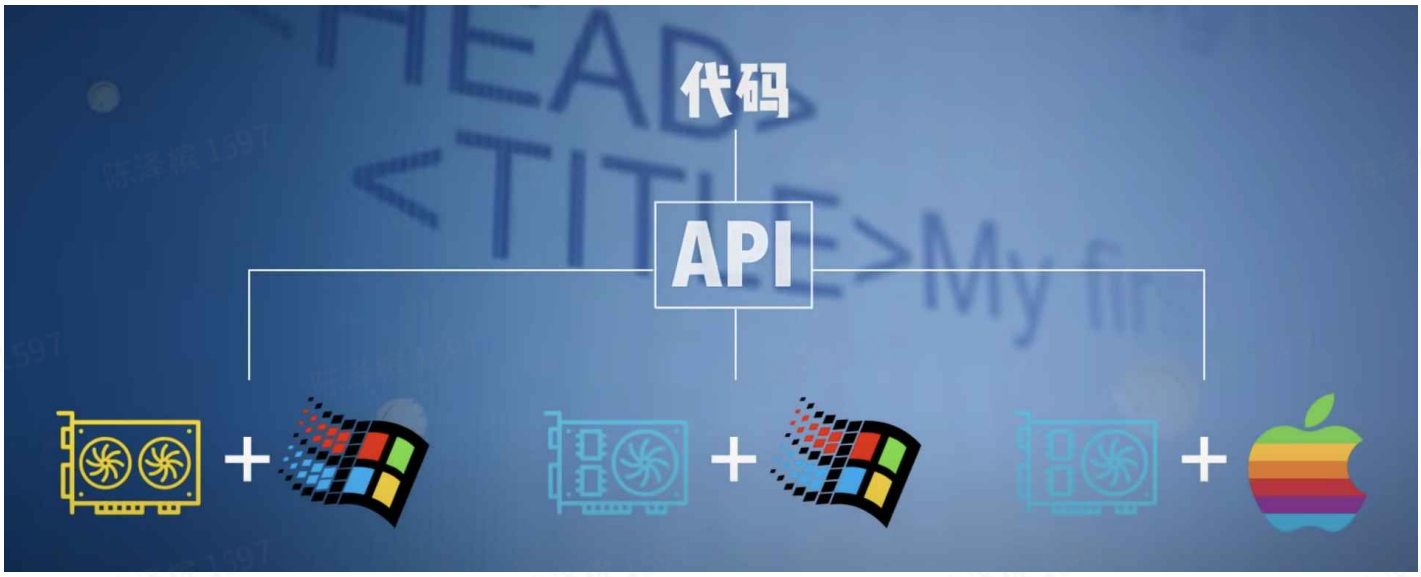
OpenGL入门介绍

OpenGL简史

OpenGL诞生



OpenGL还没有诞生的时候，图形应用的开发缺少标准，基本上不同的图形硬件都对应一套代码，对开发者特别不友好。

在上世纪90年代，SGI（Silicon Graphics, Inc.硅谷图形公司）在高端图形领域处于领先地位，我们熟悉的《侏罗纪公园》（1993年上映）就是使用 SGI的显卡渲染，有专利性的图形API，叫做IRIS GL。后来考虑到可移植性的重要，他们决定对IRIS GL进行修改，并把它当作一种开放标准，这就是1992年6月所发布的OpenGL 1.0版本，并且同年成立OpenGL架构评审委员会，即OpenGL ARB（Architecture Review Board），初始成员包括康佰（Compaq），IBM，Intel，微软 以及DEC，后来不断有其他公司加入，包括 HP，nVidia，ATI等等。

微软的DirectX（简称DX）

DirectX是一个合集，包含了Direct3D、Direct2D....
早在1995年微软发布DirectX 1.0的时候就已经包含了Direct3D，但并没有引起业界足够的重视。直到1997年8月发布的DirectX 5.0，对Direct3D进行了重大升级，提供了立即模式和保留模式两种编程模式，业界终于看到了微软对3D API的野心。

立即模式和保留模式的区别：

- 立即模式和保留模式是两套截然不同的API。
- 还没出DirectX 5.0的时候，DX是显卡驱动的接口层，数据流向是DX->显卡驱动->GPU；这时候的DX只有一种模式，就是保留模式，也就是只有一种API
- DirectX 5.0提供了直接访问GPU的接口，所以立即模式指的就是通过调用低级的API，直接操作硬件

到2000年12月Direct3D 8.0率先支持了可编程渲染管线，当时的顶点着色器和片元着色器都需要使用汇编语言编写；

两年之后的Direct3D 9.0就支持了高级着色器编程语言：HLSL。

OpenGL PK DirectX

可编程渲染管线的高速发展和OpenGL ARB的低效率形成了鲜明的对比！在这段高速发展期，OpenGL落伍了。支持高级Shader编程语言的OpenGL 2.0直到2004年9月才推出，足足比Direct3D 9.0晚了将近两年

Direct3D晚于OpenGL登场，却可以后发制人成为今天在PC游戏领域上的霸主；OpenGL曾经对Direct3D形成压倒性的优势，但由于开发上的不给力，如今已经转向专业图形领域发展，PC游戏上鲜见踪影。

OpenGL现状

2006年7月31日，OpenGL ARB宣布将OpenGL标准的控制权转交给Khronos Group（**科纳斯组织**）。Khronos Group是一个非盈利组织，它和OpenGL ARB的成员有很多交集。OpenGL的嵌入式版本的子集OpenGL ES，其1.0于2003年7月28日由该组织发布。OpenGL ES后来成为了移动端图形标准，也使得OpenGL家族焕发了第二次青春，直到2014年新的挑战者Metal的出现。

