Appears in ACM SIGGRAPH Asia 2009

Automatic Bounding of Programmable Shaders for Efficient Global Illumination

Edgar Velázquez-Armendáriz Shuang Zhao Miloš Hašan Bruce Walter Kavita Bala

Cornell University*

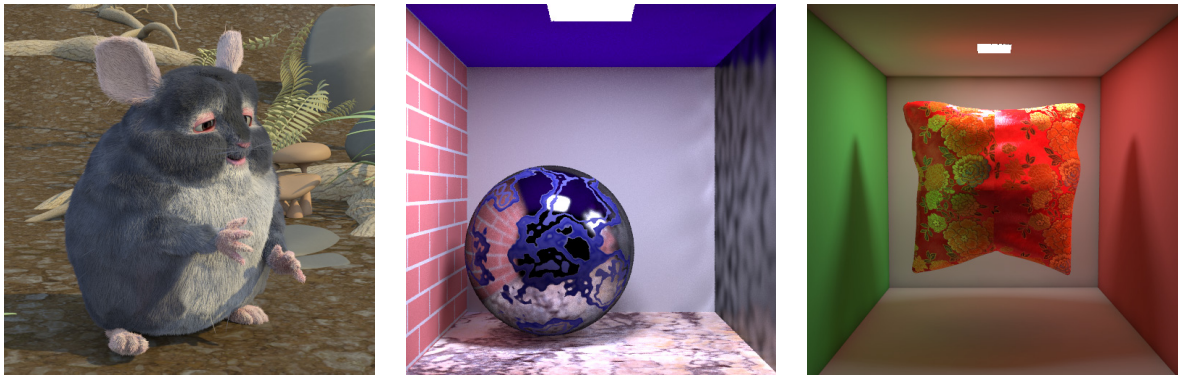


Figure 1: *Our results combining programmable shaders and global illumination: a movie character with 11 million triangles rendered using the Kajiya-Kay shader (left); a glossy ball with a marble-like pattern generated by a procedural shader (center); a pillow using a shader that implements a spatially varying BRDF (right). (l), (r) are rendered with multidimensional lightcuts, and (c) is rendered using photon mapping.*

Abstract

This paper describes a technique to automatically adapt programmable shaders for use in physically-based rendering algorithms. Programmable shading provides great flexibility and power for creating rich local material detail, but only allows the material to be queried in one limited way: point sampling. Physically-based rendering algorithms simulate the complex global flow of light through an environment but rely on higher level information about the material properties, such as importance sampling and bounding, to intelligently solve high dimensional rendering integrals.

We propose using a compiler to automatically generate interval versions of programmable shaders that can be used to provide the higher level query functions needed by physically-based rendering without the need for user intervention or expertise. We demonstrate the use of programmable shaders in two such algorithms, multidimensional lightcuts and photon mapping, for a wide range of scenes including complex geometry, materials and lighting.

CR Categories: I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism—Color, shading, shadowing, and texture; G.1.0 [Numerical Analysis]: General—Interval arithmetic

Keywords: global illumination, many-lights, interval arithmetic

*Department of Computer Science & Program of Computer Graphics. {edgar,bjw}@graphics.cornell.edu, {szhao,mhasan,kb}@cs.cornell.edu.

1 Introduction

Shading algorithms in graphics span two extremes: physically-based approaches that aim to accurately capture the global interactions of light and materials, and programmable shading based approaches that offer a high degree of flexibility and user control. Efficient physically-based algorithms typically rely on detailed mathematical knowledge to derive the functionality they need and are thus limited to well-defined analytic material models. Programmable shaders are extremely flexible and allow for artistic control and thus have a crucial role in the production pipelines of the film industry, but their black box nature provides limited knowledge about their mathematical properties. It is not currently possible to automatically combine the flexibility of programmable shading with the realism and efficiency of advanced physically-based rendering. This paper aims to bridge the gap between these two disparate approaches.

Global illumination rendering simulates the complex flow of light and material scattering properties of the real world. Due to the high dimensionality and complexity of the integrals, brute-force evaluation is prohibitively slow. Instead efficient algorithms rely on additional information about the material models, such as importance sampling and bounding functions, to intelligently sample potential light paths. However creating these functions for novel material models requires considerable time and expertise. Thus implementations are usually limited to a small range of standard BRDF models with well-known properties.

In applications like games and movies, programmable shading is critical for artistic control. However, once the goal of flexibility is achieved, these applications would often also like to leverage the benefits and realism of physically-based algorithms robustly and automatically. For example, existing approaches manually convert shaders to interface with physically-based algorithms (e.g., [Tabellion and Lamorlette 2004]).

