Qt框架下OpenGL的应用研究

第一章 OpenGL简介

OPenGL是一个API（应用程序编程接口），它为我们提供了大量可用于操作图像的函数。是一种规范

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，更重要的是可以更深入的理解图形编程

立即渲染模式immediate mode：早期的OpenGL使用的模式（也就是固定渲染模式）

OpenGL的大多数功能都被库隐藏起来，容易使用和理解，但是效率太低。

开发者很少能控制OpenGL如何进行计算

因此，从OpenGL3.2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
- 状态应用的函数 state-using function

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

一个对象是指一些选项的集合，代表OpenGL状态的一个子集

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

通常把OpenGL上下文比作一个大的结构体，包含很多子集

当前状态只有一份，需要一些快照（对象的当前值），记录某些状态信息，以便复用。

当前状态（仅一份），可以通过装配这些对象来完成（为每一个子集找出一张快照）

QOpenGLWidget：不需要GLFW

QOpenGLWidge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() 或 paintGL() 之前调用一次
```

如果需要从paintGL()以外的位置触发重新绘制（典型事例是使用计算器设置场景动画），则应调用widge的update()来安排更新

调用paintGL()，resizeGL() 或者 initializeGL() 时，widget的OpenGL呈现上下文将变为当前。如果需要从其它位置（例如，在widget的构造函数或自己的绘制函数中）调用标准OpenGL API函数，则必须首先调用makeCurrent()

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

QOpenGLFuncions_x_x_Core：不需要GLAD

QOpenGLFuncions_x_x_Core提供OpenGL x.x版本核心模式的所有功能，是对OpenGL函数的封装

initializeOpenGLFunctions：初始化OpenGL函数，将Qt里的函数指针指向显卡的函数

在OpenGL中，一切都处于三维空间，但屏幕或窗口是一个二维的像素矩阵。

OpenGL需要将三维坐标转换为适合屏幕的二维像素。

将三维坐标转换为二维像素的过程由OpenGL的图形管道管理

第二章 着色器

图形管道可以分为两个主要部分：

1. 将三维坐标转换为二维坐标
2. 将二维坐标转换为实际的彩色像素

在OpenGL中，一个片段是OpenGL渲染单个像素所需的所有数据

glDrawArrays (GL_TRIANGLES,0,3)

第一步：顶点着色器

第二部：几何着色器

第三步：形状装配

第三步：光栅化

第三步：片段着色器

第六步：测试与混合

顶点着色器：它会在GPU上创建内容，用于存储我们的顶点数据

顶点缓存对象的缓冲类型是GL_ARRAY_BUFFER

通过顶点缓冲对象（Vertex Buffer Object，VBO）管理

配置OpenGL如何解释这些内存：通过顶点数组对象（Vertex Array Object，VAO）管理

标注化设备坐标：

顶点着色器中处理过后，就是标准化设备坐标，x，y，z的值在-1和1的一小段空间（立方体）。落在范围之外的坐标都会被cut

NDC随后将通过视口变换，使用glViewport提供的数据转换为屏幕空间坐标。由此产生的屏幕空间坐标随后被转换为片段，作为输入传递给片段着色器。

OpenGL Shading Language(GLSL) 着色器的编程语言

```
/* 这是GPU的代码 */
#version 330 core

layout (location = 0) in vec3 aPos;

void main()

{

    gl_Position = vec4(aPos.x,aPos.y,aPos.z,1.0);

}
```

顶点着色器

索引缓冲对象（Element Buffer Object，EBO，也叫 Index Buffer Object，IBO）

可以绘制两个三角形来组成一个矩形（OpenGL主要处理三角形），这会生成下面顶点的集合

```
float vertices[] = {
    0.5f, 0.5f, 0.0f,   // top right
    0.5f, -0.5f, 0.0f,   // button right 重合点1
    -0.5f, 0.5f, 0.0f,   // top left  重合点2
    /* seconde triangle */
    0.5f, -0.5f, 0.0f,   // button right 重合点1
    -0.5f, -0.5f, 0.0f,   // button left
    -0.5f, 0.5f, 0.0f,   // top left  重合点2
};
```

一个矩形只需要四个点，上面有两个点多余。因此，使用 IBO来管理

```
float vertices[] = {
      0.5f, 0.5f, 0.0f,   // top right
      0.5f, -0.5f, 0.0f,   // button right 重合点1
      -0.5f, -0.5f, 0.0f,   // button left
      -0.5f, 0.5f, 0.0f,   // top left  重合点2
};
unsigned int indices[] = {
      0,1,3,
      1,2,3
};
```

QOpenGLShaderProgram

使用Qt的封装可以大大降低代码量，只需使用下面的简单函数调用即可

