

硕 士 研 究 生 读 书 报 告



题目 着色器组件：高性能模组化的着色器

作者姓名 张正昱阳

作者学号 21851458

指导教师 李启雷

学科专业 软件工程

所在学院 软件学院

提交日期 二○一八年十二月

Shader Components: Modular and High Performance Shader

A Dissertation Submitted to

Zhejiang University

in partial fulfillment of the requirements for

the degree of

Master of Engineering

Major Subject: Software Engineering

Advisor: Li QiLei

By

Zhang ZhengYuYang

Zhejiang University, P.R. China

2018

## 摘要

现代游戏引擎希望在两个有冲突的目标：高效渲染和高效软件开发之间找到平衡。为了改善CPU的性能，最新一代的实时图形API提供了一些新的原型，可以向着色器参数进行高效的批量更新。然而，现代游戏引擎所拥有的大型着色器代码库难以利用这一优势。问题在于，尽管着色器参数可以被组织成模组，并高效地以不同频率绑定到流水线上，但现代着色语言缺少对应的将着色逻辑组织为模组的原型。其结果就是复杂的着色器通常被编译成使用一个整块参数块的样子，摈弃了新的参数绑定API的设计理念以及性能优势。本文提议通过引入着色器组件（shader components），一种着色器程序里的一类模组化单元来解决这种不匹配。着色器组件封装了着色逻辑和该逻辑使用时必需的参数。我们将通过使用组件搭建成熟的着色器，来展示我们可以在保持性能的一些关键方面（使用中的着色器逻辑的静态具体化以及在组件粒度下的高效参数更新）的同时，保持模组化着色器代码结构体，正如今日的高端游戏引擎所渴求的那样。

## Abstract

Modern game engines seek to balance the conflicting goals of high rendering performance and productive software development. To improve CPU performance, the most recent generation of real-time graphics APIs provide new primitives for performing efficient batch updates to shader parameters. However, modern game engines featuring large shader codebases have struggled to take advantage of these benefits. The problem is that even though shader parameters can be organized into efficient modules bound to the pipeline at various frequencies, modern shading languages lack corresponding primitives to organize shader logic (requiring these parameters) into modules as well. The result is that complex shaders are typically compiled to use a monolithic block of parameters, defeating the design, and performance benefits, of the new parameter binding API. In this paper we propose to resolve this mismatch by introducing shader components, a first-class unit of modularity in a shader program that encapsulates a unit of shader logic and the parameters that must be bound when that logic is in use. We show that by building sophisticated shaders out of components, we can retain essential aspects of performance (static specialization of the shader logic in use and efficient update of parameters at component granularity) while maintaining the modular shader code structure that is desirable in today’s high-end game engines.

## 引言

现代游戏引擎，在渲染包含复杂材质和光照的高细节程度场景时，必须有着很好的性能表现。与此同时，描述这些材质和光照的着色器的大量代码也需要被CPU高效地生产出来。现今的CPU（甚至是多核CPU）并不能以足够快的速率向GPU提供渲染指令，从而造成了实时渲染系统的性能瓶颈。这些渲染指令当中的很大一部分都和绑定图形流水线的着色器参数相关（绑定材质，绑定缓存，等等）。这个问题一度非常紧急关键，以至于Khronos Group 和Microsoft特意设计了新一代实时图形API，提供一些新的原语以提高向着色器批量更新参数的效率。但是现代游戏引擎并没有很好地利用这些新功能。

它们的问题在于：尽管着色器参数可以被组织成高效的模组形式，而且可以以合适的频率绑定到流水线上，但现代着色器编程语言缺乏与之对应的原语去把对应的着色逻辑组也组织成同样的模组。于是，复杂的着色程序被编译为一个大整块，并且以大整块的形式获取参数。这些参数通常情况下由引擎负责整体更新，从而违背了那些新API的设计原则，放弃了新API的性能提升。

本文将会讨论引入***着色器组件****（****shader components****）*，一种一类着色器程序模组以解决这个问题。着色器组件会封装着色逻辑以及在该逻辑被使用时所必须要绑定的参数。从着色器程序编写者的角度来看，着色器组件提供了一种管理庞大的着色器代码库的方法。从游戏引擎的角度来看，着色器组件不但提供了用于将某种渲染特性组装进完整着色器程序的逻辑单元（如某个特定的材质，某个动画效果，和一个光照模型），而且提供了着色器参数块的粒度大小。当着色器语言直接支持着色器组件之后，我们的系统能够提供服务，例如对接口的着色器组件的静态检查，对使用中的特定组件（特性）的完整着色器的具体化，以及生成有助于高效绑定的着色器参数块接口.这些服务会使得良好编程的引擎充分享受到模块化着色器开发的优势，同时体会到现代图形API原本应有的高渲染性能/低CPU占用。

## 背景

## 2.1 如何让渲染器的性能更好

为了获得更好的性能表现，渲染器必须尽可能少地改变GPU状态。这些状态包括固定功能管线阶段的设定，可编程管线阶段的已编译着色器内核，以及需要绑定的着色器参数的集合。

