Pacific Graphics 2014
J. Keyser, Y. J. Kim, and P. Wonka
(Guest Editors)

Volume 33 (2014), Number 7

shade.js: Adaptive Material Descriptions

Kristian Sons[1,2], Felix Klein[2], Jan Sutter[1] and Philipp Slusallek[1,2]

[1]German Research Center for Artificial Intelligence, Germany
[2]Intel VCI & Saarland University, Germany

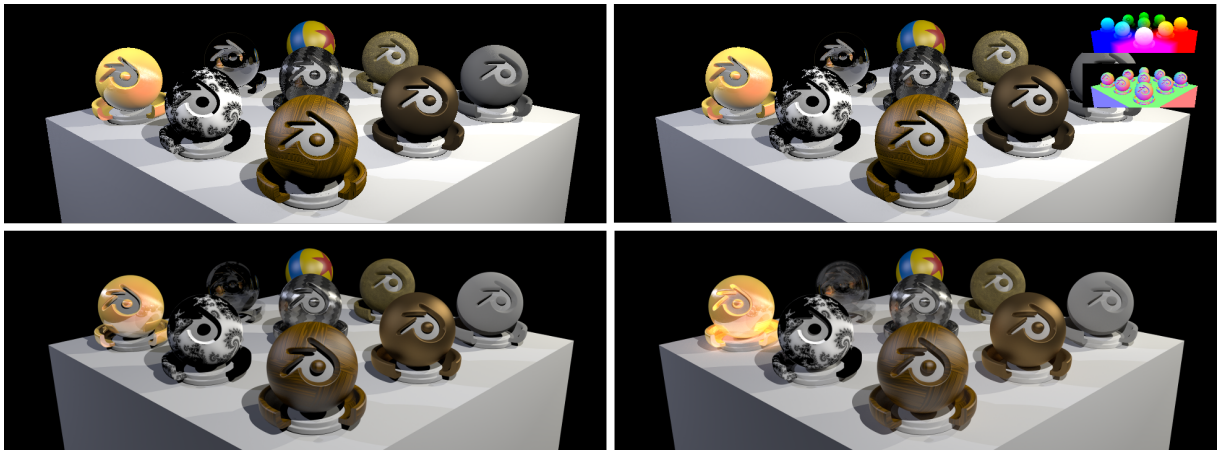

Figure 1: *A collection of procedural materials written in shade.js and rendered in WebGL using forward shading (upper left), deferred shading (upper right), Blender Cycles ray-traced (lower left), and with global illumination (lower right). We achieve conceptually equal materials for all four rendering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

Categories and Subject Descriptors (according to ACM CCS): Computer Graphics [I.3.7]: Methodology and Techniques—Languages; Computer Graphics [I.3.7]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; Programming Languages [D.3.2]: Processors—Compilers;

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 (exists(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 (Section 5). 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 (Section 5.5) as well as OSL shaders for Whitted-style ray tracing and path tracing using Blender's Cycles render engine (Section 5.5).

2. Related Work

2.1. Specialized Shaders

Common shading languages include the RenderMan Shading Language (RSL) [HL90, Ups89] and the Open Shading Language (OSL) [GSKC10]. Both, RSL and OSL are fully procedural languages inspired by C with additional domain-specific data types, e.g. for colors and vectors. Both languages provide means to define values for parameters that are not available in the execution environment (default values) and ad hoc polymorphism using function overloading. However, all parameters and variables are strictly typed and thus fixed at authoring time. Moreover, there is no possibility to alter the control flow based on the type or existence of parameters. Thus the adaptivity of these shaders to new scenes and renderers remains limited.

GPU languages such as Cg [MGAK03], HLSL [Mic02], and GLSL [KBR03] are highly specialized. They are specifically designed to run efficiently on the programmable pipelines of recent graphics cards. Similar to RSL and OSL, these languages are "little languages" inspired by C. They come with a static type system and provide no means to introspect their execution environment. Their polymorphism is restricted to operator overloading and default values. Even worse, authors have to declare the sources of parameters (uniform or per-vertex) at authoring time using type qualifiers.

Preprocessor directives are a common way to change GPU shader programs to compile in different execution environments. Some extensions in GLSL implicitly define macros for the preprocessor, making it possible to adapt the shader code based on optional hardware capabilities. However, all other code changes performed by the preprocessor need to

be controlled by the application, which implements the necessary adaptation logic.

