江南烟雨

江南好，风景旧曾谙。日出江花红胜火，春来江水绿如蓝。能不忆江南？ ！

个人资料

江南烟雨

访问： 729317次

积分： 9011

等级： BLOG 6

排名： 第811名

原创： 121篇 转载： 15篇

译文： 6篇 评论： 801条

个人简介

2014年3月毕业于某男子职业技术学院，专业计算机。4月开始就职于杭州某互联网大型砖厂。职位开发工程师-Java。个人邮箱：badboyhust@163.com。

【OpenGL】OpenGL帧缓存对象(FBO：Frame Buffer Object)

2012-02-22 16:36 14736人阅读 评论(8) 收藏 举报

object buffer generation parameters

目录(?)

翻译的，如果不正确，敬请谅解和指正。

OpenGL Frame BufferObject(FBO)

Overview:

在OpenGL渲染管线中，几何数据和纹理经过多次转化和多次测试，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作帧缓存（framebuffer）。帧缓冲是一些二维数组和OpenG所使用的存储区的集合：颜色缓存、深度缓存、模板缓存和累计缓存。一般情况下，帧缓存完全由window系统生成和管理，由OpenGL使用。这个默认的帧缓存被称作“window系统生成”（window-system-provided）的帧缓存。

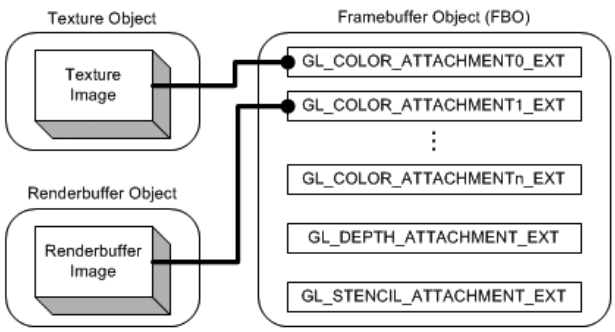
在OpenGL扩展中，GL_EXT_framebuffer_object提供了一种创建额外的不能显示的帧缓存对象的接口。为了和默认的“window系统生成”的帧缓存区别，这种帧缓冲成为应用程序帧缓存（application-createdframebuffer）。通过使用帧缓存对象（FBO），OpenGL可以将显示输出到引用程序帧缓存对象，而不是传统的“window系统生成”帧缓存。而且，它完全受OpenGL控制。

相似于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中这些逻辑缓存称之为“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维数组像素。

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（renderbuffer 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”（render to texture）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）。

这里要提到的是，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_ATTACHMENT_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_ATTACHMENTS_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关联区。这种“多渲染目标”（multiple rendertargets,MRT）可以通过GL_ARB_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。

创建FBO

创建FBO和产生VBO类似。

glGenFramebuffersEXT()

Void gelGenFramebuffersEXT(GLsizei n,GLuint* ids);

void glDeleteFramebuffersEXT(GLsizei n, const GLuint* ids);

glGenFramebuffersEXT()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。

当FBO不再被使用时，FBO可以通过调用glDeleteFrameBuffersEXT()来删除。

glBindFramebufferEXT()

一旦一个FBO被创建，在使用它之前必须绑定。

void glBindFramebufferEXT(GLenum target, GLuint id)

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，之后的所有的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FBO。

渲染缓存对象（Renderbuffer Object）

另外，渲染缓存是为离线渲染而新引进的。它允许将一个场景直接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。

glGenRenderbuffersEXT()

void glGenRenderbuffersEXT(GLsizei n, GLuint* ids)

void glDeleteRenderbuffersEXT(GLsizei n, const Gluint* ids)

一旦一个渲染缓存被创建，它返回一个非零的正整数。ID为0是OpenGL保留值。

glBindRenderbufferEXT()

void glBindRenderbufferEXT(GLenum target, GLuint id)

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。他target参数应该是GL_RENDERBUFFER_EXT。

glRenderbufferStorageEXT()

void glRenderbufferStorageEXT(GLenum target, GLenum internalFormat,
GLsizei width, GLsizei height)

当一个渲染缓存被创建，它没有任何数据存储区域，所以我们还要为他分配空间。这可以通过用glRenderbufferStorageEXT()实现。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，etc.），用于深度的（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。

width和height必须比GL_MAX_RENDERBUFFER_SIZE_EXT小，否则将会产生GL_UNVALID_VALUE错误。

glGetRenderbufferParameterivEXT()

void glGetRenderbufferParameterivEXT(GLenum target, GLenum param,GLint* value);

