Shader 笔记

郑华

2018 年 9 月 28 日

目录

第一章　着色器

第二章　基础概念

2.1 Draw Call

2.2 Shader

可以干啥　屏幕特效实现下雪

静态纹理的动态化，如波浪、水流等

光照的应用

投影

常用的 Shader 语言

- OpenGL GLSL(opengGL Shader Language) SGI 公司开发的跨平台。
- DirectX HLSL(High Level Shader Language) 微软
- **->CG** 微软和英伟达联合开发的跨平台，基于 C 语言。
- **->Shader Lab** Unity3D 自己的 shader 语言。

矩阵变换　Unity 3D 矩阵一般是**左乘**。在Shader 中的前缀为UNITY_MATRIX_

1. M: 将物体坐标系变换到世界坐标系
2. V: 将世界坐标系变换到相机坐标系
3. P: 将 3D 坐标系转换成 2 维屏幕坐标系

当然在脚本中也可以完成类似功能，如

`Transform.parent.localToWorldMatrix.MultiplyPoint(tranform.localPosition)`.

2.3 Shader 版本

Shader 1.0（DirectX8.0）、Shader 2.0（DirectX9.0b）、Shader 3.0（DirectX9.0c）、Shader 4.0（DirectX10）、Shader 4.1（DirectX10.1）和 Shader 5（DirectX11）。

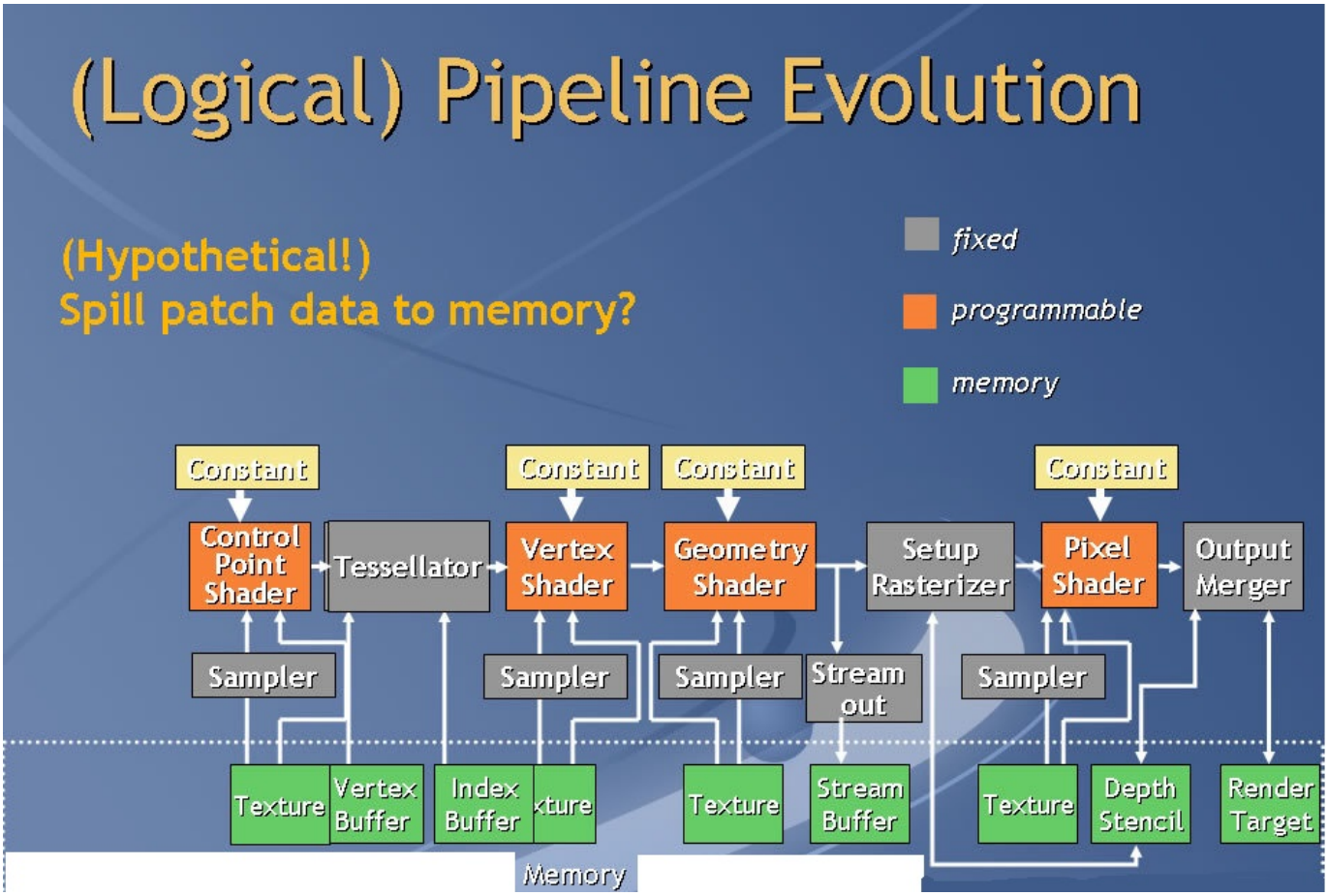


图 2.1: shader 通用架构

2.4 结构

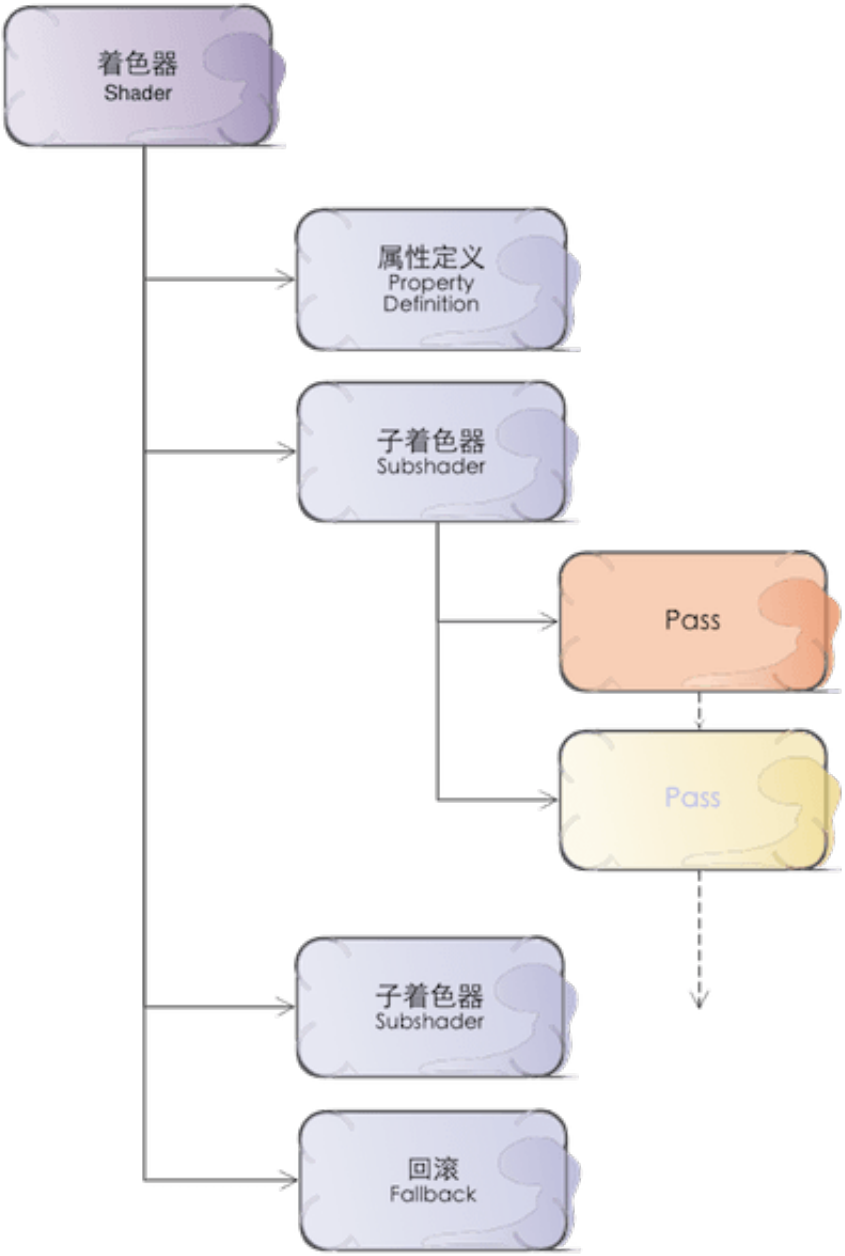


图 2.2: 着色器结构

- **属性定义（Property Definition）**：定义 Shader 的输入，并且绑定到编辑器上。
- **子着色器（SubShader）**：一个 Shader 可以有多个子着色器。这些子着色器互不相干且**只有一个会在最终的平台运行**。编写多个的目的是解决兼容性问题。*Unity* 会自己选择兼容终端平台的 *Shader* 运行。

