外 文 翻 译

**毕业设计题目：****虚拟现实购物平台-WebGL展示系统研 究与开发**

**原文1：shade.js: Adaptive Material Descriptions**

**译文1：shade.js: 自适应的材质描述**

**原文2：Fast robust and precise shadow algorithm for WebGL 1.0 platform**

**译文2：对于WebGL的1.0平台的快速稳健和精确的 阴影算法**

原文1

shade.js: Adaptive Material Descriptions

*German Research Center for Artificial Intelligence, Germany  
2Intel VCI & Saarland University, Germany*

图片包含 照片, 展示, 不同, 室内

已生成极高可信度的说明

**Figure 1:** *A collection of procedural materials written in shade.js and rendered in WebGL using forward shading (upper left),**deferred shading (upper right), Blender Cycles ray-traced (lower left), and with global illumination (lower right). We achieve conceptually equal materials for all four rendering techniques.*

**Abstract:**

In computer graphics a material is a visual concept that is parameterizable and should work for arbitrary 3D assets and rendering systems. Since provided parameters and attributes as well as the capabilities of rendering systems vary considerably, a material needs to adapt to its execution environment. In current approaches, the adaptation logic is ‘baked’ into the rendering application based on string manipulation, compiler directives, or metaprogramming facilities. However, in order to achieve application-independent and self-contained material descriptions, the adaptation logic needs to be part of the material description itself. In this paper we present shade.js, a novel material description using a dynamic language to achieve the necessary adaptivity. A shader can inspect its execution environment and adapt to the available parameters and renderer capabilities at run time. Additionally, shade.js exploits the polymorphism that comes with non-explicit declaration of types. These two novel features allow for writing adaptable and thus more general material descriptions. Based on the concrete execution environment at run time, the accompanied compiler generates specialized shader code that is specifically typed and optimized for the target rendering system and algorithm. We evaluate shade.js with examples targeting four different rendering approaches (forward and deferred rasterization, ray-tracing, and global illumination). We show that we can improve convenience and flexibility for specifying materials without sacrificing performance.

**1. Introduction**

In most games a specialized and optimized GPU shader is closely intertwined with game assets as well as the game engine and renderer. Such a shader will not work on its own and cannot be applied in other scenes. Most other applications, however, would benefit strongly from the concept of more general *materials* that can easily be reused in different scenes and with arbitrary rendering systems. Such materials could be exchanged, refined, verified, organized in reusable libraries, and shared across the Internet. For this to happen, a material needs to be self-contained and independent of 3D scene descriptions and rendering algorithms. The representation and efficient rendering of such generic material descriptions is still an unsolved issue.

All applications that try to offer such general material concepts require an adaptation logic and a process for generating shaders specialized for a concrete execution environment. This includes adapting to available uniform and per-vertex parameters, supplied textures, the capabilities of the current renderer, and extended functionality of the GPU. As a simple example, consider a material that takes its diffuse coefficient either from a default constant value, a per-object uniform variable, an interpolated per-vertex attribute, or from a supplied texture map, such as:

kd = Color(0.8);

if (exists(diffuseMap) && isTexture(diffuseMap) && exists(texcoords))

{

kd = sample(diffuseMap,texcoords);

}

else if(exists(vertexColor)){

kd = vertexColor;

}

else if(exits(uniformColor)){

kd = uniformColor;

}

Surprisingly, describing such a logic is not possible in current shading languages, because none of them allows introspection, i.e. to query the type and existence of parameters.

Such adaptivity is often implemented in the application through Übershaders (e.g. [McG05]). Obviously, the features of an Übershader are never complete and customizing the logic requires changing the application itself. The same restriction applies to approaches exploiting the metaprogramming facilities of the host-language [MQP], where the adaptation logic is still determined at compile time of the application.

A material can only be portable between applications if the adaptation logic is part of the material description itself. Hence, we need to shift this logic from the application to the material description.

In this paper we present shade.js, a system for specifying adaptive, self-contained, and portable material descriptions. We make the following key contributions:

• We propose a novel shading language that offers introspection of its execution environment as an integral part of the language (Section 4). This enables authors to write materials that adapt to their current execution environment independent of the application.

