GPU GLSL Programming (2)

GLSL着色语言

OpenGL Shading Language (GLSL)

- OpenGL内置的，专用于编写着色器程序的语言
  - 类C语言
  - 着色器程序的源代码可以直接写在C++程序中，用字符串常量表示
- 引进新的数据类型
  - 矩阵
  - 向量
  - 采样器（Samplers）
- OpenGL的状态通过内置变量传递

两个主要的着色工作

- 顶点着色器：将3D顶点映射到2D片元（屏幕位置）
- 片元着色器：决定2D片元的最终呈现的颜色

顶点（片元）着色器是每个顶点（片元）
都需要（并行）运行一遍的程序

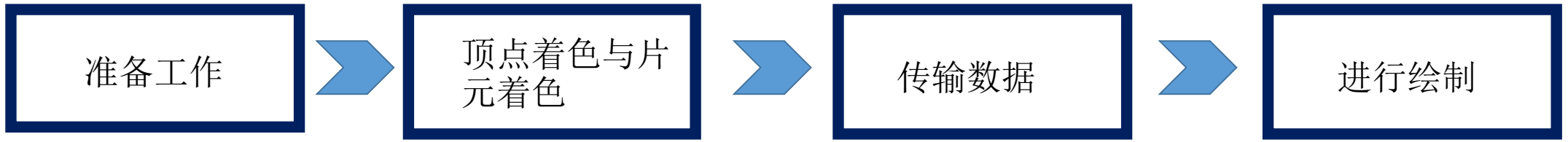
3.可编程管线的工作流程

可编程管线的工作流程

- 渲染管线只是一套标准化的流程
- 按照公认的划分方式，一套完整的可编程管线分为10个步骤（出自《OpenGL编程指南》），但很多步骤不是必须的
- 我们主要学习四个部分：准备工作、顶点着色与片元着色、传输数据、绘制阶段

可编程管线的工作流程

- 一套简单的可编程管线的流程如下图所示：
  - 1.进行准备工作。
  - 2.编写顶点着色与片元着色代码，生成可执行的GPU程序。
  - 3.向GPU程序中传输数据。
  - 4.运行你的程序，绘制图像。



下面以一个小的工程ShaderTest（见src目录”ShaderTest.rar”）来说明

3.1准备工作

3.1准备工作——配置库

- glew与glut
  - glew: OpenGL扩展库
  - glut: 提供OpenGL中的窗口，鼠标交互等支持

- 需要在代码中进行库的初始化工作

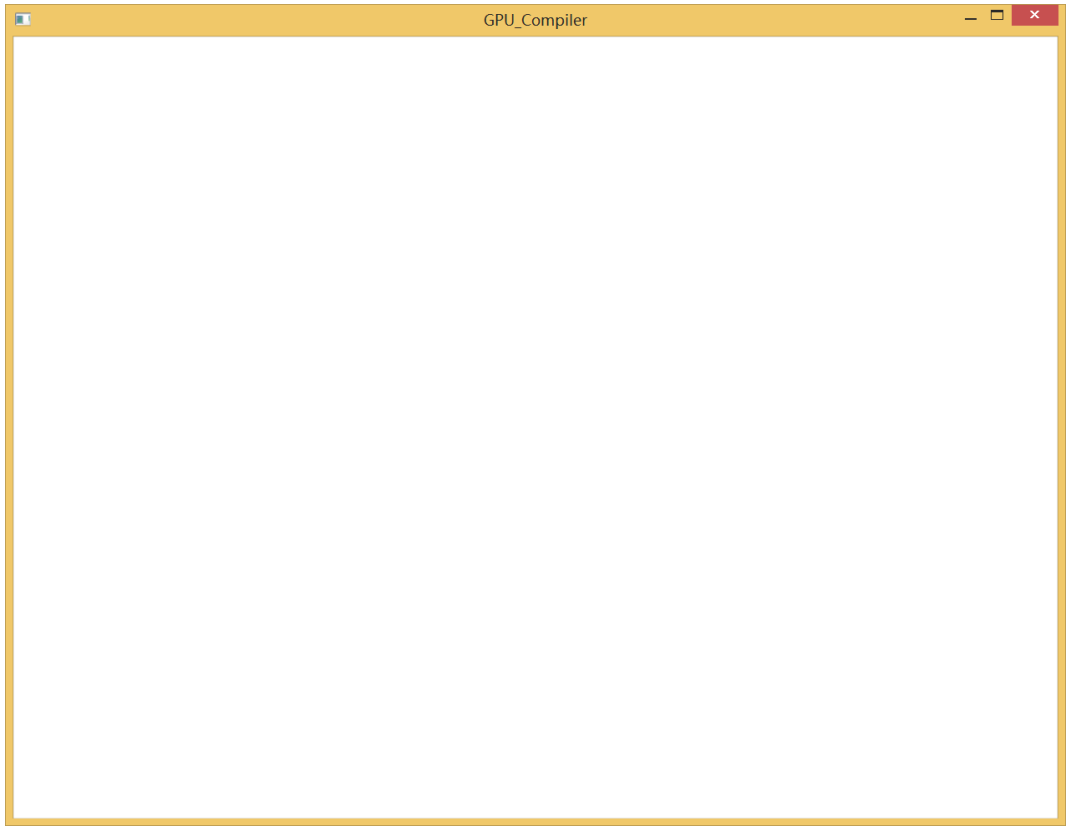
3.1准备工作——配置库

```
int main(int argc, char **argv)
{
    glutInit(&argc,argv);                    //初始化glut库
    glutInitWindowSize(1000,750);            //设置窗口大小
    glutInitWindowPosition(140,60);          //设置窗口位置
    glutInitDisplayMode(GLUT_RGBA);          //设置绘制模式
    glutCreateWindow("GPU_Compiler");        //设置窗口标题

    glewInit();                              //初始化glew库
    ........
```

3.1 准备工作——配置库

- 运行结果：



3.1 准备工作——配置数据

- 库配置完成后，我们可以开始配置模型数据。

- 数据配置为后面的工作提供了两点信息：
  - 三维数据存储在哪块内存区域
  - 内存区块的大小是多少

  OpenGL会从这里申请的内存区域中获取数据。

3.1 准备工作——配置数据

- //含有21个元素的数组，表示三个顶点的位置坐标分别是(0,0,-5),(0,5,-5),(5,0,-5);颜色值分别为(1,0,0,1),(0,1,0,1),(0,0,1,1).当然，这里无法看出这些数字的解读方式。
- GLfloat vertices[21] =
- {
- 0,0,-5, 1,0,0,1,
- 0,5,-5, 0,1,0,1,
- 5,0,-5, 0,0,1,1,
- };

- GLuint vbo;
- glGenBuffers(1,&vbo); //申请1块数据缓冲区，将缓冲区编号保存在vbo中.
- glBindBuffer(GL_ARRAY_BUFFER, vbo); //告诉OpenGL与vbo关联的这块缓冲区将保存与顶点信息相关的数据(GL_ARRAY_BUFFER)
- glBufferData(GL_ARRAY_BUFFER, sizeof(vertices), vertices, GL_STATIC_DRAW); //将vertices数组内的数据传入vbo关联的缓冲区
  //至此，OpenGL可以通过vbo找到vertices中的数据

3.1 准备工作——配置数据

- 在配置数据部分，我们把三维模型的数据与vbo关联起来，以后我们如果需要使用这块内存中的数据，只要调用函数glBindBuffer(GL_ARRAY_BUFFER, vbo)即可。

- 但是，我们把数据一股脑地塞给OpenGL，却没有告诉它如何去识别这些数据：好比有一个8个字节大小的内存块，究竟保存了两个int还是一个double？是position还是color？

- 这些问题在数据传输阶段会得到解决。

3.2顶点着色与片元着色

GLSL的基本语法

- 与C语言类似，着色器从main( )函数开始运行；可自定义函数；注释方式相同
- 不同之处：
  - **#version 330 core**：指定所用的OpenGL语言版本，什么模式
  - **uniform**：变量修饰符，指变量为用户应用程序传递给着色器的数据，它对于给定的图元而言是一个常量
  - **in**：变量修饰符，指这个变量为着色器的输入变量
  - **out**：变量修饰符，指这个变量为着色器的输出变量
  - **layout**：布局限定符，指定变量的布局规范