We propose a system that takes programmable shaders and automatically generates the additional access methods needed by efficient physically-based rendering algorithms, namely bounding

queries and guaranteed quality importance sampling. A compiler generates additional versions of the shader using interval arithmetic as needed to compute upper bounds over input ranges. Adaptive importance sampling functions are also generated using these interval versions, with guaranteed quality even for very complex shaders. We demonstrate how this approach can interface programmable shading with rendering algorithms such as photon mapping [Jensen 1996], and lightcuts [Walter et al. 2005; Walter et al. 2006].

Our contributions include:

- Formulation of the interface between programmable shading and physically-based shading;
- An automated compiler that respects this interface by compiling shaders to create interval-based shaders that can be used in a wide variety of rendering algorithms;
- Interval-based techniques to adaptively bound shaders, thus enabling importance sampling and rendering with error bounds;
- Demonstration of programmable shaders in lightcuts and photon mapping algorithms.

Our work is based on a few assumptions. The context of this research is in ray tracing based applications. We also fundamentally assume the shader is a BRDF (or close to being one), though we do not enforce this. A user can decide if their shaders are appropriate to use in the context of physically-based rendering. For example, coupling an NPR shader with photon mapping might not produce any reasonable output.

2 Related Work

There is a vast body of literature on programmable shading though most of the focus has been on efficiently handling shaders, rather than on generalizing the support for physically-based rendering.

Production quality shaders are often expensive to execute [Tabellion and Lamorlette 2004; Jensen and Christensen 2007]. One approach to make them manageable has been through automatic simplification. Olano et al. [2003] and Pellacini [2005] automatically process shader source code to produce new shaders with reduced computational requirements. These shaders are aimed at hardware rendering [Olano et al. 2003] and for the REYES engine [Pellacini 2005]. However these simplified versions still perform linearly with the number of lights and samples per pixel.

Another case for shader conversion comes from cinematic relighting, in which the shaders that meet appropriate restrictions are transformed to a format suitable for GPU execution for accelerated or interactive use. Among these systems LPics [Pellacini et al. 2005] requires the user to manually convert the shaders, while Lightspeed [Ragan-Kelley et al. 2007] does it automatically. However these techniques do not interface with physically-based rendering algorithms.

Shaders are not limited to artistic usage. Due to their flexibility they have also been used to implement compact representations of measured real world BRDFs such as [Latta and Kolb 2002; Lawrence et al. 2004] by storing compressed coefficients in textures and reconstructing the functions during the shader evaluation [Olano et al. 2003; Wang et al. 2008].

There has been prior work on using interval arithmetic to analyze shaders; the work of [Heidrich et al. 1998] is most closely related to this paper. They generated affine interval arithmetic versions of Renderman surface shaders to provide guaranteed-quality area sampling of parametric surfaces with procedural materials. Branches and control flow were converted to expressions using step functions. The proposed applications were computing form factors in diffuse radiosity and anti-aliased rasterization of procedural textures. While similar in general approach, our work has several key differences. By analyzing the directional as well as the spatial components of procedural shaders, our method can be used with a much wider range of rendering algorithms including photon mapping and lightcuts. Instead of step functions, we use an extension of single static assignment (SSA) form [Cytron et al. 1991] to handle control flow. SSA is widely used in the compiler community and has the advantages of handling more general control flow, producing more efficient code, and often producing tighter bounds than the equivalent step-function conversion.

Interval-based analysis has been applied to ray tracing without tesselation [Heidrich and Seidel 1998], procedural displacement-map shaders for adaptive tesselation [Moule and McCool 2002], or pretesselation culling [Hasselgren et al. 2009] during rasterization. Other graphics applications of interval analysis include providing guarantees in interpolating radiance samples [Bala et al. 1999], for intersecting implicit surfaces [Snyder 1992; Flórez et al. 2006], and in plotting functions [Comba and Stolfi 1993; Tupper 1996].

Some prior work has reported performance gains by using more computationally expensive interval variants such as affine [Comba and Stolfi 1993; Heidrich et al. 1998] or Taylor [Hasselgren et al. 2009] intervals that typically produce tighter bounds. In our experiments we found standard interval arithmetic to be more cost-effective for our usage than these more complex variants but further testing is warranted on this question.

