Lightweight, Generative Variant Exploration for High-Performance Graphics Applications

Kai Selgrad Alexander Lier Franz Köferl Marc Stamminger Daniel Lohmann
Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
{kai.selgrad, alexander.lier, franz.koerferl, marc.stamminger, daniel.lohmann}@fau.de

Abstract

Rendering performance is an everlasting goal of computer graphics and significant driver for advances in both, hardware architecture and algorithms. Thereby, it has become possible to apply advanced computer graphics technology even in low-cost embedded appliances, such as car instruments. Yet, to come up with an efficient implementation, developers have to put enormous efforts into hardware/problem-specific tailoring, fine-tuning, and domain exploration, which requires profound expert knowledge. If a good solution has been found, there is a high probability that it does not work as well with other architectures or even the next hardware generation.

Generative DSL-based approaches could mitigate these efforts and provide for an efficient exploration of algorithmic variants and hardware-specific tuning ideas. However, in vertically organized industries, such as automotive, suppliers are reluctant to introduce these techniques as they fear loss of control, high introduction costs, and additional constraints imposed by the OEM with respect to software and tool-chain certification. Moreover, suppliers do not want to share their generic solutions with the OEM, but only concrete instances.

To this end, we propose a light-weight and incremental approach for meta programming of graphics applications. Our approach relies on an existing formulation of C-like languages that is amenable to meta programming, which we extend to become a lightweight language to combine algorithmic features. Our method provides a concise notation for meta programs and generates easily sharable output in the appropriate C-style target language.

Categories and Subject Descriptors Programming Languages [*Processors*]: compilers, optimization, code generation; Software Engineering [*Design Tools and Techniques*]: evolutionary prototyping; Computer Graphics [*Three-Dimensional Graphics and Realism*]: Raytracing;

General Terms algorithms, experimentation, languages, performance

Keywords exploratory programming, general purpose code generation, prototyping, ray tracing

1. Introduction

Thanks to the rise of dedicated graphics processing units (GPUs) and the ongoing extension of CPUs with ever more sophisticated vector instructions (such as AVX, AltiVec, NEON), computationally intensive graphics has become available not only on every PC, but even on the average cell phone. In the near future we will see an increasing application of 3D rendering for user interaction even in cost-sensitive embedded control systems, such as household appliances or cars.

An example of this trend is the Virtual Cockpit of the upcoming Audi TT model, where the complete instrument cluster (incl. speedometer, rev counter, maps, etc.) is just "virtual", that is, rendered in high quality and with configurable skins and layouts on a dashboard-shaped display.[1]

A classical, yet rising problem for computer graphics engineers is the constantly increasing heterogeneity of hardware platforms: Despite existing abstractions (such as CUDA, OpenGL, or GLSL): To deliver optimal performance, graphics engineers still have to work on a rather low level, as they need to tailor their algorithms and implementation specifically to the problem and the employed hardware. This goes down to the level of working around compiler issues or revision-dependent silicon bugs. In practice, this requires profound expert knowledge and high efforts for domain exploration: Performance-sensitive parts are developed incrementally by implementing and testing numerous combinations of algorithms and implementation tricks (such as using or prohibiting certain instructions) in order to find a solution that performs well for a concrete problem/architecture combination. If such a solution has been found, it, however, commonly does not last for long – in most cases it is not transferable to another architecture. In this realm, recent DSL-based approaches for graphics applications, such as HIPAcc [37] or Halide [44] provide promising means for dealing with hardware heterogeneity on a more abstract level, even though these approaches target the intermediate developer, who does neither want nor need full control regarding low-level problem/architecture-specific optimizations.

However, especially in the domain of embedded control systems (such as, automotive applications), engineers are reluctant to employ high-level DSL approaches also because of other nonfunctional constraints that require them to have much more fine-grained control over the resulting code: In most cases, these systems are developed by external suppliers for some OEM, such as an automotive vendor, who performs the final integration into the end product and also holds

[1] http://www.audi.com/com/brand/en/vorsprung_durch_technik/content/2014/03/audi-virtual-cockpit.html

the liability. Certifiability of software is a big issue, which in practice leads to mandatory coding styles and defined tool chains with old, but proven-in-use compilers. Software components are generally delivered as source code and finally compiled at the OEM site, so engineers on the supplier side cannot easily integrate new DSLs and compilation tools. Moreover, suppliers generally do not want to share a generic DSL-based solutions with the OEM, but only the concrete instance they get paid for, delivered as source-code in an OEM-specified C-type language (C, C++, CUDA, etc.). In a different context, researching new approaches and algorithm-combinations benefits from systems that target the domain-expert, instead of the intermediate user, by providing abstractions while keeping full control over the generated instructions.

About this Paper. In this setting, we advocate for C-Mera as a transparent, light-weight, and incrementally adoptable basis towards meta programming that provides a high level of control over the generated source code in C-type languages. C-Mera, which we have already presented in a previous workshop paper [49], is a simple source-to-source compiler that transforms a notation based on S-Expressions [34] (sexp) for C-type languages to the native syntax of that language, that is, for example, from sexp-C to C, and from sexp-CUDA to CUDA. In this paper, we show how it can be applied to come up with a light-weight and simple, yet powerful feature composition framework for implementation-domain exploration in performance-sensitive computer graphics applications.

The proposed style of meta programming is *transparent*, as the semantics of the sexp-based notation is identical to that of the native language and the output is fully controllable, including all aspects of source-code formatting, if necessary. It is *light-weight*, as it requires only little tool support. It provides *incremental adoption*, as the sexp-based notation can easily be mixed with existing code in the native language. Furthermore, it involves only little adoption risks, as it is possible to quit and continue with the native language at any stage of the development process. For instance, C-Mera can be used for domain exploration of algorithmic variants only, whereas the outcome of this exploration is then passed as C++ or CUDA code to the next engineer for further architecture-specific fine tuning or to the OEM.

The remainder of this paper is structured as follows: In Section 2 we give a brief introduction into the ray-tracing domain as one subfield of performance-critical computer graphics to provide readers a glimpse of the vast number of algorithmic variants and optimization approaches that are available (and need to be explored) for a particular application. In Section 3 we provide a brief introduction to C-Mera. This is followed by our main contributions in Section 4, a lightweight feature composition system, and its application to domain exploration for ray tracing in Section 5. We then discuss strengths and limitations of our system in the context of competing approaches and related work in Section 6, and conclude with Section 7.

2. Domain Analysis

One of the classic problems of computer graphics is to compute the intersection of a ray with a three dimensional scene, i.e. ray tracing. This has been an active area of research for almost half a century [5, 28]. As a result of this research there is a tremendous number of approaches to tackle this problem, both in hard- and software.