• shade.js is the first shading language to exploit the polymorphism of a dynamic language. It deliberately refrains from declaring types and other qualifiers. As a result, we achieve generic material descriptions that work for a wider range of input parameters and renderers.

• We present an accompanying compiler that performs the required analysis to specialize the generic material description to a specific execution environment at run time. During this analysis, the compiler infers types (including type checking), semantics, and compute frequencies. Then it optimizes the shader code accordingly.

• We generate code and present results for four different rendering algorithms and three renderers: GLSL shaders for forward and deferred rasterization via WebGL as well as OSL shaders for Whitted style ray tracing and path tracing using Blender’s Cycles render engine.

**2. Our Approach**

In order to achieve more adaptive materials, we propose a new domain-specific language based on a subset of JavaScript. We have chosen JavaScript mainly for its polymorphic features: It is not explicitly typed, an algorithm described in JavaScript can be used with any types that provide the necessary methods used in the algorithm. This allows writing procedures more generically and increases the expressiveness of the material description.

Additionally, the language offers mechanisms to introspect its execution environment. We provide three different kinds of introspection: Authors can query the environment for i) the existence of parameters; ii) the type of parameters; and iii) the availability of optional functions. The JavaScript language provides natural ways to query types and the existence of properties of parameters which we exploit for shade.js. As a result, material authors can adapt the control flow of a material description based on information about the execution environment.

Therefore, shade.js comes with a JIT compiler that performs static code analysis based on the concrete execution environment at run time. At this point in time the compiler can evaluate all queries into the execution environment statically. Depending on the result of the analysis, the compiler changes the control flow of the shader and eliminates dead code. Additionally, the compiler infers all variable types based on the types of the parameters in the execution environment. As a result, the cross-compiled code is adapted and highly specialized to its execution environment, i.e. it has static types and the resulting control flow only depends on the values of parameters, not on their types. If the compiler fails to do so, we can produce meaningful error messages based on the results of the code analysis

**3. Language**

Shade.js supports a subset of JavaScript that includes all arithmetic, logical, and assignment operators, conditional statements, loops, break, and continue statements (without labeling). We support the built-in data types undefined, number, boolean, string, object, and function.

However, we do not support the entire functionality of JavaScript: Array sizes may not change and array elements need to have homogeneous types. We support functions as an abstraction mechanism, but not the JavaScript prototyping functionality. Also, we support only some predefined objects such as vectors with two, three, and four components, and 3× 3 and 4× 4 matrices.

Since all objects in shade.js are immutable, we circumvent dynamic memory allocations for all current target languages which provide vectors and matrices as built-in primitive data types. All these restrictions are not of a conceptual nature but we enforce them for now to simplify the mapping to the target languages.

**3.1. Polymorphism and Introspection**

In shade.js, we exploit the properties of the JavaScript object to query the existence of parameters and optional renderer capabilities and the instanceof and typeof operators to query the type of parameters. Recall the pseudo code from the introduction. With shade.js one can implement this logic in straight-forward JavaScript syntax:

function shade(env){

var kd = new Vec3(0.8);

if(env.texcoords && env.diffuseMap instanceof Texture) {

kd = env.diffuseMap.sample2d(env.texcoords).rgb();

} else if (env.vertexColor) {

kd = env.vertexColor;

} else if (env.uniformColor){

kd = env.uniformColor;

}

...

}

The first parameter of the shade function is an object that provides all parameters of the execution environment as its properties. If an input parameter does not exist, accessing the property returns undefined, which evaluates to false in logical expressions as well as in expressions with the instanceof operator. This mechanism can not only be used to query for the existence of shader parameters but also for the availability of optional functions, accessible through the this keyword:

if(this.fwidth){

//the execution environment supports derivatrives

var fw = this.fwidth(env.texcoord);

...

}

Since shade.js uses implicit types and sources for all input parameters, we can further simplify the logic of the simple shader by replacing diffuseMap, vertexColor, and uniformColor with a single input parameter color:

function shade(nev) {

var kd = new Vec3(0.8);

if (env.color && env.color.sample2d && env.texcoords){

kd = env.color.sample2d(env.texcoords).rgb();

}else if (env.color && env.color.rgb){

kd = env.color.rgb();

}

...

}

In this code snippet, instead of using the instanceof operator, we check for available methods of the color parameter to determine its type: if sample2d is defined, it can be used as a texture and if rgb is available, it can be converted to a color. By checking for available methods instead of explicit types, we further expand the range for supported input parameters. For instance, both Vec3 and Vec4 implement the rgb() method and are therefore candidates for a color. Additionally, since the source of input parameters is implicitly determined, color can be provided as both a uniform parameter or vertex attribute. In cases where the execution environment provides both – a vertex attribute and a uniform parameter with the same name – the more specific per-vertex definition is used.

Another step towards a more adaptable shader is the support of multiple, semantically equivalent input parameters of different names. The following example shader accepts the diffuse color according to the naming conventions of Wavefront OBJ (Kd), COLLADA (diffuse) and XML3D (diffuseColor):

function shade(env){

var kd = env.diffuse || env.diffuseColor || env.kd;

var finalkd = new Vec3(0);

if(kd && kd.sample2d && env.texcoord){

finalkd = kd.sample2d(env.texcoord).rgb();

}else if(kd && kd.rgb){

finalkd = kd.rgb();

}

}

Again we interpret all input parameters either as texture or color, which is required e.g. to properly support the COLLADA convention. Although the same specialization can be achieved using auto-generated macro definitions, converting this 8-line code snippet to an equivalent GLSL shader using preprocessor directives to accept input arguments for the diffuse color with different names and varying types, results in over 40 lines of code (see supplemental material).

Similarly, we can handle input arguments with slightly varying semantics. For example, the following code accepts both transparency (with 1 being fully transparent) and opacity (with 1 being fully opaque):

function shade(env){

var alpha = 1;

if(env.transparencuy != undefined )

alpha = 1 - env.transparency;

else if(env.opactity != undefined)

alpha = env.opacity;

...

}

In shade.js, functions are generic templates that can be used with multiple types. A specialization of the function based on the types of parameters can be done using the introspection mechanisms. This concept replaces function overloading in typed languages.

译文1

作者：Kristian Sons1;2, Felix Klein2, Jan Sutter1 and Philipp Slusallek1;2

国籍：德国

出处：Computer Graphics Forum. 2014, 33(7): 51-60.

**Shade.js: 自适用的材质描述**

图片包含 照片, 展示, 不同

已生成极高可信度的说明

图 1: 这是用shade.js书写的呈现在WebGL中材质的图片集合。分别使用正面投影(左上)，延迟渲染投影(右上)，拟光影追踪(左下)，以及全局照明(右下)四种方式。我们使用四种不同的渲染方式呈现相等的材质。

**摘要：**在计算机图形学中，材质是一个可视化的概念，它是可参数化的，并且应该适用于任意的3D建模和渲染系统。由于提供了参数和属性以及呈现的功能系统有很大的差异，所以材质需要适应其执行环境。在目前的方法中，适配逻辑基于字符串操作，编译器指令或元编程设备被“烤”到呈现应用程序中。然而，为了实现应用独立和独立的材质描述，适应逻辑需要成为材质描述本身的一部分。在这片文章中我们提出shade.js，一种使用动态语言来实现自适用的新颖材质描述方法。着色器可以检查其执行环境，并在运行时适应于可用参数和渲染功能。此外，shade.js利用了非显式声明类型的多态性。这两个新颖的功能允许编写适应性更强的材料描述。根据运行时的具体执行环境，伴随编译器生成特殊类型化和优化渲染目标渲染系统、算法的渲染代码。我们评估shade.js使用示例针对四种不同的呈现方法(正面投影和延迟渲染、光线追踪和全局照明)。上述表明，我们可以提高指定材料的便利性和灵活性，而不会牺牲性能。

1. 简介

在大多数游戏中，专门优化的GPU着色器是与游戏资源、游戏引擎和渲染器紧密地交织在一起。这样的着色器不能单独工作而且也不能在其他场景中使用。然而，大多数其他应用程序，可以很容易地在不同的场景和任意渲染系统中重用更一般材质的概念中受益。这些材料可以交换，改进，验证，组织在可重复使用的库中，并通过互联网共享。要实现这一目标,一种材料需要独立的，独立于3D场景描述和渲染算法。这种通用材料描述的表示和高效渲染仍然是一个未解决的问题。

