OpenGL/GLSL数据传递小记(3.x)

2012-9-9 13:51:16 | **发布**:zwqxin

OpenGL/GLSL规范在不断演进着，我们渐渐走进可编程管道的时代的同时，崭新的功能接口也让我们有点缭乱的感觉。本文再次从OpenGL和GLSL之间数据的传递这一点，记录和介绍基于OpenGL3.x的新方式，也会适时介绍Unform Buffer Objecct(UBO)这一重要特性。——ZwqXin.com

本文可视为大致一年半前的本博客的[OpenGL/GLSL数据传递小记(2.x)]一文的延续。对这方面不熟悉的话请先浏览一下该文中介绍的基本概念。在该文中，我把这些传递分为attribute变量、uniform变量、varying变量和Fragment Shader输出，这四部分（主要讲述前两部分）。而本文再次按此四部分，谈谈在GL3.x（NVidia 8Series以后显卡所支持的OpenGL版本）中的数据传递方式的变化。

1. attribute变量

在前文中提及到GLSL中每一个attribute变量都有一个“位置”值(Location)，在ShaderProgram链接(link)前，可以Bind之，链接之后，可以Get之。通过这两种方式都可以建立attribute变量与顶点属性的联系。如今引入第三种方式——直接在GLSL代码中指定这些位置值：

glsl3.x代码 (Vertex Program)

```
1.  #version 330
2.
3.  layout(location = 0) in vec3 attrib_position;
4.  layout(location = 1) in vec2 attrib_texcoord;
5.  layout(location = 2) in vec3 attrib_normal;
6.  layout(location = 3) in int  attrib_clustercount;
```

在上面的Vertex Program代码中，第一行（#version 330），表明我们现在使用的GLSL版本是GLSL 3.3，以区别于以前的版本并允许我们使用基于GLSL3.3的功能。在过去，OpenGL的版本和GLSL版本是不统一的（前文中的GL2.2所对应的是GLSL1.2，而后来的对应关系是GL3.0-GLSL1.3，GL3.1-GLSL 1.4，GL3.2-GLSL1.5），直到2010年OpenGL3.3/4.0规范的提出，khronos委员会决定让两者版本统一，所以就有了现在本博客所使用的OpenGL3.3-GLSL3.3的对应关系（注，ShaderModel4.0的显卡可达到的最高版本）。

接下来的几行声明了4个attribute变量。在GL2.x中一个attribute变量通常是“attribute vec3 attrib_position;”这样来表示，在GL3.x中，废弃了attribute关键字（以及varying关键字），属性变量统一用in/out作为前置关键字，对每一个Shader stage来说，in表示该属性是作为输入的属性，out表示该属性是用于输出的属性。这里，attribute变量作为Vertex Shader的顶点输入属性，所以都用in标记。另外，这里使用了layout关键字（通常是layout(layoutAttrib1=XXX, layoutAttrib2=XXX, ...)这样的形式）。这个关键字用于一个具体变量前，用于显式标明该变量的一些布局属性，这里就是显式设定了该attribute变量的位置值(location)，其作用跟ShaderProgram(着色程序)链接前调用glBindAttribLocation来设定atribut

e变量的位置值是等效的。

为什么采用这种方式更好呢？其一当然是编码量减少了，二来也避免了去Get某个attribute的location带来的开销，三来，最重要的是，它重定义了OpenGL和GLSL之间attribute变量属性的依赖。过去我们的OpenGL端必须首先要知道GLSL端某个attribute的名字，才能设置/获得其位置值，如今两者只需要location对应起来就可以完成绘制时顶点属性流的传递了。不再需要在ShaderProgram的compile和link之间插入代码也更方便于其模块化。

2.uniform变量

对于uniform变量的声明方式，跟GL2.x的一致，使用uniform关键字就可以了。

glsl代码

```
1.  #version 330
2.
3.  uniform sampler2D   basetex1;
4.  uniform float fAlphaRestrictVal;
5.  uniform mat4 matModel;
6.  uniform mat4 matView;
7.  uniform mat4 matProj;
```

