OpenGL/GLSL数据传递小记(2.x)

2011-2-16 20:45:40 | **发布**:zwqxin

本篇记录一下关于OpenGL程序中绑定各种GLSL变量的一些注意问题（有些是近期编写代码感受强烈的）。以供参考。——ZwqXin.com

首先回顾一下shader程式的装载（包括vertexShader、geometryShader、fragmentShader，近期新提出的tessellation相关的Shaders没使用过显卡也没支持）：

1. shader代码可以用文本文件保存，可以在程序中用字符串保存，但最终还是必须在程序中以字符串的形式传入Shader对象中；
2. 创建Shader对象（glCreateShader——成功时返回非0的无符号Handle值，指涉Shader对象）；
3. 把shader代码传入shader对象（glShaderSource——注意此函数的参数，字符流地址参量是GLchar*，不支持宽字符。执行后可删除内存上保存的shader代码字符串副本）；
4. 编译Shader对象——正如我们编译任何代码一样（glCompileShader——应该以GL_COMPILE_STATUS为参调用glGetShaderiv检查编译是否成功。如果代码出现问题会在这个阶段报错，debug时可用glGetError查看更具体的错误类型）；
5. 按上步骤把一个“过程”中需要的各类型shader编译好（通常每种类型最多建一个shader对象——因为对于一个渲染管道（流程）来说，顶点、单几何体、像素都只处理一遍，不可能返回。shaders只是插入这个流程中取代对应的固定管道处理的“外挂”[在抹除固定管道处理、全shader时代来临之前可以这么说]，在一个渲染流程中起作用的shaders姑且统称为一个shader过程）；
6. OpenGL通过一个名为shaderProgram的对象来与shader交互。也可以说shaders通过这个对象连接到我们的OpenGL应用程序中，它具体地指涉shader过程。宏观概念上类似于我们平时写的“程序”，不过它是基于GPU的；
7. 创建这么一个shaderProgram对象（glCreateProgram——成功时返回非0的无符号Handle值，指涉Shader程序对象）；
8. 把之前创建的shaders，一个一个地Attach到这个shaderProgram对象上（glAttachShader——当然理论上可以attach多于一个的同类型完整的shader，譬如vertexShader。但是在一个特定的渲染流程中只允许其中一个起作用，你明白的）；
9. 链接shaderProgram——正如我们链接任何程序一样（glLinkProgram——应该以GL_LINK_STATUS为参调用glGetProgramiv检查链接是否成功。）
10. 至此一个shader程式装载完毕。在进入一批渲染流程前（即渲染一组物件前）启用这个shaderProgram对象（glUseProgram），它就会在这批渲染流程中起作用了。渲染完后可以（也应该）调用glUseProgram(NULL)来关闭这种介入，或者以其他shaderProgram渲染别的物件。

对以上有疑问或想知道更具体的同学可参考opengl的Document。希望发现bug或者不妥之处的同学在鄙人博客(www.zwqxin.com)指出。

开始正题了。

在openGL3.0+中，shader里的输入输出变量以in/out这种看上去不甚优雅的限定符标识。先不管，我们下面沿用2.0的限定符，说说在shader中传递我们的变量的时候的一些点：

1. attribute变量

一般是指vertex attribute（顶点属性）——每个顶点都有一份。在vertexShader中，我们处理的是每个顶点，而我们希望传入的变量时每个顶点各异的时候，就使用这种变量（在shader中以attribute为限定符），它不必是传统意义上的“顶点属性”（顶点位置、法线之类），但它确实又是一种顶点的“属性”。OpenGL3.0之前（或者说，固定管道被严重BS之前），我们可以很舒心地使用gl_Vertex, gl_MultiTexcoord[], gl_Normal这类内置的attribute变量来指涉传入shader里的传统的顶点属性，但如今其实我们最好习惯于“没有你们的日子”（因为被BS了）。

这种变量需要在GPU里的Shader的存储空间中有固定的位置（地址）。在链接shaderProgram（见上文）之前，这个位置是未确定的，因此我们可以在这个shaderProgram调用glLinkProgram之前，为这个attribute变量指定一个位置（用无符号值表示）：glBindAttribLocation：

```
//为shader中的attribute变量attribName绑定到一个位置
GLuint nHandle = glCreateProgram()

glAttachShader(nHandle, nShaderHandle1);
....

glBindAttribLocation(nHandle, 2,  "attribName");

glLinkProgram(nHandle);
```