所有试图提供这种通用材料概念的应用程序都需要适应逻辑和生成专门用于具体执行环境的着色器的过程。这包括适应可用的统一和每个顶点参数，提供的纹理，当前的功能渲染器，以及GPU的扩展功能。作为一个简单的举例来说，考虑一个从默认常量值，每个对象统一变量，每个顶点插值属性或者从一个提供的纹理贴图中获取其漫射系数的材质，例如：

kd = Color(0.8);

if (exists(diffuseMap) && isTexture(diffuseMap) && exists(texcoords))

{

kd = sample(diffuseMap,texcoords);

}

else if(exists(vertexColor)){

kd = vertexColor;

}

else if(exits(uniformColor)){

kd = uniformColor;

}

令人惊讶的是，描述这样的逻辑在当前的着色语言中是不可能的，因为它们都不允许内查，即查询参数的类型和存在。

这种适应性通常通过Ubershaders在应用程序中实现。显然，Ubershader的特性从来都不是完整的和自定义的逻辑需要更改应用程序本身。同样的限制适用于利用主机语言的元编程设施的方法，其中适配逻辑仍然在应用程序的编译时间被确定。一种材料只能在应用程序之间移动适应逻辑是材料描述本身的一部分。因此，我们需要将这个逻辑从应用程序转移到材质描述。在本文中，我们提出shade.js，一个用于指定的系统自适应的，独立的，便携式的材质描述。

我们的主要贡献如下:

* 我们提出了一种新颖的着色语言，它将执行环境作为语言的一个组成部分进行自查。这使作者能够独立于应用程序编写适应当前执行环境的材料。
* shade.js是第一个利用动态语言的多态性的着色语言。它故意没有声明类型和其他限定符。因此，我们实现了针对更广泛的输入参数和渲染器的通用材质描述。
* 我们提供了一个执行所需分析的附带编译器，以便在运行时将通用材料描述专用于特定的执行环境。在此分析过程中，编译器推断类型（包括类型检查），语义和计算频率。然后，它相应地优化着色器代码。
* 我们生成代码并呈现了又四种不同渲染算法和三种渲染器生成的结果: GLSL着色器，用于通过WebGL进行前向和延期光栅化，以及OSL着色器，用于使用Blender Cycles渲染引擎进行Whitted-Style光线追踪和路径追踪。

1. 我们的方法

为了实现更多的自适应材料，我们提出了一种基于JavaScript子集的新的域特定语言。我们之所以选择JavaScript主要是因为它的多态特性：它没有明确的类型，JavaScript中描述的算法可以用于提供算法中使用的必要方法的任何类型。这使得编写程序更通用，并增加了材料描述的表现力。

此外，该语言还提供了对其执行环境进行自查的机制。我们提供三种不同的类型:作者可以查询环境1） 参数的存在；2）参数类型； 3）可选功能的可用性。JavaScript语言提供了查询我们为shade.js开发的参数属性存在和类型的故有方法因此，材料作者可以根据有关执行环境的信息来调整材料描述的控制流程。

因此，shade.js附带了一个JIT编译器，它基于运行时的具体执行环境执行静态代码分析。此时，编译器可以静态评估所有查询到执行环境中。根据分析结果，编译器会更改着色器的控制流程并消除死代码。此外，编译器根据执行环境中参数的类型推断所有变量类型。因此，交叉编译的代码适应并且高度专用于其执行环境，即它具有静态类型，而结果的控制流只取决于参数的值，而不是它们的类型。如果编译器无法做到这一点，我们可以根据代码分析的结果生成有意义的错误消息。

1. 语言

Shade.js支持包含所有算术，逻辑和赋值运算符，条件语句，循环，中断和继续语句（不带标签）的JavaScript子集。我们支持内置的数据类型undefined，数字，布尔值，字符串，对象和函数。

