Lecture2 渲染流水线

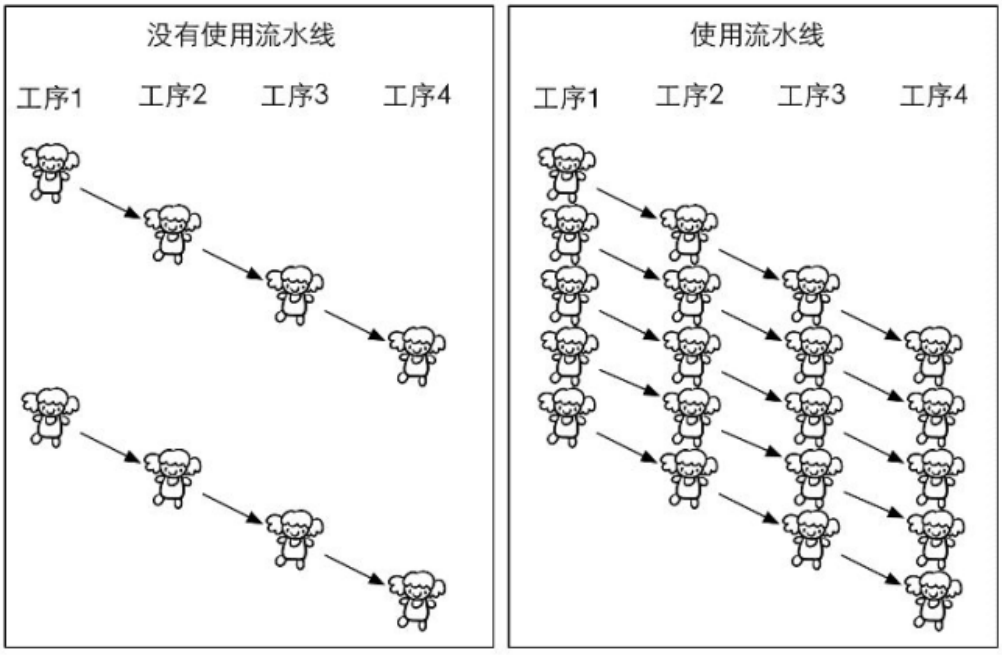
- 输入：一个虚拟摄像机、一些光源、一些Shader以及纹理等
- 输出：生成或者说是渲染一张**二维纹理**，即我们在电脑屏幕上看到的所有效果

1. 综述

什么是流水线

假设，老王有一个生产洋娃娃的工厂，一个洋娃娃的生产流程可以分为4个步骤：

- 第1步，制作洋娃娃的躯干
- 第2步，缝上眼睛和嘴巴
- 第3步，添加头发
- 第4步，给洋娃娃进行最后的产品包装



- 在流水线出现之前，只有在每个洋娃娃完成了所有这4个工序后才能开始制作下一个洋娃娃。如果说每个步骤需要的时间是1小时的话，那么每4个小时才能生产一个洋娃娃
- 使用流水线后，每个步骤由专人完成，所有步骤**并行**进行。用流水线的好处在于可以提高单位时间的生产量。在洋娃娃的例子中，使用了流水线技术后每1个小时就可以生产一个洋娃娃

流水线系统中决定最后生产速度的是**最慢的工序所需的时间**

- 理想情况下，如果把一个非流水线系统分成 n 个流水线阶段，且每个阶段耗费时间相同的话，会使整个系统得到 n 倍的速度提升

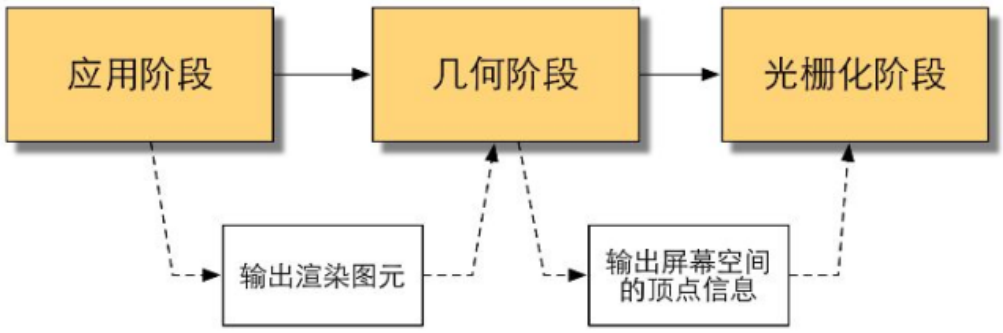
什么是渲染流水线

渲染流水线的工作任务在于由一个三维场景出发、生成（或者说渲染）一张二维图像

参考书籍 Real-Time Rendering, Third Edition 主要分为三个阶段

- 应用阶段（Application Stage）
- 几何阶段（Geometry Stage）
- 光栅化阶段（Rasterizer Stage）

这里仅仅是概念性阶段，每个阶段**本身通常也是一个流水线系统**，即包含了**子流水线阶段**



应用阶段 Application Stage

开发者具有这个阶段的绝对控制权，通常由 CPU 负责实现

这一阶段主要由三个任务

1. 准备好所需**场景数据**
    - 摄像机的位置
    - 视锥体
    - 场景中的模型
    - 光源
2. **粗粒度剔除**
    - 把不可见的物体剔除出去
3. 设置每个模型的**渲染状态**
   这一阶段最重要的输出是渲染所需的几何信息，即**渲染图元（rendering primitives）**，通俗来讲，渲染图元可以是点、线、三角面
    - 材质
    - 纹理
    - Shader

几何阶段 Geometry Stage

决定需要绘制的图元是什么，怎样绘制它们，在哪里绘制它们，通常在 GPU 上进行

- 与每个渲染图元打交道，进行逐顶点、逐多边形的操作
- 把顶点坐标变换到**屏幕空间**中
- 再交给光栅器进行处理

这一阶段将会输出屏幕空间的**二维顶点坐标**、**每个顶点对应的深度值**、**着色**等相关信息，并传递给下一个阶段

光栅化阶段 Rasterizer Stage

使用上个阶段传递的数据来产生屏幕上的像素，并渲染出最终的图像，通常在 GPU 进行

- 光栅化的任务主要是决定**每个渲染图元中的哪些像素应该被绘制在屏幕上**
- 它需要对上一个阶段得到的逐**顶点数据**（例如纹理坐标、顶点颜色等）进行**插值**，然后再进行**逐像素处理**