改变GPU的状态是有时间开销的，CPU开销和GPU开销都有。CPU开销包括应用程序和驱动在准备和下达硬件指令上的时间开销。GPU开销包括某些可能的工作需要先在旧GPU状态下执行带来的硬件流水线延后。这两种开销都会对帧率有负面影响，具体取决于应用是CPU密集型还是GPU密集型。如果在渲染过程中能够减少CPU开销，那么被释放出来的CPU资源可以供给引擎的其他关键任务，比如AI，动画，声音或者输入处理。渲染过程中的每个毫秒都很重要。尤其是VR环境下，每帧的间隔只有11ms。

为了减少状态改变，引擎通常会根据不同的切换速率重新组织渲染流程。比如，一次渲染过程可以被看作一系列的嵌套循环，代码如下：

for(v in view)

for(m in materials)

for(o in objects)

draw(v, m, o);

其中的关键在于，某些状态（比如每个对象的形变）的变化速率比其他状态（比如视角参数）要高。

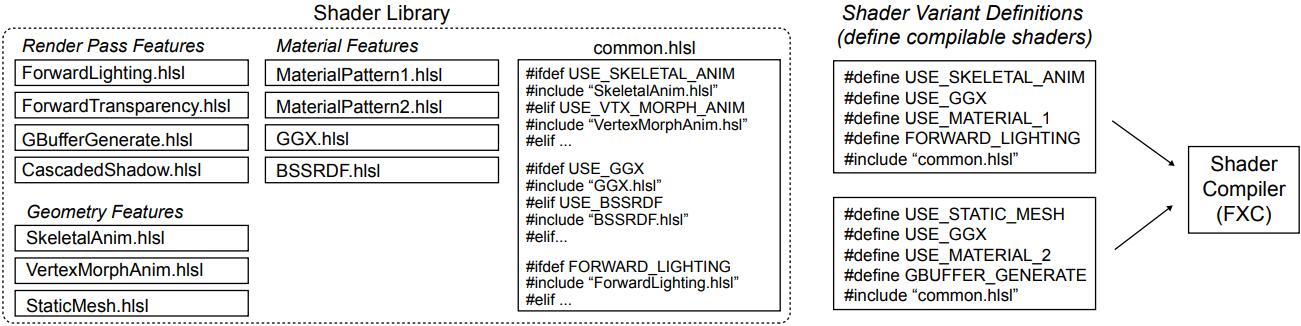
为了在GPU状态改变时最大化CPU效率，图形API提供一种机制，允许把状态组合成对象，从而可以以一个整体的形式进行更新。例如在Direct3D 10中，它将固定功能的状态组织为粗粒度对象。现代API，比如Vulkan和Direct3D 12 允许将着色器参数的绑定状态组合成模块，并称之为描述符集合和描述符表格。为了便于理解，本文将上述API对象称为着色器参数块。多个参数块可以同时被绑定到GPU流水线，和被着色器取用。

将参数绑定状态组合成块具有两个主要优点。一、参数块自身的块性质意味着一次API调用就可以改变很大一部分的GPU状态；比如所有的材质绑定参数。二、由于多个参数块可以被同时使用，则状态可以在一个期望的更新频率下进行改变。比如上文中的嵌套循环，如果参数被组成了参数块，那么一个引擎可以确保只有最内循环那一小部分的状态需要改变，从而省下运算力。

在现代API允许着色器参数被定义为使用多重参数块的时候，GPU状态的其他方面被设定为使用一整块流水线状态对象（pipeline state object，PSO）。一个PSO包括固定功能阶段的状态，以及将要使用的着色器内核的集合。我们将每个不同的内核组合成为一个着色器变体。本文会忽略固定功能状态，而专注于着色器变体的选择和生成。对如何处理着色器变体的陈述可以不失一般性地扩展到PSO上。

## 2.2 从模组中组合着色器

再观察一下前文的嵌套循环。在内循环中，如何正确地选择着色器变体，也许需要考虑到许多不同来源的数据。object可能提供动画，曲面细分，或其他几何效果。material可能提供材质层的组成和获取的逻辑。而该次渲染过程自身可能提供关于如何计算光源的逻辑、如何写入G-buffer，等等。图表1描绘了从这些不同的关切角度出发，组成着色器变体的一种方法。



图表1. 一个以代码片段形式组织而成的着色器库，这些片段结合起来定义了一个可编译的着色器变体。在上例中，代码的模组化程度由HLSL预处理器决定。不同变体的启用和禁用通过预处理器的define来实现，而所需的代码则通过对通用着色器模版“common.hlsl”进行选择性include来提供。由于模组化是通过预处理器宏定义，则模组化信息会在着色器变体代码进入着色器编译器时丢失。

对于一个有着庞大着色器和效果库的引擎来讲，它希望能够将不同的关切或特性定义成可组合的模组。比如说，任意给定的表面着色器应该能接受任意给定的光线，反之亦然。在图表1中，不同的模组被画为不同的长方形，按不同的关切进行分组。