Primitives. Optimzied algorithms most commonly work on triangle primitives but other simple representations [21] as well as very high level surface [7, 47] descriptions have been and still are under active investigation. Such higher level descriptions can also be converted to triangular data by tesselation.

The ray tracing algorithm is usually employed to determine the part of the scene visible from the observer's point of view [21] as well as to compute the end-points of light paths during the simulation of light transport [15]. However, the algorithm can also be used, e.g., for collision detection [29].

Acceleration Structure. Even in the simple case, where only the geometry visible to the observer is to be computed, a large number of rays has to be traced (two million at 1080p), which does not yet include anti-aliasing. Furthermore, any scene of interest contains a vast number of geometric primitives (a few million are not uncommon). This geometric complexity is managed by the use of acceleration structures by which large parts of the scene can be rejected early. Examples include grids [3, 30] and tree structures such as bounding volume hierarchies [27], Octrees [20], kd-trees [25] and bounding interval hierarchies [56].

Generation of these acceleration structures is, too, an active topic of research, with different goals, e.g. producing hierarchies of very high quality [25, 33, 53], faster construction of quality hierarchies [57], construction of inferior hierarchies, but at interactive rates [26, 31], and finding data layouts that match hardware models [2, 13, 17], and preprocessing of the data to be stored [16].

Ray Traversal. A large part of the complexity is in the traversal of the respective structures, e.g. approaches tailored to exploit graphics hardware [1, 23] and modern CPU architectures [6, 13, 17, 61]. Traversal of hierarchies can also be implemented in a stack-less fashion [24, 42] and rays can be grouped into ray packets [23, 59], or not [60], both resulting in different trade-offs. Custom ray tracing hardware is also proposed [46, 48], including embedded solutions [32, 40].

Feature Set. Furthermore, if the application allows to restrict the set of features that have to be supported (potentially with different variants for individual passes) it is possible to improve traversal performance by finding any hit-point, instead of the closest, to choose a traversal order that exploits the problem's structure [18, 39], or to ignore transparency or alpha maps which makes it possible to ignore surface textures during primitive intersection. Also note different time constraints, e.g. algorithms suitable for interactive rendering [58] are not generally good choices in a movie production context [10].

Summary. The implementation of a production ray tracing system as well as research considering new approaches has to take into account a *vast* number of existing variants from which to choose, which is not a trivial task. Because of the holistic nature of the problem (caused by the tangled interaction between visual features, algorithmic representations, and the actually employed hardware architecture), it is generally not possible to determine image quality and performance of a given algorithm ahead of time. Thus, each solution has to be compared to many other implementations using different choices. In practice, this domain exploration is typically an ad-hoc process with lots of repetitive work and copy-and-paste. This is where comprehensive domain specific systems can help developers and vendors to both make well founded implementation choices and arrive at the best product possible.

In this paper, we investigate how the problem of exploring and managing variants of ray traversal algorithms can be tackled using a simple, existing meta-programming framework. The following section will give a short introduction to this system, and Section 5 will show how we apply it to construct a lightweight formalism that will be used for describing ray traversal variants.

3. C-Mera: Rewriting C

The C-Mera system [49] is a very simple compiler that transforms a notation based on S-Expressions [34] (sexp) for C-like languages to the native syntax of that language, e.g. from sexp-C to C, and from sexp-CUDA to CUDA. The semantics of the sexp-based notation is identical to that of the native language, i.e. no inherent abstraction or layering is introduced.

Using S-Expressions means that the program is syntactically written in form of nested lists, such as the following example of a CUDA kernel definition:

```
1  (define-cuda-kernel copy ((int N))
2    (if (< threadIdx.x N)
3        (set dst[threadIdx.x] src[threadIdx.x])))
```

This notation is simple and strictly uniform (nested lists where the first entry provides the content's interpretation), albeit somewhat unfamiliar to users accustomed to C. We chose to use it as it directly maps to a syntax tree: the first element of each list gives the type of the tree node and the following elements its sub-nodes. This is also the reason that prefix notation is used for arithmetic expressions.

As C-Mera itself is built as an embedded language in Common Lisp, we exploit its macro system to implement syntax tree transformations. This works by introducing special node types which will not itself be found in the compiler's output, but are invoked to restructure parts of the tree below it. As an example, in the kernel definition above we used a node called `define-cuda-kernel`. This node could be implemented to expand into a syntax tree that generates a `global void` function with the following macro:

```
1  (defmacro define-cuda-kernel (name args &body body)
2    `(function ,name ,args -> (__global__ void)
3       ,@body))
```

In its simplest form this is just a templating mechanism that, given the example above, yields

```
1  (function copy ((int N)) -> (__global__ void)
2    (if (< threadIdx.x N)
3        (set dst[threadIdx.x] src[threadIdx.x])))
```

which in turn generates the following CUDA code:

```
1  __global__ void copy(int N) {
2     if (threadIdx.x < N)
3         dst[threadIdx.x] = src[threadIdx.x];
4  }
```

In the remainder of this section we will present some more examples of C-Mera's notation and describe a few other simple transformations, while the following sections will show how thereby more sophisticated, yet lightweight meta-programming techniques can be applied in a graphics context while keeping full control over the generated code.

Figure 1 demonstrates the basic syntax used to write C-Mera code, and shows the code generated by it. It can be seen that, as long as no meta programming is used, the input syntax is trivially converted to the output code. The first function (lines 1-2) is a simple dot product of three-component vectors. It shows how function definitions are

```
1  (function dot ((vec3f a) (vec3f b)) -> float
2    (return (+ (* a.x b.x) (* a.y b.y) (* a.z b.z))))
3
4  (function dot ((vecNf a) (vecNf b)) -> float
5    (decl ((acc (* a[0] b[0])))
6      (for ((int i 1) (< i (min a.N b.N)) ++i)
7        (+= acc (* a[i] b[i])))
8      (return acc)))
```

⇓

```
1  float dot(vec3f a, vec3f b) {
2      return (a.x * b.x) + (a.y * b.y) + (a.z * b.z);
3  }
4
5  float dot(vecNf a, vecNf b) {
6      float acc = a[0] * b[0];
7      for (int i = 1; i < min(a.N, b.N); ++i)
8          acc += (a[i] * b[i]);
9      return acc;
10 }
```

Figure 1. Dot-product definitions written in C-Mera syntax, together with the generated output program.

structured, and even though on first glance the code looks quite different from C, at closer inspection the mapping is straightforward. The second example (lines 4-8) illustrates local variable declarations and control structure. Note how some shorthands (most notably increments and subscripts) have been carried over from C for use in simple expressions.