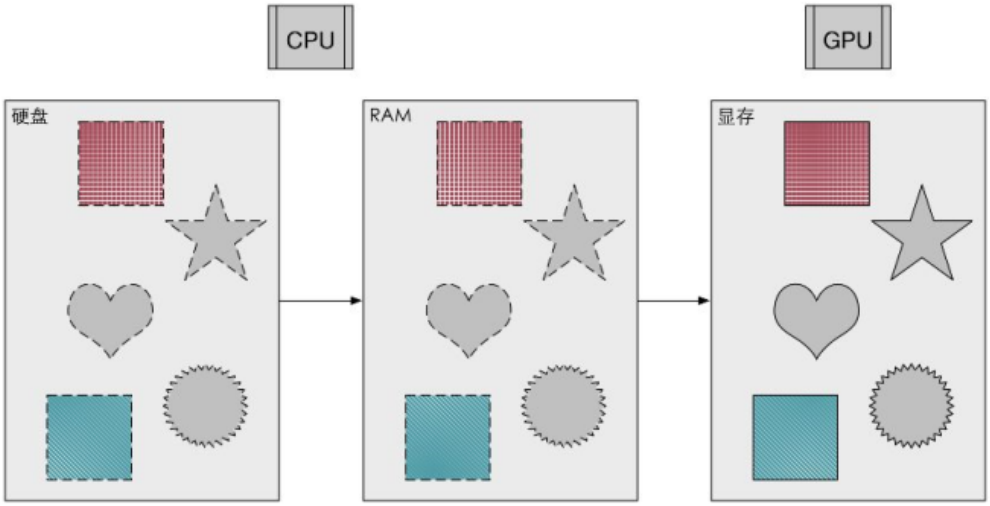
2. CPU 与 GPU 之间的通信

渲染流水线的起点是 CPU，即应用阶段。应用阶段大致可分为下面 3 个阶段

把数据加载到显存中

所有渲染所需的数据都需要从

- 从**硬盘（Hard Disk Drive，HDD）**中
- 加载到**系统内存（Random Access Memory，RAM）**
- 然后，**网格和纹理等数据**又被加载到显卡上的存储空间——**显存（Video Random Access Memory，VRAM）**中（显卡对于显存的访问速度更快）



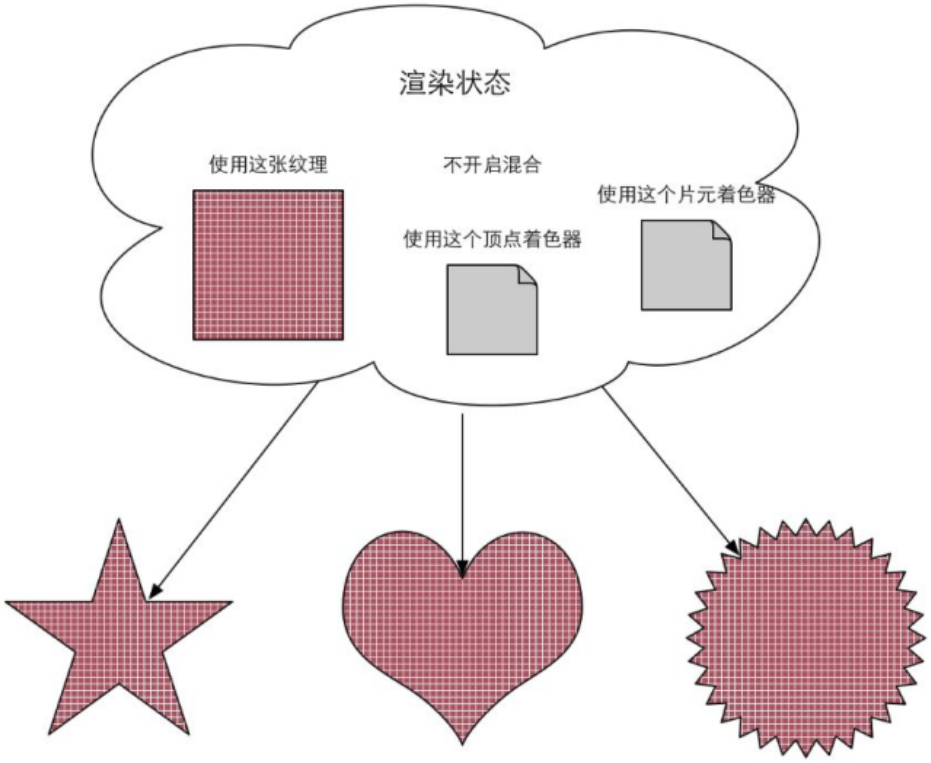
- 真实渲染中需要加载到显存中的数据往往比图复杂许多。例如，**顶点的位置信息、法线方向、顶点颜色、纹理坐标等**

设置渲染状态

指导 GPU 如何进行渲染工作

这些状态定义了场景中的网格是怎样被渲染的

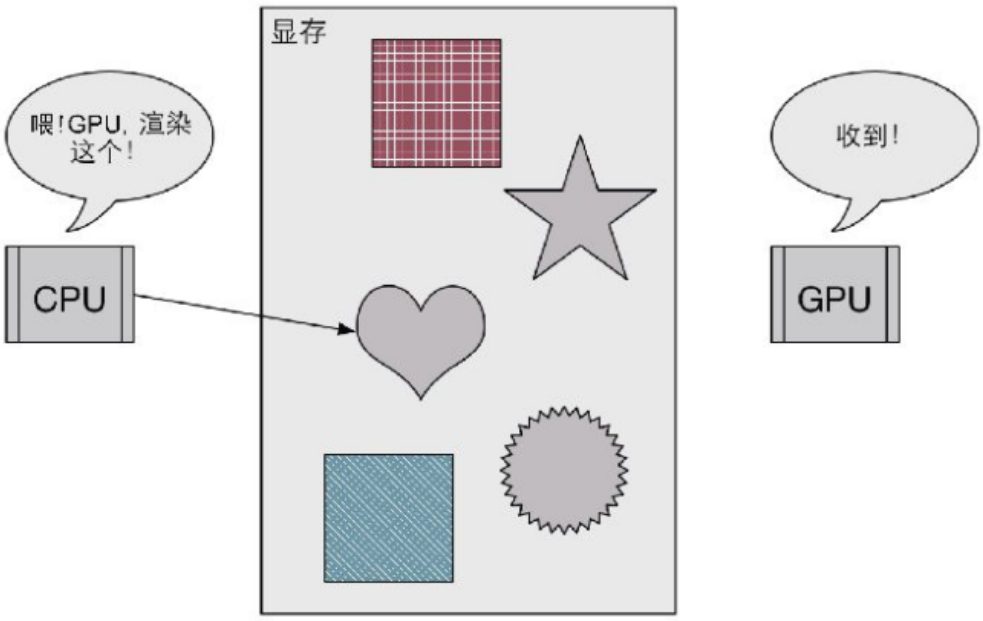
- 顶点着色器（Vertex Shader）
- 片元着色器（Fragment Shader）
- 光源属性
- 材质
- 等等