在其他领域中，将分头开发的模块，凭借某种程度上的间接（例如使用函数指针），在运行时动态组合到一起是很常见的。然而，在实时着色器的环境中，运行时间接的开销通常不可接受，而且几乎所有的着色器组成都是由静态具体化得到的。

如图表1中所描述的，让引擎进行具体化操作的一个典型方式就是使用C语言预处理器去进行临时代码生成。对于某个给定特性，其代码被一些#ifdef保护，由一个#define启用。这种情况下，着色器编译器必须在每一个特性组合里都要被调用，以生成一个“链接”着不同特性的静态具体化的变体。在游戏发行时，所有必需的变体都通常会提前编译好，并储存在某个变体数据库里。

由于具体化的变体都是静态生成的，则若想要运行时动态组合模组的行为符合预期，就必须要在数据库中查找到一个正确的变体。该查询所需的键值会取决于对最终着色器代码有影响的一切，比如object, material, 以及单次渲染过程。

## 2.3 模组化着色器和高效绑定无法共存

理想情况下，一个游戏引擎应该能够以模组化的风格定义并组合着色器特性，同时通过对参数块的使用和更新频率的知识实现高效参数绑定。而在实践中，在着色器语言的常见使用方法中是很难同时实现这两个目标的。为了描述其挑战性，本文将使用虚幻4引擎，它是一个同时考虑到模组化和效率的游戏引擎的范例。

虚幻4的着色器特性以HLSL编写，其中使用了传统的#ifdef技巧。为了加强动态组合时的模组化程度，许多重要的着色器特性都在主代码中暴露为C++类。这些类的实例封装了着色器特性的选项（#ifdef 标志）以及这些特性所需的着色器参数值。于是，虚幻4中着色器特性的组合就由C++对象的运行时组合来表达。

在现代API中，为了高效地绑定着色器参数， C++对象由于包含了着色器参数的具体值，它和参数块的协作就显得非常重要。于是该参数块的布局就需要从对应的着色器代码中派生出来——而这里就是问题所在。

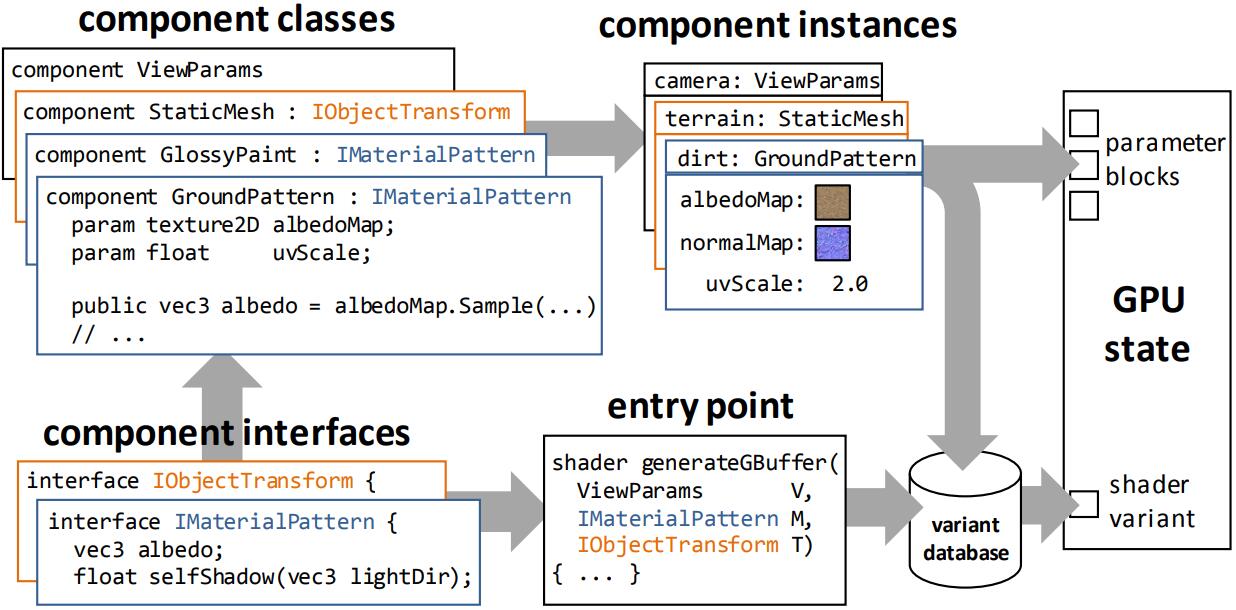
当着色器模组通过使用预处理器技巧以临时代码风格被定义时，着色器编译器没办法在参数组上利用模组的信息达到高效绑定的目标。比如说，GLSL编译器的默认行为是把所有的着色器参数组合成一个块。那些依赖这种默认编译器行为的引擎，在切换着色器特性时就必须填充并绑定一个完整的参数大整块，就算仅仅只需要修改很少的几个参数也得照做。