我们也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本身没有图像存储区。我们必须帧缓存关联图像（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，它节省了不必要的数据拷贝和内存消耗。比如，一个纹理可以被关联到多个FBO上，图像存储区可以被多个FBO共享。

把2D纹理图像关联到FBO

```
glFramebufferTexture2DEXT(GLenum target,
                          GLenumattachmentPoint,
                          GLenum textureTarget,
                          GLuint textureId,
                          GLint  level)
```

glFramebufferTexture2DEXT()把一幅纹理图像关联到一个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的纹理的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。如果纹理对象在依然关联在FBO上时被删除，那么纹理对象将会自动从当前帮的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，而不会被从其他非绑定的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glFramebufferTexture2DEXT()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，渲染缓存图像将会从FBO的关联点分离。如果渲染缓存图像在依然关联在FBO上时被删除，那么纹理对象将会自动从当前绑定的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上，在执行FBO的操作之前，你必须检查FBO的状态，这可以通过调用glCheckFramebufferStatusEXT()实现。如果这个FBObuilding完整，那么任何绘制和读取命令（glBegin(),glCopyTexImage2D(), etc）都会失败。

```
GLenum glCheckFramebufferStatusEXT(GLenum target)
```

glCheckFramebufferStatusEXT()检查当前帧缓存的关联图像和帧缓存参数。这个函数不能在glBegin()/glEnd()之间调用。Target参数必须为GL_FRAMEBUFFER_EXT。它返回一个非零值。如果所有要求和准则都满足，它返回GL_FRAMEBUFFER_COMPLETE_EXT。否则，返回一个相关错误代码告诉我们哪条准则没有满足。

FBO完整性准则有：

（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, etc)。

（3）如果一幅被图像关联到GL_DEPTH_ATTACHMENT_EXT，那么这幅图像必须有深度可渲染的内部格式(GL_DEPTH_COMPONENT,GL_DEPTH_COMPONENT24_EXT, etc)。

（4）如果一幅被图像关联到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, etc)。

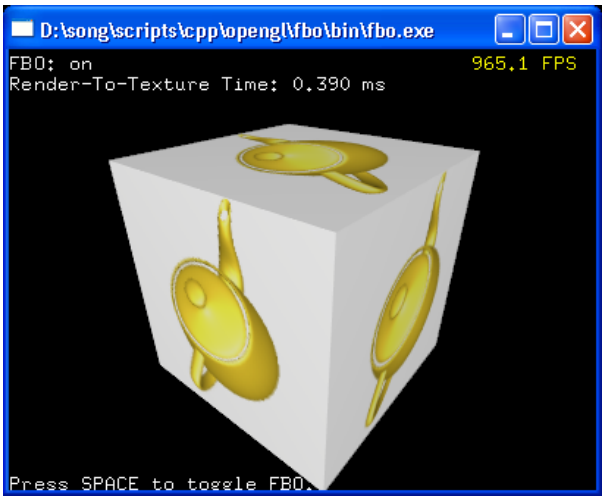
（5）FBO至少有一幅图像关联。

（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。

注意：即使以上所有条件都满足，你的OpenGL驱动也可能不支持某些格式和参数的组合。如果一种特别的实现不被OpenGL驱动支持，那么glCheckFramebufferStatusEXT()返回GL_FRAMEBUFFER_UNSUPPORTED_EXT。

示例：渲染到纹理



源代码下载：http://www.songho.ca/opengl/gl_fbo.html

包括渲染到纹理、只渲染到深度缓存和使用模板缓存渲染对象的轮廓。

有时候，你需要产生动态纹理。比较常见的例子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()拷贝帧缓存图像至纹理。

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）；。

这个demo实现了使用FBO和不使用FBO两种情况下渲染到纹理的操作，并且比较了性能差异。除了能够获得性能上的提升，使用FBO的还有另外一个优点。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被剪切掉。然后，使用FBO就不会有这个问题。你可以产生比显示窗口大的帧缓存渲染图像。