GLSL的代码编写与命名

- GLSL的代码有两种编写方式
  - 利用字符串数组直接嵌入代码（优点：不需要进行文件读写）

  例如：

```
Const char *vShader = {
  “#version 330 core”
  “layout(location = 0) in vec4 aPosition;”
  …
};
```

  - 文件编写存成文件，然后读写载入（优点：代码修改方便）
- 若采用文本编辑器，GLSL代码的文件后缀建议使用
  - .vert：顶点着色器
  - .frag：片元着色器
  - .tesc，.tese：细分控制/求值着色器
  - .geom：几何着色器
  - .comp：计算着色器

3.2.1着色工作总览

- 准备工作完成后，我们可以开始顶点着色与片元着色工作。

- 这个阶段是可编程渲染管线的核心，因为它决定了从数据到图像的全部加工过程，其他阶段都是为本阶段服务的。事实上，对于其他阶段，可编程管线仍然是采用为函数设置参数的方式进行的，只有这个阶段体现出“可编程”三字的精髓。

- 我们先要介绍着色器(shader)的相关概念。事实上，顶点着色与片元着色最重要的工作就是编写相应的着色器程序。

3.2.1 着色工作总览

- 顶点着色器与片元着色器是两个主要的着色器程序，也是可编程管线不可或缺的内容。

- 顶点着色器处理的是顶点数据，片元着色器处理的是片元数据。我们可以简单地把片元理解为属性还没有确定的像素，片元着色器的主要任务就是确定一个片元的属性，当片元的属性确定之后，它就成为模型显示在屏幕上的一个像素值。

- 顶点着色器接收顶点数据，加工数据，最后传出到片元着色器中；片元着色器则接收顶点着色器中传出的数据，进行加工，生成最后的图像。再次强调：这两个着色器决定了数据到图像的全部处理过程。

3.2.2 顶点着色阶段

- 一个简单的顶点着色器如下：

```
const GLchar *vertexShaderSrc =
        "#version 430\n"        //OpenGL的版本号，可通过getString(GL_VERSION)查询。
        "void main()\n"         //和C语言相似的main函数。
        "{\n"
        "}\n";
```

- 这是一个最简单的顶点着色器，有一个和C语言类似的main()函数。但这个着色器什么也干不了。
- 接下来我们将向这个着色器添加内容。

3.2.2 顶点着色阶段

•const GLchar *vertexShaderSrc =

```
"#version 430\n"
"layout(location=0) in vec3 position;\n"
"layout(location=1) in vec4 color;\n"
"void main()\n"
"{\n"
"}\n";
```

•我们从右向左解读新加的代码：position 是变量名；vec3 表示position这个变量是一个具有三个分量的向量；in表示这是一个输入变量，layout(location=0)表明这个变量在顶点着色器中的编号为0。

•因此，上面两行代码表明：我们需要输入一个vec3类型的名为position的变量，以及一个vec4类型的名为color的变量。这两个变量在顶点着色器中的编号分别为0和1。

3.2.2 顶点着色阶段

- 有一点需要注意的是：顶点着色器是每个顶点都需要运行一遍的程序。

- 举个例子：按照我们在3.1节定义的数据，在数据传输阶段完成后，我们的着色器程序会分别以position = (0,0,-5),color = (1,0,0,1),position = (0,5,-5),color = (0,1,0,1),position = (5,0,-5),color = (0,0,1,1)独立地运行3次。

- 同样的，每个片元都需要运行一次片元着色器。

```
GLfloat vertices[21] =
{
        0,0,-5,  1,0,0,1,
        0,5,-5,  0,1,0,1,
        5,0,-5,  0,0,1,1,
};
```

3.2.2 顶点着色阶段

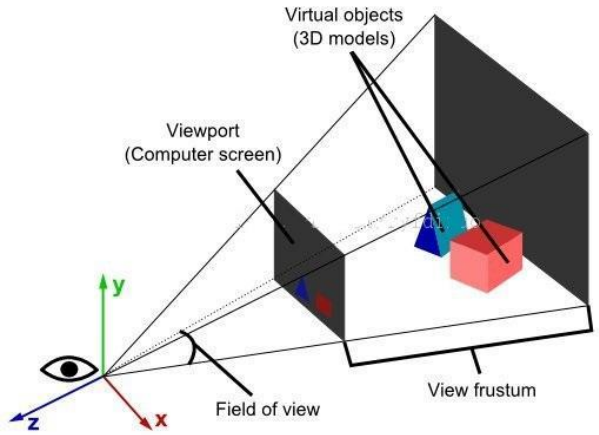
- 同in变量相对应的是out变量，表明该变量将要输出给下一阶段。

```
•const GLchar *vertexShaderSrc =
                "#version 430\n"
                "layout(location=0) in vec3 position;\n"
                "layout(location=1) in vec4 color;\n"
                "out vec4 ocolor;\n"              //声明ocolor是一个vec4类型的输出变量。
                "void main()\n"
                "{\n"
                "        ocolor = color;\n“       //将输入的color赋给ocolor进行输出。
                "}\n";
```

3.2.2 顶点着色阶段

- 顶点着色器最重要的工作是决定一个顶点在屏幕上的位置。
- 片元着色器最重要的工作是决定某块像素的最终颜色。

- 想要把点的三维坐标投影为屏幕上的二维坐标，只要作用一个矩阵变换（模型视图变换与投影变换）。



3.2.2 顶点着色阶段

- const GLchar *vertexShaderSrc =

```
"#version 430\n"
"uniform mat4 PVM;\n"
                    //声明一个名为PVM的4*4的float矩阵，uniform的含义下面解释。
"layout(location=0) in vec3 position;\n"
"layout(location=1) in vec4 color;\n"
"out vec4 ocolor;\n"
"void main()\n"
"{\n"
"           gl_Position = PVM * vec4(position,1.0);\n"
                    //由输入的position值计算这个顶点在屏幕上的坐标。
"           ocolor = color;\n"
"}\n";
```

3.2.2 顶点着色阶段

- 我们加入了两行新的代码
  - "uniform mat4 PVM;\n" 声明了一个矩阵类型的变量，和vec3类似，mat4也是GLSL内置类型，表明这是一个4*4且元素为float的矩阵。

  - uniform关键字和in关键字对应，修饰一个不随顶点属性变化而变化的变量。举例：在输入position和color变量时，不同的顶点会有不同的位置与颜色；但所有的顶点都要进行相同的矩阵变换。uniform类型的变量从头到尾只需要输入一次即可；而in变量在每次运行顶点着色器时都要输入一次。

3.2.2 顶点着色阶段

- gl_Position = PVM * vec4(position,1.0) 利用矩阵变换决定了三维坐标为position的顶点在屏幕上的位置。

- gl_Position是GLSL内置变量，专用于表示顶点在屏幕上的坐标信息。注意：gl_Position是一个含有4个分量的向量，还要通过处理（透视除法）才能得到真正的屏幕坐标，不过这是OpenGL自动进行的工作

3.2.2 顶点着色阶段

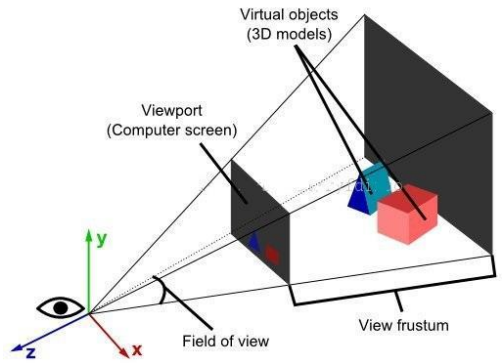
- 对于复杂的变换而言，PVM矩阵可能很难计算。幸运的是，我们可以通过函数调用去获取变换矩阵，只要传递简单的参数即可。

//以下函数定义在vmath.h头文件中。

Matrix4f modelView =Matrix4f::createLookAt(Vector3f(0,0,0),Vector3f(0,0,-1),Vector3f(0,1,0)); //模型视图矩阵

Matrix4f proj = Matrix4f::createFrustum(-1,1,-1,1,1,50);//投影矩阵

Matrix4f PVM = proj*modelView; //最终的变换矩阵为二者相乘。



3.2.2 顶点着色阶段

- 以上就是一个具有完整功能的顶点着色器，它提供了必不可少的功能：用每个顶点的三维坐标计算其在屏幕上的位置。同时它还读取了每个顶点的颜色信息，并将之作为out变量输出，传递给片元着色器。

- 片元着色器的写法与顶点着色器类似，它获取顶点着色器输出的out变量作为输入，产生片元最终的颜色值（也就是片元对应像素地颜色值）。

3.2.3 片元着色阶段

```
const GLchar *fragmentShaderSrc =
        "#version 430\n"
        "in vec4 ocolor;\n"             //顶点着色器的输出作为输入。
        "out vec4 Fragment;\n"          //输出该片元最终颜色值。
        "void main()\n"
        "{\n"
        "       Fragment = ocolor;\n"
        "}\n";
```

一般而言，片元着色器第一个声明为输出的vec4型变量代表该片元最终颜色值。

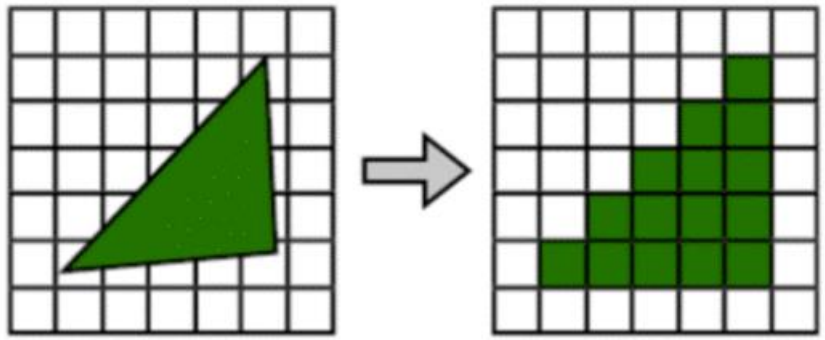
3.2.4 着色中的光栅化

- 如前所述，片元着色器获取顶点着色器的out变量作为in变量。

- 但是一般而言，片元的数量要远远多于顶点数目，而且片元与顶点无对应关系，OpenGL是如何由顶点属性确定片元属性的？

3.2.4 着色中的光栅化

- 原来，在顶点着色与片元着色之间，OpenGL自动完成了一步重要的工作——光栅化。



- 光栅化过程产生了一系列片元，根据片元所在的三角形，计算每块片元的重心坐标，利用重心坐标插值三角形顶点的属性值，得到该片元相应的属性值。

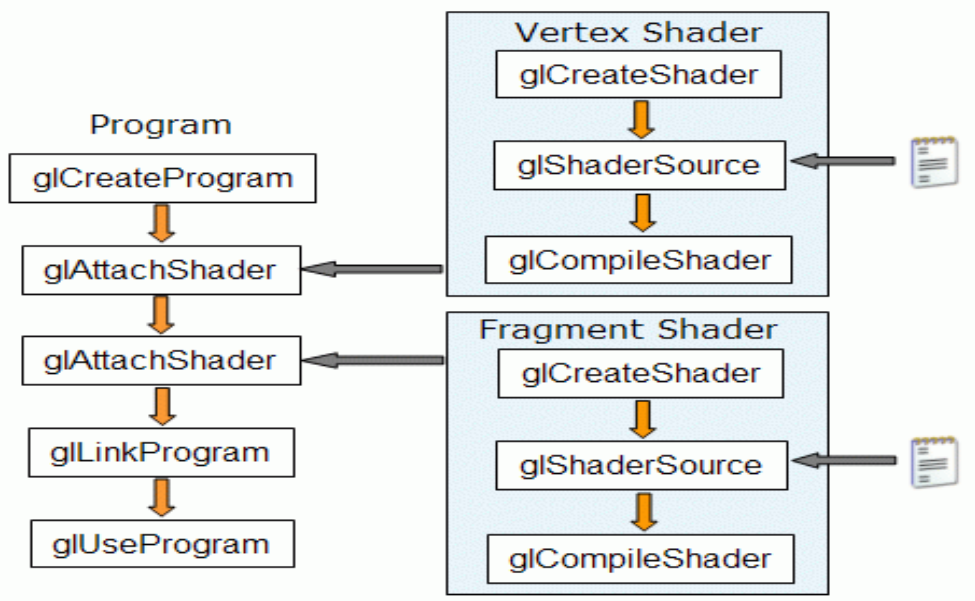
3.2.5 着色器的编译和链接