- **回滚（Fallback）**：如果子着色器在终端平台上都无法运行，那么使用 Fallback 指定的备用 Shader，**俗称备胎**。
- **Pass**：一个 *Pass 就是一次绘制*。**对于表面着色器，只能有一个 Pass**，所以不存在 Pass 节。**顶点片段着色器可以有多个 Pass**。多次 Pass 可以实现很多特殊效果，例如当人物被环境遮挡时还可以看到人物轮廓就可以用多 Pass 来实现。
- **Cg 代码**:每个 Pass 中都可以包含自定义的 Cg 代码,**从 CGPROGRAM 开始到 ENDCG 结束**。

常用语义 POSITION **语义**等相当于告诉渲染引擎，这个变量是代表什么含义。

- `POSITION` 获取模型顶点信息。
- `NORMAL` 获取法线信息。
- `TEXCOORD n` 第 (n) 张贴图的 uv 坐标
- `COLOR n` 第 n 个定点色。`float4`
- `TANGENT` 获取切线信息
- `SV_POSITION` 表示经过 MVP 矩阵已经转化到屏幕坐标的位置
- `SV_TARGET` 输出到哪个 Render Target 上

常用坐标系

- 模型坐标系：也叫物体坐标系，3D 建模的时候每个模型都是在自己的坐标系下建立的，如果一个人物模型脚底是 (0,0,0) 点的话它的身上其它点的坐标都是相对脚底这个原点的。
- 世界坐标系：我们场景是一个世界，有自己的原点，模型放置到场景中后模型上的每个顶点就有了一个新的世界坐标。这个坐标可以通过模型矩阵 × 模型上顶点的模型坐标得到。
- 视图坐标系：又叫观察坐标系，是以观察者（相机）为原点的坐标系。场景中的物体只有被相机观察到才会绘制到屏幕上，相机可以设置视口大小和裁剪平面来控制可视范围，这些都是相对相机来说的，所以需要把世界坐标转换到视图坐标系来方便处理。
- 投影坐标系：场景是 3D 的，但是最终绘制到屏幕上是 2D，投影坐标系完成这个降维的工作，投影变换后 3D 的坐标就变成 2D 的坐标了。投影有平行投影和透视投影两种，可以在 Unity 的相机上设置。
- 屏幕坐标系: 最终绘制到屏幕上的坐标。屏幕的左下角为原点。

常用矩阵表示

- UNITY_MATRIX_MVP：当前模型 -> 视图 -> 投影矩阵。（注：模型矩阵为本地-> 世界）
- UNITY_MATRIX_MV：当前模型 -> 视图矩阵
- UNITY_MATRIX_V：当前视图矩阵
- UNITY_MATRIX_P：当前投影矩阵
- UNITY_MATRIX_VP：当前视图 -> 投影矩阵
- UNITY_MATRIX_T_MV：转置模型 -> 视图矩阵
- UNITY_MATRIX_IT_MV:逆转置模型 -> 视矩阵, 用于将法线从ObjectSpace 旋转到WorldSpace。为什么法线变化不能和位置变换一样用UNITY_MATRIX_MV 呢？一是因为法线是 3 维的向量而- UNITY_MATRIX_MV 是一个 4x4 矩阵，二是因为法线是向量，我们只希望对它旋转，但是在进行空间变换的时候，如果发生非等比缩放，方向会发生偏移。
- UNITY_MATRIX_TEXTURE0 to UNITY_MATRIX_TEXTURE3：纹理变换矩阵

2.5 Shader2.0

可以实现编程。

https://www.cnblogs.com/lixiang-share/p/5025662.html

大概流程

- MeshRender 将 CPU 数据传入至 GPU
  - 然后传递给顶点着色器
    - 顶点着色器处理完成后，将自己的输出数据传递给片段着色器处理

定义顶点着色器入口函数 #pragma vertex _vertexFunctionName

定义片段着色器入口函数 #pragma fragment _fragmentFunctionName

定义平面着色器入口函数 #pragma surface _surfaceFunctionName

2.6 材质、贴图、纹理

材质 Material 包含贴图 Map，贴图包含纹理 Texture。

纹理 ->

是最基本的数据输入单位，游戏领域基本上都用的是位图。常见格式有 PNG，TGA，BMP，TIFF 此外还有程序化生成的纹理 Procedural Texture。在内存中通常表示为二维像素数组。

贴图 ->

英语 Map 其实包含了另一层含义就是“映射”。其功能就是把 **纹理**通过 UV 坐标 映射 到 *3D 物体表面*。

贴图包含了**除了纹理以外其他很多信息**，比方说 **UV 坐标、贴图输入输出控制**等等。一张图便能说明其之间的关系

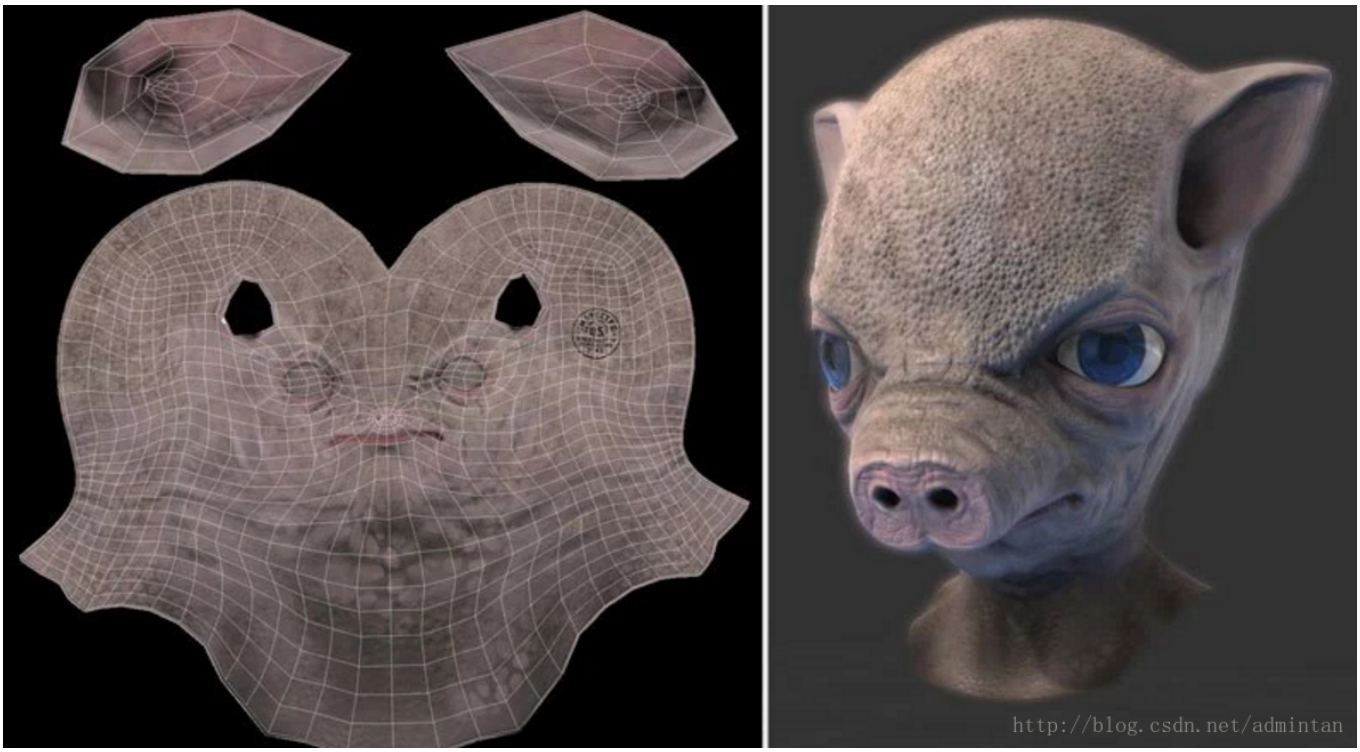


图 2.3: 纹理与贴图

材质 ->

本质就是一个**数据集**，主要功能就是**给渲染器***提供数据和光照算法*。

贴图就是其中**数据的一部分**,根据用途不同,贴图也会被分成不同的类型,比方说 Diffuse Map，

Specular Map，Normal Map 和 Gloss Map 等等。

另外一个重要部分就是**光照模型 Shader**，用以实现不同的渲染效果。贴图种类繁多：我做个不完全总结

Diffuse Map 漫反射贴图/也被称作反照率贴图 albedo map，存储了物体相应部分漫反射颜色

Normal Map 法线贴图本质上存储的是被 RGB 值编码的法向量表现凹凸，**比如一些凹凸不平的表面，光影在表面产生实时变化，常用来低精度多边形表现高精度多边形细节，比如在高多边形下生成 normal map 在匹配给低多边形模型是一种常见的降低性能要求的做法**

