Rodent: Generating Renderers without Writing a Generator

ARSÈNE PÉRARD-GAYOT, Saarland University
RICHARD MEMBARTH, DFKI & Saarland University
ROLAND LEISSA, Saarland University
SEBASTIAN HACK, Saarland University
PHILIPP SLUSALLEK, DFKI & Saarland University

Monte-Carlo Renderers must generate many color samples to produce a noise-free image, and for each of those, they must evaluate complex mathematical models representing the appearance of the objects in the scene. These models are usually in the form of shaders: Small programs that are executed during rendering in order to compute a value for the current sample.

Renderers often compile and optimize shaders just before rendering, taking advantage of the knowledge of the scene. In principle, the entire renderer could benefit from a-priori code generation. For instance, scheduling can take advantage of the knowledge of the scene in order to maximize hardware usage. However, writing such a configurable renderer eventually means writing a compiler that translates a scene description into machine code.

In this paper, we present a framework that allows generating entire renderers for CPUs and GPUs without having to write a dedicated compiler: First, we provide a rendering library in a functional/imperative language that elegantly abstracts the individual rendering concepts using higher-order functions. Second, we use *partial evaluation* to combine and specialize the individual components of a renderer according to a particular scene.

Our results show that the renderers we generate outperform equivalent high-performance implementations written with state-of-the-art ray tracing libraries on the CPU and GPU.

CCS Concepts: • **Computing methodologies → Rendering**; • **Software and its engineering → Compilers**.

Additional Key Words and Phrases: Rendering, Ray-tracing, Generator, Partial Evaluation

ACM Reference Format:
Arsène Pérard-Gayot, Richard Membarth, Roland Leißa, Sebastian Hack, and Philipp Slusallek. 2019. Rodent: Generating Renderers without Writing a Generator. *ACM Trans. Graph.* 38, 4, Article 40 (July 2019), 12 pages. https://doi.org/10.1145/3306346.3322955

Authors' addresses: Arsène Pérard-Gayot, Saarland University, Saarland Informatics Campus E1 1, Saarbrücken, Germany, 66123, perard@cg.uni-saarland.de; Richard Membarth, DFKI & Saarland University, Saarland Informatics Campus D3 2, Saarbrücken, Germany, 66123, membarth@dfki.de; Roland Leißa, Saarland University, Saarland Informatics Campus E1 3, Saarbrücken, Germany, 66123, leissa@cs.uni-saarland.de; Sebastian Hack, Saarland University, Saarland Informatics Campus E1 3, Saarbrücken, Germany, 66123, hack@cs.uni-saarland.de; Philipp Slusallek, DFKI & Saarland University, Saarland Informatics Campus D3 2, Saarbrücken, Germany, 66123, slusallek@dfki.de.

0730-0301/2019/7-ART40 $15.00
https://doi.org/10.1145/3306346.3322955

1 INTRODUCTION

Monte-Carlo Renderers are naturally complex and configurable pieces of software: They typically incorporate powerful renderer-specific languages to describe 3D scenes. Most of those languages allow the designer to define small programs that define the appearance of an object, called *shaders* [Pixar 1988; Sony Pictures Imageworks 2017]. Before rendering, the renderer *compiles* these shader programs into either machine code or an internal representation. During shader compilation, most renderers perform standard code optimizations, such as constant folding or dead code elimination. At this stage, some renderers also apply *specialization* [Guenter et al. 1995; McCool et al. 2002; Sons et al. 2014]: Since the scene is known, some computations in the shaders might have become superfluous and can thus be eliminated. For instance, the renderer can avoid fetching and interpolating vertex attributes if those attributes are not present or unused, or remove redundant texture lookups.

Even when shaders are specialized, concrete implementations of rendering systems suffer from the same problem as many other high-performance codes: In terms of software engineering, all components (e.g. shading system, integrator, primitive intersection and acceleration structure traversal) are ideally decoupled from each other and implemented in a high-level, abstract way that focuses on the algorithmic properties to make code reusable and easier to understand. However, to achieve this decoupling and abstraction one typically has to pay a performance price. Essentially, every code abstraction technique that the programmer uses needs a compiler optimization to remove it. Because compilers usually do not succinctly remove all of these abstractions, programmers sacrifice genericity for performance. Code is specialized manually and generic code is tainted by target-dependent artifacts: Developers use SIMD intrinsics to vectorize their algorithms, or write CUDA code that can only run on a GPU.

A solution to this problem is to optimize across rendering library layers. If the scene description is known at every layer during compilation, constants can be folded, branches pruned, and the renderer becomes more efficient, because it is specialized for that scene. Of course, not all scene parameters will enable performance critical optimizations, and some parameters may be unknown at compile-time, like the camera position. In practice, knowing the *type* of each object is enough. However, to generate a specialized renderer from a scene description would normally require to write a dedicated, *domain-specific compiler* that can not only specialize shaders, but also the other components. From the scene description and rendering parameters, this compiler would generate machine code that corresponds to a renderer in that specific configuration.

In this paper, we use the first Futamura projection [Futamura 1982], a classic result from metaprogramming, to avoid writing such a renderer generator. Based on this idea, we present the design of the rendering library Rodent that uses the partial evaluation framework AnyDSL [Leißa et al. 2018]. Rodent is declarative and separates algorithmic descriptions from their *mapping* to the target architecture. This allows Rodent to generate renderers for CPUs and GPUs whose implementations *share* most of their code. On top of this, the renderers generated by Rodent via partial evaluation outperform equivalent renderers written using Embree [Wald et al. 2014] and OptiX [Parker et al. 2010]. These two libraries are designed and hand-optimized by hardware vendors. Our renderers all use Path Tracing with Multiple Importance Sampling, and contain enough features to render scenes modelled by independent artists (see Figure 6).

Contributions. In summary, this paper:

(i) defines a declarative rendering library that follows the mathematical foundations of rendering and cleanly separates hardware-related aspects from rendering concepts (Section 4),
(ii) explains how to apply partial evaluation to this renderer-generating library in order to produce specialized renderers without writing a compiler (Section 4.6 and Section 4.7),
(iii) demonstrates that those generated renderers outperform equivalent ones developed with state-of-the-art, industrial-grade libraries on both, CPUs and GPUs (Section 5.3).

2 RELATED WORK

This paper draws from previous work in different, but connected areas: rendering libraries, specialization of rendering algorithms and shaders, and partial evaluation.

2.1 Rendering Libraries

Generic libraries and software systems for rendering exist in the literature [Döllner and Hinrichs 2002; Slusallek and Seidel 1995]: Using object orientation, the rendering system is split into classes representing lights, shaders, integrators, and other components. However, the focus of these libraries is typically on flexibility, rather than performance. In fact, these frameworks are implemented in C++ and make heavy use of virtual function calls, incurring a significant performance penalty.

Other, lower-level libraries only provide the basic infrastructure to build a renderer [Parker et al. 2010; Wald et al. 2014; Zellmann et al. 2017], like the traversal and intersection routines, and let the programmer write his own modules on top of it.

Among those, OptiX [Parker et al. 2010] is a framework to write high-performance GPU-based renderers. It lets the user write C++ programs in order to specify the behavior of a ray tracer when emitting rays, intersecting a ray with a surface, or shading a surface. Most of the library is tainted with GPU-specific aspects: For instance, data has to be exchanged between the host and device through buffers, vertex attributes are passed between programs using global variables, and user-specified programs have to follow CUDA [NVIDIA 2019] semantics. Additionally, the OptiX compiler can only generate a megakernel from the user-specified programs, which is known to be less efficient than the wavefront execution model [Laine et al. 2013].

