

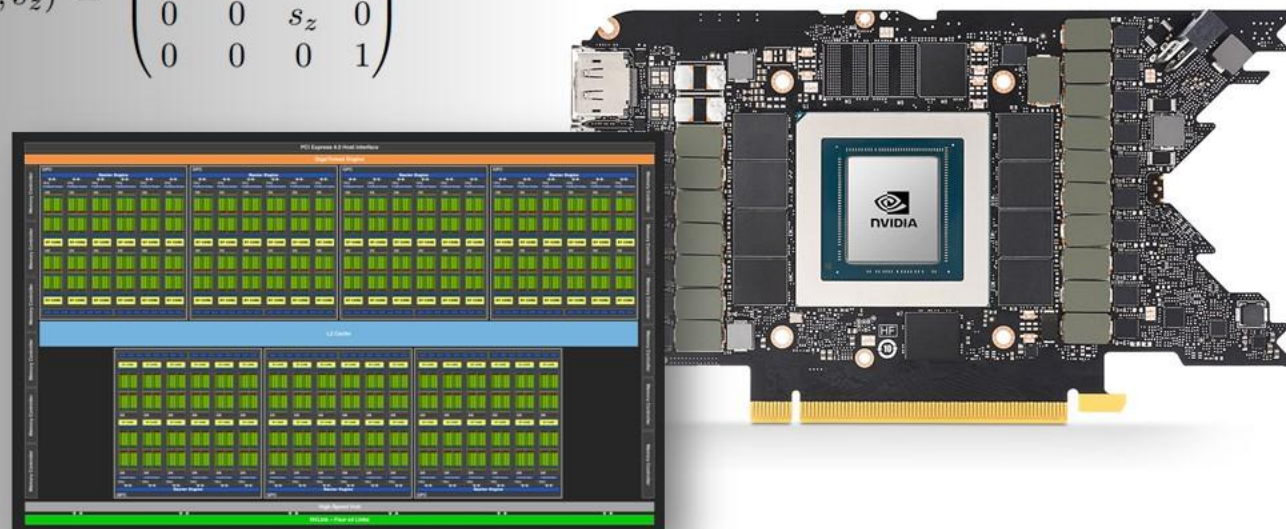
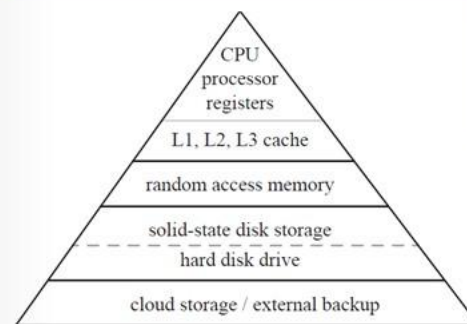
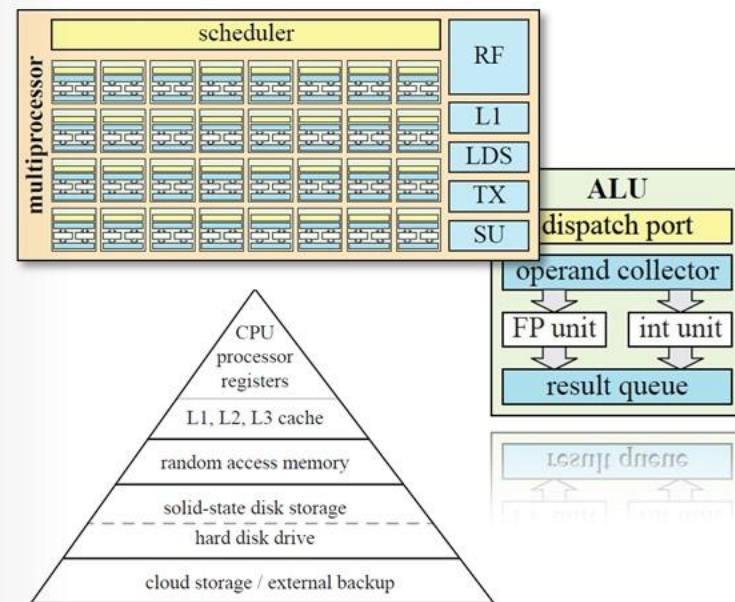
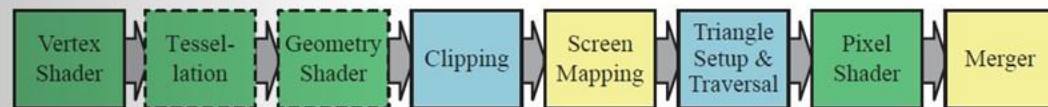
# Source 2 引擎图形原理基础

## 渲染管线与图形处理器

The Rendering Pipeline & GPU

$$\mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

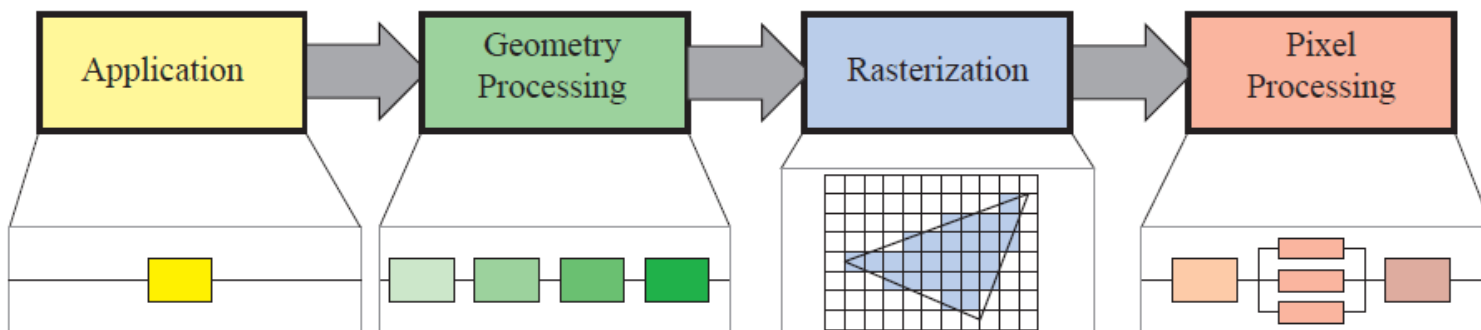
$$\mathbf{S}(s_x, s_y, s_z) = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$



B1UEMICR0, S2CT TEAM

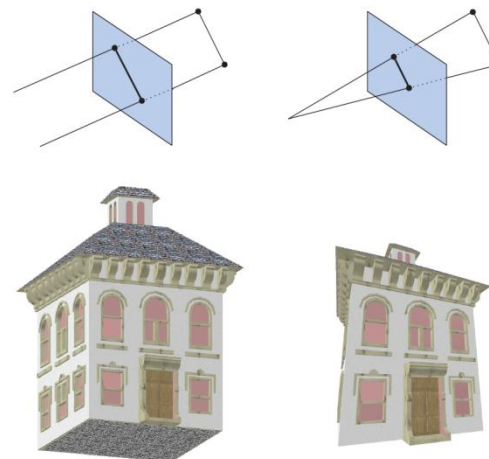
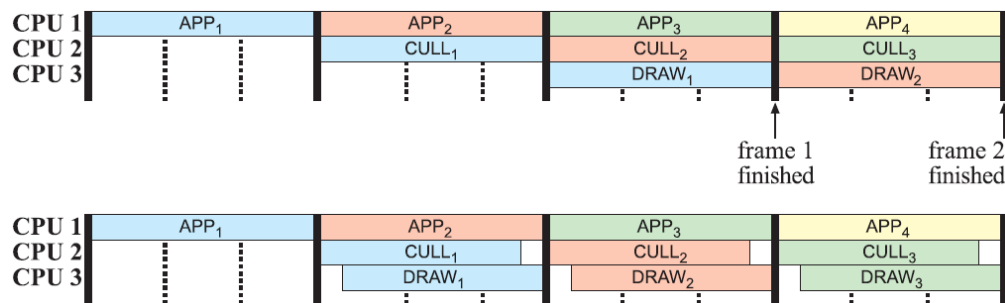
# 引入：渲染管线概论

渲染管线(The Rendering Pipeline)是现代计算机图形学中一个非常重要的概念。渲染引擎本质上是以流水线形式组织起来的，于是就有了渲染管线这个称谓。



Technically, 我们具体把它分为 4 个阶段。分别是 **Application**, **Geometry**, **Rasterization** 以及 **Pixel** 阶段。其中 Application 阶段是在 CPU 上完成的, 其余阶段大多使用 GPU.

# 渲染管线概论



最开始的 Application 阶段，由 CPU 发送绘制指令(drawcall)来驱使整个渲染管线运作。

在这个阶段 CPU 可以剔除(Cull)掉视锥体以外的物体，节省接下来工作的不必要开销。

接下来就是 Geometry 阶段了。

为什么要使用视锥体呢？因为我们一般使用透视法投影，有近大远小的效果，故需要一个锥体来表达。

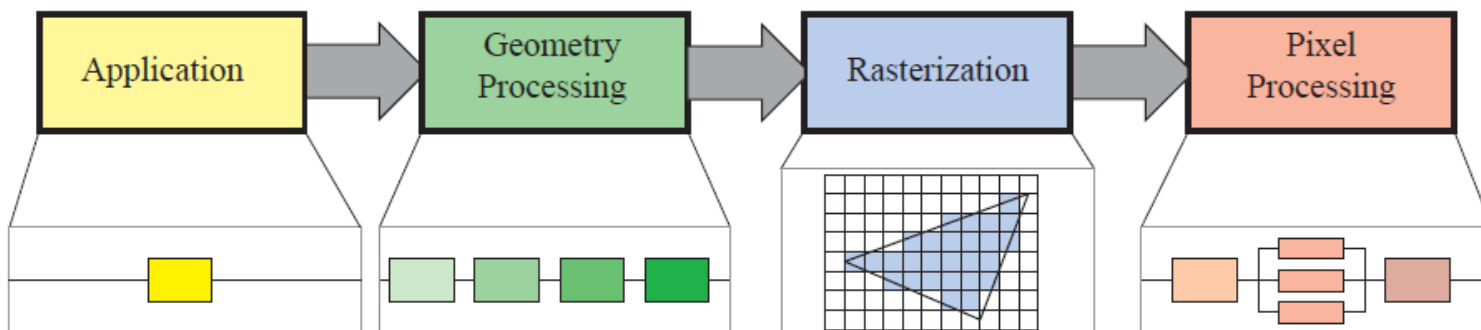
Geometry 阶段分为顶点着色(Vertex)，投影(Projection)，裁剪(Clip)和屏幕映射(Screen Mapping)四个阶段。这个阶段定义了渲染对象该以什么形式呈现出来。

顶点着色器拿到的信息很多，有纹理坐标(texCoord)，法线(Normal)等等，把它们都转化为坐标后进行下一步操作。

# 渲染管线概论

渲染管线(The Rendering Pipeline)是现代计算机图形学中一个非常重要的概念。

渲染引擎本质上是以流水线形式组织起来的，于是就有了渲染管线这个称谓。

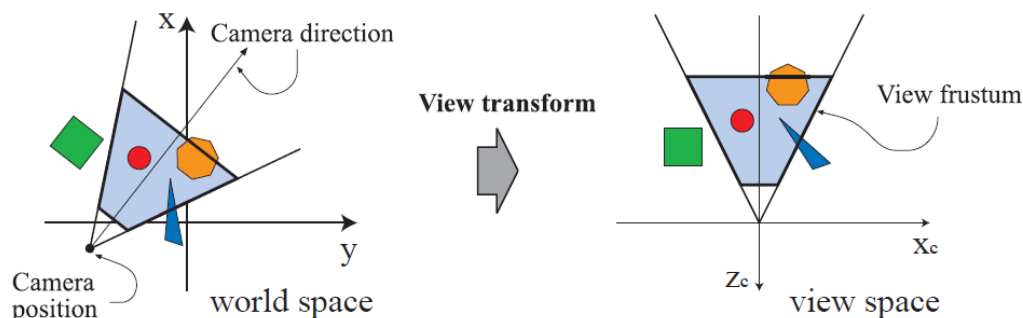


Technically, 我们具体把它分为 4 个阶段。分别是 **Application**, **Geometry**, **Rasterization** 以及 **Pixel** 阶段。其中 Application 阶段是在 CPU 上完成的，其余阶段大多使用 GPU.