以下代码在渲染循环开始之前，对FBO和帧缓存关联图像进行了初始化。注意只有一幅纹理图像被关联到FBO，但是，一个深度渲染图像被关联到FBO的深度关联点。实际上我们并没有使用这个深度缓存，但是FBO本身需要它进行深度测试。如果我们不把这个深度可渲染的图像关联到FBO，那么由于缺少深度测试渲染输出结果是不正确的。如果在FBO渲染期间模板测试也是必要的，那么也需要把额外的渲染图像和GL_STENCIL_ATTACHMENT_EXT关联起来。

[cpp] view plain copy print ?

```
// create a texture object
GLuint textureId;
glGenTextures(1, &textureId);
glBindTexture(GL_TEXTURE_2D, textureId);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_GENERATE_MIPMAP, GL_TRUE); // automatic mipmap
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, TEXTURE_WIDTH, TEXTURE_HEIGHT, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, 0);
glBindTexture(GL_TEXTURE_2D, 0);

// create a renderbuffer object to store depth info
GLuint rboId;
glGenRenderbuffersEXT(1, &rboId);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, rboId);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT, GL_DEPTH_COMPONENT,
                         TEXTURE_WIDTH, TEXTURE_HEIGHT);
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, 0);

// create a framebuffer object
GLuint fboId;
glGenFramebuffersEXT(1, &fboId);
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);

// attach the texture to FBO color attachment point
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT, GL_COLOR_ATTACHMENT0_EXT,
                          GL_TEXTURE_2D, textureId, 0);

// attach the renderbuffer to depth attachment point
glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT, GL_DEPTH_ATTACHMENT_EXT,
                             GL_RENDERBUFFER_EXT, rboId);

// check FBO status
```

```
GLenum status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);
if(status != GL_FRAMEBUFFER_COMPLETE_EXT)
    fboUsed = false;

// switch back to window-system-provided framebuffer
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
...
```

渲染到纹理的过程和普通的绘制过程基本一样。我们只需要把渲染的目的地由window系统提供的帧缓存改成不可显示的应用程序创建的帧缓存（FBO）就可以了。

```
[cpp] view plain  copy  print  ?
...
// set rendering destination to FBO
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);

// clear buffers
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

// draw a scene to a texture directly
draw();

// unbind FBO
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

// trigger mipmaps generation explicitly
// NOTE: If GL_GENERATE_MIPMAP is set to GL_TRUE, then glCopyTexSubImage2D()
// triggers mipmap generation automatically. However, the texture attached
// onto a FBO should generate mipmaps manually via glGenerateMipmapEXT().
glBindTexture(GL_TEXTURE_2D, textureId);
glGenerateMipmapEXT(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, 0);
...
```

注意到，glGenerateMipmapEXT()也是作为FBO扩展的一部分，用来在改变了纹理图像的基级之后显式生成mipmap的。如果GL_GENERATE_MIPMAP被设置为GL_TRUE，那么glTex{Sub}Image2D()和glCopyTex{Sub}Image2D()将会启用自动mipmap生成（在OpenGL版本1.4或者更高版本中）。然后，当纹理基级被改变时，FBO操作不会自动产生mipmaps。因为FBO不会调用glCopyTex{Sub}Image2D()来修改纹理。因此，要产生mipmap，glGenerateMipmapEXT()必须被显示调用。

原网址：http://www.songho.ca/opengl/gl_fbo.html

英文原文：

In OpenGL rendering pipeline, the geometry data and textures are transformed and passed several tests, and then finally rendered onto a screen as 2D pixels. The final rendering destination of the OpenGL pipeline is called *framebuffer*. Framebuffer is a collection of 2D arrays or storages utilized by OpenGL; colour buffers, depth buffer, stencil buffer and accumulation buffer. By default, OpenGL uses the framebuffer as a rendering destination that is created and managed entirely by the window system. This default framebuffer is called *window-system-provided* framebuffer.

The OpenGL extension, **GL_EXT_framebuffer_object** provides an interface to create additional non-displayable framebuffer objects (FBO). This framebuffer is called *application-created*framebuffer in order to distinguish from the default *window-system-provided* framebuffer. By using framebuffer object (FBO), an OpenGL application can redirect the rendering output to the*application-created* framebuffer object (FBO) other than the traditional *window-system-provided*framebuffer. And, it is fully controlled by OpenGL.