Embree [Wald et al. 2014] is another low-level library providing highly optimized traversal and intersection routines for x86 CPUs. Since Embree itself does not provide support for writing shaders, the reference renderer provided in the Embree SDK uses `ispc` [Pharr and Mark 2012] to vectorize shading code. The latter is a vectorizing compiler that understands a Single Program Multiple Data dialect of C. Bindings to Embree are available for `ispc`, such that an `ispc` program can use the traversal kernels contained in Embree. However, this forces the renderer to be written mostly with `ispc`, using a C-based language in which building abstractions is difficult if not impractical (see discussion in Section 5.5). On top of this, and much like with OptiX, the design of `ispc` and Embree force the rendering code to be tainted with hardware- or platform-specific aspects like the `varying`-ness of the arguments of a function.

2.2 Specialization and Rendering

Specialization, in the context of rendering, is often restricted to the shading system [Guenter et al. 1995; He et al. 2018; McCool et al. 2002; Sons et al. 2014], or focuses on simple rendering algorithms [Andersen 1995; Asai 2002; Georgiev and Slusallek 2008], like Whitted-style ray tracing [Whitted 1980].

Typically, shader specialization is done by performing some form of partial evaluation on the shading program, written in a dedicated language: a subset of C [Guenter et al. 1995], a DSL embedded inside C++ [McCool et al. 2002], or a subset of Javascript [Sons et al. 2014]. Consequently, a major drawback of these systems is that they require to design a compiler specific to the chosen language.

Nevertheless, it is widely accepted that shader specialization is beneficial for performance. In the literature, specialized shaders typically perform between a couple of percent faster to around two times faster than the original, non-specialized shaders, depending on the type of specialization applied and the shader complexity.

Traversal algorithms are also a typical use-case for specialization [Georgiev and Slusallek 2008; Pérard-Gayot et al. 2017; Selgrad et al. 2015]. The programmer specializes a traversal algorithm to configure it for a particular primitive type or intersection routine, for instance. The result is an optimal traversal kernel that is as fast or even faster than a highly optimized manual implementation.

Rather than advocating the design of a compiler specific to renderers, this paper argues, like others before [Andersen 1995; Futamura 1982; Pérard-Gayot et al. 2017], that specialization can be performed using partial evaluation.

2.3 Partial Evaluation and Metaprogramming

Traditional compiler optimizations must terminate and prevent exponential growth of the program. For these reasons, the heuristics used in many optimizations like inlining are carefully chosen, so that a compiler wouldn't blindly inline recursive functions. Thus, traditional compiler optimizations cannot reliably remove abstractions from a given program.

Partial evaluation ignores these constraints and aggressively evaluates parts of the program in the following way [Futamura 1982; Jones 1996]: Given a program P with dynamic inputs D (run-time)

and static inputs S (compile-time), a partial evaluator will then take P and values for S, and produce a *residual program* R (or just *residuum*) in which the static inputs have been specialized to the given values. Note that partial evaluation does not only mean substituting the static variables with their values but also evaluating the part of the program that only depends on the static variables. Due to the halting problem, partial evaluators rely on user annotations to decide which parts to specialize. Note further that many compiler passes such as loop unrolling or function inlining are special cases of partial evaluation.

One can achieve similar goals with metaprogramming, and in particular C++ templates. With metaprogramming the programmer must manually dissect the program into two (or more) stages—usually compile-time and run-time. This makes metaprograms hard to read and understand as the stage is a notorious feature of the syntax. The programmer essentially has to write a program that generates another program. Metaprogramming has some serious drawbacks: First, the residuum generated by the metaprogram may be ill-typed. This is the reason for the infamous error messages when dealing with C++ templates. With partial evaluation, well-typedness of the residuum comes for free. Second, the programmer must potentially implement different versions of the same function for different compile/run time combinations. For instance, a C++ programmer wanting to implement a power function for both a compile-time known exponent and a run-time known exponent will write two functions:

```
int pow(int a, int b) { /*...*/ }
template <int B> int pow(int a) { /*...*/ }
```

Note that the **constexpr** feature of C++ would not help here because it can only evaluate `pow` *fully*, and not *partially*.

In this paper, we use the first Futamura projection [Futamura 1982] to implement our renderer-generating library. In essence, the first Futamura projection states that the compiled version of some program P can be obtained by *partially evaluating* an *interpreter* to P. By following the so-called *tagless interpreter* approach [Carette et al. 2007], the interpreter itself can be written as a library of higher-order functions whose composition constitutes the (abstract syntax) of P. The important consequence from this is that using partial evaluation, *the difference between a language and a library essentially vanishes.*

To put this into context, Rodent is a form of tagless interpreter: It is an interpreter of the scene description language that describes what the renderer should do for every possible scene. By partially evaluating Rodent with the scene description, we obtain a renderer specialized for that scene.

3 BACKGROUND

Rodent is written in the programming language Impala [Leißa et al. 2018]. Impala provides several features that are crucial for implementing Rodent. Most importantly, it provides partial evaluation filters that allow the programmer to control the partial evaluator in Impala's compiler. In this section, we briefly discuss the properties of Impala that are most important for our work.

3.1 Basic Features

In Impala, the loop

```
for i in range(0, 3) { print(i) }
```

is syntactic sugar for:

```
range(0, 3, |i| { print(i) })
```

The function `range` is called with the body of the **for** loop as a last argument. The loop body is nothing more than an anonymous function taking the loop counter `i` as parameter. This syntax allows the programmer to define custom iteration functions: In particular, the implementation of `range` is written in Impala itself!

The Impala compiler also offers several built-in iteration functions: **vectorize**, **parallel**, **cuda**, and others. These functions let the programmer specify what should be vectorized, parallelized, or executed on the GPU (and via which GPGPU API). Vectorization is performed using the Region Vectorizer [Moll and Hack 2018], and parallelization uses Intel Threading Building Blocks [Reinders 2007]. Note that these functions are *not* pragmas: They behave like any other higher-order function that is written by the user and can be passed around. The only difference is that their implementation is provided by the compiler.

3.2 Partial Evaluation

In Impala, *filters* [Consel 1988] control partial evaluation. These are Boolean expressions that independently determine for each call-site whether to specialize a function call at compile time. They are denoted by the `@` symbol, and we highlight them in every code snippet. Consider the filter of a recursive `power` function in the following example:

```
fn @(n < 64) pow(x: i32, n: i32) -> i32 {
  if n == 0 {
    1
  } else if n % 2 == 0 {
    let y = pow(x, n / 2);
    y * y
  } else {
    x * pow(x, n - 1)
  }
}
```

The filter will evaluate to **true** if the argument that is passed to `n` at a particular call-site is statically known and is less than 64. This behavior is recursive. Thus, if yet another filter-annotated function is called inside the specialized one, the partial evaluator will examine this call, too. For instance, the call

```
let z = pow(x, 5);
```

will be replaced with the following equivalent code, since each specialization yields another call to `pow` until the recursion has been unrolled:

```
let y1 = x * x; let y2 = y1 * y1; let z = x * y2;
```

On the other hand, calls such as `pow(x, 64)` or `pow(x, n)` where `n` is unknown at compile-time, will remain, because in these cases, the filter does not evaluate to **true** but to **false** (in the former case) or to some symbolic expression (in the latter case).

Functions can be annotated with empty filters, or no filters at all. If the filter is empty, it is assumed to always evaluate to **true**, and

Table 1. Overview of the rendering abstractions.

