureCoord");

modelCube2.setTranslate(0.0f, 0.0f, 50.0f);

modelCube2.link();

modelCube2.initTexturePath("..\\front.jpg");

modelCube2.initData(verticesCube2, indicesCube2, sizeof(verticesCube2) / sizeof(GL\_FLOAT), sizeof(indicesCube2) / sizeof(GL\_FLOAT));

modelCube2.begin();

}

void FBO::render()

{

glBindFramebuffer(GL\_FRAMEBUFFER, fboHandler);

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

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

glEnable(GL\_DEPTH\_TEST);

modelCube.use();

modelCube.update();

modelCube.draw(projection, view);

glBindFramebuffer(GL\_FRAMEBUFFER, 0);

glClearColor(1.0f, 1.0f, 1.0f, 1.0f);

glClear(GL\_COLOR\_BUFFER\_BIT);

modelCube2.setTextureId(fboTexture);

modelCube2.use();

modelCube2.draw(projection, view);

}

在不采用FBO的时候，下面是摘抄的一段：

glClearColor(1, 1, 1, 1);

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

glPushAttrib(GL\_COLOR\_BUFFER\_BIT | GL\_PIXEL\_MODE\_BIT); // for GL\_DRAW\_BUFFER and GL\_READ\_BUFFER

glDrawBuffer(GL\_BACK);

glReadBuffer(GL\_BACK);

// draw a rotating teapot at the origin

glPushMatrix();

glRotatef(angle\*0.5f, 1, 0, 0);

glRotatef(angle, 0, 1, 0);

glRotatef(angle\*0.7f, 0, 0, 1);

glTranslatef(0, -1.575f, 0);

drawTeapot();

glPopMatrix();

// copy the framebuffer pixels to a texture

glBindTexture(GL\_TEXTURE\_2D, textureId);

glCopyTexSubImage2D(GL\_TEXTURE\_2D, 0, 0, 0, 0, 0, TEXTURE\_WIDTH, TEXTURE\_HEIGHT);

glBindTexture(GL\_TEXTURE\_2D, 0);

glPopAttrib(); // GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT

