An Optimizing Compiler for Automatic Shader Bounding

Petrik Clarberg[1] Robert Toth[1] Jon Hasselgren[1] Tomas Akenine-Möller[1,2]

[1]Intel Corporation [2]Lund University

Abstract

Programmable shading provides artistic control over materials and geometry, but the black box nature of shaders makes some rendering optimizations difficult to apply. In many cases, it is desirable to compute bounds of shaders in order to speed up rendering. A bounding shader can be automatically derived from the original shader by a compiler using interval analysis, but creating optimized interval arithmetic code is non-trivial. A key insight in this paper is that shaders contain metadata that can be automatically extracted by the compiler using data flow analysis. We present a number of domain-specific optimizations that make the generated code faster, while computing the same bounds as before. This enables a wider use and opens up possibilities for more efficient rendering. Our results show that on average 42–44% of the shader instructions can be eliminated for a common use case: single-sided bounding shaders used in lightcuts and importance sampling.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; G.1.0 [Numerical Analysis]: General—Interval arithmetic

1. Introduction

The advent of highly realistic computer-generated graphics in feature films and games has largely been made possible by the separation of rendering algorithms and visual content. Programmable shading provides means for artists to create wonderful environments, without having to deal with much of the technicalities of the renderer. From the rendering system's point of view, a shader is a black box, which can only be point-sampled. This presents a problem, as higher level information about a shader is often required to make use of more efficient rendering algorithms. For example, in global illumination where the light transport integrals are very complex, it is critical to be able to compute *bounds* of a shader in order to use algorithms such as lightcuts [WFA*05, WABG06] and importance sampling. In a rasterization pipeline, shader bounds may be used to avoid computations that do not contribute to the image [HAM07].

A shader bounding function for an arbitrary shader can be carefully handcrafted, but this is tedious and error-prone for all but the simplest shaders. Alternatively, a compiler can be used to automatically derive a bounding function using interval analysis [HSS98,HAM07,VAZH*09]. The compiler transforms the shader instructions to operate on *intervals* rather than single values. The end result is a *bounding shader*, which given bounds on the shader inputs computes bounds on its outputs, i.e., a conservative range of possible outputs. Some examples are shown in Figure 1.

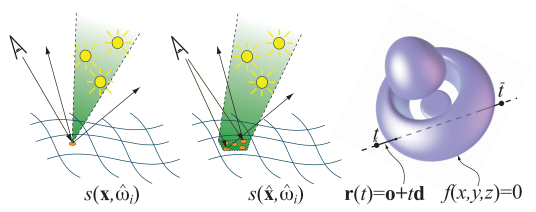
$s(\mathbf{x},\hat{\omega}_i)$
$s(\hat{\mathbf{x}},\hat{\omega}_i)$
$\bar{t}$
$\underline{t}$
$\mathbf{r}(t)=\mathbf{o}+t\mathbf{d}$
$f(x,y,z)=0$

Figure 1: *The bounding shader s computes bounds for the shaded result at a specific shading point,* $\mathbf{x}$*, or cluster of points,* $\hat{\mathbf{x}}$*, given bounds on the light directions,* $\hat{\omega}_i$*. This is critical functionality in lightcuts and importance sampling. The right image shows a ray tracing application, where a bounding shader is used to find the closest intersection.*

There are two factors directly affecting the efficiency of the rendering system: the execution time of the bounding shader, and the tightness of the computed bounds. In this paper, we focus on generating faster bounding shaders, without changing the computed bounds. Naïve transformation from scalar to interval shader code is relatively straightforward for a compiler. Each arithmetic instruction is replaced by an instruction sequence that performs the same operation on intervals. However, by exploiting domain-specific knowledge and data flow analysis, we show that it is possible to achieve much better results. Although we focus on interval arithmetic (IA) [Moo66], where an interval is simply represented as a minimum and a maximum value, higher-order methods such as affine arithmetic [CS93], or Taylor model arithmetic [BH98] can also be used.

One of our key contributions is a method we call *static bounds analysis*, in which bounds are propagated through the shader at compile time in order to determine the type and possible range of each variable. This information allows us to generate optimized interval arithmetic code. Similar results cannot be achieved using standard compiler techniques. We also propose an optional extension called *valid range analysis*, which exploits the fact that some instructions put strict limits on their inputs. Another important contribution is the use of *dynamic bounds assumptions*, which creates a fast path for the common case, but falls back on a more general bounding shader if the assumptions fail at runtime.

2. Related Work

Originally developed in the 1950s to compute bounds on rounding errors in numerical computations, interval analysis [Moo66] is now used in a wide range of scientific and engineering disciplines.

Hardware/Software Support for Interval Analysis Specialized hardware architectures with support for interval analysis have been proposed, see, e.g., [SS00], but the extra area/power is hard to motivate for non-scientific workloads. Software implementations provide more flexibility and are currently the only alternative for graphics applications. There are, however, few compilers with support for interval analysis on existing hardware [SZAB99, ASS04], and most users are left to using publicly available libraries [Ž05] or GPU implementations [CFD08]. It should be noted that none of the existing implementations apply any interval-specific compiler optimizations.

Interval Analysis in Computer Graphics A full overview is beyond the scope of this paper, so we limit the discussion to a few selected applications. Snyder [Sny92] use interval analysis to robustly solve a wide range of problems related to parametric surfaces. Mitchell [Mit90] proposed to use interval arithmetic for robust ray tracing of implicit surfaces, which initiated a lot of work in this direction. See Hijazi et al.'s survey [HHHJ07] for an overview. Collision detection is another successful application of interval analysis, with methods for implicit surfaces [Duf92], rigid bodies [RKC02], and articulated models [ZRLK07]. Some applications, especially with long interval computation chains, benefit from using higher-order methods, such as affine arithmetic [CS93] and Taylor models [BH98].

Programmable Shading Programmable shading has been a keystone in offline rendering for more than two decades. The RenderMan shading language [HL90] was one of the first languages for production-quality rendering, and it introduced many of the concepts used by today's shading languages. Programmable shading is also critical for bringing flexibility to ray tracing systems [PBBR07]. GPUs capable of executing shaders first appeared in 2001, and recent graphics APIs define many types of shaders, each performing a specific task in the graphics pipeline.

Interval Analysis of Programmable Shaders A variety of applications have used interval analysis to compute bounds of shaders. Greene and Kass [GK94] bound shaders that can be expressed in data flow form (i.e., shaders without intricate control flow) to achieve error-bounded antialiasing. They use a compiler to automatically generate interval arithmetic code [Kas92]. Standard compiler optimizations (e.g., common-subexpression elimination) and mathematical simplifications are applied, but no interval-specific optimizations. Heidrich et al. [HSS98] compute bounds on procedural RenderMan shaders using affine arithmetic for sampling purposes. They note that affine arithmetic should not be used for expressions involving only non-affine values, and that it is important to use optimized approximations, e.g., square instead of general multiplication, where possible. A similar framework has been used for ray tracing of procedural displacement shaders [HS98].

