**OpenGL使用shader渲染图形**

**（一）实验环境**

1. 显卡驱动支持OpenGL3.2及以上版本

2. freeglut库或者glut库

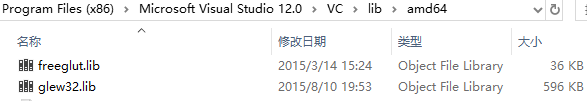
3. glew库

4. 集成开发环境（Visual Studio 2013)

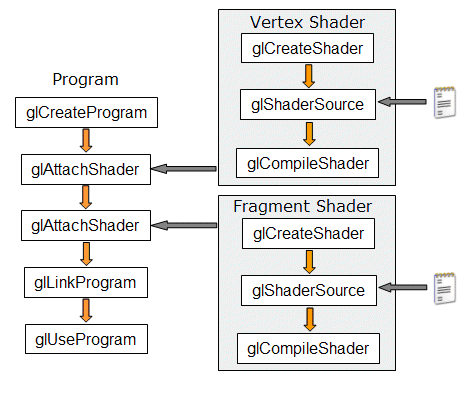
**（二）环境配置**

1.在VS安装目录下的include文件夹下新建文件夹gl，把需要的头文件放进去，如下图所示。

2.所需库文件配置如下图所示。



**（三）实验过程**



**执行流程图**

在OpenGL程序中使用shader的初始化一般需要依次执行以下步骤：

1、顶点着色程序的源代码和片段着色程序的源代码分别写入到一个文件里（或字符数组）里面，一般顶点着色器源码文件后缀为.vert，片段着色器源码文件后缀为.frag；

2、使用glCreateshader()分别创建一个顶点着色器对象和一个片段着色器对象；

3、使用glShaderSource()分别将顶点/片段着色程序的源代码字符数组绑定到顶点/片段着色器对象上；

4、使用glCompileShader()分别编译顶点着色器和片段着色器对象（最好检查一下编译的成功与否）；

5、使用glCreaterProgram()创建一个着色程序对象；

6、使用glAttachShader(）将顶点和片段着色器对象附件到需要着色的程序对象上；

7、使用glLinkProgram()分别将顶点和片段着色器和着色程序执行链接生成一个可执行程序（最好检查一下链接的成功与否）；

8、使用glUseProgram()将OpenGL渲染管道切换到着色器模式，并使用当前的着色器进行渲染；

**（四）代码**

项目一共包含5个文件。2个资源文件（VertexShader.vert和FragmentShader.frag，分别是顶点着色器源码文件和片段着色器源码文件），2个cpp文件（Hello GLSL.cpp和Textfile.cpp）,1个头文件Textfile.h。

VertexShader.vert文件内容：

1. in vec4 VertexPosition;
2. in vec4 VertexColor;
3. out vec4 Color;
4. **void** main()
5. {
6. Color =VertexColor;
7. gl\_Position = VertexPosition;
8. }

FragmentShader.frag文件内容：

1. in vec4 Color;
2. out vec4 FragColor;
3. **void** main()
4. {
5. FragColor = Color;
6. }

Hello GLSL.cpp文件内容：

#include <GL/glew.h>

#include "Textfile.h"

#include <GL/freeglut.h>

#include <iostream>

#pragma comment(lib,"glew32.lib")

using namespace std;

GLuint vShader, fShader;//顶点/片段着色器对象

GLuint vaoHandle;// VAO对象

//顶点位置数组

float positionData[] = {

-0.5f,-0.5f,0.0f,1.0f,

0.5f,-0.5f,0.0f,1.0f,

0.5f,0.5f,0.0f,1.0f,

-0.5f,0.5f,0.0f,1.0f

};

//顶点颜色数组

float colorData[] = {

1.0f, 0.0f, 0.0f,1.0f,

0.0f, 1.0f, 0.0f,1.0f,

0.0f, 0.0f, 1.0f,1.0f,

1.0f,1.0f,0.0f,1.0f

};

void initShader(const char \*VShaderFile, const char \*FShaderFile)

{

//2、编译着色器

//创建着色器对象：顶点着色器

vShader = glCreateShader(GL\_VERTEX\_SHADER);

//错误检测

if (0 == vShader)

{

cerr << "ERROR : Create vertex shader failed" << endl;

exit(1);

