OpenGL基础

OpenGL基础

坐标系

- 顶点坐标系

以view的center为原点，z轴向屏幕上方的三维坐标系。x、y、z的取值范围是-1~1，左下角是(-1,-1,0)，右上角为(1,1,0)

Plain Text

```
                    (0,1,0)
                    Y
                    ^    Z
                    |   /
                    |  /
                    | /
                (0,0) |/
(-1,0,0)------------/---------- > X(1,0,0)
                    /|
                   / |
                  /  |
                 /   |
                (0,-1,0)
```

- 纹理坐标系

以图元左下角为原点的二维坐标系，横轴为S，纵轴为T。点的坐标一般用(U,V)表示，U、V的取值范围是0~1，左下角为(0,0)，右上角为(1,1)

Plain Text

```
T
^ (0,1)
|
|
|
|
|
|
|
|
|----------------------- > S(1,0)
(0,0)
```

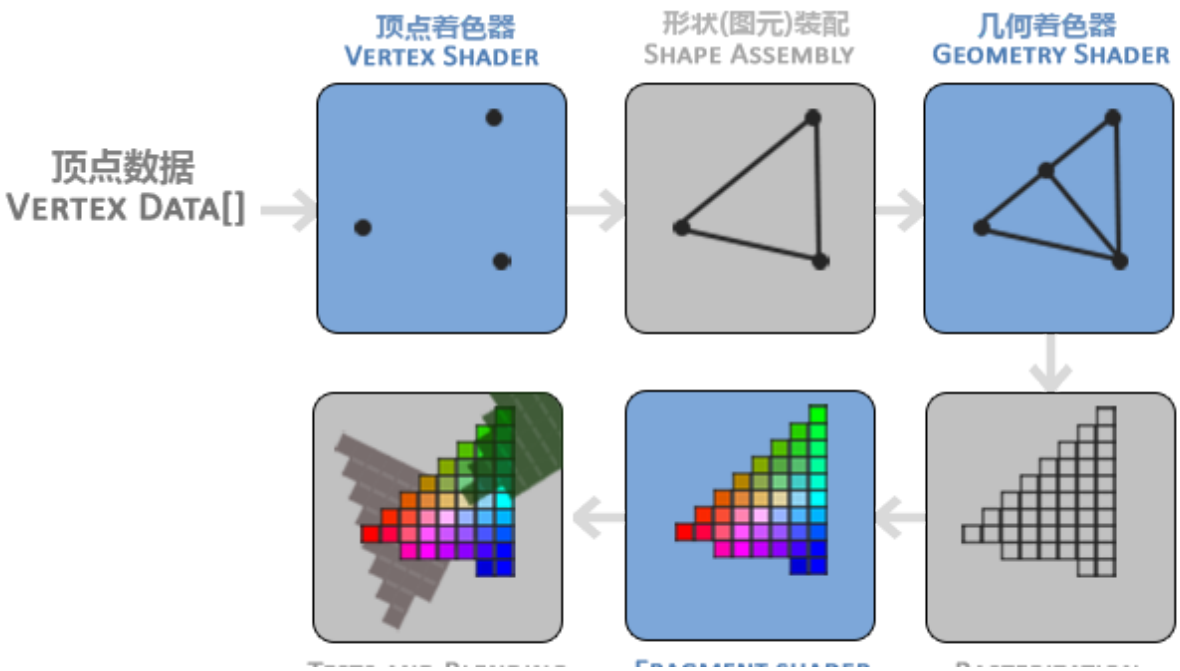
OpenGL对象的一般操作方式

C

```c
unsigned int objectId = 0;// 创建对象ID
glGenObject(1, &objectId);// 生成对象ID
glBindObject(GL_WINDOW_TARGET, objectId);// 绑定对象
glSetObjectOption(GL_WINDOW_TARGET, GL_OPTION_WINDOW_WIDTH, 800);// 设置对象参数
```

图形管线

图形管线处理流程图如下



主要步骤如下

- 初始化管线、顶点着色器和片段着色器
- 输入顶点数据
- 使用顶点着色器将顶点数据装配成图元
- 使用片段着色器进行颜色填充

缓存

OpenGL一部分运行在CPU上，一部分运行在GPU上，为了两个部分的数据交换，定义了缓存(Buffer)的概念。

缓存的使用分为7步

- 生成：生成缓存表示符 glGenBuffer()
- 绑定：绑定一个缓存 glBindBuffer()
- 缓存数据：从CPU的内存中将数据复制到缓存中 glBufferData()
- 启用或禁用缓存：glEnableVertexAttribArray()/glDisableVertexAttribArray()

- 设置指针：告知缓存数据类型及数据的偏移量 glVertexAttribPointer()
- 绘图：glDrawArrays()/glDrawElements()
- 删除：glDeleteBuffers()

图形上下文

Objective-C

```
self.context = [EAGLContext.alloc initWithAPI:kEAGLRenderingAPIOpenGLES2];
[EAGLContext setCurrentContext:_context];
```

顶点输入

将顶点坐标定义为一个由x,y,z构成的float数组，OpenGL ES只能支持渲染三角形，多边形需要由多个三角形组成

C

```
const GLfloat vertices[] = {
        0.0f,  0.5f, 0.0f,
        -0.5f, -0.5f, 0.0f,
        0.5f,  -0.5f, 0.0f
      };
```

GPU会创建内存储存顶点数据

可以通过顶点缓冲(Vertex Buffer Objects,VBO)管理这个内存

由于CPU的数据发送至显卡比较慢，使用这个对象可以一次性尽可能多的发送顶点数据至显卡的内存中

加载纹理

可以使用GLKit的GLKTextureLoader进行纹理绑定

Objective-C

```
NSString *imagePath = [NSBundle.mainBundle pathForResource:@"player"
ofType:@"png"];
UIImage *image = [UIImage imageWithContentsOfFile:imagePath];
GLKTextureInfo *textureInfo = [GLKTextureLoader
textureWithCGImage:image.CGImage options:@{GLKTextureLoaderOriginBottomLeft :
@(YES)} error:nil];
self.baseEffect = GLKBaseEffect.new;
self.baseEffect.texture2d0.name = textureInfo.name;
self.baseEffect.texture2d0.target = textureInfo.target;
```

或者使用OpenGL的方式进行纹理绑定

Objective-C

```
- (GLuint)setupTexture {
    //转化uiimage为cgimageref
    CGImageRef textureImage = [UIImage imageNamed:@"player"].CGImage;
    size_t width = CGImageGetWidth(textureImage);
    size_t height = CGImageGetHeight(textureImage);
    //绘制图片
    void *textureData = malloc(width * height * 4);
    CGContextRef textureContext = CGBitmapContextCreate(textureData, width, height, 8, width * 4, CGImageGetColorSpace(textureImage), kCGImageAlphaPremultipliedLast);
    //翻转坐标系
    CGContextTranslateCTM(textureContext, 0, height);
    CGContextScaleCTM(textureContext, 1, -1);
    //绘制
    CGContextDrawImage(textureContext, CGRectMake(0, 0, width, height), textureImage);
    //释放内存
    CGContextRelease(textureContext);
    //生成纹理
    GLuint textureID;
    glGenTextures(1, &textureID);
    glBindTexture(GL_TEXTURE_2D, textureID);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, (float)width, (float)height, 0, GL_RGBA, GL_UNSIGNED_BYTE, textureData);
    //设置映射方式
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glBindTexture(GL_TEXTURE_2D, 0);
    free(textureData);
    return textureID;
}
```

着色器

可以使用GLKit提供的着色器进行着色

也可以使用以下方式进行自定义着色器

顶点着色器

OpenGL Shading Language

```
attribute vec4 position;//输入值，传入一个四维向量，表示顶点位置
attribute vec2 textCoordinate;//输入值，传入一个二维向量，表示纹理位置
uniform mat4 rotateMatrix;//全局变量，传入一个四维矩阵，表示旋转矩阵
varying vec2 varyTextCoord;//varying变量，传入一个二维向量，包含纹理坐标
void main() {//着色器的入口函数
   varyTextCoord = textCoordinate;
   //计算旋转后的位置
   vec4 vPos = position;
   vPos = vPos * rotateMatrix;
   gl_Position = vPos;//设置顶点位置gl_Position的值
}
```

片段着色器

OpenGL Shading Language

```
precision mediump float;//着色器中浮点变量的默认精度
uniform sampler2D colorMap;//全局变量，传入一个2D纹理采样器，包含纹理的色彩数据
varying vec2 varyTextCoord;//varying变量，传入一个二维向量，包含纹理坐标
void main() {//着色器的入口函数
   gl_FragColor = texture2D(colorMap, varyTextCoord);//设置输出颜色gl_FragColor的值，由纹理的色彩和纹理坐标构成
}
```

编译着色器

对于着色器文件，需要运行时动态编译得到着色器对象

Objective-C

```
- (GLuint)compileShader:(NSString *)filePath withShaderType:(GLenum)shaderType
{
    //读取文件
    NSString *shaderString = [NSString stringWithContentsOfFile:filePath
encoding:NSUTF8StringEncoding error:nil];
    const GLchar* source = (GLchar *)shaderString.UTF8String;
    //创建着色器
    GLuint shader = glCreateShader(shaderType);
    //将着色器源码加载到着色器上
    glShaderSource(shader, 1, &source, NULL);
    //运行时编译着色器
    glCompileShader(shader);
    //检查是否编译成功
    GLint compileSuccess;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compileSuccess);
    if (compileSuccess == GL_FALSE) {//输出错误信息
        GLchar messages[256];
        glGetShaderInfoLog(shader, sizeof(messages), 0, &messages[0]);
        NSString *messageString = [NSString stringWithUTF8String:messages];
        NSLog(@"%@", messageString);
        return shader;
    }
    return shader;
}
```

着色器对象必须要链接到一个管线后，才能绘制图形

Objective-C

```
- (void)setupProgram {
    //创建一个管线
    self.program = glCreateProgram();
    //生成一个顶点着色器
    NSString *vertFile = [NSBundle.mainBundle pathForResource:@"Shader" ofType:@"vsh"];
    GLuint vertShader = [self compileShader:vertFile withShaderType:GL_VERTEX_SHADER];
    //生成一个片段着色器
    NSString *fragFile = [NSBundle.mainBundle pathForResource:@"Shader" ofType:@"fsh"];
    GLuint fragShader = [self compileShader:fragFile withShaderType:GL_FRAGMENT_SHADER];

    //将两个着色器挂载到管线上
    glAttachShader(self.program, vertShader);
    glAttachShader(self.program, fragShader);

    //链接Program
    glLinkProgram(self.program);
    //检查是否链接成功
    GLint linkSuccess;
    glGetProgramiv(self.program, GL_LINK_STATUS, &linkSuccess);
    if (linkSuccess == GL_FALSE) {//输出错误信息
        GLchar messages[256];
        glGetProgramInfoLog(self.program, sizeof(messages), 0, &messages[0]);
        NSString *messageString = [NSString stringWithUTF8String:messages];
        NSLog(@"%@", messageString);
    } else {
        NSLog(@"link OK");
        //执行Program
        glUseProgram(self.program);
    }
    //释放两个着色器 参考https://www.jianshu.com/p/bf1aac8bda9a
    if (vertShader) {
        glDetachShader(self.program, vertShader);//解除绑定
        glDeleteShader(vertShader);//删除着色器
    }
    if (fragShader) {
        glDetachShader(self.program, fragShader);//解除绑定
        glDeleteShader(fragShader);//删除着色器
    }
}
```

CAEAGLLayer

GLSL下需要绑定CAEAGLLayer才能输出图像

Objective-C

```
- (void)setupCAEAGLLayer {
    self.eagLayer = (CAEAGLLayer *)self.layer;
    //设置缩放比例
    self.eagLayer.contentsScale = UIScreen.mainScreen.scale;
    self.eagLayer.opaque = YES;
    self.eagLayer.drawableProperties = @{
        kEAGLDrawablePropertyRetainedBacking : @(NO),
        kEAGLDrawablePropertyColorFormat : kEAGLColorFormatRGBA8,
    };
}

- (void)destoryRenderAndFrameBuffer {
    if (_renderBuffer) {
        glDeleteRenderbuffers(1, &_renderBuffer);
        _renderBuffer = 0;
    }
    if (_frameBuffer) {
        glDeleteFramebuffers(1, &_frameBuffer);
        _frameBuffer = 0;
    }
}

- (void)setupRenderAndFrameBuffer {
    //绑定渲染缓存到layer上
    glGenRenderbuffers(1, &_renderBuffer);
    glBindRenderbuffer(GL_RENDERBUFFER, _renderBuffer);
    [_context renderbufferStorage:GL_RENDERBUFFER fromDrawable:self.eagLayer];

    //将渲染缓存绑定到帧缓存上
    glGenFramebuffers(1, &_frameBuffer);
    glBindFramebuffer(GL_FRAMEBUFFER, _frameBuffer);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
GL_RENDERBUFFER, _renderBuffer);
}
```

设置OpenGL

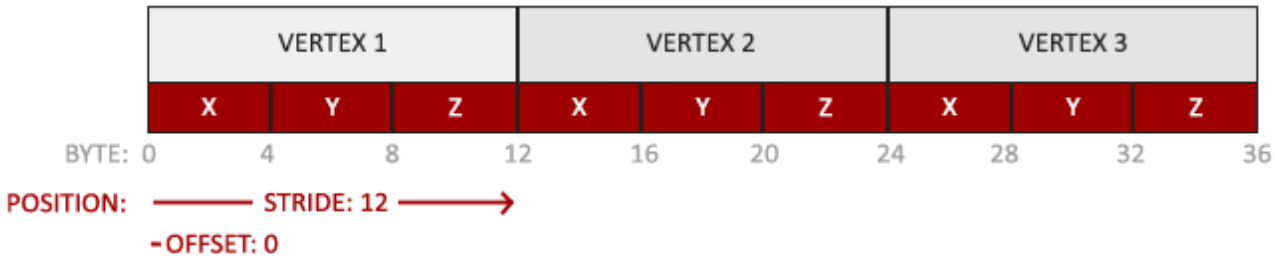
Objective-C

```
- (void)setupGL {
    glClearColor(0, 1.0, 0, 1.0);
    glClear(GL_COLOR_BUFFER_BIT);

    //设置混合模式，去除透明底部
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    CGFloat scale = UIScreen.mainScreen.scale;
    glViewport(self.frame.origin.x * scale, self.frame.origin.y * scale, self.frame.size.width * scale, self.frame.size.height * scale);
}
```

解析顶点数据

顶点着色器允许指定顶点的输入格式，所以需要指定OpenGL解析顶点数据的方式

顶点数据在内存中的状态如下图



并且，不同类型的变量使用的方式不一样

Objective-C

```
//生成缓存标识符
GLuint attrBuffer;
glGenBuffers(1, &attrBuffer);
//绑定缓存
glBindBuffer(GL_ARRAY_BUFFER, attrBuffer);
//将数据存入缓存
glBufferData(GL_ARRAY_BUFFER, sizeof(attrArray), attrArray, GL_DYNAMIC_DRAW);

//获取position变量 attribute
GLuint position = glGetAttribLocation(self.program, "position");
//传入position变量值
glVertexAttribPointer(position, 3, GL_FLOAT, GL_FALSE, sizeof(GLfloat) * 5, (GLfloat *)NULL + 0);
//启用缓存
glEnableVertexAttribArray(position);

//获取colorMap变量 uniform
GLuint colorMap = glGetUniformLocation(self.program, "colorMap");
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, textureID);
glUniform1i(colorMap, 0);//将colorMap赋值为GL_TEXTURE0，GL_TEXTURE0对应值为0
```

glVertexAttribPointer函数的参数非常多，下面逐一说明:

- GLuint indx:指定变量
- GLint size：指定变量的大小
- GLenum type：指定变量的类型
- GLboolean normalized：定义变量是否被标准化(Normalize)。如果设置为GL_TRUE，所有数据都会被映射到0（对于有符号型signed数据是-1）到1之间。因为我们传入的数据就是标准化数据，所以我们把它设置为GL_FALSE
- GLsizei stride：设置连续的顶点属性组之间的间隔。
- const GLvoid *ptr：表示位置数据在缓冲中起始位置的偏移量(Offset)。

进行绘制

Objective-C

```
glDrawArrays(GL_TRIANGLES, 0, 3);
```

参数说明：

- GLenum mode：绘制方式,这里是绘制三角形，所以选择GL_TRIANGLES，除GL_TRIANGLES之外，还可以选择复用顶点的方式GL_TRIANGLE_STRIP和GL_TRIANGLE_FAN
- GLint first：从数组缓存中的哪一位开始绘制，一般为0。
- GLsizei cout：数组中顶点数据的数量。

Objective-C

```
1  [self.context presentRenderbuffer:GL_RENDERBUFFER];
```

Origin Link: https://wepie.yuque.com/tcsdzz/ios_team/pxtbpd