High Performance Graphics (2010)
M. Doggett, S. Laine, and W. Hunt (Editors)

# AnySL: Efficient and Portable Shading for Ray Tracing

Ralf Karrenberg*[1] Dmitri Rubinstein*[1,2] Philipp Slusallek*[1,2,3] Sebastian Hack*[1]

[1]Saarland University
[2]DFKI Saarbrücken
[3]Intel Visual Computing Institute Saarbrücken

**Abstract**

*While a number of different shading languages have been developed, their efficient integration into an existing renderer is notoriously difficult, often boiling down to implementing an entire compiler toolchain for each language. Furthermore, no shading language is broadly supported across the variety of rendering systems.*

*AnySL attacks this issue from multiple directions: We compile shaders from different languages into a common, portable representation, which uses* subroutine threaded code: *Every language operator is translated to a function call. Thus, the compiled shader is* generic *with respect to the used types and operators.*

*The key component of our system is an embedded compiler that instantiates this generic code in terms of the renderer's native types and operations. It allows for flexible code transformations to match the internal structure of the renderer and eliminates all overhead due to the subroutine threaded code.*

*For SIMD architectures we automatically perform vectorization of scalar shaders which speeds up rendering by a factor of 3.9 on average on SSE. The results are highly optimized, parallel shaders that operate directly on the internal data structures of a renderer. We show that both traditional shading languages such as RenderMan, but also C/C++-based shading languages, can be fully supported and deliver high performance across different CPU renderers.*

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Raytracing, Shading D.3.4 [Programming Languages]: Processors—Compilers, Code Generation

## 1. Introduction

Rendering systems use programmable *shaders* to provide flexibility for certain aspects of the rendering process, such as the appearance of surfaces, the emission of light sources, the displacement of geometry, and others. Shaders are functions typically written in a domain-specific language (often C-like with rendering specific features, e.g. RenderMan [AG00], HLSL [PM03], GLSL [RKL04], or others) which are called by the renderer to compute specific values given the current rendering state.

Shaders are similar to plugins in other applications but differ in that they are typically called from the innermost loops of a renderer and therefore are highly performance critical. A shader is always mapped to both a hardware architecture *and* a renderer. For example, RenderMan's illuminance loop [HL90] needs to interact with the renderer to query all light sources. However, each renderer has a different API to perform this task. To avoid costly glue code, shader compilers often target one specific renderer. This forces renderer implementors to develop a whole accompanying compiler infrastructure with their renderer. For example, there are at least three different RenderMan compilers publicly available.

Even more critical, shaders are written as scalar code while most renderers internally use sophisticated parallelism on today's CPUs and GPUs, further complicating the integration into a specific renderer. An example is the use of SIMD instructions which require a specific data layout. This is one reason why no common API exists between a renderer and a shader – and is unlikely to ever exist, as it would impose limits on the design of renderers or require costly wrapper functions.

* e-mail: {karrenberg, rubinstein, slusallek, hack}@cs.uni-sb.de



DOI: 10.2312/EGGH/HPG10/097-105

Furthermore, shading languages may require support for advanced features such as bounding the values of a shader over a larger domain [HSS98, VAZH*09] or taking derivatives of arbitrary expressions in the code [HL90]. While this can be easily approximated in some rendering implementations (e.g. taking divided differences between neighboring samples on GPUs), it requires solutions such as automatic differentiation (AD) [Ral81] in those scenarios where neighboring samples are not readily available. This in turn requires one to replace all computations in a shader that the expressions depend upon by their AD counterparts.

As a result it is notoriously hard and cumbersome to integrate programmable shading into a rendering system efficiently. This has resulted in many rendering systems defining shading systems and languages that are specialized to their internal structure (often causing issues as the implementations evolve over time). Where standard shading languages are used, a complex compiler infrastructure for each language and rendering algorithm must be created, making it uneconomic to support more than one language.

Early shader implementations used an interpreter approach to programmable shading to overcome some of these issues: The shading engine interpreted an abstract shader program by sequentially calling the function implementing each op-code, allowing to substitute different implementations as required. This approach was used in the Reyes renderer [HL90] as it was operating on large batches of shader samples, allowing to amortize the interpreter overhead. With increasing shader functionality and greatly reduced batch sizes this approach is no longer viable today.