调用 Draw Call

Draw Call 就是一个命令，它的发起方是 CPU，接收方是 GPU

这个命令仅仅会指向一个需要被渲染的图元（primitives）列表，而不会再包含任何材质信息

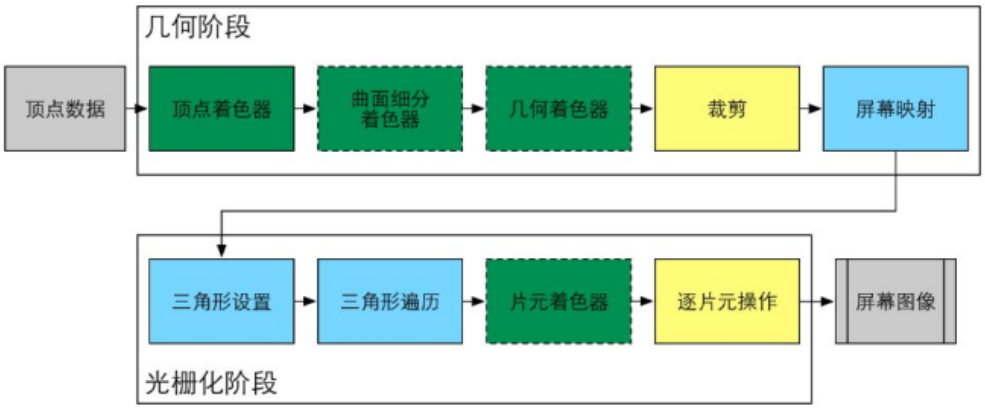


- CPU 通过调用 Draw Call 来告诉 GPU 开始进行一个渲染过程。 一个 Draw Call 会指向本次调用需要渲染的图元列表
- 当给定了一个 Draw Call 时，GPU 就会根据渲染状态（例如材质、纹理、着色器等）和所有输入的顶点数据来进行计算，最终输出成屏幕上显示的那些漂亮的像素

3. GPU 流水线

概述

- 在应用阶段，CPU 与 GPU 通信，通过调用 Draw Call 命令来进行 GPU 渲染，**GPU 渲染的过程就是 GPU 流水线**
- 对于概念阶段的后两个阶段，即几何阶段和光栅化阶段，**开发者无法拥有绝对的控制权**，其实现的载体是GPU
- 虽然我们无法完全控制这两个阶段的实现细节，但 GPU 向开发者开放了很多控制权

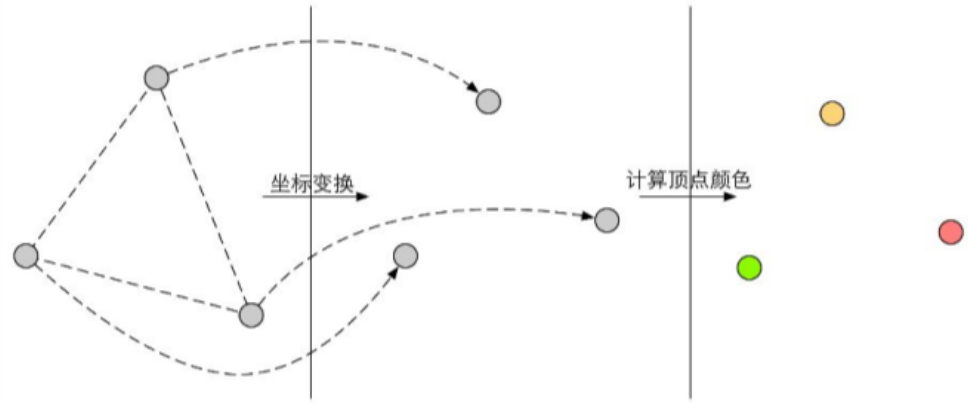


- 几何阶段和光栅化阶段分成**若干更小的流水线阶段**，这些流水线阶段由 GPU 来实现，每个阶段 GPU 提供了不同的**可配置性或可编程性**

几何阶段	描述
顶点数据	GPU 的渲染流水线接收顶点数据作为输入，这些顶点数据是由应用阶段加载到现存中，再由 Draw Call 指定的，这些数据随后被传递给顶点着色器
顶点着色器 Vertex Shader	是**完全可编程**的，它通常用于实现顶点的**空间变换、顶点着色**等功能
曲面细分着色器 Tessellation Shader	是一个**可选的着色器**，用于细分图元
几何着色器 Geometry Shader	是一个**可选的着色器**，它可以被用于执行逐图元（Per-Primitive）的着色操作，或者被用于产生更多的图元
裁剪 Clipping	将**不在摄像机视野内的顶点裁剪掉**，并提出某些三角图元的面片
屏幕映射 Screen Mapping	不可配置和编程的，它负责把每个图元的坐标转换到**屏幕坐标系**中

光栅化阶段	描述
三角形设置 Triangle Setup	不可配置和编程
三角形遍历 Triangle Traversal	不可配置和编程
片元着色器 Fragment Shader	**是完全可编程的**，它用于实现逐片元（Per-Fragment）的着色操作
逐片元操作 Per-Fragment Operations	负责执行很多重要的操作，例如修改颜色、深度缓冲、进行混合等，它不是**不可编程**的，但具有很高的**可配置性**

顶点着色器 Vertex Shader



- 处理单位：顶点
  - 对于每个输入进来的顶点都会调用一次顶点着色器
  - 顶点着色器本身不可以创建或者销毁任何顶点，而且无法得到顶点与顶点之间的关系
- 主要工作：**坐标变换和逐顶点光照**
- 输出方式
  - 最常见的输出路径是经光栅化后交给片元着色器进行处理
  - 在现代的 Shader Model 中，它还可以把数据发送给曲面细分着色器或几何着色器

坐标变换

对顶点的坐标（即位置）进行某种变换

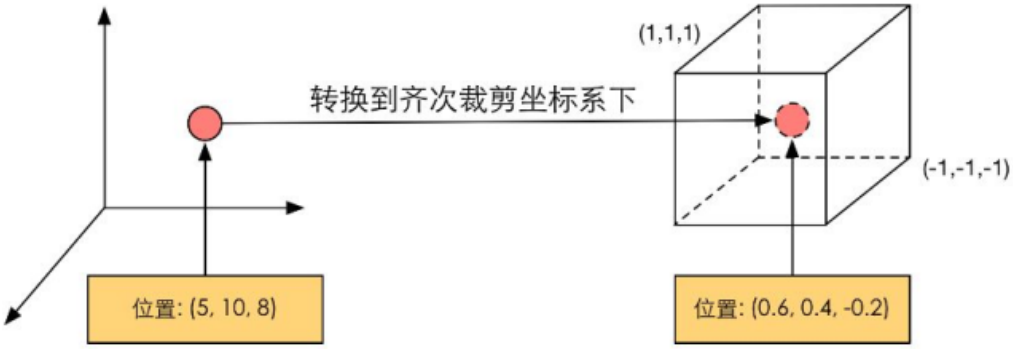
- 顶点着色器可以在这一步中改变顶点的位置，这在顶点动画中是非常有用的
- 我们可以通过改变顶点位置来模拟水面、布料等