Images & Textures (4.1)	Lights (4.3)
`struct Image`	`struct Light`
`type Texture`	`struct DirectLightSample`
`struct BorderHandling`	`struct EmissionSample`
`type ImageFilter`	**Geometries (4.4) & Shaders (4.5)**
Materials & BSDFs (4.2)	`struct Geometry`
`struct Bsdf`	`struct Shader`
`struct BsdfSample`	**Renderers (4.6) & Devices (4.7)**
`struct Material`	`struct Tracer`
`struct EmissionValue`	`struct Device`

thus acts like a recursive "always inline" attribute, useful for small functions:

```
fn @inc(i: i32) -> i32 { i + 1 } // will be always inlined
let inc = @|i|{ i + 1 }; // equivalent (anonymous function syntax)
```

If the function contains no filter, then the filter is assumed to always evaluate to `false`, and thus prevents partial evaluation.

Partial evaluation can drastically affect the performance of the program, much like traditional compiler optimizations. However, using filter annotations in Impala, the programmer can decide exactly when and where to specialize. This is in stark contrast to traditional optimizations that are driven by heuristics and hardly in the hands of the programmer.

3.3 Continuations

Continuations are functions that never return. Their return type is `!` ("bottom"). For example, `fn (i32) -> !` is the type of a continuation that takes an integer. As the type `!` has no value constructor, continuations must call other continuations to be correctly typed. Internally, Impala represents every function as a continuation. For instance, `fn get_42() -> i32 { 42 }` is just syntactic sugar for:

```
fn get_42(return: fn (i32) -> !) -> ! { return(42) }
```

Continuations allow the programmer to write functions that return to different code blocks that expect different types—similar to checked exceptions:

```
fn safe_div(x: i32, y: i32, error: fn() -> !) -> i32 {
  if y != 0 { x / y } else { error() }
}
```

Since continuations are first-class citizens in Impala, programmers can even capture standard continuations such as `return`, `break`, or `continue` and pass them around just like any other variable. This allows programmers to emulate a multi-level `break`:

```
for i in range(0, 100) {
  let exit_outer = break;
  for j in range(0, 100) { exit_outer() }
}
```

4 RENDERING LIBRARY

It seems reasonable to expect any high-level rendering library to be a direct translation of the mathematical foundations. Slusallek and Seidel [1995] already discuss such an architecture. Unfortunately, in most languages, there is a cost for *abstraction*: For instance, virtual functions in C++ are often expensive because inlining calls to such functions requires to perform devirtualization, an optimization that is not always possible. Impala enables us to freely design the abstractions of Rodent and use partial evaluation to remove the incurred overhead.

Rodent consists of a collection of composable rendering abstractions built on top of each other (see Table 1). Application developers combine these abstractions to create a complete scene description, and then generate a renderer specialized for that scene—either completely, if everything is known, or partially. Rodent consists of two different parts that separate *what* is computed—the algorithm in a declarative form—from *how* it is computed—the hardware-specific mapping. This design philosophy stems from domain-specific languages like Halide [Ragan-Kelley et al. 2013] or GraphIt [Zhang et al. 2018]. We only present a set of abstractions for a Path Tracer with Multiple Importance Sampling, due to the limited space available in the paper. The approach can be generalized to handle more complex rendering effects not mentioned here, since it only relies on partial evaluation—a standard compiler optimization technique—and not a specifically crafted compiler. We start by presenting the *declarative* part of Rodent that represents the scene (Section 4.1 through Section 4.5) and rendering algorithm (Section 4.6), and then describe the *mapping* of these abstractions to the target hardware (Section 4.7).

4.1 Images and Textures

Rodent defines Images as a 2D discrete collection of pixels:

```
struct Image {
  pixels: fn (i32, i32) -> Color,
  width:  i32, height: i32,
}
```

A filter is as a function that maps an image and a position in the unit square $[0, 1]^2$ to a color:

```
type ImageFilter = fn (Image, Vec2) -> Color;
```

Note how the following constructor yields a *nearest neighbor* filter:

```
fn @make_nearest_filter() -> ImageFilter {
  @ |img, uv| {
    let x = min((uv.x * img.width  as f32) as i32, img.width -1);
    let y = min((uv.y * img.height as f32) as i32, img.height-1);
    img.pixels(x, y)
  }
}
```

This is not so much different from object oriented languages: There, a filter would be represented as an `ImageFilter` interface, and a nearest filter would be a class that derives from `ImageFilter`. In our case, returning a function is similar to creating an object that contains a *vtable*. The fundamental difference, however, is the partial evaluation annotations (see Section 3) that we added to the constructor and the returned function. These annotations instruct the compiler to always specialize those functions at compile-time, meaning that the residual program will not contain any calls to `make_nearest_filter`. This particular way of annotating constructors together with the functions they return is essential to remove closures and thus ensure performance: Much like with virtual functions, compilers generally cannot optimize calls to closures since they do not know which function is actually going to be called. Moreover, closures that *capture* their environment need additional storage, typically dynamically allocated, unless the environment is small enough to be held in a

pointer. Therefore, running code that creates closures on a GPU is a challenge. In the remaining of this paper, we follow this principle to the letter: Constructors and the functions they return are always annotated to force specialization. Most top-level functions in Rodent are just constructors: The actual computation happens inside the functions returned by those constructors. Thanks to this, placing partial evaluation filters is just a matter of applying the rule mentioned previously. For more complicated, and possibly recursive functions, the developer has to find a termination criterion for the function and use it as a filter, just like for `pow`, in Section 3.2. Note that the user of our library does not need to write filters, since he should only call constructors, and those already contain filters.

In order to bring texture coordinates back into the unit square, Rodent defines a border handling abstraction:

```
struct BorderHandling {
  horz: fn (f32) -> f32, vert: fn (f32) -> f32,
}
```

Rodent applies the `horz` border mode for the first texture coordinate and `vert` for the second one. For example, the following constructor yields a *clamping* border handling mode:

```
fn @make_clamp_border() -> BorderHandling {
  let clamp = @ |x| min(1.0f, max(0.0f, x));
  BorderHandling{horz: clamp, vert: clamp}
}
```

Finally, Rodent wraps all these objects into a texture. This is a function that maps a texture coordinate to a color:

```
type Texture = fn (Vec2) -> Color;
```

To create a texture, Rodent provides the following constructor:

```
fn @make_texture(border: BorderHandling, filter: ImageFilter,
                 image: Image) -> Texture {
  @ |uv| {
    let (u, v) = (border.horz(uv.x), border.vert(uv.y));
    filter(image, make_vec2(u, v))
  }
}
```

Application developers define textures by calling this constructor with the desired border handling mode and filter:

```
let tex = make_texture(make_repeat_border(),
                       make_bilinear_filter(), image);
```

Evaluating the color for a particular texture coordinate is just a matter of invoking the texture function:

```
let color = tex(make_vec2(0.5f, 0.7f));
```

Since textures are just functions, Rodent allows procedural textures:

```
fn @make_checkerboard_texture() -> Texture {
  @ |uv| {
    let (x, y) = (uv.x as i32, uv.y as i32);
    if (x + y) % 2 == 0 { white } else { black }
  }
}
```

4.2 Materials and BSDFs

When rendering physically-based models, materials are represented with their associated Bidirectional Scattering Distribution Function (BSDF). This function is used in conjunction with the *rendering equation* [Kajiya 1986] to describe light scattering on a surface:

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega L_i(x, \omega_i) f(\omega_i, x, \omega_o) \cos(\theta_i) \, \mathrm{d}\omega_i \quad (1)$$

In this equation, the L_i and L_o terms represent the incoming radiance and outgoing radiance at point x in direction ω_i or ω_o, respectively. These two terms are computed by the rendering algorithm. $L_e(x, \omega_o)$ is the emitted radiance at point x in direction ω_o, and is non-zero for lights. Finally, f is the BSDF and θ_i is the angle between the surface normal at x and the direction ω_i.