HLSL Shader Model 5 adds *Dynamic Linking* [Mic08] as a more sophisticated approach to shader permutations. It allow for referencing abstract interfaces in the shader at compile time. The application can then choose one of the concrete implementations (compiled into the same program) at run-time. Thus, the adaptation logic remains in the application.

2.2. Meta-Systems

A number of approaches have been proposed to solve the issues present in specialized shading languages. In general, these describe meta-systems that create code for one or more specialized languages in the end.

McCool et al. [MQP] propose shader metaprogramming within the host language. The proposed API allows for using generic types at authoring time by exploiting C++ templates. The types are checked and determined at compile time. Additionally, the API offers specialization during run time, e.g. branching based on a query for the type of a parameter. However, this approach has some disadvantages: The C++ polymorphism is only available for those parameters that can be determined during compile time of the application. Also the adaptation logic is determined already at application compile time, resulting in predefined specialization similar to Übershaders. Additionally, shader metaprogramming offers no means to query the existence of parameters in the execution environment.

Similarly, Kuck et al. [KW09] specialize shaders at run time using C++ classes as an abstraction layer. Again, the adaptation logic is fixed at compile time of the application and cannot be altered by the shader author.

Vertigo [Ell04] is a metaprogramming system using the functional language Haskell as host language. It supports shading and expression rewriting for optimizations, but offers no adaptivity based on the execution environment.

Declarative approaches such as Shade Trees [Coo84, AW90, MSPK] do not expose the program's control flow to the user. As a result, the author cannot adapt the control flow based on the current execution environment. For instance, it is not possible to branch a shading program based on the existence of a specific parameter.

More recent approaches mix declarative and imperative elements: Material Definition Language (MDL) [Nvi12], supports the customization of certain aspects procedurally, MetaSL [Men10] offers authoring of procedural building blocks with arbitrary input and output parameters. Again, the procedural parts are inspired by the C language with limited polymorphism due to fixed types.

In GPU languages, logically related computations are spread over multiple stages in the rendering pipeline. RTSL [PMTH01], Renaissance [Aus05] and Spark [FH11] tackle this issue using compiler technology to partition one shader description to the stages of a multi-stage pipeline. In shade.js, we use a similar approach to partition computation from a single material description to the CPU, vertex shader, and fragment shader stage. However, the previous approaches require typed input parameters and provide no mechanism to introspect their execution environment.

ShaderDSL.js [Ado12] is a compiler from a subset of JavaScript to GLSL. In contrast to shade.js, it requires declaring the input parameters explicitly (including types) and has no interface to the lighting system. As a result, ShaderDSL.js is mainly a syntax transformation.

Other systems address shader partitioning [CNS*02, RLV*04] and shader simplification [Pel05, SAMWL11]. These techniques are related but orthogonal to *shade.js* and could eventually be included.

A type of inverse approach to adaptation is described in [LS02], where the geometric data is specialized by a compiler based on the knowledge of the bound shading program.

3. 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.

On the practical side, we integrated shade.js into XML3D [SKR*10], an open source JavaScript render engine based on WebGL. Thus we are able to describe the material in the host language. Moreover, shade.js can easily be ported to other WebGL engines and – with little extra effort – also to native renderers. Since a shade.js shader is written in legal JavaScript, it would even be possible to run it in a dynamically interpreted environment. Although this feature is interesting for instance for debugging, we aim to generate specialized shader code for common renderers and platforms.

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.

In addition to these novel concepts, we cherry-picked the most useful concepts from existing approaches and integrated them into shade.js. For portability across different lighting systems we adopt interfaces based on radiance closures similar to OSL [GSKC10]. The compiler resolves the radiance closures and generates appropriate code for the target rendering system. Similar to RTSL [PMTH01] and Renaissance [Aus05], our compiler partitions the shader program by computation frequencies in order to optimize the run time performance, i.e. it extracts parts of the code that can be executed on the CPU or on the vertex shader stage respectively. Similar to AST [MSPK], we have a semantically rich type system that has additional type properties such as computation frequency (constant, uniform, or per-vertex), interpretation (e.g. color, normal, point), and the coordinate space (e.g. object space, world space etc.) of a vector. In contrast to previous approaches, we deliberately refrain the authors from declaring these properties. Instead, the compiler automatically derives them from the execution environment and interfaces used in the shader code.

4. 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.

