

三 维 动 画 和 交 互 设 计 课 程 读 书 报 告



作者姓名 杜 耀

作者学号 21860392

年级专业 2018级宁波分院2班

所在学院 工程师学院

完成日期 2018.12.25

Research on Computer Animation and Interaction Design

A Report Submitted to

Zhejiang University

Major Subject: Software Engineering

Advisor: Li Qilei

By

Du Yao

Polytechnic Institute, Zhejiang University

2018

# 目录

[目录 3](#_Toc533605695)

[一、 论文简介 5](#_Toc533605696)

[二、 该文研究背景与问题描述 5](#_Toc533605697)

[2.1 研究背景 5](#_Toc533605698)

[2.2 主要观点 5](#_Toc533605699)

[2.3 论文贡献 6](#_Toc533605700)

[三、 设计与实现 6](#_Toc533605701)

[3.1生成和使用着色器变体 6](#_Toc533605702)

[3.2材质着色的分离阶段 9](#_Toc533605703)

[3.3专业照明环境 10](#_Toc533605705)

[3.4添加依赖于BXDF的光类型 14](#_Toc533605707)

[3.5重构一个渲染器来使用Slang 16](#_Toc533605708)

[四、 结果分析 17](#_Toc533605709)

[4.1性能方面 17](#_Toc533605710)

[4.2可扩展性 19](#_Toc533605711)

[4.3总结 20](#_Toc533605712)

[五、 思考与建议 20](#_Toc533605713)

[参考文献 21](#_Toc533605714)

# 论文简介

论文题目：Slang: language mechanisms for extensible real-time shading systems，Slang：实时可扩展着色系统的语言机制。

作者： YONG HE, Carnegie Mellon University

KAYVON FATAHALIAN, Stanford University

TIM FOLEY, NVIDIA

发表于：ACM Transactions on Graphics，2018年7月

# 该文研究背景与问题描述

## 2.1 研究背景

实时渲染引擎的设计者必须平衡两个相互矛盾的目标：即着色系统的可扩展可维护性和提高系统的渲染性能。为此，引擎架构师们已经为创建着色系统发明了诸多有效的设计模式，并且开发了不少特定于引擎的代码集成工具，从预处理器劫持到特定领域的着色语言，以有效地实现这些模式。然而这些专有工具给现代引擎增加了极大地复杂性，缺乏高级语言特性，并为学习和应用他们制造了多余的挑战。

## 2.2 主要观点

这篇文章提出了一种新的思路来解决上述设计实时渲染引擎的矛盾，不同于以往设计师们的编写专有工具的方法，他们认为专有工具带来的优势完全可以直接使用底层GPU着色语言来实现，只需要采用少量来自现代成熟的编程语言的最佳实践原则予以扩展即可。这些原则包括添加接口约束、关联类型的泛型，以及对现有的类C风格的GPU着色语言的接口/结构扩展。

## 2.3 论文贡献

该论文的主要贡献是确定了一批必要的通用模型编程语言特点，它们对于构建可扩展可维护且高性能的着色系统至关重要，具体而言本论文的贡献如下：

1. 提出了Slang着色语言，属于HLSL的一种变体，增加了如下通用语言的特性：具有接口边界和关联类型的泛型、以及接口/结构扩展。以上特征是作为满足性能要求的最小扩展集同时为当前的HLSL提供了一种增量开发的途径。
2. 定义了Slang运行时API，它为渲染器提供内省阴影效果模块和将阴影效果集合装配成高效的静态优化的GPU着色器的服务。Slang被用来为定义着色效果和内省与编译着色器提供与着色无关的机制。这使得引擎可以重新获得对于决定关键的特定于应用的着色器优化执行策略的控制权。
3. 贡献了一个利用Slang的特性实现的面向大型可扩展渲染库参考设计。并且使用这个库和Slang的编译器服务重构了一个开源的着色器。结果证明最终的着色系统足够的简单，易于扩展新特性，而且对比原本的HLSL实现还提高的性能。

# 设计与实现

## 3.1生成和使用着色器变体

清单3显示了一个类似于图1中的forwardPass的片段着色器入口点，它使用泛型和接口边界而不是预处理器启用的元编程来实现特化。材料和光照效果的选择分别用通用类型参数M和L表示;着色器主体评估材料的表面图案，然后请求整合来自照明环境的入射照明。插入M和L的具体类型产生此入口点的特殊变体，这些步骤具有不同的行为。

在类型参数上使用Slang接口边界允许编译器对typeMain进行类型检查，并确定它与实现指定接口的任何材料和光类型兼容。即使对于在单独编译的文件中实现的类型，也保证了兼容性，因此我们的静态检查保证也有利于可扩展性;用户可以放心地扩展现有入口点以支持新效果。

清单3还演示了如何使用参数块来封装着色器参数，以便使用现代图形API进行有效通信。在此示例中，材料参数块使用泛型类型参数M，因此选择具体材料类型不仅可以影响专用变量的行为，还可以影响它接受的参数。

渲染器可以使用Slang编译器的运行时API（第5节）提供的反射信息为特定材料类型分配参数块。该API还可用于使用所选材料类型生成forwardPass的专用变体。

通过对泛型和参数块的不同机制使用语言和API支持，这个着色器系统实现了着色器组件设计模式，而不需要具有一流“组件”的DSL [He et al。 2017年]。

float3 forwardPass<M : IMaterial, L : ILightingEnv>(

ParameterBlock<Camera> camera,

ParameterBlock<M> material,

ParameterBlock<L> lights,

SurfaceGeometry geometry)

{

float3 viewDir = normalize(camera.P - geometry.P);

M.Pattern bxdf = material.evalPattern(geometry);

return lights.illuminate(bxdf, geometry, viewDir);

}

清单3。片段着色器入口点作为泛型函数写入类型参数。为这些参数插入不同的类型就会得到结果着色器的变化具有不同的行为，和不同的参数数据

interface IBxDF {

float3 eval(float3 wo, float3 wi);

}

interface IMaterial {

associatedtype Pattern : IBxDF;

Pattern evalPattern(SurfaceGeometry geom);

}

清单4.用于定义表面反射函数和表面材料的Slang接口。 IMaterial接口的每个实现都将与它产生的特定类型的反射功能相关联。

struct Lambertian : IBxDF {

float3 albedo;

float3 eval(float3 wo, float3 wi) {

return albedo / PI;

}

}

struct TexturedLambertian : IMaterial {

typedef Lambertian Pattern;

Texture2D albedoMap;

SamplerState sampler;

Lambertian evalPattern(SurfaceGeometry geom)

{

Lambertian bxdf;

bxdf.albedo = albedoMap.Sample(sampler, geom.uv);

return bxdf;

}

}

struct DisneyBRDF : IBxDF { ... }

struct RustedMetal : IMaterial {

typedef DisneyBRDF Pattern;

DisneyBRDF evalPattern(SurfaceGeometry) { ... }

...

}

清单5.反射和模式生成函数分别定义为实现IBxDF和IMaterial接口的类型（清单4）。.

## 3.2材质着色的分离阶段

如2.3节所述，希望将材料阴影分离为不同的模式生成和BxDF评估步骤。在基于预处理器的专门化系统的上下文中，这涉及有条件地定义存储所选BxDF的参数的类型。当使用泛型进行专业化时，相关类型会提供类似的功能（第3.3节）。

清单4显示了IBxDF和IMaterial接口的Slang声明，分别表示了反射函数和材质表面着色器的概念。 IBxDF的定义很简单，而IMaterial则使用相关类型（第3.3节）来捕获反射函数类型对表面着色器选择的依赖性。IMaterial的具体实现将定义specifec用于关联类型Pattern的类型。例如，清单5中的表面着色器TexturedLambertian将Pattern定义为Lambertian类型，它实现了一个简单的漫反射BRDF。

采用材质类型参数的着色器代码，例如清单3中的forwardPass函数的参数M可以将相关的反射函数类型称为M.Pattern。因为关联类型Pattern是使用IBxDF接口限制的，所以只有该接口提供的操作才能在forwardPass中使用。