3 Problem

In this section we formulate the problem that arises when coupling programmable shaders with physically-based global illumination algorithms. A shader allows us to evaluate the color contribution at a point from a single direction or light. If we only have a few point lights, we can compute pixels by brute-force evaluation of every light at every pixel. However, as the light sources become more numerous and complex, such as area lights, environment maps, and indirect illumination, this approach quickly becomes infeasible. Advanced rendering algorithms such as photon mapping and multidimensional lightcuts use Monte Carlo coupled with carefully chosen sparse sampling to compute complex illumination at much lower cost. However to achieve this, they require that the shader support additional types of operations beyond just evaluating the shader for particular positions and directions. In this paper we automatically generate two new abilities for procedural shaders: the ability to importance sample directions according to the shader, and the ability to bound the shader over spatial and/or directional regions.

Directional importance sampling is an important component in nearly all Monte Carlo physically-based rendering algorithms, including path tracing, photon mapping and all algorithms built on them. Bounding is less widely used in rendering but can be an essential component in robust scalable algorithms such as lightcuts. We will first review bounding, how it is used in lightcuts, and how we bound shaders before discussing how we leverage shader bounding support to generate importance sampling.

3.1 Background

Instant radiosity-based [Keller 1997] formulations of rendering convert all lighting into virtual point lights (VPLs). Each point is then shaded by summing the contribution of all the VPLs. Using

the notation of lightcuts, the radiance at a point x is:

$$L_r(x, \omega_{\text{eye}}) \quad = \quad \sum_i M_i G_i V_i L_i \tag{1}$$

where, x is the point being shaded, i is the i-th VPL, the summation is over all the VPLs, ω_{eye} is the viewing direction, and the M, G, V, and L terms are the material, geometry, visibility and intensity terms between point x and VPL i.

Lightcuts [Walter et al. 2005] approximates this summation using a hierarchical clustering of lights, called the light tree, to achieve scalable rendering with many lights. The illumination at a point (called a gather point) is accurately estimated by adaptively picking a clustering, called a cut, from the light tree. The contribution from each light cluster is estimated based on the evaluation of one of its members, called the representative light. The algorithm starts with an initial cut, such as the root of the light tree, and then refines it until the error bound for each cluster is below a given threshold of the total. The error bounds for each cluster L_C are computed by bounding the M and G terms individually (respectively called M^{ub} and G^{ub}), and using the upper bound of one for visibility.

$$\text{error}(L_C) \quad \leq \quad M^{ub} G^{ub} \sum_{i \in L_C} L_i \tag{2}$$

Multidimensional lightcuts [Walter et al. 2006] (MDLC) extends to summation over multiple gather points to support effects like anti-aliasing, depth-of-field, and motion blur by summing over the connections between the j-th gather point and the i-th light:

$$L_{pixel} \quad = \quad \sum_{ij} S_j M_{ji} G_{ji} V_{ji} L_i \tag{3}$$

where, S is the strength of a gather point. It generalizes the notion of a cut to apply to the cartesian product of a gather tree and a light tree and adaptively picks a subset of all the gather-light interactions to approximate the total shading in a pixel. The error bound between a gather cluster X_C and a light cluster L_C is:

$$\text{error}(X_C, L_C) \quad \leq \quad M^{ub} G^{ub} \sum_{j \in X_C} S_j \sum_{i \in L_C} L_i \tag{4}$$

The original Lightcuts papers [Walter et al. 2005; Walter et al. 2006] provided bounding functions for a few standard material models, but required implementors to hand-craft bounding functions for any other material type they wanted to use. One goal of this paper is to support the use of arbitrary procedurally defined materials by automatically generating the corresponding bounding functions, M^{ub}. Similar techniques could be used to support light shaders with their corresponding bounds, G^{ub}, but for the rest of this paper, we will focus on material or surface shaders.

3.2 Generalization of the shaders

Let us assume we are given a shader program (a *surface shader* in the Renderman Interface nomenclature) that can compute the shading for a given point and light direction and that we will call $s_1(x, \omega)$. The shading point x represents not only the spatial location of that particular surface sample, but also all other associated parameters with that position such as the geometric normal and shading normal (e.g., for bump maps). For physically-based algorithms, the shading function is equal to the cosine-weighted BRDF and is the same as the M_i term in lightcuts:

$$s_1(x, \omega_i) = \quad f_r(x, \omega_{\text{eye}}, \omega_i) \cos(\omega_i) \quad = M_i \tag{5}$$

where ω_i is the direction from x to light i.