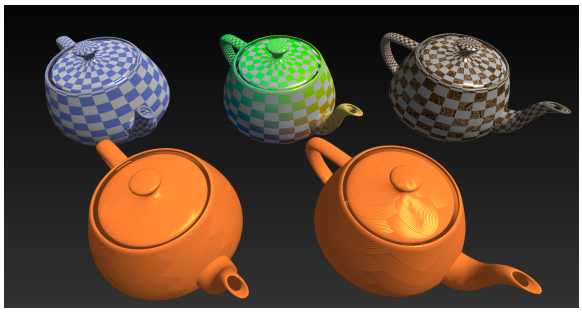

Figure 2: *Example of adaptable shaders using polymorphism and introspection. The checkerboard shader used in the rear works for a uniform color (left), vertex colors (middle), and a texture (right). The orange teapots in the front feature normal mapping, using the tangent vertex attribute if available (left) and otherwise an approximated tangent based on the derivative of the vertex position (right).*

4.1. 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 derivatives
  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(env) {
  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.transparency != undefined)
    alpha = 1 - env.transparency;
  else if(env.opacity != 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.

See Figure 2 for a number of adaptable shaders based on shade.js. For more complex materials refer to the supplemental material.

4.2. Radiance Closures

Similar to OSL, we provide a set of radiance closures to interface with the lighting system. A radiance closure enables the evaluation of interactions between lights and the material surface without providing an explicit viewing direction. Radiance closures are parameterizable. The return value of the *shade* procedure is either a radiance closure, a linear combination of radiance closures, or *undefined*. The latter discards the current fragment.

We implemented a set of radiance closures for shade.js, including the Oren-Nayar, Phong, Cook Torrance, and Ward reflection models, mirror-like reflection, and others. A checkered material that discards half of the squares and the resulting shader applied to a cube and ray-traced using the Cycles renderer is shown in Figure 3.

```
function shade(env) {
  var smod = (env.texcoord.x() * env.frequency) % 1.0,
      tmod = (env.texcoord.y() * env.frequency) % 1.0;

  if((smod < 0.5 && tmod < 0.5) ||
     (smod >= 0.5 && tmod >= 0.5)) {
    return; // Discards this fragment
  }

  return new Shade().diffuse(env.kd, env.normal)
                    .phong(env.ks, env.normal,
                           env.shininess);
}
```

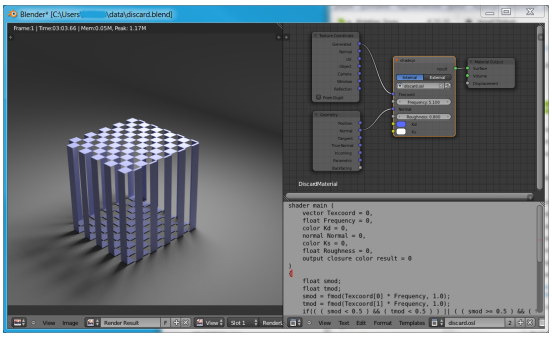

Figure 3: *A checkered material in shade.js compiled to OSL and rendered with Cycles in Blender.*

5. Compiler & Architecture

As a key concept of shader.js, the compiler specializes the material description at a late stage when the implementation and the bound geometry have been determined. To achieve

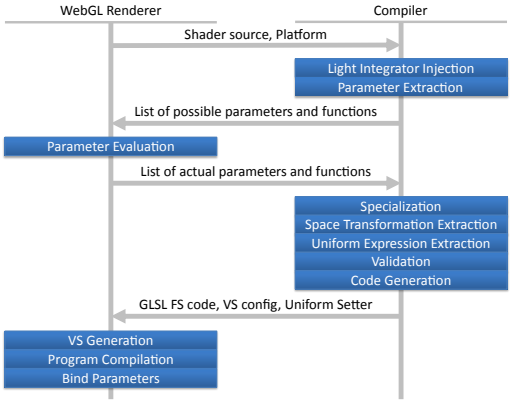

Figure 4: *Two-step Communication between the shade.js compiler and a WebGL based renderer.*

this, we have chosen a two-step communication process between renderer and compiler, which is shown for WebGL in Figure 5.

After passing the material description to the compiler, the compiler injects the light integrators for all used radiance closures. We integrate the light integrators in this early step in order to exploit the specialization also for the lighting part. Then, the compiler determines all *potentially* used shader parameters and optional functions in the code.