The most compelling property of using such a uniform notation is the ease with which it can be transformed. Figure 2 shows a simple example where we first define a macro to unroll a code fragment, and then use it to unroll the loops of a matrix multiplication. This example shows how seamlessly meta code in C-Mera interacts with target code (meta code is highlighted in red). It furthermore illustrates the impact of even small-scale meta programming: the four lines of macro code are used in the `mmult` function in place of for-loops to generate 64 lines of simple assignment statements (indicated at the bottom). No further language support was required for this transformation.[2]

Regarding broader applicability it should be noted that C-Mera supports transformations for C, C++, CUDA, and GLSL code and can easily integrate even nonstandard language extensions one often finds in the proprietary compilers for specific architectures or domains. This is already visible in Figure 1 as the generated code is not at all restricted to a single target platform. Furthermore, programs can be written to properly transform to a number of platforms by using macros. We shall show an example for this in Section 4.

To conclude: C-Mera is, at first, just another syntax to write C programs, which makes it accessible to ordinary programmers. Since mapping from input to output code is straightforward pragmatic solutions are easily possible in cases where the output must be exactly of a given form. However, programs written in C-Mera can easily be transformed by means of meta programming, i.e. with semantics cast into syntax sophisticated transformations become possible. Thereby, C-Mera paves a lightweight and unobtrusive path to the provisioning of generative abstractions.

[2] Of course, such simple transformation are for illustration only and could be equally well performed with many other techniques, such as Eigen's [22] expression templates, or are even built in into modern optimizing compilers. However, both are not necessarily available in our domain.

```
(defmacro unroll ((var start end) &body code)
  `(progn ,@(loop for i from start to end collect
              `(symbol-macrolet ((,var ,i))
                 ,@code))))

(macrolet ((access (obj row col)
             `(aref ,obj ,(+ (* col 4) row)))
           (mult (i) `(* (access lhs.a y ,i)
                         (access rhs.a ,i x)))
           (curr () `(access res.a x y)))
  (function mmult ((mat4 lhs) (mat4 rhs)) -> mat4
    (decl ((mat4 res))
      (unroll (x 0 3)
        (unroll (y 0 3)
          (set (curr) (mult 0))
          (unroll (i 1 3)
            (+= (curr) (mult i)))))
      (return res))))
```

⇓

```
mat4 matrix_mult(mat4 lhs, mat4 rhs) {
    mat4 res;
    res.a[0]  = (lhs.a[0] * rhs.a[0]);
    res.a[0] += (lhs.a[4] * rhs.a[1]);
    ...
    return res;
}
```

Figure 2. Top: Macro for unrolling code, similar to the for-loop syntax. Center: The macro is used multiple times to completely unroll the matrix multiplication. Bottom: Yielding 64 lines of straightforward assignments. Note how a few convenience macros (access, curr, mult) make for very concise code in the loop body. Meta code is highlighted.

4. Lightweight Feature Composition for Computer Graphics

To manage and explore the multitude of implementation options (cf. Section 2) we propose to use a method inspired by feature-oriented programming [4, 43], where we define a *feature* to be a conceptional, composable element of the problem domain. In computer graphics, the (efficient) implementation of features always depends on the employed *target* architecture, which we account for by target-specific feature derivatives.

4.1 Composition System

In the following, we introduce these basic concepts of our composition system by the example of a potentially parallel computation.

Features are introduced with

```
(define-feature feature-name (arguments-to-invocation))
```

which, in our example of a parallelizable process can be written as:

```
(define-feature parallel-fn
  (name args (i N) &body body))
```

This specifies that the syntax tree node `parallel-fn` can be used to define a function that runs the embedded code in parallel, where `i` will denote the iteration count and `N` the range. Without any details of the actual implementation, it can be used as follows:

```
(parallel-fn accum ((float *a) (float *b)) (idx N)
  (+= a[idx] b[idx]))
```

Here, a function of two arguments is parallelized, where the iteration count and limit are `idx` and `N`, respectively. The details of the parallelization do not clutter up the algorithm.

```
(implement parallel-fn (cuda-target)
  (let ((kernel (symbol-append 'kernel- name))
        (args+N (append args '((int ,N))))
        (actual (loop for arg in args
                      append (last (flatten arg)))))
    `(progn
       (function ,kernel ,args+N
           -> (__global__ void)
         (decl ((int ,i (+ (* blockIdx blockDim)
                           threadIdx)))
           (if (>= idx N)
               (return))
           ,@body))
       (function ,name ,args+N -> void
         (decl ((int thd 256)
                (int blk (+ (/ ,N thd)
                            (? (% ,N thd) 1 0))))
           (funcall ,kernel
                    (blk thd) ,@actual ,N))))))
```

Figure 3. Implementation for general parallelization of array operations that generates host (line 14) and device-side (line 17) code and properly manages border cases (lines 11-12 and 4-5) and forwarding of arguments (lines 4-5 and 19).

Implementation derivatives for such a feature (honoring different targets) can be defined by

```
(implement feature-name (targets) implementation-body)
```

A CPU implementation of the 'parallel-function' feature, using OpenMP, is provided below:

```
(implement parallel-fn (cpu-target-omp)
  `(function ,name ,(append args '((int ,N))) -> void
     (pragma omp parallel)
     (for ((int ,i 0) (< ,i ,N) (+= ,i 1))
       ,@body)))
```

As can be seen, this is a plain function definition with an iteration statement wrapped around the provided code and, given the invocation above, expands accordingly:

```
void accum(float *a, float *b, int max) {
        #pragma omp parallel
        for (int idx(0); idx < max; idx += 1)
                a[idx] += b[idx];
}
```

As shown above, implementations are selected by targets which are defined as:

```
(define-target target-name [base-target])
```

This can be used to also define derived targets, which, if no implementation for them is present, are configured to use the parent's implementation. The brackets indicate that the base-target argument is optional (for default implementations, see Section 4.2). The CPU target and the derived OpenMP target, followed by the CUDA target, are specified by:

```
(define-target cpu-target)
(define-target cpu-target-omp cpu-target)
(define-target cuda-target)
```

Note that many more targets are possible, and that multiple targets can be specified for implementations, such as:

```
(implement parallel-mult (cpu-target-omp avx2) ...)
```

During composition, the most specialized target will be used. Hence, it is easy to provide and explore target-specific feature derivatives.