同年，微软发布DirectX 12（至今最新版本）；15年Khronos Group宣布了Vulkan API，被称为OpenGL的继任者。

Khronos Group是OpenGL的维护者，为啥还要发布Vulkan呢？

- 成也萧何败也萧何，OpenGL对软件开发者特别友好，也就代表它对硬件厂商并不友好。这在早期可能不是太大的问题，但随着GPU越来越强大、越来越复杂，GL驱动这一层也越来越厚。OpenGL显示驱动要维护所有的状态，对API调用进行各种检查，导致Draw Call的额外负担过重。
- 由于被设计为一个整体的状态机，所以OpenGL API不支持多线程。在CPU走向多核之后，并行编程越来越重要，这就成了一个明显的缺点。

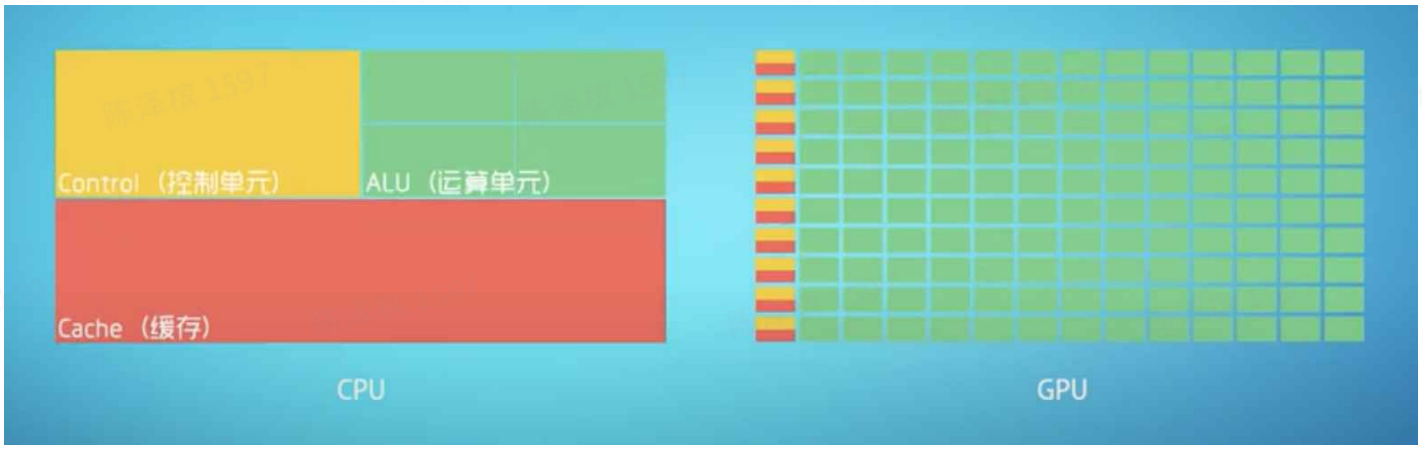
总结OpenGL

OpenGL是在上世纪90年代诞生的一套标准化的图形接口规范，由SGI发布，解决当时存在的两个痛点：

- 统一API，显卡商针对接口规范实现显卡驱动，开发者不需要知道内部实现细节，只需要知道接口规范，大大减轻了上层开发者的负担
- 跨平台

硬件介绍

CPU和GPU的区别



上面这个图，就是CPU和GPU的内部结构简图，从图上我们发现，虽然CPU和GPU都有控制、缓存和运算单元，但是因为最初设计目的的不同，所以他们各个单元的占比也不一样。

CPU的缓存占了很大的一块，GPU是运算单元占了大部分，因为CPU是通用计算，他要计算各种类型的数据，所以CPU的控制单元和缓存单元主要是负责一些控制和数据转发的功能，再就是用来放一些已经计算完成的数据或者是后面马上要用到的数据。

但是GPU的控制和缓存单元，主要是合并和转发数据。

比如GPU的运算单元，有100个运算单元需要用到同样的数据，这时候GPU的控制和缓存单元就会把大家的需求合并到一起，然后再去内存里面拿，拿回来之后再统一转发给大家。

最后就是CPU和GPU的运算单元，CPU的运算单元拥有非常强大的逻辑运算能力，既能做奥数题也能做加减法；而GPU的运算单元虽然没有那么强大的逻辑运算能力，但是在面对海量的，简单，单一的加减运算，比如10万以内的加减运算，显然GPU的速度要快于CPU。

所以GPU就像工厂的流水线，工作量非常大，但是没有太多技术含量，简单而重复。

说回图形这块，GPU主要是负责绘制，CPU也可以绘制，但是效率没有GPU高；一般硬件绘制就是指GPU绘制，软件绘制指CPU绘制

显卡相关

- 显卡的主要组成硬件就是GPU+显存
- 显存：比起运行内存，显存离GPU更近，意味着GPU从显存读存数据更快
- 显卡驱动：显卡驱动是OpenGL的内核层（也就是OpenGL的具体实现），上层就是OpenGL接口层了

OpenGL基本知识点

纹理和图片的区别

纹理（Texture）这个词来自OpenGL，使用范围更窄。
图片（Image）则是一个更广泛的称呼。
从OpenGL的角度看，纹理是显卡中一段连续的内存，可以用来存储图片数据，也可以用来存储计算过程中的中间结果等任意数据。
图片则是内存中的像素数组。
我们注意到，显卡有自己的内存，称为**显存**；而人们常说的**内存**（RAM），则是计算机的主存，显卡不能直接访问。
狭义地讲，纹理是存在于显存的，可以存放任意数据；图片是存在于内存的，内容是像素数组。

把图片加载到纹理的代码

```
// GLuint可以理解为指针变量，标识
GLuint *texID;
// 生成纹理的数量
glGenTextures(1, (GLuint*)&texID);
// 告诉OpenGL下面代码中对2D纹理的任何设置都是针对索引为textID的纹理的
glBindTexture(GL_TEXTURE_2D, texID);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
// 将imagedata加载到纹理中
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE,
imagedata);
```

OpenGL状态机

在 OpenGL 的世界里，大多数元素都可以用状态来描述，比如：

- 颜色、纹理坐标、光源的各种参数…
- 是否启用了光照、是否启用了纹理、是否启用了混合、是否启用了深度测试…
- …
- OpenGL会保持状态，比如我们调用`glClearColor (GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha)`函数去设置背景颜色，OpenGL会记住这个状态，也就是说，之后我们绘制的所有物体都会使用这个颜色，直到我们将其设置为其他颜色
- 理解了状态机这个概念，我们再来看 OpenGL ES 提供的 API，就会非常明了，因为OpenGL 当中很多 API，其实仅仅是向 OpenGL 这个状态机传数据或者读数据。

```
783 /* OpenGL ES 2.0 */
784
785 GL_API void             GL_APIENTRY glActiveTexture (GLenum texture) OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
786 GL_API void             GL_APIENTRY glAttachShader (GLuint program, GLuint shader)   OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
787 GL_API void             GL_APIENTRY glBindAttribLocation (GLuint program, GLuint index, const GLchar* name)   OPENGLES_DEPRECATED(ios(3.0, 12.0),
        tvos(9.0, 12.0));
788 GL_API void             GL_APIENTRY glBindBuffer (GLenum target, GLuint buffer) OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
789 GL_API void             GL_APIENTRY glBindFramebuffer (GLenum target, GLuint framebuffer)   OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
790 GL_API void             GL_APIENTRY glBindRenderbuffer (GLenum target, GLuint renderbuffer)   OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
791 GL_API void             GL_APIENTRY glBindTexture (GLenum target, GLuint texture) OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
792 GL_API void             GL_APIENTRY glBlendColor (GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha)   OPENGLES_DEPRECATED(ios(3.0, 12.0),
        tvos(9.0, 12.0));
793 GL_API void             GL_APIENTRY glBlendEquation (GLenum mode)   OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
794 GL_API void             GL_APIENTRY glBlendEquationSeparate (GLenum modeRGB, GLenum modeAlpha)   OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0,
        12.0));
795 GL_API void             GL_APIENTRY glBlendFunc (GLenum sfactor, GLenum dfactor) OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
796 GL_API void             GL_APIENTRY glBlendFuncSeparate (GLenum srcRGB, GLenum dstRGB, GLenum srcAlpha, GLenum dstAlpha)
        OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
797 GL_API void             GL_APIENTRY glBufferData (GLenum target, GLsizeiptr size, const GLvoid* data, GLenum usage) OPENGLES_DEPRECATED(ios(3.0,
        12.0), tvos(9.0, 12.0));
798 GL_API void             GL_APIENTRY glBufferSubData (GLenum target, GLintptr offset, GLsizeiptr size, const GLvoid* data)
        OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
799 GL_API GLenum           GL_APIENTRY glCheckFramebufferStatus (GLenum target)   OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
800 GL_API void             GL_APIENTRY glClear (GLbitfield mask) OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
801 GL_API void             GL_APIENTRY glClearColor (GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha) OPENGLES_DEPRECATED(ios(3.0, 12.0),
        tvos(9.0, 12.0));
802 GL_API void             GL_APIENTRY glClearDepthf (GLclampf depth) OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
803 GL_API void             GL_APIENTRY glClearStencil (GLint s) OPENGLES_DEPRECATED(ios(3.0, 12.0), tvos(9.0, 12.0));
```

OpenGL上下文（Context）

应用程序调用任何OpenGL的指令之前，首先创建一个OpenGL的上下文。这个上下文是一个非常庞大的状态机，保存了OpenGL中的各种状态，这也是OpenGL指令执行的基础。

调用OpenGL的函数，本质上就是修改上下文的状态

一个窗口创建一个上下文

一个GPU硬件服务于多个上下文，这个有点类似于客户端和服务器的关系

不能多线程同时操作一个上下文，所以一般的做法的创建一个绘制线程，专门去做这件事

上下文中的纹理是可以共享的，这样可以避免重复资源的绘制

顶点数据、顶点数组、顶点缓冲区

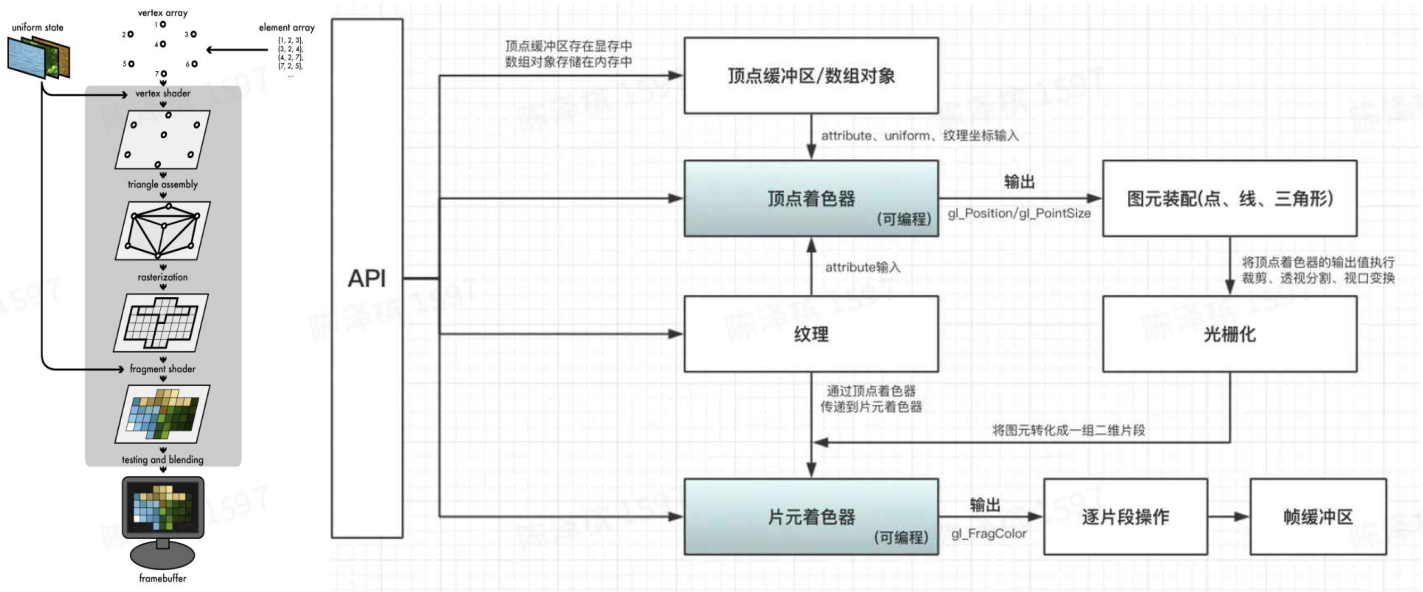
顶点数据：一系列顶点的集合

我们将顶点数组的数据，称为顶点数据；这些顶点数据是由GPU处理的；
顶点数组是存在内存中，如果在显存中划分一块区域来存在顶点数组，那么这块区域我们就将其称为顶点缓冲区。
GPU去顶点缓冲区读取顶点数据就更快
比如我们绘制一个三角形，往图形渲染管线传递含有三个顶点元素的顶点数组
那么为什么三个顶点就是三角形呢，不能是三个点，一条直线？
所以这里还需要指定图元类型，OpenGL才知道是画线，还是画三角形

着色器

着色器全称着色器程序，是运行在 GPU 中负责渲染算法的一类总称。相对地，我们通常写的代码是执行在 CPU 中的
GLSL（OpenGL Shading Language）是在 OpenGL 对应的着色器语言

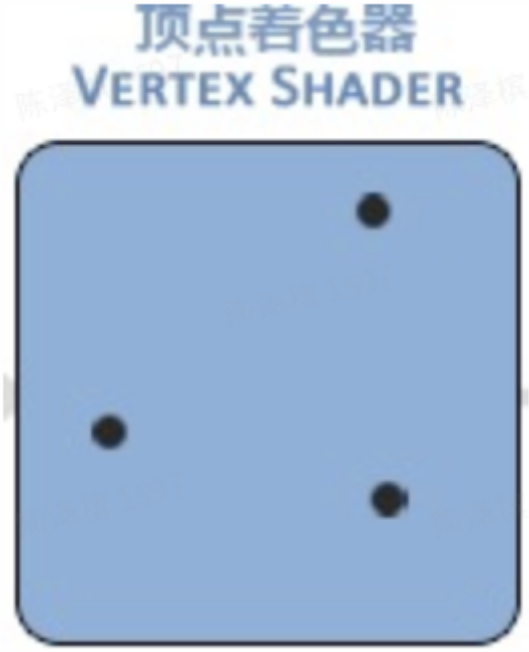
OpenGL ES渲染管线



所谓的渲染管线，实际上就是渲染过程流水线，指的不是具体某一样东西，而是一个流程。

- 将物体3D坐标转变为屏幕空间2D坐标
- 为屏幕每个像素点进行着色
- 这是显卡执行的、从几何体到最终渲染图像的、数据传输处理计算的过程。
- 管线流程：
- 顶点数据（Vertices） > 顶点着色器（Vertex Shader） > 图元装配（Assembly） > 光栅化（Rasterization） > 片元着色器（Fragment Shader） > 逐片段处理（Per-Fragment Operations） > 帧缓冲（FrameBuffer）>再经过双缓冲的交换（SwapBuffer）>渲染内容就显示到了屏幕上

顶点着色器

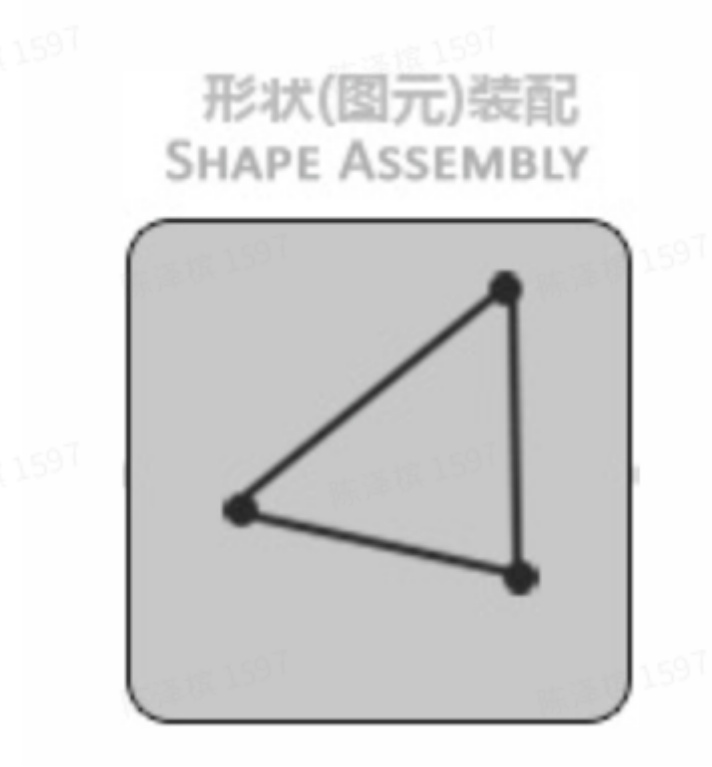


首先顶点着色器是一个程序源代码或者可执行文件，具体实现功能就是将顶点进行模型变换、视变换、投影变换、光照处理

顶点着色器接受单个顶点的输入，而顶点着色器这个过程才是接收顶点数组的输入

```
// 这个函数代表顶点着色器处理所有顶点的阶段
void vertexHandle(Vector list)
{
    for (...) {
        // 这个函数代表顶点着色器
        vertex_shader(顶点)
    }
}
```