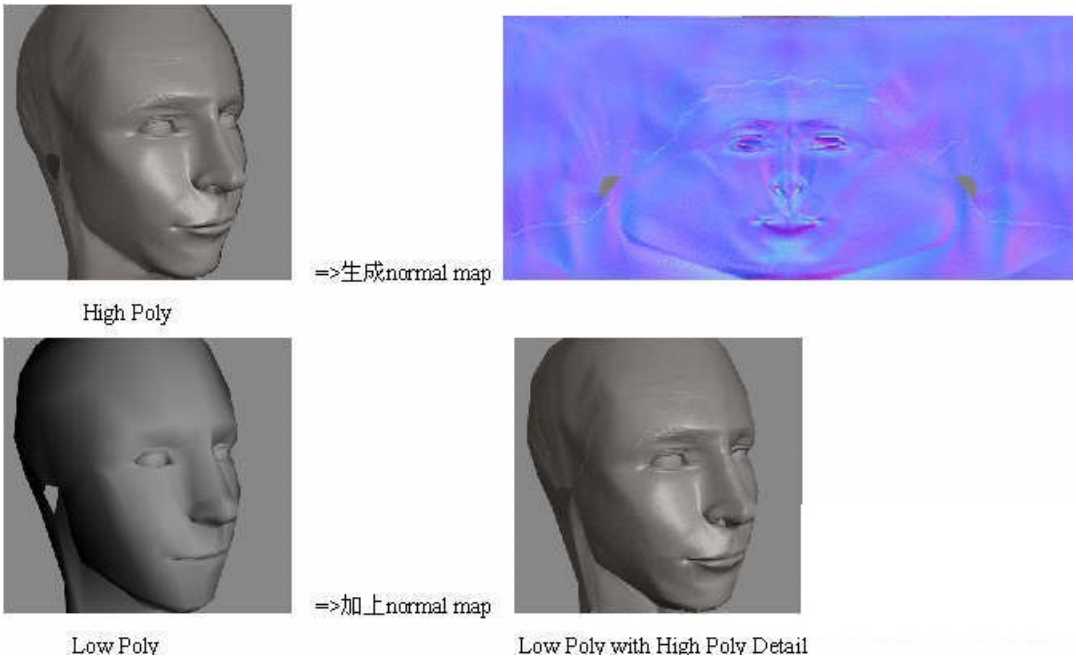


图 2.4: 法线贴图演示

建议参考：https://blog.csdn.net/cywater2000/article/details/749341#comments

常规来说就是**光向量**和**法向量**的**点积**来确定明暗以表现凹凸质感，也就是把法线存在纹理中。

Specular Map 高光贴图表现质感高光区域大小可真实反映材质区别

Gloss Map 光泽贴图，每个纹理元素上描述光泽程度

总体上说他们都是要被 shader 加工的原材料.

2.7 光照模型

参考：https://blog.csdn.net/admintan/article/details/53913624

2.7.1 Phone 光照模型

真实世界中的光照效果抽象为三种独立的光照效果的叠加。

$$Color = Ambient + Diffuse + Specular$$

环境光-Ambient 此为模拟环境中的整体光照水平，是间接反射光的粗略估计，间接反射的光使阴影部分不会变成全黑关于环境光还有个事实，1 某个可以独立分析的局部场合的环境光强和能够进入这个地方的光的强度有关。

$$Ambient = K_a * GlobalAmbient$$

漫反射光-Diffuse 模拟直接光源在表面均匀的向各个方向反射，能够逼近真实光源照射到哑光表面的反射。比如在阳光下，由于路面粗糙的性质，我们发现从任意一个角度观察路面，亮度都是差不多。

$$Diffuse = K_d * lightColor * dot(L, N)$$

镜面反射光-Specular 模拟在光滑表面会看到的光亮高光。会出现在光源的直接反射方向。镜子、金属等表面光亮的物体会有镜面反射光。镜面反射光同时与物体表面朝向、光线方向、视点位置有关。如图：

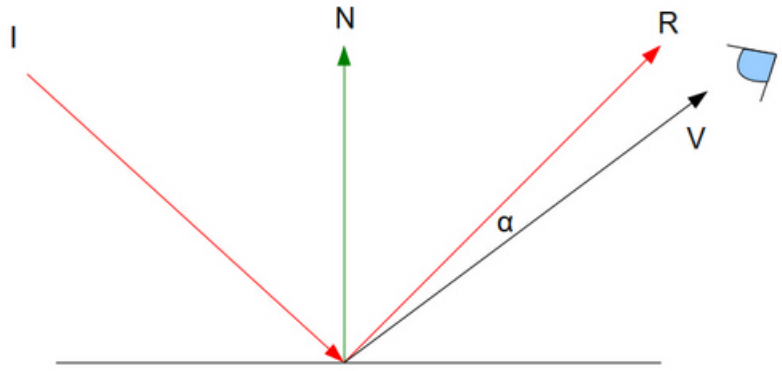


图 2.5: 高光演示

I 是入射光，N 是表面法线，R 是反射光线，V 是从物体上的目标观察点指向视点的向量，a 是 V 和 R 的夹角。

我们可以判断出一个规律，夹角 a 越小，即视线与反射方向的偏离越小，则目标点的光强越大, 计算公式为：

$$Specular = K_s * lightColor * (dot(V, R))^{shininess}$$

- Ks 为物体对于反射光线的衰减系数
- Shininess 为高光指数, 高光指数反映了物体表面的光泽程度。
  - Shininess 越大，反射光越集中，当偏离反射方向时，光线衰减的越厉害，只有当视线方向与反射光线方向非常接近时才能看到镜面反射的高光现象，此时，镜面反射光将会在反射方向附近形成亮且小的光斑；
  - Shininess 越小，表示物体越粗糙，反射光分散，观察到的光斑区域小，强度弱。

总结 至于更多的光照模型结合了更多的物理光学等信息 在模拟单种材质例如塑料 合金石膏陶瓷等等上要好于 Phone 式模型的效果但是基本上属于 Phone 式模型的扩充

2.8 参考

入门参考网址：https://blog.csdn.net/ring0hx/article/details/46440037

第三章　流水线

The Main Function of the pipeline is **to Generate** or **Render**, **a two-dimensional Image**, *given a virtual camera, three-dimensional objects, light sources, and more.*

3.1　渲染流水线

具体细节可以参考**图形学原理笔记**。

应用阶段　这个阶段主要由应用主导的，通常由 CPU 负责实现。即这个阶段开发者拥有绝对控制权。

1. **准备场景数据**
   - 模型、贴图
   - 相对位置 (世界变换)
   - 摄像机
   - 视椎体 (投影变换)
   - 光源等。
2. **粗粒度剔除工作**
   - 将不在场景中的物体剔除出去。
3. **设置好每个模型的渲染状态**

- 使用的材质
- 使用的纹理
- 使用的 Shader

几何阶段　这个阶段决定需要绘制的图元是什么，怎么绘制他们，在哪里绘制他们，这一阶段一般在 GPU 上。

1. **顶点着色器**
   - 计算顶点的颜色
   - 将物体坐标系转换到相机坐标系 (相机变换)
2. 裁剪
3. 屏幕映射

光栅化阶段　将**顶点**转换为 **像素**。

1. 三角形设置
2. 三角形遍历
   - 输入：顶点
   - 插值：将适当地方填充
   - 输出：像素
3. **片段着色器**
   - 输入：像素 (RGBA 4 通道组成)
   - **纹理采样**: 从纹理像素赋值给像素，**覆盖**以前默认像素（三角形遍历阶段插值的像素颜色）颜色值。
   - 像素跟**灯光计算**。
4. **三大测试**
   - alpha 测试：挑选合格的 **alpha 像素**显示。即透明度达到某个阈值予以显示。
   - 模板测试：达到合格 RGBA 像素，像素还可以携带**模板信息**，达到条件的模板值予以显示。

- 深度测试：测试与相机间的距离。

5. Blend 混合：将当前**要渲染的像素**和 **已经渲染出来的像素**进行混合。

6. GPU Buffer ：RGBA、模板值、深度值等

7. Front Buffer

8. Frame Buffer

9. 显示器

Notice 当场景中有一个 4 个顶点的组成的矩形，并假设其映射在屏幕的区域为 100*100，那么在 GPU 计算的过程中:

- 在几何阶段的顶点着色器阶段运行 4 次.
- 在光栅化阶段的片段着色器阶段运行 10000 次.

所以在顶点着色器和片段着色器对的运算级别不是一个级别。故而能把运算放到顶点着色器阶段的时候尽量不要放到片段着色器中。ss

3.2 OpenGL 渲染流程

3.3 渲染队列

首先看一下 Unity 中的几种内置的渲染队列，按照渲染顺序，**从先到后进行排序，队列数越小的，越先渲染，队列数越大的，越后渲染**。

- **Background**（1000）最早被渲染的物体的队列。
- **Geometry** （2000）**不透明物体的渲染队列**。大多数物体都应该使用该队列进行渲染，也是 Unity Shader 中默认的渲染队列。
- **AlphaTest** （2450）**有透明通道**，需要进行 Alpha Test 的物体的队列，比在 Geomerty 中更有效。
- **Transparent**（3000）**半透物体的渲染队列**。一般是不写深度的物体，Alpha Blend 等的在该队列渲染。

- **Overlay** （4000）最后被渲染的物体的队列，一般是覆盖效果，比如镜头光晕，屏幕贴片之类的。

Unity 中设置渲染队列也很简单，我们不需要手动创建，也不需要写任何脚本，只需要在 shader 中增加一个 Tag 就可以了，当然，如果不加，那么就是默认的渲染队列 Geometry。比如我们需要我们的物体在 Transparent 这个渲染队列中进行渲染的话，就可以这样写：`Tags { "Queue" = "Transpa`

第四章　CPU 与 GPU

第五章　Shader 着色器

5.1　基础概念

CPU　将 FBX、OBJ 等文件信息 (UV，顶点位置，法线，切线等) 读入`MeshRender`.