In the second step, the renderer examines the available subset of the potentially shader parameters and annotates their type and source. It also annotates, which of the optional functions are available, e.g. based on available extensions. Note that the two-way approach allows the renderer to take further actions to make shader parameters available, e.g. requesting tangents from the server if it could be used by the material.

With the annotations available, the compiler

- performs type inference and dead code elimination based on the input types in order to specialize the shader by removing polymorphism and resolving introspection,
- computes all type qualifiers that are required for the target language. This includes the source (e.g. uniform or vertex attribute) and the semantic (e.g. vector, color or normal) of parameters,
- validates the code to ensure that no references to unavailable input parameters or functions remain,
- performs optimizations usually done manually by the shader author, and
- generates code for the target shading language.

Back on the renderer side, we use the information returned by the compiler to properly integrate the shader code. In case of a WebGL-based forward renderer, we generate a matching vertex shader based on a configuration that defines the coordinate spaces for each used vertex attribute. The renderer creates the shader program and uses a collection of *Uniform Setters* to bind parameters and evaluate extracted uniform expressions on the CPU (see Section 5.4).

5.1. Specialization

As part of the compiler-based specialization, we have to derive the concrete types of all used variables. This is necessary because the target languages require explicit types.

JavaScript provides the built-in data types undefined, number, boolean, string, object, and function. The compiler can infer all types based on the literals in the program and the type information of the parameters received from the renderer. In the current system, we do not support dynamic types and report an error if they occur. Without dynamic types, the type inference is sound and no ambiguities occur.

In JavaScript, the number type does not distinguish between integers and doubles. Our type inference tries to represent numbers as integers wherever possible. This is a common approach in commercial JavaScript compilers [HG12] and necessary, for instance, for loops and dynamic array access in GLSL and OSL.

Introspection is only available in shade.js and must not appear in the target languages. In contrast to GPU shader compilers, which may perform constant propagation and dead code elimination up to an arbitrary extent in order to optimize the performance of a shader, a thorough analysis is essential in shade.js to guarantee all introspection features (including branches that depend on the result of the introspection) are removed. See Figure 5 for an example.

It is not sufficient to only evaluate constant expressions (constant folding). The compiler has to perform an analysis to substitute variables by a constant value if the variable holds this constant whenever execution reaches a pro-

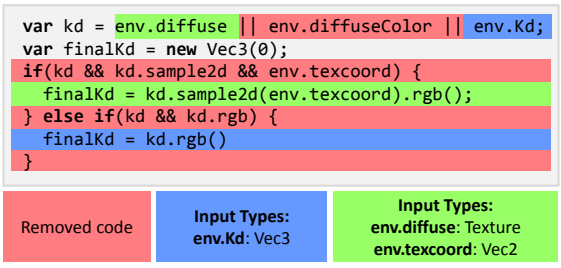

Figure 5: *Example on how dead code elimination is applied to remove the use of introspection features. All code marked red is removed regardless of execution environment. The blueish and greenish marked code represent two versions of the resulting shader depending on passed input parameters (described at the bottom).*

gram point (constant propagation). We eliminate unreachable branches in the same analysis. Since we allow loops and conditional branching, we have to formulate all our program analyses as monotone data flow frameworks [NNH99].

Note that the result of the specialization does not depend on the values of input parameters, but only on types and computation frequency. Thus, setting uniforms and attributes does not invoke recompilation.

5.2. Error Reporting

JavaScript reports basic syntax errors immediately but type related errors only at run time. In addition the error reporting of JavaScript is very relaxed. For instance, it tolerates the use of undefined variables as long as no property is accessed. This becomes especially confusing when undefined variables are used in arithmetic expressions: this produces *NaN* (Not a number) values that will be propagated throughout the code.

Despite being a valid subset of JavaScript, shade.js has stricter error checking. Once the type inference and dead code elimination have been performed, the types of all variables need to be determined. Consequently, the use of undefined values or invalid types is not supported and will be reported immediately, prior to execution. These errors usually relate directly to missing or incorrect input arguments, thus the compiler can create descriptive error messages, e.g.:

```
NotANumberError: env.roughness is undefined: env.roughness
    * 2 (Line 4)
```

5.3. Semantics

The built-in vector data types of shade.js come without semantics such as *normal* or *color* and we omit a mechanism to annotate these semantics. However, some shading languages (e.g. OSL) define colors and normals as basic data types that do not cast to each other implicitly. Also, knowing the semantics of input parameters is useful, for instance, to populate user interfaces in the application.