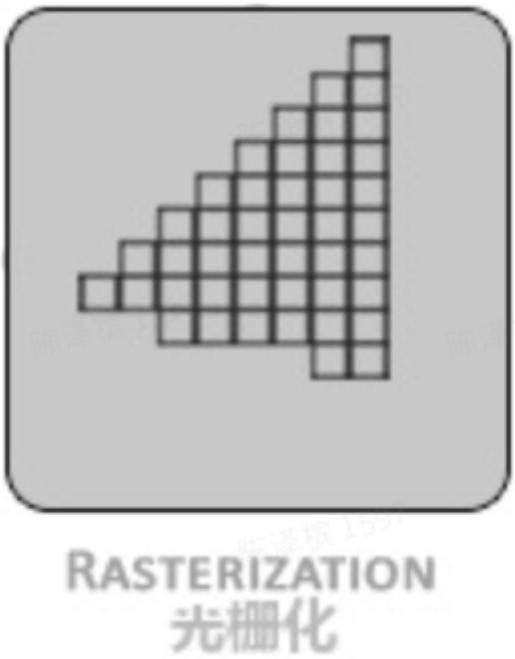
图元装配



先介绍下什么是图元
基本图形元素，简称图元。
图元的类型有点、直线、线条、三角形、四边形….

顶点着色器的下一个阶段是图元装配
将顶点数据计算成一个个图元，在这阶段会执行裁剪，透视分割，视口变换操作后进入光栅化阶段。

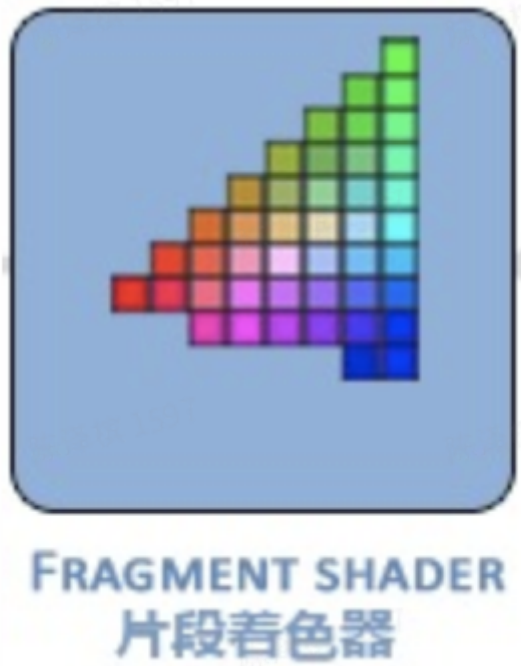
光栅化



屏幕显示的画面是由像素组成的，而三维物体是由点线面组成的。要让点线面变成屏幕上显示的像素，就需要光栅化这个过程。
那么光栅化这个过程，具体就是将上个阶段“图元装配”产生的图元转换成片元

> 我们先这样理解片元，大家都知道像素是什么，那么片元就是像素的前身，片元到像素需要经历深度测试、透明度测试、模板测试

片元着色器



片元着色器又叫片段着色器或像素着色器
片元着色器跟顶点着色器一样，一次只能处理一个片元
片元着色器业务：

- 计算颜色
- 获取纹理值
- 往像素点中填充颜色值(纹理值/颜色值)
- 它可以用于图片/视频/图形中每个像素的颜色填充(比如给视频添加滤镜,实际上就是将视频中每个图片的像素点颜色填充进行修改.)

模板缓冲区

模板缓冲区的一个应用就是限制屏幕上可显示的区域，举个例子，如果你要存储一个形状奇特的挡风玻璃，你只需要模板缓冲区存储这个挡风玻璃的形状，接着绘制整个屏幕，只有挡风玻璃所在区域中的内容会显示出来，这就是模板缓冲区的功能。

图形流水线中有一个叫模板测试的流程，作用就是限制绘制的图元区域，其做法是按照窗口宽高创建一个矩阵，矩阵由0,1组成，其中由1组成的区域代表相匹配的图元需要提交到后续流程进行测试和绘制，而由0组成的区域的片元则直接被丢弃，起到一个筛选作用，而这个0,1数值矩阵所在的显存区域则称为模版缓冲区。

深度缓冲区

如果两个不同的片段具有相同的xy坐标值，那么我们可以根据这两个片段的z值选择性丢弃，因为z值更大的那个片段实际上被“遮挡”了，GPU则可以根据z值判断是否需要渲染，这就提高了效率，这在图形学中叫做z消隐。

而z值保存的显存区域就是深度缓冲区

而比较判断相同xy坐标的片段（经过顶点光栅化，插值贴图着色后片段）的z值的过程叫做深度测试，深度测试成功，则渲染新像素，失败则丢弃。如果有透明度，那么涉及另一个概念叫颜色混合