一个最基本的顶点着色器**必须完成**的一个工作：把**顶点坐标从模型空间转换到齐次裁剪空间**

在顶点着色器中会看到如下的代码

- `o.pos = mul(UNITY_MVP, v.position);`

类似上面这句代码的功能，就是把**顶点坐标**转换到**齐次裁剪坐标系**下，接着通常再由硬件做透视除法后，最终得到**归一化的设备坐标（Normalized Device Coordinates，NDC)**



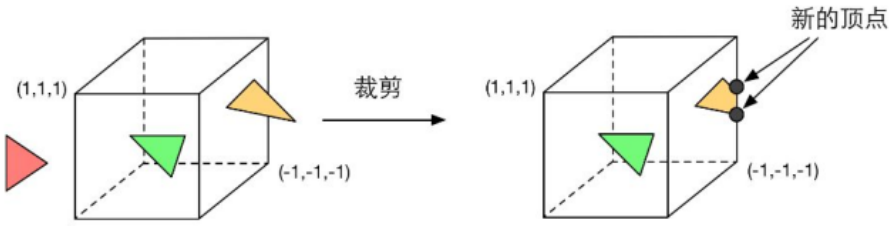
- 注意，图中给出的坐标范围是 **OpenGL 和 Unity 下的 NDC**，它的 z 分量范围在 [-1, 1] 之间，而在 **DirectX** 中，NDC 的 z 分量范围是 [0, 1]

裁剪 Clipping

场景可能会很大，而摄像机的视野范围很有可能不会覆盖所有的场景物体，那些不在摄像机视野范围的物体不需要被处理

和顶点着色器不同，这一步是**不可编程**的，即我们无法通过编程来控制裁剪的过程，而是**硬件**上的固定操作，但我们可以**自定义一个裁剪操作来对这一步进行配置**

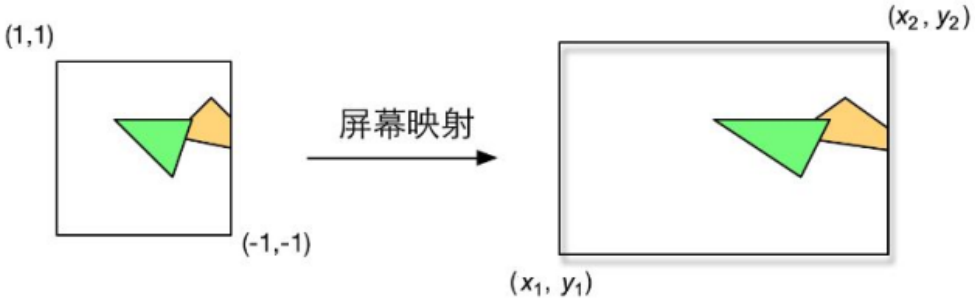
图元与摄像机的关系



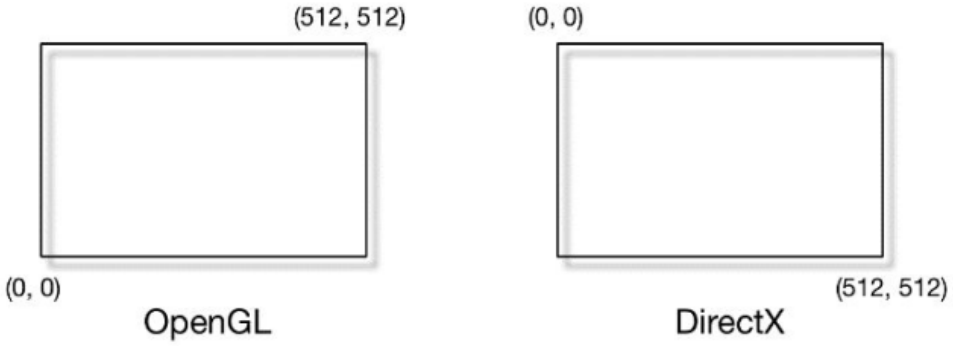
- **完全在视野内**：继续传递给下一个流水线阶段
- **部分在视野内**：不会向下传递（不需要被渲染）
- **完全在视野外**：进行一个处理，例如，一条线段的一个顶点在视野内，而另一个顶点不在视野内，那么在视野外部的顶点应该使用一个新的顶点来代替，这个新的顶点位于这条线段和视野边界的交点处

屏幕映射 Screen Mapping

- 输入：图元在 NDC 下的坐标
- 输出：把每个图元的 x 和 y 坐标转换到**屏幕坐标系 Screen Coordinates**，这个过程实际是一个**缩放**的过程
    - 屏幕映射不会对输入的 z 坐标做任何处理，实际上，屏幕坐标系和 z 坐标一起构成了一个坐标系，叫做**窗口坐标系（Window Coordinates）**