Different models can be used to represent f, including analytical models or measured data. In practice, the artist usually designs a material by combining predefined models whose inputs are controlled by textures or mesh attributes.

The rendering algorithm may need to perform several operations on a BSDF: Sampling, evaluation, and querying for special properties that may be useful for the rendering algorithm. In the case of Path Tracing (PT), the algorithm needs to sample the BSDF to create new directions at every hit point, and it needs to evaluate the BSDF in order to perform Next Event Estimation (NEE). Additionally, if the algorithm uses Multiple Importance Sampling (MIS) [Veach and Guibas 1995], the renderer should also be able to query the sampling probability for a given direction. Finally, as a small optimization, NEE can be disabled for purely specular materials, as it is not useful in this case.

Rodent defines a BSDF like so:

```
struct Bsdf {
  eval:        fn (Vec3, Vec3) -> Color,
  pdf:         fn (Vec3, Vec3) -> f32,
  sample:      fn (&mut RndState, Vec3) -> BsdfSample,
  is_specular: bool
}
```

The `sample` function takes a random number generator state, an incoming direction, and returns a `BsdfSample` containing the sample value, direction, sampling probability, and the cosine between the surface normal and the sample:

```
struct BsdfSample {
  color: Color, in_dir: Vec3,
  pdf:   f32,   cos:    f32,
}
```

The simplest analytical BSDF model is the one for a purely diffuse BSDF. A constructor for this model might look like this:

```
fn @make_diffuse_bsdf(elem: SurfaceElement, kd: Color) -> Bsdf {
  Bsdf {
    eval: @ |in_dir, out_dir| kd * (1.0f / pi),

    pdf:  @ |in_dir, out_dir|
      cosine_hemisphere_pdf(positive_cos(in_dir, elem.normal)),

    sample: @ |rnd, out_dir| {
      let (u, v) = (randf(rnd), randf(rnd));
      let sample = sample_cosine_hemisphere(u, v);
      BsdfSample {
        color:  kd * (1.0f / flt_pi),
        in_dir: mat3x3_mul(elem.local, sample.dir),
        pdf:    sample.pdf,
        cos:    sample.dir.z
      }
    },

    is_specular: false
  }
}
```

This diffuse BSDF is rather standard: Its value is constant and equal to `kd`/π, so that the values in `kd` are in the interval $[0, 1]$. It uses cosine-weighted hemisphere sampling, provided by the two functions `cosine_hemisphere_pdf` and `sample_cosine_hemisphere`, which evaluate the sampling probability of cosine sampling and perform sampling, respectively. The function `sample_cosine_hemisphere` returns a sample on the unit hemisphere whose normal is the Z-axis, and must thus be transformed into the local coordinate system of the surface element. As a consequence, the cosine between the normal and the direction is simply the Z-coordinate of the sampled direction. Note the parameter `elem` of type `SurfaceElement`: It contains information about the current point on the surface (see Section 4.4).

A material can be viewed as a combination of a BSDF with an emitter, if any. This matches quite well the right-hand side of the rendering equation (see Equation (1)), representing both the L_e (emitted radiance) and f (the BSDF) terms.

```
struct Material {
  bsdf:        Bsdf,
  emission:    fn (Vec3) -> EmissionValue,
  is_emissive: bool,
}
```

This `Material` contains a BSDF and an emission profile. If the material does not emit light, the `is_emissive` flag is set to **`false`**. This flag is a way to optimize the renderer when the emission profile is *not* known: The renderer does not need to evaluate the emission value and test every component of the returned color at runtime, but only perform a boolean check. The value returned by the `emission` function contain the intensity, and probabilities for a given direction:

```
struct EmissionValue {
  intensity: Color,
  pdf_area:  f32,
  pdf_dir:   f32,
}
```

The `pdf_area` member represents the probability of sampling the current point on the surface, and the `pdf_dir` member corresponds to the probability of sampling the direction using emission sampling (sampling from the light source). These terms are necessary for bidirectional algorithms and MIS. Rodent defines a simple non-emissive material as follows:

```
fn @make_material(bsdf: Bsdf) -> Material {
  Material {
    bsdf: bsdf,
    emission: @ |dir| EmissionValue {
      intensity: black,
      pdf_area:  1.0f,
      pdf_dir:   1.0f
    },
    is_emissive: false
  }
}
```

Note that the emission probabilities are irrelevant if the emission profile is always black. Setting them to 1 lets the compiler optimize away any division or multiplication if the programmer forgot to check the `is_emissive` flag.

4.3 Lights

Light sources are usually kept distinct from other surfaces, in order to make NEE possible. Depending on the rendering algorithm, it might be necessary to sample them and produce a position and direction on the light source, as is for instance the case in Photon Mapping (PM) or any algorithm which includes tracing paths from the light sources. Furthermore, lights may not have an area and may be handled differently by the integrator: This is the case for point light sources, for instance.

In order to support these features, an adequate Impala definition could be:

```
struct Light {
  sample_direct:   fn (&mut RndState, Vec3) -> DirectLightSample,
  sample_emission: fn (&mut RndState) -> EmissionSample,
  emission:        fn (Vec3, Vec2) -> EmissionValue,
  has_area:        bool,
}
```

Direct emission sampling is done during NEE: The integrator invokes `sample_direct` with a random number generator state and a point on a surface, and in return gets a sample containing a position on the light source, the light intensity and cosine on the light source for the corresponding light direction, and a pair of probabilities:

```
struct DirectLightSample {
  pos:       Vec3,
  intensity: Color,
  cos:       f32,
  pdf_area:  f32,
  pdf_dir:   f32,
}
```

The probability `pdf_area` represents the probability of sampling the point on the surface of the light source. The other probability `pdf_dir` represents the probability of sampling the direction between the point on the surface and the point on the light source using direction sampling.

The `sample_emission` function generates a point and a direction (along with the probabilities) from a random number generator state:

```
struct EmissionSample {
  pos:       Vec3,
  dir:       Vec3,
  intensity: Color,
  cos:       f32,
  pdf_area:  f32,
  pdf_dir:   f32
}
```

This structure is mostly identical to `DirectLightSample` except that it additionally contains a direction. The probabilities are computed in a different way, though: When calling `sample_direct`, the `pdf_dir` member of the returned `DirectLightSample` value is the probability of sampling the direction between the given surface point and the sampled point on the light source, as if it had been sampled using emission sampling. When calling `sample_emission`, the probability is the actual probability of sampling the returned direction.

As an example, a point light source could be implemented by connecting the point on the surface to the light position to create a direction in `sample_direct`, and sampling a direction uniformly around a sphere centered on the light position in `sample_emission`.

4.4 Geometric Objects

Inside a rendering algorithm, geometric objects are viewed as proxies to the surface information (e.g. position on surface, normal, geometric normal, …), and are associated with a *shader*. With this in mind, Rodent represents geometric objects like this:

```
struct Geometry {
  surface_element: fn (Ray, Hit) -> SurfaceElement,
  shader: Shader
}
```

The `SurfaceElement` structure represents the geometry at the hit point. The renderer invokes `surface_element` on the geometry with the current ray and hit information to obtain surface information at the hit point. Rodent passes this information to the shader contained in the geometry, along with the ray and hit, in order to produce a `Material`. Note that the shader is an ordinary Impala function, and not a shader program written in another shading language.