颜色缓冲区

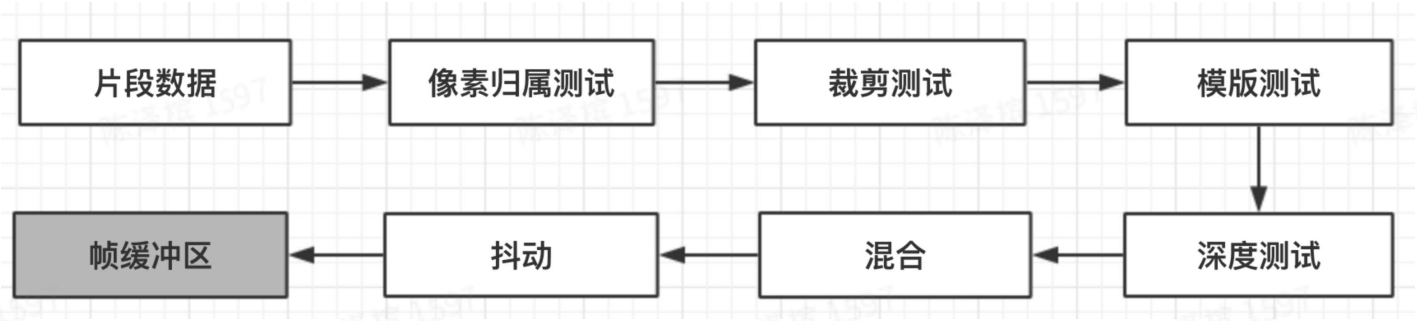
颜色缓冲区就是最终在显示屏硬件上显示颜色的GPU显存区域了，这个缓冲区储存了每帧更新后的最终颜色值，图形流水线经过一系列测试，包括片段丢弃、颜色混合等，最终生成的像素颜色值就储存在这里，然后提交给显示硬件显示。

帧缓冲区

前面介绍了模板缓冲区、深度缓冲区、颜色缓冲区，把这几种缓冲结合起来就叫做帧缓冲，它被储存于内存中。

帧缓存是接收渲染结果的缓冲区，为GPU指定存储渲染结果的区域。它存储着 OpenGL ES 绘制每个像素点最终的所有信息：颜色，深度和模板值。更通俗点，可以理解成存储屏幕上最终显示的一帧画面的区域。