为了更高效地绑定参数而使用多重参数块，开发者眼下必须使用显式指令进行参数组合，即手动指定参数到固定的绑定地点。然而，该方法在模组着色器的开发中是有代价的，即成组和布局决策必须在所有的模组之间进行协调。比起接受这种复杂程度，许多引擎的开发者，包括虚幻4，在最初将他们的引擎升级至Vulkan或者Direct3D 12时都选择了更简单的替代方案：参数大整块。

我们观察到这种问题的症结在于着色器语言缺少合适的模组化结构体。有了它之后，现代API当中的着色器参数绑定模型就会获得更高的效率。

## 系统设计

着色器组件在着色器和主应用代码之间起桥梁作用，因此我们需要将面向着色器设计的决策和面向应用设计的决策放在一起讨论。一个最高优先级的设计目标是要让着色器组件感觉自己是单相干的，尽管它可以同时从CPU和GPU代码被访问。



图表2. 着色器组件的概念模型。GPU状态由着色器入口函数和着色器组件的实例构成的参数而驱动。着色器特性和它们的参数均被定义为组件类。在现代API中，这些类的实例封装了参数的值和参数块的索引图。一个入口和组件实参共同决定了一个着色器变体。组件类和实例均以不同的颜色来和它们实现的接口进行匹配。

## 3.1 一个着色器组件同时封装了特性代码和参数

一个典型的着色器特性，例如表面材质，需求一定数量的参数，例如材质贴图，以完成它的计算工作。同一类型特性的不同实现方法（比如不同的材质）通常会需要不同的参数。为了允许一个特性的多种实现能够轻松地换进换出，必须要想办法封装它们的参数。

在我们的设计中，一个着色器组件类是一个特定着色器特性的代码和参数的模组化定义。比如说，图表2中的GroundPattern就是一个组件类，它包含一张反射率纹理贴图的参数和一个用于顶点材质坐标的放缩因子。另一个材质组件类，比如GlossyPaint，通常会拥有不同的参数。

## 3.2 一个着色器组件对主代码暴露为一个对象

着色器组件模组化的优势应该同样被其主代码享用。具体来说，对代码和和参数的封装应该被保留，从而特性的不同实现和/或不同参数值的组合的切换可以在一次操作内完成。

在着色器组件类这个想法之上，我们的设计允许主代码从这些类当中创建出叫做着色器组件实例的对象。一个实例储存着类中声明的参数的实际值。例如，图表2中的dirt就是GroundPattern类的一个实例，它把反射率贴图绑定到一个特定材质上。

## 3.3 一个着色器组件描绘出一个参数块

通过使用多重着色器组件实例，一个应用程序可以很方便的在着色器参数集合之间切换。为了让这个操作更高效，每个由引擎创建的组件实例都有目标图形API的一个参数块作为支撑。在图标2中，组件实例dirt可以被用于设定GPU状态中的一个参数块。

值得注意的是，着色器组件类由着色器编译器实现，与此同时，着色器组件的实例是由具体游戏引擎实现的。我们并没有特意要求一个一刀切的，为所有引擎都能实现组件实例的运行时库；而是希望编译器框架提供的服务能够允许引擎使用参数块来高效地实现组件实例。

## 3.4 接口上的入口被参数化

在我们的设计中，游戏引擎中的每个渲染过程都对应着一个着色器入口函数，该函数负责协调当次渲染中所有着色器代码的执行和数据流动。例如，图表2中的generateGBuffer入口对应着渲染过程中的一次G-buffer生成。

设计中的一个关键点在于，一个入口应该被看成是不完整的，有“洞”的函数，特定的组件会插在这里。这些洞从技术角度来讲仍然是参数，但是不应和着色器参数搞混了，所以我们没有使用“参数”这个术语。洞的形状可以由实在的组件类给出（例如图表2中的ViewParams），或者由组件接口给出。例如，generateGBuffer()入口就取决于一个必须要实现IMaterialPattern接口的表面材质组件。

一个组件接口代表了着色器库里的某一类特性（材质，光照等），并且声明了该特性必须要实现的方法和作用域。比如说，图表2中的IMaterialPattern接口声明了一个叫做反射率的域；于是GroundPattern类必须定义一个对应的域，从而实现该接口。

## 3.5 组件使用静态多态

我们在着色器组件中使用的术语和面向对象概念中的术语相同，比如类和接口。它们都很典型地和动态调度有关联（比如虚函数表）。然而我们模型的语义属于静态多态，即为不同类型的组件组合生成不同的代码。总的来说，可以将着色器入口函数想象成如下这样：

shader Simple( ILight light, IMaterial material ) {...}

它是如下“模版化”定义的语法糖：

shader Simple<L, M>( L light, M material ) {...}

一种替代方法则是为组件实现动态调度。但是正如在2.2节中讨论过的那样，实时着色器代码总是会从其激进的具体化中受益；基于预处理器的默认方法就是这样的，而静态多态可以为组件获得类似的结果。