In real-time graphics, interval analysis has recently been used to compute conservative bounds for fragment programs over a tile of pixels in order to discard, or *cull*, shading computations [HAM07]. The authors propose a hardware architecture capable of interval arithmetic, thereby avoiding the problem of generating optimized scalar code from an interval-based shader. Later work has extended this concept to compute bounds on vertex programs and cull base primitives prior to tessellation in a DirectX 11-style pipeline using Taylor model arithmetic [HMAM09].

Velázquez-Armendáriz et al. [VAZH*09] present a basic compiler for automatic bounding shading generation, and showcase its utility on a wide range of photo-realistic rendering examples. We extend their work with a number of domain-specific optimizations for generating faster code.

3. Interval Analysis Primer

The goal of interval analysis is to compute conservative bounds for arbitrary computations. This is done by redefining all operations to operate on intervals rather than individual values. We limit ourselves to extended real numbers (i.e., including $\pm\infty$), and define an interval as:

$$\hat{a} = [\underline{a}, \overline{a}] = \{x \in \mathbb{R} \,|\, \underline{a} \leq x \leq \overline{a}\}. \tag{1}$$

3.1. Arithmetic Operations

The basic arithmetic operations are easily extended to operate on intervals. Addition and subtraction become:

$$\hat{a}+\hat{b} = \left[\underline{a}+\underline{b}, \overline{a}+\overline{b}\right] \quad \text{and} \quad \hat{a}-\hat{b} = \left[\underline{a}-\overline{b}, \overline{a}-\underline{b}\right]. \tag{2}$$

Multiplication is slightly more complicated due to the cases where one or both of the intervals overlap zero:

$$\hat{a}\cdot\hat{b} = \left[\min(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}), \max(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b})\right]. \tag{3}$$

Some operations, such as division, are more complex:

$$\begin{aligned} \left[\underline{a},\overline{a}\right] / \left[\underline{b},\overline{b}\right] &= \left[\underline{a},\overline{a}\right] \cdot \left(1/\left[\underline{b},\overline{b}\right]\right), \quad \text{where} \\ 1/\left[\underline{b},\overline{b}\right] &= \begin{cases} \left[1/\overline{b}, 1/\underline{b}\right] & \text{if} \quad 0 \notin \left[\underline{b},\overline{b}\right], \\ \left[-\infty, \infty\right] & \text{if} \quad 0 \in \left[\underline{b},\overline{b}\right]. \end{cases} \end{aligned} \tag{4}$$

In the last example, useful information is lost if the denominator interval contains zero. This problem is not limited to division, but applies to any function that is piecewise continuous, e.g., tan. By working with *multi-intervals*,

$$\hat{a} = \bigcup_i \left[\underline{a}_i, \overline{a}_i\right], \tag{5}$$

we can split $\left[\underline{b},\overline{b}\right]$ into $\left[\underline{b},0\right] \cup \left[0,\overline{b}\right]$ and get:

$$1/\left[\underline{b},\overline{b}\right] = \left[-\infty, 1/\underline{b}\right] \cup \left[1/\overline{b}, \infty\right] \quad \text{if} \quad 0 \in \left[\underline{b},\overline{b}\right]. \tag{6}$$

Such multi-intervals may be further split, or merged if overlapping, depending on the operations performed. We propose using multi-intervals for the compile-time analysis, but single intervals at runtime for efficiency reasons.

3.2. General Functions

A function, $f : \mathbb{R}^n \to \mathbb{R}$, may be extended to interval form to compute bounds for the result based on bounds of the arguments. We define the *interval extension*, $\hat{f}$, of f as:

$$\hat{f}(\hat{\mathbf{x}}) \supseteq \{f(\mathbf{x}) \mid \mathbf{x} \in \hat{\mathbf{x}}\}. \tag{7}$$

Using interval arithmetic, we treat $\hat{\mathbf{x}} = (\hat{x}_1, \ldots, \hat{x}_n)$ as individual intervals and compute intervals for each intermediate result independently. Tighter bounds may be achieved by keeping information on how the intermediate results depend on $\hat{\mathbf{x}}$. These dependencies may be modeled as linear functions (affine arithmetic), or at a higher cost, as general polynomials (Taylor model arithmetic).

3.3. Rounding Considerations

In practice, an application using interval arithmetic is forced to work with floating-point numbers of finite precision. Most general purpose implementations are designed to guarantee interval enclosure of real operations, i.e., produce mathematically conservative results. This requires *outward* rounding, where the computations of the upper/lower intervals are rounded up/down respectively.

In computer graphics, most applications ignore round-off errors and view the result of a shader evaluation as ground truth. Thus, it is sufficient to compute bounds that are conservative only up to machine precision. That is, a shader should never return a result outside the bounds computed by the corresponding bounding shader, but we need no guarantee that the bounds are mathematically conservative. This is ensured if there is a path through the bounding shader that executes the same sequence of floating-point operations, using the same rounding mode as in the original shader. Normally, this is always the case, as each interval operation performs the equivalent floating-point operation on all relevant combinations of extreme values, and then picks the outer bounds using min/max operations.

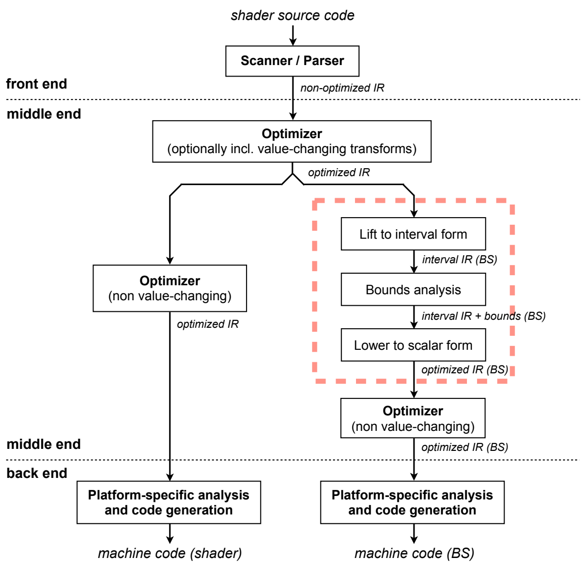

Figure 2: *The middle end of the compiler performs initial optimizations on the intermediate representation (IR) before lifting it to interval form (right) to get a bounding shader (BS). After bounds analysis and lowering to optimized scalar form, further non-value changing optimizations may be applied. The front and back ends are the same as in a traditional shader compiler (left).*

3.4. Application to Shader Languages

In addition to arithmetic operations, shader languages (e.g., RenderMan, GLSL, and HLSL) support functionality such as conditionals, loops, and texture lookups, which must be handled. Many of these features have been extensively studied. Interval-based texture lookups can be done using mipmap hierarchies of minimum and maximum values [MM02]. The bounds on the texture coordinates are used to compute an appropriate mipmap level, from which a number of min/max samples are drawn. Cube map lookups can be handled similarly. Previous work has also explored how to handle noise functions and derivatives [HSS98].

Conditionals present a problem, as whenever overlapping intervals are compared, the condition is not unambiguously true or false. One solution is to use step functions and rewrite conditional expressions as arithmetic [HSS98]. A more general method, which we use, is to evaluate both execution paths and merge the results. This is most easily done by transforming the intermediate representation to *static single assignment* (SSA) form [CFR*91], and treating the Φ functions that are inserted at joints in a non-standard way (they are normally replaced by copies) as selector functions [VAZH*09] .

4. An Optimizing Interval Compiler

In this section, we will introduce a compiler infrastructure and a number of optimizations targeted at generating faster bounding shaders. There are two main steps involved. First, we *lift* the original shader to interval form, i.e., replace scalar instructions by their interval arithmetic equivalents. This lifting step is straightforward, and can be done trivially by a compiler with an intermediate representation (IR) supporting interval instructions.

The second step is to *lower* the shader on interval form to efficient scalar code. The standard approach is to replace each interval instruction by a general sequence of scalar instructions performing the desired interval computation. However, better results can be obtained if we have prior knowledge about what range of values each instruction operates on. Much of this paper deals with automatic extraction of such information through *bounds analysis*.

Figure 2 shows the proposed design of an optimizing compiler for automatic generation of bounding shaders (BS). We will focus on the algorithms enclosed in dashed red, and first look at some examples to highlight the types of optimized interval to scalar conversions that are possible. Then, we will discuss the lifting step and introduce several novel algorithms for compile-time bounds analysis.

4.1. Lowering to Optimized Scalar Form

Most interval operations can be implemented in a variety of ways depending on the generality of the computation. For example, multiplying two positive intervals is much easier than the case where the intervals may overlap zero. Our goal is to select the most compact implementation possible, exploiting bounds information determined at compile time.

We define a table of implementation alternatives for each instruction, along with the necessary conditions. There is always a fully general implementatation, which works under *any* condition. This job is tedious, but fortunately has to be done only once. Note that the listed conditions are applied at compile time, so the implementation must be valid for all possible runtime intervals matching the condition. Next, we will look at a few illustrative examples.

Multiplication The table below shows different possibilities for evaluating the multiplication operator with *scalar-scalar*, *scalar-interval*, and *interval-interval* operands. In this case, knowing the signs of one or both of the operands enables significantly more efficient code (the cases where $\hat{a}$ and $\hat{b}$ are reordered have been left out for brevity):

expr	condition	implementation	#inst
$x \cdot y$	any	$x \cdot y$	1
$x \cdot \hat{a}$	any	$[\min(x\underline{a}, x\overline{a}), \max(x\underline{a}, x\overline{a})]$	4
	$x \geq 0$	$[x\underline{a}, x\overline{a}]$	2
	$x \leq 0$	$[x\overline{a}, x\underline{a}]$	2
$\hat{a} \cdot \hat{b}$	any	$[\min(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}),$ $\max(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b})]$	10
	$\hat{a} \geq 0$	$[\min(\underline{a}\underline{b}, \overline{a}\underline{b}), \max(\underline{a}\overline{b}, \overline{a}\overline{b})]$	6
	$\hat{a} \leq 0$	$[\min(\underline{a}\overline{b}, \overline{a}\overline{b}), \max(\underline{a}\underline{b}, \overline{a}\underline{b})]$	6
	$\hat{a}, \hat{b} \geq 0$	$[\underline{a}\underline{b}, \overline{a}\overline{b}]$	2
	$\hat{a} \leq 0 \leq \hat{b}$	$[\underline{a}\overline{b}, \overline{a}\underline{b}]$	2
	$\hat{a}, \hat{b} \leq 0$	$[\overline{a}\overline{b}, \underline{a}\underline{b}]$	2

Square If the operands of a multiplication come from the same source, we can use a more efficient *square* operator:

expr	condition	implementation	#inst
x^2	any	$x \cdot x$	1
$\hat{a}^2$	any	$[\max(\underline{a}, -\overline{a}, 0)^2, \max(-\underline{a}, \overline{a})^2]$	7
	$\hat{a} \geq 0$	$[\underline{a}\underline{a}, \overline{a}\overline{a}]$	2
	$\hat{a} \leq 0$	$[\overline{a}\overline{a}, \underline{a}\underline{a}]$	2

Absolute Value Some operations can be completely removed under special circumstances. An absolute value, for example, requires no evaluation for positive operands:

expr	condition	implementation	#inst
$\lvert x \rvert$	any	$\lvert x \rvert$	1
	$x \geq 0$	x	0
$\lvert \hat{a} \rvert$	any	$[\max(\underline{a}, -\overline{a}, 0), \max(-\underline{a}, \overline{a})]$	5
	$\hat{a} \geq 0$	$[\underline{a}, \overline{a}]$	0
	$\hat{a} \leq 0$	$[-\overline{a}, -\underline{a}]$	2

Square Root Other instructions are valid only on a limited range, e.g., a square root requires a positive argument. If we at compile time can guarantee the result will be *Not-a-Number* (NaN), we generate a compiler warning (shown in red) as this condition likely indicates a programming error:

expr	condition	implementation	#inst
$\sqrt{x}$	any	$\sqrt{x}$	1
	$x < 0$	NaN	0
$\sqrt{\hat{a}}$	any	$[\sqrt{\underline{a}}, \sqrt{\overline{a}}]$	2
	$\hat{a} < 0$	[NaN, NaN]	0

Additional Notes It is often the sign of operands that differentiate between implementation alternatives, but there are exceptions. The interval pow$(\hat{a}, \hat{b})$ function, for example, is monotonically increasing if $\hat{a} \geq 1$, and decreasing if $\hat{a} \leq 1$. The general case requires 4 pow and 6 min/max instructions, but it can be reduced to only 2 pow in some cases.

In these examples, we have counted each basic instruction (add, mul, neg etc) as a single instruction for simplicity. Assuming the bounding shader is executed on a throughput-oriented architecture, this should be a rough approximation of its execution cost. Naturally, the cost depends on the hardware and the exact implementations chosen.

4.2. Lifting to Interval Form

In the lifting step, we analyze the dependency graph to determine which instructions are operating on interval data. Those instructions are replaced by their equivalents operating on intervals rather than scalar values, e.g., an add instruction is replaced by an interval-add and so on. We assume the user or application specifies which inputs are given as intervals. This can be accomplished by annotating selected inputs with an `interval` keyword [SZAB99], or by designing the compiler API to allow sufficient control. Next, we will look at a simple example. Consider the function:

```
float f(interval float a, interval float b) {
    float c = abs(a);
    return sqrt(c * b);
}
```

After lifting this to interval form and converting to three-address code on SSA form [CFR*91], we get:

expression	#inst
$\hat{a}_0 \leftarrow \hat{a}$	
$\hat{b}_0 \leftarrow \hat{b}$	
$\hat{c}_0 \leftarrow \lvert\hat{a}_0\rvert$	5
$\hat{t}_1 \leftarrow \hat{c}_0 \cdot \hat{b}_0$	10
$\hat{t}_2 \leftarrow \sqrt{\hat{t}_1}$	2
`return` $\hat{t}_2$	
sum:	17

The cost of each interval operation is measured in terms of scalar instructions, assuming the most general implementation of each instruction, as defined in Section 4.1. This is what a compiler not using our optimizations would generate.