### 1.1. Contributions

AnySL provides solutions to most of the above problems by a combination of several techniques (see Figure 1):

- We borrow the idea of "subroutine-threaded code" from interpreters to represent shaders in an abstract and portable way (Section 4).
- At runtime we use type replacement to substitute the renderer's internal types and operations to be used for shading and provide implementations for renderer dependent functionality (Section 4). We demonstrate how type replacement can be used to perform code transformations such as automatic differentiation (Section 5).
- We use an embedded compiler (LLVM [LA04]) to eliminate the overhead of subroutine-threaded code, and finally perform low-level transformations such as vectorization to optimize shaders for parallel execution in packet-based ray tracers (Section 6).

We demonstrate the performance and flexibility of AnySL by several experiments. We support standard RenderMan shaders to demonstrate the coverage of advanced shading features by AnySL (e.g. including derivatives of arbitrary expressions).

Because many proprietary rendering systems still use C/C++-based shading languages, we have also implemented a C++-based shading language. It demonstrates that even these low-level languages can make use of threaded code.

We show that the same shader can be used across different ray tracers like RTfact [GS08], Manta [BSP06], and the Physically Based Ray Tracer (PBRT) [PH10] with identical visual appearance.

Our automatic SIMD vectorization technique achieves an average rendering speedup factor of 3.9 in RTfact when compared to sequential shading.

## 2. The Design of AnySL

This section gives an overview of the design of AnySL. The rest of this paper presents the most important features in further detail.

**Front End and Portable Threaded Code.** The front ends for the shading languages supported by AnySL emit so called *subroutine-threaded code* which is a well-known technique for interpreters and virtual machines [PR98]. In contrast to the common approach of compiling a shader into a sequence of instructions or op-codes, subroutine-threaded code represents the program as a sequence of (possibly nested) function calls. The functions called represent the language constructs, types, and operators of the shading language with their signatures defined by the front end (see Section 4 for more details).

Threaded code is fully portable, independent of the renderer, and contains no detail about the *implementation* of the used types and operators. For performance and portability reasons AnySL represents the portable threaded code in the bitcode file format of LLVM. LLVM is a highly flexible open-source compiler toolkit that we use also for the subsequent processing and optimization.

**Type Replacement.** In order to be able to compile shaders to executable code we need to instantiate the abstract types and operations with concrete implementations of the corresponding calls in the threaded code. These are provided by the renderer at runtime in the form of LLVM bitcode, using its internal implementation of the types and their operations for best efficiency (see Section 4).

The front end also defines the basic language constructs (control flow, intrinsic functions like `illuminance()`, etc.) as well as the interface to the renderer by generating appropriate function calls. This is very different from traditional API-based approaches that need to map each language construct to a predefined set of interfaces that each

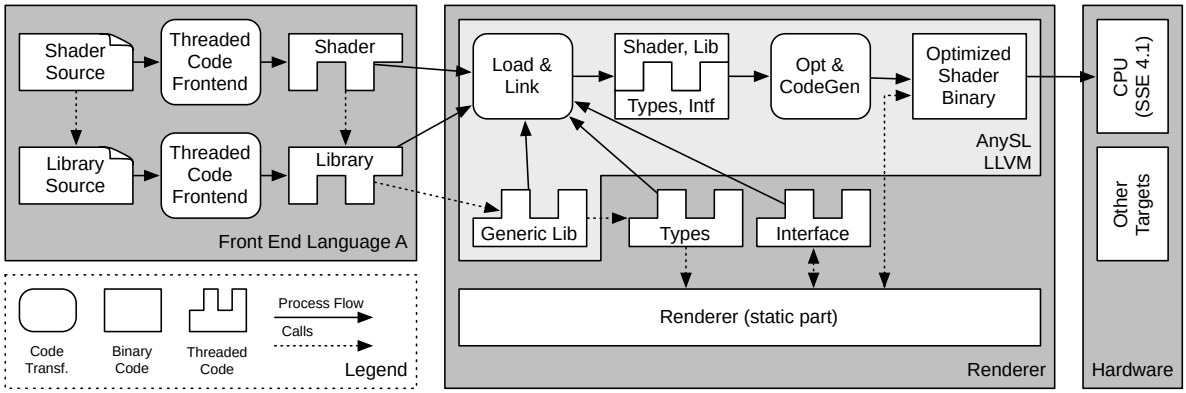


**Figure 1:** *The AnySL System: Shaders written in some shading language are compiled to abstract and portable* subroutine-threaded code *represented in LLVM bitcode (jagged boxes). The code merely consists of calls to abstract functions that represent and implement language constructs, types, operators. Most shader functionality can be implemented by a generic shader library also provided in threaded code. At runtime, the renderer provides its native types and operators (Types) as well as implementations for renderer-dependent language constructs and a wrapper for the shader (Interface). An embedded compiler (LLVM) combines the individual code fragments and eliminates any overhead through inlining and other optimizations. The optimized scalar shader code can then be transformed into parallel SIMD code. The results are finally compiled to machine code for the desired hardware architecture and can be executed immediately.*

renderer must implement. Instead, it is the task of the renderer to provide suitable implementations for all needed language constructs. This leaves much more flexibility for both the language and front end developers and—even more importantly—also for the implementations of those constructs in a specific renderer.