逐片段操作



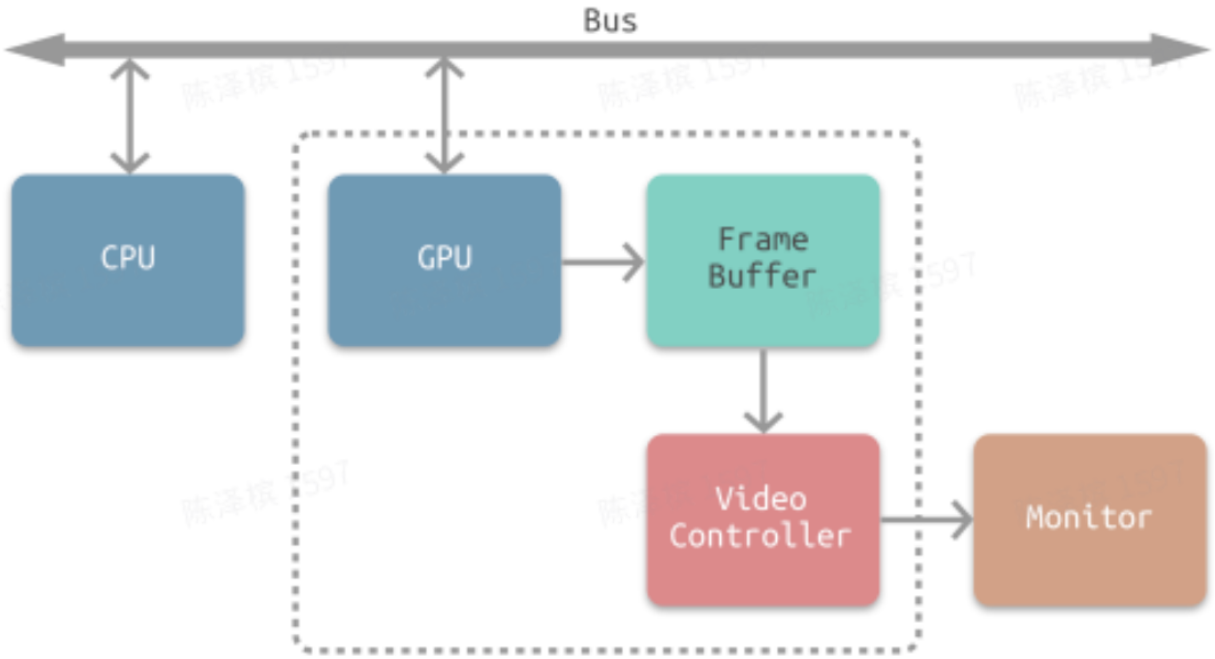
经过上图一系列的处理，像素信息就被存在了帧缓冲器中

屏幕成像

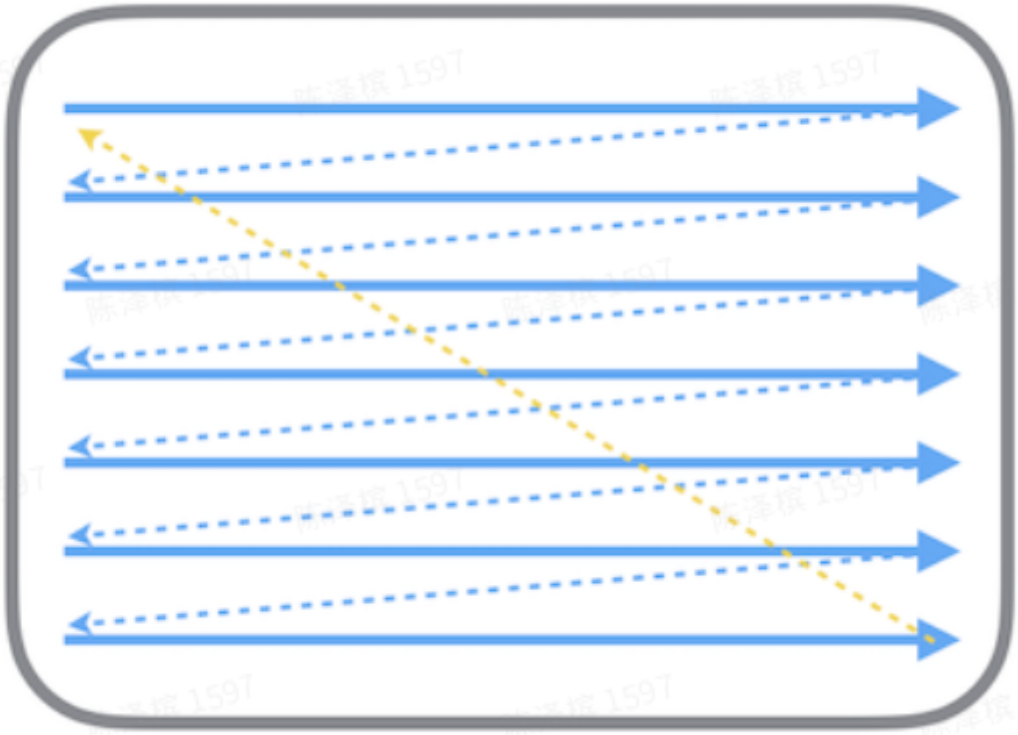
从帧缓存到屏幕

在图像渲染流程结束之后，接下来就需要将得到的像素信息显示在物理屏幕上了。
GPU 最后一步渲染结束之后像素信息，被存在帧缓冲器（Framebuffer）中，之后视频控制器（Video Controller）会读取帧缓冲器中的信息，经过数模转换传递给显示器（Monitor），进行显示。

完整的流程如下图所示：



经过 GPU 处理之后的像素集合，也就是位图，会被帧缓冲器缓存起来，供之后的显示使用。
显示器的电子束会从屏幕的左上角开始逐行扫描，屏幕上的每个点的图像信息都从帧缓冲器中的位图进行读取，在屏幕上对应地显示。
扫描的流程如下图所示：



电子束扫描的过程中，屏幕就能呈现出对应的结果，每次整个屏幕被扫描完一次后，就相当于呈现了一帧完整的图像。屏幕不断地刷新，不停呈现新的帧，就能呈现出连续的影像。
而这个屏幕刷新的频率，就是帧率（Frame per Second，FPS）。由于人眼的视觉暂留效应，当屏幕刷新频率足够高时（FPS 通常是 50 到 60 左右），就能让画面看起来是连续而流畅的。
对于 iOS 而言，app 应该尽量保证 60 FPS 才是最好的体验。

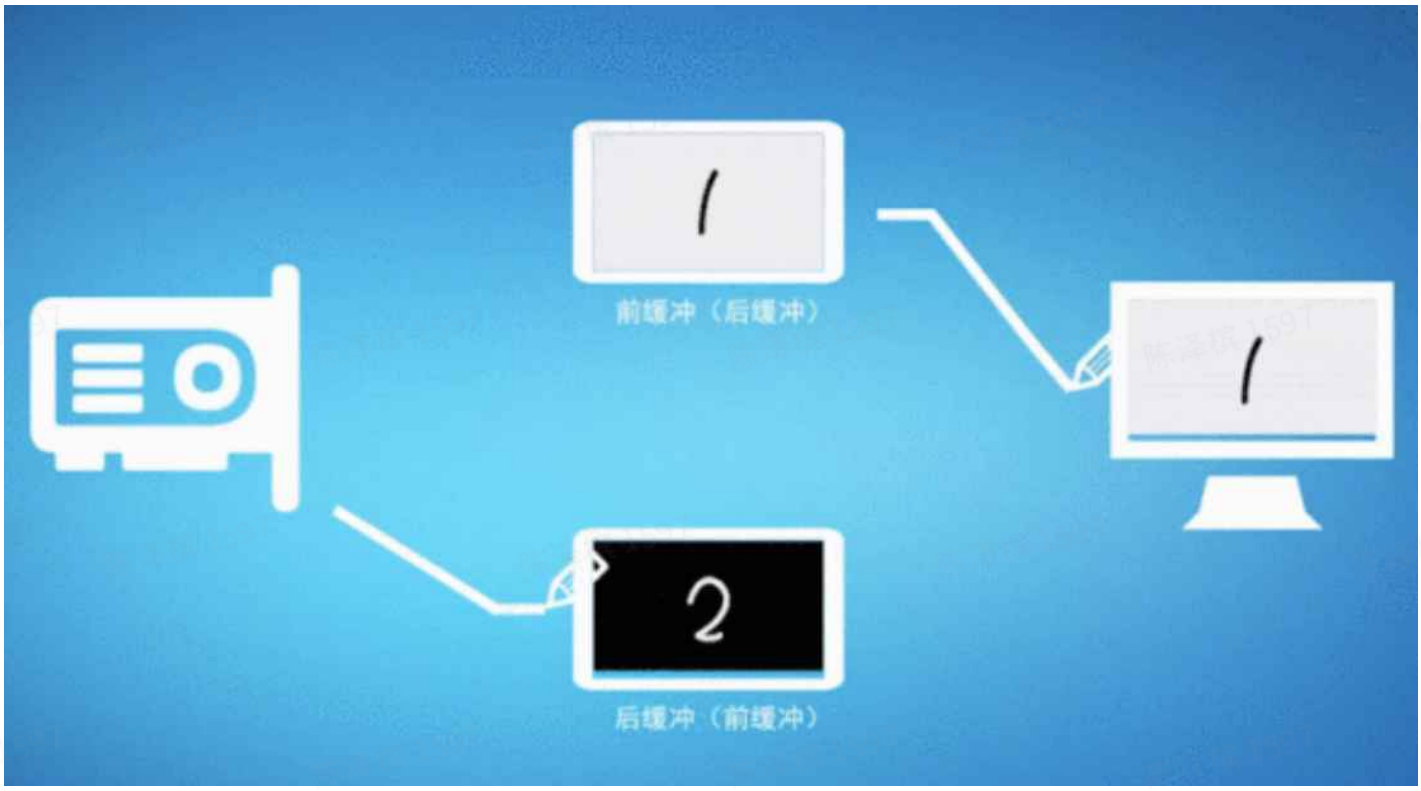
屏幕撕裂

在这种单一缓存的模式下，最理想的情况就是一个流畅的流水线：每次电子束从头开始新的一帧的扫描时，CPU+GPU 对于该帧的渲染流程已经结束，将渲染好的位图已经放入帧缓冲器中。
但这种完美的情况是非常脆弱的，很容易产生屏幕撕裂：