每一个uniform变量也都有其一个“位置值”(Location)，在OpenGL中，我们可以通过glGetUniformLocation来获得。那么我们可以不可以像attribute变量那样，在Shader代码中显式指定这个Location呢？（其好处也是跟上述差不多的，但就是如果uniform变量太多的话这样做也麻烦，因为得在代码中一个一个指定不重复的location。）嘛，attribute变量location的显式指定，是经由GL扩展GL_ARB_explicit_attrib_location实现的，而事实上，现在也有GL_ARB_explicit_uniform_location这样一个GL扩展，能实现这样的功能，只不过它是OpenGL4.3标准的一部分，隶属于GLSL4.3，所以即使GL3.x支持这个扩展，我们还是暂时不要用的好。

那我们就像往常一样，在glUseProgram启用了某个ShaderProgram之后，一个一个地给每个unifom变量关联数据咯（通过其location）——等等，这是在运行期间设置数据值吧，那如果我这个关联数据并不是每帧都变化的，甚至它是一个固定值，这样做岂不太无聊太浪费了？事实上我们还是可以在glUseProgram之外绑定数据的——乃至直接在初始化时。这得益于*glProgramUniform*系列函数的引入，它比起往常的glUniform要多一个参数用来接收一个ShaderProgram的ID。在建立ShaderProgram后，我们也不需要glUseProgram来预先绑定它就可以直接取得某个uniform变量的location值并用glProgramUniform系列函数关联数据，而且这个数据在其后运行期间的每次glUseProgram后都不会失效。从理论上将，这族函数完全可以替代glUniform系列函数（是它们功能的一个超集），但是就不知道会不会有性能上的损失了（这个暂时目前找不到说法），所以我暂时建议是只对那些非动态变化的uniform变量使用了。

再来看看uniform变量的问题。通常一个稍微复杂点点、更多控制参数的Shader，都会有大量的Uniform变量需要设置，所以导致了我们很多时候在glUseProgram之后要调用一长串的glUniform函数来传递该Pass的数据。有没有方法尽量把这些操作合并呢？另外，我们知道一个Shader的可用Uniform数据大小是有一个上限值的（例如我目前显卡的一个vertex shader的GL_MAX_VERTEX_UNIFORM_COMPONENTS值是4096，意味着我在一个VertexShader里使用的active uniforms，大概就是最多4096个float/int值了，或者说最多1024个vec4、最多256个mat16），那么有没办法提高这个上限呢？在[MD5模型的格式、导入与顶点蒙皮式骨骼动画II]这篇文章中，因为担心uniform数量不足以支撑传入的众多个骨骼矩阵，所以优先选择TBO（Texture Buffer Object）作为传入数据的媒介，把数据装入一个一维纹理的Buffer中以提供给Shader。那么除了使用纹理数据外，还有没有更直接的方式呢？

Uniform Buffer Object(UBO)

UBO，顾名思义，就是一个装载Uniform变量数据的Buffer Object。就概念而言，它跟VBO([学一学，VBO])之类Buffer Object差不多，反正就是显存中一块用于储存特定数据的区域了。在OpenGL端，它的创建、更新、销毁的方式都与其他Buffer Object没什么区别，我们只不过把一个或多个uniform数据交给

它，以替代glUniform的方式传递数据而已。这里必须明确一点，这些数据是给到这个UBO，存储于这个UBO上，而不再是交给ShaderProgram，所以它们不会占用这个ShaderProgram自身的uniform存储空间，所以UBO是一种全新的传递数据的方式，从路径到目的地，都跟传统uniform变量的方式不一样。自然，对于这样的数据，在Shader中不能再使用上面代码中的方式来指涉了。随着UBO的引入，GLSL也引入了uniform block这种指涉工具。

glsl代码

```
#version 330

layout(std140) uniform matVP
{
   mat4 matProj;
   mat4 matView;
};
```

uniform block是Interface block的一种，（layout意义容后再述）在unifom关键字后直接跟随一个block name和大括号，里面是一个或多个uniform变量。一个uniform block可以指涉一个UBO的数据——我们要把block里的uniform变量与OpenGL里的数据建立关联。 因为这些uniform变量不是存储在Shader的“uniform区域”里的，所以也就没有那一套“位置值”(location)，那么我们通过什么建立关联呢?