A different (and more elaborate) implementation for invoking CUDA functions for parallel computations is shown in Figure 3. In contrast to the single function using OpenMP

```
1  (defclass default () ())
2  (defvar *generics* nil)
3
4  (deflmacro define-target
5      (name &optional (base-target 'default))
6    `(defclass ,name (,base-target) ()))
7
8  (deflmacro define-feature (name args)
9    (let ((gen (symbol-append 'feature- name)))
10     `(progn
11        (push (list (defgeneric ,gen (c1 c2 c3))
12                    (cons ',name (list ',args)))
13              *generics*)
14        (defmethod ,gen ((c1 t) (c2 t) (c3 t)) (values)))))
15
16 (deflmacro implement
17     (feature (c1 &optional (c2 'default) (c3 'default))
18              &body body)
19   `(defmethod ,(symbol-append 'feature- feature)
20        ((c1 ,c1) (c2 ,c2) (c3 ,c3))
21      `(let ((parent ,(call-next-method)))
22         (declare (ignorable parent))
23         ,',@body)))
24
25 (deflmacro make-config (&rest rest)
26   (let ((config (gensym)))
27     `(progn
28        (defclass ,config ,rest ())
29        (make-instance ',config))))
30
31 (defmacro with-config (config &body body)
32   (lisp (let ((c (eval config)))
33     `(macrolet ,(loop for (feature head) in *generics*
34                       collect `(,@head ,(funcall feature c c c)))
35        ,@body))))
```

Figure 4. The complete implementation of our composition system using C-Mera and Common Lisp is remarkably short.

it generates two functions: a host-side stub (line 14), and a CUDA kernel (line 7) with a derived name (line 2). The host-side function takes care to properly invoke the CUDA kernel. This example relies on some meta programming (lines 4-5) to extract the list of actual function arguments which have to be forwarded to the kernel (line 19). For the use case of the simple accumulation function shown above a much shorter implementation could be devised. However, the implementation given in Figure 3 is very general and applies to any function parallelized over one-dimensional arrays, i.e. it always generates the matching pair of host and device-side functions and takes care of proper argument forwarding (lines 4-5 and 19) and border handling (lines 11-12).

Overall, the presented composition system is a simple, yet very clear method to implementing variants, especially with respect to separation of targets.

4.2 Complete Implementation

The complete code of our composition scheme is given in Figure 4 and its implementation will be described in the following. It should be noted that the complete code is a bare 35 lines, which is due to the fact that it maps the feature/target system described above to object oriented meta code that runs at compile time (mapping it to CLOS [8]). This shows yet again that the basic C-Mera system facilitates the provisioning of powerful and clean abstractions with minimal engineering overhead.

As can be seen in lines 4-6 of Figure 4, the definition of a target declares a class of the same name, which also provides for derived targets and mix-ins.

The definition of features (lines 8-14) is, correspondingly, mapped to the definition of generic functions on targets (line 11). Note that each feature is defined with an empty default implementation (line 14) and that multiple targets are provided in the method parameter list to enable dispatch on combined targets (the limitation to triple-dispatch is easily extended).

Similarly, feature implementations are provided by methods on those generic functions (lines 16-23), and it can be seen that derived features obtain the parent feature's code in an established, multi-parent fashion (line 21). Note the default-target defined on line 1. It can be used to provide default implementations that suit all targets that don't define a more specialized method. Furthermore, with derived features, default implementations on any level of the feature-hierarchy can be provided, simply by implementing a feature for the intermediate target.

Different targets can be combined to a compound configuration using `make-config` (lines 25-29), and finally, the `with-config` (lines 31-35) form can be used to provide all the necessary definitions. These are stored during feature definition (lines 11-12) and then collected (line 33-34) during configuration. At this time the previously defined methods are called to obtain the definitions to be used for the provided configuration, as in the following example:

```
1  (with-config (make-config (cpu-target-omp avx2))
2    (parallel-fn accum ((float *a) (float *b)) (idx N)
3      (+= a[idx] b[idx])))
```

In conclusion, our composition system is provided by a lightweight and elegant mapping to object oriented code that is run at the time the to-composed program is being compiled.

5. Case Study: Ray Tracing

To evaluate the applicability of our composition system defined in the previous section to the context of rendering we implemented a ray tracing system that allows for easy domain exploration by feature composition, while also making it simple to extend the set of available variants. The system is used to facilitate exploration by domain experts; interfaces for intermediate users can be built on this basis. We chose ray traversal as a realistic use case as even though there is a plethora of options regarding algorithmic and architecture-specific variants (see Section 2) the resulting implementations typically consist of only a few dozen lines of code and are comprehensible even in a paper context.

Using the system described in Section 4 we will describe two use cases. Section 5.1 shows how code can be formulated by use of domain-specific features, and how different implementations of these features can be added. The focus of this examination is how two different top-level approaches to the primary hierarchy traversal loop can be managed, namely standard traversal where each thread only considers a single ray, and traversal using persistent threads [1], a method that manually recycles GPU threads and takes care of high warp utilization. The second use-case, presented in Section 5.2, shows strongly tangled features at the stage where leaf nodes are tested for overlap with the ray. Our example considers 64 different (and relevant) combinations where the longest resulting variant is 20 lines of code.

5.1 High Level Ray Traversal

In the previous section we have shown a very small, but quite effective configuration system implemented with C-Mera. We have demonstrated how it can be used to generalize the

```
1  (deftracer default (make-instance 'default-config)
2    (prepare-ray (rid (thread-x) (thread-y))
3      (with-local-stack (stack sp 32)
4        (with-local-node
5          (ray-management (rid (thread-x) (thread-y) node-idx)
6            (while true
7              (load-node node-idx)
8              (if (is-inner curr)
9                  (intersect-inner stack sp node-idx curr)
10                 (progn (intersect-leaf curr)
11                        (if (>= sp 0)
12                            (set node-idx stack[sp--])
13                            (break)))))))))
```

Figure 5. Implementation of the standard ray traversal scheme. Note that the different syntax tree nodes are features that are configurable for specific variants via **implement**.

concepts of ray-iteration and kernel definition for different target platforms, namely C++ and CUDA.

This section will continue the previous example and show how our system can be used to implement ray traversal for the mentioned platforms while also providing a choice of very different algorithms. To this end we present a straightforward textbook ray tracing layout, as well as ray traversal using persistent threads [1]. For the CUDA target the choice between those two will be available. This higher level example illustrates how easily a structurally different implementation can be constructed based on existing features.

The complete system is available in our supplemental material (together with a version implemented using C++ template meta programming) and contains a few more variants, see Section 5.3. Some of them will be shortly introduced in the following presentation of the default, textbook style traversal algorithm. This default algorithm is a stack based BVH traversal with a single loop. Within this loop the boxes of inner nodes are intersected with the ray and their children pushed to the stack, if appropriate. Leaf nodes are intersected and, for shadow rays, can terminate ray traversal. This algorithm is a good fit for CPU architectures as, e.g., simultaneous multi-threading, benefits from thread divergence. We believe that our notation, shown in Figure 5, nicely reflects this description and provide a few details in the following.

