### OpenGL总结

#### OpenGL基本介绍

OpenGL（Open Graphics Library）是用于[渲染](https://baike.baidu.com/item/%E6%B8%B2%E6%9F%93" \t "_blank)[2D](https://baike.baidu.com/item/2D)、[3D](https://baike.baidu.com/item/3D" \t "_blank)[矢量图形](https://baike.baidu.com/item/%E7%9F%A2%E9%87%8F%E5%9B%BE%E5%BD%A2)的跨[语言](https://baike.baidu.com/item/%E8%AF%AD%E8%A8%80)、[跨平台](https://baike.baidu.com/item/%E8%B7%A8%E5%B9%B3%E5%8F%B0)的[应用程序编程接口](https://baike.baidu.com/item/%E5%BA%94%E7%94%A8%E7%A8%8B%E5%BA%8F%E7%BC%96%E7%A8%8B%E6%8E%A5%E5%8F%A3)。OpenGL是一个状态机的工作方式，通过上下文（Context）来控制绘图流程。OpenGL相关三方库包括有**Glfw**（取代Glut、FreeGlut，用于创建OpenGL上下文和进行窗口操作）、**Glad**（继GL3W、GLEW之后，用于访问OpenGL规范接口）、**stb\_image**（取代Soil，用于纹理加载）、**glm**(用于矩阵运算)、**Assimp**(用于三维模型加载)、**FreeType**（用于字体文本加载）。

从3.3版本以后，编程模式分为核心（Core）和兼容（Compatibility）两种，其中Core模式不兼容废弃的接口函数（固定渲染管线相关接口），新版本编程规范推荐Core模式。图1展示可编程渲染管线模式的工作流程。

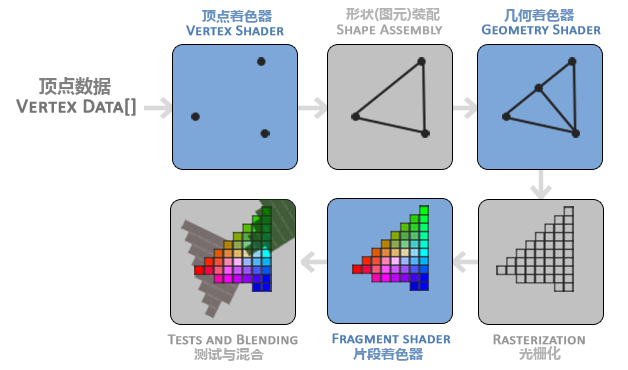


图1 可编程渲染管线流程

首先，我们以数组的形式传递3个3D坐标作为图形渲染管线的输入，用来表示一个三角形，这个数组叫做顶点数据(Vertex Data)；顶点数据是一系列顶点的集合。一个顶点(Vertex)是一个3D坐标的数据的集合。而顶点数据是用顶点属性(Vertex Attribute)表示的，它可以包含任何我们想用的数据（顶点坐标、颜色等等）。

图形渲染管线的第一个部分是顶点着色器(Vertex Shader)，它把一个单独的顶点作为输入。顶点着色器主要的目的是把3D的模型坐标转为3D裁剪坐标，同时顶点着色器允许我们对顶点属性进行一些基本处理。

图元装配(Primitive Assembly)阶段将顶点着色器输出的所有顶点作为输入（如果绘制GL\_POINTS，那么就是一个个顶点），并所有的点装配成指定图元的形状；图1中是一个三角形。

图元装配阶段的输出会传递给几何着色器(Geometry Shader)。几何着色器把图元形式的一系列顶点的集合作为输入（几何着色器和其他着色器不同，几何着色器每次运算的输入时一个顶点的集合，不是逐个顶点作为输入来进行计算，几何着色器在整个流程中只运行一次，而顶点和片段着色器会运行多次），它可以通过产生新顶点构造出新的图元来生成其他形状。图1例子中，它生成了另一个三角形。