- OpenGL：把屏幕左下角当成最小的窗口坐标值
- DirectX：屏幕的左上角为最小的窗口坐标值



三角形设置 Triangle Setup

从上一个阶段输出的信息是屏幕坐标系下的顶点位置以及和它们相关的额外信息，如深度值（z 坐标）、法线方向、视角方向等

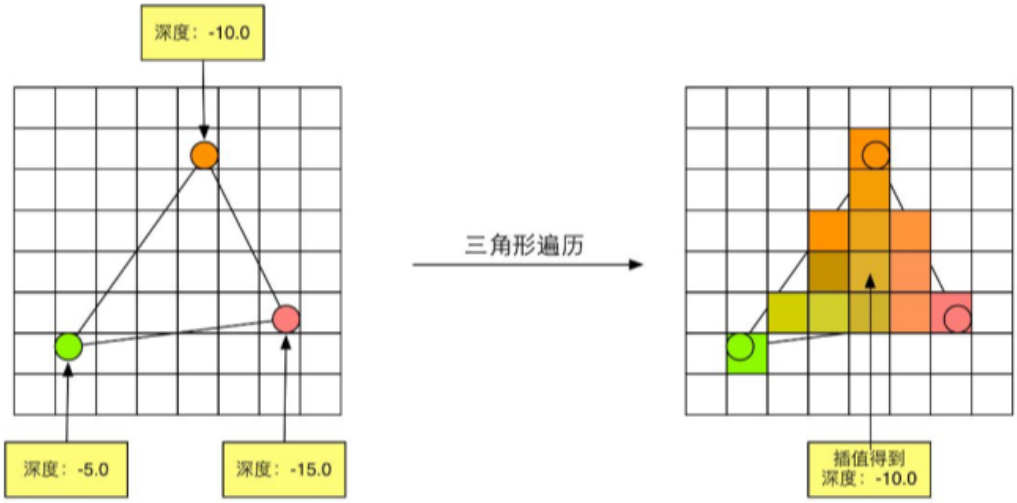
光栅化阶段有两个最重要的目标

- 计算每个图元覆盖了哪些像素
- 为这些像素计算它们的颜色

这个阶段会计算光栅化一个三角网格所需的信息。具体来说，上一个阶段输出的都是三角网格的顶点，即我们得到的是三角网格每条边的两个端点，但如果要得到整个三角网格对像素的覆盖情况，我们就必须计算每条边上的像素坐标。为了能够计算边界像素的坐标信息，我们就需要得到三角形边界的表示方式。这样一个计算三角网格表示数据的过程就叫做三角形设置。它的输出是为了给下一个阶段做准备

三角形遍历 Triangle Traversal

阶段将会检查每个像素是否被一个三角网格所覆盖。如果被覆盖的话，就会生成一个**片元（fragment）**，一个找到哪些像素被三角网格覆盖的过程就是三角形遍历



- 三角形遍历阶段会根据上一个阶段的计算结果来判断一个三角网格覆盖了哪些像素，并使用三角网格3个顶点的顶点信息对整个覆盖区域的像素进行插值

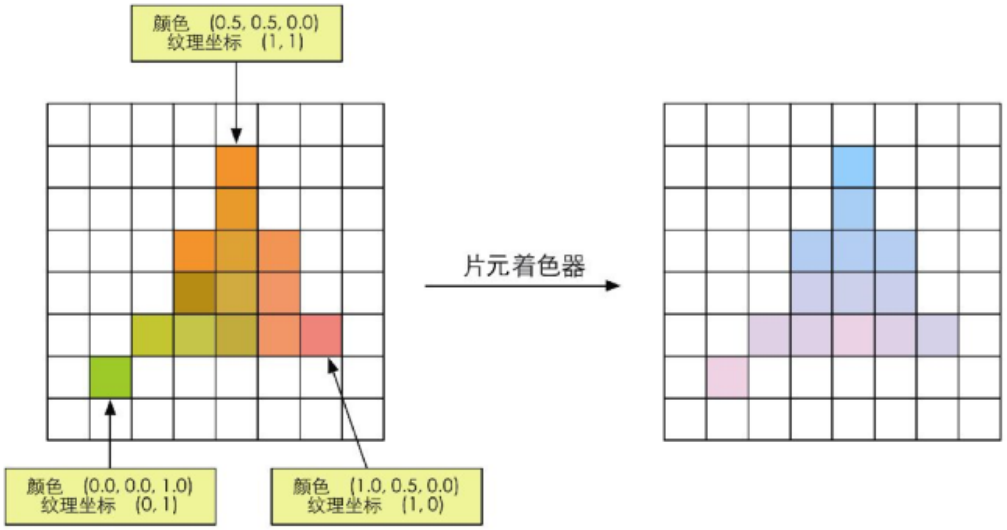
这一步的输出就是得到一个片元序列

- 需要注意的是，一个片元并不是真正意义上的像素，而是包含了很多状态的集合
  - 屏幕坐标
  - 深度信息
  - 其它几何阶段输出的顶点信息
    - 法线、纹理坐标等

片元着色器 Fragment Shader

- 在 DirectX 中，片元着色器被称为**像素着色器（Pixel Shader）**

前面的光栅化阶段实际上并不会影响屏幕上每个像素的颜色值，而是会产生一系列的**数据信息**，用来表述**一个三角网格是怎样覆盖每个像素的。**而每个片元就负责存储这样一系列数据



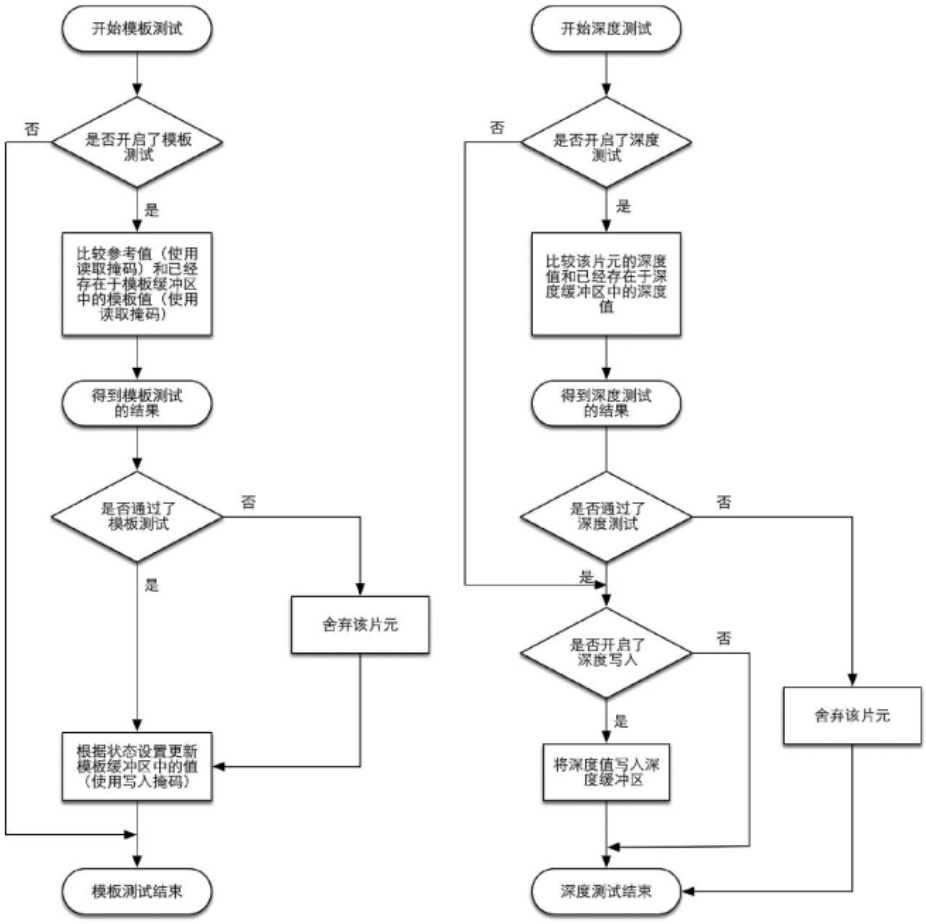
- 输入：上一个阶段对顶点信息插值得到的结果
- 输出：一个或者多个颜色值
- 局限：仅可以影响单个片元，当执行片元着色器时，它不可以将自己的任何结果直接发送给它的邻居们
    - 不过，片元着色器可以访问到导数信息（gradient，或者说是derivative）