Because it is supplied at runtime, the renderer's implementation can be adapted easily as the renderer evolves or even at runtime depending on the scene's content or shading features activated. The same mechanism is also used to implement the renderer-dependent language constructs, such as RenderMan's illuminance loop, derivative statements, access to the current rendering state, or other features.

**Embedded Compiler.** To eliminate the overhead of threaded code we have implemented functionality that directs the compiler to perform inlining of all implementations of threaded code and apply various other optimization techniques. After these steps, the shader code is indistinguishable from a native shader written and compiled specifically for this renderer. Finally, a number of the relevant standard compiler optimization passes are applied to optimize for performance.

**Generic Shader Library.** Shading languages usually have large similarities in terms of data types (floats, vectors, etc.) and the usual graphics functions (fresnel, smoothstep, etc.). To facilitate the integration of a language, AnySL provides these operations as a language-independent, generic library implemented in subroutine-threaded code which can be directly linked into the shader.

**Low-Level Transformation & Vectorization.** All modern processor architectures provide SIMD instructions for increased performance. Using them effectively, however, requires non-trivial transformations that change the code even semantically (e.g. from a function that takes a single vector to one that takes an array of vectors). Such low-level transformations are beyond the scope of what can be implemented through type replacement. Instead, we implemented additional compiler passes in LLVM, described briefly in Section 6.

### 2.1. Using AnySL

The following steps are required to integrate a shading language into a renderer: First, the compiler of the shading language of choice has to emit subroutine-threaded code as well as meta information about shader parameters (e.g. name, abstract type, default value). This allows AnySL to automatically generate code for parameter introspection and modification that can easily be plugged into corresponding functionality of the renderer. Second, the renderer has to implement all renderer-dependent language constructs by mapping them to the renderer-specific internal API (see Section 4.1 for an example). Finally, the renderer needs an entry function for AnySL, e.g. some material that calls AnySL inside its `shade`-function.

While there is no way to avoid the implementation of a compiler for the language, large parts of the renderer-specific infrastructure can be reused if an additional language is integrated into the same renderer.

## 3. Related Work

The original and still widely used shading language is the RenderMan Shading Language (RSL) [Pix89]. The RSL compilers for Photorealistic RenderMan [Lan06] and BMRT [GH96] transform RSL programs into their own custom op-code representations and evaluate them at run time [HL90]. The byte-code instruction set is SISD (single instruction, single data) but execution happens in a virtual SIMD (single instruction, multiple data) manner. However, since parallelization is virtual and instructions are interpreted, it is difficult to achieve maximum performance using such an approach.

In real-time rendering systems, shading language compilers leverage the parallelism provided by the target architecture. On CPUs, this involves automatic vectorization and optimization of scalar (SISD) shader programs [PBBR07]. On GPUs, vectorization is implemented in hardware and the Cg, HLSL, and GLSL languages operate in SISD [MSK*03, PM03, RKL04]. Still, a translator from the shading language to machine instructions is required. Other interesting and more recent shading languages are OSL [Son10] and MetaSL [Men09], which explicitly try to support both ray tracing and rasterization based shading.

Sh [MQP02] targeted GPUs and avoided the need for a parser by using C++ meta programs as a shading language embedded in the application. The poor capabilities of C++ as a meta-language host for domain-specific languages (e.g. the lack of control structures overloading) lead to syntactical and practical inconveniences.

Guenter et al. [GKR95] produced optimized versions of shaders and allowed interactive feedback on specific parameter changes. However, their approach could only be applied in restricted scenarios where only shader parameters could be changed in a scene. This and related work, e.g [VAZH*09, OKS03] could easily be integrated into AnySL.

A common issue of all mentioned approaches is that they require building specialized compilers, which requires significant development effort. This has often led to restricting the features and expressiveness of the shading languages. Additionally, specialized CPU compilers (e.g. for RTSL [PBBR07]) have to perform most optimizations themselves. They produce vectorized C code and essentially use a standard compiler as a machine code emitter.

