OpenGL常用命令备忘录(Part B)

2012-4-4 14:46:59 | **发布**:zwqxin

给一些容易忘记的opengl命令做做备忘录吧~想这么说的时候，突然想起貌似好久好久以前也在博客上说过类似的话……于是便记得有这么个小坑，坑得不成样子了。——ZwqXin.com

Part.A见此文：[OpenGL常用命令备忘录(Part A)]

可以稍微给个规则，那就是此系列相关的的API都会是自己觉得有一定“历史沉淀”的，但又可能会时常有机会用到。**备忘**为主。

3.glPixelStore

像glPixelStorei(GL_PACK_ALIGNMENT, 1)这样的调用，通常会用于像素传输(PACK/UNPACK)的场合。尤其是导入纹理(glTexImage2D)的时候：

c++代码

```
1.  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
2.
3.  glTexImage2D(,,,, &pixelData);
4.
5.  glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
```

很明显地，它是在改变某个状态量，然后再Restore回来。——为什么是状态？你难道8知道OpenGL就是以状态机不？——什么状态？其实名字已经很直白了，glPixelStore这组函数要改变的是像素的存储格式。

涉及到像素在CPU和GPU上的传输，那就有个**存储格式**的概念。在本地内存中端像素集合是什么格式？传输到GPU时又是什么格式？格式会是一样么？在glTexImage2D这个函数中，包含两个关于颜色格式的参数，一个是纹理（GPU端，也可以说server端）的，一个是像素数据（程序内存上，也就是client端）的，两者是不一定一样的，哪怕一样也无法代表GPU会像内存那样去存储。或者想象一下，从一张硬盘上的图片提取到内存的像素数据，上传给GPU成为一张纹理，这个“纹理”还会是原来的那种RGBARGBA的一个序列完事么？显然不是的。作为一张纹理，有其纹理ID、WRAP模式、插值模式，指定maip map时还会有一串各个Level下的map，等等。就纹理的数据来说，本质纹理是边长要满足2的n次方（power of two）的数据集合，这样首先大小上就有可能不一样，另外排列方式也未必就是RGBA的形式。在OpenGL的“解释”中，纹理就是一个“可以被采样的复杂的数据集合”，无论外面世界千变万化，GPU只认纹理作为自己“图像数据结构”，这体现着“规范化”这条世界纽带的伟大之处。

姑且把GPU里面的像素存储格式看做一个未知数，把该存储空间内那批像素看做一*堆X*。不要深究一*堆X*究竟是什么样子的，嘛，反正就想象成一堆软绵绵的，或者模糊不清的，打满马赛克的，(哔——)的一样

的东西就可以了。与此相比，内存中的像素数据实在太规则规范了！可能源文件各种图片格式，什么bmp、jpg、png甚至dds，但只要你按该格式的算法结构来提取（类似[Bmp文件的结构与基本操作(逐像素印屏版)]），总可以提取出一列整齐的RGBARGBA（或者RGBRGB什么的，反正很整齐就行了管他呢）的数据堆出来，是可以在程序中实测的东西。

涉及到像素在CPU和GPU上的传输，那就有个**传输方向**的概念。那就是大家耳濡目染的PACK和UNPACK。嘛，装载和卸载也可以，打包和解压也可以，随你怎么译了。结合上述存储格式的概念：

PACK —— 把像素从一*堆X*的状态转变到规则的状态（把一堆泥土装载进一个花盆，把散散的货物装上货柜，或者把一堆各样的文件打包成一个rar压缩包，等等）；
UNPACK —— 把像素从规则的状态转变到一*堆X*的状态（把花盆里的泥倒出来，把货柜中的货物卸载到盐田港，或者解压压缩包，等等）。

我认为这两个概念还是很容易混淆的，所以形象化一点总好点嘛。从本地内存向GPU的传输（UNPACK），包括各种glTexImage、glDrawPixel；从GPU到本地内存的传输（PACK），包括glGetTexImage、glReadPixel等。也正因如此，PBO也有PACK和UNPACK模式的区别。

好像说了好多不相关的事情。嘛，适当也当做延伸。回头来真正看一下glPixelStore吧。它的第一个参数，譬如ALIGNMENT、ROW_LENGTH、IMAGE_HEIGHT等等，都有PACK和UNPACK的两种版本，所以对应的也是上述关于PACH和UNPACK的两类函数。所以对于glTexImage2D，才使用GL_UNPACK_ALIGNMENT的版本。但要说明的是，无论是哪种传输方式，它都是针对本地内存端（client端）上的像素数据的。在上述例子中，它起着补充glTexImage2D中关于传输起点——本地像素集合的格式，的作用。