# 渲染管线概论

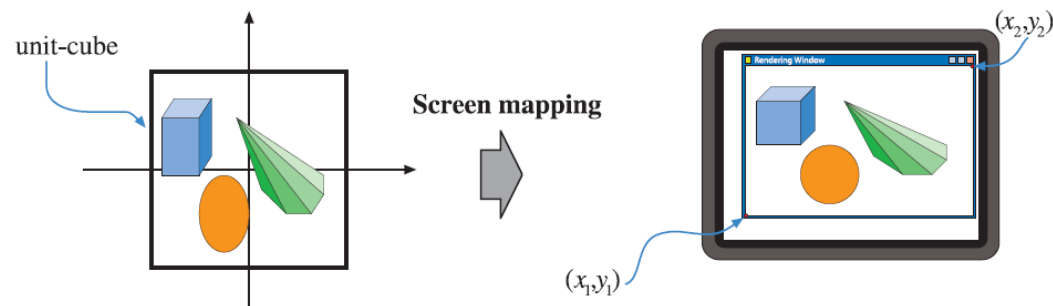
这些信息处理好了，接下来就是让他“变”到屏幕上面来。这里我们需要做一些变换(Transform)。

1. 把物体放到世界空间的坐标系 (Model Trans.)
2. 转化到摄像机视角为原点的坐标系 (View Trans.)
3. 转化到屏幕空间的坐标系 (Projection Trans.)

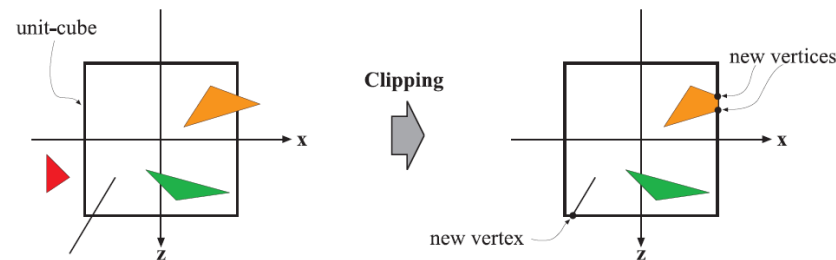


最后把视锥体变换为一个单位正方体进行裁剪，  
单位正方体以外的信息将被剔除。

这些变换都需要表示为  $4 \times 4$  的矩阵。

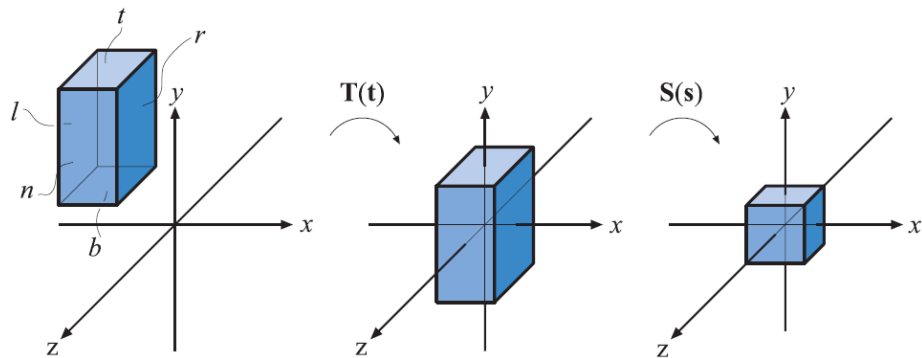


这就是图形学中著名的 "MVP 变换", 它蕴含着 3D 世界  
结构渲染的基本思想。



# 渲染管线概论

一个简单的图形变换：



$$M_{T(t_x, t_y, t_z)} = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$M_{S(s_x, s_y, s_z)} = \begin{pmatrix} s_x & 0 & 0 & 1 \\ 0 & s_y & 0 & 1 \\ 0 & 0 & s_z & 1 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

利用顶点着色器，我们可以将一个物件在视锥体内自由变换。例如我们这里先移动再缩小，只需将各顶点乘上一个平移矩阵  $M_{T(t_x, t_y, t_z)}$  再乘一个放缩矩阵  $M_{S(s_x, s_y, s_z)}$  即可。

经过简单的编程，您也能实现这样的效果。如果您想为起源重写 Renderer，那么这些是您必须掌握的技能。



# 渲染管线概论

欲了解更多顶点着色器编程信息，请访问 Valve 开发者社区 [developer.valvesoftware.com/w/index.php?title=Shader/zh&uselang=zh](https://developer.valvesoftware.com/w/index.php?title=Shader/zh&uselang=zh)。请坐和放宽，这个页面的内容也是我全部负责的。

顶点着色器 [\[编辑\]](#)

顶点着色器应用于 *可编程管线 (Programmable Pipeline)* 上运作的每个 *顶点 (Vertex)* 。其最基本的功能是将几何体的顶点通过矩阵变换 (Transform) (具体包括世界空间 (Model Transform) 变换, 视角变换 (View Transform) 和屏幕空间投影变换 (Projection Transform), 三者合一即为图形学上大名鼎鼎的 "MVP 变换" )转为屏幕空间坐标等等, 以便下一流程 (例如三角形装配, 遍历, 像素着色器等) 操作, 顶点着色器负责处理模型数据并对这些数据执行输出操作, 然后系统对顶点着色器输出的数据进行插值, 再把插值结果送到像素着色器。顶点着色器也从网格(Mesh)接收其他信息, 包括法线、切线和纹理 UV 坐标 等。

自 注意:

顶点着色器无法创建或销毁顶点。它处理的各顶点之间是无法相互访问的。

顶点着色器 示例 [\[编辑\]](#)

顶点着色器多数情况下只是扮演搬运工的角色, 它不会对顶点数据进行大改。

1. // 头文件

2. #include "common\_vs\_fx.c.h"

3. // 输出结构

4. struct VS\_OUTPUT

5. {

6. // 位置矢量 (float4)

7. float4 pos : POSITION0;

8. // 纹理坐标 (uv - float2)

9. float2 texCoord : TEXCOORD0;

10. };

11. // 获取位置矢量 (float4)

12. // 返回 VS\_OUTPUT

13. VS\_OUTPUT main( float4 inPos: POSITION )

14. {

15. // 声明需补充的空的 VS\_OUTPUT

16. VS\_OUTPUT o = (VS\_OUTPUT) 0;

17. // 计算 "输入位置" 的符号

18. inPos.xy = sign( inPos.xy );

19. // 用 已输入的 xy 值来确定输出的位置值

20. o.pos = float4( inPos.xy, 0.0f, 1.0f );

21. // 进入范围 [0,1]

22. o.texCoord = (float2(o.pos.x, -o.pos.y) + 1.0f)/2.0f;

23. return o;

24. }

▪ (当前 | 之前) ●

2023年1月4日 (三) 01:54

BlueMicro

(讨论 | 贡献)

.. (5,983字节)

(+358) .. (→*Post-process 后处理*) (撤销)

▪ (当前 | 之前) ●

2023年1月4日 (三) 01:32

BlueMicro

(讨论 | 贡献)

.. (5,625字节)

(+33) .. (→*着色器概述*) (撤销)

▪ (当前 | 之前) ●

2023年1月4日 (三) 01:29

BlueMicro

(讨论 | 贡献)

.. (5,592字节)

(+1,584) .. (→*着色器概述*) (撤销)

▪ (当前 | 之前) ●

2023年1月4日 (三) 00:51

BlueMicro

(讨论 | 贡献)

.. (4,008字节)

(-39) .. (→*着色器概述*) (撤销)

▪ (当前 | 之前) ●

2023年1月4日 (三) 00:50

BlueMicro

(讨论 | 贡献)

.. (4,047字节)

(+869) .. (→*着色器概述*) (撤销)

▪ (当前 | 之前) ●

2023年1月3日 (二) 23:56

BlueMicro

(讨论 | 贡献)

.. (3,178字节)

(+9) .. (→*着色器概述*) (撤销)

▪ (当前 | 之前) ●

2023年1月3日 (二) 23:16

BlueMicro

(讨论 | 贡献)

.. (3,169字节)

(-2) .. (→*着色器概述*) (撤销)

▪ (当前 | 之前) ●

2023年1月3日 (二) 23:07

BlueMicro

(讨论 | 贡献)

.. (3,171字节)

(+1,098) .. (→*着色器概述*) (撤销)

▪ (当前 | 之前) ●

2023年1月3日 (二) 22:49

BlueMicro

(讨论 | 贡献)

.. (2,073字节)

(-8) .. (→*着色器*) (撤销)

▪ (当前 | 之前) ●

2023年1月3日 (二) 22:44

BlueMicro

(讨论 | 贡献)

.. (2,081字节)

(+1,137) .. (→*着色器*) (撤销)

▪ (当前 | 之前) ●

2023年1月3日 (二) 20:54

BlueMicro

(讨论 | 贡献)

.. (944字节)

(+492) .. (→*着色器*) (撤销)

▪ (当前 | 之前) ●

2023年1月3日 (二) 20:47

BlueMicro

(讨论 | 贡献)

.. (452字节)

(+452) .. (*Created page with "==" 着色器 == 着色器是在GPU上运行的一套程序,*

着色器在起源引擎中的应用 [\[编辑\]](#)