## 3.6 引擎负责变体的查询和缓存

在运行时，引擎会用合适的组件填满一个入口函数中所有的洞，顺带着选定了要用的着色器参数和变体。在我们的设计中，一个变体仅仅依赖于用作形参的组件类，而不依赖于动态参数值，所以就有可能提前将可能的组件类的空间枚举出来，并填充进变体数据库里。

在变体数据库里的查询性能是至关重要的，因为变体很有可能在渲染的最内循环被选定。我们的设计将变体查询和缓存的责任交给引擎，同时让编译器提供特定的服务以让具体实现更高效，而不是试图在语言运行时库里实现一个一刀切的变体数据库。

## 3.7 组件支持多速率编程

为了更好的支持着色器模组化，我们的设计是基于多速率着色语言上实现的。特别的，多速率渲染可以用于支持那些必须和任意数量每顶点属性一起工作的几何效果的模组化定义。那些使用GPU的可编程细分着色器和几何着色器阶段的大多数效果都符合这一描述。我们还发现，将着色器入口表达为多速率程序、横跨多流水线阶段协同工作是一件很自然的事情。

我们的多速率实现方法基于RTSL，Spark，和Spire。在Spire中，我们在单一速率代码中让速率限定符隐藏起来，使得大多数程序员不需要对多速率结构体有感知。

## 评估

## 4.1 着色器组件库

正如在2.2节中描述过的那样，一个模组化着色器代码库允许许多不同的渲染效果从可重用片段中组成出来。为了获得编写着色器组件的经验，我们将一个很大的渲染效果库实现为组件。

我们的着色器组件库在可公开获取到的3A游戏引擎中有功能性呈现之后，它被模仿，并一直包含着超过40个组件。该库包括如下特性：

* 骨骼动画
* PN三角形曲面细分
* 级联阴影贴图
* 平行光照
* 基于物理的环境光照
* 视差遮蔽映射
* 大气散射
* 材质图案生成
* 材质定义的顶点动画
* 双面光照
* 透明遮罩和alpha遮罩

我们的渲染器同样提供三种实现成着色器入口的渲染过程：为不透明对象提供的G-Buffer生成过程；为透明对象提供的前向光照过程；以及阴影贴图的生成过程。

将着色器授予使用着色器组件的权限之后，我们能够将不同的关切干净利落地分割成独立的模组（例如将PN三角形曲面细分从顶点变换和材质图样生成当中解耦出来），并且通过组件接口让模组之间的数据依赖更加清晰。通过开源的3A引擎对着色器库全面细致的检查，可以说我们的基于模组的着色器库要很明显地更整洁，精炼，而且在同样的模组化程度和性能要求面前，我们的库比使用了C预处理器的实现方法要更容易阅读。

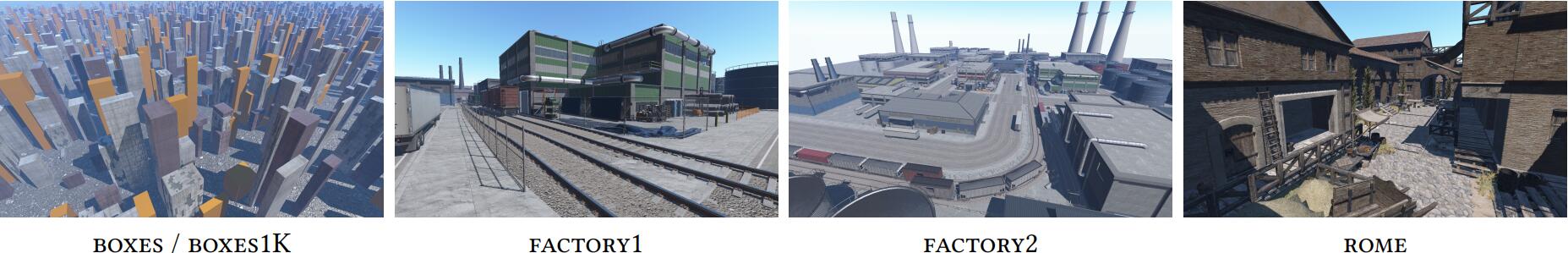
## 4.2 性能

一个精心设计的渲染器应当能够在使用现代图形API时，通过着色器组件库获得很高的着色器参数绑定性能。具体来讲，它应当在渲染多材质复杂场景时具有低CPU占用。在CPU繁忙型渲染情景中，减少掉的CPU占用自动会转化成更高的帧率。在GPU繁忙型渲染情景中，减少CPU占用依然重要，因为空闲出来的计算力可以用于其他任务（例如AI，动画，物理等）。