```
bool QOpenGLShaderProgram::addShaderFromSourceCode(QGLShader::ShaderType
type,const char* source);
bool QOpenGLShaderProgram::addShaderFromSourceFile(QGLShader::ShaderType
type,const QString &fileName);
virtual bool link()
bool bind()
QString log() const;
```

```
//     int success; char infoLog[512];
```

```
//      glGetShaderiv(vertexShader,GL_COMPILE_STATUS,&success);
//      if(!success)
//      {
//            glGetShaderInfoLog(vertexShader,512,NULL,infoLog);
//            qDebug()<<"error, shader,vertex compilation failed"<<infoLog;
//      }

//      /* 编译片段着色器 */
//      unsigned int fragmentShader = glCreateShader(GL_FRAGMENT_SHADER);
//      glShaderSource(fragmentShader,1,&fragmentShaderSource,NULL);
//      glCompileShader(fragmentShader);

//      glGetShaderiv(fragmentShader,GL_COMPILE_STATUS,&success);
//      if(!success)
//      {
//            glGetShaderInfoLog(fragmentShader,512,NULL,infoLog);
//            qDebug()<<"error, shader,fragmentShader compilation failed"
<<infoLog;
//      }


      /* 链接 link shaders */
//
shaderProgram.addShaderFromSourceCode(QOpenGLShader::Vertex,vertexShaderSo
urce);
//
shaderProgram.addShaderFromSourceCode(QOpenGLShader::Fragment,fragmentShad
erSource);

/* 改为以下几行就可 */
QOpenGLShaderProgram shaderProgram;
shaderProgram.addShaderFromSourceFile(QOpenGLShader::Vertex,":/Shaders/sha
pes.vert");
shaderProgram.addShaderFromSourceFile(QOpenGLShader::Fragment,":/Shaders/s
hapes.frag");
bool success = shaderProgram.link();
if(!success)
{
    qDebug()<<shaderProgram.log();
}
shaderProgram.bind();
```

着色器位于GPU上，针对图形管线特定部分运行的小程序

本质是输入转输出的程序，且相互独立，仅通过输入和输出通信

一个典型的shader程序的典型结构

```
#version version_number
in type in_variable_name;
out type out_variable_name;
uniform type uniform_name;
void main(){
    out_variable_name = weird_stuff_we_processed;
}
/* 对于顶点着色器，输入变量为顶点属性 vertex attribute */
```

能够声明的顶点属性是有上限的，可以通过下面的代码获取

```
int nrAttributes;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS,&nrAttributes);   //  nrAttributes >=
16
```

OpenGL确保至少有16个包含4分量的顶点属性可用，但是有些硬件或许允许更多的顶点属性

类型：GLSL中包含C中大部分默认基础数据类型

int, float, double, uint, bool

GLSL也有两种容器：

1. 向量Vector
   vecn: the default vector of n floats，如vec2
   bvecn: a vector of n booleans
   ivecn: a vector ofn intergers
   uvecn: a vector of n unsigned intergers
   dvecn: a vector of n components
2. 矩阵Matrix

向量允许灵活的向量选择方式，叫做重组 swizzling

```
vec2 vect = vec2(0.5,0.7);
vec4 result = vec4(vec,0.0,0.0);
vec4 otherResult = vec4(result.xyz,1.0);   //  ==
vec4(result.x,result.y,result.z,1.0)
vec4 otherResult2 = vec4(result.xxx,1.0);   //  ==
vec4(result.x,result.x,result.x,1.0)
```

```
/*  输入输出
在发送方着色器声明一个输出 vertexColor
在接收方着色器声明一个类似的输入 vertexColor
当类型和名字都一致，OpenGL将把变量链接在一起（在链接程序对象时完成）

*/


/* 顶点着色器 */
#version 330 core
layout(location = 0) in vec3 aPos;
out vec4 vertexColor;
void main(){
      gl_Position = vec4(aPos.x,aPos.y,aPos.z,1.0f);
      vertexColor = vec4(0.5,0.0,0.0,1.0);
}


/* 片段着色器 */
#version 330 core
out vec4 FragColor;
in vec4 vertexColor;
void main(){
      FragColor = vertexColor;
}
```

顶点着色器接收的是一种特殊形式的输入，否则就会效率低下

从顶点着色器中直接接收输入，为了定义顶点数据该如何管理，可以使用 location指定输入变量，这样就可以在CPU上配置顶点属性，例如

layout(location = 0) 的layout这个标识，能把它链接到顶点数据

也可以省略 layout(location = 0) 说明符，并通过 glGetAttribLocation 在OpenGL代码中查询属性位置，但更推荐在顶点着色器中设置它们。这样更容易理解，也更高效。