Source 提供两种不同形式的着色器, 分别是 *后处理型* 和 *透对象型*, Source 中的大多数特效和材质都非常非常依赖其像素着色器组件。

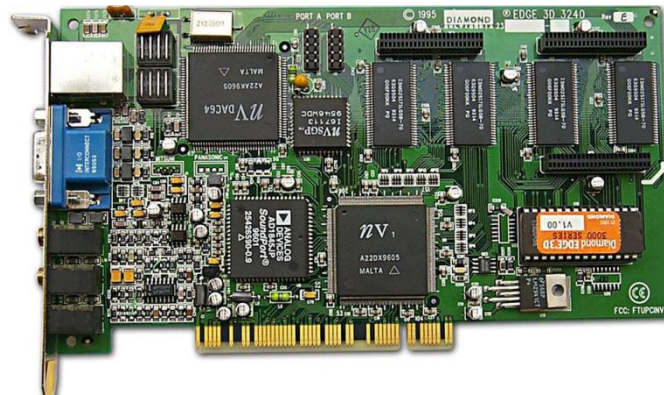
# 渲染管线概论

现在我们已经有了足够的信息去绘制画面了，但在那之前我们需要先准备好基本的框架。

这个框架就是众多的三角形，您如果玩过稍微早期的 3D 游戏就能看出来，世界是由无数三角形组成的。



当前图形领域只认三角形为基本单位。NVIDIA 曾经试图使用矩形来代替三角形，但是它失败得彻彻底底。(NV1 Chip)

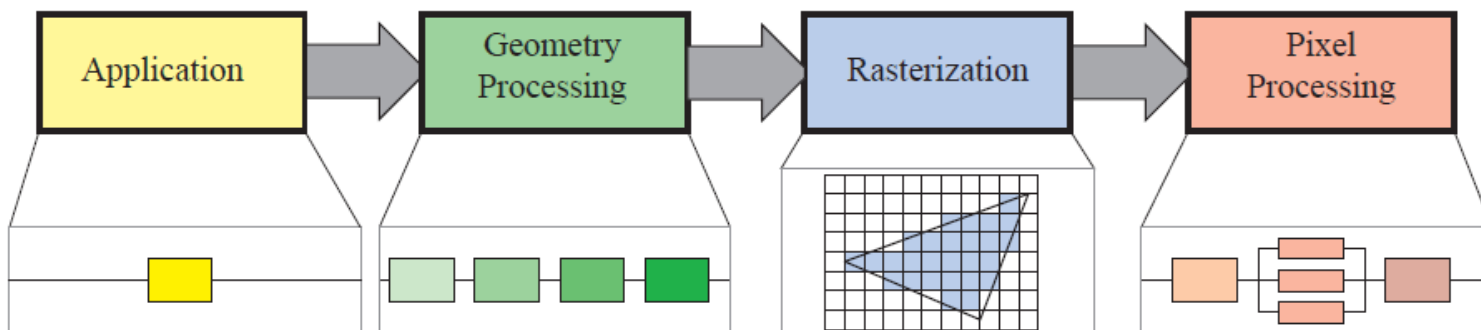




# 渲染管线概论

渲染管线(The Rendering Pipeline)是现代计算机图形学中一个非常重要的概念。

渲染引擎本质上是以流水线形式组织起来的，于是就有了渲染管线这个称谓。



Technically, 我们具体把它分为 4 个阶段。分别是 **Application**, **Geometry**, **Rasterization** 以及 **Pixel** 阶段。其中 Application 阶段是在 CPU 上完成的，其余阶段大多使用 GPU.

# 渲染管线概论

确定三角形覆盖在哪些像素上的操作就是大名鼎鼎的 “光栅化” (Rasterization)

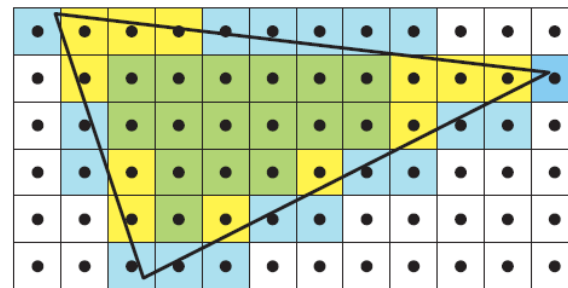
光栅化也分为两个阶段：1.三角形装配(Setup) 2.三角形遍历(Traversal)

由于这两个阶段几乎无可编程性可言，对起源来说价值不大。

这里仅作简单展开。

1.三角形装配：把顶点之间连起来，形成基本的三角形框架。

2.三角形遍历：寻找哪些像素被三角形覆盖，通过刚刚形成的连线来判定。

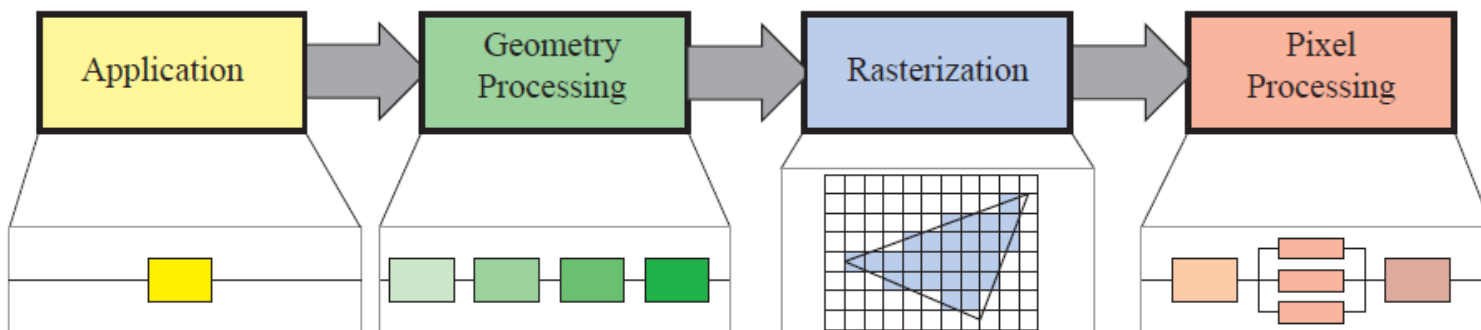


如果程序没有出错，走完这两步以后，就可以放心大胆地染色了！因为这时我们已经知道了哪些像素是准备好计算的了。

因为这个流程非常固定，现在的 GPU 有一种叫 "Raster Engine" 的专用电路来执行光栅化。

# 渲染管线概论

渲染管线(The Rendering Pipeline)是现代计算机图形学中一个非常重要的概念。渲染引擎本质上是以流水线形式组织起来的，于是就有了渲染管线这个称谓。



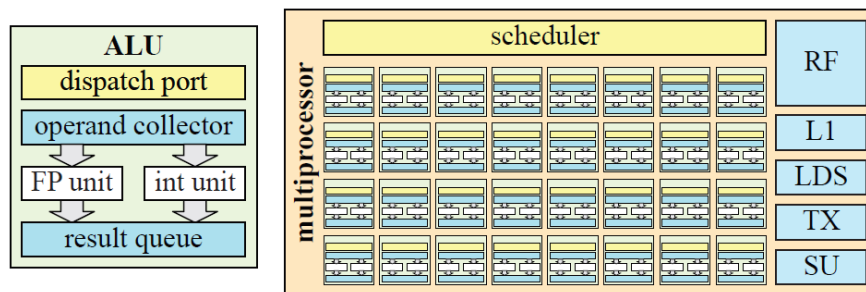
Technically, 我们具体把它分为 4 个阶段。分别是 **Application**, **Geometry**, **Rasterization** 以及 **Pixel** 阶段。其中 Application 阶段是在 CPU 上完成的，其余阶段大多使用 GPU。

Pixel

# 渲染管线概论

接下来就是 **Pixel** 阶段了。顾名思义，这个阶段由像素着色器(Pixel Shader)来完成的。

这个阶段是最复杂的，它执行大量的着色方程求解。例如各种光照模型，PBR 渲染等等。所以在现代 GPU 上，它一般由通用的 SIMD 计算单元(ALU)承接。



不过我希望这里成为渲染管线和 GPU 的对接，因为关于起源着色和光照的东西在 02 期里讲了很多了。但是在那之前，还有一件事。



## Source 2 的渲染管线

其实刚才的内容大部分是前置知识。因为游戏的渲染引擎走的管线都是通用的思想，起源也不例外。算是在讲 generally 的计算机图形学了。

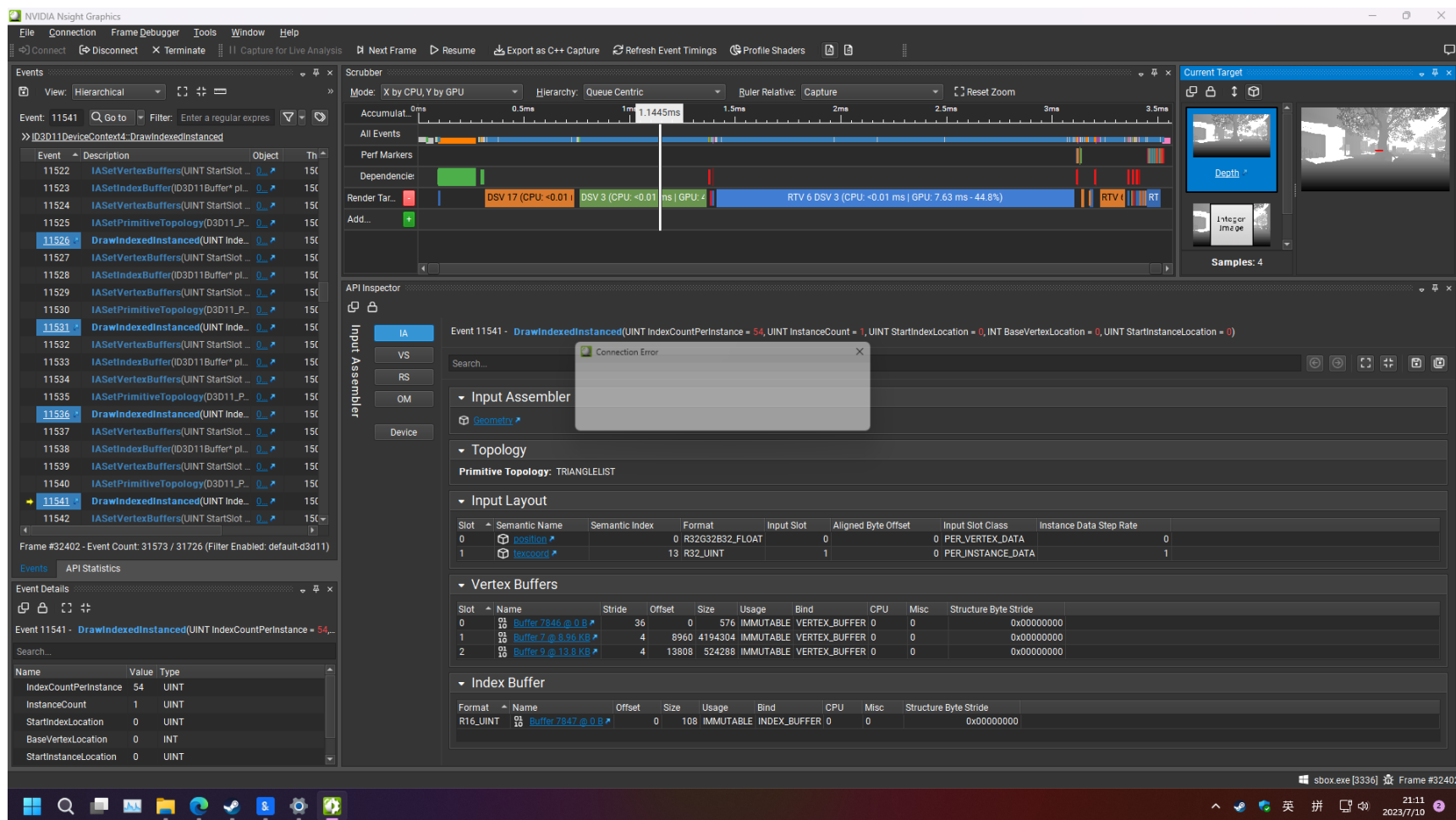
有了刚刚的知识储备，我们现在才能一窥起源 2 具体的渲染管线。

如果您对图形学略有了解，您可能已经发现我少描述了一个重要阶段 —— ROP 或称 Merging Stage. 对显卡有了解的朋友可能也知道有这样一个参数 “ROPs 光栅单元”（例如 RTX 3070 Ti 有 96 个 ROP），其实指的就是这个 “ROP”。