CPU+GPU 的渲染流程是一个非常耗时的过程。如果在电子束开始扫描新的一帧时，位图还没有渲染好，而是在扫描到屏幕中间时才渲染完成，被放入帧缓冲器中，那么已扫描的部分就是上一帧的画面，而未扫描的部分则会显示新的一帧图像，这就造成屏幕撕裂。

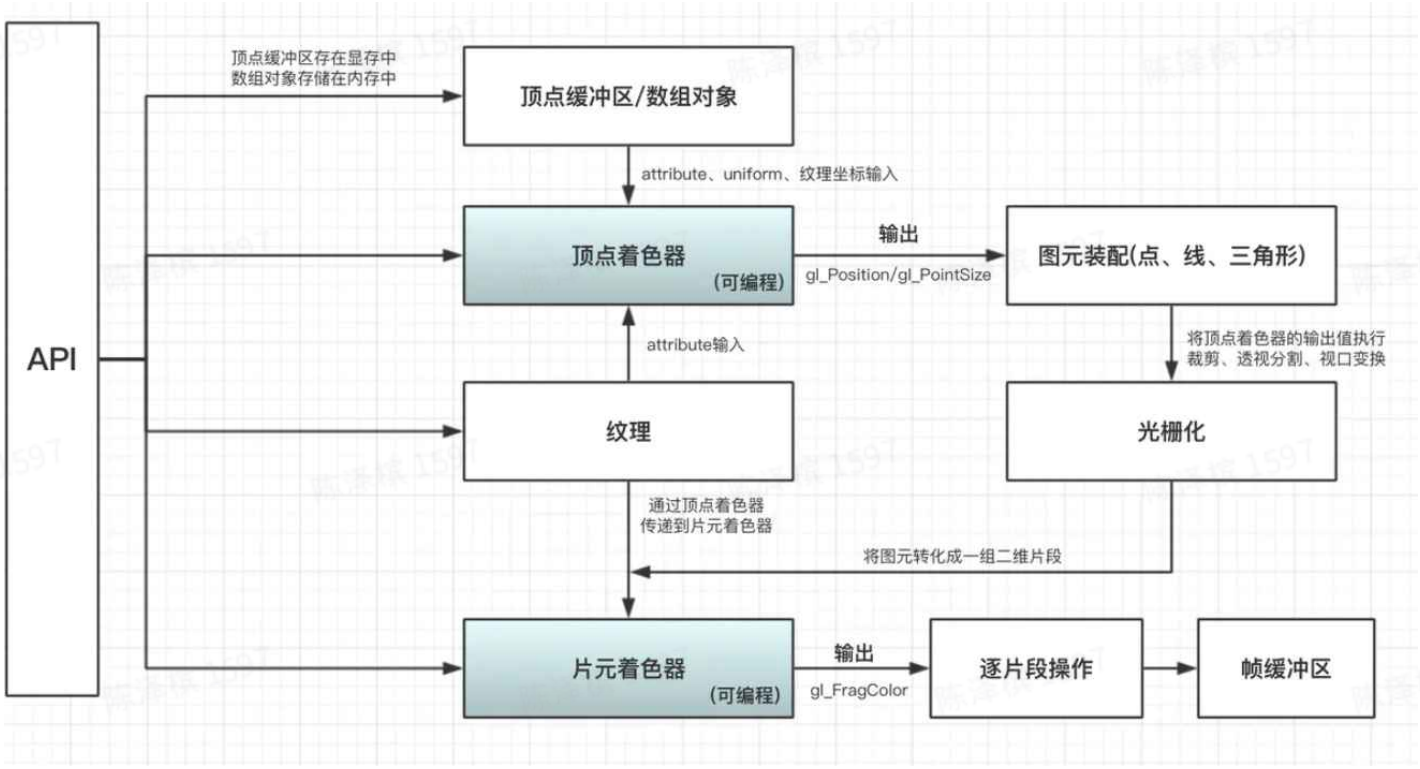
垂直同步 Vsync + 双缓冲机制 Double Buffering

垂直同步信号相当于给帧缓冲器加锁：当电子束完成一帧的扫描，将要从头开始扫描时，就会发出一个垂直同步信号。只有当视频控制器接收到 Vsync 之后，才会将帧缓冲区中的位图更新为下一帧，这样就能保证每次显示的都是同一帧的画面，因而避免了屏幕撕裂。
但是这种情况下，视频控制器在接受到 Vsync 之后，就要将下一帧的位图传入，这意味着整个 CPU+GPU 的渲染流程都要在一瞬间完成，这是明显不现实的。所以双缓冲机制会增加一个新的备用缓冲区（back buffer）。渲染结果会预先保存在 back buffer 中，在接收到 Vsync 信号的时候，视频控制器会将 back buffer 中的内容置换到 frame buffer 中，此时就能保证置换操作几乎在一瞬间完成（实际上是交换了内存地址）。



虽然vsync+双缓冲解决了屏幕撕裂的问题，但是引入了掉帧的问题，实际体验就是卡顿现象。
如果在接收到 Vsync 之时 CPU 和 GPU 还没有渲染好新的位图，视频控制器就不会去替换 frame buffer 中的位图。这时屏幕就会重新扫描呈现出上一帧一模一样的画面，导致一种卡顿的感觉。

结语



到这里，大家应该了解了OpenGL的来龙去脉，一些基础概念，工作流程，怎么渲染上屏；有了这个大体宏观的认识后，接下来，我们就是细化去研究上面这个图中的每一个流程，再具体到OpenGL ES函数的使用，以及应用场景。