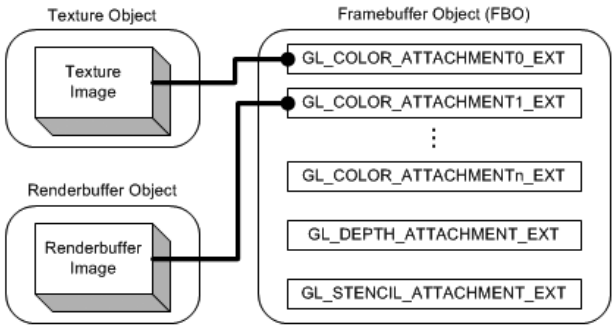
Similar to *window-system-provided* framebuffer, a FBO contains a collection of rendering destinations; color, depth and stencil buffer. *(Note that accumulation buffer is not defined in FBO.)*These logical buffers in a FBO are called *framebuffer-attachable images*, which are 2D arrays of pixels that can be attached to a framebuffer object.

There are two types of framebuffer-attachable images; texture images and renderbuffer images. If an image of a texture object is attached to a framebuffer, OpenGL performs *"render to texture"*. And if an image of a renderbuffer object is attached to a framebuffer, then OpenGL performs *"offscreen rendering"*.

By the way, renderbuffer object is a new type of storage object defined in

GL_EXT_framebuffer_object extension. It is used as a rendering destination for a single 2D image during rendering process.

The following diagram shows the connectivity among the framebuffer object, texture object and renderbuffer object. Multiple texture objects or renderbuffer objects can be attached to a framebuffer object through the attachment points。



There are multiple color attachment points (GL_COLOR_ATTACHMENT0_EXT,..., GL_COLOR_ATTACHMENT*n*_EXT), one depth attachment point (GL_DEPTH_ATTACHMENT_EXT), and one stencil attachment point (GL_STENCIL_ATTACHMENT_EXT) in a framebuffer object. The number of color attachment points is implementation dependent, but each FBO must have at least one color attachement point. You can query the maximum number of color attachement points with GL_MAX_COLOR_ATTACHMENTS_EXT, which are supported by a graphics card. The reason that a FBO has multiple color attachement points is to allow to render the color buffer to multiple destinations at the same time. This *"multiple render targets"* (MRT) can be accomplished by**GL_ARB_draw_buffers** extension. Notice that the framebuffer object itself does not have any image storage(array) in it, but, it has only multiple attachment points.The following diagram shows the connectivity among the framebuffer object, texture object and renderbuffer object. Multiple texture objects or renderbuffer objects can be attached to a framebuffer object through the attachment points.

Framebuffer object (FBO) provides an efficient switching mechanism; detach the previous framebuffer-attachable image from a FBO, and attach a new framebuffer-attachable image to the FBO. Switching framebuffer-attachable images is much faster than switching between FBOs. FBO provides **glFramebufferTexture2DEXT()** to switch 2D texture objects, and**glFramebufferRenderbufferEXT()** to switch renderbuffer objects.

Creating Frame Buffer Object (FBO)

Creating framebuffer objects is similar to generating vertex buffer objects (VBO).

glGenFramebuffersEXT()

```
void glGenFramebuffersEXT(GLsizei n, GLuint* ids)void
glDeleteFramebuffersEXT(GLsizei n, const GLuint* ids)
```

glGenFramebuffersEXT() requires 2 parameters; the first one is the number of framebuffers to create, and the second parameter is the pointer to a GLuint variable or an array to store a single ID or multiple IDs. It returns the IDs of unused framebuffer objects. ID 0 means the default framebuffer, which is the window-system-provided framebuffer.

And, FBO may be deleted by calling **glDeleteFramebuffersEXT()** when it is not used anymore.

glBindFramebufferEXT()

Once a FBO is created, it has to be bound before using it.

```
void glBindFramebufferEXT(GLenum target, GLuint id)
```

The first parameter, target, should be GL_FRAMEBUFFER_EXT, and the second parameter is the ID of a framebuffer object. Once a FBO is bound, all OpenGL operations affect onto the current bound framebuffer object. The object ID 0 is reserved for the default window-system provided framebuffer. Therefore, in order to unbind the current framebuffer (FBO), use ID 0 in glBindFramebufferEXT().