但是，我们不支持JavaScript的全部功能:数组大小可能不会改变，数组元素需要具有同类类型。我们支持作为抽象机制的函数，而不是JavaScript原型功能。此外，我们只支持一些预定义的对象，比如向量，有两个、三个和四个分量以及3×3和4×4矩阵。由于shade.js中的所有对象都是不可变的，因此我们规避了所有当前目标语言的动态内存分配，这些目标语言提供向量和矩阵作为内置原始数据类型。所有这些限制都不是概念性的，但我们现在强制执行，以简化到目标语言的映射。

在shade.js中，我们利用JavaScript对象的属性来查询参数和可选渲染器功能的存在性，以及查询参数类型的instanceof和typeof运算符。回想一下引言中的伪代码。通过shade.js，可以用直接的JavaScript语法实现这个逻辑：

function shade(env){

var kd = new Vec3(0.8);

if(env.texcoords && env.diffuseMap instanceof Texture) {

kd = env.diffuseMap.sample2d(env.texcoords).rgb();

} else if (env.vertexColor) {

kd = env.vertexColor;

} else if (env.uniformColor){

kd = env.uniformColor;

}

...

}

渲染函数的第一个参数是一个对象，它提供执行环境的所有参数作为其属性如果输入参数不存在，则访问该属性将返回未定义的值，该值在逻辑表达式中以及在具有instanceof运算符的表达式中计算为false。这个机制不仅可以用来查询着色器参数的存在，而且可以通过这个关键字来访问可选函数的可用性：

if(this.fwidth){

//the execution environment supports derivatrives

var fw = this.fwidth(env.texcoord);

...

}

由于shade.js使用所有输入参数的隐式类型和来源，我们可以通过用一个输入参数color替换diffuseMap，vertexColor和uniformColor来进一步简化简单着色器的逻辑：

function shade(nev) {

var kd = new Vec3(0.8);

if (env.color && env.color.sample2d && env.texcoords){

kd = env.color.sample2d(env.texcoords).rgb();

}else if (env.color && env.color.rgb){

kd = env.color.rgb();

}

...

}

在这个代码片段中，我们检查color参数的可用方法来确定它的类型，而不是使用instanceof运算符：如果定义了sample2d，则可以将其用作纹理，如果rgb可用，则可以将其转换为颜色。通过检查可用方法而不是明确类型，我们进一步扩展支持的输入参数的范围。例如，Vec3和Vec4都实现rgb()方法，因此是颜色的候选对象。此外，由于输入参数的来源是隐式确定的，颜色可以作为统一参数或顶点属性提供。在执行环境同时提供顶点属性和统一参数的情况下，使用更具体的每个顶点定义。

对于更适应的着色器的另一个步骤是支持多个不同名称的语义等价的输入参数。以下示例着色器根据Wavefront OBJ（Kd），COLLADA（diffuse）和XML3D（diffuseColor）的命名约定接受漫反射颜色：

function shade(env){

var kd = env.diffuse || env.diffuseColor || env.kd;

var finalkd = new Vec3(0);

if(kd && kd.sample2d && env.texcoord){

finalkd = kd.sample2d(env.texcoord).rgb();

}else if(kd && kd.rgb){

finalkd = kd.rgb();

}

...

}

我们再次将所有输入参数解释为纹理或颜色，这是所需的，合理支持COLLADA公约。尽管使用自动生成的宏定义可以实现相同的专业化，但是使用预处理器指令将此8行代码片段转换为等效的GLSL着色器来接受具有不同名称和不同类型的漫反射颜色的输入参数，会导致超过40行代码。同样，我们也可以使用稍微不同的语义来处理输入参数。例如，以下代码接受透明度(1为完全透明)和不透明度(1为完全不透明):

function shade(env){

var alpha = 1;

if(env.transparencuy != undefined )

alpha = 1 - env.transparency;

else if(env.opactity != undefined)

alpha = env.opacity;

...