几何着色器的输出会被传入光栅化阶段(Rasterization Stage)，这里它会把图元映射为最终屏幕上相应的像素，生成供片段着色器(Fragment Shader)使用的片段(Fragment)。在片段着色器运行之前会执行裁切(Clipping)。裁切会丢弃超出你的视图以外的所有像素，用来提升执行效率。

片段着色器的主要目的是计算一个像素的最终颜色，这也是所有OpenGL高级效果产生的地方。通常，片段着色器包含3D场景的数据（比如光照、阴影、光的颜色等等），这些数据可以被用来计算最终像素的颜色。

在所有对应颜色值确定以后，最终的对象将会被传到最后一个阶段，我们叫做Alpha测试和混合(Blending)阶段。这个阶段检测片段的对应的深度（和模板(Stencil)）值（存储在深度和模板缓冲中，和颜色缓冲统称为**帧缓冲**），通过投影片换后光栅化生成片段的坐标(gl\_FragCoord) z分量来判断这个像素是其它物体的前面还是后面，决定是否应该丢弃。这个阶段也会检查alpha值（alpha值定义了一个物体的透明度）并对物体进行混合(Blend)。所以，即使在片段着色器中计算出来了一个像素输出的颜色，在渲染多个三角形的时候最后的像素颜色也可能完全不同。

我们必须定义至少一个顶点着色器和一个片段着色器（因为GPU中没有默认的顶点/片段着色器），而几何着色器是可选的，通常使用默认着色器就行。

#### OpenGL绘图对象

OpenGL绘图常用的对象包括有：

* 顶点缓冲对象：Vertex Buffer Object，VBO

对应顶点相关缓冲数据，glBufferData接口调用指定VBO数据对应缓存时，会有CPU数据拷贝到GPU显存的过程。

* 顶点数组对象：Vertex Array Object，VAO

包含顶点属性（Vertex Attribute，一般一个VAO最多包含16个，顶点属性是用于指定VBO数据读取的规则）集合和EBO数据。VAO与VBO数据内存上是分离的，但是通过VBO数据🡪VAO顶点属性指定数据读取规则🡪顶点着色器指定顶点属性的引用🡪顶点着色器输入数据这样的流程，可以实现从顶点数据到顶点着色器输入的关联关系。

* 索引缓冲对象：Element Buffer Object，EBO

顶点数据的索引，包含在VAO中，用于顶点重复情况下减少内存消耗（比如绘制一个矩形，需要绘制两个三角形共6个顶点内存，使用EBO可以减少到4个）

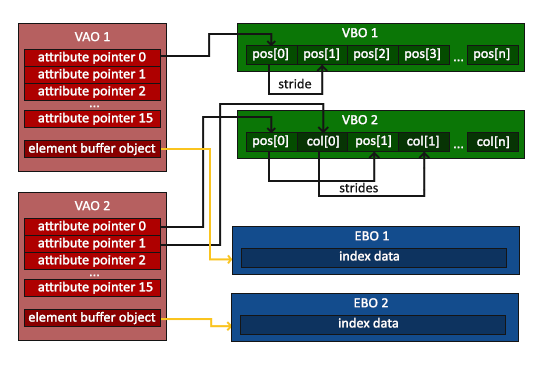


图2 OpenGL绘图对象VAO与VBO、EBO的关系

VBO、VAO、EBO对象关联关系如图2所示，一般情况下绘图需要配置并绑定VAO来进行绘图。绘图时根据VAO配置信息，将读取顶点属性数据传入顶点着色器，顶点着色器读取对应顶点数据作为输入，然后进行后续的渲染管线流程。

#### 坐标变换

将坐标变换为标准化设备坐标(Normalized Device Coordinate, NDC)，接着再转化为屏幕坐标的过程通常是分步进行的，也就是类似于流水线那样子。在流水线中，物体的坐标会变换到几个过渡坐标系中，其的优点在于，在这些特定的坐标系统中，一些操作或运算更加方便和容易。坐标转换流程如下图所示：



图3坐标变换流程

* 局部坐标是对象相对于局部原点的坐标，也是物体起始的坐标。
* 下一步是将局部坐标变换为世界空间坐标，世界空间坐标是处于一个更大的空间范围的。这些坐标相对于世界的全局原点，它们会和其它物体一起相对于世界的原点进行摆放。
* 接下来我们将世界坐标变换为观察空间坐标，使得每个坐标都是从摄像机或者说观察者的角度进行观察的。
* 坐标到达观察空间之后，我们需要将其投影到裁剪坐标。裁剪坐标会被处理至-1.0到1.0的范围内，并判断哪些顶点将会出现在屏幕上。

最后，我们将裁剪坐标变换为屏幕坐标，我们将使用一个叫做视口变换(Viewport Transform)的过程。视口变换将位于-1.0到1.0范围的坐标变换到由glViewport函数所定义的坐标范围内。最后变换出来的坐标将会送到光栅器，将其转化为片段。

结合查询资料，个人理解，顶点着色器中实现从局部坐标到裁剪坐标的转换过程，窗口裁剪的操作由OpenGL内部实现，后续经过光栅化插值，将片段数据送到片段着色器进行计算。

#### 着色器

着色器是运行在GPU上的一段程序。着色器使用一种叫GLSL的类C语言写成的，代码中支持结构体和数组的定义。着色器常用的为顶点着色器和片段着色器（顶点着色器会对每个顶点进行一次计算，入参是每个顶点的属性，同理片段着色器对每个片段进行计算，入参为每个片段的属性）。

顶点着色器的输入是VAO中一个或多个顶点属性（Vertex Attribute），每个顶点属性指定了对应的VBO数据。顶点属性中应当至少指出了顶点的位置属性(模型的坐标)。顶点着色器将这些位置信息转换为屏幕位置，以便可以正确的显示。顶点属性中可能还包括一些其它顶点属性，例如，顶点颜色或纹理UV坐标等。顶点着色器将这些作为输出（一些输出需要经过坐标变换或者其他计算处理），以便其可在光栅化单元进行插值，然后作为输入传递给片段着色器。

片段着色器将光栅化插值运算输出的片段作为输入，最终决定每个片段的颜色信息。

着色器中常用关键字包括有：

layout (location = 0) 对应VAO的顶点属性索引，顶点着色器使用

in/out 着色器输入/输出

uniform 外部程序传递给shader的变量

以颜色调节的片段着色器代码为例（原始的纹理数据已经是RGB数据，如果是YUV需要进行一次矩阵变换），相关代码如下：

#version 330 core

out vec4 FragColor; // 片段颜色，最终片段着色器的输出

in vec2 TexCoord; // 纹理坐标，顶点着色器从VBO中读取，并输出作为片段着色器输入

uniform sampler2D ourTexture; // 纹理数据，外部通过glTexImage2D来传入纹理数据

// 通过uniform定义，渲染前由外部传入着色器

uniform float brightness; // 亮度参数

uniform float contrast; // 对比度参数

uniform float saturation; // 饱和度参数

// 颜色调整算法，输入输出分别为调整前后的rgb颜色向量，rgb分量从0-255映射到0-1的范围内

vec3 colorAdjust(vec3 srcColor)

{

// 亮度调节，直接乘以亮度系数

vec3 destColor = srcColor \* brightness;

// 饱和度调整，和亮度参数组成的颜色向量进行插值（保证亮度不变）

float Luminance = 0.215 \* srcColor.r + 0.7154 \* srcColor.g + 0.0721 \*srcColor.b;

vec3 LuminanceColor = vec3(Luminance,Luminance,Luminance);

destColor = mix(LuminanceColor, destColor, saturation);

//对比度调整，理论是要和平均亮度进行线性插值，简化平均亮度为128来进行运算

vec3 avgColor = vec3(0.5,0.5,0.5);

destColor = mix(avgColor, destColor, contrast);

return destColor;

}

void main()

{

// 纹理采样，采样完是一个rgba的向量数据

vec4 tmp = texture(ourTexture, TexCoord);

// vec4.rgb是一个vec3的数据，代表采样的rgb分量，vec4.a代表alpha分量

FragColor = vec4(colorAdjust(tmp.rgb), tmp.a);

}

部分场景需要考虑调试着色器相关的输入输出是否正常，常用的工具gDebugger和RenderDoc（都是hook API的方式工作）等。在不借助工具情况，最直接的方式是通过gl\_FragColor输出对应变量参数，画面上用变量值代替片段rgb值，来判断变量是否正确。

部分场景需要拷贝渲染画面的颜色缓冲数据出来，可以借助PBO（Pixel Buffer Object，像素缓冲对象）来将颜色缓冲通过glReadPixels接口拷贝到临时VBO缓存中，通过map方式将buffer拷贝到内存。基本流程如下：

// 创建一个空的输出缓冲，缓存位于GPU上

GLuint outBuffer;

glGenBuffers(1, &outBuffer);

glBindBuffer(GL\_ARRAY\_BUFFER, outBuffer);

glBufferData(GL\_ARRAY\_BUFFER, screenWidth\*screenHeight\*3, NULL, GL\_STREAM\_COPY);

glBindBuffer(GL\_ARRAY\_BUFFER, 0);

// 窗体渲染循环

while (!glfwWindowShouldClose(window))

{

// 画面绘制工作

……

// 绑定VBO到PBO上，pack对应读帧缓冲操作，unpack对应写缓冲操作

glBindBuffer(GL\_PIXEL\_PACK\_BUFFER, outBuffer);

// 颜色缓冲读取数据，如果PBO绑定了缓冲，直接拷贝到缓冲上

glReadPixels(0, 0, screenWidth, screenHeight, GL\_RGB, GL\_UNSIGNED\_BYTE, NULL);

void\* data = glMapBuffer(GL\_PIXEL\_PACK\_BUFFER, GL\_READ\_WRITE);

if (data)

{

// data buffer处理工作

……

glUnmapBuffer(GL\_PIXEL\_PACK\_BUFFER);

}

// 交换缓冲，绘制画面

glfwSwapBuffers(window);

glfwPollEvents();

}

对于一些模糊、插值的算法，需要临近片段颜色等参数信息。比如在视频渲染进行纹理贴图的场景下，可以通过在片段着色器中对临近点纹理坐标进行纹理采样，然后执行计算流程。以模糊的场景为例，片段着色器如下：

#version 330 core

out vec4 FragColor; // 片段颜色，最终片段着色器的输出

in vec2 TexCoord; // 纹理坐标，顶点着色器从VBO中读取，并输出作为片段着色器输入

uniform sampler2D ourTexture; // 纹理数据，外部通过glTexImage2D来传入纹理数据

void main()

{

vec2 offsets[9] = vec2[](

vec2(-offset, offset), // 左上

vec2( 0.0f, offset), // 正上

vec2( offset, offset), // 右上

vec2(-offset, 0.0f), // 左

vec2( 0.0f, 0.0f), // 中

vec2( offset, 0.0f), // 右

vec2(-offset, -offset), // 左下

vec2( 0.0f, -offset), // 正下

vec2( offset, -offset) // 右下

);

float kernel[9] = float[](

1.0 / 16, 2.0 / 16, 1.0 / 16,

2.0 / 16, 4.0 / 16, 2.0 / 16,

1.0 / 16, 2.0 / 16, 1.0 / 16

);

vec3 sampleTex[9];

for(int i = 0; i < 9; i++)

{

// 根据纹理坐标偏移进行纹理采样

sampleTex[i] = vec3(texture(ourTexture, TexCoord.st + offsets[i]));

}

vec3 col = vec3(0.0);

for(int i = 0; i < 9; i++)

col += sampleTex[i] \* kernel[i];

// 片段颜色使用加权平均的颜色值

FragColor = vec4(col, 1.0);

}

对于一些3D模型的场景，如果是对整体场景进行算法处理，可以采用离屏渲染（自定义纹理作为渲染的颜色缓冲）技术，将整个场景画面渲染到纹理上，然后对这个纹理进行后处理，最后渲染到屏幕窗口上。

#### 文本渲染

一种方式是通过FreeType加载字体中字符，最终在opengl矩形中渲染每一个字符，拼在一起形成整个字符串的渲染。过程中会根据每个字符的数据和位置信息，计算字符的矩形顶点坐标，然后使用加载的字符Buffer来进行纹理渲染。

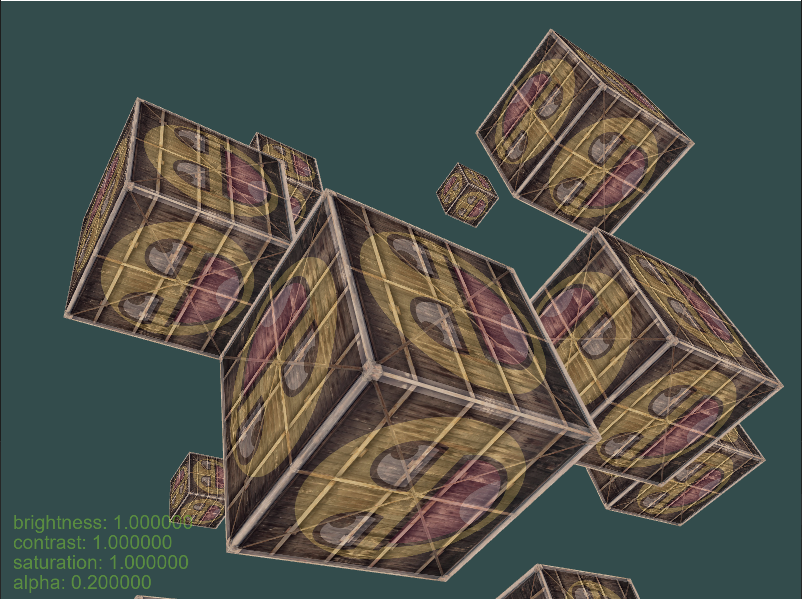


图4 字体渲染流程

如图4左所示，每个字符会渲染在每个对应矩形框内，每个绿色矩形框的位置对应的字符的纹理坐标，右图是字符Buffer通过纹理贴图后显示文本的效果（透明度0.5）。

#### 抗锯齿

OpenGL抗锯齿主要分两个部分：

* 纹理的抗锯齿

纹理抗锯齿属性主要通过纹理过滤的方式，主要包括临近过滤和线性过滤两个方式。临近过滤会选择中心点最接近纹理坐标的那个像素，线性过滤会基于纹理坐标附近的纹理像素，计算出一个插值。

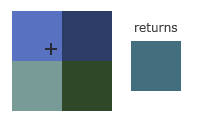
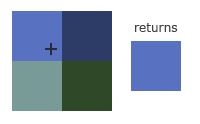


图5 临近过滤和线性过滤

在纹理放大的情况下，一般推荐使用线性过滤方式，使纹理更加光滑。

* 自绘物体的抗锯齿

与纹理采样不同，这个过程是关于光栅器采样导致的锯齿问题。OpenGL提供了一种多重采样抗锯齿（Multisample Anti-aliasing, MSAA)算法来应对这个问题。

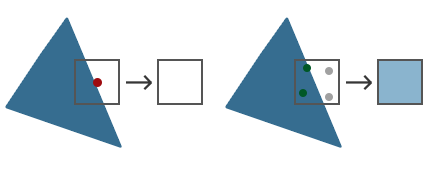


图6 MSAA算法实例

正常光栅采样每个像素中心只有一个采样点，最终对于每个像素来说，效果就是颜色有或者无的情况，效果不好。MSAA算法使用多个采样点，如图6每个像素有4个采样点，每个采样点包含一个颜色信息。对于图元覆盖的采样点就使用像素中心点的采样颜色代替，未覆盖的采样点不做修改，最终通过颜色平均，决定图元边缘片段像素的最终输出颜色，这种情况下锯齿效果比较好。