The first line of Figure 5 defines a tracer function (either as normal C++ function, or as CUDA kernel with appropriate host-stub). The next line sets up the index of the ray to trace. We define this feature as follows:

```
1  (define-feature prepare-ray
2    ((ray-idx x y) &body body))
```

And implement it as CPU and CUDA variants:

```
1  (implement prepare-ray (default)
2    `(decl ((const int ,ray-idx (+ (* ,y w) ,x)))
3       ,@body))
4
5  (implement prepare-ray (cuda-target)
6    `(if (and (< ,x w) (< ,y h))
7       'parent))
```

This means that a variable to hold the ray-index is introduced and, for the CUDA variant, we also check for valid thread indices. Note how the CUDA declaration is derived from the CPU version. This allows more specialized implementations to take advantage of more general versions with a similar root. In lines 3-5 of Figure 5 we introduce stack management, the current node (configurable for different node types) and ray data (configurable for exact and inexact rays). The rest

```
1  (define-feature ray-management
2    ((ray-idx x y node-idx) &body body))
3
4  (implement ray-management (cuda-target persistent-threads)
5   `(decl ((__shared__ volatile int next-ray[6])
6           (int ,ray-idx))
7     (while true
8       ...
9       (if terminated
10          ...
11          (if (>= ,ray-idx (* w h))
12              (break))
13          (set closest t-max)
14          (load-ray ,ray-idx))
15      ,@body
16      (set (aref intersections ,ray-idx) closest))))
```

Figure 6. Reduced version of our persistent threads implementation illustrating the change in control flow.

of the code specifies a very general traversal framework where nodes are intersected and managed via the previously introduced stack. The last four lines take care of leaf nodes and potentially terminate traversal. Apart from adding persistent threads to this scheme, the management of triangle intersections will also be covered later in this section.

Traversal using persistent threads [1] requires a different code layout. Here, nodes are still managed by a stack, but the kernel is only invoked with as many threads as can run in parallel. These threads then manage their mapping to rays using atomic counters and dynamically reload rays should warp occupancy fall below a certain threshold. Therefore, the algorithm loops as long as there are still rays to be processed. This method is in contrast to the approach presented above and only applicable to ray traversal on the GPU.

Figure 6 shows a (reduced) version of how the corresponding feature, `ray-management`, is declared and implemented for the `cuda-target` with `persistent-threads`. The most interesting part is that the code this feature wraps around is enclosed in another loop which implements the procedure described above. The scope of this loop as well as the wrapped code are highlighted in Figure 6. This extension thus introduces a marked change in control flow of the general algorithm which, by virtue of our abstraction mechanism, is hidden as an implementation detail. We believe that this example illustrates how higher level algorithmic variations can be implemented while preserving the clarity of the basic control flow.

5.2 Low Level Triangle Intersection

The previous example employed a rather high-level point of view, where the algorithm was modified top-down to change the primary flow of the program. In the following we will present a lower level scenario, along with its features and how they are configured to work together. We believe this example to be very relevant as many different choices have to be taken for this part of the traversal algorithm and it is crucial for an exploration system to support such highly tangled features.

Figure 8 shows code to process triangles from leaf nodes, together with the features that influence each line. As an example the first two lines are appropriate when triangles are stored in leaf nodes. However, when only a single triangle index is stored leaf node addresses can be interpreted as triangle addresses [2]. The number of triangles stored in each leaf is then encoded by invalid triangles stored at the end of each triangle list. This choice would change line 1 and remove

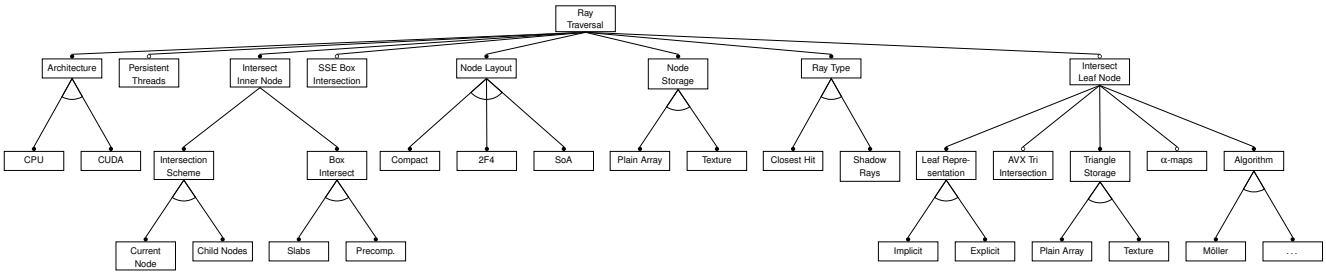

Figure 7. Family of ray tracing variants implemented using our composition system.

```
1  int start = ...;                                  // FROM NODE | IMPLICIT LEAF
2  int end = ...;                                    // FROM NODE | N/A
3  tri_intersection is(invalid);
4  for (; start < end; ++start)                      // WITH | W/O BORDER
5      tri_t tri = load_triangle(start);             // STORAGE
6      if (invalid_triangle(tri))                    // IMPLICIT LEAF
7          break;                                    // IMPLICIT LEAF
8      if (intersect(tri, ray, is))                  // ALGORITHM
9          if (is.t < closest.t)                     // CLOSEST HIT
10             if (eval_alpha_map() == 0)            // ALPHA MAPPING
11                 continue;                         // ALPHA MAPPING
12             closest = is;
13             break;                                // ONLY SHADOW RAYS
14 if (closest.valid())                              // ONLY SHADOW RAYS
15     break;                                        // ONLY SHADOW RAYS
```

Figure 8. An illustration of feature tangling for the traversal of leaf nodes.

line 2. Furthermore for the iteration in line 4 the termination case would be dropped and in its place the explicit check for invalid triangles (lines 6 and 7) would be used. Similar changes are required for different storage options for triangles, different intersection methods, whether the closest hit is to be found, or shadow rays are cast, and if code supporting alpha mapping should be inserted. Our supplemental material contains a version of this code using our feature-oriented approach (and our C++ traversal mentioned above also includes some of the choices illustrated in Figure 8).

