**OpenGL + QT**

**一、简介**

OpenGL 3.3核心模式 + Qt 5.9 MinGW

优势：借助于Qt的封装，无需配置**第三方库**；更接近实战

**第三方库**，例如GLFW+GLAD

* GLFW：创建支持OpenGL的窗口，需要解决操作系统层面的不同：创建窗口、定义上下文、处理用户输入；GLFW使得无论在哪个操作系统，操作都是一致的。
* GLAD：使得代码可以用于不同的OpenGL驱动，OpenGL本身只是标准/规范，各个厂家的具体实现方式不同

如果没有类似GLAD的库，Windows下：需要通过**函数指针**调用**显卡的函数**，但是显卡驱动具体函数的地址，运行时才知道

例：

typedef void (\*GL\_GENBUFFERS)(GLsizei, GLuint\*);

GL\_GENBUFFERS glGenBuffers = (GL\_GENBUFFERS)wglGetProcAddress(“glGenBuffers”);

unsigned int buffer;

glGenBuffers(1,&buffer);

即自定义的指针glGenBuffers绑定了显卡的指针glGenBuffers

获取显卡在当前上下文状态下的OpenGL函数的地址

**二、OpenGL**

Open Graphics Library,是由Khronos组织制定并维护的规范

OpenGL核心是一个C库，同时也支持多种语言的派生

* **核心模式（Core Profile）**

也叫可编程管线，提供了更多的灵活性，更高的效率，可以更深入的理解图形编程

理解：GPU渲染是一个流水线

顶点数据-**顶点着色器**-形状图元装配-**几何着色器**-光栅化-**片段着色器**-测试与混合

三个着色器是可以编程，几何着色器有自动模式，顶点和片段着色器必须自己写

* **立即渲染模式（Immediate Mode）**

也叫固定渲染管线，OpenGL大多数功能被库隐藏起来，容易使用和理解，但效率太低；开发者难以控制如何计算；OpenGL 3.2开始，推出核心模式。

状态机（State Machine）

OpenGL自身是一个巨大的状态机：描述该如何操作的所有变量的大集合

OpenGL的状态通常被称为上下文（Context）

状态设置函数（State-changing Function）

状态应用函数（State-using Function）

数据输入-当前绘制状态（材质、光照效果、如何解读点数据…）-图像输出

我们通过改变一些上下文变量来改变OpenGL状态，从而告诉OpenGL如何去绘图。

**三、对象**

一个对象是指一些选项的集合，代表OpenGL的一个子集。

例如，可以用一个对象来代表绘图窗口的设置，大小，支持的颜色位数，可以把对象看作一个C风格的结构体。

例子：

struct OpenGL\_Context

{

…

object\* object\_Window\_Target;(窗口大小、颜色位数)

…

}

当前状态只有一份，如果每次显示不容的效果，都重新配置会很麻烦。

需要一些小助理（对象），帮忙记录某些状态信息，以便复用。

如果有10种子集，每个子集有10种不同的状态，就需要100个小助理(对象)

当前状态只有一份，可以通过装配这些对象来完成（喊来10个小助理）

例子：

创建对象，分配ID

GLuint objectId=0;

glGenObject(1,&objectId);

绑定后确定该对象的工作内容

glBindObject(GL\_WINDOW\_TARGET,objectId);

设置

glSetObjectOption(GL\_WINDOW\_TARGET,GL\_OPTION\_WINDOW\_WIDTH,800);

glSetObjectOption(GL\_WINDOW\_TARGET,GL\_OPTION\_WINDOW\_HEIGHT,600);

对象已经记录了上面的内容，可以休息了。需要查看记录的时候再喊。

glBindObject(GL\_WINDOW\_TARGET,0);

一旦重新绑定这个对象到GL\_WINDOW\_TARGET位置，这些选项会重新生效

**四、Example**

**4-1 Window**

QOpenGLWidget:不需要GLFW

QOpenGLWidget提供了三个便捷的虚函数，可以重载，用来重新实现典型的OpenGL任务。

* + paintGL：渲染OpenGL场景，widget需要更新时调用；
  + resizeGL:设置OpenGL视口，投影等。Widget调整大小（或首次显示）时调用
  + initializeGL:设置OpenGL资源和状态，第一次调用resizeGL()/paintGL()之前调用

如果需要从paintGL()以外的位置触发重新绘制（典型示例时使用计时器设置场景动画），则应调用widget的update()函数来安排更新。

调用paintGL(), resizeGL(), initailizeGL()时，widget的OpenGL呈现上下文变为当前。如果要从其他位置（如widget的构造函数或自己的绘制函数中）调用标准OpenGL API函数，则必须先调用make current();