对于每一个uniform block，都有一个“索引值”(index)，这个索引值我们可以在OpenGL中获得，并把它与一个具体的UBO关联起来。这样block内的数据声明就会与UBO中的实质数据联系起来了:

OpenGL代码

```
GLint nMatVPBlockIndex = glGetUniformBlockIndex(nProgramHandler, "matVP");

//GLint nMatVPBlockIndex = glGetProgramResourceIndex(nProgramHandler, GL_UNIFORM_BLOCK, "matVP");

if (GL_INVALID_INDEX != nMatVPBlockIndex)
{
    GLint nBlockDataSize = 0;

    glGetActiveUniformBlockiv(nProgramHandler, nMatVPBlockIndex, GL_UNIFORM_BLOCK_DATA_SIZE, &nBlockDataSize);

    glGenBuffers(1, &m_nUBO);

    glBindBuffer(GL_UNIFORM_BUFFER, m_nUBO);

    glBufferData(GL_UNIFORM_BUFFER, nBlockDataSize, NULL, GL_DYNAMIC_DRAW);

    glBindBufferRange(GL_UNIFORM_BUFFER, 0, m_nUBO, 0, nBlockDataSize);

    glUniformBlockBinding(nProgramHandler, nMatVPBlockIndex, 0);

    glBindBuffer(GL_UNIFORM_BUFFER, NULL);
}
```

一般我们可以使用glGetUniformBlockIndex来获取这个Index，但扩展GL_ARB_program_interface_query引入了比较统一的获取ShaderProgram内资源的相关属性的API（详见此扩展的spec），所以也可以以GL_UNIFORM_BLOCK调用glGetProgramResourceIndex来获取资源的Index。得到名为matVP的uniform block的Index后，我们可以查询这个block的相关信息(glGetActiveUniformBlockiv)。为了建立合适大小的UBO，这里查询了这个block所需的字节大小(GL_UNIFORM_BLOCK_DATA_SIZE)的值（注意这个值代表此block所占的大小，它可能会比block内数据实际相加后的值要大，下面会再述）。

建立一个UBO的过程跟建立其他类型的Buffer Object相似，不过Target是GL_UNIFORM_BUFFER，数据为空。接下来是把一个UBO（ID为m_nUBO）和Shader内的uniform block（Index为nMatVPBlockIndex）相关联：把它们都关联到同一个uniform buffer binding-point。其中前者通过glBindBufferBase或glBindBufferRange来完成，其中第二个参数就是binding-point，这里选择的是binding-point_0（参数值为0，当然你可以输入1、2、3...以选择binding-point_1、binding-point_2、binding-point_3...）；同样，对于后者uniform block，也通过glUniformBlockBinding来完成，其中第三个参数是binding-point，这里同样选择了第0个binding-point——这样OpenGL端的UBO和GLSL端的uniform block就联系在一起了。Shader中需要使用block中的uniform变量时，就会索引到对应的UBO中对应的位置的数据。

所谓binding-point（或者说binding-location），我理解为是OpenGL的Context上的一个个状态位。通常来说，我们可以建立非常多的UBO，它们的数据区在显存中，以ID标识，一般通过Context绑定一个UBO的ID的方式让OpenGL去寻找对应的显存位置——这是一种非常耗时的操作（应该说，所有bind类的操作都是）。数据需要更新就算了，但如果Shader执行时也必须为每个uniform block去绑定、寻觅数据区……为避免这样的情况所以就需要一个足以减少消耗的桥梁物，这个中间物件保存着能够直达具体某个UBO数据区的“方式”（不妨暂假想为该数据区的起始显存地址、长度等），然后我们把这个中间物件的位置告诉Shader，让Shader在需要时直接“来到”这个中间件中获取某个显存区的实质数据。这里与前者最大的区别应该就是Shader到中间件的用时——这应该足够快。所以首先这个中间物件应该存储在OpenGL的Context上（于是它名义上就是一个OpenGL状态），OpenGL内的对象的交流是比较便捷的，至少比Bind方式去存取“遥远的”显存数据要快不少，其次这个中间物件自身也应该容易表示，让Shader能“直接认门牌”——这些中间物件就是单纯Zero-Base数字序列形式的uniform binding-point，OpenGL通过它一步定位到实质数据处。