4.3. Static Bounds Analysis

As we have seen in Section 4.1, there is a lot to gain from using optimized implementations of the interval operations. We propose to use data flow analysis to determine conservative bounds on each shader variable at compile time. First, initial bounds are assigned to each shader input. The bounds are then propagated through the shader using a generalized form of interval arithmetic. The method is related to constant propagation [WZ91], but instead of compile-time constants, we track bounds for the runtime range of each variable. These are then used to pick the most efficient implementation of each instruction when lowering the code to scalar form. Next, we will go over the details.

Initial Bounds The data type of a variable is the first source of loose conservative bounds, simply due to the range of representable numbers the variable can hold. Most shading languages support at least a subset of the following basic types:

data type	possible range
half, float, double	$\{x \in \mathbb{R} \mid -\infty \leq x \leq \infty, \mathrm{NaN}\}$
integer (n bits)	$\{x \in \mathbb{Z} \mid -2^{n-1} \leq x \leq 2^{n-1}-1\}$
unsigned integer (n bits)	$\{x \in \mathbb{Z} \mid 0 \leq x \leq 2^{n}-1\}$
boolean	$\{$`true`$,$`false`$\}$
signed normalized	$\{x \in \mathbb{R} \mid -1 \leq x \leq 1\}$
unsigned normalized	$\{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$

For aggregate types, such as vectors and matrices, we assign bounds to each element individually based on its basic type.

Shader languages usually also support a number of predefined *state* variables with basic information about the point being shaded. In HLSL, these are accessed using shader input semantics, e.g., `SV_Position` (D3D10), and in GLSL there are built-in variables, such as `gl_FragCoord`. Some state variables have strict limits set by the rendering system. For example, `SV_Position` in the pixel shader specifies the sample position in screen space coordinates (x,y), and the depth z in normalized device coordinates. Hence we know that $z \in [0,1]$ and $x,y \in [0,N]$, where N is the size of the largest supported render target.

To be fully general, we represent a variable's compile-time bounds as a union of zero or more discrete values and zero or more disjoint intervals. Given a variable, x, we define its bounds, B_x, as:

$$B_x = \bigcup_i B_x^i, \qquad \text{where } B_x^i = \begin{cases} x_i, & \text{or} \\ [\underline{x}_i, \overline{x}_i]. \end{cases} \tag{8}$$

Additionally, we have found it useful to store flags indicating whether a variable can be NaN, and whether both positive and/or negative zeros are possible. The latter is particularly useful to optimize division so that $1/[0,x] = [1/x,\infty]$ rather than $[-\infty,\infty]$ if the denominator is known to exclude -0.

Bounds Propagation In the compile-time propagation of bounds information, we use a generalized form of interval arithmetic, which handles bounds on the form in Equation 8. For binary operations, $z \leftarrow x \text{ op } y$, we evaluate all combinations of discrete values and/or intervals in x and y:

$$B_z = \bigcup_{i,j} (B_x^i \text{ op } B_y^j). \tag{9}$$

Any overlapping bounds are merged to create a new disjoint set of bounds representing the possible values of z. Unary operations, $y \leftarrow \text{op } x$, are handled by applying the operator to each B_x^i. Type casts retain bounds as far as possible and may be a source of additional bounds information. For instance, a bool cast to float in GLSL/HLSL will be a variable in $\{0,1\}$ rather than $[-\infty,\infty]$.

In practice, a variable rarely has more than a single interval bound or a few discrete values, but there are cases that give rise to more complex bounds information. For example, branches can introduce different assignments to a variable, and divisions may introduce multi-intervals (Equation 6). Functions that introduce discontinuities, e.g., the step function, is another example. For instance, $\hat{c} \leftarrow \hat{a} \cdot \text{step}(x,\hat{b})$ will be in the range $\{0\} \cup [\underline{a},\overline{a}]$ if $x \in \hat{b}$. Note that if $0 \in \hat{a}$, we would merge the result and only propagate $[\underline{a},\overline{a}]$.

Many shader instructions grow the bounds, and as we start with often very loose compile-time bounds, it appears there would be little to gain from static bounds analysis. Fortunately, there is a long list of shader instructions with strict limits on their output:

instruction	**max output range**
sign(x), step(c,x)	$\{0,1\}$
fract(x), smoothstep(c_1,c_2,x)	$[0,1]$
clamp(x,c_1,c_2)	$[c_1,c_2]$
abs(x), exp(x), pow(x,y), sqrt(x), rsqrt(x), length$(\vec{x})$, acosh(x)	$[0,\infty]$
cosh(x)	$[1,\infty]$
sin(x), cos(x), tanh(x), noise$(\vec{x})$	$[-1,1]$
arcsin(x), arctan(x)	$[-\frac{\pi}{2},\frac{\pi}{2}]$
arccos(x)	$[0,\pi]$
atan2(y,x)	$[-\pi,\pi]$
mod(x,y)	$[\min(y,0),\max(y,0)]$

This shows that even with *unbounded* shader inputs, we are likely to introduce useful bounds information during bounds propagation. Other examples of when an unbounded input can result in reduced bounds is division, e.g., $1/[1,\infty] \in [0,1]$, and squares, $[-\infty,\infty]^2 = [0,\infty]$.

Example The algorithm is best illustrated by an example. Applying the method to the program in Section 4.2 gives:

expression	**bounds**	**#inst**
$\hat{a}_0 \leftarrow \hat{a}$	$\hat{a}_0 \in [-\infty,\infty]$	
$\hat{b}_0 \leftarrow \hat{b}$	$\hat{b}_0 \in [-\infty,\infty]$	
$\hat{c}_0 \leftarrow \lvert\hat{a}_0\rvert$	$\hat{c}_0 \in [0,\infty]$	5
$\hat{t}_1 \leftarrow \hat{c}_0 \cdot \hat{b}_0$	$\hat{t}_1 \in [-\infty,\infty]$	6
$\hat{t}_2 \leftarrow \sqrt{\hat{t}_1}$	$\downarrow$ $\hat{t}_2 \in [0,\infty]$	2
return $\hat{t}_2$		
	sum:	13

In this case, the inputs $\hat{a}$ and $\hat{b}$ can lie anywhere in $[-\infty,\infty]$, as we have no additional metadata. At the $|\cdot|$ instruction, $\hat{c}_0$ is limited to positive numbers, which means the multiplication $\hat{c}_0 \cdot \hat{b}_0$ can be done using an optimized *positive · unknown* interval multiplication (6 vs. 10 instructions).