在paintGL()以外的地方调用绘制函数，没有意义。绘制图像最终被paintGL()覆盖。

工程：创建OpenGL绘制窗口

**4-2 VAO VBO**

标准化设备坐标（Normalized Device Coordinated,NDC）顶点着色器处理过后,就应该是标准化设备坐标了，xyz的值在-1.0到1.0之间，落在范围外的坐标都会被裁剪。

**顶点着色器：**

如何把内存数据放到显卡？内存数据通过GL\_ARRAY\_BUFFER成组传送给显卡

它会在GPU上创建内存，用于存储顶点数据

它通过**顶点缓冲对象(Vertex Buffer Objects, VBO)**管理

顶点缓冲对象的缓冲类型是GL\_ARRAY\_BUFFER

配置OpenGL如何解释这些内存

通过**顶点数组对象（Vertex Array Objects, VAO）**管理

OpenGL允许我们同时绑定多个缓冲，只要它们是不同的缓冲类型。每个缓冲类型类似于前面说的子集，每个VBO是一个小助理，VBO1,VBO2…切换）

VAO并不保存实际数据，而是放顶点结构定义

显卡如何解释收到的数据？

VAO没有类型，也就是类型唯一

数组里的每一个项对应一个属性的解析

例子：

#version 330 core

layout(location = 0) in vec3 aPos;

void main()

{

gl\_Position = vec4(aPos.x,aPos.y,aPos.z,1.0);

}

例子：

float vertices[]=

{

-0.5f,-0.5f,0.0f,

0.5f,-0.5f,0.0f,

0.0f,0.5f,0.0f

};

创建VAO，VBO对象并赋予ID

unsigned int VBO,VAO;

glGenVertexArrays(1,&VAO);

glGenBuffers(1,&VBO);

绑定VAO、VBO对象

glBindVertexArray(VAO);

glBindBuffer(GL\_ARRAY\_BUFFER,VBO);

为当前绑定到target的缓冲区对象创建一个新的数据存储

如果data不是NULL，则使用来自此指针的数据初始化数据存储

*void glBufferData(GLenum target, GLsizeiptr size, const GLvoid \*data, GLenum usage)*

glBufferData(GL\_ARRAY\_BUFFER,sizeof(vertices),vertices,GL\_STATIC\_DRAW);

告知显卡如何如何解析缓冲里的属性值

Void glVertexAttribPointer(GLuint index, Glint size, GLenum type, GLboolean normalized, GLsizei stride, const void \*offset)

glVertexAttribPointer(0,3,GL\_FLOAT,GL\_FALSE,3\*sizeof(float),(void\*)0);

开启VAO管理的第一个属性值

glEnableVertexAttribArray(0);

让VAO，VBO休息

glBindBuffer(GL\_ARRAY\_BUFFER,0);

glBindVertexArray(0);

绘制

glBindVertexArray(VAO);

glDrawArrays(GL\_TRIANGLES,0,3);

此时无着色器却画出了白色三角形，因为调用了默认着色器，按理说不应该绘制。

**4-3 编译链接着色器**

着色器源码

const char \*vertexShaderSource = "#version 330 core\n"

"layout (location = 0) in vec3 aPos;\n"

"void main()\n"

"{\n"

"gl\_Position = vec4(aPos.x,aPos.y,aPos.z,1.0);\n"

"}\0";

const char \*fragmentShaderSource = "#version 330 core\n"

"out vec4 FragColor;\n"

"void main()\n"

"{\n"

"FragColor = vec4(1.0f,0.5f,0.2f,1.0f);\n"

"}\n\0";

创建并且编译着色器

unsigned int vertexShader = glCreateShader(GL\_VERTEX\_SHADER);

*void glShaderSource(GLuint shader, GLsizei count, const GLchar \*\*string,const Glint \*length)*

glShaderSource(vertexShader,1,&vertexShaderSource,NULL);

glCompileShader(vertexShader);

检查编译错误

int success;

char infoLog[512];

glGetShaderiv(vertexShader,GL\_COMPILE\_STATUS,&success);

if(!=success)

{

glGetShaderInfoLog(vertexShader,512,NULL,infoLog);

qDebug()<<“Error::VSHADER“;

}

创建片段着色器

unsigned int fragmentShader = glCreateShader(GL\_FRAGMENT\_SHADER);

void glShaderSource(GLuint shader, GLsizei count, const GLchar \*\*string,const Glint \*length)

glShaderSource(fragmentShader,1,&fragmentShaderSource,NULL);

glCompileShader(fragmentShader);

链接

unsigned int shaderProgram = glCreateProgram();

glAttachShader(shaderProgram,vertexShader);

glAttachShader(shaderProgram,fragmentShader);

glLinkProgram(shaderProgram);

glDeleteShader(vertexShader);

glDeleteShader(fragmentShader);

**4-4 EBO矩形绘制**