The implementation using our configuration system is as follows.

```
1  (decl ((tri_intersect closest -1)
2         (tri_intersect is))
3   (for-each-triangle-in-node
4     (load-node
5       (intersect-triangle tri ray is))))
```

where different features can add to the implementation of `intersect-triangle`, while the outer traversal loop is implemented by the feature `for-each-triangle-in-node`, which takes care of explicit or implicit nodes. The following example illustrates the former case:

```
1  (implement for-each-triangle-in-node (explicit-node)
2    `(decl ((int start (load_from_node))
3            (int end (load_from_node)))
4       (while (< start end)
5         ,@body
6         ++start)
7       (early-out)))
```

5.3 Study Results

In this section we show results of our approach as applied to the two examples mentioned previously. The configuration options implemented for the high-level ray tracing variants, as described in Section 5.1 and depicted in Figure 7 (except the "Intersect Leaf Node" part), are: tracing on the CPU, tracing on the GPU with CUDA, optionally with persistent threads [1] (on the GPU), or using SSE (on the CPU), directly intersecting the nodes during traversal, or intersecting and sorting the two child nodes of each node encountered. Furthermore we provide different node layouts (using an integer-based, compact structure, a float-based array [2], and a structure of arrays), two different variants to intersect bounding boxes [1, 27], and whether to find the closest intersection, or terminate at the first intersection found. We compare using our feature-oriented system to a version using C++ template meta programming (TMP) regarding code-size. Our implementation is 275 lines of code, whereas the TMP approach takes 960 lines to implement the same set of feature-combinations. Furthermore, the structure of our implementation is easily accessible while with the TMP version it is hidden behind template-syntax and parameter-forwarding.

We also implemented the low-level example described in Section 5.2 using our approach. The "Intersect Leaf Node" part of Figure 7 shows a diagram of the implemented feature variations, namely if leaf nodes are explicitly stored (offset and length) or implicit, leaving room for the offset, only. This requires that the length of the triangle-array also be stored implicitly, which is done by appending an invalid triangle to each array [2]. Further variations are how the nodes are stored (in global or texture memory), the ray type (see above), which ray-triangle intersection algorithm should be used [38], and whether alpha-maps (transparency by textures) is active. Our implementation consists of 150 lines of code resulting in 64 different versions of this sub-tree of the diagram (plus node layout).

Overall there are already more than 2300 valid and plausible variants provided by the features implemented – even those are yet only a small subset of the features described in Section 2. Our current implementation focuses on architecture-specific variations and limitations in the supported feature-set of ray traversal. We did not yet consider different acceleration structures, primitives, and other traversal-related options, such as using packets, stack-less traversal, or ray sorting, but adding further variants is, as has been shown in this section, easily possible.

As the goal of our method is making the process of domain-exploration more manageable we do not show performance measurements. With our generative technique, developers are able to generate exactly the desired code, which includes any reference provided. Hence, there is no inherent performance trade-off.

For reference, our supplemental material contains all the implementations mentioned above.

6. Discussion and Related Work

In this paper we propose an approach to problem-domain exploration using generative programming techniques. Section 4 presents a simple technique to implement algorithmic variation and in Section 5 we apply it to the domain of ray traversal kernels. Therefore we will focus on related work concerned with domain-specific abstractions, and only touch upon the topic of general generative programming methods. For a comprehensive summary we refer to Czarnecki and Eisenecker [11].

The most ubiquitous approach to generative programming is C++ template meta programming (TMP) [11, 55]. It has been applied to a vast range of problems, also in computer graphics. An example is RTfact [52], an active library [12] for ray-tracing variants. However, we argue that, while certainly comfortable and unobtrusive to use, exploration is seriously impaired by this approach as the maintenance overhead of the meta code becomes a burden in itself [19, 35]. We provide TMP reference code with our supplemental material to compare it to our proposed solution. In general, implementors of active libraries strive for easy-to-use interfaces for intermediate users and the latter do not necessarily have to deal with the inherent complexity of TMP. Domain-specific languages take this approach one step further by providing languages and generators tailored to a specific problem set, thereby completely hiding the inner workings. Examples include HIPAcc [37] and Halide [44], image processing languages that target the generation of performant code for heterogeneous architectures from problem-oriented input program specifications.

However, while these abstractions are a tremendous tool for intermediate developers to come up with portable, yet fairly efficient graphics code in many cases, implementation-centric domain exploration is done by experts that require full disclosure. In a sense, C-Mera addresses rather the designers of active graphics libraries and DSLs than their users, as the former also have to do quite a bit of domain exploration to come up with efficient generation schemes for each particular architecture. Nevertheless, using C-Mera also to design DSLs built on top of our ray-tracer family is perfectly possible. Such a scheme would also appeal to DSL users that still require fine-grained control over the generated code.

Multi-stage programming is a paradigm where a programming language is embedded into another language, the host language. The embedded program is transparently compiled or transformed into the host language as part of the compilation of the embedding program. Examples of such approaches include AnyDSL [36], which allows control of compile-time computation (partial evaluation), MetaOCaml [9], a multi-stage extension for OCaml, and Terra [14], a low-level language embedded in Lua. Rompf and Odersky [45] implement staging by type annotations (LMS). Taha [54] provides a very approachable introduction to multi-stage programming. Fredriksson [19] proposes an idea similar to the approach taken by C-Mera. Also, Terra and AnyDSL are very close to our work in that we use a very high level language to describe program transformations of a lower level language. Similarly, they also provide homogeneous notation.

In contrast to AnyDSL and LMS, our approach is not truly embedded, i.e. during more advanced meta programming the user is required to write Lisp code, not C code, to implement transformations. Furthermore, as C-Mera's macro system is borrowed from Common Lisp it is not hygienic, i.e. care has to be taken to avoid unwanted capture. Since its focus is not on working on a typed AST, but on syntactic abstractions, global modifications are not trivial. But even in the face of these limitations, we have shown how easily an advanced feature composition system can be built using C-Mera: The more direct approach taken by C-Mera also makes it trivial to force the compiler to generate a certain C or machine code sequence. So while providing less abstraction than other approaches, direct control, little requirements in tooling (none in the OEM case) and simple porting to different platforms makes C-Mera and our light-weight composition system attractive, especially for pragmatic settings.

The general importance of implementation-domain exploring (that is, the systematic evaluation of implementation variants) has been demonstrated in recent work of Peters and Klein [41], who evaluate a vast number of different basis functions to find the best trade-off between shadow quality and run time for shadow filtering algorithms. Another example is the work of Aila and Laine [1], who investigate how to best map ray traversal to modern GPU architectures. In Section 2 we show that a large body of research is available on ray traversal. Based on different aspects of this research we show how our system can be used to capture common concepts while providing ample opportunity for exploring further variants.

The opportunity to easily add "ad-hoc" target-specific derivatives of features offered by our composition system is often necessary in computer graphics because of the tangled interaction of the concrete problem, chosen algorithms, and hardware properties. Subtle implementation details can have a tremendous impact on performance and quality, which so far leaves little opportunity to apply measures for the automatic prediction or optimization of nonfunctional properties, as for instance suggested by Siegmund and colleagues [50, 51] for software product lines.

Even though our case study focuses on ray traversal, we would like to stress that our approach is not specifically designed for this use case: C-Mera as well as our composition system implemented with it are generally applicable.