例如，尝试访问BxDF的反照率字段会产生编译时错误，因为不是每个BxDF都保证具有这样的参数。

相关类型在第2.3节中实现了与临时方法类似的结果，并且可以独立编译和验证诸如forwardPass之类的入口点和TexturedLambertian之类的材料。因此，当使用新材料扩展引擎时，用户可以确信新材料将适用于需要材料来实现IMaterial的所有入口点。

interface ILightEnv {

float3 illuminate<B:IBxDF>(B bxdf, SurfaceGeometry geom,

float3 wo);

}

struct DirectionalLight : ILightEnv {

float3 direction;

float3 intensity;

float3 illuminate<B:IBxDF>(B bxdf, SurfaceGeometry geom,

float3 wo) {

return integrateSingleRay(

bxdf, geom, wo, direction, intensity);

}

}

struct PointLight : ILightEnv { ... }

## 清单6.照明环境表示为实现ILightEnv接口的类型。 单个光源（例如，DirectionalLight，PointLight）被视为照明环境的简单情况。

## 3.3专业照明环境

2.4节表明，简单的基于预处理器的解决方案同时支持对照明环境的专业化并保留使用新光源类型扩展系统的能力，这具有挑战性。这激发了更多通用的代码生成技术，如字符串粘贴。使用Slang的语言机制，我们的着色系统可以支持对照明环境的专业化和对新光类型的扩展，而无需借助基于字符串的代码生成。

清单6显示了用于定义我们将在此处和下一部分开发的照明环境的框架。 ILightEnv接口声明每个照明环境必须提供照亮表面样本的操作，并且该操作在表面的BxDF中必须是通用的。预期照明操作将来自环境的光整合在方向wo的表面上。

在我们的照明系统中，单个光源被视为照明环境的简单情况。例如，清单6中的DirectionalLight实现了一个符合ILightEnv接口的简单定向灯。为清楚起见，DirectionalLight使用第3.1节中定义的integrateSingleRay函数实现其集成。另外，可以类似地定义诸如PointLight之类的极小光类型。

鉴于清单6中的定义，可以专门化

清单3中的着色器入口点，用于任何单个光源。但是，当需要更复杂的照明环境时，我们可以使用泛型来定义用于构建复杂照明环境的类型，而不是更简单的环境。

清单7显示了两种类型的复合照明环境。

LightArray类型实现了一个动态大小（但静态有界）的灯光阵列。 L type参数表示灯光阵列中元素的类型，而N是通用值参数，它是可能出现的灯光数量的上限。例如，LightArray <DirectionalLight，16>类型表示（最多）16个方向灯的数组。

虽然LightArray用于支持动态大小但均匀的复合，但LightPair类型可用于构成异构照明环境。一双轻便的

LightPair <DirectionalLight，PointLight>

简单地总结其组成照明环境的贡献。

这两种简单类型可用作合成运算符，以构建更复杂的照明环境。例如，如果某个应用程序想要使用带有级联阴影贴图的单个方向光渲染着色器入口点，再加上最多16个点光源，它可以构造类型：

LightPair <CascadedShadowMap <DirectionalLight的>>

ArrayLight <PointLight，16 >>

在实践中，我们预期渲染器设计，其中应用程序使用数据驱动的代码（基于表示场景光源的应用程序或框架数据结构）来为复杂的场景照明环境生成Slang类型。 Slang编译器的运行时API（第5节）为构造这些复合类型提供应用程序支持，并为查询在参数块中存储这些类型的值所需的布局信息提供支持。

通过允许将光合成操作符表示为类型，Slang提高了照明环境的主代码中的抽象级别。引擎现在不是将着色器代码字符串粘贴在一起，而是组合着色器类型，然后创建这些类型的实例。

struct LightArray<L:ILight, const N:int> : ILightEnv {

L lights[N];

int lightCount;

float3 illuminate<B:IBxDF>(Surface<B> surface, float3 wo) {

float3 result = 0;

for(int i = 0; i < lightCount; i++)

result += lights[i].illuminate(surface, wo);

return result;

}

}

struct LightPair<H:ILightEnv, T:ILightEnv> : ILightEnv {

H head;

T tail;

float3 illuminate<B:IBxDF>(Surface<B> surface, float3 wo) {

return head.illuminate(surface, wo)

+ tail.illuminate(surface, wo);

}

}

清单7.使用泛型定义的复合光照环境。 LightArray可用于封装均匀的灯光阵列，而LightPair可用于“展开”异构灯光列表。 每个允许清单3中的简单入口点透明地使用单个灯或许多灯的列表。

struct QuadLight : ILightEnv {

float3 vertices[4];

float3 intensity;

float3 illuminate<B:IBxDF>(B bxdf, SurfaceGeometry geom,

float3 wo) {

return bxdf.acceptQuadLight(

this,

geometry,

wo);

}

}

interface IAcceptQuadLight {

float3 acceptQuadLight(

QuadLight light,

SurfaceGeometry geom,

float3 wo);

}

extension IBxDF : IAcceptQuadLight {}

extension Lambertian : IAcceptQuadLight {

float3 acceptQuadLight(QuadLight light,

SurfaceGeometry geom,

float3 wo) {

return albedo \* LTC\_Evaluate(light, geometry, wo,

float3x3(1,0,0, 0,1,0, 0,0,1));

}

}

extension DisneyBRDF : IAcceptQuadLight { ... }

## 清单8.扩展着色器系统以支持（近似）区域灯光类型。 现有的IBxDF接口无法很好地支持新的灯光类型，因此引入了接受区域灯光的表面的新接口。 扩展声明可用于使预先存在的反射函数（如Lambertian）支持新灯光类型所需的接口。

## 3.4添加依赖于BXDF的光类型

第2.5节讨论了添加依赖BxDF的光类型的挑战，它依赖于BxDF特定的闭合形式评估或性能近似。挑战的关键在于执行的代码取决于光和BxDF。通过在Slang中使用扩展机制，应用程序可以通过将光类型特定操作注入到现有BxDF类型中来解决这一挑战，而无需修改到目前为止呈现的框架的简单抽象（例如，清单6）。

清单8中的QuadLight类型实现了四边形区域光。这种类型看似超级简单，但请注意，照亮实现在BxDF上调用名为acceptQuadLight的方法，而清单4中的IBxDF的原始定义未声明这样的方法。

相反，acceptQuadLight操作被定义为清单8中IAcceptQuadLight接口的一部分。三个扩展声明用于使现有框架类型符合新接口。

首先，IBxDF接口扩展了新的需求，因此任何符合类型的类型也必须支持IAcceptQuadLight接口。接下来，Lambertian和DisneyBRDF BxDF类型使用acceptQuadLight的具体实现进行扩展，在我们的示例中，使用线性变换余弦执行闭合形式近似[Heitz et al。 2016年]。

任何未扩展为支持的BxDF实现新接口将导致编译器错误。 Swift和Rust中的扩展实现需要在扩展IBxDF时提供acceptQuadLight的默认实现;然后，该实现将用作未明确扩展的BxDF的后备。我们正在考虑将此功能添加到Slang中。

扩展将代码“注入”现有类型的能力允许应用程序开发人员编写一个模块，该模块可以扩展渲染器框架的着色系统的功能而不会打扰其现有实施或增加其他用户的复杂性。

与第2.5节中描述的基于预处理器的解决方案不同，清单8中的模块围绕已扩展的内容和新接口要求的显式做出了策略决策（例如，很明显IBxDF的接口契约已经扩展）。

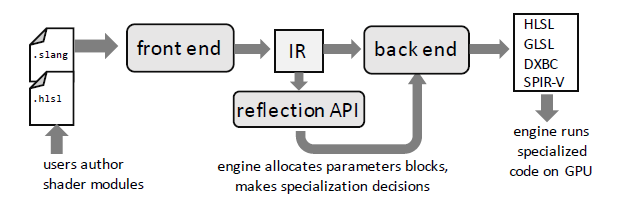


图2. Slang系统实现包括编译器前端（用于检查和确定非专用模块的参数布局）和后端（用于生成专用GPU内核）以及支持API。 应用程序使用Slang反射API获取用于分配GPU参数块的着色器参数布局信息，并指导低级代码生成所需的专业化选择。

// Front End

Module\* loadModule(const char\* path);

// Reflection

EntryPoint\* findEntryPoint(Module\* module, const char\* name);

Type\* findType(Module\* module, const char\* name);

TypeLayout getTypeLayout(Type\* type);

Type\* specializeType(Type\* type, Type\* types[], int ntypes);

// Back End

Kernel\* specializeEntryPoint(EntryPoint\* entryPoint,

Type\* types[], int ntypes);

清单9. Slang实现提供的API服务概述，根据图2中的系统图分组。.