## 4. Subroutine-Threaded Code and Type Replacement

AnySL-enabled shaders come in a special form. One of the goals of AnySL is to represent the shader code as independently from the renderer as possible. To realize this independence, we use a well-known interpreter technique called *subroutine-threaded code* [PR98]. In subroutine-threaded code, an operator occurrence in the source program is translated to a call to a function that implements the operator.

Consider the implementation of the `faceforward` function in the RenderMan shading language.

```
vector faceforward(vector N, vector I) {
  return sign(-I.N) * N;
}
```

Its subroutine-threaded code version

```
void faceforward(void* res, void* N, void* I) {
  void* t1 = ar_alloca("float");
  void* t2 = ar_alloca("float");
  void* t3 = ar_alloca("float");

  ar_dot(t1, I, N);
  ar_negf(t2, t1);
  ar_sign(t3, t2);
  ar_mulfv(res, t3, N);
}
```

differs in the following aspects:

- All types have gone. Every variable is represented by a `void` pointer. The concrete type for each variable will be provided by the renderer later. The `void*` type can thus be seen as a type variable as in C++ template meta programming. The compiler inserts calls to allocation functions (`ar_alloca`) that receive a type tag, and return memory for a variable of that type.
- The operators are replaced with function calls. The compiler creates temporary variables to hold the results of the operations. Pointers to the result variables are passed to the operator calls.

The subroutine-threaded code is completely independent of whatever type the renderer wants to use to implement `float` and `vector`. Assume, for instance, the renderer wants to use a special fixed-point arithmetic class `Fixed` to implement the type `float`. Then, it has to provide the implementation of the basic, irreducible operators of the `float` type, e.g. the multiplication:

```
ar_mulff(void* z, void* x, void* y) {
  Fixed& fx = Cast<Fixed>(x);
  Fixed& fy = Cast<Fixed>(y);
  Fixed& fz = Cast<Fixed>(z);
  fz = fx * fy;
}
```

### 4.1. Renderer-Dependent Language Constructs

Code threading can also be applied to control structures. Consider RenderMan's `illuminance` statement:

```
illuminance(P, Nf, PI/2) {
  /* body */
}
```

In a ray tracer, `illuminance` is usually implemented by iterating over light sources and shooting secondary rays. However, different ray tracers have different APIs to perform these tasks. In AnySL this poses no problem because `illuminance` is a function that is implemented by the renderer. The only difference to the multiplication operator discussed above is that one of its arguments is actually *code*. Hence, an AnySL shader compiler for RenderMan will create a second function which contains the code of the body of the `illuminance` loop and pass it to the function (`ar_illumin`) implementing the `illuminance` operator:

```
void body_fun(void* L, void* Cl, void* Ol)
{ /* body of the illuminance loop */ }

void shade(...) {
  ...
  ar_illumin(P, N, PI/2, body_fun);
  ...
}
```

When the shader is loaded by AnySL, the implementation of `ar_illumin` is provided by the renderer in LLVM bitcode. To give an example, consider our (simplified[‡]) implementation of `ar_illumin` in RTfact.

```
void ar_illumin(void* P, void* N, void* angle,
                IlluminBody* body_fun) {
  for (int i = getNumLights(); i > 0; --i) {
    Vec Cl, Ol, L;
    sampleLightSource(i, &Cl, &Ol, &L);
    if (getLightContribution(i, &Cl))
      body_fun(&L, &Cl, &Ol);
  }
}
```

By inlining `body_fun` into `ar_illumin` and successively into `shade`, the overhead of the function calls is completely optimized away.

**Limitations.** Note that we currently only support *structural* control flow, that is if-then-else and loops *without* break and continue. Supporting the latter is feasible but would require a special transformation phase after all functions are inlined.

**Local Variables** As we compile the bodies of control structures (this also includes loop and if statements) to separate functions, we must provide an access to the local variables of the shader in those functions. This is achieved by creating a *closure* struct that contains all local variables. A pointer to this struct is passed along the functions belonging to the shader.

[‡] To ease the presentation, we omitted several technical details such as the renderer context pointer and the pointer to the local variables closure.

When the shader is loaded and all functions are inlined, AnySL uses LLVM to promote aggregate types (such as structs) to scalars, effectively breaking up the struct into its components. Those can further be promoted to registers, again leaving no performance overhead behind. All local variables in that struct then "look" as if they had been declared in the shader directly.