OpenGL内部默认开启MSAA算法，但是默认（不启用离屏渲染）使用屏幕的颜色缓冲。屏幕的默认颜色缓冲不使用多采样点，因此需要手动设置窗口开启多重采样缓冲（glfwWindowHint(GLFW\_SAMPLES, 4)）。

#### 光照

物体呈现的颜色是由物体反射的颜色决定，物体颜色的rgb的分量可以认为是物体对各个单色光的反射率。 因此光照下物体呈现的颜色简单就有如下计算公式:

glm::vec3 lightColor(0.33f, 0.42f, 0.18f);

glm::vec3 objectColor(1.0f, 0.5f, 0.31f);

FragColor = vec4(lightColor \* objectColor, 1.0);

实际环境中光照比较复杂，用于模拟光照的模型也比较多。其中在冯氏光照模型中，将光照分为三个分量，分别如下：

* 环境光照(Ambient Lighting)：即使在黑暗的情况下，世界上通常也仍然有一些光亮（月亮、远处的光），所以物体几乎永远不会是完全黑暗的。为了模拟这个，我们会使用一个环境光照常量，它永远会给物体一些颜色。
* 漫反射光照(Diffuse Lighting)：模拟光源对物体的方向性影响(Directional Impact)。它是冯氏光照模型中视觉上最显著的分量。物体的某一部分越是正对着光源，它就会越亮。
* 镜面光照(Specular Lighting)：模拟有光泽物体上面出现的亮点。镜面光照的颜色相比于物体的颜色会更倾向于光的颜色。

物体片段上展现的颜色是三个分量的叠加，使用过程中大部分环境光照和漫反射光照对于物体来说无差别，因此，提出来漫反射贴图和镜面贴图两个概念，用来表示漫反射光照下和镜面光照下物体本身的纹理颜色（具体计算可以参考：<https://learnopengl-cn.github.io/02%20Lighting/02%20Basic%20Lighting/>）。

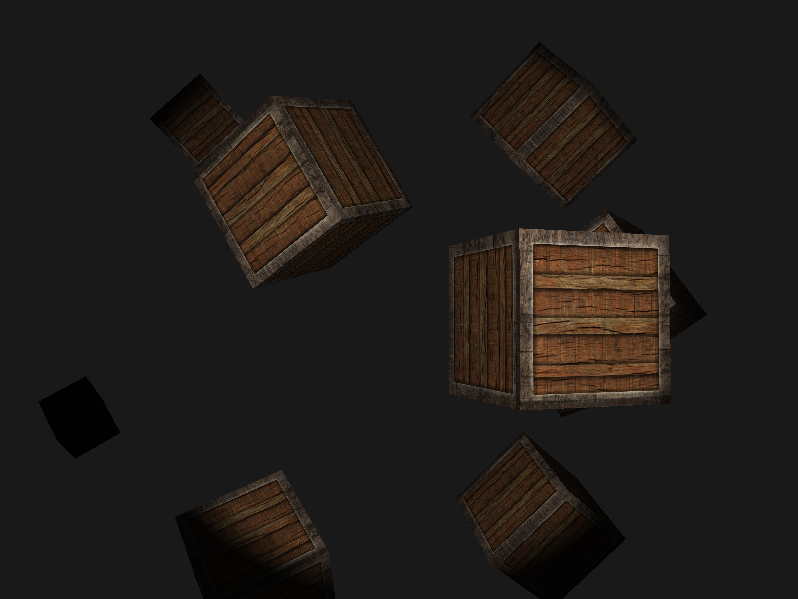


图7 光照的应用实例

参考链接：

Opengl教程: <https://learnopengl-cn.github.io/>