Experience has shown, however, that C-Mera could be improved with respect to "programmer compatibility": The unfamiliar notation of C-Mera discourages many developers in our domain (most of which have an upbringing as "hard-core C hackers") – even though, on the other hand they are typically very appealed by the approach in general. While programmer compatibility could certainly be improved by switching to a syntax developers from this context are more accustomed to, we also think that another aspect underlines the importance of lightweight, low-risk and transparent approaches towards meta programming: In a team setting, single developers can start using C-Mera on small pieces of code to ease their own work with little to no impact on their team mates. Instead of manually deriving variants to explore the implementation domain, they apply generative techniques, but keep full control over the resulting C-type code, which they eventually pass to their colleagues.

7. Conclusions

Computationally intensive 3D computer graphics has become an ubiquitous element of modern computing systems, from high-end movie production down to user interfaces in small-scale embedded control systems, such as car instruments. However, despite of the availability of higher-level abstraction frameworks (such as CUDA or OpenGL): To come up with a truly efficient implementation, developers still have to put enormous efforts into finding fitting solutions – and the situation gets even worse by the increasing heterogeneity

of the underlying hardware. Small changes in algorithms, data structures, hardware revisions or even instruction scheduling by the compiler can have a dramatic impact on the resulting performance [1]; in practice even experienced graphics engineers have to explore and refine many variants to come up with a good solution.

In this setting, we propose meta programming with C-Mera and a very lightweight feature composition system as a pragmatic means for efficient implementation-domain exploration. Our approach makes it easy to define new variants and architecture-specific feature derivatives can be adopted incrementally, and it requires only little tool support. The possibility to provide high-level feature abstractions, yet keep full control over the resulting code in a C-type language facilitates application in domains that require fine control over the code regarding externally imposed constraints, such as certifiability, or the necessity to use old and proprietary, but proven-in-use, language and compiler technology as well as during research striving for maximal performance while evaluating numerous alternatives.

Acknowledgments

This work was supported by the Research Training Group 1773 "Heterogeneous Image Systems" (`http://hbs.fau.de`), funded by the German Research Foundation (DFG).

References

[1] T. Aila and S. Laine. Understanding the efficiency of ray traversal on gpus. In *Proc. High-Performance Graphics 2009*, pages 145–149, 2009.

[2] T. Aila, S. Laine, and T. Karras. Understanding the efficiency of ray traversal on GPUs – Kepler and Fermi addendum. NVIDIA Technical Report NVR-2012-02, NVIDIA Corporation, June 2012.

[3] J. Amanatides and A. Woo. A fast voxel traversal algorithm for ray tracing. In *In Eurographics '87*, pages 3–10, 1987.

[4] S. Apel and C. Kästner. Virtual separation of concerns - a second chance for preprocessors. *Journal of Object Technology*, 8(6):59–78, 2009.

[5] A. Appel. Some techniques for shading machine renderings of solids. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), pages 37–45, New York, NY, USA, 1968. ACM.

[6] R. Barringer and T. Akenine-Möller. Dynamic ray stream traversal. *ACM Trans. Graph.*, 33(4):151:1–151:9, July 2014.

[7] C. Benthin, S. Woop, M. Nießner, K. Selgrad, and I. Wald. Efficient ray tracing of subdivision surfaces using tessellation caching. In *Proc. High-Performance Graphics 2015*, 2015.

[8] D. G. Bobrow, L. G. Demichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, and D. A. Moon. Common Lisp Object System specification 1. programmer interface concepts. *Lisp and Symbolic Computation*, 1(2):245–298, Sept. 1988.

[9] C. Calcagno, W. Taha, L. Huang, and X. Leroy. Implementing multi-stage languages using asts, gensym, and reflection. In *Proceedings of the 2Nd International Conference on Generative Programming and Component Engineering*, GPCE '03, pages 57–76, New York, NY, USA, 2003. Springer-Verlag New York, Inc. ISBN 3-540-20102-5.

[10] P. Christensen, J. Fong, D. Laur, and D. Batali. Ray tracing for the movie 'cars'. *Symposium on Interactive Ray Tracing*, 0:1–6, 2006.

[11] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications*. Addison-Wesley, May 2000. ISBN 0-20-13097-77.

[12] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, and T. Veldhuizen. Generative programming and active libraries. In M. Jazayeri, R. Loos, and D. Musser, editors, *Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 2000. ISBN 978-3-540-41090-4.

[13] H. Dammertz, J. Hanika, and A. Keller. Shallow bounding volume hierarchies for fast simd ray tracing of incoherent rays. *Comput. Graph. Forum*, 27(4):1225–1233, 2008.

[14] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek. Terra: A multi-stage language for high-performance computing. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 105–116, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6.

[15] P. Dutre, K. Bala, P. Bekaert, and P. Shirley. *Advanced Global Illumination*. AK Peters Ltd, 2006.

[16] M. Ernst and G. Greiner. Early split clipping for bounding volume hierarchies. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 73–78, Washington, DC, USA, 2007. IEEE Computer Society.

[17] M. Ernst and G. Greiner. Multi Bounding Volume Hierarchies. In *IEEE Symposium on Interactive Ray Tracing*, pages 35–40, Aug. 2008.

[18] N. Feltman, M. Lee, and K. Fatahalian. SRDH: Specializing BVH Construction and Traversal Order Using Representative Shadow Ray Sets. In C. Dachsbacher, J. Munkberg, and J. Pantaleoni, editors, *Eurographics/ ACM SIGGRAPH Symposium on High Performance Graphics*. The Eurographics Association, 2012.

[19] A. Fredriksson. Amplifying C. http://voodoo-slide.blogspot.de/2010/01/amplifying-c.html, 2010.

[20] A. S. Glassner. Tutorial: Computer graphics; image synthesis. chapter Space Subdivision for Fast Ray Tracing, pages 160–167. Computer Science Press, Inc., New York, NY, USA, 1988. ISBN 0-8186-8854-4.

[21] A. S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press Ltd., London, UK, UK, 1989. ISBN 0-12-286160-4.

[22] G. Guennebaud, B. Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[23] J. Gunther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on gpu with bvh-based packet traversal. In *Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing*, RT '07, pages 113–118, Washington, DC, USA, 2007. IEEE Computer Society.

[24] M. Hapala, T. Davidovic, I. Wald, V. Havran, and P. Slusallek. Efficient stack-less bvh traversal for ray tracing. In *27th Spring Conference on Computer Graphics (SCCG 2011)*, 2011.

[25] V. Havran and J. Bittner. On improving kd-trees for ray shooting. *Journal of WSCG*, 10(1):209–216, February 2002.