}

在shade.js中，函数是可以与多种类型一起使用的通用模板。基于参数类型的函数的专门化可以使用自查机制来完成。这个概念取代了键入语言中的函数重载。

原文2

**Fast robust and precise shadow algorithm for WebGL 1.0 platform**

*T. Milet, M. Tóth, J. Peciva, T. Starka, J. Kobrtek and P. Zem ˇ cík ˇ  
Brno University of Technology, Czech Republic*

图片包含 室内, 地板, 红色

已生成极高可信度的说明

**Figure 1：***Three testing scenes - Sponza, Conference Room and Sibenik*

**Abstract：**

This paper presents fast and robust per-sample correct shadows for WebGL platform. The algorithm is based on silhouette shadow volumes and it rivals the standard shadow mapping performance. Our performance is usually superior when compared with high resolution shadow maps. Moreover, it does not suffer from a number of artefacts of shadow mapping and always provides per-pixel correct results. WebGL 1.0 provides just vertex and fragment shaders. Thus, we put all our algorithms evaluating silhouette edges to vertex shaders. Specially precomputed data are fed to the vertex shaders that extrude shadow volume sides just for silhouette edges. Some optimizations are deployed for performance and data size reasons that are important especially on low performance configurations, such as cost- effective tablets and mobile phones. The paper evaluates our solution on number of models. Our solution performs on par with high resolution omnidirectional shadow mapping.

1. **Introduction**

Shadows are an important visual cue for human perception of 3D space, providing human brain with additional information about the structure of the scene that is usually visualized as a 2D image on the computer screen.

Two popular methods for shadow visualizations are used nowadays - shadow volumes [Cro77] and shadow mapping. Shadow mapping got its hardware acceleration in 2001 with GeForce 3 and became popular shadow method in computer games and other areas. However, shadow mapping is prone to visual artefacts [LGMM07]. Tremendous amount of research was done to lessen these artifacts to some extent, for instance [SD02] [WSP04] [ZSXL06] [Ros12]. Another approach is to adjust the scene design in a way that artefacts will not become visible. Such approach is probably largely used in computer game industry but it is not acceptable in CAD or toher systems, where precise visualization of the model is required.

Shadow volumes got their hardware support in 1991 when hardware stencil buffer was introduced [Hei91]. Unfortunately, Heidmann’s algorithm, that became known as z-pass, was not robust when the viewer himself was in shadow. Thus, [CK00] and [BS99] proposed z-fail algorithm. Finally, [EK02] presented robust z-fail algorithm that should provide correct visual results for any arbitrary scene.

As shadow volumes work partially in image space, they are capable to deliver per-pixel precise shadows. However, rasterizing large shadow volume extents made them fill-rate limited in most cases. The extrusion of shadow volumes only from silhouette edges, instead of every edge, became the main performance optimization. The algorithm for finding silhouette edges on manifold meshes was described by [Ber86]. [BS03] was the first to show the silhouette algorithm implemented entirely in graphics hardware. [McG04] described silhouette algorithm implemented in vertex shader only using specially precomputed mesh. [vW06] developed optimized silhouette construction using SSE2 instructions for computer game Doom 3. [SWK07] used then-new geometry shader for silhouette algorithm.

Many silhouette algorithms require 2-manifold triangle meshes to work correctly. More general algorithms were proposed to alleviate this restriction. [AW04] requires the mesh to be orientable only. [KKT08] showed the algorithm that works on arbitrary meshes however it exhibits visual artefacts from time to time due to limited numerical precision of both GPU and CPU computing units. [PSM∗13] made the algorithm robust and verified its artefact-free property on a number of computing platforms. [MKZP14] showed the further performance gains when implemented using tessellation shaders.

1. **Algorithm**

Our solution is based on the robust silhouette algorithm developed by [PSM∗13] and enhanced algorithm by [MKZP14]. However, they present CPU, multi-core CPU, geometry shader, OpenCL and tessellation shader implementations in their papers, while the only GPU computing capabilities available in WebGL 1.0 are vertex and fragment shaders. For our solution, we have chosen vertex shader and took the idea of [McG04] to feed vertex shader by a specially constructed mesh data. These mesh data processed by vertex shaders will extrude shadow volume only on silhouette edges. [McG04] designed solution just for 2-manifold meshes so we had to design a new solution, merging McGuire’s approach with the algorithm presented in [PSM∗13].

译文2

作者：Milet T, Starka T, Kobrtek J.