逐片元操作 Per-Fragment Operations

- 在 DirectX 中，这一阶段被称为**输出合并阶段（Output-Merger）**

这一阶段有几个任务

- 决定每个片元的可见性。这涉及了很多测试工作，例如深度测试、模板测试等
- 如果一个片元通过了所有的测试，就需要把这个片元的颜色值和已经存储在颜色缓冲区中的颜色进行合并，或者说是混合
- 逐片元操作阶段是**高度可配置性的**



模板测试 Stencil Test

- 高度可配置
- 如果开启了模板测试，GPU 会首先读取（使用读取掩码）模板缓冲区中该片元位置的模板值，然后将该值和读取（使用读取掩码）到的参考值（reference value）进行比较，如果这个片元没有通过这个测试，该片元就会被舍弃
- 不管一个片元有没有通过模板测试，我们都可以根据模板测试和下面的深度测试结果来修改模板缓冲区
- 模板测试通常用于**限制渲染的区域。**另外，模板测试还有一些更高级的用法，如**渲染阴影**、**轮廓渲染**等

深度测试 Depth Test

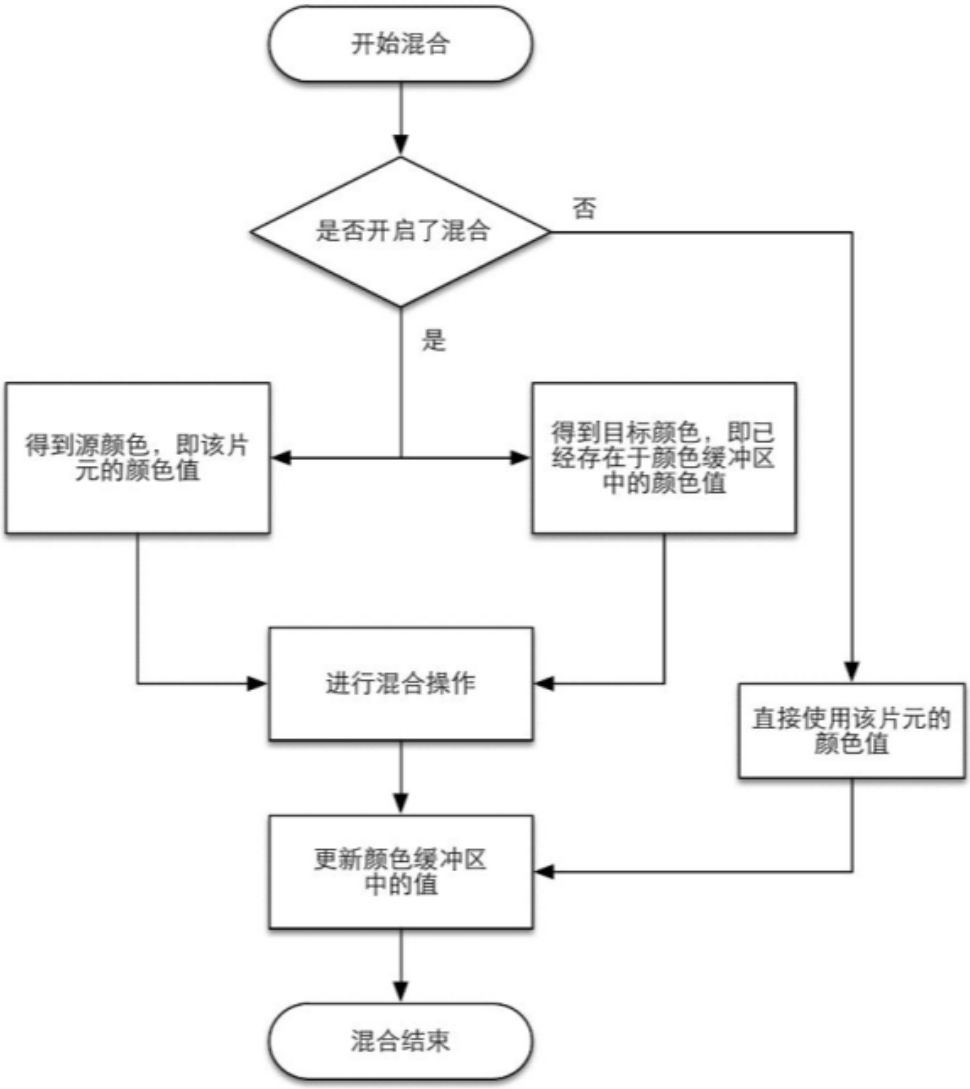
- 高度可配置
- 如果开启了深度测试，GPU会把该片元的深度值和已经存在于深度缓冲区中的深度值进行比较
- 我们总想只显示出离摄像机最近的物体，而那些被其他物体遮挡的就不需要出现在屏幕上
- 如果这个片元没有通过这个测试，该片元就会被舍弃
- 和模板测试有些不同的是，如果一个片元没有通过深度测试，**它就没有权利更改深度缓冲区中的值**
- 而如果它通过了测试，开发者还可以指定是否要用这个片元的深度值覆盖掉原有的深度值，这是通过**开启/关闭深度写入**来做到的

混合 Blend

- 高度可配置

当我们执行这次渲染时，颜色缓冲中往往已经有了上次渲染之后的颜色结果，那么，我们是使用这次渲染得到的颜色完全覆盖掉之前的结果，还是进行其他处理

- **不透明物体**：开发者可以关闭混合，这样片元着色器计算得到的颜色值就会直接覆盖掉颜色缓冲区中的像素值
- **半透明物体**：使用混合操作来让这个物体看起来是透明的



- 如果开启了混合，GPU会取出源颜色和目标颜色，将两种颜色进行混合
  - **源颜色**：片元着色器得到的颜色值
  - **目标颜色**：已经存在于颜色缓冲区中的颜色值