Renderbuffer Object

In addition, renderbuffer object is newly introduced for offscreen rendering. It allows to render a scene directly to a renderbuffer object, instead of rendering to a texture object. Renderbuffer is simply a data storage object containing a single image of a renderable internal format. It is used to store OpenGL logical buffers that do not have corresponding texture format, such as stencil or depth buffer.

glGenRenderbuffersEXT()

```
void glGenRenderbuffersEXT(GLsizei n, GLuint* ids)void
glDeleteRenderbuffersEXT(GLsizei n, const Gluint* ids)
```

Once a renderbuffer is created, it returns non-zero positive integer. ID 0 is reserved for OpenGL.

glBindRenderbufferEXT()

```
void glBindRenderbufferEXT(GLenum target, GLuint id)
```

Same as other OpenGL objects, you have to bind the current renderbuffer object before referencing it. The target parameter should be GL_RENDERBUFFER_EXT for renderbuffer object.

glRenderbufferStorageEXT()

```
void glRenderbufferStorageEXT(GLenum target, GLenum
internalFormat,                              GLsizei width,
GLsizei height)
```

When a renderbuffer object is created, it does not have any data storage, so we have to allocate a memory space for it. This can be done by using glRenderbufferStorageEXT(). The first parameter must be GL_RENDERBUFFER_EXT. The second parameter would be color-renderable (GL_RGB, GL_RGBA, etc.), depth-renderable (GL_DEPTH_COMPONENT), or stencil-renderable formats (GL_STENCIL_INDEX). The width and height are the dimension of the renderbuffer image in pixels.
The width and height should be less than GL_MAX_RENDERBUFFER_SIZE_EXT, otherwise, it generates GL_INVALID_VALUE error.

glGetRenderbufferParameterivEXT()

```
void glGetRenderbufferParameterivEXT(GLenum target, GLenum param,
GLint* value);
```

You also get various parameters of the currently bound renderbuffer object. *target* should be GL_RENDERBUFFER_EXT, and the second parameter is the name of parameter. The last is the pointer to an integer variable to store the returned value. The available names of the renderbuffer parameters are;

```
GL_RENDERBUFFER_WIDTH_EXTGL_RENDERBUFFER_HEIGHT_EXTGL_RENDERBUFFE
R_INTERNAL_FORMAT_EXTGL_RENDERBUFFER_RED_SIZE_EXTGL_RENDERBUFFER_
GREEN_SIZE_EXTGL_RENDERBUFFER_BLUE_SIZE_EXTGL_RENDERBUFFER_ALPHA_
SIZE_EXTGL_RENDERBUFFER_DEPTH_SIZE_EXTGL_RENDERBUFFER_STENCIL_SIZ
E_EXT
```

Attaching images to FBO

FBO itself does not have any image storage(buffer) in it. Instead, we must attach framebuffer-attachable images (texture or renderbuffer objects) to the FBO. This mechanism allows that FBO quickly switch (detach and attach) the framebuffer-attachable images in a FBO. It is much faster to switch framebuffer-attachable images than to switch between FBOs. And, it saves unnecessary data copies and memory consumption. For example, a texture can be attached to multiple FBOs, and its image storage can be shared by multiple FBOs.

Attaching a 2D texture image to FBO

```
glFramebufferTexture2DEXT(GLenum target,
GLenum attachmentPoint,                          GLenum
textureTarget,                          GLuint textureId,
GLint  level)
```

glFramebufferTexture2DEXT() is to attach a 2D texture image to a FBO. The first parameter must be GL_FRAMEBUFFER_EXT, and the second parameter is the attachment point where to connect the texture image. A FBO has multiple color attachment points (GL_COLOR_ATTACHMENT0_EXT, ..., GL_COLOR_ATTACHMENT*n*_EXT), GL_DEPTH_ATTACHMENT_EXT, and GL_STENCIL_ATTACHMENT_EXT. The third parameter,*"textureTarget"* is GL_TEXTURE_2D in most cases. The fourth parameter is the identifier of the texture object. The last parameter is the mipmap level of the texture to be attached.

If the *textureId* parameter is set to 0, then, the texture image will be detached from the FBO. If a texture object is deleted while it is still attached to a FBO, then, the texture image will be automatically detached from the currently bound FBO. However, if it is attached to multiple FBOs and deleted, then it will be detached from only the bound FBO, but will not be detached from any other un-bound FBOs.

Attaching a Renderbuffer image to FBO

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

A renderbuffer image can be attached by calling glFramebufferRenderbufferEXT(). The first and second parameters are same as glFramebufferTexture2DEXT(). The third parameter must be GL_RENDERBUFFER_EXT, and the last parameter is the ID of the renderbuffer object.