在我的ZWGLSL类中，封装后提供给上层的shader装载API是setXXShader（建立各类型shader对象）和Load（类似上面的代码，建立shaderProgram，attach，bind和link）。上层怎么指定一个attribute的位置呢？当然可以把glLinkProgram独立起来——不过这样就拆开了上面的代码，上层需要多个调用，个人不喜欢。于是便在类内提供一个map容器：

```
//Load函数大致：
GLuint nHandle = glCreateProgram()
glAttachShader(nHandle, nShaderHandle1);
....
for(std::map<GLuint, const GLchar *>::iterator p = m_LocAttribMap.begin();p != m_LocAttribMap.end(); ++p)
{
    glBindAttribLocation(m_nShaderProgram, p->first, p->second);
}
glLinkProgram(nHandle);


//BindAttributeLocation函数：
void ZWGLSL::BindAttributeLocation(const GLchar *AttributeName, GLuint nLocation)
{
    m_LocAttribMap.insert(std::pair<GLuint, const GLchar *>(nLocation, AttributeName));
}


//外部调用一个shader的装载代码（需要绑定一个attribute变量的位置的情形）：
    m_ZWShader.SetVertexShader("XXX.vert");
```

```
21.  m_ZWShader.SetFragmentShader("XXX.frag");
22.  m_ZWShader.BindAttributeLocation("attribName",  2);
23.  m_ZWShader.Load();
```

我们利用这个“位置”（上面为2）来指定需要传给shader里的attribue变量的数据（见下文）。

另一种获得这个“位置”的方法：我们不要去显式设定这个位置，而是去获取它。通常，如果shader里有attribute变量，且我们没有为它绑定一个位置（见1上文），那么shaderProgram在链接后，会自动为它分配一个位置。我们可以在任何需要的时候获取（查询）这个位置：glGetAttribLocation，就不必局限于在shaderProgram的创建和链接之间去绑定了：

```
1. //GetAttributeLocation函数：
2.  GLint ZWGLSL::GetAttributeLocation(const GLchar *AttributeName)
3. {
4.     return glGetAttribLocation(m_nShaderProgram, AttributeName);
5. }
6.
7. //外部调用一个shader的装载代码：
8.     m_ZWShader.SetVertexShader("XXX.vert");
9.     m_ZWShader.SetFragmentShader("XXX.frag");
10.    m_ZWShader.Load();
11.
12.    ....
13.    //获取一个attribute变量的位置
14.    GLint  nAttribLoc = m_ZWShader.GetAttributeLocation("attribName");
15.
16.    if(-1 != nAttribLoc )
17.    {
18.        //使用
19.     }
```

这里返回一个有符号的int值，因为当要查询的这个变量在shader中不存在，或者它没有作用（非活动的：non-active），就会返回-1，否则才是它的位置。（当我们在shader里定义了一个变量，但是代码里却没见它有什么作为，就说它是非活动的。glGetActiveAttrib可返回那些活动的attributes。）

使用atribute变量的“位置”为它传递数据：

a）传统的glVertex3f类逐点绘制下（使用glVertexAttrib3f函数，以“位置”为首参）：

```
1. //nAttribLoc是获得的一个vec3的attribute变量的位置
2.     glBegin(GL_QUADS);
3.     glNormal3f(vNormal.x, vNormal.y, vNormal.z);
4.     glVertexAttrib3f(nAttribLoc, vAttribData.x, vAttribData.y, vAttribData.z);
5.     glTexCoord2d(0.0, 0.0);
6.     glVertex3d(pt1.x, pt1.y, pt1.z);
7.
8.     glNormal3f(vNormal.x, vNormal.y, vNormal.z);
9.     glVertexAttrib3f(nAttribLoc, vAttribData.x, vAttribData.y, vAttribData.z);
10.    glTexCoord2d(0.0,  1.0);
11.    glVertex3d(pt2.x, pt2.y, pt2.z);
12.
13.        .......
14.    glEnd();
```

b）VBO【学一学，VBO】下：

```
1. //nAttribLoc是获得的一个vec3的attribute变量的位置
2.
3.    glBindBuffer(GL_ARRAY_BUFFER，nPosVBO);
4.    glEnableClientState(GL_VERTEX_ARRAY);
5.    glVertexPointer(3, GL_FLOAT, 0, NULL);
6.
7.    glBindBuffer(GL_ARRAY_BUFFER, nNormVBO);
8.    glEnableClientState(GL_NORMAL_ARRAY);
9.    glNormalPointer(GL_FLOAT, 0, NULL);
10.
11.   glBindBuffer(GL_ARRAY_BUFFER, nTexcoordVBO);
12.   glEnableClientState(GL_TEXTURE_COORD_ARRAY);
13.   glTexCoordPointer(2, GL_FLOAT, 0, NULL);
14.
15.   glBindBuffer(GL_ARRAY_BUFFER, nAttrbDataVBO);
16.   glEnableVertexAttribArray(nAttribLoc);
17.   glVertexAttribPointer(nAttribLoc, 3, GL_FLOAT, GL_FALSE, 0, NULL);
18.
19.   glDrawElements/glDrawArrays
20.
21.   glDisableVertexAttribArray(nAttribLoc);
22.   glDisableClientState(GL_TEXTURE_COORD_ARRAY);
23.   glDisableClientState(GL_NORMAL_ARRAY);
24.   glDisableClientState(GL_VERTEX_ARRAY);
25.   glBindBuffer(GL_ARRAY_BUFFER, NULL);
```