4.5 Shaders

Shaders create materials by combining textures, BSDFs, and lights. Since the input of a shader is the current hit point information, it follows that they should be typed as:

```
type Shader = fn (Ray, Hit, SurfaceElement) -> Material;
```

Unlike in libraries like OptiX, these shaders are only describing a material and do not have to return a new ray to continue the path: Only the renderer decides how paths are traced and defined. As an example, here is a physically-corrected *Phong* shader [Dutre et al. 2001] controlled by a texture:

```
let image = device.load_image("data/textures/wall.png");
fn phong_like_shader( ray: Ray
                    , hit: Hit
                    , elem: SurfaceElement
                    ) -> Material {
  let tex = make_texture(
    make_repeat_border(),
    make_bilinear_filter(),
    image);
  let diffuse = make_diffuse_bsdf(elem, tex(elem.uv_coords));
  let ks = make_color(0.9f, 0.9f, 0.9f);
  let ns = 150f;
  let specular = make_phong_bsdf(elem, ks, ns);
  let bsdf = make_mix_bsdf(diffuse, specular, 0.5f);
  make_material(bsdf)
}
```

This code uses the `device` (see Section 4.7) to load a texture, and then creates a material from the combination of two BSDFs, one being diffuse and the other specular. The diffuse component is controlled via a texture with *repeat* border handling mode and *bilinear* filtering. In this example, the texture data comes from an image loaded into device memory.

4.6 Renderers

Monte-Carlo renderers need to trace paths and evaluate the rendering equation (1) at every hit point. Textbooks on ray tracing [Glassner 1989, e.g.], usually describe this process in several key steps (see Figure 1): The renderer first emits a ray from the camera or light source and then enters a loop. This loop first traces this ray. If there is no intersection, the path is done and the renderer proceeds with the next sample. Otherwise, the renderer takes the emission of the surface (if any) into account. Then, if NEE is required, the renderer traces a new shadow ray towards a light source. Finally, either the renderer generates a continuation ray or the path terminates. To reflect this idea, renderers in Rodent use the following type to generate a path:

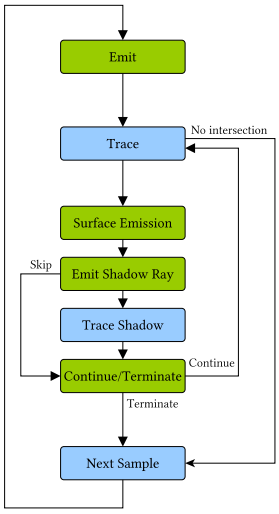

Fig. 1. Functional representation of the rendering process in Rodent. Green nodes are described in the `Tracer` structure.

```
struct Tracer {
  on_emit:   OnEmitFn,   on_hit:    OnHitFn,
  on_shadow: OnShadowFn, on_bounce: OnBounceFn,
}
```

This structure contains four members functions that define where to emit rays, what to do when a surface is hit, where to trace shadow rays, and whether the path should be continued or not. They directly correspond to nodes of the rendering loop above (colored in green in Figure 1), allowing a direct translation from a textbook algorithm into code.

The type of those functions contains *continuations*: For instance, `on_shadow` either traces a shadow ray, or simply does nothing depending on the type of surface (purely specular surfaces, for example, do not benefit from NEE). To encode this behaviour, the `OnShadowFn` type is defined as follows:

```
type OnShadowFn =
  fn (Ray, Hit, &mut RayState, SurfaceElement, Material,
      fn (Ray, Color) -> !, fn () -> !) -> !;
```

Read this type as: "A function that takes a ray, a hit, a ray state, a surface element, and a material. It either returns a ray and a color, or nothing". The other member types `OnEmitFn`, `OnBounceFn`, and `OnHitFn` use the same idea.

With the information contained in the `Tracer` structure combined with the scene information, the `trace` function in the device (see Section 4.7) schedules shading and tracing rays in order to maximize device utilization and parallelism. Note that the `Tracer` definition allows the user of the library to implement bidirectional algorithms as well, by calling `trace` twice: Once with a `Tracer` that emits rays from the camera, and once with one that emits rays from light sources.

4.7 Rendering Devices

In Rodent, the concept of a device abstracts the hardware architecture and scheduling mechanisms. It also contains members to load buffers or images into device memory (as seen in Section 4.5).

```
struct Device {
  trace:      fn (Scene, Tracer) -> (),
  load_image: fn (&[u8]) -> Image,
  // ...
}
```

The first parameter of the `trace` function is a `Scene` containing geometric objects, shaders, lights, and camera. This function traces paths according to the description of the ray tracing algorithm provided as second argument (see Section 4.6).

Thanks to this library design, each rendering algorithm uses the same `Tracer` abstraction, and the `Device` takes care of scheduling the tracing and shading functions. They also hold device-specific data for the scene, such as images or acceleration data structures. We have currently implemented three devices:

(1) A CPU device that uses vectorization in conjunction with counting sort to shade rays.
(2) A GPU device that traces and shades rays in a *megakernel*.
(3) A GPU device that runs as a *wavefront* [Laine et al. 2013].

All those devices perform shader specialization: They either incorporate some dispatch mechanism based on the material index (as in the megakernel GPU device), or sort rays by material and process contiguous ranges of rays (as in the CPU and wavefront GPU devices).

As an example, in the `trace` function of the CPU device, Rodent renders tiles of the image in parallel. For each of these tiles, Rodent maintains a stream (wavefront) of rays that it traverses and shades together. Before shading, Rodent sorts rays by increasing geometry ID—and thus, also by shader—and processes ranges of rays within the stream with the same shader:

```
fn trace(scene: Scene, tracer: Tracer) -> () {
  // ...
  for geom_id in unroll(0, scene.num_geometries) {
    // Get the range of rays that hit this geometry
    let (begin, end) = geometry_range(geom_id);
    for i in vectorize(vector_width, begin, end) {
      // Use tracer.on_hit/on_bounce/...
    }
  }
  // ...
}
```

In this excerpt, we assume that the rays are already traced and sorted, and iterate over geometry indices. For each geometry index, we get the corresponding range of rays in the stream (those that hit the geometry which has the current index), and loop over it using the **`vectorize`** built-in function.

The `unroll` iteration function—not built-in, but implemented in Impala—unrolls the loop body, thus triggering partial evaluation: If the scene contains n geometric objects, the outer loop will be unrolled n times, generating different vectorized versions of the loop body for every geometry. Because the loop body is specialized for a specific geometry and shader, unused attributes in the surface element and computations that depend on the type of material are automatically removed by the compiler.

On the wavefront GPU device, the idea is similar but the ray streams correspond to the entire image and the inner loop uses **`cuda`** instead of **`vectorize`**, thus creating different kernels for every shader.

The megakernel GPU device does not use streams. We generate a ray that is local to the current execution thread, and implement the rendering loop *inside* the compute kernel:

```
fn trace(scene: Scene, tracer: Tracer) -> () {
  for work_item in cuda(grid, block) {
    let (x, y) = (work_item.gidx(), work_item.gidy());
    let (ray, state) = tracer.on_emit(x, y);
    let mut terminated = false;
    while !terminated {
      // Intersect, shade and continue path
    }
  }
}
```

In this extract, the kernel contains the entire path tracing loop.

Typical megakernel GPU renderers often exhibit a high register pressure and important execution divergence. Because we do not enforce any particular schedule on the functions of the `Tracer` structure, we can choose the most appropriate schedule for each device. For instance, the megakernel device schedules operations such that small shaders are fused together, but the CPU device does not, as this optimization is detrimental to its performance.

5 RESULTS

In this section, we compare Rodent with existing, state-of-the-art tools and libraries to write renderers. We measure both the performance and code complexity of every approach across a set of real scenes modeled by independent artists.