### 4.2. Removing the Overhead

A shader represented in threaded code directly would be usable but have a high performance overhead. This is due to the function calls and stack-allocated variables used to implement threaded code. To remove all performance obstacles, AnySL transforms the code by using a series of LLVM's optimizations:

First, types are concretized: To this end, the renderer provides a mapping from the type tags occurring in the shader to an LLVM type representation. Then, all calls to the allocation functions with abstract types (e.g. `ar_alloca("vector")`) are replaced by allocation instructions of the corresponding real types. Note that at this stage, every local variable is member of the local variable struct, as described above. For example, if the renderer supplies the type

```
struct Vector { ... };
```

for the type tag `"vector"`, AnySL patches the corresponding `ar_alloca` statements to (the LLVM equivalent) of

```
alloca(sizeof(Vector))
```

Then, aggressive function inlining is performed. This removes all function calls introduced by threaded code. Second, the shader is inlined into the interface stub provided by the renderer that is used to call the shader. At this stage, the shader code "looks" like it was written for that specific renderer, but the code is still unoptimized.

Finally, LLVM is used to run a series of standard compiler optimizations in order to remove the overhead of code threading. Most importantly, the local variables struct is (recursively) broken up into its components using *scalar replacement of aggregates*. The resulting scalar variables then undergo memory to register promotion. Hence all variables will be exposed to the register allocator, avoiding allocation on the stack if possible. With all abstraction layers removed and scalar optimizations applied, the shader can be prepared for the target platform.

## 5. Automatic Differentiation

Being able to take derivatives of arbitrary expressions with respect to screen space has been instrumental in supporting analytic antialiasing in shaders by avoiding generation of frequencies above the sample rate. Our use of threaded code

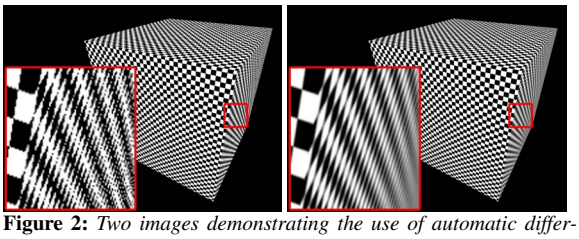

**Figure 2:** *Two images demonstrating the use of automatic differentiation for filtering of a checker board shader. The implementation in AnySL is both easy and elegant, as it only requires different data-types (each float has two additional "shadow" values) in the runtime library.*

in combination with type replacement enables us to easily support such custom high-level transformations on shader code *before* code generation.

Since we can replace any types in the threaded code, we can implement automatic differentiation (AD) by simply replacing the normal `float` type by one that maintains additional derivatives of the main value with respect to some parameterization (e.g. typically the image plane). In addition, the basic arithmetic operations on `float` are extended to maintain these derivatives in all computations, which also requires AD versions of the basic math functions (unless they are already implemented in terms of the basic data type). Finally, the renderer must also supply the shader input with derivatives. We can then implement the language features that provide derivatives, such as RSL's `Du(x)` function, in terms of the computed derivatives of $x$. Figure 2 demonstrates the technique on a checkerboard shader where aliasing artifacts are removed.

A simple implementation will replace all basic types with their AD versions and rely on the compiler optimization to eliminate computations and storage that does not contribute to the output of the shader. We have verified that this is the case for our implementation.[§] This approach also directly extends to other techniques including the use of interval or affine arithmetic to compute conservative bounds on shaders [HSS98, VAZH*09].

## 6. SIMD Vectorization

Using the SIMD instructions of modern CPUs is best done automatically. Even smaller shaders are very complicated to translate into *efficient* SIMD code. To use SIMD instructions, the programmer has to resort to so-called *intrinsics*, basically compiler-known functions that expand to specific machine instructions. The compiler then merely serves as a register allocator and code scheduler. While the intrinsics can be hidden behind overloaded operators (see e.g. [GS08]), the mere usage of SIMD instructions is not the key to performance. To exploit SIMD instructions effectively, the control flow of the shader has to be replaced by data flow. To this end, branches are eliminated (except for loop-back branches). Control-flow joins are substituted by blend operations that select the appropriate value.

This technique is well-known in the area of vectorizing compilers (see e.g. [AKPW83]) and has recently found application in shader compilers for packet-based ray tracers. Such compilers [PBBR07] perform shader vectorization directly in the front end of the compiler and generate C code using instrinsics. The reason for this is that performing this kind of vectorization on an abstract syntax tree in the front end is less complex than on arbitrary control flow of low-level code such as LLVM bitcode.