我其实是要放在起源 2 这个大背景下进行描述。因为 ROP 阶段有很多工作（例如可见性计算）是随着现代图形 API 的发展而转移到了通用计算单元上。起源 1、2 之间可能略有不同，这里我们说起源 2 的，它的深度计算转移到了 ALU 上。但是 Alpha Blend 以及 MSAA 反走样仍然在 ROP 上完成。

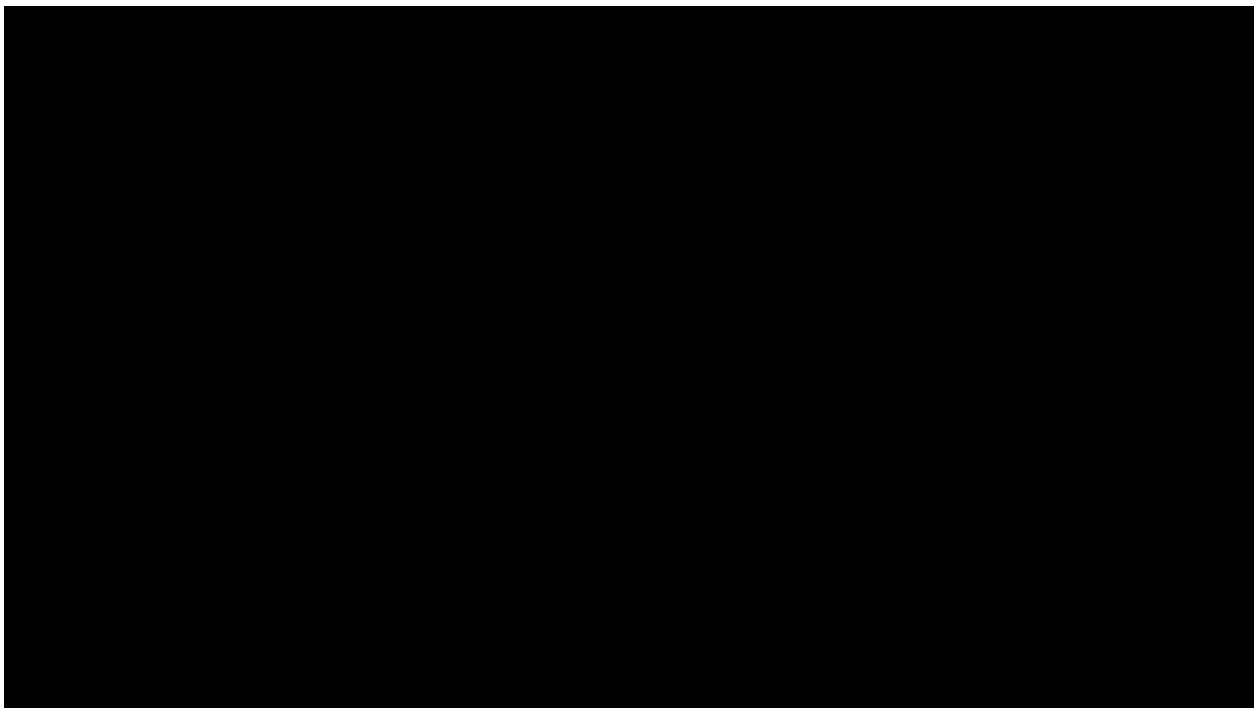
# Source 2 的渲染管线

这里我们使用了一个很强大的工具，叫 NVIDIA Nsight Graphics，它可以截获渲染管线内的很多信息。



## Source 2 的渲染管线

我有一期视频录制了作为起源 2 引擎的 Counter-Strike 2 的渲染。您可以看出，它是一种基于前向渲染的绘制方式。



因为起源 2 并未绘制延迟渲染特有的 G-Buffer，而是有一个单独的深度缓冲区，并且您还能发现，物体是一个一个地渲染出来的，这充分证明了起源 2 渲染引擎是基于前向(Forward)的渲染管线。

## Source 2 的渲染管线

什么是前向渲染呢？三角形一步一步地往下走就是前向渲染了。(Real-Time Rendering 的原话是 Send Down the triangles, 您应该能意会它在说什么)，之前的所有内容实际上都是围绕着前向管线展开的。

不过作为对比，我们在此解释延迟渲染(Deferred Rendering)管线，因为起源 2 还提供了可选的延迟渲染支持。

延迟渲染管线其实也很简单。我们这里引入一个 G-Buffer (Geometric Buffer, 几何缓冲区) 的概念。G-Buffer 是众多 Buffer 的集合, Albedo(反照率)、Depth(深度信息)、Specular(高光信息)、Normal(法线信息) 甚至 Motion Vector(运动矢量)等都存于此处。

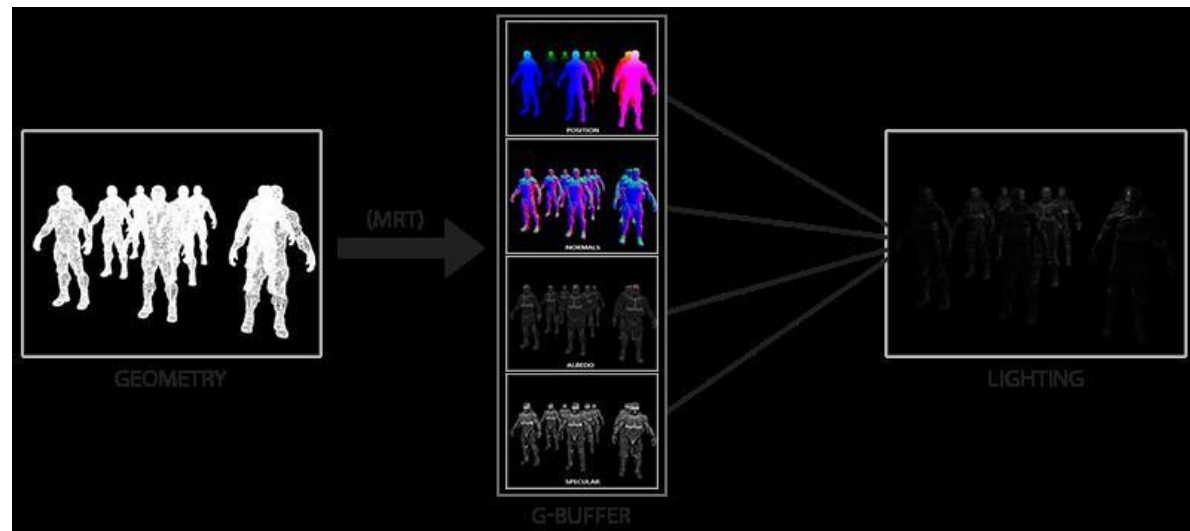
(P.S. 但起源 2 几乎没有任何的 Temporally-rely on 的技术, 实用性起见我们在以后都很少展开运动矢量)

不过这么多信息, 我们应该怎么实现 G-Buffer 呢?

# Source 2 的渲染管线

Multi-Render Target, MRT(多渲染目标)

	R8	G8	B8	A8
RT0	world normal (RGB10)			GI
RT1	base color (sRGB8)			config (A8)
RT2	metalness (R8)	glossiness (G8)	cavity (B8)	aliased value (A8)
RT3	velocity.xy (RGB8)			velocity.z (A8)



- 和前向渲染管线不同的是，延迟渲染的 input 变量直接从 G-Buffer 而不是从顶点着色器中取得的。
- “延迟”一词非常形象地概括了它的光照计算特点，因为延迟管线的光照计算阶段拿到的信息已经是经深度测试处理的最终结果，每个像素只需计算 1 次光照。时间复杂度是  $O(m+n)$ 。相比前向管线的  $O(m*n)$  来说，是一个非常大的开销节省。




# Source 2 的渲染管线

但是起源 2 目前存在 Bug，无法走多 Pass Rendering，这使得延迟渲染很难实现。我曾经试图实现一个延迟管道，但是失败了，因为它仅仅只支持走一个 Pass，然而延迟渲染管线有一个 Geometry Pass 和一个 Shading Pass。

## Shader Pass support #1067

Open ogniK5377 opened this issue on Oct 9, 2021 · 0 comments



ogniK5377 commented on Oct 9, 2021

Contributor

What can't you do?

Currently, there's no easy way to handle shader passes. The way to deal with multiple shader passes is to manually render the object twice with a RenderEntity and calling drawing the same object multiple time using different materials.

How would you like it to work?

```
ShaderPassCount(2);

#if (S_SHADER_PASS == 0)
// Do shader pass 1
#elif (S_SHADER_PASS == 1)
// Do shader pass 2
#endif
```

What have you tried?

Currently, the only other way to deal with this is to force an object to render right after the other manually. One method of this is to use the DoRender override for RenderEntities.

```
public override void DoRender( SceneObject obj )
{
    _vertexBuffer.Draw( _materialPass1 );
    _vertexBuffer.Draw( _materialPass2 );
}
```

Additional context

Define an extra pass should just render the object multiple times in successive order and rendering on the correct pass

10

Assignees

No one assigned

Labels

feature request

Projects

None yet

Milestone

No milestone

Development

No branches or pull requests

Notifications

Subscribe

You're not receiving notifications from this thread.