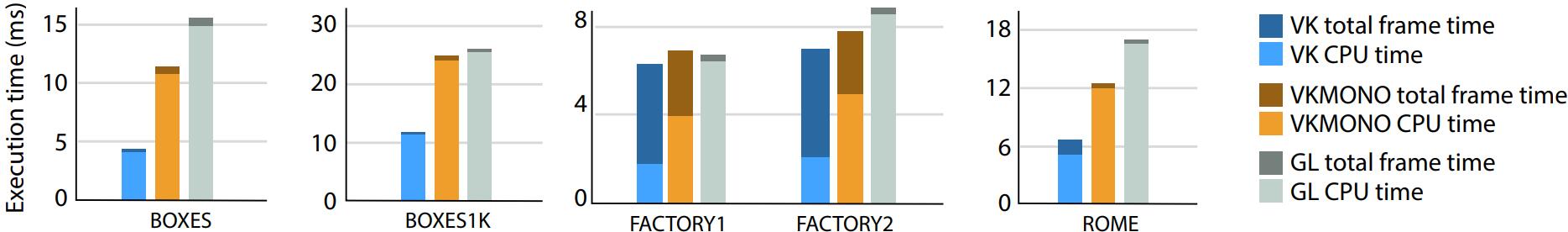
为了理解高效管理着色器和高效着色器参数绑定所带来的性能优势，我们实现了一个基于Vulkan的渲染器（下文称vk）。该渲染器在场景加载时创建所有组件的实例，并且给每一个实例预分配一个Vulkan描述符集合。组件实例的切换会导致一个Vulkan描述符集合的绑定。在绘制具有不同材质参数的连续对象时，只生成了三条Vulkan指令去绑定材质的描述符集合、对象组件，并签发签发绘制命令。为了避免无用的CPU负载或GPU流水线拖延，渲染器在不透明对象渲染过程中按材质对场景对象进行排序，并只在需要时对新的着色器变体进行绑定。

我们将vk的性能与其他两个基准渲染器进行对比。vkmono是基于Vulkan的渲染器，它通过在每个绘制命令之前动态分配和初始化一个包含着所有着色器参数的描述符集合的大整块的行为来模拟许多先进游戏引擎的常见做法。gl是一个精心优化过的基于OpenGL的渲染器。它在每两个绘制命令之间只执行必要的状态改变API调用。所有的渲染器都以单线程工作，所以基于Vulkan的渲染器并没有占有Vulkan提供的CPU的并发优势。

我们对三个渲染器在三种不同的微评分场景（用于模拟压力测试），以及两个购买自虚幻市场的完整场景中进行评估。

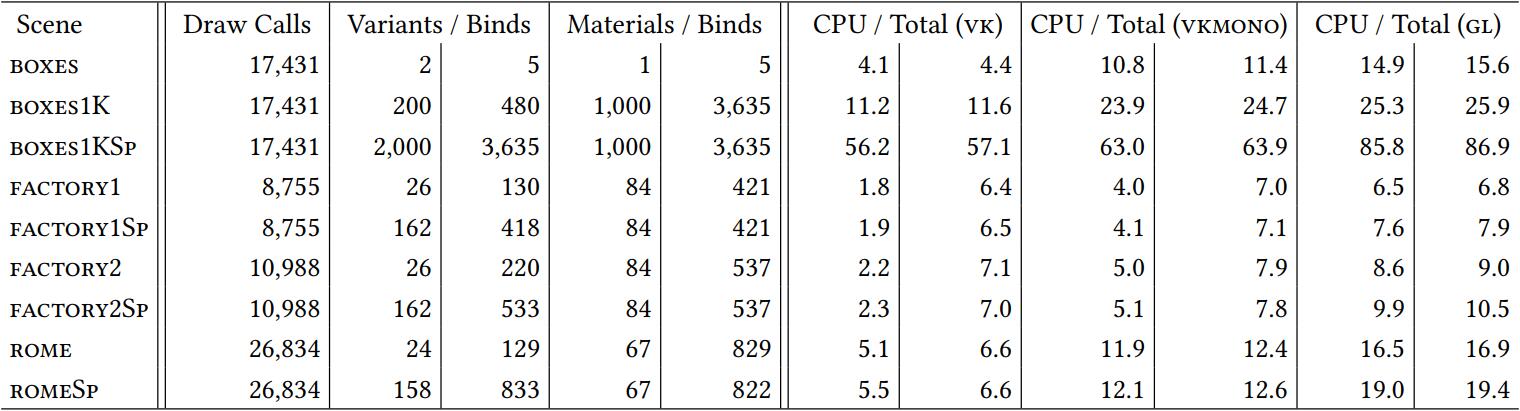


图表3. 性能评估所用的场景一览。这些场景的具体数据可以在表格1中找到。



图表4. 三个渲染器的单核CPU时间（ms）和完整帧时间。所有的实现都做到了高帧率吞吐，但vk与vkmono和gl相比起来，CPU占用时间减少了两倍。注意完整帧时间包括了CPU时间；条柱是重叠的，不是堆叠的。完整结果参加表格1。