However, this approach has also three major disadvantages: First, portability is lost, because the intrinsics target one specific processor architecture. Second, supporting a new processor architecture will cause large extensions being made to the front end. And lastly, optimization potential is lost: Compiler optimizations do not work well on vectorized code. The fact that control flow is no longer explicit disturbs many compiler optimizations. Hence, it is better to first apply optimizations on the scalar code and then perform the vectorization. Thus, already-vectorized code cannot exploit all the optimization potential of the C compiler.

These problems are avoided in AnySL's vectorization algorithm: To this end, we extended the algorithm of Allen and Kennedy [AKPW83] to work on low-level static single-assignment programs as they occur in LLVM bitcode. Our algorithm transforms an entire bitcode program into another bitcode program; it is not limited to loops or other code blocks. LLVM provides SIMD instructions in its instruction set, the transformation is therefore independent of the target platform. Hence, our vectorization algorithm can be used with every LLVM back end that supports vector instructions. Furthermore, AnySL will thereby also profit from upcoming back ends, for example for the AVX instruction set or Larrabee. For the sake of brevity we do not go into further detail and refer to [Kar09].

## 7. Evaluation

To evaluate AnySL we integrated it into three ray tracers (RTfact [GS08], Manta [BSP06], and PBRT [PH10]), adapted two language front ends (RenderMan and C++), and extended LLVM by the vectorization pass described in Section 6.

**Setup.** All experiments were conducted on a Core 2 Quad (Q9550) with a SIMD width of 4 (SSE4.1) at a clock rate

[§] A more sophisticated implementation might use a data flow analysis to determine only those variables whose types *have* to be substituted.

**Figure 3:** *Scenes rendered with RTfact, Manta (2nd from right), and PBRT (rightmost). The surface shaders are written in RenderMan or scalar C++ and all compiled to a platform-independent intermediate representation. The AnySL system loads, vectorizes, and optimizes the shaders and seemlessly integrates them into the renderer at runtime, achieving close to native shading performance.*

of 2.8GHz and 4GB of memory running Ubuntu Linux 9.10. The resolution was set to 512x512 pixels.

To improve performance for complex procedural shaders using noise (cf. [Per02]) we implemented a variant of noise optimized for vectorization, similar to [Ola05]. The original noise function uses complicated branching patterns and repeated indexing into a permutation table, which requires splitting and reassembling of packets due to the lack of a scattered load instruction in SSE. Our optimized variant is entirely branch-free and generates the pseudo-random numbers on the fly instead of using a table, resulting in similar performance gains for shaders with noise components as for those without. Note that our noise function is written in scalar code and automatically vectorized.

The scalar shaders are loaded from threaded-coded LLVM bitcode files at runtime. Before vectorization, they are specialized into our shading system and optimized using LLVM's internal passes. The whole procedure is efficient and allows for recompilation at runtime: The compilation times range from ∼55ms for the Phong shader to ∼200ms for complex shaders with inlined noise (e.g. Parquet). This allows dynamic modification of shader parameters at runtime with an immediate recompilation to get optimal performance and full vectorization.

**RTfact.** We compare the rendering performance of automatically vectorized shaders against scalar shading where packets are split and the scalar shader is executed sequentially. This is really the only option for non-trivial shaders if automatic vectorization is not available. Because of the complexity of writing vectorized shaders by hand (see Section 6), we can only compare to a few simple hand-written native shaders.

The vectorized versions of the shaders outperform their scalar counterparts by an average factor of 3.9 (see Table 1). In combination with the better cache coherence that comes with the packet shading, we even achieve superlinear speedups of factors up to 5.0.

**Manta.** Manta, although also being a packet ray tracer, internally differs largely from RTfact: RTfact is able to remove large parts of the overhead that comes with a flexible, object-oriented design by heavy usage of templates, specialization, and inlining. Most importantly however, its internal algorithms are entirely based upon SIMD primitives and operations on those, whereas Manta only employs vectors and vector operations in certain parts of its algorithms and requires careful tuning by hand. This is especially visible for calls from the shader back to the rendering engine (such as `illuminance()`). These calls are highly performance-critical and are required to operate seamlessly on SIMD data types to expose the full potential of the vectorized shaders.