We annotate the arguments of the predefined radiance closures with their semantics. Based on this information, the compiler can derive the semantics of the input parameters by doing a data flow analysis along the reversed control flow graph. The analysis supports generic vectors, colors, and normals. Since a parameter could be used as color *and* normal (e.g. for debugging) the analysis detects this ambiguity and delivers both semantics as the result.

5.4. Optimizations

Uniform Expressions The idea of extracting uniform expressions to compute them only per-geometry patch was first introduced in [HL90]. In shade.js, the compiler does not only identify single uniform expressions in the shader program, but propagates these expressions to find more complex uniform expressions within the control flow. To avoid the extraction of expressions that are essentially free on graphics hardware (e.g. swizzling of components) we additionally introduced a cost estimation. The same estimation can be used to extract only the most expensive expressions when we are at risk of exceeding the maximum number of uniform values supported by the hardware.

A uniform expression gets replaced by a new uniform parameter whose value is computed by the application. In case of the WebGL-based renderer that is written in JavaScript, the extracted expression can be evaluated in the application as is. If the compiler extracts interdependent expressions it provides a method to set the original uniforms, transparently handling the update of all generated uniforms.

Coordinate Space Transformations Similarly, expressions that depend on vertex attributes only can be moved to the vertex shader stage. Additionally, linear operations can be factored out to the vertex shader. A major use case for this is the transformation of vertex attributes into different coordinate spaces. Our type system stores information on requested coordinate spaces for vectors and points. Some parameters of the radiance closures (e.g. normals) are requested in a predefined space; which space is renderer dependent. Additionally, the material author can request transformations explicitly via provided functions, e.g.:

```
var P = Space.transformPosition(Space.VIEW, env.pos);
var N = Space.transformDirection(Space.WORLD, env.normal);
```

Based on this information we can propagate requested spaces along the reversed control flow. If a non-linear operation takes place, the compiler injects the transformation into the fragment shader. If only linear operations get applied up to the entry point of the shader, it is safe to move the transformation into the vertex shader. This is sufficient, for instance, to shift the coordinate space transformation through a regular normal mapping algorithm.

5.5. Code Generation

GLSL Due to the design of the shade.js data types, the code generation for GLSL is fairly straight forward. However, we have to explicitly add the code for lighting calculations. This code is not part of the material description but comes from the renderer and is written in JavaScript as well. It is integrated at the beginning of the compilation process, thus the specialization and optimizations are not only applied to the material description provided by the user but also to the renderer-specific light calculations.

Each exit point of the shade routine that results in a (linear combination of) radiance closure is replaced with a function call that internally iterates over all available light sources. This function is specifically generated to reverse the looping: Each light is only evaluated once before the light material interaction described by the radiance closure is applied.

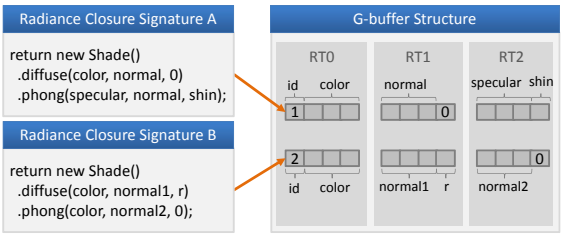

Figure 6: *Example on how different combinations of radiance closure invocations are stored inside the G-buffers. An id is assigned for each unique combination of radiance closure invocations and used in the light pass to correctly interpret the parameters.*

From a shade.js material description we can generate code for a forward renderer as well as for a deferred renderer. As a result, switching between and combining both rendering techniques is simple and does not require having multiple shader versions for the same material. This is very useful, for instance, if one wants to solve the limitations of deferred shading with respect to semi-transparent objects by rendering those objects using forward shading.

For the deferred shading pipeline the compiler generates two types of shaders: An object shader that writes the parameters into the geometry buffer (G-buffer), and a light pass shader that reads the properties of the G-buffer and performs the lighting. The object shader includes all code of the original shade.js program, but instead of invoking the radiance closures, it writes their parameters into the G-buffers. To handle different combinations of radiance closures (which we call radiance closure signatures) across all object shaders using one global G-buffers, the compiler assigns a unique id to each radiance closure signature which is stored in the G-buffer in addition to all parameters (see Figure 6). This id is then read by the light pass shader in order to properly extract all parameters and perform the lighting with the selected radiance closures.