MeshRender　将这些信息传递到 GPU。主要有两种

- Skin Mesh Render : 带蒙皮的骨骼。
- Mesh Render + Mesh Filter：不带蒙皮的模型、没骨骼的，如 cube 等基础场景信息。其中 meshRender 将数据传递到 GPU，而 filter 做的是决定将哪个模型传递给 GPU。

哪些我们可以操作

- 顶点着色器
- 片段着色器
- 三大测试

Unity Shader 分类

- Fixed Shader: Shader 1.0，开关式
- 顶点、片段着色器：Shader 2.0，功能里面的**公式**可以自己定义。
- Surface Shader: Fixed + 顶点、片段着色器

Shader 结构　Unity Shader Lab 的语法结构。

```
Shader "Custom/My_XX_Shader"
{
    // 定义面板显示的属性
    Properities
    {

    }

    // 定义某种条件为显卡，如果合适就进行运行。
    // 可以定于多个，但是最少要有一个
    SubShader
    {
        // subShader 的标签
        [Tags]

        // 给多个 pass 公用的设置，如是否 开启测试等。
        [Common State]

        // 可能存在多个pass，每个pass 都会引起一次渲染过程。
        Pass
        {
            // Pass 的标签
            [Pass Tags]

            // 渲染设置，如颜色混合
            [Render Setup]

            // 纹理设置，只有在 Fixed Function Shader 中才可以使用。
            [Texture Setup]
        }
    }

    // 当所有的 subShader 失败的时候，使用该项指定的shader.
    [Fallback]


}
```

材质球与 Shader 的关系

材质球 比喻为人的衣服

Shader　材质（衣服）跟灯光的作用，Shader **决定**这个衣服怎么去作用。

Properities 使用说明　定义使用 3 步曲

1. 定义每个属性的名称
2. 定义每个属性对应的变量名称与类型
3. 定义变量的默认值

```
name ("Display Namee", Int) = number
```

```
Properities
{
    _TestInt ("TestInt", Int) = 1
    _TestFloat ("TestFloat", Float) = 1.0
    _TestRange ("TestRange", Range(1,5)) = 2
    _TestVector ("TestVector", Vector) = (1,1,1,1)
}
```

SubShader 使用说明

Tags　设置按照什么方式、什么时候渲染。

```
Tags {"Queue" = "Transparent"}
Tags {"RenderType" = "Opaquee"}
Tags {"DisableBatching" = "True"}
Tags {"ForceNoShadowCasting" = "True"}
Tags {"IgnoreProjector" = "True"}
Tags {"CanUseSpriteAtlas" = "True"}
Tags {"PreviewType" = "Plane"}
```

RenderSetup　设置显卡的各种状态

```
Cull Back //Front Off 设置剔除方式
ZTest Less Greater // GEqual 设置深度测试使用的函数
ZWrite On // Off 开启关闭深度写入
Blend SrcFactor DstFactor // 开启并设置混合模式
```

Pass

5.2 顶点着色器

计算顶点颜色 在 Pass 中有如下两种方式：

- 直接使用自定义颜色值：`Color (0,1,0,1)`
- 使用属性变量：`Color [_Var] //_Var ("TestColor", Color) = (1,1,1,1)`

顶点变换

灯光作用

$$\textbf{Ambient} * AmbientIntensity + (LightColor * \textbf{Diffuse} + LightColor * \textbf{Specular}) + \textbf{Emission}$$

Ambient 环境光

Diffuse 漫反射

Specular 镜面反射

Emission 自发光

渲染脚本灯光开关

- 所有灯光的开关 `Lighting On`
- 高光开关 `SeperateSpecular On`

```
Pass{
    Color[_TestCorlor]

    Material
    {
        Ambient[_TestColor]
        Diffuse[_TestColor] // Or Diffuse(1,1,1,1)
        Specular[_TestColor]
        Emission[_TestColor]
    }
```

```
    Lighting On
    SeparateSpecular On
}
```

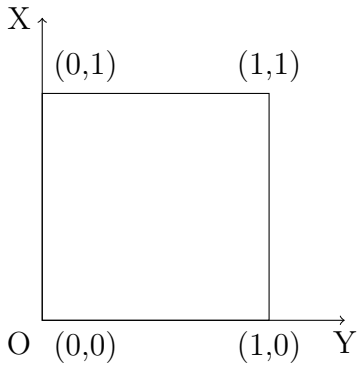
5.3 片段着色器-1.0 各显卡均可

纹理采样

纹理等于显示区域时 1 对 1 映射

等比例映射：UV 坐标

.



纹理大于显示区域时 这样的话，一个屏幕像素需要显示纹理的多个像素，此时就存在多种选择方式，常用的方式有

- Point：就近采样,
- Bi-linear: 就近周围 4 个像素的平均
- Tri-linear: 就近周围 8 个像素的平均

纹理小于显示区域时 那么屏幕像素将出现多对一的情况，即多个屏幕像素格都将显示同一个纹理像素，这样将产生锯齿和马赛克现象。

所以在实际的应用中，会在这种情况应用很多抗锯齿的算法。

设置纹理 格式：`SetTexture[VarTextureName]{block}`

在 block 中常常使用 Combine 进行混合，并且常常使用以下预定于的变量，需要注意以下几点

- Previous: 表示前一个 SetTexture 的 Texture 变量
- Primary: 表示顶点计算出来的颜色
- Texture: 等于 SetTexture 当前的 Texture
- Constant: 表示一个固定的颜色值

在使用 combine 利用中间图进行插值计算是,具体使用如下:函数格式:`combine src1 lerp(src2) src` 混合 src1 与 src3. 混合的方式取决于 Src2 的 alpha 值，并且 lerp 的具体公式如下 $(1-t)A+tB$

```
Pass{
    Color[_TestColor]
    SetTexture[_TestTexture]
    {
        // Texture Block
        combine Primary*Texture // 融合顶点颜色与纹理颜色
        combine Texture // 只使用纹理颜色，不使用顶点颜色
    }

    SetTexure[_TestTexture2]
    {
        combine Previous*Texture // 将之前调用的SetTexture 的纹理变量与当前Texture 混合
    }

    SetTexture[_TestTexture]
    {
        combine Texture lerp(Previous) Previous // 利用之前的纹理进行之前纹理与当前纹理的
            混合。
    }

    SetTexture[_TestTexture]
    {
        constantColor(1,1,1,1)
        combine Texture + constant
    }
}
```

示例-轮廓

方法一：一大一小 写两个 Pass，先渲染大的，然后再渲染小的。

```
SubShader{
```

```
// 先渲染轮廓
Pass
{
    CGPROGRAM
    #pragma vertex vert
    #pragma fragment frag

    #include "UnityCG.cginc"

    struct appdata
    {
        float4 vertex : POSITION;
        float2 uv: TEXCOORDO;
    };

    struct v2f
    {
        float2 uv: TEXCOORDO;
        float4 vertex: SV_POSITION;
    };

    // 定义的顶点着色器
    v2f vert(appdata v)
    {
        v2f o;
        v.vertex.xy *= 1.2f; // 将顶点坐标放大的x、y 放大
        o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
        o.uv = v.uv;
        return o;
    }

    // 定义的片元着色器
    fixed4 frag(v2f i):SV_Target
    {
        reuturn fixed4(1,0,0,1); //给每个片元都返回红色
    }
    ENDCG
}

// 再渲染原纹理
Pass
{
    ZTest Always // 深度测试总是通过
```

```
        CGPROGRAM
        #pragma vertex vert
        #pragma fragment frag

        #include "UnityCG.cginc"

        struct appdata
        {
            float4 vertex : POSITION;
            float2 uv: TEXCOORDO;
        };

        struct v2f
        {
            float2 uv: TEXCOORDO;
            float4 vertex: SV_POSITION;
        };

        // 定义的顶点着色器
        v2f vert(appdata v)
        {
            v2f o;
            o.vertex = mul(UNITY_MATRIX_MVP, v.vertex);
            o.uv = v.uv;
            return o;
        }

        //引用变量需要在这块重新声明
        sampler2D _MainTex;
        // 定义的片元着色器
        fixed4 frag(v2f i):SV_Target
        {
            fixed4 col = tex2D(_MainTex, i.uv);
            return col;
        }
        ENDCG
    }

}
```

方法二：梯度求边缘

- 找到边缘

- 给边缘着色
- 非边缘地带正常纹理采样

示例-屏幕后期特效 Render 运行完成后，会生成一张已经渲染完成的**图片**，此后会进入 RenderImage 步骤，完成屏幕后期渲染。

具体的实践过程如下：

1. 创建脚本，重写`OnRenderImage()`方法
2. 在脚本中创建一个新的材质 (Material), 并将新的 shader 添加到该材质球中，在此处即是下一步 Blit 用到的 Shader.
3. 在该方法中用 Unity 提供的接口完成对图片的渲染，`Graphics.Blit()`
4. 最后将该脚本挂到主相机上。

一般实现方法如下 :

1- 重写 OnRenderImage:

```
void OnRenderImage(RenderTexture sourceTexture, RenderTexture desTexture)
{
    Graphics.Blit(sourceTexture, desTexture, newShader);
}
```