| Shader | Native | Scalar | Vectorized | Speedup |
|---|---|---|---|---|
| Brick | - | 8.8 | 31.4 | 3.6x |
| Checker | 34.5 | 8.8 | 31.8 | 3.6x |
| Glass | - | 0.9 | 4.5 | 5.0x |
| Granite | - | 7.2 | 24.6 | 3.4x |
| Venus | - | 7.6 | 25.7 | 3.4x |
| Parquet | - | 4.3 | 18.6 | 4.3x |
| Phong | 35.5 | 14.1 | 32.5 | 2.3x |
| Screen | - | 4.6 | 22.7 | 4.9x |
| Starball | - | 4.5 | 20.0 | 4.4x |
| Wood | - | 4.4 | 19.1 | 4.3x |
| Average | - | 6.5 | 23.3 | 3.9x |

**Table 1:** *Performance of RTfact for different RSL shaders, measured in frames per second in scenes with one shader on a (triangulated) sphere and two point lights. Due to the difficulties of writing packet shaders by hand, we can only compare against two hand-optimized native Phong and Checker shaders, which show a performance difference of less than 10%.*

In combination with our limited knowledge of Manta, these issues do not make the renderer a choice as ideal for our automatically vectorized shaders as RTfact.

Despite these drawbacks, Manta still reaches an average speedup of 1.6 with peaks up to 2.5 for some shaders in our experiments.

**PBRT.** While RTfact and Manta are real-time ray tracers that focus on performance, PBRT is an offline ray tracer aimed at image quality and realism, resulting in two major differences: First, PBRT does not trace packets of rays and therefore cannot benefit of our vectorization algorithm. Second, it uses a different, physically-based shading model.

RTfact and Manta use an *active* model where the shader is entirely responsible for everything that affects a material's visual appearance including all lighting computations and

recursive ray shooting. In PBRT however, shading is controlled by an integrator that queries the *passive* material for its reflectance properties and performs lighting computations accordingly, whereas the material itself is only responsible for *local* computations, e.g. determining a base color by texture lookup.

Both RenderMan and our C++ shading language assume the active model and therefore have constructs for lighting computations that happen *inside* the shader.

Mapping both shading models to one unified API that can still be mapped to the actual shading language without overhead is—if possible at all—out of the scope of this paper. Even more importantly, portability would be lost: We want to provide the same visual appearance of a material in different renderers without manually modifying the shader code. Thus, the only way to reuse our portable shaders in PBRT and obtain visually identical results is to remove all lighting computations from the integrator and leave them to the shader.

If visual equality is not of importance, AnySL still allows to use all of PBRT's possibilities. The renderer-dependent implementation of e.g. `illuminance` would then be left empty, leaving the integrator with all responsibility again.

**Visual Appearance.** The visual analogy between different renderers using the same shaders with AnySL is exemplified in Figure 4. Note that we did not perform any major changes to the systems beyond the integration of AnySL and adjustment of tone mapping and 3D scene formats.

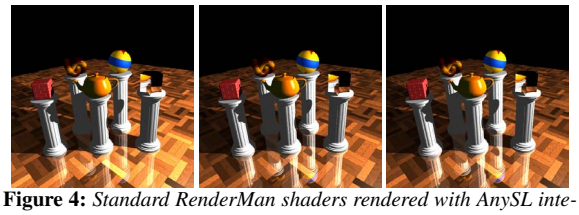

**Figure 4:** *Standard RenderMan shaders rendered with AnySL integrated into three different ray tracers (from left to right): RTfact, Manta, and PBRT.*

## 8. Conclusions and Future Work

We showed that AnySL greatly reduces the overhead of supporting multiple shading languages by mapping them into a portable representation based on threaded code. Type replacement and the embedded compiler then allow to compile such shaders in the context of different renderers and still achieve close to native shading performance. Type replacement also allows to transparently perform automatic differentiation and related techniques on any shader. We further showed that AnySL enables different renderers to obtain visually identical results for our portable shaders.

The support of automatic vectorization of scalar shaders for SIMD architectures is a key feature, given that it is almost impossible to write non-trivial SIMD shaders by hand. We demonstrated its effectiveness, reaching an average speedup factor of 3.9.

Future work includes supporting additional front ends for languages like OpenSL, HLSL, etc. and optimized back ends for other execution platforms such as OpenCL, HLSL, or GLSL to support GPUs.

As currently the only cross-language and cross-renderer shading framework, AnySL also forms a good basis to finally start defining a standard exchange format for programmable shading for computer graphics.

## Acknowledgements

Ralf Karrenberg is supported by the Cluster of Excellence on Multimodal Computing and Interaction. The bunny and buddha models are used by courtesy of the Stanford Computer Graphics Laboratory.