国籍：捷克共和国

出处：International Conference on Artificial Reality and Telexistence and, Eurographics Symposium on Virtual Environments. Eurographics Association, 2015:85-92.

**对于WebGL的1.0平台的快速稳健和精确的阴影算法**

**摘要：**本文为WebGL平台的每个样本提供了快速且健壮的正确阴影。该算法基于轮廓阴影向量，并与标准阴影映射性能相媲美。与高分辨率阴影图相比，我们的表现通常更好。 而且，它不会受到许多阴影映射的影响，并且总是提供每个像素的正确结果。 WebGL 1.0只提供了顶点和片段着色器。 因此，我们把我们所有的算法评估轮廓边缘到顶点着色器。 特别是预先计算的数据被馈送到顶点着色器，这些顶点着色器仅仅为了轮廓边缘而挤出阴影体积边。 一些优化部署的性能和数据大小的原因，特别是在低性能配置，如经济型平板电脑和手机，尤其重要。本文我们使用了大量的模型类评估我们的解决方案。我们的解决方案与高分辨率全向阴影映射相媲美。

1. 简介

阴影是人类三维空间感知的重要视觉提示，为人脑提供关于场景结构的额外信息，这些信息通常在计算机屏幕上可视化为二维图像。

现在使用两种流行的阴影可视化方法 - 阴影向量和阴影映射。阴影映射在2001年用GeForce 3得到了硬件加速，并在电脑游戏等领域成为流行的阴影可视化方法。 然而，阴影映射很容易出现视觉伪影。人们进行了大量的研究以在一定程度上减轻这些伪影，例如[SD02] [WSP04] [ZSXL06] [Ros12]。 另一种方法是调整场景设计，使得伪影变得不可见。 这种方法可能在很大程度上用于计算机游戏行业，但在CAD或其他需要模型的精确可视化的系统中是不可接受的。

在1991年推出硬件模板缓冲器时，阴影向量得到了他们的硬件支持。 不幸的是，当观众自己处于阴影中时，海德曼的算法，即z-pass被认为是不健壮的。 因此，[CK00]和[BS99]提出了z-fail算法。 最后，[EK02]提出了强大的z-fail算法，可以为任意场景提供正确的视觉效果。

当阴影向量在图像空间工作时，它能够提供每像素精确的阴影。 但是，在大多数情况下，光栅化较大的阴影向量范围会使其填充率受限。阴影体积仅从轮廓边缘而不是每个边缘挤出成为主要的性能优化。 [Ber86]描述了在流形网格上寻找轮廓边的算法。 [BS03]是第一个显示完全在图形硬件中实现的轮廓算法。 [McG04]描述了在顶点着色器中使用专门预先计算好的网格的轮廓算法。 [vW06]使用SSE2指令为电脑游戏Doom 3开发了优化的轮廓构造。[SWK07]使用了新的几何着色器来进行轮廓算法。

许多轮廓算法需要2-manifold三角网格才能正常工作.更通用的算法被提出来减轻这个限制。 [AW04]要求网格只能定向。 [KKT08]展示了可以在任意网格上工作的算法，但由于GPU和CPU计算单元的数值精度有限，因此它会时不时会出现视觉伪影。 [PSM \* 13]使得该算法更加健壮，并在多个计算平台上验证了其伪影的可消除性。 当使用曲面细分着色器实现时，[MKZP14]显示了进一步的性能提升。

1. 算法

我们的解决方案是基于[PSM \* 13]开发的健壮轮廓算法和[MKZP14]增强算法。 然而，他们在论文中提供了CPU，多核CPU，几何着色器，OpenCL和镶嵌着色器实现，而WebGL 1.0中唯一可用的GPU计算功能是顶点着色器和片段着色器。 对于我们的解决方案，我们选择了顶点着色器，并采用了[McG04]的思想，通过专门构建的网格数据来提供顶点着色器。 这些由顶点着色器处理的网格数据将仅在轮廓边上挤出阴影体积。 [McG04]设计的解决方案只适用于2-manifold网格，所以我们不得不设计一个新的解决方案，将McGuire的方法与[PSM \* 13]中介绍的算法合并。