- 以上，我们介绍了顶点着色器与片元着色器。但是到目前为止，我们只是写出了字符串类型的源代码，无法直接运行。

- 同C语言类似，着色器程序也需要编译和链接，生成由GPU执行的可执行程序。这些工作通过调用OpenGL函数完成。

3.2.5 着色器的编译和链接

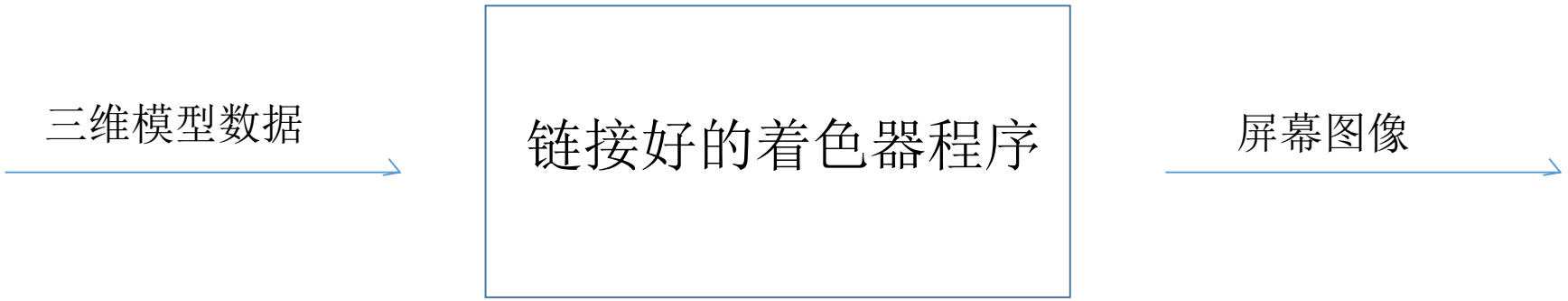
```
//编译顶点着色器
GLuint vertexShader = glCreateShader(GL_VERTEX_SHADER);
glShaderSource(vertexShader, 1, vertexShaderSrc, NULL);
glCompileShader(vertexShader);
//编译片元着色器
GLuint fragShader = glCreateShader(GL_FRAGMENT_SHADER)
glShaderSource(fragShader, 1, fragmentShaderSrc, NULL);
glCompileShader(fragShader);
//链接顶点着色器与片元着色器，生成最终着色程序
GLuint program = glCreateProgram();
glAttachShader(program, vertexShader);
glAttachShader(program, fragShader);
```



3.3数据传输

3.3数据传输

- 至此，我们已经介绍了可编程管线的主要部分——着色器程序的编写。一个完整的着色器程序应当能够接受三维模型数据，产生屏幕图像。



- 此外，我们已经在准备工作阶段保存了模型数据，以下将要介绍数据传输阶段，了解如何将数据传到着色器程序中。

3.3数据传输

- 遗留问题：我们在配置数据时，一股脑的把待传输的in型数据的内存塞给OpenGL，却没有告诉它数据应当怎样划分，又代表什么意义。

- glEnableVertexAttribArray()与glVertexAttribPointer()函数解决了上述问题。这两个函数定义了内存块中数据的读取方式与含义，并将之上传到着色器中。