## References

[AG00] APODACA A. A., GRITZ L.: *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann, 2000. ISBN: 1558606181. 1

[AKPW83] ALLEN J. R., KENNEDY K., PORTERFIELD C., WARREN J.: Conversion of control dependence to data dependence. In *POPL* (New York, NY, USA, 1983), ACM, pp. 177–189. 6

[BSP06] BIGLER J., STEPHENS A., PARKER S. G.: Design for Parallel Interactive Ray Tracing Systems. *IEEE Symposium on Interactive Ray Tracing* (2006). 2, 6

[GH96] GRITZ L., HAHN J. K.: BMRT: A Global Illumination Implementation of the RenderMan Standard. *Journal of Graphics Tools 1* (1996), 29–47. 4

[GKR95] GUENTER B., KNOBLOCK T. B., RUF E.: Specializing Shaders. In *SIGGRAPH* (New York, NY, USA, 1995), ACM, pp. 343–350. 4

[GS08] GEORGIEV I., SLUSALLEK P.: RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *IEEE/Eurographics Symposium on Interactive Ray Tracing 2008* (Aug 2008). 2, 6

[HL90] HANRAHAN P., LAWSON J.: A Language for Shading and Lighting Calculations. *SIGGRAPH 24*, 4 (August 1990), 289–298. ISBN: 0-201-50933-4. 1, 2, 4

[HSS98] HEIDRICH W., SLUSALLEK P., SEIDEL H.-P.: Sampling procedural shaders using affine arithmetic. *ACM Trans. Graph. 17*, 3 (1998), 158–176. 2, 6

[Kar09] KARRENBERG R.: *Automatic Packetization*. Master's thesis, Saarland University, 2009. 6

[LA04] LATTNER C., ADVE V.: LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *CGO* (Palo Alto, California, Mar 2004). 2

[Lan06] LANCASTER T.: Pixar's PhotoRealistic RenderMan version 13. In *SIGGRAPH* (New York, NY, USA, 2006), ACM. 4

[Men09] MENTAL IMAGES: MetaSL Specification, 2009. 4

[MQP02] MCCOOL M. D., QIN Z., POPA T. S.: Shader Metaprogramming. In *HWWS* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 57–68. 4

[MSK*03] MARK W. R., STEVEN R., KURT G., MARK A., KILGARD J.: Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics 22* (2003), 896–907. 4

[OKS03] OLANO M., KUEHNE B., SIMMONS M.: Automatic shader level of detail. In *HWWS* (Aire-la-Ville, Switzerland, Switzerland, 2003), Eurographics Association, pp. 7–14. 4

[Ola05] OLANO M.: Modified noise for evaluation on graphics hardware. In *HWWS '05* (New York, NY, USA, 2005), ACM, pp. 105–110. 7

[PBBR07] PARKER S., BOULOS S., BIGLER J., ROBISON A.: RTSL: A Ray Tracing Shading Language. *IEEE Symposium on Interactive Ray Tracing* (2007). 4, 6

[Per02] PERLIN K.: Improving Noise. In *SIGGRAPH* (2002), pp. 681–682. 7

[PH10] PHARR M., HUMPHREYS G.: *Physically Based Rendering: From Theory To Implementation, 2nd Edition*. Morgan Kaufmann, 2010. 2, 6

[Pix89] PIXAR: *The RenderMan Interface*, September 1989. 4

[PM03] PEEPER C., MITCHELL J. L.: Introduction to the DirectX 9 High-Level Shader Language, 2003. 1, 4

[PR98] PIUMARTA I., RICCARDI F.: Optimizing direct threaded code by selective inlining. In *PLDI* (New York, NY, USA, 1998), ACM, pp. 291–300. 2, 4

[Ral81] RALL L.: *Automatic Differentiation: Techniques and Applications.* Springer, 1981. 2

[RKL04] ROST R. J., KESSENICH J. M., LICHTENBELT B.: *OpenGL Shading Language*. Addison Wesley, 2004. 1, 4

[Son10] SONY PICTURES IMAGEWORKS: Open Shading Language Specification, Version 0.9, January 2010. 4

[VAZH*09] VELÁZQUEZ-ARMENDÁRIZ E., ZHAO S., HAŠAN M., WALTER B., BALA K.: Automatic bounding of programmable shaders for efficient global illumination. In *SIGGRAPH Asia* (New York, NY, USA, 2009), ACM, pp. 1–9. 2, 4, 6