}

//把着色器源代码和着色器对象相关联

const GLchar \*vShaderCode = textFileRead(VShaderFile);

const GLchar \*vCodeArray[1] = { vShaderCode };

//将字符数组绑定到对应的着色器对象上

glShaderSource(vShader, 1, vCodeArray, NULL);

//编译着色器对象

glCompileShader(vShader);

//检查编译是否成功

GLint compileResult;

glGetShaderiv(vShader, GL\_COMPILE\_STATUS, &compileResult);

if (GL\_FALSE == compileResult)

{

GLint logLen;

//得到编译日志长度

glGetShaderiv(vShader, GL\_INFO\_LOG\_LENGTH, &logLen);

if (logLen > 0)

{

char \*log = (char \*)malloc(logLen);

GLsizei written;

//得到日志信息并输出

glGetShaderInfoLog(vShader, logLen, &written, log);

cerr << "vertex shader compile log : " << endl;

cerr << log << endl;

free(log);//释放空间

}

}

//创建着色器对象：片断着色器

fShader = glCreateShader(GL\_FRAGMENT\_SHADER);

//错误检测

if (0 == fShader)

{

cerr << "ERROR : Create fragment shader failed" << endl;

exit(1);

}

//把着色器源代码和着色器对象相关联

const GLchar \*fShaderCode = textFileRead(FShaderFile);

const GLchar \*fCodeArray[1] = { fShaderCode };

glShaderSource(fShader, 1, fCodeArray, NULL);

//编译着色器对象

glCompileShader(fShader);

//检查编译是否成功

glGetShaderiv(fShader, GL\_COMPILE\_STATUS, &compileResult);

if (GL\_FALSE == compileResult)

{

GLint logLen;

//得到编译日志长度

glGetShaderiv(fShader, GL\_INFO\_LOG\_LENGTH, &logLen);

if (logLen > 0)

{

char \*log = (char \*)malloc(logLen);

GLsizei written;

//得到日志信息并输出

glGetShaderInfoLog(fShader, logLen, &written, log);

cerr << "fragment shader compile log : " << endl;

cerr << log << endl;

free(log);//释放空间

}

}

//3、链接着色器对象

//创建着色器程序

GLuint programHandle = glCreateProgram();

if (!programHandle)

{

cerr << "ERROR : create program failed" << endl;

exit(1);

}

//将着色器程序链接到所创建的程序中

glAttachShader(programHandle, vShader);

glAttachShader(programHandle, fShader);

//将这些对象链接成一个可执行程序

glLinkProgram(programHandle);

//查询链接的结果

GLint linkStatus;

glGetProgramiv(programHandle, GL\_LINK\_STATUS, &linkStatus);

if (GL\_FALSE == linkStatus)

{

cerr << "ERROR : link shader program failed" << endl;

GLint logLen;

glGetProgramiv(programHandle, GL\_INFO\_LOG\_LENGTH,

&logLen);

if (logLen > 0)

{

char \*log = (char \*)malloc(logLen);

GLsizei written;

glGetProgramInfoLog(programHandle, logLen,

&written, log);

cerr << "Program log : " << endl;

cerr << log << endl;

}

}

else//链接成功，在OpenGL管线中使用渲染程序

{

glUseProgram(programHandle);

}

}

void initVBO()

{

//绑定VAO

glGenVertexArrays(1, &vaoHandle);

glBindVertexArray(vaoHandle);

// Create and populate the buffer objects

GLuint vboHandles[2];

glGenBuffers(2, vboHandles);

GLuint positionBufferHandle = vboHandles[0];

GLuint colorBufferHandle = vboHandles[1];

//绑定VBO以供使用

glBindBuffer(GL\_ARRAY\_BUFFER, positionBufferHandle);

//加载数据到VBO

glBufferData(GL\_ARRAY\_BUFFER, 16 \* sizeof(float),

positionData, GL\_STATIC\_DRAW);

//绑定VBO以供使用

glBindBuffer(GL\_ARRAY\_BUFFER, colorBufferHandle);