1 participant

Two-pass deferred shading algorithm

Pass 1: geometry pass

- Write visible geometry information to G-buffer

Pass 2: shading pass

For each G-buffer sample, compute shading

- Read G-buffer data for current sample
- Accumulate contribution of all lights
- Output final surface color

## Source 2 的渲染管线

所以我在这里下个暴论：

起源 2 的延迟渲染是未来的技术，并且永远都是！

# Source 2 的渲染管线

但是我们转念一想，对于当前的起源来说，延迟渲染似乎是一个糟透了的选择。

- 1.延迟渲染的最大优点是光源计算。
  - 起源游戏很少有一大堆光源的场景。
- 2.延迟渲染很难做 MSAA，一般使用 TAA 或 FXAA 抗锯齿。
  - 相比起源一直以来的优良传统(MSAA)，效果差劲。
- 3.延迟渲染的 G-Buffer 占用大量内存带宽，一些 GPU 可能会造成严重的冯诺依曼瓶颈。
  - 不符合 Valve 一直以来降低开销的游戏开发基本思想。
- 4.延迟渲染支持的 Shader 数量少，很难同时实现多样的渲染效果。
  - 不符合起源社区的风格。

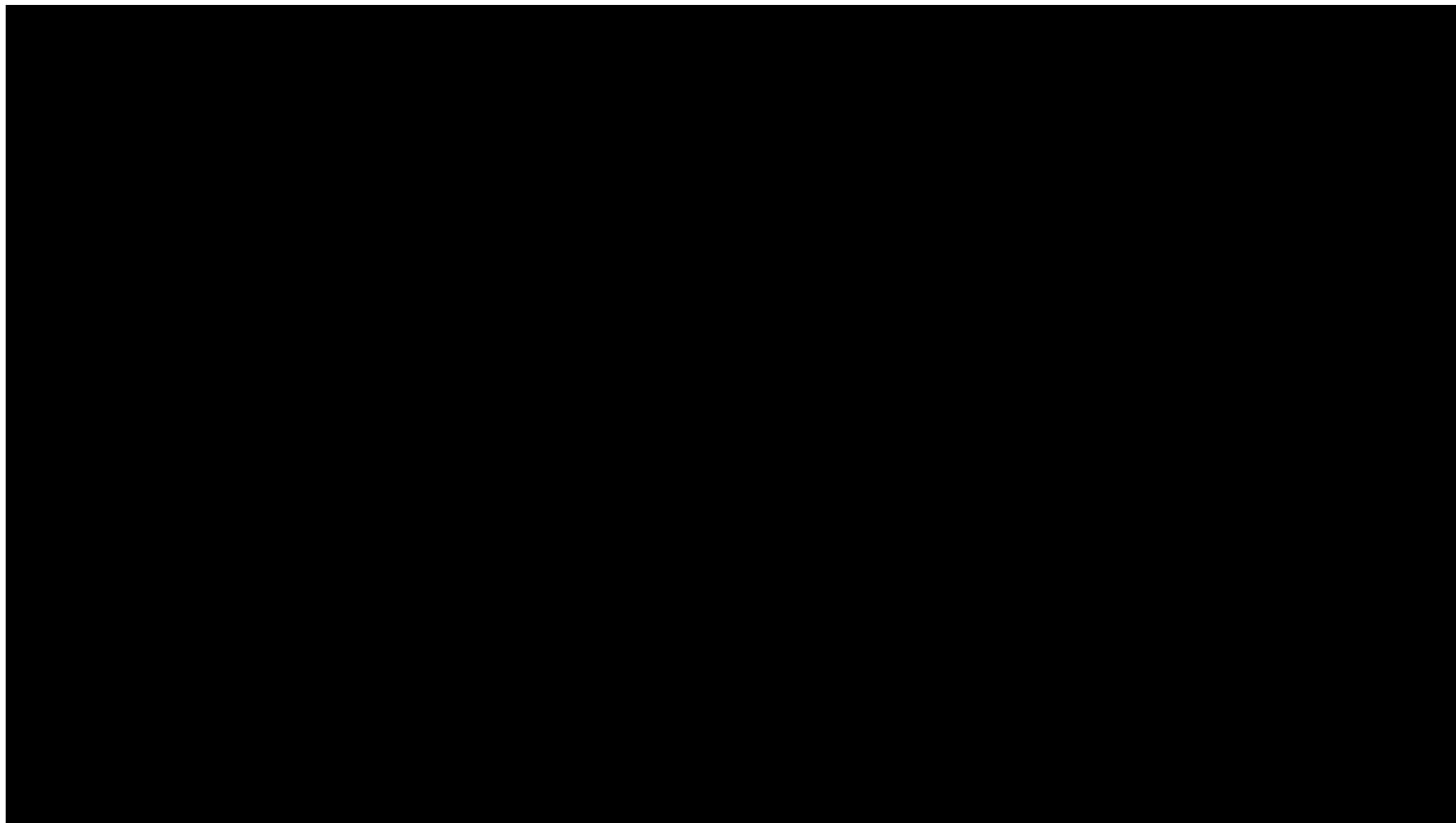
	Trad. Deferred	Tiled Deferred	Tiled Forward	Clustered Deferred	Clustered Forward
Bandwidth Use	High	Low	Low	Low	Low
Transparency	No	Maybe	Yes	Maybe	Yes
MSAA	Maybe	Maybe	Yes	Maybe	Yes
Shadow Map reuse	Yes	No	No	No	No
Geometry Passes	1	1	1-2*	1	1-2*
Register Pressure	Low	High	High	High	High
Innermost loop	Pixels	Lights	Lights	Lights	Lights
FB Precision	High	Low	Low	Low	Low
View dependence	Low	High	High	Low	Low
Vary Shading Models	Hard	Hard	Simple	Hard	Simple

So, forget about the ~~Freeman~~ Deferred Shading !

除非您想打算使用起源 2 开发一个开放世界的游戏，否则我不建议您上延迟渲染。

## Source 2 的渲染管线

好，我们回归到起源 2 默认的前向管线上来。我们首先来欣赏一段视频。



## Source 2 的渲染管线

这么多光源我们是如何实现帧数稳定的呢？

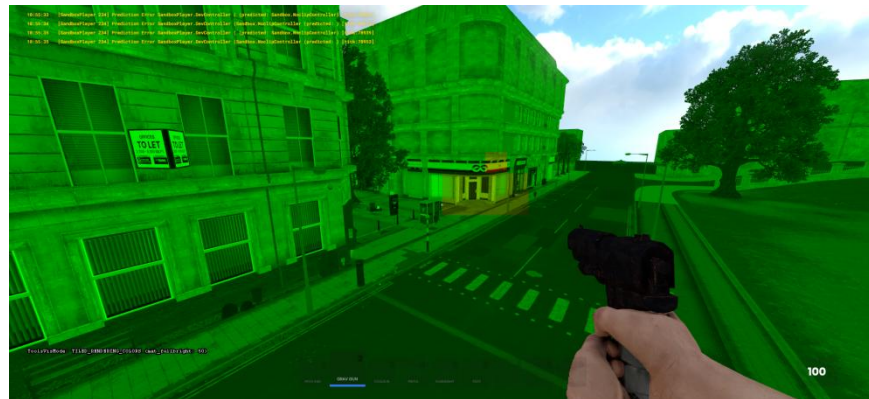
答案是 Tile-Based Rendering，起源 2 使用了一种较为不同的前向管线。

您也可以在 Garry's Mod 里进行一个测试，如果您的 GPU 性能不高，您会发现您很快就卡顿闪退了。

"Tiles" 是一个个小方块，起源 2 的渲染管线把缓冲区分成很多个 Tile，不同的游戏引擎，这些 Tile 可能也有不同的分辨率。

起源 2 的是单个 48\*48 的 Tile.

我们把这种分 Tile 的前向渲染管线叫做 "Forward+" 管线。





## Source 2 的渲染管线

这个方法可以优化原本前向管线并不擅长的多光源处理  
仅需设计一个简单程序即可，伪代码如下：

```
foreach object in scene
    get depth
foreach tile in screen:
    get max min depth
    Frustum Intersection test
    Generate a list of light
foreach pixel in screen:
    foreach light in light_list_of_this_tile:
        pixellighting += light_contribution_to_pixel(light,pixel)
```

这段代码所想表达的，是通过深度信息确定各 Tile 内包含的几何体是否受光照影响。

但是 Tile list 和 Light list 造成了一些 I/O 访问多，可能对一些低带宽的 GPU 还不是那么友好。但是我的评价是比延迟渲染好。

## Source 2 的渲染管线

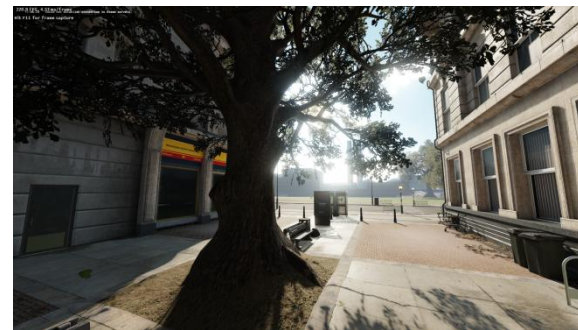
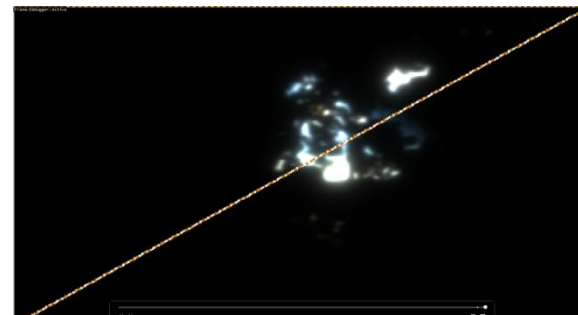
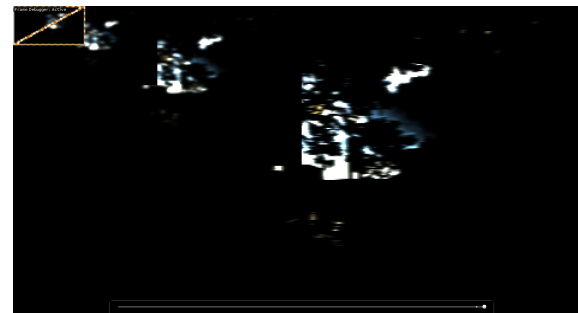
我们这里也简单说说起源 2 的后处理 (Post-Processing)  
我们这里只是提一下 Bloom 泛光特效。

起源 2 的 Bloom 效果令人印象深刻，它是一种后处理特效。

我们选择降采样渲染一个 1:16 的 Buffer，然后对它进行  
高斯滤波 (Gaussian Filtering)，再进行线性插值来柔化  
模糊图，得到最终图像再叠加到原像上。

在 1920\*1080 分辨率下，这个过程仅需 20  $\mu$ s. (0.02 ms)