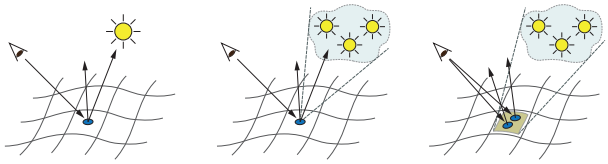


Figure 2: *The different types of shader bounding versions. The standard shader s_1 evaluates the interaction between a gather point and a light. Shaders s_2 and s_3 bound the interaction between a cluster of lights and a single gather point or a cluster of gather points respectively.*

To support one of our target physically-based rendering algorithms, lightcuts, we need to augment this shader with two additional versions that compute bounds over ranges of inputs to the shader: from a single point over a set of directions, and over both a set of positions and directions. The bounding shader versions are illustrated in Figure 2.

The input shader can evaluate the contribution of a light to a point x. We refer to this shader as s_1 (Figure 2a). Further, we create two new versions of the shader, $s_2(x, L_C)$ and $s_3(X_C, L_C)$ (Figures 2b and 2c) that can bound the contribution at point x or a cluster of points X_C from a cluster of lights L_C respectively. Once we have this interface we can compute M^{ub} and evaluate Equations 2 and 4 to support programmable shaders with lightcuts.

There are many ways in which bounding functions can be created. We chose to use interval arithmetic [Moore and Bierbaum 1979]. We note that we also tried linear [Moore and Bierbaum 1979] and affine arithmetic [Comba and Stolfi 1993] for bounding, but we found the relative gains of using these formulations (in terms of tighter bounding functions) did not justify the much higher cost in computation that they impose.

3.3 Generating importance functions

Another critical computation for rendering is importance sampling which is necessary for reducing variance during Monte Carlo sampling, for example in path tracing or photon tracing. But with programmable shaders one does not have an analytical representation of the shader, and sampling must be done by treating the shader as a black box using non-adaptive sampling. Standard cosine-weighted hemispherical sampling can be used but is only ideal for diffuse surfaces and becomes an increasingly poor importance sampling function as materials become less lambertian. Thus, using programmable shading with algorithms like photon mapping or path tracing can be highly inefficient for general materials.

We propose using our shader bounding functions to automatically generate a good importance function for arbitrary shaders. Let the set L_S be the sphere of outgoing directions centered at the gather point x, and $\{L_{S_i}\}_{i=1}^n$ a partition of L_S. The maximum of $s_2(x, L_{S_i})$ defines a conservative estimate of the shader over the solid angle Ω_i subtended by the i-th partition element. Applying this bounding over the hemisphere creates a piecewise constant upper bound to the shader function that we then use for importance sampling. Constructing this importance function is somewhat expensive but for strongly directional shading functions it can greatly reduce the variance in the directional sampling.

Discussion: Note that we also considered sampling the shader

function to construct a shader estimate as an importance function. But this approach is not robust, and can be quite problematic for highly glossy materials unless an extremely large number of samples are used to compute the estimate. A poor importance function *increases*, rather than decreases, variance. To avoid this problem we chose to construct conservative bounds on the shader function to drive importance sampling.

4 Shading Language and Compiler

Given a programmable shader function s_1 our goal is to automatically generate s_2 and s_3 to enable coupling the programmable shader with global illumination algorithms.

Programmable shaders are commonly written in languages which follow the imperative paradigm. The production artists who write shaders would likely find it inconvenient to manually write multiple versions of the same shader to support bounds and importance sampling like in [Parker et al. 2007]. Instead, we only require the user to write the s_1 shader, and propose an automated compilation technique to generate the s_2 and s_3 versions of s_1 without manual intervention.

4.1 Shading Language

Our shading language, inspired by Renderman surface shaders, provides a set of predefined variables related to a single gather point being shaded and restricts the evaluation to a single light. In other words, there is no `illuminate` loop – the renderer takes care of looping over light sources. These variables are:

- `P` - position,
- `N` - normal vector,
- `L` - light direction,
- `I` - ray incoming direction,
- `S, T` - texture coordinates.

The language has two main types: **`float`** and a 3D **`vector`**. We also support Renderman-style keywords like **`surface`**, **`color`** and **`point`**, which are just synonyms for **`vector`**. Note that the `N`, `L` and `I` input variables are already normalized, unlike in Renderman. A simple example shader in our language is:

```
// a simple diffuse shader
color diffuse(color Kd)
{
    return Kd * max(dot(N, L), 0);
}
```

The types of expressions in this shader can be derived easily:

```
L : vector
N : vector
Kd : vector
dot(N, L) : float
max(dot(N, L), 0) : float
Kd * max(dot(N, L), 0) : vector
```

4.2 Lifting Shaders to Intervals

To produce s_2 and s_3 from s_1, we "lift" the types of some predefined input variables to intervals. This lifting will automatically propagate through all other variables and function calls that depend on the lifted inputs. The interval versions of the shaders will contain two new types: **`interval`** and **`box`** (a 3D interval).