[26] T. Karras. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees. In *Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics*, EGGH-HPG'12, pages 33–37, Aire-la-Ville, Switzerland, Switzerland, 2012. Eurographics Association.

[27] T. L. Kay and J. T. Kajiya. Ray tracing complex scenes. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 269–278, New York, NY, USA, 1986. ACM.

[28] A. Keller, T. Karras, I. Wald, T. Aila, S. Laine, J. Bikker, C. P. Gribble, W. Lee, and J. McCombe. Ray tracing is the future and ever will be.. In *International Conference on Computer Graphics and Interactive Techniques, SIGGRAPH 2013, Anaheim, CA, USA, July 21-25, 2013, Courses*, pages 9:1–9:7, 2013.

[29] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-dops. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, Jan. 1998.

[30] A. Lagae and P. Dutré. Compact, fast and robust grids for ray tracing. *Computer Graphics Forum*, 27(4):1235–1244, 2008.

[31] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 2009.

[32] W.-J. Lee, Y. Shin, J. Lee, J.-W. Kim, J.-H. Nah, S. Jung, S. Lee, H.-S. Park, and T.-D. Han. Sgrt: A mobile gpu architecture for real-time ray tracing. In *Proceedings of the 5th High-Performance Graphics Conference*, HPG '13, pages 109–119, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2135-8.

[33] D. J. MacDonald and K. S. Booth. Heuristics for ray tracing using space subdivision. *Vis. Comput.*, 6(3):153–166, May 1990.

[34] J. McCarthy. History of programming languages i. chapter History of LISP, pages 173–185. ACM, New York, NY, USA, 1981. ISBN 0-12-745040-8.

[35] M. McCool, S. Du, T. Tiberiu, P. Bryan, and C. K. Moule. Shader algebra. *ACM Transactions on Graphics*, pages 787–795, 2004.

[36] R. Membarth, P. Slusallek, M. Köster, R. Leißa, and S. Hack. High-performance domain-specific languages for gpu computing. GPU Technology Conference (GTC), March 2014.

[37] R. Membarth, O. Reiche, F. Hannig, J. Teich, M. Körner, and W. Eckert. HIPAcc: A domain-specific language and compiler for image processing. *IEEE Transactions on Parallel and Distributed Systems*, PP(99):1–14, 2015.

[38] T. Möller and B. Trumbore. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools*, 2(1):21–28, Oct. 1997.

[39] J.-H. Nah and D. Manocha. SATO: Surface Area Traversal Order for Shadow Ray Tracing. *Computer Graphics Forum*, 2012.

[40] J.-H. Nah, H.-J. Kwon, D.-S. Kim, C.-H. Jeong, J. Park, T.-D. Han, D. Manocha, and W.-C. Park. Raycore: A ray-tracing hardware architecture for mobile devices. *ACM Trans. Graph.*, 33(5):162:1–162:15, Sept. 2014.

[41] C. Peters and R. Klein. Moment shadow mapping. In *Proceedings of the 19th Symposium on Interactive 3D Graphics and Games*, i3D '15, pages 7–14, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3392-4.

[42] S. Popov, J. Guenther, H.-P. Seidel, and P. Slusallek. Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. *Computer Graphics Forum*, 2007.

[43] C. Prehofer. Feature-oriented programming: A fresh look at objects. In M. Aksit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, Lecture Notes in Computer Science, pages 419–443. Springer-Verlag, June 1997.

[44] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6.

[45] T. Rompf and M. Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled dsls. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering*, GPCE '10, pages 127–136, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0154-1.

[46] J. Schmittler, I. Wald, and P. Slusallek. Saarcor: A hardware architecture for ray tracing. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 27–36, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[47] T. W. Sederberg and T. Nishita. Curve Intersection using Bezier Clipping. *Computer-Aided Design*, 22(9):538–549, 1990.

[48] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan. Larrabee: A many-core x86 architecture for visual computing. In *ACM SIGGRAPH 2008 Papers*, SIGGRAPH '08, pages 18:1–18:15, New York, NY, USA, 2008. ACM.

[49] K. Selgrad, A. Lier, M. Wittmann, D. Lohmann, and M. Stamminger. Defmacro for C: Lightweight, ad hoc code generation. In *Proceedings of ELS 2014 7rd European Lisp Symposium*, pages 80–87, 2014.

[50] N. Siegmund, S. Kolesnikov, C. Kastner, S. Apel, D. Batory, M. Rosenmuller, and G. Saake. Predicting performance via automated feature-interaction detection. In *Proceedings of the 34nd International Conference on Software Engineering (ICSE '12)*, pages 167–177, Washington, DC, USA, June 2012. IEEE Computer Society Press.

[51] N. Siegmund, M. Rosenmüller, M. Kuhlemann, C. Kästner, S. Apel, and G. Saake. SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3-4):487–517, 2012.

[52] P. Slusallek and I. Georgiev. Rtfact: Generic concepts for flexible and high performance ray tracing. In R. J. Trew, editor, *Proceedings of the IEEE / EG Symposium on Interactive Ray Tracing 2008*, pages 115–122, RT08 Reception Warehouse Grill 4499 Admiralty Way Marina del Rey, CA 90292, 2008. IEEE Computer Society, Eurographics Association, IEEE.

[53] M. Stich, H. Friedrich, and A. Dietrich. Spatial splits in bounding volume hierarchies. In *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, pages 7–13, New York, NY, USA, 2009. ACM.

[54] W. Taha. A gentle introduction to multi-stage programming. In *Domain-specific Program Generation, LNCS*, pages 30–50. Springer-Verlag, 2004.

[55] T. Veldhuizen. Template metaprograms. *C++ Report*, May 1995.

[56] C. Wächter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, EGSR '06, pages 139–149, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. ISBN 3-905673-35-5.

[57] I. Wald and V. Havran. On building fast kd-trees for ray tracing, and on doing that in O(N log N). In *Proceedings of IEEE Symposium on Interactive Ray Tracing 2006*, pages 61–69, Sept. 2006.

[58] I. Wald and P. Slusallek. State of the art in interactive ray tracing. *State of the Art Reports, EUROGRAPHICS*, 2001: 21–42, 2001.

[59] I. Wald, P. Slusallek, C. Benthin, and M. Wagner. Interactive rendering with coherent ray tracing. In *Computer Graphics Forum*, pages 153–164, 2001.

[60] I. Wald, C. Benthin, and S. Boulos. Getting rid of packets - efficient simd single-ray traversal using multi-branching bvhs, 2008.

[61] I. Wald, S. Woop, C. Benthin, G. S. Johnson, and M. Ernst. Embree: A kernel framework for efficient cpu ray tracing. *ACM Trans. Graph.*, 33(4):143:1–143:8, July 2014.