4.4. Valid Range Analysis

Some operations put restrictions on the range of valid arguments. For example, the argument to a square root must be positive, otherwise NaN is returned. Other examples are:

instruction	**valid input range**
log$(\hat{a})$, sqrt$(\hat{a})$, rsqrt$(\hat{a})$, pow$(\hat{a},x)$	$\hat{a} \in [0,\infty]$
arcsin$(\hat{a})$, arccos$(\hat{a})$, atanh$(\hat{a})$	$\hat{a} \in [-1,1]$
acosh$(\hat{a})$	$\hat{a} \in [1,\infty]$

We can generate more efficient code if we at compile time make the *assumption* that all such instructions will receive valid input at runtime. This puts bounds on the valid range of variables, which can be propagated *backwards* through the code to put stricter bounds on the possible runtime range of variables. Applied to the same example as before:

expression	**bounds**	**valid range**	**#inst**
$\hat{a}_0 \leftarrow \hat{a}$	$\hat{a}_0 \in [-\infty,\infty]$	$\uparrow$ $\hat{a} \in [-\infty,\infty]$	
$\hat{b}_0 \leftarrow \hat{b}$	$\hat{b}_0 \in [-\infty,\infty]$	$\hat{b} \in [0,\infty]$	
$\hat{c}_0 \leftarrow \lvert\hat{a}_0\rvert$	$\hat{c}_0 \in [0,\infty]$	$\hat{a}_0 \in [-\infty,\infty]$	5
$\hat{t}_1 \leftarrow \hat{c}_0 \cdot \hat{b}_0$	$\hat{t}_1 \in [-\infty,\infty]$	$\hat{b}_0 \in [0,\infty]$	2
$\hat{t}_2 \leftarrow \sqrt{\hat{t}_1}$	$\hat{t}_2 \in [0,\infty]$	$\hat{t}_1 \in [0,\infty]$	2
return $\hat{t}_2$			
		sum:	9

Here the values in the *bounds* column have been determined using static bounds analysis in a first pass. We note that the square root puts a limit on its argument, $\hat{t}_1 \geq 0$, for the result to be valid. Tracking this backwards, we find that $\hat{b}_0$ must be positive for the result of $\hat{t}_1 \leftarrow \hat{c}_0 \cdot \hat{b}_0$ to be positive (as $\hat{c}_0$ is known to be positive). Hence, the interval multiplication can be replaced by an efficient 2 instruction version for *positive · positive* interval (Section 4.1).

It is important to point out that since we *assume* each instruction will produce valid results, the generated bounding shader will be undefined if executed for bounds violating this assumption. There is no way to detect this solely based on its output, so valid range analysis must only be used when the application knows it is safe for a given input. This can be ensured by first executing a bounding shader with this optimization disabled (i.e., using NaN checks). Also note that several iterations of static bounds analysis and valid range analysis may be necessary for the best results. After an initial pass, the refined bounds may be propagated further using static bounds analysis. This may in turn introduce possibilities for further improvements in a backwards pass. We iterate until convergence, which in our experience occurs within a few iterations, although pathological cases may exist.

4.5. Dynamic Bounds Assumptions

In general, if an input parameter's bounds are limited, the generated code will be more efficient. Several versions of a bounding shader can be generated with varying extents of input *bounds assumptions*. At runtime, we dynamically select the most restrictive (fastest) version that is valid for the current input. For instance, a restrictive version of the bounding shader may assume a texture access returns a value in $[0,1]$, and more general versions $[0,\infty]$ (e.g., for HDR textures) and $[-\infty,\infty]$. The appropriate version is then determined at runtime when a specific texture is bound to the shader. An alternative approach is to reactively compile specialized bounding shaders when an assumption is met. This is a more viable approach if the input parameter space is large.

4.6. Complex Control Flow

With interval arithmetic, whenever the control flow graph diverges, we must potentially execute both paths and at joints merge the results. This is easy for if-then-else branches (see Section 3.4), but more complex situations may lead to a tree of possible execution paths, where the result is the union of all possible outcomes. Previous work has thus been limited to statically unrollable loops [VAZH*09].

We handle dynamic loops with interval conditions by, at runtime, accumulating the outcome of each iteration where the loop potentially terminates. Consider, for example, a loop on the form `do` $\hat{x} = \ldots$ `while`$(\hat{x} < c)$. When the condition is unambiguously true, i.e., $\overline{x} < c$, we loop as expected. Otherwise, we accumulate the result, $\hat{x}_\cup =$

$[\min(\underline{x},\underline{x}_\cup),\max(\overline{x},\overline{x}_\cup)]$, where $\hat{x}_\cup$ is initially empty, and continue looping until the branch condition is unambiguously false, i.e., $\underline{x} \geq c$. At that point, $\hat{x}_\cup$ holds conservative bounds for the union of all outcomes. At compile-time, the same technique is hard to apply due to often very wide compile-time bounds, and we resort to assuming that variables coming from back edges in the graph are unbounded. In some cases, it may help to iterate through loops a few times to refine the compile-time bounds.

Recursive functions, $\hat{y} = f(\hat{x})$, can be handled similarly in the bounds analysis by first assuming $\hat{x}$ and $\hat{y}$ are unbounded, and by replacing all recursive calls to f by the return value $\hat{y}$. This will give possibly refined compile-time bounds on $\hat{y}$, and the analysis can be iterated. Finally, the refined bounds are used to generate an optimized version of f, which includes real recursive calls. In practice, as noted previously [HAM07], it is hard to guarantee runtime termination of interval code with complex control flow. One way is to manually add explicit termination criteria, e.g., maximum loop iterations or recursion depth, but how to do this automatically is an unsolved problem.

4.7. NaN and Infinity Issues

To guarantee correct results, a rigorous approach to detect and handle floating-point exceptions in the bounding shader must be used. The IEEE 754 standard specifies that all operations that produce a floating-point output must propagate NaNs, *except* for minimum and maximum operations (e.g., `minps` in SSE returns the source operand if one operand is NaN). This presents a problem, as we can construct bounding functions that give erroneous bounds [Pop96]. Another way NaNs can be suppressed is through conditionals involving operands with NaN values.

We have identified three main approaches in the context of bounding shaders (sorted from strict to relaxed):

1. *Propagate NaNs.* If we make sure that NaNs are always propagated, we will get an indication that an invalid operation has occured. This requires explicit NaN detection for each min/max operation (at an overhead of 3 SSE instructions). We also have to ensure NaN values in branches are propagated.
2. *Suppress NaNs.* Assume the original shader never returns NaNs for any input generated by the rendering system – this is not unreasonable, as there is little point trying to bound a shader resulting in NaNs. Under this assumption, we may clamp the inputs to any instruction in the bounding shader that could generate NaNs, as we know the inputs will never fall outside the valid range in the original shader. It also requires us to redefine multiplication, so that $0 \cdot \pm\infty = 0$.
3. *Do not care about NaNs.* This may be the most reasonable approach in certain situations, as some applications can tolerate errors or do not depend on the bounds being strictly conservative for correctness.