* BOXES场景：包含10000个随机放置的盒子。所有的对象共用一个材质（导致了两个着色器变体：一个用于前向渲染过程，另一个用于生成阴影）。由于该场景每次绘制命令只需要绑定一个参数块，它起一个渲染性能的高标准标杆作用。
* BOXES1K场景：和上一个场景基本相同，但拥有1000个不同的材质，每个入口（总共200个）需求100个着色器变体。该场景意图模拟高材质多样性。
* BOXES1KSP场景：同样的场景，但每个材质（总共2000个）拥有一个不同的着色器变体。该场景模拟引擎通过把材质参数以即时常量值预制到代码里的方式挑选着色器（同虚幻4工作原理）。
* FACTORY1和FACTORY2：购买自虚幻市场。场景拥有84个材质（26个着色器变体）并且包含有透明的几何体（阻止按材质排序）。这会增加渲染时所需的着色器变体的数量和材质切换的次数。FACTORY1SP和FACTORY2SP是同一场景的不同版本，每个材质都有一个上文提到过的特殊的着色器变体（总共162个变体）。
* ROME（和它的特殊版本ROMESP）是来自虚幻市场的第二复杂的场景。



表格1. 所有场景的全部数据

所有场景均以1920×1080分辨率渲染，场景包括平行光照和四级联阴影贴图，以及基于图像的环境光照。场景的预览图可在图表3中找到，而关键数据诸如绘制命令的次数，着色器变体的变化，着色器参数的每帧变化次数，则在表格1中可见。

图表4展示了所有渲染器在所有场景中的性能表现。细节数据则在表格1中。测试环境运行在Intel i7-5820k（单线程渲染）和Nvidia GeForce GTX960Ti上。所有的渲染器都达到了高渲染吞吐量，尽管每个场景都有着高绘制复杂度。这说明测试基准都是经过良好优化的。

由于vkmono基准遵照商业引擎的通常做法，所以它获得了类似虚幻4的性能表现（当它被设置为使用D3D12模式时）。例如，vkmono在渲染factory1sp场景时，每帧时间为4.1ms，而同机器上的虚幻4引擎仅在基础渲染和阴影渲染过程上就花了4.0ms的CPU时间。这一对比并不是很直接；我们将虚幻4独有特性的耗时排除掉了。因此，该数据仅可以认为vkmono是一个非常坚固的基准线。

高效参数绑定的优势：图表4中的结果显示出，将着色器参数状态的改变程度最小化，可以有效地减少CPU消耗。在factory1，factory2和rome场景中，vk渲染器的CPU消耗比vkmono少了两倍以上。该程度的CPU节省不但可以转化为5%-50%不等的帧渲染时间减少，而且最重要的好处是，vk渲染器的高效率可以让引擎的其他子系统受益。正如我们所期待的那样，vk的性能优势在boxes微测试中发挥到了极致。其中，vk的CPU时间比vkmono和gl分别减少了2.6倍和3.6倍（vk渲染器在单一CPU核心的情况下，每秒签发大约400万次绘制命令）。CPU耗时差距在boxes1K测试中，减少到2.1倍和2.3倍。该场景中存在1000个不同的材质，因此需要更频繁的参数绑定。

更激进的着色器具体化的效果：正如期待的那样，在当引擎过度根据参数值具体化着色器时，vk比vkmono的性能优势会缩水。这是因为这种风格的渲染器设计需要频繁的参数和着色器变体绑定的改变。该效果在boxes1KSp场景中非常明显。其中vk仅仅提供了10%的CPU占用优势，而不是boxes1K中的2倍以上。在更贴近现实应用的场景中，就算具体化比较激进，但vk的优势依然十分明显：在factory1Sp，factory2Sp和romeSp中，vk比vkmono的CPU占用又减少了2倍以上。

人为制作的boxes1K场景是为了评估频繁材质改变的开销。boxes1K和boxes1KSp使用了比factory和rome要多10倍以上的材质。在这种程度下，每材质和每变体的工作量开始在帧渲染时间中占主导地位，所以由过度具体化带来的额外的绑定工作有了更大的影响：vk对boxes1KSp的渲染时间是boxes1K的5倍以上。

动态查询着色器变体的开销：最后，我们测量了动态查询着色器变体的CPU开销。虽然我们的实现可以进一步的优化，但我们发现变体查询的开销已经很低了：在factory中少于5%的总CPU时间，在boxes压力测试中则少于12%。当对象具有静态指定的材质，且流水线状态在渲染对象时不需要动态改变时，那么将着色器变体和缓存到和其相关的对象中储存是可行的。这样就避免了任何的动态查询。我们实现了该优化，但是在这一节的结果中并没有启用它。

## 其他相关工作

我们相信着色器组件是一个很优雅的解决方案，拥有着精心挑选出来的设计理念。这些理念中的许多都已经在之前的系统中出现过，所以在本节中我们将指出之前那些实现方法的共同点。在某些情况下，之前的工作已经被赋予了不同的术语，语法或者实现，因此我们将它们全部对齐至着色器组件的术语和概念模型上。

## 5.1 组件要封装代码和参数

Cg接口，HLSL接口以及Spark允许组件使用面向对象的类以着色器语言进行表达，它封装了代码和相关的参数数据；我们的系统遵循这一方法。在Sh中元编程的使用，允许着色器组件在主代码中以C++类的形式实现。RTSL则使用了函数语法，不过它受到了RenderMan Shading Language的启发；其中着色器在语法上被视为类。