一般深度测试和模板测试会在混合之前，这是为了将i的计算成本，将深度测试提前执行的技术通常也被称为 **Early-Z** 技术

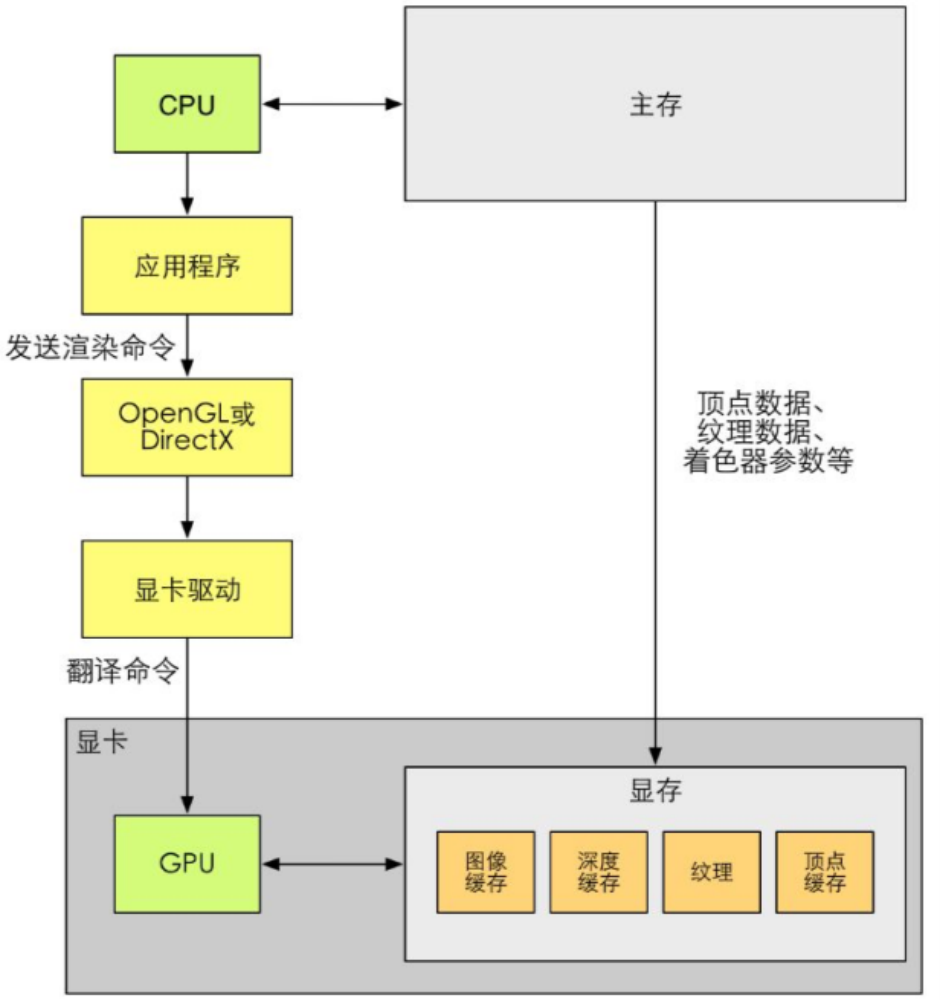
双重缓冲 Double Buffering

- 当模型的图元经过了上面层层计算和测试后，就会显示到我们的屏幕上
- 我们的屏幕显示的就是**颜色缓冲区**中的颜色值
- 但是，为了避免我们看到那些**正在进行光栅化的图元**，GPU会使用**双重缓冲**，即在**后置缓冲 Back Buffer**中，一旦场景已经被渲染到了后置缓冲中，GPU就会交换后置缓冲区和**前置缓冲（Front Buffer）**中的内容，而前置缓冲区是之前显示在屏幕上的图像

4. 释疑

OpenGL / DirectX

- 要开发者直接访问 GPU 是一件非常麻烦的事情，而**图像编程接口**在这些硬件的基础上实现了一层**抽象**
- OpenGL 和 DirectX 就是**图像应用编程接口**，这些接口用于渲染二维或三维图形，接口架起了上层应用程序和底层 GPU 的沟通桥梁
- 一个**应用程序**向这些**接口**发送渲染命令
- 而这些**接口**会依次向**显卡驱动（Graphics Driver）**发送渲染命令
- 这些**显卡驱动**是真正知道如何和 **GPU 通信**的角色，它们把OpenGL或者DirectX的函数调用翻译成了GPU能够听懂的语言，同时它们也负责把纹理等数据转换成GPU所支持的格式



- 应用程序运行在 CPU 上
- 应用程序通过调用 OpenGL 或 DirectX 的图形接口将渲染所需的数据（顶点、纹理、材质等）存储在显存的特定位置
- 开发者可以通过图像编程接口发出渲染命令（Draw Call）
- 它们将会被显卡驱动翻译成 GPU 能够理解的代码，进行真正的绘制

显存 Video Random Access Memory

- GPU 可以在显存中存储任何数据，但对于渲染来说一些数据类型是必需的，例如用于屏幕显示的**图像缓冲、深度缓冲**等

HLSL / GLSL / CG

顶点着色器和片元着色器是可编程的，我们可以使用一种特定的语言来编写程序

着色语言是专门用于编写着色器的，常见的着色语言有

- DirectX 的 HLSL（High Level Shading Language）
- OpenGL 的 GLSL（OpenGL Shading Language）
- NVIDIA 的 CG（C for Graphic）

这些语言会被编译成与机器无关的**汇编语言**，也被称为**中间语言（Intermediate Language，IL）**，再交给显卡驱动来翻译成真正的**机器语言**

优点

- GLSL：**跨平台性**，可以在Windows、Linux、Mac甚至移动平台等多种平台上工作，GLSL是依赖**硬件**，而非操作系统层级的
- HLSL：支持HLSL的平台相对比较有限，几乎完全是微软自己的产品，如Windows、Xbox 360、PS3等。这是因为在其他平台上没有可以编译HLSL的编译器
- CG：跨平台，它会根据平台的不同，编译成相应的中间语言