OpenGL Context本身也应该是一个尽量小体积的东西，所以不便在它身上放太多这种binding-point。在我的显卡上，GL_MAX_UNIFORM_BUFFER_BINDINGS的个数为36，这表示同一时间能映射的UBO-uniform block关系最多只有36对(间接也限制了一个ShaderProgram中uniform block的个数)，哪怕你有大量的UBO，为了以上机制的实行，也只能接受这个限制。我们就是通过glBindBufferBase/glBindBufferRange来我UBO或UBO中的某分区的信息存储至某个binding-point上，然后通过glUniformBlockBinding来“通知”ShaderProgram某个uniform block的数据信息存储在哪个binding-point上。如果把glUniformBlockBinding当成glUniform族函数，这个操作会更亲切一点：只不过如今对于目标block使用的是Index而不是Location（事实上它的行为更类似上面提到的glProgramUniform族函数，因为不需要事先glUseProgram启用某个ShaderProgram而是作为首参罢）。

除了UBO，前面某篇博文[乱弹纪录IV:Transform Feedback]中提到的Transform Feedback Buffer也是使用binding-point（参见文中代码段）的“好手”。因为Shader同样需要快速找出需要feedback的那个Buffer的所在地，尤其是通过GL_SEPARATE_ATTRIBS的方式为每一个输出数据独立指定buffer时，就需要用到多个transform-feedback binding-point来储存各个buffer的信息了。其限制个数其实就是GL_MAX_TRANSFORM_FEEDBACK_SEPARATE_ATTRIBS了（本显卡此数字为4）。

这里引出一个问题：我们能不能像TransformFeedback那样，为一个UBO对象非配数个binding-point呢？可以的。这样做的目的也很明确——单个UBO多个uniform-block。准确地说，是每个uniform block对应该UBO存储区域中不同的分区域(sub-region)——glBindBufferRange，就是你了！

OpenGL代码

```
1.  GLint nMatVPBlockIndex = glGetUniformBlockIndex(nProgramHandler, "matVP");
2.
3.  GLint nCloudScaleIndex = glGetUniformBlockIndex(nProgramHandler, "Scale");
4.
5.  if (GL_INVALID_INDEX != nMatVPBlockIndex && GL_INVALID_INDEX != nCloudScaleIndex)
6.  {
7.      int nUniformBufferAlignSize = 0;
8.
9.      glGetIntegerv(GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT, &nUniformBuffer
```

```
    AlignSize);
10.
11.     GLint nBlockDataSize1 = 0, nBlockDataSize2 = 0;
12.
13.     glGetActiveUniformBlockiv(nProgramHandler, nMatVPBlockIndex, GL_U
    NIFORM_BLOCK_DATA_SIZE, &nBlockDataSize1);
14.
15.     glGetActiveUniformBlockiv(nProgramHandler, nCloudScaleIndex, GL_U
    NIFORM_BLOCK_DATA_SIZE, &nBlockDataSize2);
16.
17.     glGenBuffers(1, &m_nUBO);
18.
19.     glBindBuffer(GL_UNIFORM_BUFFER, m_nUBO);
20.
21.     glBufferData(GL_UNIFORM_BUFFER, nUniformBufferAlignSize + nBlockD
    ataSize2, NULL, GL_DYNAMIC_DRAW);
22.
23.     glBindBufferRange(GL_UNIFORM_BUFFER, 0, m_nUBO, 0, nBlockDataSize
    1);
24.
25.     glUniformBlockBinding(nProgramHandler, nMatVPBlockIndex, 0);
26.
27.     glBindBufferRange(GL_UNIFORM_BUFFER, 1, m_nUBO, nUniformBufferAli
    gnSize, nBlockDataSize2);
28.
29.     glUniformBlockBinding(nProgramHandler, nCloudScaleIndex, 1);
30. }
```

上面代码段中，我们把两个uniform-block关联到同一个UBO的两个区域：[0 ~ nBlockDataSize1]、[nUniformBufferAlignSize ~ nUniformBufferAlignSize+nBlockDataSize2]。为什么第二个block不是映射到[nBlockDataSize1 ~ nBlockDataSize1+nBlockDataSize2]呢？这里有个比较重要的概念：数据对齐。对于uniform-block，可以通过GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT找出这个对齐值(本显卡上此数值是256字节，所以每个uniform block都是256字节对齐，相邻uniform block的间隔必须满足256字节的整数倍，否则会发现数据会对不上)。记住了，block与block的数据不是紧紧pack在一起的。很容易想象，跟CPU上存储结构体一样，这是为了数据存取效率考虑，至于为什么是这个值，就要更深入研究了。