```
/* 顶点着色器 */
#version 330 core
in vec3 aPos;    //  == layout(location = 0) in vec3 aPos;
void main(){
      gl_Position = vec4(aPos.x,aPos.y,aPos.z,1.0f);
}

/* 方法一：告知显卡如何解析 */
glVertexAttribPointer(0,3,GL_FLOAT,GL_FALSE,3*sizeof(float),(GLvoid*)0);
  // 因为 location = 0
```

```
/* 开启VAO属性 */
glEnableVertexAttribArray(0);   // 因为 location = 0

/* 方法二：layout(location = 5) */
int posLocation = shaderProgram.attributeLocation("aPos");
glVertexAttribPointer(posLocation,3,GL_FLOAT,GL_FALSE,3*sizeof(float),
(GLvoid*)0);
glEnableVertexAttribArray(posLocation);

/* 方法三：不写layout(location = 0) */
int posLocation = 4;
shaderProgram.bingAttributeLocation("aPos",posLocation);
// 再进行 shaderProgram.link()
```

---

uniform：另一种从CPU的应用，向GPU中的着色器发送数据的方式，

uniform是全局的，可以被任意着色器程序在任意阶段访问

```
/* 片段着色器 */
#version 330 core
out vec4 FragColor;
uniform vec4 ourColor;
void main(){
      FragColor = vertexColor;
}
```

如果声明了一个uniform却没有使用，编译器会默认移除这个变量，导致最后编译出的版本中不包含它，这可能导致bug

案例：

让矩形颜色随时间而变化

```
int timeValue = QTime::currentTime().second();
float greenValue = (sin(timeValue)/2.0f)+ 0.5f;
shaderProgram.setUniformValue("outColor",0.0f,greenValue,0.0f,1.0f);

/*
shaderProgram.setUniformValue("outColor",0.0f,greenValue,0.0f,1.0f);
是Qt进行了封装
不进行封装的话，需要写下面三行代码
*/
int vertexColorLocation = glGetUniformLoacation(shaderProgram,"ourColor");
glUseProgram(shaderProgram);
glUniform4f(vertexColorLocation,0.0f,greenValue,0.0f,1.0f);
```

OpenGL的核心是一个C库，所以它不支持类型重载，在函数参数类型不同的时候就要为其定义新的函数，glUniform是一个典型例子。

glUniform4f有一个特定的后缀，辨识设定的uniform的类型，

后缀	含义
f	函数需要一个float作为它的值
i	int
u	unsigned int

更多属性：将颜色数据加入到顶点数据中

```
float vertices[] = {    /*  颜色数据      */
      0.5f, 0.5f,   0.0f,   1.0f, 0.0f, 0.0f,     // top right
      0.5f, -0.5f, 0.0f,   0.0f, 1.0f, 0.0f, // button right 重合点1
      -0.5f, -0.5f, 0.0f, 0.0f, 0.0f, 1.0f,   // button left
      -0.5f, 0.5f, 0.0f,   0.5f, 0.5f, 0.5f,   // top left  重合点2
};

/* 顶点着色器 */
#version 330 core
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aColor;
out vec3 ourColor;
void main(){
      gl_Position = vec4(aPos,1.0f);
      outColor = aColor;
```

```
}

/* 片段着色器 */
#version 330 core
out vec4 FragColor;
in vec4 ourColor;
void main(){
      FragColor = vec4(ourColor,1.0f);
}
```

第三章 纹理

3.1 纹理映射

当需要给图形赋予真实的颜色的时候，不太可能使用前面的方法为每一个顶点指定一个颜色

通常采用纹理贴图

每一个顶点关联一个纹理坐标（texture coordinate），之后在图形的其它片段上进行片段插值（fragment interpolation）

告诉OpenGL如何对纹理采样

```
glVertexAttribPointer(2,2,GL_FLOAT,GL_FALSE,8*sizeof(float),(void*)
(6*sizeof(float)));
glEnableVertexAttribArray(2);
```

3.2 纹理单元

纹理环绕方式

纹理坐标的范围通常是从（0，0）到（1，1），

环绕方式	描述
GL_REPEAT	对纹理的默认行为，重复纹理图案
GL_MIRRORED_REPEAT	和GL_REPEAT一样，但每次重复图片是镜像放置的

环绕方式	描述
GL_CLAMP_TO_EDGE	纹理坐标会被约束在0到1之间，超出的部分会重复纹理坐标的边缘，产生一种边缘被拉伸的效果
GL_CLAPM_TO_BORDER	超出的坐标为用户指定的边缘颜色