有必要说明一下，在VBO中，glEnableClientState/glVertexPointer(glNormalPointer,glTexCoordPointer等)可以正确地为shader中内置的gl_Vertex, gl_Normal, gl_MultiTexcoord[]这些attribute变量指定VBO中的数据，但是这些传统顶点属性之外的就需要glEnableVertexAttribArray/glVertexAttribPointer了（事实上在日后不提倡用内置attribute变量后，这些传统的顶点属性也得被一视同仁地用这两个函数指定启用和数据格式）。参数除了特别的首参需要一个“位置”外，基本是一样的（中间多了个不常用的是否normalize数据的参）。

注意，每个顶点属性的数据都依托在一个VBO中了。要想给一个attrbute变量传递数据，请把数据交给一个VBO对象。

2.uniform变量

uniform变量的特点是，对于一个vertexShader内处理的每个顶点（或者fragmentShader内处理的每个像素，诸此类推），它都是不变的。事实上它相当于一个全局量，并非每个顶点/几何/像素所雍有的变量（只是uniform对它们每一个都public而已）。当然了，你想在一个渲染流程中改变它的值也是8可能的。Shader中有gl_ModelViewMatrix等一大批内置的uniform变量（当然在OpenGL3.0/GLSL1.3后它们也被BS了）。

我们通常在shaderProgram链接后获取一个uniform变量的位置，然后向这个位置传送数据（glGetUniformLocation/glUniform）。但是要特别注意的一点是：我们只有在启用了一个shaderProgram后，才能做这样的事情：

```
1. //SendUniform函数，其中一个重载版本
2. void ZWGLSL::SendUniform(const GLchar *UniformName, GLfloat x)
3. {
```

```
    GLint location = glGetUniformLocation(m_nShaderProgram,UniformName);
    glUniform1f(location, x);
}

inline void Enable() { glUseProgram(m_nShaderProgram); };
inline void Disable(){ glUseProgram(0); };

//传送数据, m_UniformData为一个float数据:
ZWShader.Enable();
ZWShader.SendUniform("uniformName",  m_UniformData);
//Render Something
ZWShader.Disable();
```

可能是不同的shaderProgram都有一套用于分配的“位置”吧，为了不混淆就so了（擦，怎么想起了相对内存地址来了--）

有些时候，我们身不由己——我们给一个渲染对象类关联一个shader相关类的指针，shaderProgram启用与否完全交给这个渲染对象类——我们还是得在上层为shader指定uniform数据。这时候，可以在shader类指针之外，再关联一个map<uniform位置，uniform数据>（当然了这个“数据”还得根据uniform变量类型来细分），我们只把数据传给这个map。当shader在渲染对象类里头被启用之后，立即就把这些数据都通过glUniform传送。——这其实是面向对象设计要关心的内容了。

是不是可以这样呢?

```
//not that good!

//somewhere:
glUseProgram(nHandle);
...//send uniform
glUniform(data used in rendering)...
glUseProgram(nHandle);

//somewhere else
glUseProgram(nHandle);
...//Render Something
glUseProgram(nHandle);
```

坐等可能出现的意想不到的悲剧吧。

3.varying变量

这个就是shader之间传递的变量类型了。不是本文关心的。当然还是要提醒一下，在这边的shader里的varying输出，在那边的varying输入就可能是被栅格化了（说通俗点，被插值了）。

4.fragmentShader输出

其实主要是想带出另一个函数：glBindFragDataLocation。其实目前来说，它在数据传递中的作用，其实只是指定了fragmentShader最终的像素颜色信息所要输出的BUFFER（GL_DRAW_BUFFER0 /GL_DRAW_BUFFER1）。这个是OpenGL3.0/GLSL1.3要鄙视我们熟悉的（gl_FragColor/gl_FragData[]）用的——把shader里定义的一个输出的out型变量（用以输出最终像素颜色）绑定给一个输出Buffer。

好了，本文到此。有什么其他的，想到再补充了。也欢迎各位的意见——这才是最重要的。