Shader	$s(\mathbf{x},\omega_i)$	$\bar{s}(\mathbf{x},\hat{\omega}_i)$		$\bar{s}(\hat{\mathbf{x}},\hat{\omega}_i)$	
	#instr	before	after	before	after
Diffuse	20	2.8×	1.6×	5.8×	3.2×
Phong	149	2.1×	1.2×	4.5×	2.7×
IsoWard	151	3.3×	1.6×	4.7×	2.2×
Ward	174	3.4×	1.8×	5.3×	2.7×
Ashikhmin	486	2.8×	1.9×	3.8×	2.3×
Fractal	447	4.1×	2.6×	4.5×	3.0×
Average		3.1×	**1.8**×	4.8×	**2.7**×

Shader	$s(\mathbf{x},\omega_i)$	$\bar{s}(\mathbf{x},\hat{\omega}_i)$		$\bar{s}(\hat{\mathbf{x}},\hat{\omega}_i)$	
	#cycles	before	after	before	after
Diffuse	20	32	26	58	36
Phong	182	311	205	559	330
IsoWard	192	445	273	510	317
Ward	218	495	325	623	420
Ashikhmin	514	1289	813	1535	898
Fractal	1297	4546	3155	4323	3036
Average		2.3×	**1.6**×	3.0×	**1.9**×

Table 1: *Instruction count (top) and execution time (bottom) compared to the original shaders using SSE for the two types of single-sided bounding shaders needed for lightcuts, before and after our optimizations are applied. The number of instructions is reduced by on average 41.8% and 43.9%, respectively. This saves 32.8–36.7% in clock cycles. All numbers exclude the overhead of NaN handling.*

In all these cases, if a bounding shader returns a NaN, it should be interpreted as *"bounds could not be computed"*. This may trigger further subdivision of the input bounds or a fallback on a more general technique.

4.8. Influence on Standard Compiler Optimizations

Some standard compiler optimizations are relatively more important for bounding shaders than for conventional shaders. For example, common-subexpression elimination is highly useful, as similar computations often are performed for the lower and upper bounds (see, e.g., Equation 3). In many cases only one of the shader outputs needs to bounded, or only the lower or upper bound of an interval computed. An example is depth culling [HAM07], where only $z_{\min}$ is used. Hence, executable backward static slicing [XQZ*05] (cf., dead code elimination) is very important.

As outlined in Section 3.3, we have to be careful not to introduce different rounding errors compared to the original shader in order to guarantee conservative results (up to machine precision). Therefore, we do *not* allow value-changing compiler transformations (i.e., optimizations that may change the floating-point precision) to be applied *after* the shader has been lifted to interval form. However, before the lifting step, such optimizations are allowed with respect to the bounding code, as shown in Figure 2. It is theoretically possible to perform value-changing optimizations also after separating the intermediate representations, but these would have to be performed in tandem on both IRs. We have left this possibility for future work.

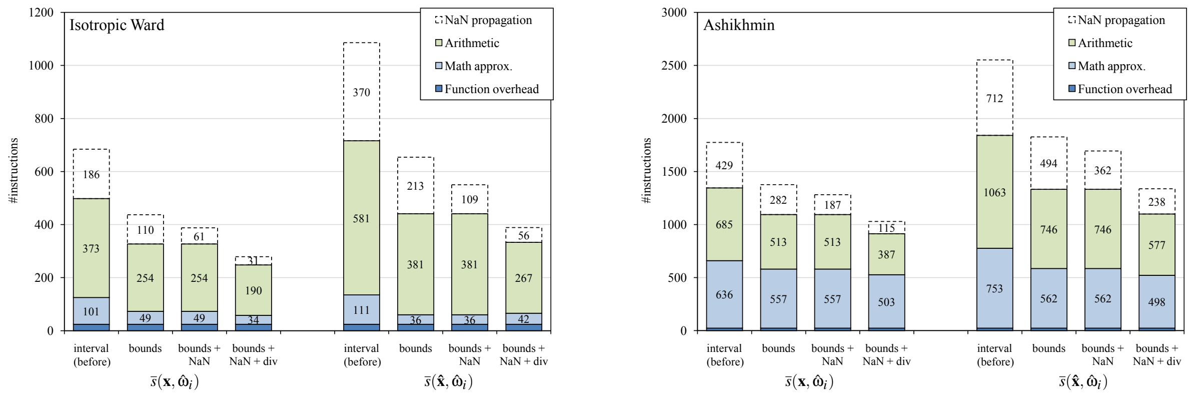

Figure 3: *The effect of our various compiler optimizations on two representative single-sided BRDF bounding shaders using SSE. The baseline is single-sided bounding shaders compiled using standard compiler optimizations (*interval*). Bounds analysis (*bounds*) significantly reduces the code size by tracking compile-time bounds and choosing optimized implementations of the interval operations. The overhead of robust NaN propagation is marked with a dashed box. This decreases when compile-time analysis of NaNs is enabled (*NaN*). Our optimizations targeted at optimizing divisions (*div*), shrink the compile-time bounds by tracking the signs of zeros (see Section 4.3), which further reduces the total instruction count.*

5. Implementation

We have implemented a compiler prototype in C++, which uses metaprogramming to build a program graph; our implementation is based around a class `Var`, which implements all necessary shader functionality in overloaded operators and friend functions. Rather than computing a result, each function inserts a corresponding node in the program graph. Each assignment allocates a new temporary, which directly gives us an intermediate representation in SSA form. Macros are used to insert appropriate constructs for loops and conditionals. The graph nodes implement Equation 9, which is used to propagate bounds information through the graph. The type of each variable, i.e., scalar or interval, is also determined in this step. Last, each node outputs an optimized sequence of scalar instructions. To compile a shader, all variables are replaced by `Var` (or a wrapper to handle vectors); all inputs are assigned a type and initial bounds (default $\pm\infty$), and the shader is executed to generate the bounding shader.

Currently, two back ends have been implemented: one for SSE, which uses auto-vectorization to generate 4-wide data parallel bounding shaders, and one for HLSL. In our prototype, the back ends output C++/HLSL source code, which is finally passed through standard optimizing compilers. This setup has enabled us to quickly prototype and debug various optimizations, as it builds on existing compiler infrastructures to provide input parsing and optimized code generation. A production-quality bounding shader compiler is under development, but that is a much larger project.

6. Results

Here, we present results for a variety of bounding shaders. The effect of our optimizations are measured in terms of instruction count and execution speed for the generated kernels. All SSE/SSE2 code was compiled with Microsoft Visual Studio 2008 using `/O2 /fp:precise` (to ensure floating-point consistency), and executed on an Intel Core 2 Extreme QX9650. We will first look at BRDF bounding shaders for lightcuts and importance sampling purposes. Finally, we will show some GPU results.

6.1. Lightcuts

It has recently been demonstrated [VAZH*09] that interval arithmetic is a viable solution to generate the bounding functions required by lightcuts [WFA*05, WABG06]. Let $s(\mathbf{x}, \omega_i) = f_r(\mathbf{x}, \omega_o, \omega_i)\cos\omega_i$ be a cosine-weighted BRDF, where $\mathbf{x}$ denotes a single shading point including all associated attributes. The upper bound at a single gather point for a cluster of lights, is given by $\overline{s}(\mathbf{x}, \hat{\omega}_i)$, i.e., only the light direction, ω_i, is an interval. Correspondingly, the upper bound for multiple gather points, e.g., to support depth-of-field and motion blur, is $\overline{s}(\hat{\mathbf{x}}, \hat{\omega}_i)$. Here all inputs are intervals.