```
/* 上下镜像，左右镜像 */
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_MIRRORED_REPEAT);

/* 超出的坐标为黄色 */
float borderColor[] = {1.0f,1.0f,0.0f,1.0f};
glTexParameterfv(GL_TEXTURE_2D,GL_TEXTURE_BORDER_COLOR,borderColor);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP_TO_BORDER);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP_TO_BORDER);
```

3.3 纹理过滤

纹理坐标不依赖于分辨率（Resolution），OpenGL需要知道怎样将纹理像素映射到纹理坐标

- 纹理坐标的精度是无限的，可以是任意的浮点值
- 纹理像素是有限的（图片分辨率）
- 一个像素需要一个颜色
- 所以采样就是，通过纹理坐标，问图片要纹理像素的颜色值

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);   /* 缩小时，就近采样，取最近的颜色 */
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);   /* 放大时，线性中和颜色 */
```

3.4 多级渐远纹理

简单来说，就是一系列的纹理图像，根据观察者和物体的距离，参考临界值，选择最合适物理的距离的那个纹理

OpenGL有一个glGenerateMipmaps()，可以产生多级渐远纹理

过滤方式	描述
GL_NEAREST_MIPMAP_NEAREST	使用最邻近的多级渐远来匹配像素大小，并使用临近插值进行纹理采样
GL_LINEAR_MIPMAP_NEAREST	使用最邻近的多级渐远纹理级别，并使用线性插值进行采样
GL_NEAREST_MIPMAP_LINEAR	在两个最匹配像素大小的多级渐远纹理之间进行线性插值，使用临近插值进行采样
GL_LINEAR_MIPMAP_LINEAR	在两个临界的多级渐远纹理之间使用线性插值，并使用线性插值进行采样

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
```

第四章 变换

4.1 向量

需求：需要改变物体的位置，如三维动画

现有解决方法：每一帧，改变顶点的值（所有的顶点）

每一个顶点用向量值表示，使用位移矩阵、缩放矩阵、旋转矩阵对所有的顶点进行操作（线性代数）

向量：具有方向direction和大小magnitude

4.2 矩阵

缩放、位移、旋转

弧度转角度：角度 = 弧度 *（180.0f/ PI）

角度转弧度：弧度 = 角度*（PI/180.0f）

4.3 坐标系统

顶点坐标需要经历五个坐标系统：

局部空间 local space

世界坐标 world coordinate

观察坐标 view coordinate

裁剪坐标 clip coordinate

屏幕坐标 screen coordinate

最重要的几个分别是模型model，观察view，投影projection三个矩阵

V（clip） = M（projection） · M（view） · M（model） · V（local）

正交投影：

透视投影：近处的东西小，远处的东西大

由投影矩阵创建的平截头体

齐次坐标

更多立方体

摄像机

OpenGL本身没有摄像机的定义，但我们可以通过把场景中的所有物体往相反方向移动的方式来模拟出摄像机，产生一种物体在移动的感觉。

要定义一个摄像机，需要它在世界空间中的位置、它所朝向的方向（反方向）、一个指向右侧的向量，以及一个从摄像机指向上方的向量。

实际上是要以摄像机的位置作为原点，创建一个具有三个相互垂直的单位轴的坐标系

第五章 Assimp 导入模型

（Open asset import library）

解析obj，fbx，flt格式，得到

场景Scene：所有场景/模型数据（材料和网格）都包含在场景对象中。场景对象也包含了场景节点的引用

Scene下的mMeshes数组储存了真正的Mesh对象，节点中的Mesh数组保存的只是场景中网格数组的索引

根节点root node：场景节点可能包含子节点（和其它节点一样），他会有一系列指向场景对象中mMeshes数组中储存的网络数据的索引

子节点child node

mesh对象：一个mesh对象本身包含了渲染所需要的所有相关数据，如顶点位置、法向量、纹理坐标、面Face和物体的材质

面Face：一个网格包含了多个面，面代表的是物体的渲染图元（三角形、方形、）

材料Material对象：一个网格也包含了一个Material对象，它包含了一些函数能让我们获取物体的材质属性，如颜色和纹理贴图（如漫反射和镜面光贴图）

数据全部都在mesh里面，一个mesh就是绘制的最小单元

5.1 Mesh类