5.1 Generating Renderers In Practice

Since Rodent expects a scene description written in Impala, we have written a small converter from a 3D scene file to Rodent's scene representation. In order to generate a renderer, we only need to run the converter on a scene of our choice, choose a device (e.g. by inserting a line like **`let`** `device = make_cpu_device()` in the converted scene), and then compile the result using Impala. The final result is a renderer specialized for that scene that runs on the chosen device. A real application can embed the Impala compiler and convert a scene at run-time, using Just-In-Time compilation.

5.2 Reference Renderers

Embree [Wald et al. 2014] and OptiX [Parker et al. 2010] offer low-level APIs to design renderers. In order to compare them against Rodent, we have implemented renderers with both Embree as well as OptiX while following the examples provided in their respective documentation (the Embree example renderer[1] and the OptiX path tracer). These renderers only support triangle meshes of a certain type and a small set of predefined shaders corresponding to the set of possible materials in our tested scenes. Moreover, our Embree implementation uses `ispc` [Pharr and Mark 2012] to vectorize shading code. Acceleration data structures performance hints suggested in the documentation have been applied for both renderer implementations (e.g. using the SBVH data structure in OptiX). We believe this comparison is fair: We have added the source code of those

[1] see https://embree.github.io/renderer.html

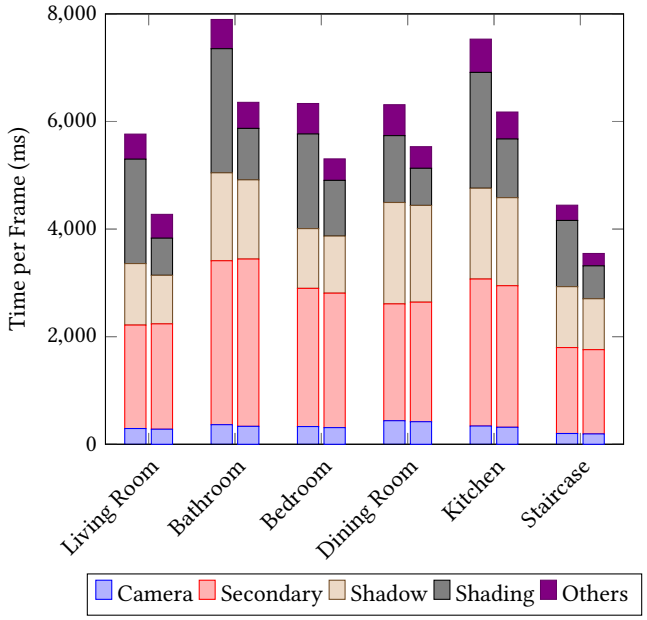

Fig. 2. Average distribution of the execution time for renderers generated by Rodent (right) and the Embree-based reference (left) over all tested scenes on an i7 6700K. We measure time tracing camera/secondary/shadow rays, and performing shading. Other tasks include generating, sorting or compacting rays.

two implementations to the supplemental material, along with the source code of Rodent.

5.3 Performance

The performance of the renderers generated by Rodent and the reference renderers is evaluated on a workstation with a i7-6700K CPU and a Titan X (Maxwell) GPU. The numbers for our test scenes are listed in Table 2, and the associated reference images are presented in Figure 6. This table provides additional numbers for an R9 Nano GPU, on which OptiX cannot run.

In all of the tested cases, our renderers outperform the reference implementations. We give a break down of the time spent in each part of the reference CPU renderer and Rodent (Figure 2). Since the two implementations use exactly the same BVH and traversal algorithm, the time spent tracing rays is similar between both renderers. However, shading is much faster in Rodent than in the reference. Indeed, vectorization is more efficient when rays are processed in a wavefront: They can be sorted by material and processed together [Áfra et al. 2016]. On top of this, Rodent's specialized renderers contain fewer branches than the Embree reference, thanks to compile-time specialization.

In order to explain which part of the performance difference is due to specialization, we implemented a synthetic benchmark that shades a batch of 4096 rays using a physically corrected Phong BSDF. Each ray in the batch is associated with an integer S in the range [0..3] that defines how the parameters kd, ks, and ns of the BSDF are chosen:

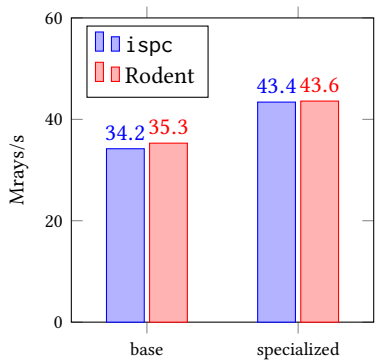

Fig. 3. Shading performance on an i7 6700K for the simple shader described in Section 5.3 implemented using Rodent and `ispc` (higher is better). Specialization is done manually in the `ispc` version.

- For $S = 0$, every parameter is a constant.
- For $S = 1$, kd is a texture, ks and ns are constants.
- For $S = 2$, ks is a texture, kd and ns are constants.
- For $S = 3$, ks and kd are textures, ns is a constant.

We implemented this shader using both Rodent and `ispc`, and tested the impact of specialization on performance. In the `ispc` version, specialization is a tedious process achieved using macros, and by duplicating some functions, in order to ensure that parameters remain uniform whenever possible. This goes against all good programming practices, by forcing the programmer to break existing abstractions: The interested reader can have a look at the code provided in the supplemental material for more details. With Rodent, specialization is done transparently by the compiler, when the number of shaders is known. In this case, the generated executable contains 4 different shaders, one for each value of S. Additionally, we do not have to write annotations to specify whether a variable or parameter is uniform or not: The compiler infers the best possible annotation for us [Moll and Hack 2018]. The results in Figure 3 show that specialization brings a performance improvement of around 25% in both versions. This confirms the intuition that specialization is, even in such a simple example with only 4 shaders, an efficient optimization technique.

5.4 Cross-Layer Optimizations

Another interesting question is whether cross-layer optimizations, which happen between module or abstraction interfaces, are necessary to achieve performance. To answer this, we designed another synthetic benchmark where a simple diffuse shader is run on a batch of rays. This time, we enabled or disabled specialization for two different interfaces: the one between the texture and the shader, and the one between the mesh attributes and the shader. An easy way to do this is to add a boolean parameter to every function that specifies whether or not to perform specialization, as in the following example:

```
fn @(specialize) f(specialize: bool, /* ... */) { /* ... */ }
```

Using this trick, the call `f(true, /*...*/)` will trigger specialization, but `f(false, /*...*/)` will not. The results of this benchmark, shown in Figure 4, demonstrate that specializing interfaces is a clear

Table 2. Performance comparison between the renderers generated by Rodent and the reference renderers (not available for the R9 Nano) in Msamples/s (higher is better). On the GPUs, we list the numbers for both the *megakernel* device (1) and the *wavefront* device (2).