If the shader code is simply a sequence of variable assignments, ending in a **`return`** statement, this is straightforward to do. Lifting to s_2 is achieved by replacing the type of `L`, the light direction, from **`vector`** to **`box`**. As an example, lifting our simple diffuse shader will result in the following new expression types, which can be automatically derived by structural recursion on the program:

```
L : box
N : vector
Kd : vector
dot_vb(N, L) : interval
max_is(dot_vb(N, L), 0) : interval
mul_vi(Kd, max_is(dot_vb(N, L), 0)) : box
```

Note that we also replaced the `dot` and `max` functions and the `*` operator by their lifted versions, denoting the argument types by a string following the function name. The letters `s`, `i`, `v`, and `b` stand for scalar, interval, vector, and box, respectively. The generated shader will thus look like:

```
box diffuse(vector Kd)
{
    return mul_vi(Kd, max_is(dot_vb(N, L), 0));
}
```

Generating the s_3 shader is analogous, with the difference that all predefined inputs are lifted, rather than just the light direction `L`.

4.3 Lifting if-then-else branches

In the discussion above, we assumed that the shader code is simply a sequence of assignments. However, if-then-else branches are a very common tool in procedural shaders. For example, movie studios often use *übershaders* that support a large number of features, each of which can be turned on or off. A typical übershader might be structured as

```
if (condition1)
{
    // compute feature 1
}
if (condition2)
{
    // compute feature 2
} else {
    // compute simple approximation to feature 2
}
...
```

The key problem in intervalizing if-then-else branches is that a condition containing interval variables can be both true and false, so executing *both* branches is sometimes necessary. However, the two branches might make completely different (and conflicting) changes to the program state. The work of [Heidrich et al. 1998] translates if-statements to three-way branches, and briefly notes that some variables will have to be renamed; however, this becomes non-trivial for complex nested branching and multiply-assigned variables. We instead propose a more general approach based on a standard compiler technique called *static single assignment* (SSA) form. SSA is an intermediate code representation in which every variable is assigned exactly once. We handle if-then-else constructs by executing both branches, and then *merging* variables that correspond to the same variable in the original code. For example, the code

```
y = z = 0;
if ( x > 0 ) y = 1;
else z = 1;
```

will be translated to

```
y1 = 0;
z1 = 0;
y2 = 1;
z2 = 1;
y3 = phi(greater(x, 0), y2, y1);
z3 = phi(greater(x, 0), z1, z2);
```

Here `greater` is an intervalized comparison function that returns a tri-state boolean (can be true, false, or both), and `phi` is a merging function that returns either its second argument, its third argument, or the interval union of them, depending on whether the comparison result is true, false, or both respectively. This approach scales to an arbitrary number of arbitrarily nested branches. Note that unlike standard SSA usage where the `phi` functions are typically eliminated at a later compilation stage, our `phi` functions must be kept and translated into runtime code for performing interval union.

We currently support iteration using loop unrolling for a finite number of iterations and use the approach for conditionals explained above. We leave support for more general looping for future work.

5 Importance Sampling and Bounding

We now show how our interval-based shaders can be used to provide the functionality needed by physically-based renderers: importance sampling and bounding.

5.1 Importance Function Generation

The goal of importance sampling is to generate samples whose probability density is proportional to the function values at those points. For simple functions, an exact importance sampling can be achieved by computing the cumulative distribution function and inverting it. However for arbitrary functions this is not possible, and we will have to use an approximate importance sampling.

An importance sampling function for $s(x, \omega)$ returns a direction ω and the probability density, $p(\omega)$ of choosing it. We want to minimize the value $s(x, \omega)/p(\omega)$ over all ω. Our approach is to construct an adaptive piecewise-constant function $g(\omega)$ that is an upper bound on the target function s. Using a piecewise-constant g makes it easy to importance sample g exactly and using an upper bound makes it easy to bound the worst case using:

$$\frac{s(x,\omega)}{p(\omega)} = \frac{s(x,\omega)}{g(\omega)/\int g(\omega)d\omega} \leq \int g(\omega)d\omega \tag{6}$$

Thus we can generate a good importance sampling by minimizing the volume of g. Moreover knowing the volume of g provides guarantees about the quality of the importance sampling in terms of maximum values and variance.