If *renderbufferId* parameter is set to 0, the renderbuffer image will be detached from the attachment point in the FBO. If a renderbuffer object is deleted while it is still attached in a FBO, then it will be automatically detached from the bound FBO. However, it will not be detached from any other non-bound FBOs.

Checking FBO Status

Once attachable images (textures and renderbuffers) are attached to a FBO and before

performing FBO operation, you must validate if the FBO status is complete or incomplete by using**glCheckFramebufferStatusEXT()**. If the FBO is not complete, then any drawing and reading command (glBegin(), glCopyTexImage2D(), etc) will be failed.

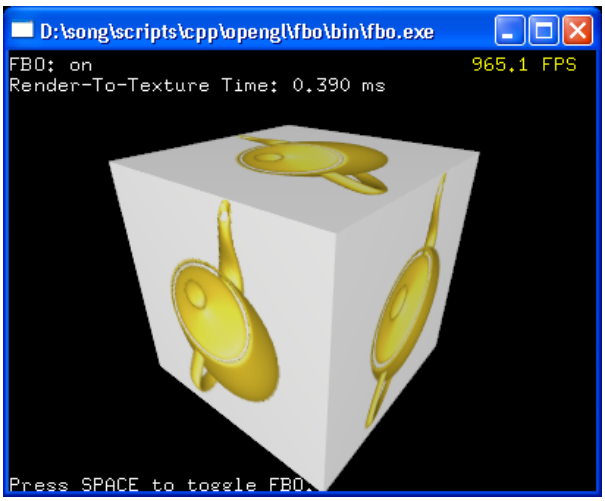
```
GLenum glCheckFramebufferStatusEXT(GLenum target)
```

glCheckFramebufferStatusEXT() validates all its attached images and framebuffer parameters on the currently bound FBO. And, this function cannot be called within glBegin()/glEnd() pair. The target parameter should be GL_FRAMEBUFFER_EXT. It returns non-zero value after checking the FBO. If all requirements and rules are satisfied, then it returns **GL_FRAMEBUFFER_COMPLETE_EXT**. Otherwise, it returns a relevant error value, which tells what rule is violated.
The rules of FBO completeness are:

- The width and height of framebuffer-attachable image must be not zero.
- If an image is attached to a color attachment point, then the image must have a color-renderable internal format. (GL_RGBA, GL_DEPTH_COMPONENT, GL_LUMINANCE, etc)
- If an image is attached to GL_DEPTH_ATTACHMENT_EXT, then the image must have a depth-renderable internal format. (GL_DEPTH_COMPONENT, GL_DEPTH_COMPONENT24_EXT, etc)
- If an image is attached to GL_STENCIL_ATTACHMENT_EXT, then the image must have a stencil-renderable internal format. (GL_STENCIL_INDEX, GL_STENCIL_INDEX8_EXT, etc)
- FBO must have at least one image attached.
- All images attached a FBO must have the same width and height.
- All images attached the color attachment points must have the same internal format.

Note that even though all of the above conditions are satisfied, your OpenGL driver may not support some combinations of internal formats and parameters. If a particular implementation is not supported by OpenGL driver, then glCheckFramebufferStatusEXT() returns GL_FRAMEBUFFER_UNSUPPORTED_EXT.

Example: Render To Texture



Download the source and binary: fbo.zip

Extras:

- Rendering to the depth buffer only: fboDepth.zip
- Rendering the outlines of an object using stencil buffer: fboStencil.zip

Sometimes, you need to generate dynamic textures on the fly. The most common examples are generating mirroring/reflection effects, dynamic cube/environment maps and shadow maps. Dynamic texturing can be accomplished by rendering the scene to a texture. A traditional way of render-to-texture is to draw a scene to the framebuffer as normal, and then copy the framebuffer image to a texture by using glCopyTexSubImage2D().

Using FBO, we can render a scene directly onto a texture, so we don't have to use the window-system-provided framebuffer at all. Further more, we can eliminate an additional data copy (from framebuffer to texture).

This demo program performs *render to texture* operation with/without FBO, and compares the performance difference. Other than performance gain, there is another advantage of using FBO. If the texture resolution is larger than the size of the rendering window in traditional render-to-texture mode (without FBO), then the area out of the window region will be clipped. However, FBO does not suffer from this clipping problem. You can create a framebuffer-renderable image larger than the display window.