麻烦可不止这一个——单个uniform-block里面的数据，也有字节对齐的机制——这给uniorm变量的数据更新带来更大的麻烦。

先来大致了解一下上面的GLSL代码的uniform block前面的layout内容。一般uniform block按数据组织类型可分为三种(目前)：*packed*、*shared*、*std140*，我们可以在它们前面用layout去指定该block属于哪种类型（也可以全局设置，也就是把layout单独作为一语句，此时它影响随后的各个没前接layout的uniform block）。

UBO的一个最显眼的好处就是实现数据共享。譬如我上面的matPV这个uniform block就是最好的例子：通常渲染场景时，只会有一个视图矩阵和一个投影矩阵（[乱弹OpenGL中的矩阵变换(上)] [乱弹OpenGL中的矩阵变换(下)]），而且它们相对每一帧都是固定数据。而我们可能场景里物件用到的Shader不一样，但它们都得通过这两个矩阵计算最终的顶点输出啊？以前的话，我得每个Shader都传一次这些相同的矩阵数据，不仅时间上glUniform族函数会比较多而且空间上也分别占每个ShaderProgram本身的同等的存储资源。如今把它们统一在一个UBO中，每帧更新就只要更新UBO一次就可以了，而且也只占一份的资源空间（在显存上）。

为什么突然插播以上“广告”呢？因为这对数据组织形式影响甚大。为了实现数据共享，必须保证各个shader里的指涉该UBO的unifom block“一模一样”。但我们也知道（[OpenGL/GLSL数据传递小记(2.x)]），GLSL编译器会检查并自动删掉那些非active（在shader中没有实质用途）的uniform变量。那么，假如我们的多个Shader里都有相同的uniform block，而里面某个变量x被ShaderA用到而没背ShaderB用到，那么前者就会把它默默删掉，这样数据结构不统一，自然映射到同一个UBO也无法预计得到各个子数据的具体位置（必须得针对每个Shader的uniform block内每个变量查询它的Offse

t）——block内的这种“检查”机制由*packed*这种layout掌控，为了要关闭这种机制，就需要选择其他三种layout。而*shared*（顺带一提，这个是默认layout）与std140不同之处在于，它虽然不会“删掉”block内的non-active变量，而且保证这些uniform block内的数据在存储分布上的一致性（所以各个shader能共享同一个block结构），但它不会去固定统一存储分布，所以还是有必要去查询各个变量的offset（因为可能在显卡A上这个offset是16在显卡B上就变32了）。至于std140等，其实就是排除这些因素而有着严格限制的一个数据组织结构“无优化”的版本，所以一般的场合下我们应该首选这类std(OpenGL-Standard)的layout。顺带说一下因为最初UBO/uniform block是跟随OpenGL3.1/GLSL1.4引入的所以有此std140之名（其实现存的类似layout还有个std430，但它是专门留给OpenGL/GLSL4.3的storage buffer block产生更小的offset而用的，按此不表）。

说了那么多，既然一般应用首选std140，那么它那个固定的offset是多少呢？根据我的不严格查验（没验证多个显卡），其值为16字节，也就是说数据按16字节对齐。而数据中还再分为vector、数组、矩阵这些，也是按类似规则限制（不一一举出，查spec去吧）。举例一下吧：

glsl代码

```
layout(std140) uniform matVP
{
   float elapsedTime;
   mat4 matProj;
   mat4 matView;
};
layout(std140) uniform Scale
{
  uniform float cloudScale;
};
```

要更新这两个block对应的那个UBO，应该这样：

C++代码

```
{
//Render
    glBindBuffer(GL_UNIFORM_BUFFER, m_nUBO);
    glBufferSubData(GL_UNIFORM_BUFFER, 0, sizeof(GLfloat), &fElapsed);

    //current std140 alignment(for base sclaer): 16
    glBufferSubData(GL_UNIFORM_BUFFER, 16, sizeof(Matrix16), m_mtProj.mt);
    glBufferSubData(GL_UNIFORM_BUFFER, 16 + sizeof(Matrix16), sizeof(Matrix16), m_mtView.mt);

    //GL_UNIFORM_BUFFER_OFFSET_ALIGNMENT: 256
    glBufferSubData(GL_UNIFORM_BUFFER, 256, sizeof(GLfloat), &fCScale);
    glBindBuffer(GL_UNIFORM_BUFFER, NULL);
}
```