	CPU (i7 6700K)		GPU (Titan X)			GPU (R9 Nano)	
Scene	Rodent	Embree	Rodent[1]	Rodent[2]	OptiX	Rodent[1]	Rodent[2]
Living Room	9.77 (+23%)	7.94	38.59 (+25%)	43.52 (+42%)	30.75	24.87	35.11
Bathroom	6.65 (+13%)	5.90	27.06 (+31%)	35.32 (+42%)	20.64	14.95	27.31
Bedroom	7.55 (+ 4%)	7.24	30.25 (+ 9%)	38.88 (+29%)	27.72	19.25	32.90
Dining Room	7.08 (+ 1%)	7.01	30.07 (+ 5%)	40.37 (+29%)	28.58	16.22	30.83
Kitchen	6.64 (+12%)	5.92	22.73 (+ 2%)	32.09 (+31%)	22.22	16.68	28.13
Staircase	4.86 (+ 8%)	4.48	20.00 (+18%)	27.53 (+39%)	16.89	11.74	22.21

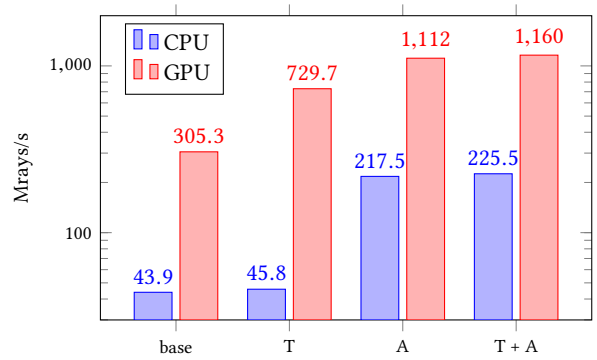

Fig. 4. Performance when specializing the interfaces between the shader and the texturing function (T), or between the shader and attribute computation (A). We present the results for an i7 6700K CPU, using all cores, and on a Titan X GPU. Note the logarithmic scale on the vertical axis.

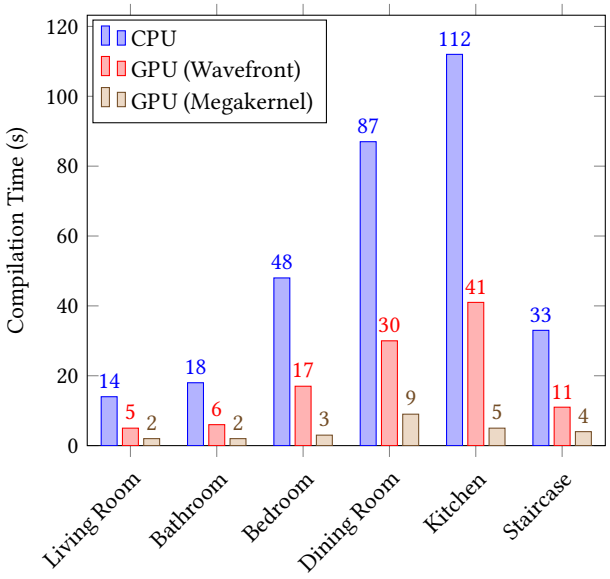

Fig. 5. Compilation times for various scenes and rendering devices.

win in this scenario. In general, users *may not* use every computed value provided by an interface. In the event that these computations are left unused, specialization of the interface can bring large performance improvements. This is typically the case for the interface between the shading system and the rendering system, for example.

Table 3. Implementation effort for Rodent's core rendering library, the CPU device, the GPU devices (including megakernel *and* wavefront), and for the Embree-based and OptiX-based references. The effort is measured using Halstead's metric (see Section 5.5).

	Rodent				
	Core	CPU	GPU	Embree	OptiX
LoC	1086	471	600	869	930
Effort	3.798M	4.403M	5.724M	7.856M	9.605M
		13.93M		17.46M	

5.5 Implementation Effort

Implementation effort can be measured in number of lines of code (LoC), but different languages may require a different amount of lines to express the same intention. A much better approximation of the effort required to write a piece of code is Halstead's well-known software complexity measure [Halstead 1977], based on the number of operators and operands in the program. We list Halstead's effort along with numbers of LoC in Table 3. Note that we only include the relevant parts of each renderer, so that the comparison is fair: For instance, the scene loader has been omitted for both reference renderers. The numbers are separated between the core library and the devices for a better comparison with the reference renderers.

Even though Rodent is more generic, has more features, and allows scenes that are not supported by the reference renderers—we even provide a wavefront-based GPU implementation not provided by OptiX—the effort required to code Rodent and all its devices is lower than the effort required to write the two references.

It is also worth noting that every renderer written with Rodent uses the same core concepts. For the reference renderers, everything had to be reimplemented from scratch: Even though both CUDA and `ispc` use a C-based language, they are too different—syntax-wise and semantics-wise—to allow sharing parts of the implementation.

Finally, remember that none of our references uses specialization. The only tools available with `ispc` or the CUDA compiler are macros and template metaprogramming. These are completely impractical in our setting: Dynamic (run-time) and static (compile-time) data should be represented in the same language because the scene may be only *partially* known.

5.6 Compilation and Specialization Time

Since we produce a new renderer for every scene, the time taken to *generate* a renderer is also important, depending on the use case. Over all the tested scenes, the Impala compiler takes between tens of seconds to 2min, depending on the number of shaders in the

(a) Living Room (b) Bathroom

(c) Bedroom (d) Dining Room

(e) Kitchen (f) Staircase

Fig. 6. Scenes used in the performance evaluation.

scene, and the device used (see Figure 5). In particular, the CPU device takes the longest to compile due to the larger code base and code transformations triggered by calls to `vectorize`. Additionally, the Impala compiler is not particularly well-optimized itself, and is only single-threaded. As a consequence, the approach presented in the result section is only valid when render times dominate.

Nevertheless, our library does not forbid partial specialization of a scene: Indeed, we already do not specialize camera parameters, so that the user can move interactively within the scene. If the set of shaders that are going to be used is known in advance, the renderer can be specialized with only that knowledge, and can therefore be reused without recompilation for all scenes sharing that property.

6 CONCLUSION

Rodent defines every component of a renderer individually using a functional and declarative API. The entire library is written in one unified, general purpose language; There is neither a dedicated shading language nor a shader specialization engine. The rendering components—*abstractions*—of the library are composable and do not contain any scheduling or execution decision. Such decisions are left to another layer of the library that combines every component to form the final renderer.

This layer takes advantage of the compile-time knowledge of the scene to optimize the rendering abstractions away, and choosing the most appropriate execution schedule in order to maximize performance. Instead of implementing a custom compiler for renderers, this layer uses partial evaluation, a well-studied compilation technique, to specialize computations. The partial evaluator itself is driven using simple code annotations that specify when to inline a function: Most of these annotations are trivially forcing code execution, and require little knowledge of the compiler technology.

Using this strategy, Rodent generates renderers that are specialized for both a particular scene and a specific target hardware. The performance evaluation shows that the generated renderers are

faster and require less effort than reference renderers written with state-of-the-art libraries designed by hardware vendors.

Future work could focus on extensions of the library to define more advanced renderers than the ones described in the article. There are also avenues for research in the partial evaluator itself, since the compilation times currently prevent changing the appearance of objects interactively. The entire source code of this paper is released as Open Source and is available on GitHub[2].

ACKNOWLEDGMENTS

This work is supported by the Federal Ministry of Education and Research (BMBF) as part of the Metacca and ProThOS projects as well as by the Intel Visual Computing Institute (IVCI) and Cluster of Excellence on Multimodal Computing and Interaction (MMCI) at Saarland University.

The tested scenes are freely available on Blendswap, and are licensed under the terms of the CC-BY 3.0, unless noted otherwise. The Living Room scene is from Blendswap user Wig42 © 2014. The Bathroom scene is from Blendswap user nacimus © 2014. The Staircase scene is from Blendswap user Wig42 © 2015. The Bedroom scene is from Blendswap user SlykDrako © 2011, licensed under CC0 1.0. The Dining Room scene is from Blendswap user MaTTeSr © 2016. The Kitchen scene is from Blendswap user Jay-Artist © 2012. Some assets where adapted from Benedikt Bitterli's rendering resources [Bitterli 2016].