Since we know the semantic of the radiance closure's arguments, we can easily apply compression techniques to the arguments, e.g. quantize normals, in order to automatically minimize the number of required G-buffers. The light pass shader is generated from the list of all used radiance closures and reuses the light integrators of the forward shading algorithm to implement the lighting.

Open Shading Language Our third compiler back-end generates Open Shading Language code from the shade.js material. We can omit the step to inject code for the lighting, because in OSL the light integration is not implemented in the shader but in the renderer. We map the shade.js radiance closures to the closures defined for the Cycles renderer provided by Blender. In addition to the common specialization we have to compute the semantics of the generic vectors, because OSL requires different types for colors and normals.

6. System Experience

The compiler framework itself is written entirely in JavaScript as well. This allows the easy integration of shade.js into the WebGL-based rendering system XML3D, which in turn can run in any WebGL-capable browser on many platforms including smart phones and tablet computers. Additionally, we implemented a HTML5 shader editor that allows for interactive authoring of shade.js shaders (Figure 7). Outside the browser, we can use the compiler framework to generate OSL shaders via the command line.

Figure 1 (the teaser image) shows a scene we created to evaluate and demonstrate the capabilities of shade.js. The scene consists of 9 non-trivial shaders, ported from RSL, OSL, and other sources. This includes shaders with procedural textures, fractals, layered BRDFs, reflections, and normal mapping. We generate shaders for forward and deferred rasterization in WebGL, as well as OSL shaders for ray-tracing and global illumination in Blender's Cycle renderer. As the figure shows, we achieve conceptually equal materials for all four rendering techniques. The resulting images differ only with respect to the incoming light and differences in the renderers.

We measured the performance of shade.js in terms of the generated GLSL shaders and the impact of optimizations. To do this, we translated nine GLSL shaders to shade.js and compared the results. Since we designed our data types to map very well to GLSL data types, there was no significant difference between the original shaders and the shaders generated by our compiler. For instance, translating the "Beating Circles" shader from ShaderToy (`www.shadertoy.com/view/4d23Ww`), results in 65 instructions versus 64 instructions for the original version.

In addition, we translated the predefined Phong material

Figure 7: *Screenshot of an interactive editor for shade.js running in the browser with immediate feedback and review of the generated GLSL shaders.*

	Tablet fps			Laptop fps			Desktop fps			Instructions		
Uniform Extraction	off	on	Δ	off	on	Δ	off	on	Δ	off	on	Δ
Glowing dots shader	6.9	17.0	2.4x	27.4	53.2	1.9x	222	366	1.6x	83	24	29%
Beating Circles Shader	15.6	19.7	1.3x	37.1	48.5	1.3x	270	359	1.3x	37	31	84%
BPM shader	12.4	20.7	1.7x	34.2	56.9	1.7x	271	371	1.4x	69	27	39%

Table 1: *Performance gain achieved by uniform expression extraction. We tested uniform extraction on a Tablet (Qualcomm Adreno 330), a Laptop (Nvidia NVS 5400M), and a Desktop computer (Nvidia GeForce GTX 660 Ti). All shaders were applied in a WebGL-based rendering system, running in Google Chrome in Full HD resolution. In order to increase the shader workload and reduce other overhead, each shader was drawn 20 times over the whole screen. All 3 example shader include procedural animations based on a uniform time value. The* Glowing dots *shader was created by us, while* Beating Circles *and* BMP *are shaders from* `shadertoy.com` *translated to shade.js. The extracted uniform expressions were executed exactly once per frame in JavaScript, which had no measurable impact on the performance.*

from the XML3D render library [SKR*10]. This shader is an Übershader that is specialized by the library using preprocessor macros. Using the same execution context, our compiler generates GLSL code that results in 52 instructions whereas the specialized shader of XML3D results in 58 instructions with negligible differences in rendering performance.

In cases where the compiler can extract uniform expressions out of the fragment shader we are able to measure significant speed-ups in rendering performance. Table 6 shows that the performance impact is especially apparent for platforms with less powerful GPUs such as laptops and tablets.