其实UBO除了能够共享unifom变量数据外，上面的叙述还隐含有它的两个重要优点：一点是索引、切换binding-point的速度比较快，比起多个glUniform的调用传递数据也更快；另一点是对于显存中的uniform数据，可用的存储空间也大幅增加（而且对比TBO，UBO更适合需要线性存取的数据）——这回应了介绍UBO前的那两个问题。

UBO的介绍暂到此为止，真的很费口水——因为我觉得它本身就是包含不少要点的OpenGL功能——其实要点还不止这些，也还是需要更进一层地了解才行。uniform数据应尽量使用UBO来存放，尤其是那些需要Shader共享的数据，当然了零碎细小的数据还是glUniform/glProgramUniform类函数会更方便点吧~

3.varying变量

varying变量主要用于在Shader Stage间进行传递，注意的是在光栅化(Rasterization)的时候，这些变量

也会跟着一起被光栅插值。那如果我们不想某个顶点属性被光栅化，该怎么办呢？在[OpenGL常用命令备忘录(Part A)]这篇文章提到的一个古老API，glShadeModel，它在固定管道渲染流水线上能起到控制图元属性是否被插值的功效（需要光栅化时传入参数GL_SMOOTH，不需要时传入GL_FLAT），那么当选择不插值时（GL_FLAT），流水线上发生了什么呢？

假设现在流水线上，经过裁剪、归一化等，生成了一个屏幕上的三角图元（三个顶点上的颜色属性分别是c1、c2、c3），进入光栅化阶段。假如进行插值，三角图元里各像（假设共n个）素会根据其各自位置对三个顶点的颜色值进行线性插值，生成对应的n个颜色值（cList[n]）；假如不插值，则该三角形里所有像素都会是同一个值（cConst），这个值可能等于c1、c2或c3其中一个。到底是哪一个呢？这取决于哪个顶点是provoking-vertex（在[乱弹纪录I:Geometry Shader] 中也提及过它）。你可以在OpenGL端通过glProvokingVertex函数改变这个设置（参数GL_FIRST_VERTEX_CONVENTION/GL_LAST_VERTEX_CONVENTION决定取图元绘制顺序的第一个顶点还是最后一个顶点作为provoking-vertex）。

其实要让GLSL中某个作为顶点属性的varying变量不被光栅化，只要在它前面加一个flat关键字就可以了。这样它就像上述的那样，到达Fragmen Shader的图元上所有像素的该varying值都是相同的值（provoking vertex上的值）：

glsl代码

```
1.  //vertex shader or geometry shadr
2.  flat out float visibility;
3.
4.  //fragment shader
5.  flat in float visibility;
```

同样在[乱弹纪录I:Geometry Shader]中也提到这样一个问题：一个ShaderProgram中不能有两个同为输入的同名varing变量，也不能有两个同为输出的同名varing变量存在。所以即使表示的是同一个变量，也得使其名字不一样：

glsl代码

```
1.  //Vertex Shader
2.  out vec2 varying_vg_texcoord;
3.
4.  //Geometry Shader
5.  in  vec2 varying_vg_texcoord[];
6.  out vec2 varying_gf_texcoord;
7.
8.  //Fragment Shader
9.  in vec2 varying_gf_texcoord;
```

这样的话，在有些场合需要实现不同shader的组合——譬如实现一个可加入也可不加入的Geometry Shader，就难办了（何况当代流水线上的Shader可不止这三个呢）。为解决这个麻烦，也为了把变量声明组织得更“好看”一些，我们再次用到interface block。上面的uniform block是其中一种，但它还包括in block和out block这两种可用于varing变量的：

C++代码

```
1.  //Vertex Shader
2.  out Varying
3.  {
4.       vec2 texcoord;
5.  }VaryingOut;
6.
7.  //Geometry Shader
8.  in Varying
9.  {
10.      vec2 texcoord;
11. }VaryingIn[];
12.
13. out Varying
```

```
{
    vec2 texcoord;
}VaryingOut;

//Fragment Shader
in Varying
{
    vec2  texcoord;
}VaryingIn;
```