2- 添加新的片元 Shader, 上述的 newShader

```
float _Offset;
fixed4 frag(v2f i):SV_Target
{
    float2 tmpUV = i.uv;
    fixed4 col1 = tex2D(_MainTex, tmpUV);
    fixed4 col2 = tex2D(_MainTex, tmpUV+float2(0, -Offset));
    fixed4 col3 = tex2D(_MainTex, tmpUV+float2(0, Offset));
    fixed4 col4 = tex2D(_MainTex, tmpUV+float2(-Offset,0));
    fixed4 col5 = tex2D(_MainTex, tmpUV+float2(Offset,0));

    return (col1 + col2 + col3 + col4 + col5)/6.0;
}
```

5.4 片段着色器-2.0

5.5 动态纹理

河流

核心 平移纹理坐标 `i.uv*Time.xy`

动态加载

- UV 旋转
- 在 GPU 中使用旋转矩阵进行旋转。
- Notice: 旋转矩阵是以原点为基准的。

旋转矩阵 以原点的为旋转中心点，计算公式如下

以 Z 轴为旋转轴的旋转矩阵

$$\begin{bmatrix} X_n \\ Y_n \\ Z_n \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

以 X 轴为旋转轴的旋转矩阵

$$\begin{bmatrix} X_n \\ Y_n \\ Z_n \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\theta) & -sin(\theta) \\ 0 & sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

以 Y 轴为旋转轴的旋转矩阵

$$\begin{bmatrix} X_n \\ Y_n \\ Z_n \end{bmatrix} = \begin{bmatrix} cos(\theta) & 0 & sin(\theta) \\ 0 & 1 & 0 \\ -sin(\theta) & 1 & cos(\theta) \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

旋转过程

1. 先将纹理以中心为基准平移至原点

2. 旋转

3. 再将纹理原路返回

```
fixed frag(v2f i):SV_Target
{
    float2 tmpUV = i.uv;
    // 将以纹理中心为基准平移至原点
    tempUV -= float2(0.5, 0.5);
    // 防止UV坐标上的对角线超过旋转半径
    if(length(tempUV) >0.5)
        return fixed4(0,0,0,0);

    // 旋转操作
    // 1- 平移
    float2 finalUV = 0;
    float angle = _Time.x*_speed;
    // 2- 旋转
    finalUV.x = tempUV.x*cos(angle) - tempUV.y*sin(angle);
    finalUV.y = tempUV.x*sin(angle) + tempUV.y*cos(angle);
    finalUV.z = tempUV.z;
    // 3- 平移回原点
    finalUV += float2(0.5,0.5);

    fixed4 col = tex2D(_MainTex, finalUV);
    return col;
}
```

5.6 三大测试

5.6.1 透明测试

Alpha test 是可以通过测试然后丢弃不需要渲染的像素的。顾名思义就是根据他的 alpha 值与某特定值比较。

5.6.2 模板测试

模板缓冲一般为 8 位的，存贮整数，最大值为 255。在使用的过程中步骤一般时，开启模板缓冲，绘制一个物体作为我们的模板，这个过程实际上就是写入模板缓冲的过程；接着我们利用

模板缓冲中的值决定是丢弃还是保留后续绘图中的片元。下面我们建立一个举行模板，通过矩形模板选择性地将上一节绘制的场景显示出来，这个过程示意如下图所示

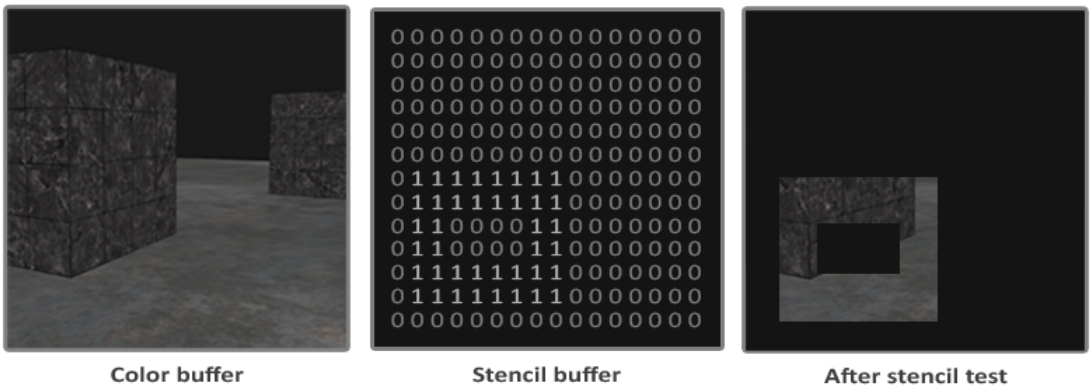


图 5.1: 模版测试原理图

如图中所示，**模板缓冲中为 1 的地方我们选择保留图形，而其他部分则丢弃**，形成最终的效果。

使用模板缓冲需要三个要素：

- 正确的时间开启和关闭深度缓冲
- 模板测试函数
- 模板测试函数失败或者成功后的执行的动作

一般地绘制模板以及利用模板选择性地绘制物体时则开启模板缓冲，绘制其他物体时关闭模板缓冲。使用模板缓存的步骤一般如下：

1. 开启模板测试
2. 绘制模板，写入模板缓冲 (不写入 color buffer 和 depth buffer)
3. 关闭模板缓冲写入
4. 利用模板缓冲中的值，绘制后续场景

5.6.3 深度测试

在绘制 3D 场景的时候，我们需要决定哪些部分对观察者是可见的，或者说哪些部分对观察者不可见，**对于不可见的部分，我们应该及早的丢弃**，例如在一个不透明的墙壁后的物体就不应该渲染。这种问题称之为**隐藏面消除** (Hidden surface elimination)

解决这一问题比较简单的做法是画家算法 (painter’s algorithm)。画家算法的基本思路是，先绘制场景中离观察者较远的物体，再绘制较近的物体。例如绘制下面图中的物体，先绘制红色部分，再绘制黄色，最后绘制灰色部分，即可解决隐藏面消除问题。

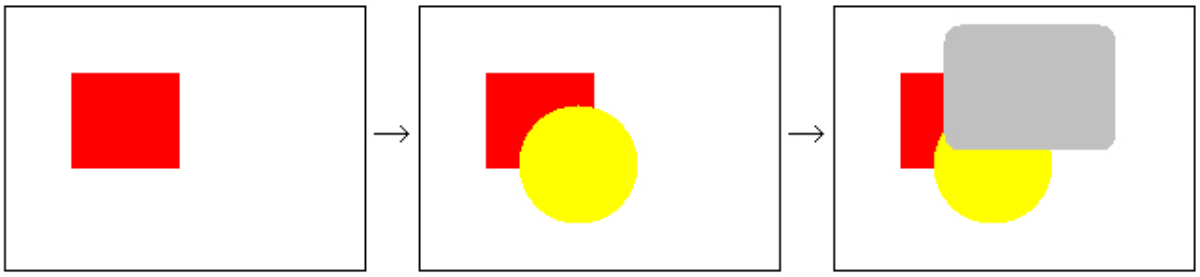


图 5.2: 画家算法效果

使用画家算法时，只要将场景中物体按照离观察者的距离远近排序，由远及近的绘制即可。画家算法很简单，但另一方面也存在缺陷，例如下面的图中，三个三角形互相重叠的情况，画家算法将无法处理：

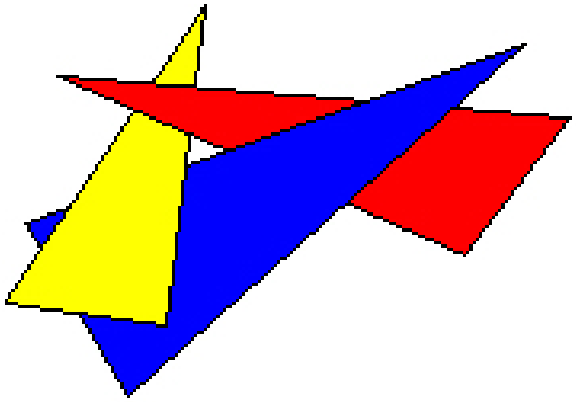


图 5.3: 交叉问题

结合 OpenGL，我们使用的是 Z-buffer 方法，也叫深度缓冲区 Depth-buffer。

深度缓冲区 (Detph buffer) 同**颜色缓冲区** (color buffer) 是对应的，**颜色缓冲区**存储的像素的颜色信息，而**深度缓冲区**存储像素的深度信息。**在决定是否绘制一个物体的表面时**，首先将表面对应像素的深度值与当前深度缓冲区中的值进行比较，**如果大于等于深度缓冲区中值，则丢弃这部分**; **否则利用这个像素对应的深度值和颜色值，分别更新深度缓冲区和颜色缓冲区**。这一过程称之为**深度测试** (Depth Testing)。

5.7 混合-Blend

就是将当前要绘制的物体的颜色和颜色缓冲区中已经绘制了的物体的颜色进行混合，最终决定了当前物体的颜色。例如下面的图中，狙击枪的瞄准器本身是带有蓝色的，将它和后面的任务混合在一起，形成了我们看到的最终效果，这个效果里既有瞄准器的蓝色成分，也有后面人物的像素，主要是后面人物的像素。

图 5.4: 混合示例

5.8 背面剔除

5.9 帧缓冲对象

5.10 渲染队列-RenderQueue

第六章　光照

如何在 shader 中实现光照模型，如漫反射、高光反射等。

$$\textbf{Ambient} + (LightColor * \textbf{Diffuse} + LightColor * \textbf{Specular}) + \textbf{Emission}$$