The interface to the lighting system based on radiance closures is less flexible when compared to approaches such as RSL [HL90], which allow for actively shooting rays and for implementing custom shading models. On the other hand, the higher abstraction level of radiance closures gives the renderer more freedom of choice in the approximations and light integration algorithms to solve the rendering equation, e.g. importance sampling. In particular for GPU-based renderers with their limited access to scene information, we found it useful to be able to hide all the necessary "tricks" behind the predefined shading models. For instance, it is possible to apply screen-space ambient occlusion (SSAO) as part of the *diffuse* radiance closure. For the *reflection* closure, the renderer can generate dynamic reflection maps. The computational effort and the techniques used to approximate these effects do not require changes in the material description. Even more importantly, such approximations can be turned on and off, either by user request or as a result of insufficient hardware or software capabilities, again without rewriting existing shaders. Overall, the radiance closure based interface greatly increases the portability of our material shaders.

6.1. Limitations

When creating adaptive shaders, there is a natural trade-off that type errors can only be determined once all input types are known. In contrast, specialized languages with static types and fixed input signature allow type errors to be detected independent of run time input values. However, if an application specializes shaders e.g. via preprocessors, it runs into the same issues. One approach is to generate and validate all possible permutations in a preprocess (which would be possible for shade.js shaders as well). Otherwise, the correctness of a permutation is only guaranteed at run time and it is left to the application to analyze and report errors that might come with a specific input signature.

Specialization in general can result in a combinatorial increase in the number of generated shaders. shade.js generates one shader per unique signature given by name, type and source of input parameters. Using a signature cache, the system keeps the number of compilation passes to a minimum. However, many different signatures can potentially increase the number of generated shaders. It would be possible to specify parameters to be excluded from the specialization. In that case the compiler could introduce an additional uniform variable that is automatically set within the shader's setter method and which is dynamically tested in the shader.

7. Discussion and Future Work

In this paper we presented shade.js, a system for writing adaptive material descriptions that are portable between different scenes, hardware architectures, and rendering approaches.

A key element of shade.js is its ability to describe all aspects of materials in a single place and the ability to automatically adapt and specialize the shader to the specific execution environment being used.

Directly porting a significant number of non-trivial shaders showed that shade.js can achieve essentially the same performance despite the much higher level of abstraction. Exploiting some of the high-level features of shade.js allows for improving the performance even further via optimizations made possible by code analyses and transformations in the shade.js compiler. The provided optimizations proved particularly valuable for mobile devices.

Any implicitly typed programming language would be

suitable to implement the shade.js concepts. However, using JavaScript for shading, compilation, and as the host language, allows it to be used directly in web-based 3D applications. And though JavaScript's syntax may seem unfamiliar to professional GPU programmers, it is the lingua franca of the web and hence known by all web developers.

The approach presented here can be applied with only minor changes to programmable light descriptions, portable specifications of atmospheric and volume effects, image-space effects like post-processing and tone-mapping, and others.

Radiance closures provide a good abstraction and decouple the material description from the system's lighting capabilities. Other system capabilities on the other hand are tested and used without further abstraction. For future work, it would be favorable to find suitable abstractions for other system capabilities as well.

Currently, the set of available radiance closures is fixed by the renderer. In future work we plan to explore ways to also specify the closures in a portable way, in particular with respect for sample-based BRDFs, e.g. BTFs from measured materials.

Acknowledgements

We would like to thank Stefan Herhut, Sebastian Hack and the anonymous reviewers for their valuable comments and suggestions. The research presented in this paper has been generously supported by the Intel Visual Computing Institute and the EU projects VERVE, FI-CONTENT, and Dreamspace.

References

[Ado12] ADOBE: ShaderDSL.js, 2012. https://github.com/adobe-webplatform/shaderdsl/. 3

[Aus05] AUSTIN C. A.: *Renaissance: A Functional Shading Language*. Master's thesis, Iowa State University, 2005. URL: http://chadaustin.me/hci_portfolio/thesis.pdf. 3, 4

[AW90] ABRAM G. D., WHITTED T.: Building block shaders. *SIGGRAPH Comput. Graph. 24*, 4 (Sept. 1990), 283–288. 3

[CNS*02] CHAN E., NG R., SEN P., PROUDFOOT K., HANRAHAN P.: Efficient Partitioning of Fragment Shaders for Multipass Rendering on Programmable Graphics Hardware. In *Proceedings of HWWS* (2002), Eurographics, pp. 69–78. 3