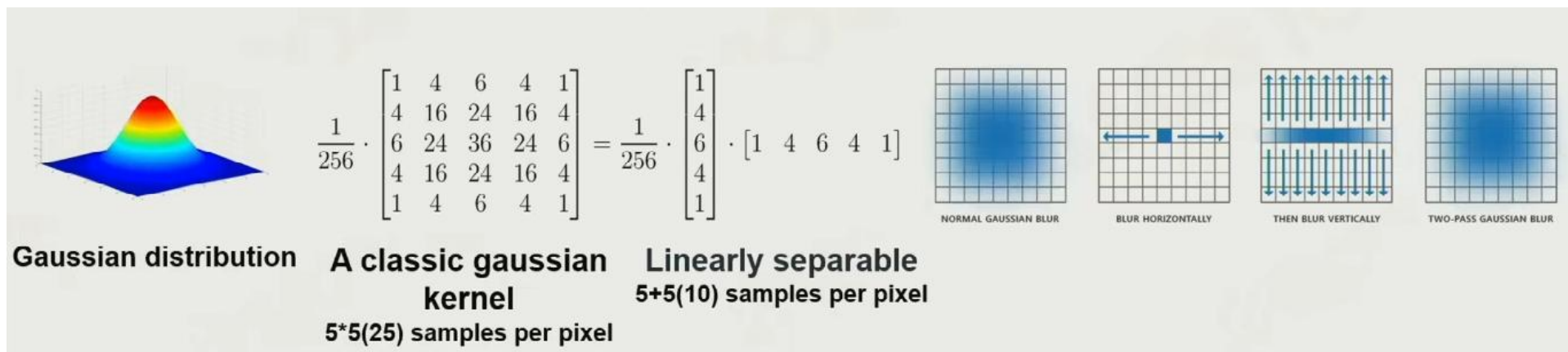
这样的优化是怎么做到的呢？



## Source 2 的渲染管线

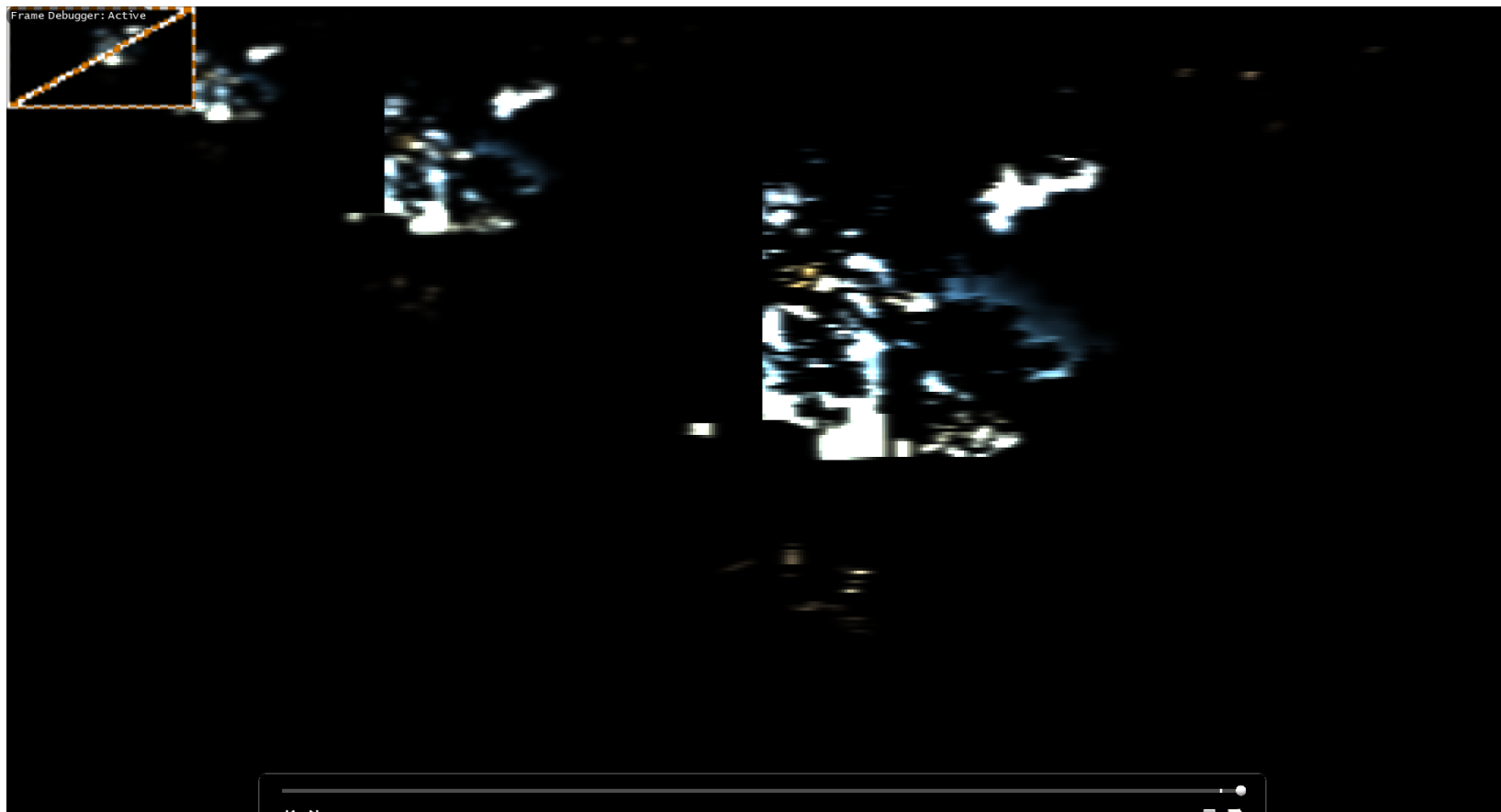
其实你已经看出来了一些端倪，我们降采样进行了渲染。  
最终呈现的是一张分辨率比例为 1:16 的泛光图。

其实，仅仅通过渲染一张很小的图，它的泛光范围就和 Garry 的 dick 一样小。这里我们用到的是一种叫"Pymarid Gaussian Blur"的手段来进行优化。这是一个很常见的方法。



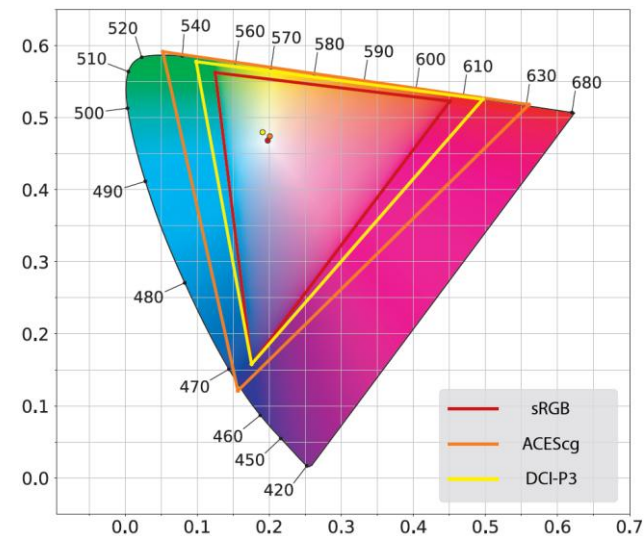
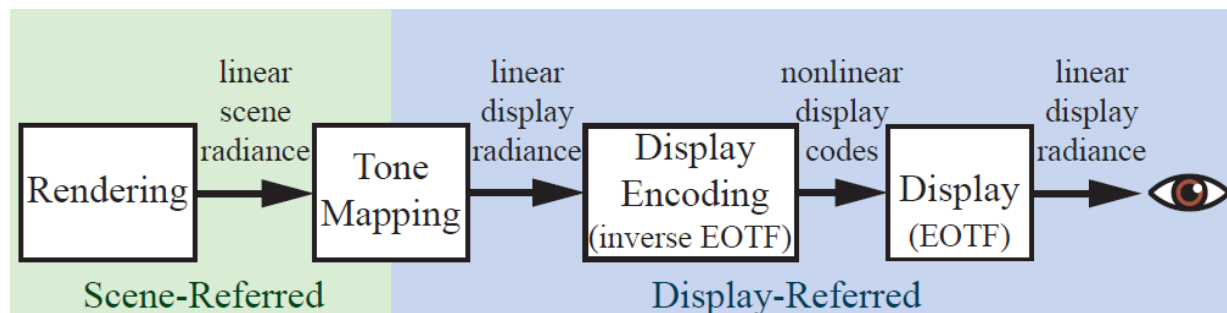
## Source 2 的渲染管线

通过对各层级加入权重，我们就可以得到理想的泛光图了。



## Source 2 的渲染管线

后处理进行得差不多的时候，是时候给玩家展示渲染成果了。但在此之前.....





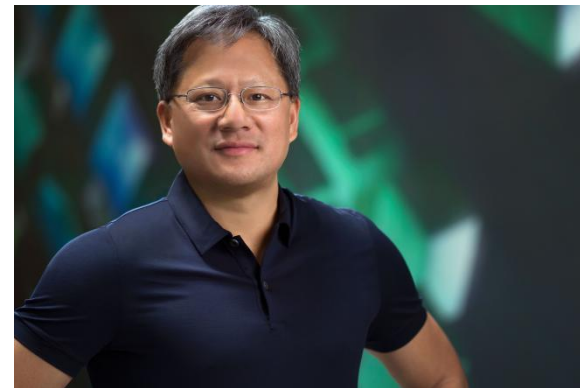
## Source 2 的渲染管线

至此，我们起源 2 的渲染管线就告一段落了。这个视频只是基础。

*前面的区域，以后再来探索吧*

# 在 GPU 上的实现

*“The Display is the Computer.”*  
—— NVIDIA Founder & CEO, Jensen Huang



我们需要 GPU，GPU 撑起了现在人类科技生活的一切。建筑，医疗，自动化，机器学习等等领域都离不开 GPU。学习 GPU 原理您会成为合格的开发者或者卡吧大佬。

注意，接下来的知识是整个领域通用的，于是几乎不会涉及起源 2。但是对深入了解起源 2 的工作原理很有用。

我们返璞归真，GPU 一开始是为游戏玩家服务的，所以我们这里只展开游戏 GPU，不展开任何关于计算卡的东西(例如 NVIDIA H100 或 AMD Instinct)。

就算您是普通玩家，肯定或多或少对 GPU 略有概念了，否则您应该不会来看我的视频。所以我们直接跳过概念部分。

## 在 GPU 上的实现

在之前的渲染管线部分，我们稍微提到了一些 GPU 功能的实现。那么，它们是怎么一回事呢？

让我们先登上时光机，穿越回 1999 年。

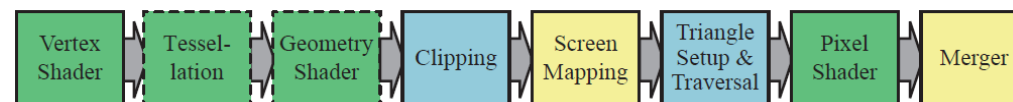
1999 年 8 月

NVIDIA 面向消费市场推出第一款 GPU ——  
NVIDIA GeForce 256 SDR

它搭载的 *Independent Pipelined QuadEngine™* 技术从 CPU 那里分离了变换(Transform)、照明(Lighting)。