注意这里使用了block insatnce name（紧随大括号后的那个名字），这个名字对各Shader Stage来说都是独特的，所以改成上面这样的话，block之间也不会发生名字冲突，block内的varying变量也就可以用同一个名字了。使用时需要按"blockInstanceName.varyingVariable"的类似结构体内变量的样式来表示：

glsl代码

```
//Geometry Shader Example
void main(void)
{
   for(int i = 0; i < gl_in.length(); ++i)
   {
        gl_Position = gl_in[i].gl_Position;

        VaryingOut.texcoord = VaryingIn[i].texcoord;

    EmitVertex();
   }
   EndPrimitive();
}
```

block自带组织多个变量声明的功效：

glsl代码

```
//Geometry Shader Example
out Varying
{
    vec3 position;
    int  cloudcount;
    vec3 dimension;
}VaryingOut;
```

另外，对于Transform Feedback（[乱弹纪录IV:Transform Feedback]），指定输出Varing属性时，也要按上述的结构体内变量表示法：

C++代码

```
//OpenGL Code Example
const GLchar *varyingOutCloudFeed[] = {"Varying.position", "Varying.cloudcount", "Varying.dimension"};
glTransformFeedbackVaryings(m_CloudFeedShader.GetProgramHandler(), 3, varyingOutCloudFeed, GL_SEPARATE_ATTRIBS);
m_CloudFeedShader.Link();
```

4.fragmentShader输出

最后，再简单谈一下fragmentShader的输出。一般来说，输出的是颜色值，输出目标是Frame Buffer。这又包括常规的输出到屏幕Buffer、输出到FBO（[学一学，FBO]），另外还可以通过MRT（Multi Render Target）输出到两个以上的FBO中。但是，这些对于Fragment Shader来说并没太多不一样：通过ShaderProgram链接前的glBindFragDataLocation指定输出到第几个Buffer（默认是0）。类似于上述的attribute变量，我们也可以直接通过layout来指定这个location值：

glsl代码 (fragemnt shader)

```
1.  //单输出
2.  layout(location = 0) out vec4 fragColor;
3.
4.  //MRT
5.  layout(location = 0) out vec4 fragColor0;
6.  layout(location = 1) out vec4 fragColor1;
7.  ...
```

然后只要在FragmentShader中把结果对应地赋给这些输出型变量就可以了。但是，这些layout里的关键字其实还有个index——只是默认为0而已：

glsl代码 (fragemnt shader)

```
1.  layout(location = 0, index = 0) out vec4 fragColor;
2.  layout(location = 0, index = 1) out vec4 src1Color;
```

它们同样是输出到第0个缓冲区，但是其中有一个的index为1——这个src1Color是所谓的Second Output。它同样储存在一块缓冲区域中，但我们在OpenGL中怎么获得这个区域的颜色值呢？答案就是由GL_ARB_blend_func_extended扩展引入的，新的混合参数（GL_SRC1_COLOR/GL_SRC1_ALPHA等等这类新旧的enum）。它们作为混合因子而存在——这里输出的src1Color，就只能作为各个对应像素混合因子来用。简单举例：

C++代码

```
1.  glEnable(GL_BLEND);
2.  glBlendFunc(GL_SRC_ALPHA, GL_SRC1_COLOR);
```

该代码启用混合，当前绘制的内容（混合源src，即fragColor）的混合因子是自己的alpha值，而背景（混合目标dst，即绘制前此FrameBuffer的内容）处对应的被覆盖像素的混合因子则是该对应像素输出的src1Color值，其中RGBA分量分别用于混合RGBA四个通道：

$$finalColor = sourceColor * sourceAlpha + destinationColor * src1Color$$

好了，本文于此结束。如有批误或疏忽提醒，请大牛们不腻赐教或指出给ZwqXin，谢谢。

Tags: GLSL OpenGL 笔记 shader

分类:Shader技术 | 评论:4 | 引用:0 | 浏览:9933

« 地形渲染II - Continuity of Terrain Lod FBX、DAE模型的格式、导入与骨骼动画 »