第七章　纹理

可以理解为贴图，但是纹理远不止皮肤，还有皮肤的沟壑信息等，也是需要渲染的，以模拟的更加真实。

第八章　深度、法线纹理

第九章　透明效果

如何实现透明度测试，如何透明度混合。

第十章　动态效果

使用时间变量来实现纹理动画。

第十一章　屏幕后处理

高斯模糊、边缘处理。

第十二章　噪声处理

第十三章　渲染优化

第十四章　GPU 相关

大部分时间在**知其然不知其所以然**。该部分将从 *GPU* 讲到游戏引擎再到游戏逐层介绍，以打通任督 2 脉。

介绍绘画流程的 video：https://v.youku.com/v_show/id_XNjY3MTY4NjAw.html

本节参考：https://blog.csdn.net/admintan/article/details/53861781

14.1　最底层——GPU/硬件原理

14.1.1　硬件基础

ALU-Arithmetic Logic Unit 算数逻辑单元　->

整数算术运算（加、减，有时还包括乘和除，不过成本较高）、位逻辑运算（与、或、非、异或）、移位运算（将一个字向左或向右移位或浮动特定位，而无符号延伸），移位可被认为是乘以 2 或除以 2。ALU 可以说是计算机处理器的核心部件之一。

Cache-缓存 SRAM　->

SRAM 叫静态内存，“静态”指的是当我们将一笔数据写入 SRAM 后，除非重新写入新数据或关闭电源，否则写入的数据保持不变。

由于 CPU 的速度比内存和硬盘的速度要快得多，所以在存取数据时会使 CPU 等待，影响计算机的速度。SRAM 的存取速度比其它内存和硬盘都要快，所以它被用作电脑的高速缓存 (Cache)。

14.1.2　CPU with GPU 异同

参考：https://www.jianshu.com/p/fae645d70e0e

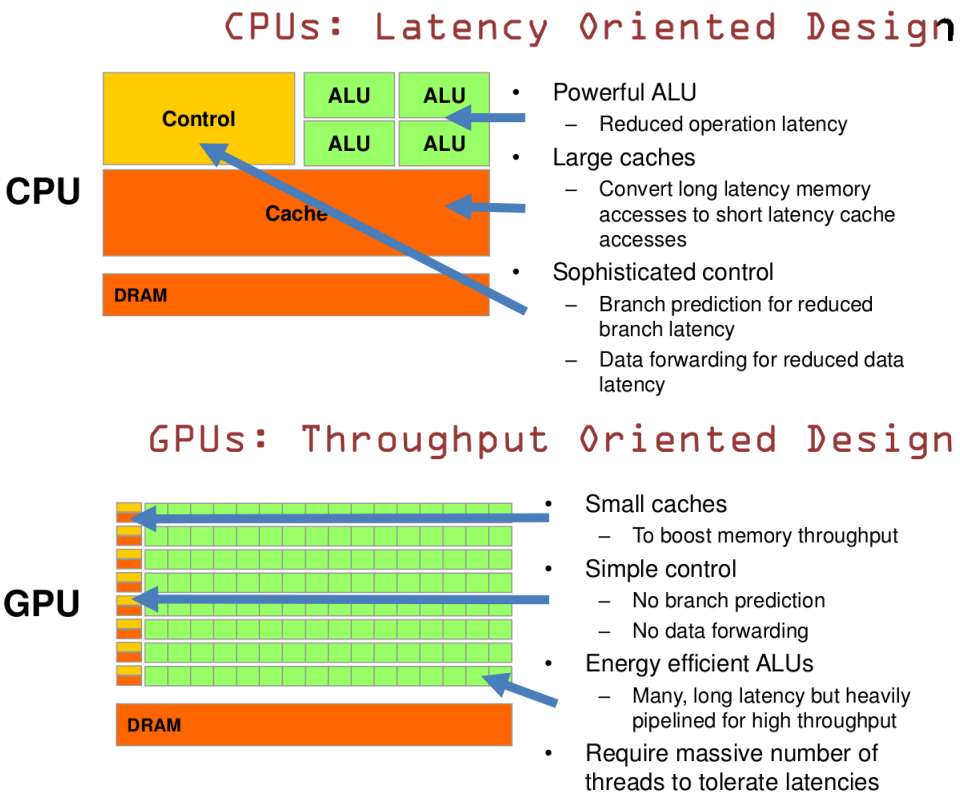


图 14.1: CPU with GPU

CPU -> 量小但能力强

- ALU **部分会很强大**，可以在很少的时钟周期内完成算数计算。对于一个 64bit 双精度的 CPU 来说浮点加法和乘法只需要 1-3 个时钟周期。
- **大的 Cache** 也将延时降低很多, 结合了现在的各种高级调节技术比如超线程、多核等技术 CPU 对于复杂逻辑的运算能力得到了极大提升。

GPU -> 量大但能力弱

- ALU 的**数量会非常大、功能会更少、能耗很低、cache 就会很小**，这样带来的好处就是针对大吞吐量的需要简单计算的数据来说，处理效率就高了非常多。

  如果有很多线程需要访问同一个相同的数据，*缓存会合并这些访问，然后再去访问 DRAM*（因为需要访问的数据保存在 DRAM 中而不是 Cache 里面），**获取数据后 Cache 会转发这个数据给对应的线程，这个时候是数据转发的角色**。但是由于需要访问 DRAM，自然会带来延时的问题。

- GPU 的控制单元（左边黄色区域块）**可以把多个的访问合并**成少的访问。

- GPU 的虽然有 DRAM 延时，**却有非常多的 ALU 和非常多线程**. 为了平衡内存延时的问题，我们可以中充分利用多的 ALU 的特性达到一个非常大的吞吐量的效果。尽可能多的分配多的线程. 通常来看 GPU ALU 会有非常重的 pipeline 就是因为这样。

结论 ->

我们可以先得出一个简单结论:对显卡来说-**更适合做高并行，高数据密度，简单逻辑的运算**。

14.1.3 GPU 架构

4D 向量和 4+1 ->

3D 物件的成像過程中，VS（Vertex Shader，**顶点着色引擎**）PS（Pixel Shader，**像素著色引擎**或片段着色器）**最主要的作用就是运算坐标（XYZW）@（RGBA）**。

此时数据的**基本单位是 scalar（标量）**，1 个单位的变量操作，为 1D 标量简称 1D。而跟标量相對的就是 vector（向量），**向量是由多個标量构成**。*例如每個周期可执行 4 個向量平行运算，就称为 4D 向量架构*。若 GPU 指令发射口只有 1 个，卻可執行 4 個数据的平行运算，这就是 SIMD(单指令多数据流) 架构。