这就是硬件加速的变换和光照，简称“硬件T&L”，它最大的意义在于将渲染管线中开销逐渐膨胀的两大负载转移到了显卡上（原本是 CPU 做的工作）。



# 在 GPU 上的实现

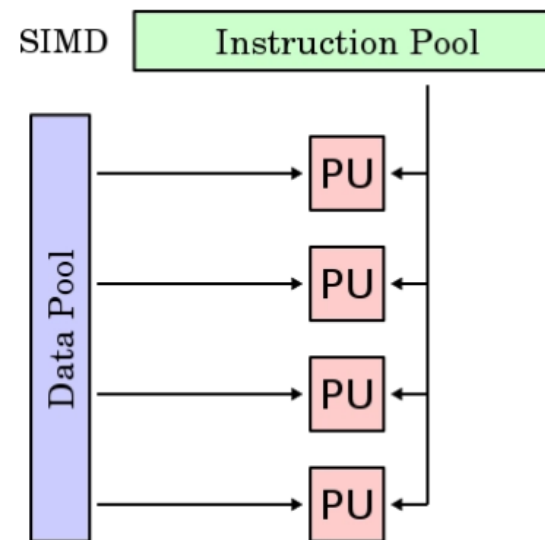
现代 GPU 硬件上是以一种叫 “单指令多数数据流” (Single Instruction Multiple Threads, SIMD) 的形式组织起来的。

为了便于理解，我们这里举一个简单的例子。

在软件编程模型上，GPU 通常是以一种叫 “单指令多线程” (SIMT) 的结构组织起来的。计算单元数量的庞大实现着指令级并行。

不同的 GPU 线程调度/执行着不同的 shader 代码，我们将其称为 "shader invocation"，它也是着色器处理一个基本单位的操作。

这样的基本单位叫图元(Primitive)，可以是点，线，三角形。



# 在 GPU 上的实现

GPU 执行的指令也是各不相同的。主要的有

- 运算指令：FMA、FMAD、FMUL、MAD、MUL 等
- 访存指令：LOAD、STORE 等

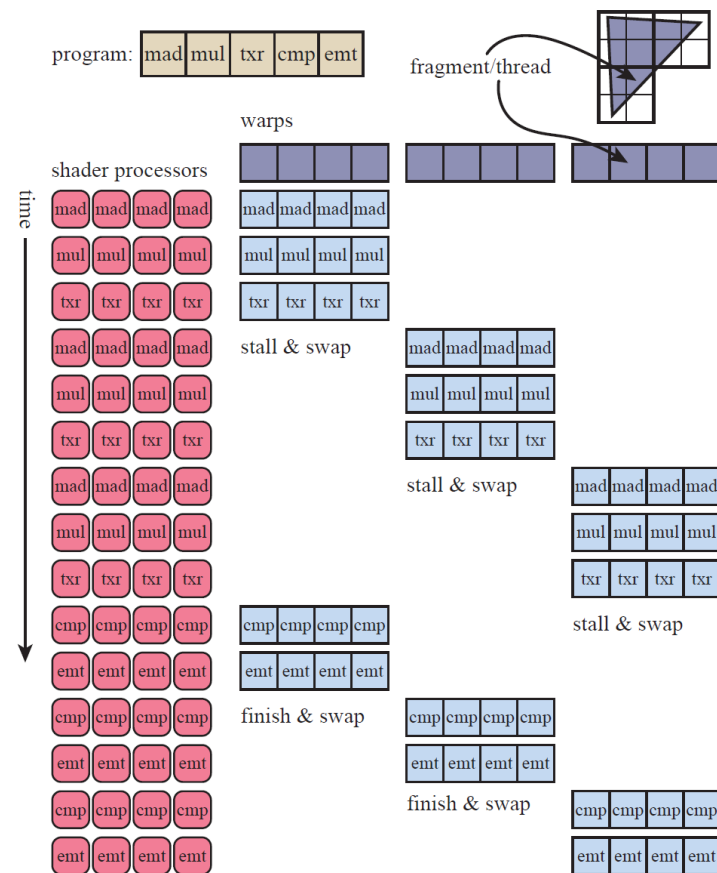
当然这些是较为底层的操作指令了，在 shader 编程里您能看到像 ...texCoord 这样的 sampler 指令等等。

现在我们广泛使用 32 位浮点数来存各种各样的信息。

例如：

矢量信息（位置信息 xyzw、颜色信息 rgba、法线 等）

标量信息（各种指数、位掩码 BitMask 等）



# 在 GPU 上的实现

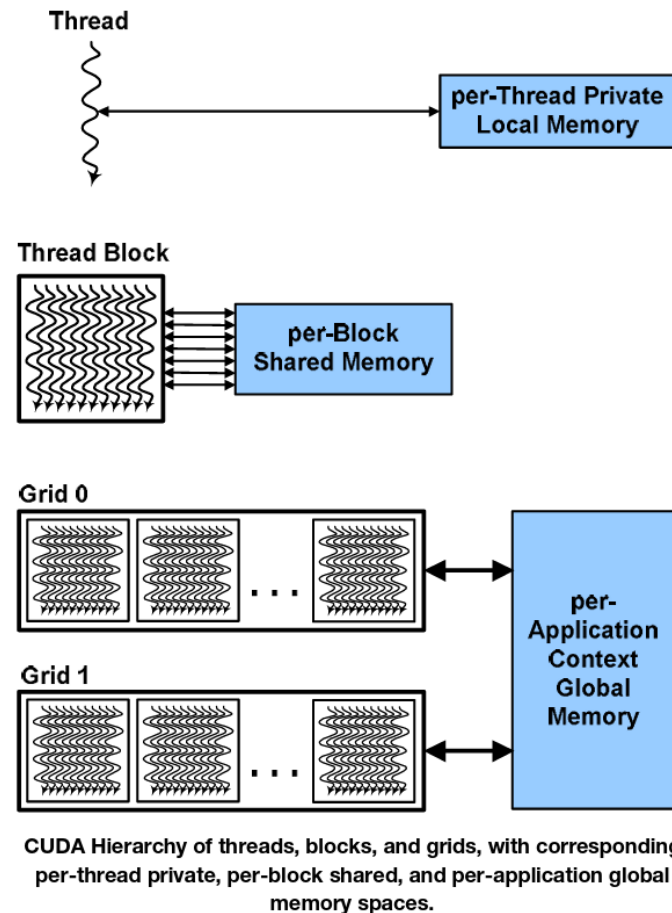
而随着计算量和要求的增加，GPU 现在发展出了层级的计算结构。

- Thread 线程，最小单位。
- Warp (NVIDIA) / Wavefront (AMD) 线程束，一般为 32 线程。
- Grids - Kernel(CUDA)，一个应用程序执行所有线程的集合。

它们都有相应的计算单元结构和内存来进行数据交换。

渲染管线中使用到 SIMD 单元(或称通用单元)的着色器，都服从着这样的体系。

同时，如果一个 Warp 内的其中一个线程走了不一样的代码路径，就会产生分歧(divergence)，这会大大降低 SIMD 的效率。所以我们需要一些优化，例如光线追踪着色器的 coherent sorting.





# 在 GPU 上的实现

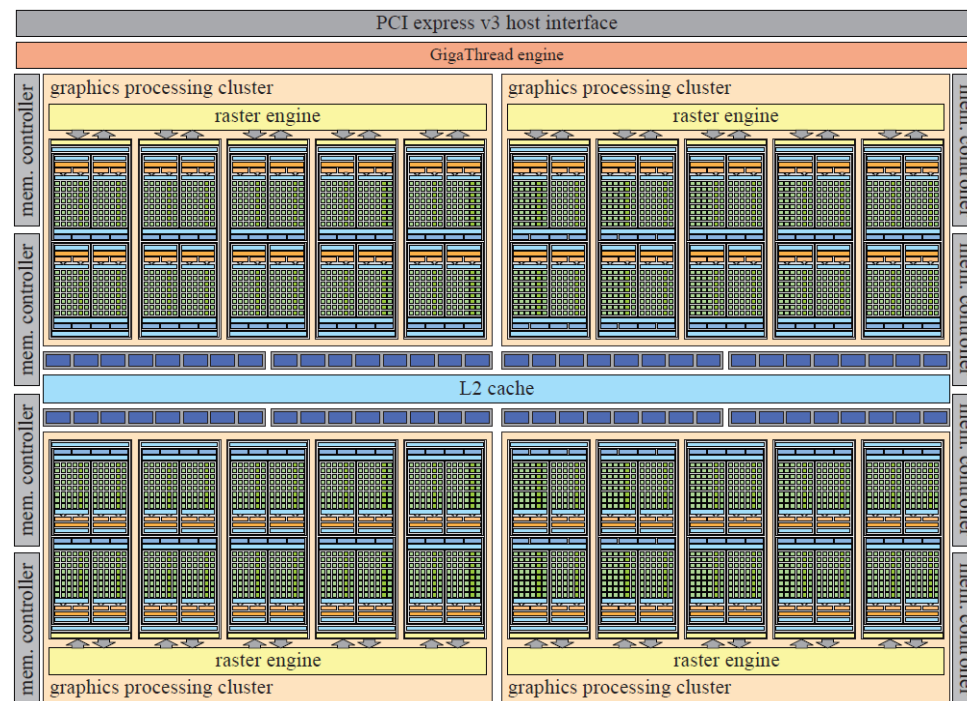
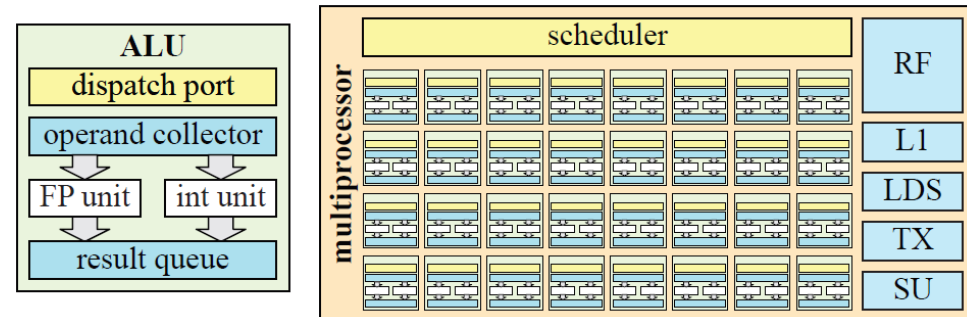
我们现在来看看现代 GPU 的基本组成结构。

主要分为：

- 流处理器 / Shader Core / ALU
- 流式多处理器 SM (NVIDIA) / 计算单元 CU (AMD)
- 多处理器集群
- 整个 GPU