Table 1 shows instruction counts and execution times using Intel SSE for the two types of bounding shaders, compared to the original point-sampled shaders, $s(\mathbf{x}, \omega_i)$, for a range of physically-based BRDFs. *Fractal* iteratively evaluates a fractal and interpolates between multiple Cook-Torrance lobes, and contains dynamic loops with both interval and scalar loop conditions. The other shaders do not contain complex control flow. With previous techniques, the average instruction counts are 3.1× (single) and 4.8× (multiple gather points) those of the original shaders. With our optimizations, 41.8–43.9% fewer instructions are generated, which translates to a 32.8–36.7% saving in execution time. Note that the optimized bounding shaders compute the *exact* same bounds as before. It is also interesting to note that bounding shaders are in general faster than would be expected based on instruction count. This is likely due to more opportunities for register renaming and latency hiding.

Shader	$s(\mathbf{x},\omega_i)$	$\hat{s}(\mathbf{x},\hat{\omega}_i)$		$\hat{s}(\hat{\mathbf{x}},\hat{\omega}_i)$	
	#instr	before	after	before	after
Diffuse	20	3.4×	2.8×	6.7×	4.9×
Phong	149	2.2×	2.1×	4.8×	4.0×
IsoWard	151	3.4×	2.4×	4.9×	3.2×
Ward	174	3.5×	2.5×	5.4×	3.7×
Ashikhmin	486	2.8×	2.3×	3.8×	3.0×
Fractal	447	4.2×	2.8×	4.6×	3.1×
Average		3.2×	**2.5×**	5.0×	**3.7×**

Table 2: *Instruction count compared to the original shaders using SSE for double-sided bounding shaders, before and after optimization. A prime application is importance sampling. The number of instructions is reduced by on average 23.8% and 27.1%, respectively. All numbers exclude the overhead of NaN propagation.*

The effect of different optimizations are shown in Figure 3 for two representative shaders: an isotropic version ($\alpha_x{=}\alpha_y$) of the Ward BRDF [War92], and the complex anisotropic Ashikhmin model [AS00]. To get these results, we specified loose compile-time bounds on each shader's inputs; all values were assumed to be finite, and all material parameters such as diffuse and specular color, and shininess, were assumed to be non-negative. These values are fetched from the respective material channels (e.g., textures) in the shader setup, but the cost of this has not been included. Note that the use of textures instead of runtime shader constants, has no impact on our optimizations, as we only work with compile-time bounds.

6.2. Importance Sampling

Importance sampling of arbitrary programmable shaders has traditionally been a difficult problem, and specialized methods have been developed for various reflectance models. By hierarchically evaluating $\bar{s}(\mathbf{x},\hat{\omega}_i)$ over the hemisphere, a piecewise constant upper bound is created, which can be used as an importance function [VAZH*09]. Note that $\bar{s}(\mathbf{x},\hat{\omega}_i)$ is the same shader as in lightcuts (see Table 1 and Figure 3 for results).

A natural extension would be to use both bounds of s, i.e., lower and upper, in order to create a more accurate importance function. A step in this direction was taken by Rousselle et al. [RCL*08], although they used precomputed max and average trees rather than a min/max hierarchy. The function $\hat{s}(\mathbf{x},\hat{\omega}_i)$ can be used to build an importance function for a single point, and $\hat{s}(\hat{\mathbf{x}},\hat{\omega}_i)$ for a cluster of shading points. When both bounds are computed, it is harder for the compiler to remove entire computation chains, and we are mainly limited to detect cases where faster implementations of specific operations can be used, e.g., positive instead of general multiplication. Despite this, our optimizations reduce the instruction count by around 25% (see Table 2).

It is interesting to compare the instruction counts for single vs double-sided intervals in Tables 1 and 2. With standard compiler optimizations, going from double-sided to code specialized for single-sided intervals, only saves on average 5%. With our optimizations, the savings are ∼27%. For *Fractal*, the difference is smaller due to control flow making it harder to eliminate long computations chains, but there is still plenty of optimization potential locally.

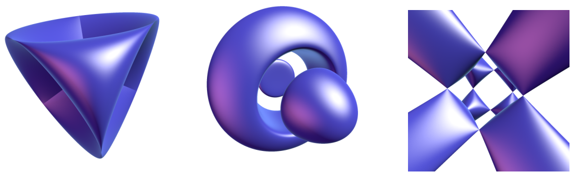

Figure 4: *Examples of implicit surfaces evaluated using interval arithmetic to find the first intersection along each ray.*

6.3. Implicit Surfaces

To measure the performance of optimized bounding code on the GPU, we have implemented a simple ray tracer for implicit surfaces, $f(x,y,z)=0$, that runs in the pixel shader. The algorithm hierarchically evaluates the surface equation using interval arithmetic to compute its *minimum*, in order to locate the first intersection along each ray using an existing algorithm [Mit90]. Figure 4 shows renderings of three well-known implicit surfaces. The table below summarizes the number of generated shader instructions for the bounding kernels, and their execution times on an NVIDIA GTX285 GPU. The instructions are counted per-component in the case of vector operations. As can be seen, the reduction in instruction count is 20–35%, and in clock cycles 15–31%.

#instructions	**Steiner**	**Mitchell**	**Kummer**
original	13	16	20
interval (before)	74	51	64
interval (after)	48 (-35.1%)	41 (-19.6%)	44 (-31.3%)
#cycles	**Steiner**	**Mitchell**	**Kummer**
original	11.5	14.9	17.5
interval (before)	62.7	44.8	55.9
interval (after)	43.6 (-30.5%)	38.3 (-14.6%)	41.4 (-25.8%)

7. Conclusions and Future Work

We believe techniques for automatically extracting shader bounds will be increasingly important to close the gap between artistic control and fast rendering. In this paper, we have showed that bounding shaders based on interval arithmetic can be significantly optimized by performing bounds analysis and case selection at compile time. To the best of our knowledge, this is the first time data flow analysis has been used to optimize interval code generation. Although we have focused on computer graphics applications, many of the techniques would be directly applicable in many other fields. It is important to remember that our focus is entirely on compiler optimizations, and that the generated code computes the exact same bounds as before, only faster.

All of our results assume bounds are tracked individually, with no knowledge about the relationship between, e.g., the components of vectors. With higher-level information, e.g., specifying that vectors are normalized, it is possible to further optimize the code. The effect of this can be simulated by inserting clamps after dot products of normalized vectors, which saves an additional 13.3%–28.4% for the Ashikhmin shaders. It would be interesting to explore these kinds of optimizations and generalizations to higher-order arithmetics.

Acknowledgements We acknowledge support from the Swedish Foundation for Strategic Research. In addition, Tomas Akenine-Möller is a *Royal Swedish Academy of Sciences Research Fellow* supported by a grant from the Knut and Alice Wallenberg Foundation. We are grateful to all reviewers for their valuable feedback.