REFERENCES

Attila T. Áfra, Carsten Benthin, Ingo Wald, and Jacob Munkberg. 2016. Local Shading Coherence Extraction for SIMD-efficient Path Tracing on CPUs. In *Proceedings of High Performance Graphics*. Eurographics Association, 119–128. https://doi.org/10.2312/hpg.20161198

P.H. Andersen. 1995. *Partial Evaluation Applied to Ray Tracing*. DIKU Research Report 95/2. DIKU.

Kenichi Asai. 2002. *Can Partial Evaluation Improve the Performance of Ray Tracing?* Technical Report Vol. 53, Nr. 1. Ochanomizu University. 97–100.

Benedikt Bitterli. 2016. Rendering resources. https://benedikt-bitterli.me/resources/

Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. 2007. Finally Tagless, Partially Evaluated. In *Programming Languages and Systems, 5th Asian Symposium, APLAS 2007, Singapore, November 29-December 1, 2007, Proceedings*. 222–238. https://doi.org/10.1007/978-3-540-76637-7_15

Charles Consel. 1988. New Insights into Partial Evaluation: the SCHISM Experiment. In *ESOP '88, 2nd European Symposium on Programming, Nancy, France, March 21-24, 1988, Proceedings*. 236–246. https://doi.org/10.1007/3-540-19027-9_16

J. Döllner and K. Hinrichs. 2002. A Generic Rendering System. *IEEE Transactions on Visualization and Computer Graphics* 8, 2 (2002), 99–118. https://doi.org/10.1109/2945.998664

Philip Dutre, Paul Heckbert, Vincent Ma, Fabio Pellacini, Robert Porschka, Mahesh Ramasubramanian, Cyril Soler, and Greg Ward. 2001. Global Illumination Compendium.

Yoshihiko Futamura. 1982. Parital Computation of Programs. In *RIMS Symposium on Software Science and Engineering, Kyoto, Japan, 1982, Proceedings*. 1–35. https://doi.org/10.1007/3-540-11980-9_13

Iliyan Georgiev and Philipp Slusallek. 2008. RTfact: Generic Concepts for Flexible and High Performance Ray Tracing. In *Proceedings of the IEEE Symposium on Interactive Ray Tracing (RT)*. IEEE, 115–122. https://doi.org/10.1109/RT.2008.4634631

Andrew S. Glassner (Ed.). 1989. *An Introduction to Ray Tracing*. Academic Press Ltd., London, UK, UK.

Brian Guenter, Todd B. Knoblock, and Erik Ruf. 1995. Specializing Shaders. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '95)*. ACM, New York, NY, USA, 343–350. https://doi.org/10.1145/218380.218470

Maurice H. Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series)*. Elsevier Science Inc., New York, NY, USA.

Yong He, Kayvon Fatahalian, and Tim Foley. 2018. Slang: Language Mechanisms for Extensible Real-time Shading Systems. *ACM Trans. Graph.* 37, 4, Article 141 (2018), 13 pages. https://doi.org/10.1145/3197517.3201380

Neil D. Jones. 1996. An Introduction to Partial Evaluation. *ACM Comput. Surv.* 28, 3 (1996), 480–503. https://doi.org/10.1145/243439.243447

James T. Kajiya. 1986. The Rendering Equation. *SIGGRAPH Comput. Graph.* 20, 4 (1986), 143–150. https://doi.org/10.1145/15886.15902

Samuli Laine, Tero Karras, and Timo Aila. 2013. Megakernels Considered Harmful: Wavefront Path Tracing on GPUs. In *Proceedings of the 5th High-Performance Graphics Conference (HPG)*. ACM, 137–143. https://doi.org/10.1145/2492045.2492060

Roland Leißa, Klaas Boesche, Sebastian Hack, Arsène Pérard-Gayot, Richard Membarth, Philipp Slusallek, André Müller, and Bertil Schmidt. 2018. AnyDSL: A Partial Evaluation Framework for Programming High-Performance Libraries. *PACMPL* 2, OOPSLA (2018), 119:1–119:30. https://doi.org/10.1145/3276489

Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. 2002. Shader Metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '02)*. Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 57–68.

Simon Moll and Sebastian Hack. 2018. Partial Control-Flow Linearization. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 543–556. https://doi.org/10.1145/3192366.3192413

NVIDIA. 2019. CUDA Toolkit Documentation. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, visited 10/01/19.

Steven G. Parker, James Bigler, Andreas Dietrich, Heiko Friedrich, Jared Hoberock, David Luebke, David McAllister, Morgan McGuire, Keith Morley, Austin Robison, and Martin Stich. 2010. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics (TOG)* 29, 4, Article 66 (July 2010), 13 pages. https://doi.org/10.1145/1778765.1778803

Matt Pharr and William R. Mark. 2012. ispc: A SPMD Compiler for High-Performance CPU Programming. In *2012 Innovative Parallel Computing (InPar)*. IEEE, 1–13. https://doi.org/10.1109/InPar.2012.6339601

Pixar 1988. *RenderMan Interface Specification*. Pixar. version 3.0.

Arsène Pérard-Gayot, Martin Weier, Richard Membarth, Philipp Slusallek, Roland Leißa, and Sebastian Hack. 2017. RaTrace: Simple and Efficient Abstractions for BVH Ray Traversal Algorithms. In *Proceedings of the 16th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 157–168. https://doi.org/10.1145/3136040.3136044

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. *SIGPLAN Not.* 48, 6 (2013), 519–530. https://doi.org/10.1145/2499370.2462176

James Reinders. 2007. *Intel Threading Building Blocks - Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly. http://www.oreilly.com/catalog/9780596514808/index.html

Kai Selgrad, Alexander Lier, Franz Köferl, Marc Stamminger, and Daniel Lohmann. 2015. Lightweight, Generative Variant Exploration for High-performance Graphics Applications. In *Proceedings of the 14th International Conference on Generative Programming: Concepts & Experiences (GPCE)*. ACM, 141–150. https://doi.org/10.1145/2814204.2814220

Philipp Slusallek and Hans-Peter Seidel. 1995. Vision - An Architecture for Global Illumination Calculations. *IEEE Transactions on Visualization & Computer Graphics* 1 (1995), 77–96. https://doi.org/10.1109/2945.468387

Kristian Sons, Felix Klein, Jan Sutter, and Philipp Slusallek. 2014. shade.js: Adaptive Material Descriptions. *Computer Graphics Forum* 33, 7 (2014), 51–60. https://doi.org/10.1111/cgf.12473

Sony Pictures Imageworks 2017. *Open Shading Language Specification*. Sony Pictures Imageworks. version 1.9.

Eric Veach and Leonidas J. Guibas. 1995. Optimally Combining Sampling Techniques for Monte Carlo Rendering. In *Proceedings of the 22Nd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '95)*. ACM, New York, NY, USA, 419–428. https://doi.org/10.1145/218380.218498

Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, and Manfred Ernst. 2014. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics (TOG)* 33, 4, Article 143 (July 2014), 8 pages. https://doi.org/10.1145/2601097.2601199

Turner Whitted. 1980. An Improved Illumination Model for Shaded Display. *Commun. ACM* 23, 6 (1980), 343–349. https://doi.org/10.1145/358876.358882

Stefan Zellmann, Daniel Wickeroth, and Ulrich Lang. 2017. Visionaray: A Cross-Platform Ray Tracing Template Library. In *Proceedings of the 10th Workshop on Software Engineering and Architectures for Realtime Interactive Systems (IEEE SEARIS 2017)*. 1–8.

Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt: A High-Performance Graph DSL. *PACMPL* 2, OOPSLA (2018), 121:1–121:30. https://doi.org/10.1145/3276491

[2]see https://github.com/AnyDSL/rodent