运算单元计算机制 ->

以 GPU 的矩阵加法为例：

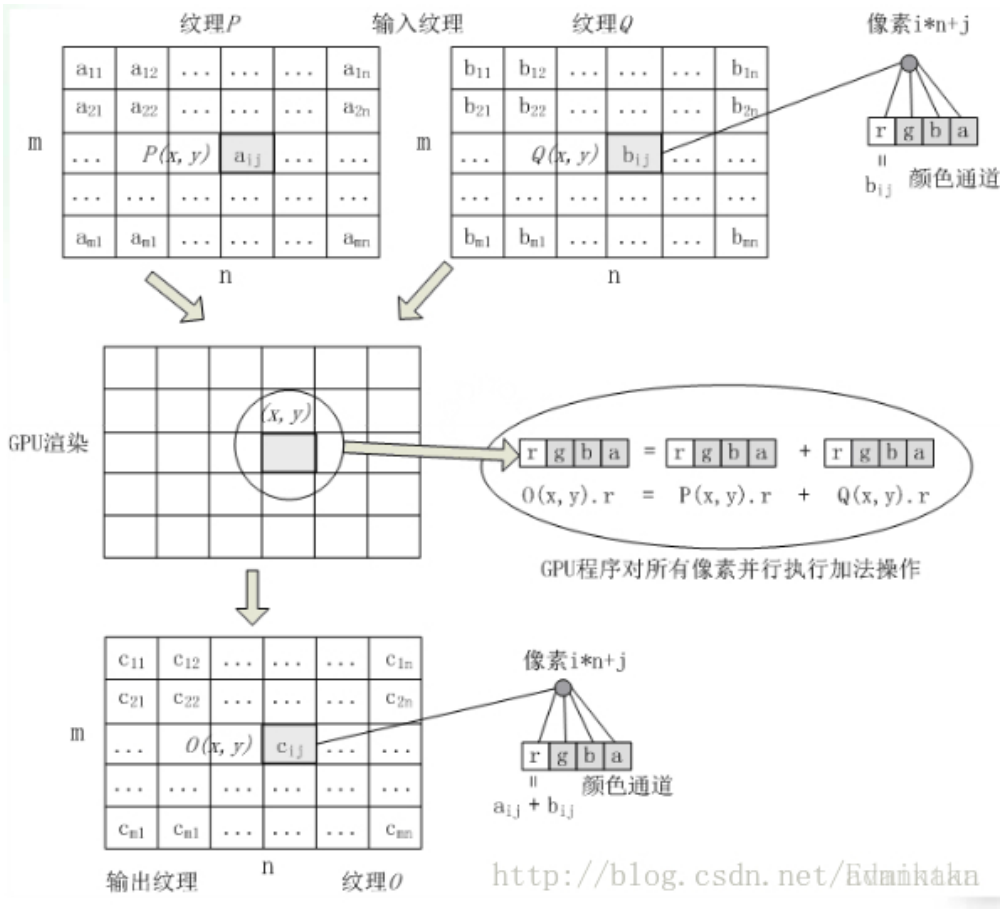


图 14.2: 运算单元计算机制

NVIDIA ->

NVIDIA 架构如下

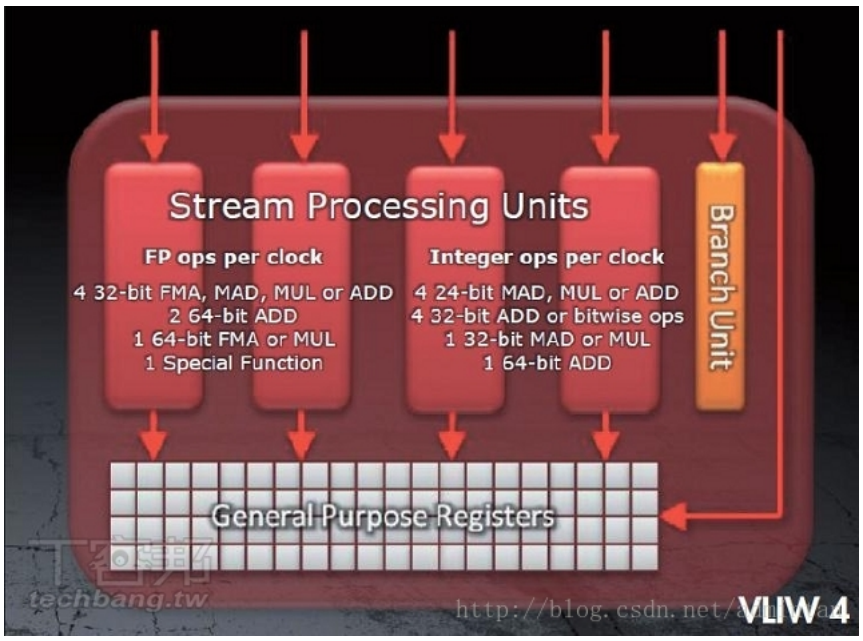


图 14.3: NVIDIA 架构演示

标注的是Stream Processing(流处理器) 数量,NVIDIA 的流处理器每个都具有完整的 ALU(可以理解为数学、逻辑等运算)。NVIDIA 从 G80 以后采用全标量设计，**所有运算全都转为标量计算。但是这么做一旦遇到 4D 矢量运算时，就需要 4 次运算才能完成**，所以 NVIDIA 显卡的 Shader 频率几乎比核心频率高一倍，就是为了弥补这个缺点。NV 的流处理器都具有完整的 ALU 功能，所以每个流处理器消耗的晶体管数量较多，成本较高。在加上现在的 CUDA 功能所以晶体管数量大幅多于 AMD-ATI。

AMD/ATI ->

AMD 架构如下

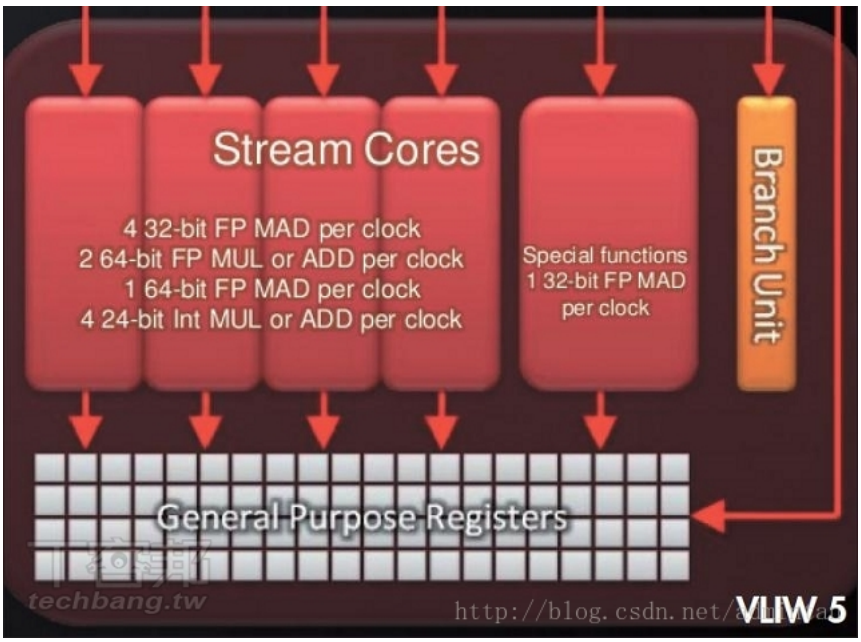


图 14.4: AMD 架构演示

标注的是 Stream Processing Units(统一渲染单元) 数量，也可以叫流处理器单元。AMD-ATI 从 RV670 以后，流处理器是 **5 个固定的统一渲染单元为一组**，4D 矢量 +1D 标量组合。**其中 4 个只能进行 MADD(乘加) 运算，1 个可以进行超运算 (函数等运算)**。*因为是 5 个固定为一组，不能拆分，所以遇到纯标量运算时就会有 4 个 SPU 处于闲置状态而无法加入其它 SP 组合协助运算*。但换句话说如果分配得当让每个 SPU 都充分工作，那么 AMD 显卡的效率可是非常高的。**这也是玩家公认 A 卡驱动提升性能比 N 卡要高**，但也就是这个原因导致 A 卡驱动设计难度非常高，游戏想要为 A 卡优化的难度也一样很高。

14.2 更高层-硬件流程

14.2.1 数据存储转换

资源信息和指令信息由**硬盘**经过 CPU 调度传输到**内存**中转，再传输进**显存**中备用.

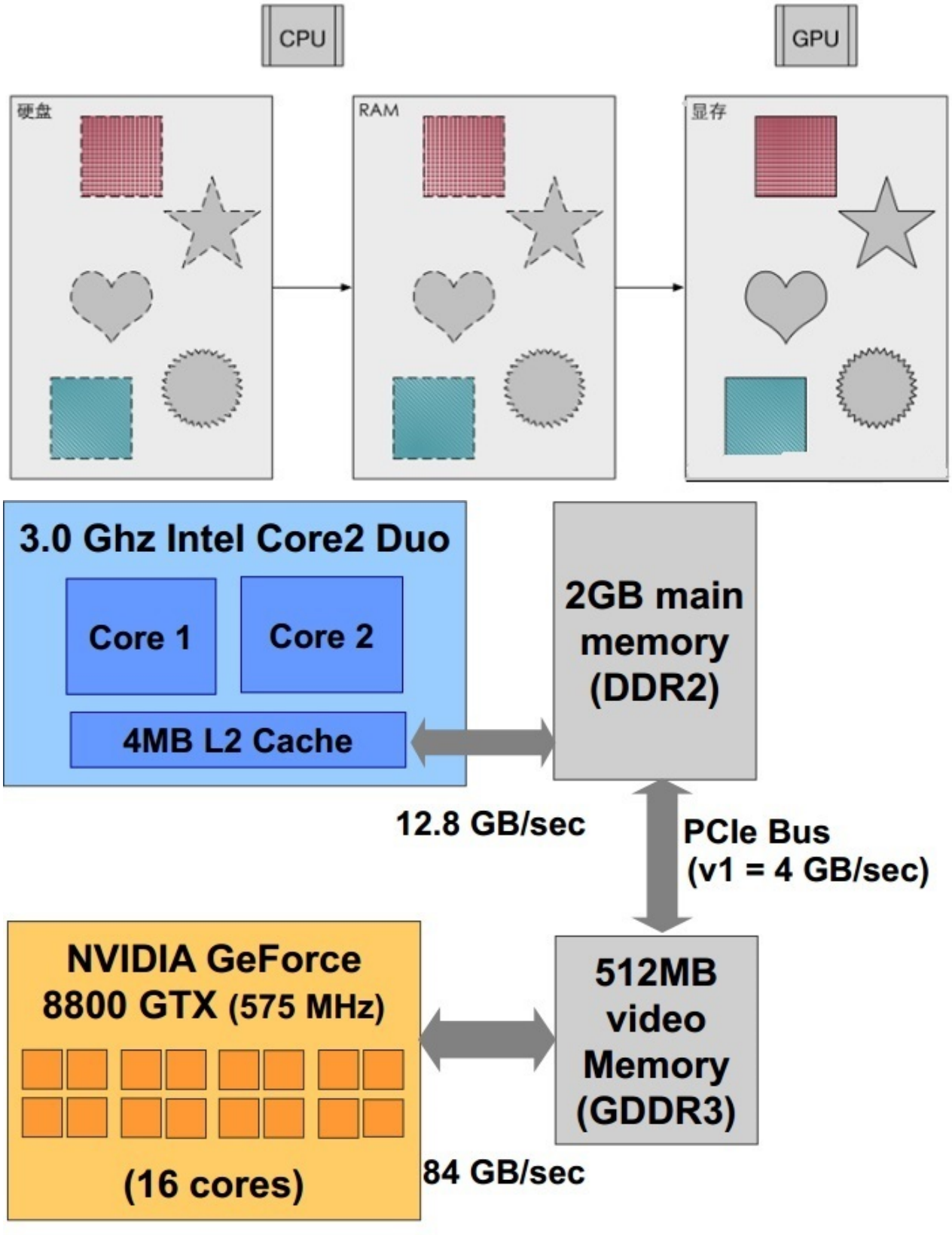


图 14.5: 资源转移

由图中的典型的带宽可见 **GPU 和显存之间的内部带宽**要比单纯的**系统总线带宽**要高很多。所以典型情况下**渲染资源和渲染指令都被加载进显存**, 所以 GPU *在渲染过程中* **只需向显存调度**