References

[AS00] ASHIKHMIN M., SHIRLEY P.: An Anisotropic Phong BRDF Model. *Journal of Graphics Tools, 5*, 2 (2000), 25–32. 9

[ASS04] AKKAŞ A., SCHULTE M. J., STINE J. E.: Intrinsic Compiler Support for Interval Arithmetic. *Numerical Algorithms 37*, 1–4 (2004), 13–20. 2

[BH98] BERZ M., HOFFSTÄTTER G.: Computation and Application of Taylor Polynomials with Interval Remainder Bounds. *Reliable Computing, 4* (1998), 83–97. 2

[CFD08] COLLANGE S., FLÓRES J., DEFOUR D.: A GPU interval library based on Boost interval. In *RNC9, Real Numbers and Computers* (2008), pp. 61–72. 2

[CFR*91] CYTRON R., FERRANTE J., ROSEN B. K., WEGMAN M. N., ZADECK F. K.: Efficiently Computing Static Single Assignment Form and the Control Dependence Graph. *ACM Transactions on Programming Languages and Systems, 13*, 4 (1991), 451–490. 4, 5

[CS93] COMBA J. L. D., STOLFI J.: Affine Arithmetic and its applications to Computer Graphics. In *Proceedings of VI SIBGRAPI 1993* (1993), pp. 9–18. 2

[Duf92] DUFF T.: Interval Arithmetic and Recursive Subdivision for Implicit Functions and Constructive Solid Geometry. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (1992), vol. 26, pp. 131–138. 2

[GK94] GREENE N., KASS M.: Error-Bounded Antialiased Rendering of Complex Environments. In *Proceedings of SIGGRAPH 1994* (1994), ACM, pp. 59–66. 2

[HAM07] HASSELGREN J., AKENINE-MÖLLER T.: PCU: The Programmable Culling Unit. *ACM Transactions on Graphics, 26*, 3 (2007), 92:1–10. 1, 2, 7

[HHHJ07] HIJAZI Y., HAGEN H., HANSEN C. D., JOY K. I.: Why Interval Arithmetic is so Useful. In *Visualization of Large and Unstructured Data Sets* (2007), pp. 148–163. 2

[HL90] HANRAHAN P., LAWSON J.: A Language for Shading and Lighting Calculations. In *Computer Graphics (Proceedings of SIGGRAPH 90)* (1990), vol. 24, pp. 289–298. 2

[HMAM09] HASSELGREN J., MUNKBERG J., AKENINE-MÖLLER T.: Automatic Pre-Tessellation Culling. *ACM Transactions on Graphics, 28*, 2 (2009), 19:1–10. 2

[HS98] HEIDRICH W., SEIDEL H.-P.: Ray-Tracing Procedural Displacement Shaders. In *Graphics Interface* (1998), pp. 8–16. 2

[HSS98] HEIDRICH W., SLUSALLEK P., SEIDEL H.-P.: Sampling Procedural Shaders using Affine Arithmetic. In *Proceedings of SIGGRAPH 1998* (1998), vol. 17, ACM, pp. 158–176. 1, 2, 3, 4

[Kas92] KASS M.: CONDOR: Constraint-Based Dataflow. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (1992), vol. 26, pp. 321–330. 2

[Mit90] MITCHELL D. P.: Robust Ray Intersection with Interval Arithmetic. In *Proceedings on Graphics interface '90* (1990), pp. 68–74. 2, 9

[MM02] MOULE K., MCCOOL M. D.: Efficient Bounded Adaptive Tessellation of Displacement Maps. In *Graphics Interface* (2002), pp. 171–180. 3

[Moo66] MOORE R. E.: *Interval Analysis*. Prentice-Hall, 1966. 2

[PBBR07] PARKER S. G., BOULOS S., BIGLER J., ROBISON A.: RTSL: a Ray Tracing Shading Language. In *Proceedings of IEEE Symposium on Interactive Ray Tracing* (2007), pp. 149–160. 2

[Pop96] POPOVA E. D.: Interval operations involving NaNs. *Reliable Computing, 2*, 2 (1996), 161–165. 7

[RCL*08] ROUSSELLE F., CLARBERG P., LEBLANC L., OSTROMOUKHOV V., POULIN P.: Efficient Product Sampling using Hierarchical Thresholding. *The Visual Computer (Proceedings of CGI 2008), 24*, 7-9 (2008), 465–474. 9

[RKC02] REDON S., KHEDDAR A., COQUILLART S.: Fast Continuous Collision Detection between Rigid Bodies. *Computer Graphics Forum (Proceedings of Eurographics), 21*, 3 (2002), 279–288. 2

[Sny92] SNYDER J. M.: Interval Analysis for Computer Graphics. In *Computer Graphics (Proceedings of SIGGRAPH 92)* (1992), vol. 26, pp. 121–130. 2

[SS00] SCHULTE M. J., SWARTZLANDER JR. E. E.: A Family of Variable-Precision Interval Arithmetic Processors. *IEEE Transactions on Computers, 49*, 5 (2000), 387–397. 2

[SZAB99] SCHULTE M. J., ZELOV V., AKKAS A., BURLEY J. C.: The Interval-Enhanced GNU Fortran Compiler. *Reliable Computing 5*, 3 (1999), 311–322. 2, 5

[VAZH*09] VELÁZQUEZ-ARMENDÁRIZ E., ZHAO S., HAŠAN M., WALTER B., BALA K.: Automatic Bounding of Programmable Shaders for Efficient Global Illumination. *ACM Transactions on Graphics, 28*, 5 (2009), 142:1–9. 1, 2, 4, 6, 8, 9

[Ž05] ŽILINSKAS J.: Comparison of Packages for Interval Arithmetic. *Informatica 16*, 1 (2005), 145–154. 2

[WABG06] WALTER B., ARBREE A., BALA K., GREENBERG D. P.: Multidimensional Lightcuts. *ACM Transactions on Graphics, 25*, 3 (2006), 1081–1088. 1, 8

[War92] WARD G. J.: Measuring and Modeling Anisotropic Reflection. *Computer Graphics (Proceedings of ACM SIGGRAPH), 26*, 2 (1992), 265–272. 9

[WFA*05] WALTER B., FERNANDEZ S., ARBREE A., BALA K., DONIKIAN M., GREENBERG D. P.: Lightcuts: A Scalable Approach to Illumination. *ACM Transactions on Graphics, 24*, 3 (2005), 1098–1107. 1, 8

[WZ91] WEGMAN M. N., ZADECK F. K.: Constant Propagation with Conditional Branches. *ACM Transactions on Programming Languages and Systems, 13*, 2 (1991), 181–210. 5

[XQZ*05] XU B., QIAN J., ZHANG X., WU Z., CHEN L.: A Brief Survey of Program Slicing. *SIGSOFT Software Engineering Notes, 30*, 2 (2005), 1–36. 7

[ZRLK07] ZHANG X., REDON S., LEE M., KIM Y. J.: Continuous Collision Detection for Articulated Models using Taylor Models and Temporal Culling. *ACM Transactions on Graphics, 26*, 3 (2007), 15:1–10. 2