The following codes is to setup a FBO and framebuffer-attachable images before the rendering loop is started. Note that not only a texture image is attached to the FBO, but also, a renderbuffer image is attached to the depth attachment point of the FBO. We do not actually use this depth buffer, however, the FBO itself needs it for depth test. If we don't attach this depth renderable image to the FBO, then the rendering output will be corrupted because of missing depth test. If stencil test is also required during FBO rendering, then additional renderbuffer image should be attached to GL_STENCIL_ATTACHMENT_EXT.

```
...// create a texture objectGLuint textureId;glGenTextures(1,
&textureId);glBindTexture(GL_TEXTURE_2D,
textureId);glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
GL_LINEAR);glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_LINEAR);glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_S,
GL_CLAMP_TO_EDGE);glTexParameterf(GL_TEXTURE_2D,
GL_TEXTURE_WRAP_T,
GL_CLAMP_TO_EDGE);glTexParameteri(GL TEXTURE_2D,
GL GENERATE_MIPMAP, GL_TRUE); // automatic
mipmapglTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, TEXTURE_WIDTH,
TEXTURE_HEIGHT, 0,             GL_RGBA, GL UNSIGNED BYTE,
0);glBindTexture(GL TEXTURE_2D, 0);// create a renderbuffer
object to store depth infoGLuint rboId;glGenRenderbuffersEXT(1,
&rboId);glBindRenderbufferEXT(GL_RENDERBUFFER_EXT,
rboId);glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
GL_DEPTH_COMPONENT,                          TEXTURE_WIDTH,
TEXTURE HEIGHT);glRenderbufferEXT(GL_RENDERBUFFER_EXT, 0);//
create a framebuffer objectGLuint fboId;glGenFramebuffersEXT(1,
&fboId);glBindFramebufferEXT(GL FRAMEBUFFER_EXT, fboId);// attach
the texture to FBO color attachment
pointglFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
GL_COLOR_ATTACHMENT0 EXT,                            GL TEXTURE_2D,
textureId, 0);// attach the renderbuffer to depth attachment
pointglFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
GL_DEPTH_ATTACHMENT_EXT,
GL_RENDERBUFFER_EXT, rboId);// check FBO statusGLenum status =
glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT);if(status !=
GL FRAMEBUFFER COMPLETE_EXT)    fboUsed = false;// switch back to
window-system-provided
framebufferglBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);...
```

The rendering procedure of render-to-texture is almost same as normal drawing. We only need to switch the rendering destination from the window-system-provided to the non-displayable, application-created framebuffer (FBO).

```
...// set rendering destination to
FBOglBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fboId);// clear
buffersglClear(GL COLOR BUFFER_BIT | GL DEPTH BUFFER_BIT);// draw
a scene to a texture directlydraw();// unbind
FBOglBindFramebufferEXT(GL FRAMEBUFFER EXT, 0);// trigger mipmaps
generation explicitly// NOTE: If GL GENERATE MIPMAP is set to
GL TRUE, then glCopyTexSubImage2D()// triggers mipmap generation
automatically. However, the texture attached// onto a FBO should
```

```
generate mipmaps manually via
glGenerateMipmapEXT().glBindTexture(GL_TEXTURE_2D,
textureId);glGenerateMipmapEXT(GL_TEXTURE_2D);glBindTexture(GL_TE
XTURE_2D, 0);...
```

Note that **glGenerateMipmapEXT()** is also included as part of FBO extension in order to generate mipmaps explicitly after modifying the base level texture image.
If **GL_GENERATE_MIPMAP** is set to GL_TRUE, then glTex{Sub}Image2D() and glCopyTex{Sub}Image2D() trigger automatic mipmap generation (in OpenGL version 1.4 or greater). However, FBO operation does not generate its mipmaps automatically when the base level texture is modified because FBO does not call glCopyTex{Sub}Image2D() to modify the texture. Therefore, glGenerateMipmapEXT() must be explicitly called for mipmap generation.
If you need to a post processing of the texture, it is possible to combine with Pixel Buffer Object (PBO) to modify the texture efficiently.

上一篇 【C++基础学习】引起类模板被实例化情形总结
下一篇 【GPU编程】基于GPU的光线投射体绘制(GPU-Based Ray-Casting Volume Rendering)入门学习