GLSL着色器子程序引入了一个模组化的结构体（一种受限制的函数指针），它可以将代码的某种选择封装起来，而不需封装对应的参数值。

许多着色器代码在生产时都以过程式风格写就，一般采用HLSL或者GLSL，没有使用接口或子程序特性。在HLSL和GLSL中，一个很重要，但很少被提起的特性：着色器参数通常被声明成全局的（而不是更自然更明确的做法：声明在某个着色器入口的参数列表里）。在全局范围内支持参数声明之后，就可以通过在单独的文件中定义每个特性的代码和参数来实现临时模块化，于是可以通过预处理器来控制特性的包含与否。

## 5.2 组件对主代码暴露为对象

正如在2.3节中讨论过的，虚幻引擎将重要的着色器特性对主代码暴露为C++对象。在Sh中，当组件被写为主代码的应用类时，这种使用方式也可以得到Sh的直接支持。在Spark中，它会自动生成类似的C++类以将着色器组件暴露给主代码。

RTSL的运行时API合并了着色器组件和实例，它只影响到那些每个类只允许存在单一实例的情况。GLSL由于其子程序只封装代码，不封装数据，它不需要关心组件的实例情况。

Cg接口的运行时API允许组件实例以句柄形式分配和引用。Direct3D 11 API（支持HLSL接口）仅仅包括了最小程度对组件实例的底层支持；在其之上应该覆盖一层更完整的类Cg API。我们的工作在这一方面更接近于HLSL和Direct3D 11，因为我们一直坚持，组件实例运行时库最好在引擎代码里实现，而不是一个一刀切的运行时库。

我们的系统首次提出将组件用着色器语言描绘到参数块上。然而之前的系统早于Vulkan和Direct3D 12出现，所以参数块不能作为一个可实现的选择。

## 5.3 在接口上的入口函数要参数化

Cg和HLSL都允许着色器入口和任意函数在接口上参数化。GLSL着色器子程序及其使用站点将会针对分开声明的子程序类型进行静态检查，这些子程序类型起接口作用。在使用Sh时，一个主应用的函数可以用作入口，同时C++抽象类用作接口。

Spark没有单独的入口概念，而是期望渲染过程的整体数据流仅由组件组合产生。

起初RTSL看起来像是没有入口或者接口的概念。然而RTSL支持surface和light着色器的声明，这些关键字可以被看作是将每个着色器和两个内建的接口之一联系在了一起。RTSL里有一个内建的入口，它在一个单一的表面着色器组件和零到多个光照组件上参数化。

## 5.4 多态组件调度的语义

Cg和Spark都将他们的面向对象语法用静态多态实现，我们的系统也是这样的。由于Sh是元编程的，在主代码里进行动态调度可以获得相同的结果。

作为对比，HLSL接口和GLSL子程序都设计为暴露动态调度的一种有限形式。例如，HLSL的前端将一个通过接口的调用，通过使用跳转表翻译为间接分支。动态调度的使用可以有效地将GPU效率交换为CPU占用的减少（因为在交换着色器内核时的状态切换会更少）。我们的经验建议是：由于CPU占用的减少大概率是由现代API造成的，所以这种交换也许并不值得。

我们的设计与之前的系统的一个关键区别在于我们坚持着色器变体的生成，具体化和选择应该交给每个引擎特定的代码来控制，而不是由不透明的语言运行库或者GPU驱动来控制。

## 讨论

本论文指出，模组化着色器开发和高性能渲染是可以兼得的。通过向实时着色语言中引入一类模组化结构体：着色器组件，并在现代图形API中高效地实现这些结构体，我们揭示出模组化和性能可以互补，而不是两个互相冲突的目标。

我们的贡献在于观察到了当图形API围绕着状态变化率进行组织时，和着色器代码库围绕着特性进行模组化时有着类似的分解现象。尽管这种观察并不难，但是前人并没有明确这一点。

我们在第五段中讨论讨论的两种模组化结构体——GLSL子程序和HLSL接口——都受到了主流图形API：OpenGL和Direct3D 11的支持。然而这并没有任何意义，因为对这些结构体的支持在API转向Vulkan和Direct3D 12的时候被移除了。我们将这个现象认作更进一步的证据，证明高效渲染和着色器模组化之间的联系并不明显，甚至领域内的专家也很难观察到。

在向现代图形API的转换过程中，我们也看到了给着色器用的低级中间语言（low-level intermediate language, IL）接口的出现，例如SPIR-V。在IL接口来到由游戏开发人员建立，为游戏开发人员服务的着色器语言和工具生态系统内之时，以及自由实验和跳出任何API标准流程的创新精神，我们希望我们的工作能够对新一代着色器编程模型的产生提供些许灵感。

## 参考文献

[1] Shader Components: Modular and High Performance Shader Development He Yong, Foley Tim, Hofstee Teguh, et al. ACM Transactions on Graphics, 2017, 36(4)