//加载数据到VBO

glBufferData(GL\_ARRAY\_BUFFER, 16 \* sizeof(float),

colorData, GL\_STATIC\_DRAW);

glEnableVertexAttribArray(0);//顶点坐标

glEnableVertexAttribArray(1);//顶点颜色

//调用glVertexAttribPointer之前需要进行绑定操作

glBindBuffer(GL\_ARRAY\_BUFFER, positionBufferHandle);

glVertexAttribPointer(0, 4, GL\_FLOAT, GL\_FALSE, 0, (GLubyte \*)NULL);

glBindBuffer(GL\_ARRAY\_BUFFER, colorBufferHandle);

glVertexAttribPointer(1, 4, GL\_FLOAT, GL\_FALSE, 0, (GLubyte \*)NULL);

}

void init()

{

//初始化glew扩展库

GLenum err = glewInit();

if (GLEW\_OK != err)

{

cout << "Error initializing GLEW: " << glewGetErrorString(err) << endl;

}

//加载顶点和片段着色器对象并链接到一个程序对象上

initShader("VertexShader.vert","FragmentShader.frag");

//绑定并加载VAO，VBO

initVBO();

glClearColor(0.0, 0.0, 0.0, 0.0);

}

void display()

{

glClear(GL\_COLOR\_BUFFER\_BIT);

//使用VAO、VBO绘制

glBindVertexArray(vaoHandle);

glDrawArrays(GL\_TRIANGLE\_FAN, 0, 4);

glBindVertexArray(0);

glutSwapBuffers();

}

//ESC键用于退出使用着色器

void keyboard(unsigned char key, int x, int y)

{

switch (key)

{

case 27:

glDeleteShader(vShader);

glUseProgram(0);

glutPostRedisplay(); //刷新显示

break;

}

}

int main(int argc, char\*\* argv)

{

glutInit(&argc, argv);

glutInitDisplayMode(GLUT\_DOUBLE | GLUT\_RGB);

glutInitWindowSize(600, 600);

glutInitWindowPosition(100, 100);

glutCreateWindow("Hello GLSL");

init();

glutDisplayFunc(display);

glutKeyboardFunc(keyboard);

glutMainLoop();

return 0;

}

Textfile.cpp文件内容：

1. #include "Textfile.h"
3. unsigned **char** \* readDataFromFile(**char** \*fn)
4. {
5. **FILE** \*fp;
6. unsigned **char** \*content = NULL;
7. **int** count = 0;
8. **if** (fn != NULL) {
9. fp = fopen(fn, "rb");
10. **if** (fp != NULL) {
11. fseek(fp, 0, SEEK\_END);
12. count = ftell(fp);
13. rewind(fp);
14. **if** (count > 0) {
15. content = (unsigned **char** \*)malloc(**sizeof**(unsigned **char**) \* (count + 1));
16. count = fread(content, **sizeof**(unsigned **char**), count, fp);
17. content[count] = '\0';
18. }
19. fclose(fp);
20. }
21. }
22. **return** content;
23. }
25. //读入字符流
26. **char** \*textFileRead(**const** **char** \*fn)
27. {
28. **FILE** \*fp;
29. **char** \*content = NULL;
30. **int** count = 0;
31. **if** (fn != NULL)
32. {
33. fp = fopen(fn, "rt");
34. **if** (fp != NULL)
35. {
36. fseek(fp, 0, SEEK\_END);
37. count = ftell(fp);
38. rewind(fp);
39. **if** (count > 0)
40. {
41. content = (**char** \*)malloc(**sizeof**(**char**) \* (count + 1));
42. count = fread(content, **sizeof**(**char**), count, fp);
43. content[count] = '\0';
44. }
45. fclose(fp);
46. }
47. }
48. **return** content;
49. }
51. **int** textFileWrite(**char** \*fn, **char** \*s)
52. {
53. **FILE** \*fp;
54. **int** status = 0;
55. **if** (fn != NULL) {
56. fp = fopen(fn, "w");
57. **if** (fp != NULL) {
58. **if** (fwrite(s, **sizeof**(**char**), strlen(s), fp) == strlen(s))
59. status = 1;
60. fclose(fp);
61. }
62. }
63. **return**(status);
64. }

Textfile.h文件内容：

1. #ifndef TEXTFILE\_H
2. #define TEXTFILE\_H
4. #include <stdio.h>
5. #include <stdlib.h>
6. #include <string.h>
8. **char** \*textFileRead(**const** **char** \*fn);
9. **int** textFileWrite(**char** \*fn, **char** \*s);
10. unsigned **char** \*readDataFromFile(**char** \*fn);

**（五）运行结果**

****