[Coo84] COOK R. L.: Shade Trees. *SIGGRAPH Comput. Graph. 18* (1984), 223–231. 3

[Ell04] ELLIOTT C.: Programming graphics processors functionally. In *Proceedings of the 2004 Haskell Workshop* (2004), ACM Press. URL: http://conal.net/papers/Vertigo/. 3

[FH11] FOLEY T., HANRAHAN P.: Spark: Modular, Composable Shaders for Graphics Hardware. *ACM Trans. Graph. 30* (2011), 107:1–107:12. 3

[GSKC10] GRITZ L., STEIN C., KULLA C., CONTY A.: Open Shading Language. In *ACM SIGGRAPH 2010 Talks* (2010), SIGGRAPH '10, ACM, pp. 33:1–33:1. 2, 4

[HG12] HACKETT B., GUO S.-Y.: Fast and Precise Hybrid Type Inference for JavaScript. In *Proceedings of PLDI 2012* (2012), ACM, pp. 239–250. 6

[HL90] HANRAHAN P., LAWSON J.: A Language for Shading and Lighting Calculations. *SIGGRAPH Comput. Graph. 24* (1990), 289–298. 2, 7, 9

[KBR03] KESSENICH J., BALDWIN D., ROST R.: The OpenGL® Shading Language, 2003. http://www.opengl.org/documentation/glsl/. 2

[KW09] KUCK R., WESCHE G.: A Framework for Object-Oriented Shader Design. In *Proceedings of ISVC 2009* (2009), Springer-Verlag, pp. 1019–1030. 3

[LS02] LALONDE P., SCHENK E.: Shader-driven compilation of rendering assets. *ACM Trans. Graph. 21*, 3 (2002), 713–720. 3

[McG05] MCGUIRE M.: *The SuperShader*. 2005, ch. 8.1, pp. 485–498. URL: http://www.cs.brown.edu/research/graphics/games/SuperShader/index.html. 2

[Men10] MENTAL IMAGES: Design Specification for MetaSL® 1.1, 2010. http://www.nvidia-arc.com/fileadmin/user_upload/PDF/MetaSL_spec_1.1.6.pdf. 3

[MGAK03] MARK W. R., GLANVILLE R. S., AKELEY K., KILGARD M. J.: Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Trans. Graph. 22* (2003), 896–907. 2

[Mic02] MICROSOFT: DirectX HLSL Shader Model 1, 2002. http://msdn.microsoft.com/. 2

[Mic08] MICROSOFT: DirectX HLSL Shader Model 5, 2008. http://msdn.microsoft.com/. 3

[MQP] MCCOOL M. D., QIN Z., POPA T. S.: Shader Metaprogramming. In *Proceedings of HWWS*. 2, 3

[MSPK] MCGUIRE M., STATHIS G., PFISTER H., KRISHNAMURTHI S.: Abstract shade trees. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. 3, 4

[NNH99] NIELSON F., NIELSON H. R., HANKIN C.: *Principles of Program Analysis*. Springer, 1999. 7

[Nvi12] NVIDIA: Nvidia Material Definition Language, 2012. http://www.nvidia-arc.com/products/iray/mdl.html. 3

[Pel05] PELLACINI F.: User-configurable Automatic Shader Simplification. *ACM Trans. Graph. 24* (2005), 445–452. 3

[PMTH01] PROUDFOOT K., MARK W. R., TZVETKOV S., HANRAHAN P.: A Real-time Procedural Shading System for Programmable Graphics Hardware. In *Proceedings of SIGGRAPH 2001* (2001), ACM, pp. 159–170. 3, 4

[RLV*04] RIFFEL A., LEFOHN A. E., VIDIMCE K., LEONE M., OWENS J. D.: Mio: Fast Multipass Partitioning via Priority-based Instruction Scheduling. In *Proceedings of HWWS* (2004), ACM, pp. 35–44. 3

[SAMWL11] SITTHI-AMORN P., MODLY N., WEIMER W., LAWRENCE J.: Genetic Programming for Shader Simplification. *ACM Trans. Graph. 30* (2011), 152:1–152:12. 3

[SKR*10] SONS K., KLEIN F., RUBINSTEIN D., BYELOZYOROV S., SLUSALLEK P.: XML3D: Interactive 3D Graphics for the Web. In *Proceedings of Web3D 2010* (2010), ACM, pp. 175–184. 3, 9

[Ups89] UPSTILL S.: *RenderMan Companion: A Programmer's Guide to Realistic Computer Graphics*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989. 2