## 3.5重构一个渲染器来使用Slang

Slang的总体目标是在不牺牲渲染器性能的情况下促进大型实时着色系统的高效开发。作为评估这一目标是否已实现的第一步，我们使用Slang重构了Falcor开源实时渲染器的着色器库和着色系统[Benty et al。 2017年]。 Falcor是一个实时渲染框架，旨在加速和支持新的实时渲染效果和算法的原型设计。虽然Falcor旨在模块化和可扩展以支持各种用例，但Falcor还必须提供高性能以支持最先进的实时渲染效果。我们为我们的评估分配了Falcor版本2.0.2，其中包含用HLSL编写的5,400行着色器代码，实现：

•灵活的分层材料系统，可以配置模型复杂的反射函数

•点，点，方向和环境光类型

•使用环境贴图进行光泽反射

•级联，指数和方差阴影贴图算法

•后期处理效果，例如屏幕空间环境遮挡和色调映射

Falcor的材料和照明系统贡献了2,100多行着色器代码，构成了渲染过程中CPU和GPU执行时间的主要部分。我们的重构专注于提高这两个子系统的可扩展性，性能和代码清晰度。

由于Slang是HLSL的扩展，我们能够使用Slang编译器立即编译整个Falcor着色器库（作为fxc的替代）。这使我们能够逐步重构Falcor着色系统，逐步采用Slang的语言功能。重构的细节略。

# 结果分析

通过使用Slang重构后的Falcor的着色系统，本文作者评估了Slang给新系统带来的性能和可扩展性的提升。

## 4.1性能方面

文中比较了Falcor渲染器的所有三个分支的性能：原始分支（基于HLSL，没有光专业化），修改的原始分支（基于HLSL，具有光专业化）和重构（基于Slang）分支。使用ORCA库中的三个测试场景[NVIDIA 2017]： TEMPLE，BISTRO-INTERIOR和BISTRO-EXTERIOR（渲染视图如图3（a）所示）。 这些场景由Unreal和Lumberyard [亚马逊2016]引擎的开发者创建，以展示其引擎的能力并代表现代游戏内容（例如，BISTRO-EXTERIOR场景具有超过280万个三角形）。

图4比较了两个渲染器在每帧生成所有GPU命令所需的CPU时间（上图）和执行所有这些命令的GPU时间（下图）方面的性能。 作者在配备Intel i7-5820K CPU和NVIDIA Titan V GPU的机器上进行了实验，渲染了1920×1080图像。

正如预期的那样，重构后的渲染器在所有测试场景中显著降低了CPU成本(超过30%)。请注意，即使渲染器不是CPU绑定的，降低CPU成本也会为其他游戏引擎任务释放CPU资源。重构渲染器的GPU性能与照明环境专门化的原始渲染器相当。

采用Slang会增加着色器编译的开销，因为Slang编译器输出必须由其编译的HLSL文本现有的HLSL编译器。 加载TEMPLE场景时，1s用于Slang，而4.5s用于HLSL编译。 这比原始（light specialization）分支略慢，这需要4.1s来编译所有着色器变体。 可以通过两种方式改进编译时间。 首先，因为仔细选择了Slang的语言功能来支持单独的编译，所以可以跨多个入口点和变体分摊前端工作。 其次，更重要的是，我们可以通过直接将Slang的IR转换为SPIR-V和DXIL等格式来消除输出HLSL和调用第二个编译器的开销。



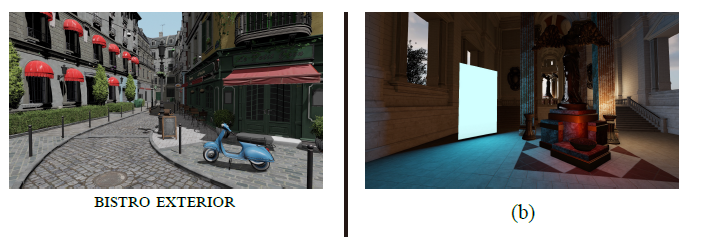


图3.（a）用于评估使用Slang构建的重构Falcor渲染器性能的场景视点（参见图4）。 （b）通过更改单个代码站点，为Falcor渲染器添加了对多边形区域灯的支持。