一般来说，这些本地的数据集合，只要知道其起始位置、大小(width*height)和颜色格式（譬如GL_RGBA等等）、值格式(GL_UNSIGNED_CHAR、GL_FLOAT等等)，就能准确地传输。而这些都是需要向glTexImage2D函数（或者上述的其他传输型函数）提供的。但是，这里头也一些细节，其实是需要glPixelStore这个函数来进行设置的。

3.1 GL_UNPACK_ALIGNMENT / GL_PACK_ALIGNMENT

通常，提取一张图像的时候，我们怎么知道一行的数据量呢？这个一行的数据量应该是：width * sizeof(Pixel)，应对最一般RGBA、各通道各占一个字节的像素结构，width * sizeof(Pixel) = width * 4 * sizeof(byte)，是4的整数倍。但是也有时候，我们的像素数据中一行的数据量不一定是4的整数倍（譬如一张RGB的图像、宽度150、各通道各占一个字节的像素结构，一行的数据量就是450个字节）。

另一方面，跟编译器一样，GPU传输时也喜欢4字节对齐，也即是说喜欢对像素数据按4字节存取。所以它更偏向于喜欢每一行的数据量是4的整数倍（按上述，这恰好是比较常见的）。所以为了更高的存取效率，OpenGL默认让像素数据按4字节4字节的方式传输向GPU——但是问题在于，对于行非4字节对齐的像素数据，第一行的最后一次存取的4字节将部分包括第一行的数据部分包括第二行的数据，当然致命的不是在这里，而是在最后一行：存取将很可能会越界。为了防止这样的情况，一是硬性把像素数据延展成4字节对齐的（就像BMP文件的存储方式一样，[Bmp文件的结构与基本操作(逐像素印屏版)]）；二是选择绝对会造成4字节对齐的颜色格式或值格式（GL_RGBA啦，或者GL_INT、GL_FLOAT之类）；三是以牺牲一些存取效率为代价，去更改OpenGL的字节对齐方式——这就是glPixelStore结合GL_UNPACK_ALIGNMENT / GL_PACK_ALIGNMENT。

c++代码

```
1.  glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
2.
3.  glTexImage2D(,,,, &pixelData);
4.
5.  glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
```

再次看回这段代码，这时候就明白了：让字节对齐从默认的4字节对齐改成1字节对齐（选择1的话，无论图片本身是怎样都是绝对不会出问题的，嘛，以效率的牺牲为代价），UNPACK像素数据，再把字节对齐

方式设置回默认的4字节对齐。至于哪种方式更适合，就看你依据硬件环境限制、麻烦程度等，去选择了。

3.2 GL_UNPACK_ROW_LENGTH/ GL_PACK_ROW_LENGTH 和 GL_UNPACK_SKIP_ROWS / GL_PACK_SKIP_ROWS 、 GL_UNPACK_SKIP_PIXELS/GL_PACK_SKIP_PIXELS

有的时候，我们把一些小图片拼凑进一张大图片内，这样使用大图片生成的纹理，一来可以使多个原本使用不同的图片作为纹理的同质物件如今能够在同一个Batch内，节省了一些状态切换的开销，二来也容易综合地降低了显存中纹理的总大小。但是，也有些时候，我们需要从原本一张大的图片中，截取图片当中的某一部分作为纹理。要能够做到这样，可以通过预先对图片进行裁剪或者在获得像素数据后，把其中需要的那一部分另外存储到一个Buffer内再交给glTexImage2D之类的函数。而上述这些参数下glPixelStore的使用将帮助我们更好地完成这个目的：

C++代码

```
1.    //原图中需要单独提取出来制成纹理的区域
2.  RECT subRect = {{100, 80}, {500, 400}}; //origin.x, origin.y, size.width, size.height
3.
4.  //假设原图的宽度为BaseWidth, 高度为BaseHeight
5.
6.  glPixelStorei(GL_UNPACK_ROW_LENGTH,  BaseWidth);     //指定像素数据中原图的宽度
7.  glPixelStorei(GL_UNPACK_SKIP_ROWS,       subRect. origin.y.); //指定纹理起点偏离原点的高度值
8.  glPixelStorei(GL_UNPACK_SKIP_PIXELS,     subRect. origin.x);  //指定纹理起点偏离原点的宽度值
9.
10. glTexImage2D(..., subRect.size.width, ubRect.size.height,.. &pixelData);  //使用区域的宽高
11.
12. glPixelStorei(GL_UNPACK_ROW_LENGTH, 0);
13. glPixelStorei(GL_UNPACK_SKIP_ROWS, 0);
14. glPixelStorei(GL_UNPACK_SKIP_PIXELS, 0);
```