渲染指令,避免了和系统总线频繁 IO。

14.2.2 进入渲染预备状态

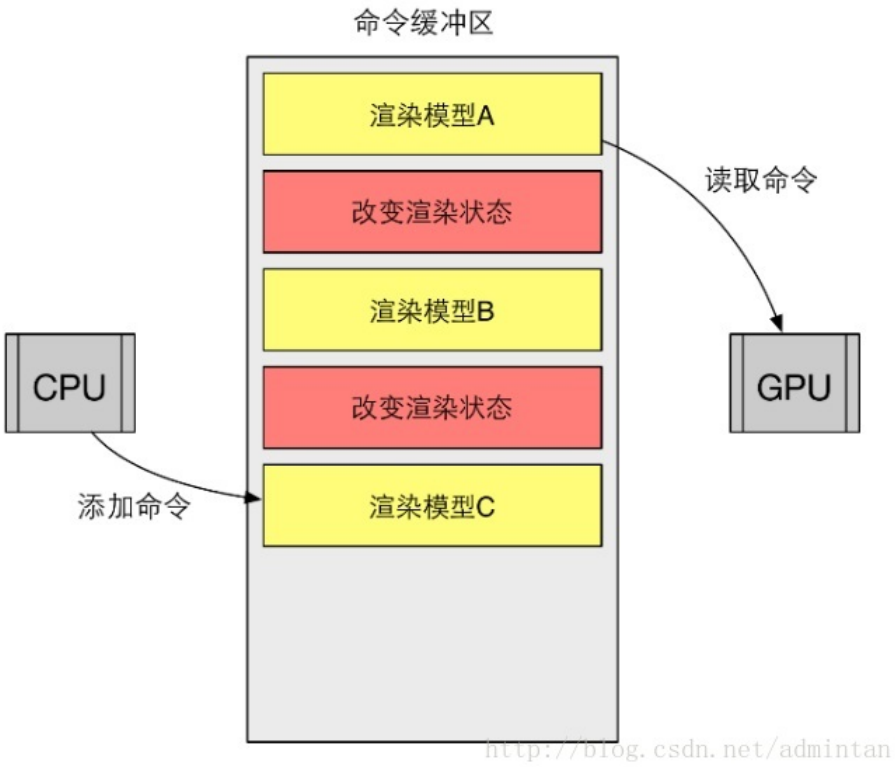


图 14.6: 渲染预备状态

此时 GPU 的**显存越大则依靠 CPU 的加载额外渲染指令的需要就越少**，显存中的信息一般包括：显存和内存一样用于存储 GPU 处理过后的数据，在显存中有几种不同的储存区域，用于储存不同阶段需要的数据。

1. **顶点缓冲区**：用于储存从内存中传递过来的顶点数据。

2. **索引缓冲区**：用于储存每个顶点的索引值，我们可以根据索引来使用相应的顶点

3. **纹理缓冲区**：用于储存从内存中传递过来的纹理数据

4. **深度缓冲区**：用于存储每个像素的深度信息

5. **模板缓冲区**：用于存储像素的模板值，且模板缓冲区域深度缓冲区公用一片内存。

6. **颜色缓冲区**：用于储存像素的颜色数据

14.3 软件

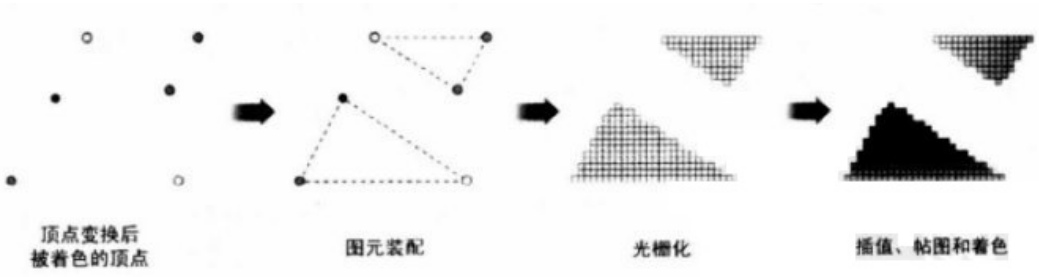
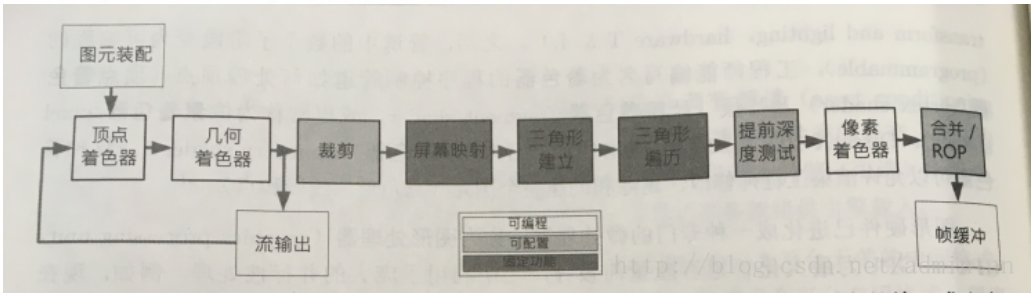


图 1-6 形象化图形流水线



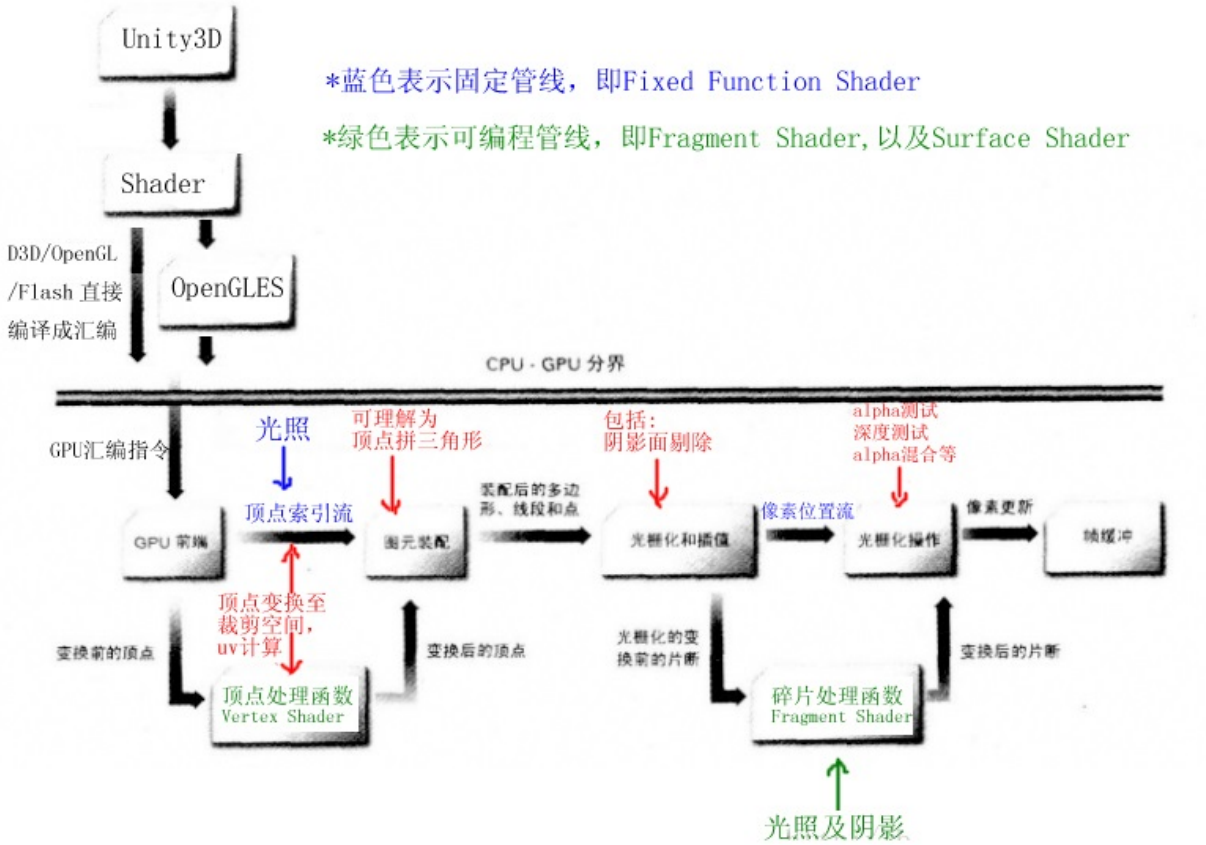


图 14.7: Shader 流程 (1,4)

第十五章　基于物理的渲染

第十六章　数学理论