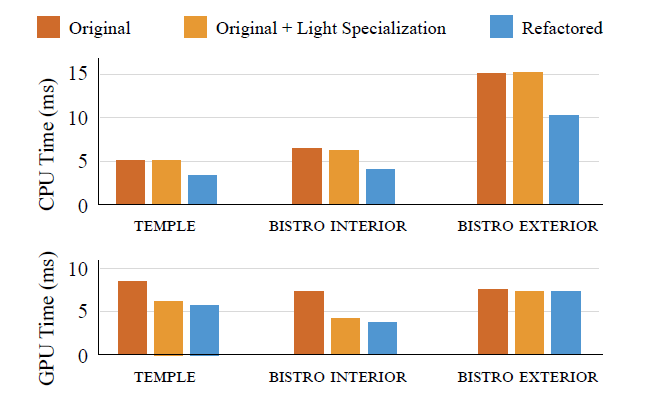
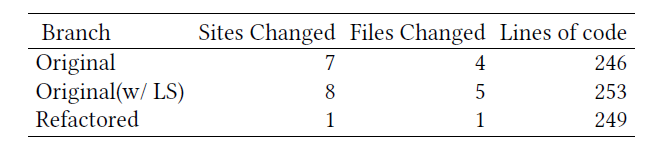


图4.比较原始分支的平均每帧CPU时间（顶部）和GPU时间（底部）（有和没有光专业化），以及Falcor渲染器的重构分支，我们发现重构的渲染器提高了CPU性能 并保留与原始分支相同级别的GPU性能和光专业化。

## 4.2可扩展性

为了评估重构Falcor代码的可扩展性，作者在原来和重构的阴影系统中都添加了一个新的多边形区域光类型。图3（b）显示了使用QuadLight类型的TEMPLE场景的渲染，它使用线性变换后的余弦进行近似估计[Heitz et al。2016年]。

下表总结了每个版本的Falcor所需的着色器代码更改:



将区域灯功能添加到原始代码需要在代码中的七个站点进行更改，跨越四个不同的文件。 这些变化包括：为近似区域灯定义新类型标签，向LightData类型添加新felds，将新分支插入主灯集成循环内的调度逻辑，以及添加逻辑以处理每个支持的BxDF的新灯类型。 支持着色器环境的着色器专业化使得原始HLSL着色器代码更难以扩展，需要在一个附加文件中再进行一次更改。 相比之下，为重构的Slang着色器库添加区域光支持是通过单个文件中的单个代码块（类型定义加扩展）完成的，并且不需要修改任何现有的Falcor函数或类型。

## 4.3总结

通过重构Falcor的着色系统来利用Slang订购的语言机制和编译器服务，我们改进了渲染器的CPU和GPU性能，并使着色器库代码更容易扩展。 定性地，我们发现重构的着色器代码更明确，更清楚地反映了引擎开发人员的心智模型。 虽然对Slang的想法的最终评估将涉及集成到复杂的生产游戏引擎中，但我们与Falcor渲染系统集成的经验一直很有希望。 由于本节描述的好处，Falcor项目现在已经采用了Slang着色器编译器，并且越来越多地在其为应用程序开发人员提供的着色器库中采用Slang的语言机制。

# 思考与建议

本文演示了通过现代通用语言机制特性的扩展，可以使得已有的着色器语言在提高可扩展性和可维护性的同时保持较高的性能表现。作者的观点是与其去开发专门的着色器语言HLSL，还不如使用的已有的C/C++语言用于着色器编程，这样可以充分利用已有语言的优秀特性。作者善于利用已有工具解决新问题的做法值得我们学习。

# 参考文献

[1] He Y, Fatahalian K, Foley T. Slang: language mechanisms for extensible real-time shading systems[J]. ACM Transactions on Graphics (TOG), 2018, 37(4): 141.