3.3 数据传输

- 举例：我们希望将3.1节中的vertices数组保存的位置信息传递到顶点着色器的position变量中，那么可以使用以下代码

```
//允许顶点着色器中编号为0的变量传输数据，编号由着色器中layout语句声明。
glEnableVertexAttribArray(0);
//从左往右参数表示的含义分别为：编号为0的变量，每次需要传入3个GL_FLOAT类型的数据；
//GL_FALSE表示不进行归一化； sizeof(float)*7表示每两个待传入数据相隔的字节数； (void*)0
//表示第一个待传输数据的起始地址为vertices数组首字节往后偏移0个字节。
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, sizeof(float)*7, (void*)0);
```

```
"layout(location=0) in vec3 position;\n"
"layout(location=1) in vec4 color;\n"
```

```
GLfloat vertices[21] =
{
        0,0,-5,  1,0,0,1,
        0,5,-5,  0,1,0,1,
        5,0,-5,  0,0,1,1,
};
```

3.3 数据传输

- 同理，传输color变量的方式如下：
  - glEnableVertexAttribArray(1);
  - glVertexAttribPointer(1, 4, GL_FLOAT, GL_TRUE, sizeof(float)*7, (void*)(sizeof(float)*3));

- 对于in型变量，由于各个顶点的属性值各不相同，因此数据传输比较复杂。

- 但是对于uniform变量，传输就简单了很多。
- 只需调用glUniform*()系列函数即可。对于不同类型的uniform变量，注意调用不同类型的glUniform*()函数。

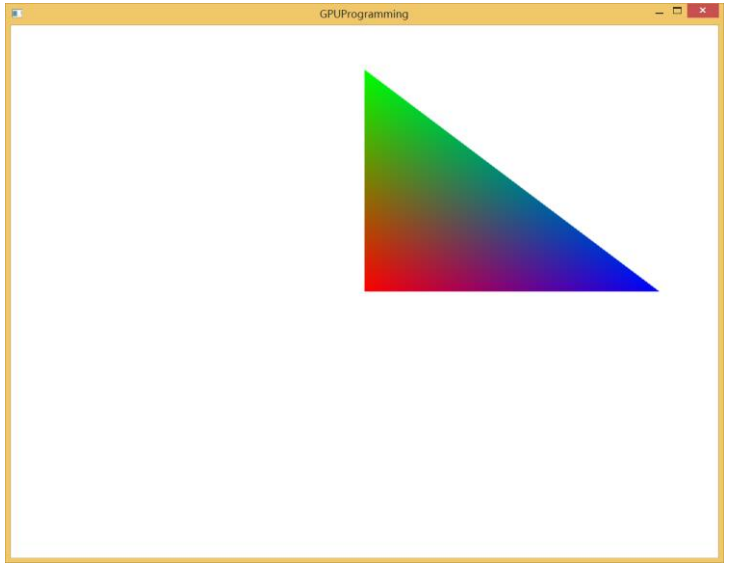
3.4 绘制阶段

3.4 绘制阶段

- 当我们完成上述所有工作后，就可以调用OpenGL绘制函数进行渲染绘制了。

//i，j为整数，表示从传给顶点着色器的第i+1个顶点开始,每三个相邻顶点连接成三角形，一共连接j个顶点，绘制出j/3个三角形。

glDrawArrays(GL_TRIANGLES,i,j);



2. 注意事項

GLSL编写的注意事项

- 着色器代码无法调试(debug)
- 在数据传输阶段与着色器编写阶段要小心谨慎，建议每输入一组数据，就进行一次测试。
- 务必细心！

更详细资料 (GLSL详细语法介绍)

- 童伟华老师：2019年《计算机图形学》第6章
  - http://staff.ustc.edu.cn/~tongwh/CG_2019/index.html

第六章 可编程着色器

- 可编程流水线
- GLSL(I)
- GLSL(II)
- GLSL(III)[Wave.zip, Wave2.zip, Morph.zip, Particle.zip, Nonphoto.zip, Phong.zip, Cubemap.zip, Bumpmap.zip]

Shader编程学习资料

- Learn OpenGL: Shaders
  - https://learnopengl.com/Getting-started/Shaders
- Snail Shader Program
  - https://www.shadertoy.com/view/ld3Gz2
- RenerMonkey
  - https://gpuopen.com/archive/gamescgi/rendermonkey-toolsuite/
- ShaderGen
  - https://github.com/mellinoe/ShaderGen
- Snail.rocks
  - https://theepicsnail.github.io/rocks/

Q&A