具有计算功能的电路一般都有自己的片上内存。

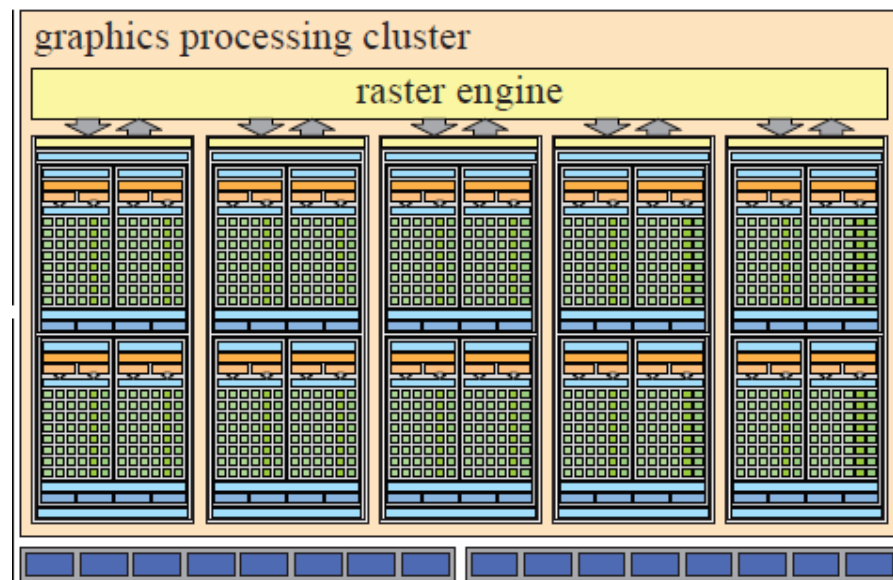
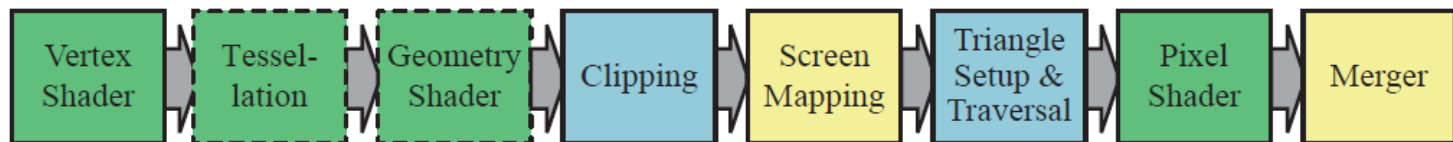
注意：流处理器的概念是自从 2006 年 NVIDIA G80 发布后才有的，这是第一款统一着色器架构的 GPU，各个 Shader Core 都可以动态被调配为 Pixel 或者 Vertex Shader。



# 在 GPU 上的实现

除了计算单元，还有专门用作图形处理的专用电路。

- 几何多边形引擎
- 光栅化引擎
- 纹理采样器 (TMU)
- 渲染后端引擎 (ROP)
- 视频编解码引擎（不展开）
- 图像引擎（不展开）
- 硬件光线追踪单元（略微展开）





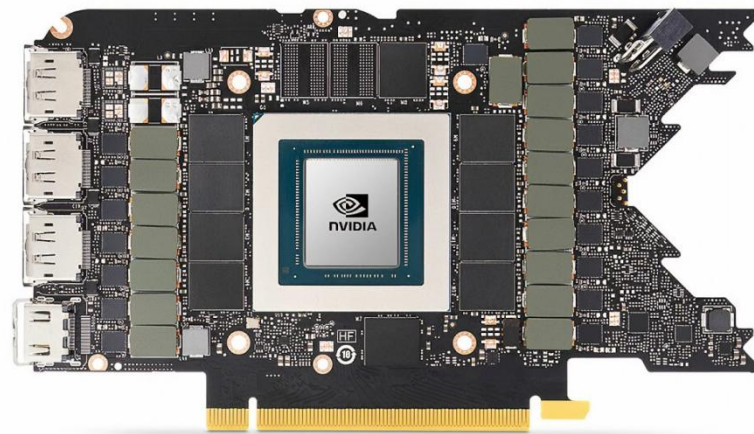
# 在 GPU 上的实现

基本的 GPU 逻辑看得差不多了，是时候来个案例分析了。  
我们以 RTX 3080 为例，它的架构和功能足以代表现代 GPU。

我们来看看这个 GPU 的硬件数据：

- Architecture: NVIDIA Ampere
- SM: 84
- Shader: 8704 Unified
- ROPs/TMUs: 96/272
- RT Core / Tensor Core: 84/168

我们怎么通过渲染管线的知识来了解它们呢？

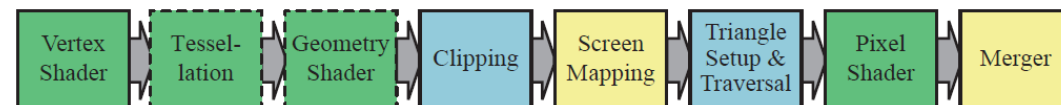


# 在 GPU 上的实现

- Architecture: NVIDIA Ampere
- SM: 84
- Shader: 8704 Unified
- ROPs/TMUs: 96/272
- RT Core / Tensor Core: 84/168

请看，这是 3080 的架构示意图。我们使用刚刚的知识解释它：

- 这个 GPU 含有 8704 个统一着色器(ALU)。
  - 它有 96 个 ROP 可供 Merger 阶段的后端操作。
- 例如：Alpha Blend、MSAA、Shadow Mapping
- 它有 272 个纹理采样器，配合着色方程进行计算。



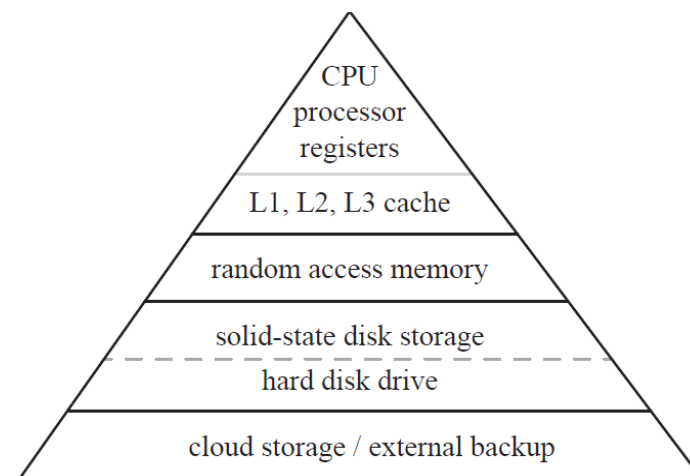
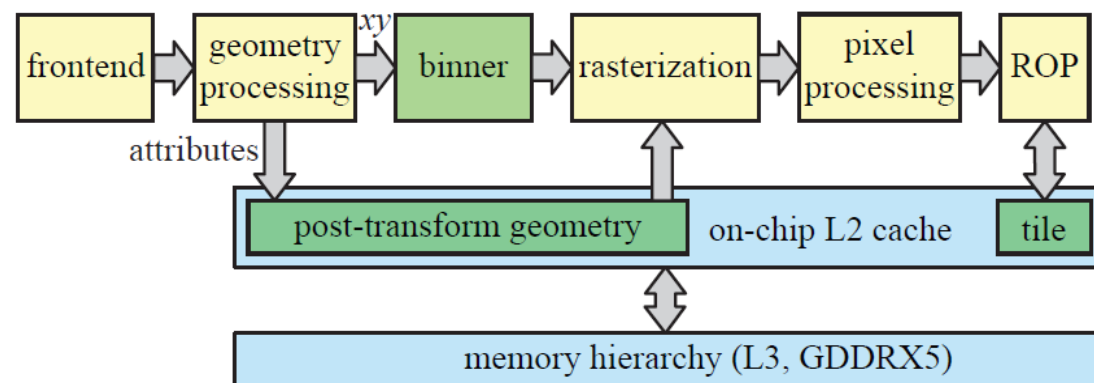
# 在 GPU 上的实现

同时，自从 Maxwell 2.0 架构开始，NVIDIA GPU 支持一种叫"Tile Caching"的技术，实现了一种 Tile-Based Rendering. 节省了很多带宽，这是好的。

AMD 也是不甘示弱的。现在它们都有这样的技术。

您看，在这里我们加入了内存子系统和渲染管线之间的联系，非常直观。所以这里再次强调访存优化的重要性。

说到这里，内存子系统一般如右图 2。



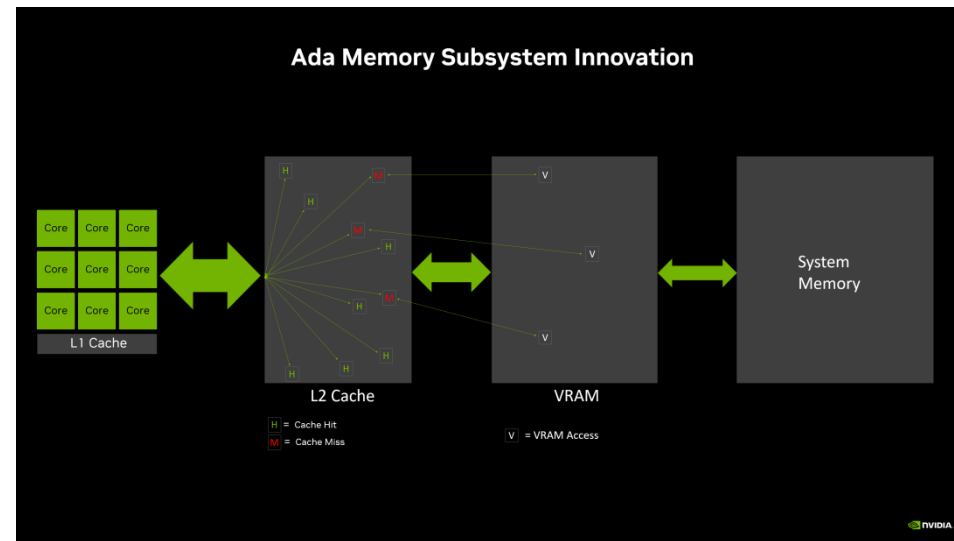
# 在 GPU 上的实现

对于 GPU 来说，则是：

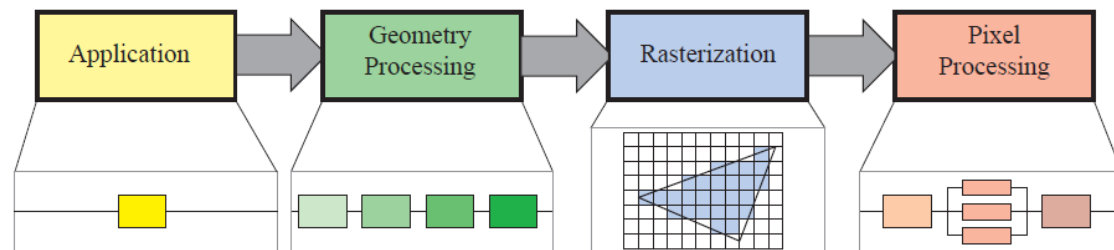
- 寄存器(Register)
- L0 指令缓存
- L1 数据缓存
- L2 全局缓存
- L3 全局缓存(AMD Infinity Cache)
- 显存 (DRAM)
- 系统内存 (DRAM)

这里有数据局部性(locality)和命中率(hitrate)的概念，如果您学习过计算机组成原理，肯定能明白它们的意思。

提高缓存命中率的最直接方法就是缓存扩容，例如 NVIDIA Ada Lovelace 架构的 GPU。这里不过多展开。



# 在 GPU 上的实现



# 结语

不觉得很酷吗？作为一名起源引擎开发者我觉得真是太酷了。

- 渲染管线 - 概念与原理，起源 2 的渲染管线
- 图形处理器 - 概念，基本原理，逻辑管线，案例分析(RTX 3080)

这两大主题这是现代计算机图形学的基础，根据包含被包含关系，这也是整个起源引擎系列的基础。

感谢您能收听到这里！如果喜欢可以三连！