Draw Call

Draw Call 本身的含义很简单，就是**CPU 调用图像编程接口，以命令GPU进行渲染的操作**

- OpenGL 中的 `glDrawElements` 命令
- DirectX 中的 `DrawIndexedPrimitive`命令

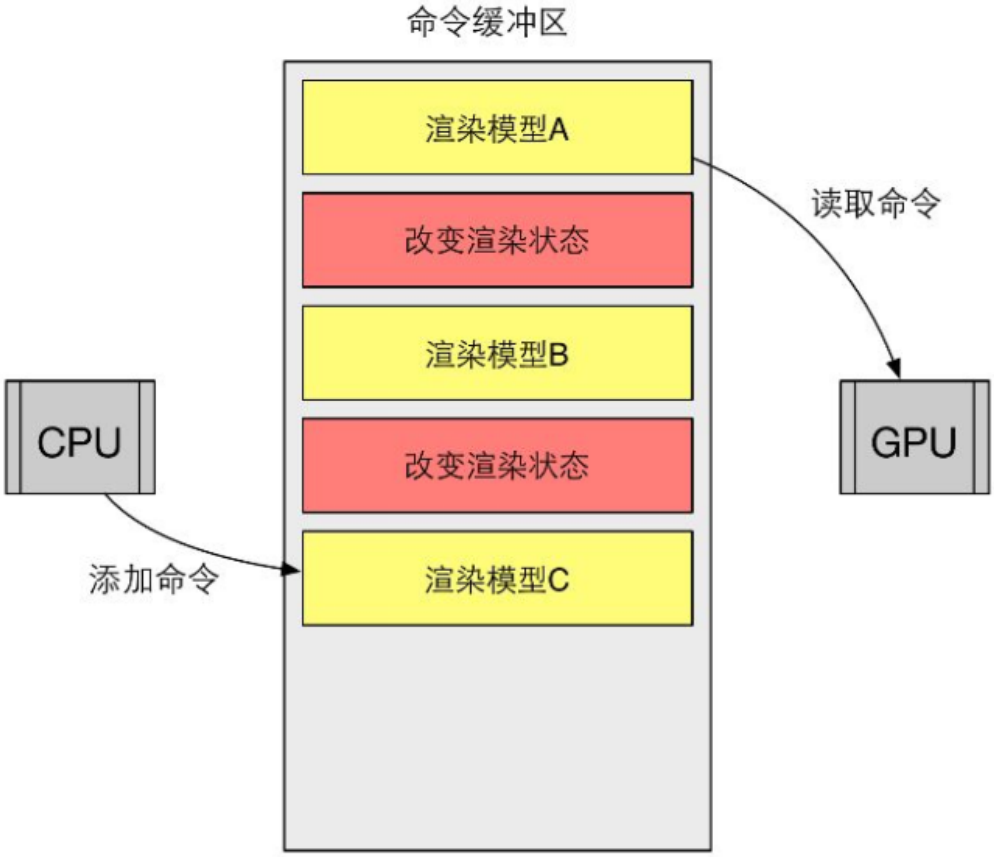
Draw Call 的性能问题可能来自于 CPU，为了采用渲染管线的流水线工作模式，需要让CPU和GPU可以并行工作。而解决方法就是使用一个**命令缓冲区（Command Buffer）**

命令缓冲区 Command Buffer

命令缓冲区包含了一个命令队列，命令缓冲区使得CPU和GPU可以相互独立工作

- **CPU** 向其中**添加**命令
- **GPU** 从中**读取**命令
- 添加和读取的过程是互相独立的

命令缓冲区中的命令有很多种类，而 Draw Call 是其中一种，其他命令还有改变渲染状态等



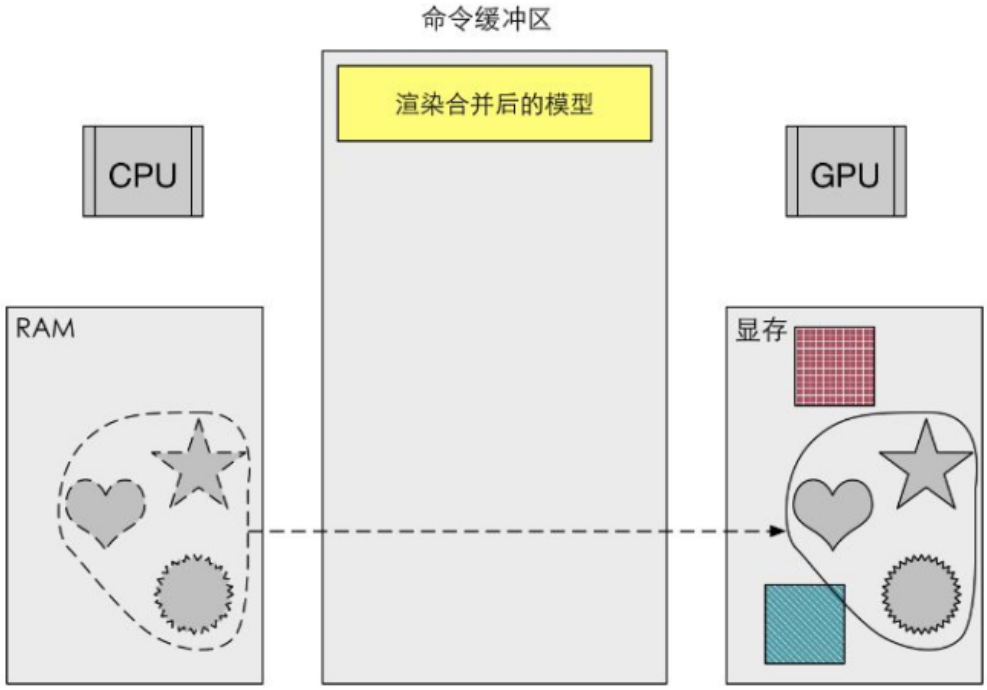
每次调用 Draw Call 之前，CPU 需要向 GPU 发送很多内容，包括**数据、状态和命令等。**在这一阶段，CPU 需要完成**很多工作**，例如检查渲染状态等。而一旦 CPU 完成了这些准备工作，GPU 就可以开始本次的渲染

如果 Draw Call 的数量太多，CPU 就会把大量时间花费在**提交 Draw Call** 上，造成 CPU 的过载

如何减少 Draw Call - 批处理 Batching

提交大量很小的 Draw Call 会造成 CPU 的性能瓶颈，一个很显然的优化想法就是**把很多小的 Draw Call 合并成一个大的 Draw Call**，这就是批处理

- 需要注意的是，由于我们需要在 CPU 的内存中合并网格，而合并的过程是需要消耗时间的，批处理技术更加适合于那些静态的物体



为了减小 Draw Call 的开销

- 避免使用大量很小的网格，如果实在需要使用，考虑是否可以合并他们
- 避免使用过多的材质，尽量在不同网格之间公用一个材质

Shader

> GPU 流水线上一些**可高度编程的阶段**，而由着色器编译出来的最终代码是会在 GPU 上运行的（对于固定管线的渲染来说，着色器有时等同于一些特定的渲染设置）

特定类型的着色器

- 顶点着色器
- 片元着色器

要得到出色的游戏画面是需要包括 Shader 在内的**所有渲染流水线阶段的共同参与**才可完成：设置适当的渲染状态，使用合适的混合函数，开启还是关闭深度测试/深度写入等

5. 扩展阅读

- OpenGL: https://www.opengl.org/wiki/Rendering_Pipeline_Overview
- DirectX: https://msdn.microsoft.com/en-us/library/windows/desktop/ff476882(v=vs.85).aspx