这段代码本身，即使没有注释也很清楚了。注意的是GL_UNPACK_ROW_LENGTH的必要性，因为为了确认区域起点的Offset，就需要把线性数据pixelData上标记起点的“游标”从0移动到OffsetToData = subRect. origin.y * BaseWidth + subRect. origin.x的位置。有了区域纹理原点的在原图数据的位置，以及区域的尺寸，glTexImage2D就可以确定区域纹理生成所需要的信息了。通过glPixelStore的使用，避免了新建Buffer和自己处理图像数据的开销和麻烦了。

说到这里，到底为什么要这样做来提取区域纹理呢？尤其是原图若其他部分都是程序所需要的，那是不是就可以直接通过纹理坐标去切割更好呢？我想到的是一种情况（也可以说我是因为这种情况才注意到glPixelStore的这种用法）：如果这块区域纹理需要作重复铺设(wrap mode选择GL_REPEAT)呢？这时候纹理坐标的方法就没用了，因为REPEAT所依据的也是纹理坐标（使用纹理坐标的小数部分进行采样）。这时候就需要上述做法了。（事实上3DSMAX等软件纹理导入的类似区域纹理平铺的功能就能如此实现。）

4.glScissor

我想这个函数也应该很常见才对。裁剪测试啊，当年跟Alpha测试、Depth测试、Stencil测试可以并列哦，而今更是不掉时髦值啊。因为我实在很难想象在Shader里能容易地实现它的功能：裁剪。当然这只是矩形裁剪，但是对于discard掉渲染中不需要的像素真是颇简单粗暴。我使用它最多的是一些二维图片缩略图栏——有时候我们只需要把这些缩略图的显示限制在一个区域里，但又要支持滑动。

C++代码 （OpenGL）

```
1.          glEnable(GL_SCISSOR_TEST);
2.          glScissor(GLint(m_rtThumbRegion.x), GLint(m_rtThumbRegio
```

```
n.y), GLint(m_rtThumbRegion.width), GLint(m_rtThumbRegion.height));

        //.....  Render

         glDisable(GL_SCISSOR_TEST);
```

其中，除了启用GL_SCISSOR_TEST外，只要给glScissor指出需要保留显示的区域就可以了。在此区域外的像素依然会被渲染（不会怎么省流水线操作，所以也别指望它附带什么提高效率之类的功能），在下图中，其实左右两侧还是继续渲染其他的图片（或者说，其实这个缩略图栏横跨整个屏幕），但是就在fragment shader之后，它们会被检测到不在该区域内而被discard掉罢。

Tags: OpenGL 初学者 笔记

« 乱弹纪录I:Geometry Shader乱弹纪录II:Alpha To Coverage »

点击这里获取该日志的TrackBack引用地址

相关文章:

乱弹纪录I:Geometry Shader (2012-4-2 15:43:51)
shader复习与深入:HDR(高动态范围) (2012-1-26 14:50:43)
MD5模型的格式、导入与顶点蒙皮式骨骼动画I (2011-10-6 21:14:49)
球体贴图小谈 (2011-10-4 22:3:26)
AB是一家?VAO与VBO (2011-10-1 17:12:38)
多重采样(MultiSample)下的FBO反锯齿 (2011-9-25 16:55:35)
MD3模型的格式、导入与骨骼概念动画 (2011-3-12 10:2:23)
GimbalLock万向节锁与四元数旋转 (2011-3-4 22:54:50)
OpenGL/GLSL数据传递小记(2.x) (2011-2-16 20:45:40)
软阴影的实现尝试 II (2010-2-13 10:29:39)

1.hbkb

虽然说的乱七八糟 不过还是给楼主道声辛苦

zwqxin 于 2013-2-1 19:59:51 回复
呵呵，THX~

2013-2-1 8:33:59 回复该留言

2.sss

非常感谢，学到很多

2014-12-4 10:58:31 回复该留言

3.Nanhu2012

赞，正在写引擎的纹理部分，非常有价值

2014-12-21 20:02:00 回复该留言