We characterize g by representing the sphere of directions using a cube-map decomposition and constructing an adaptive quad-tree on each face. We use the interval bounding shader s_2 to compute an upper bound on s over all the directions corresponding to a quad-tree cell. We iteratively refine the quad-tree cell with the largest volume (value times solid angle) until some termination criteria is met (currently we use a fixed total number of refinement steps). Note that g is piecewise constant in cube area, rather than solid angle, requiring a small correction factor when computing the probabilities. When sampling from g we use a discrete probability to pick the quad-tree cell and then a uniform distribution within the cell.

Lightcuts using the shader functions. Once we have the shading function s_1 and the bounding functions s_2 and s_3, it is simple to use them in a lightcuts framework. When evaluating representative pairs (gather and light points) for a cluster we use:

$$M_i = s_1(x_i, \omega) \tag{7}$$

For the upper bound M_C^{ub} we use either the upper bound given by s_2 or s_3 depending on whether the gather cluster consists of a single point or multiple points. If the gather cluster X_C is a single point x_i then we use:

$$M_C^{ub} = s_2(x_i, \Omega_C) \tag{8}$$

otherwise we use:

$$M_C^{ub} = s_3(X_C, \Omega_C) \tag{9}$$

where Ω_C is an interval vector containing all possible directions from points in the gather cluster X_C to points in the light cluster L_C.

5.2 Texture support

In our system, a shader queries a texture using its normalized coordinates $(s, t) \in [0, 1] \times [0, 1]$. During the lifting process these coordinates may turn into an interval box. Naïvely checking all pixels within a region to find the exact maximum and minimum, could be prohibitively expensive. Instead we use the same fast but conservative, mipmap-inspired solution as [Moule and McCool 2002; Hasselgren and Akenine-Möller 2007]. As with a mipmap, we precompute and store lower resolution versions of the texture, except that these store the minimum and maximum texture values instead of averages. During lookup we select an appropriate resolution level and query a constant number of its texels that completely cover the desired region. Repeating textures are handled by breaking the interval box at the texture borders into at most four regions.

5.3 Noise

Procedural shaders often use continuous noise functions to model the appearance of surfaces in lieu of finite resolution textures [Apodaca and Gritz 1999]. We used a table-driven noise implementation based on the open source program Pixie [2008] which returns values between zero and one. Noise functions are typically high frequency and complex making them hard to bound tightly using standard interval techniques. Thus we follow the same approach as [Heidrich et al. 1998] and only try to compute tight bounds for very small input intervals, otherwise we fallback to the default interval bound of $[0, 1]$ for the noise function.

6 Results

We now demonstrate our system using different scenes to highlight various features of our system. Our prototype system is mostly unoptimized and it is written in Java, except for the ray intersection code implemented in C++, and the compiler written in Haskell. All reported timings correspond to a dual quad-core Xeon E5440 2.83 GHz system with 16GB of RAM using Sun's Java 1.6.0-11.

6.1 Scenes and Shaders

We evaluate our system on a wide range of scenes, emphasizing geometric complexity, shading complexity and lighting complexity.

Our first set of scenes are from open source movies Big Buck Bunny [Blender Foundation 2008] and Elephants Dream [Blender Foundation 2006]. We picked these scenes to illustrate a couple of issues. One, our robust handling of high geometric and shading complexity. Two, in the original scenes, several fill lights are used to light each character to effectively fake the absent indirect illumination. We eliminated those lights (though they can be trivially added if desired), and instead show how indirect illumination can be automatically added to the scenes using our system.

Big Buck Bunny. Figure 7 shows a film shot from the open source movie Big Buck Bunny. The scene is lit by the Eucalyptus Grove environment map and a sun model. The characters have very high complexity because they have tesselated hairs that are shaded using a general shader: the Kajiya-Kay hair model. The other materials are defined through an übershader which implements several BRDF models and combines them differently for each character. We note that our code which is not memory optimized could not handle all

the hair from the original model, so we limited ourselves to 140k hairs for the whole model (we address this issue in the Chinchilla model).

Chinchilla. Chinchilla is a single character from Big Buck Bunny. Since it is a single character, we were able to support the full export of all the hairs in the model. We show this scene to demonstrate that we can handle this complexity.

Elephants Dream. The room in Figure 8, taken from the eponymous movie, has 928k polygons, and uses the übershader. The only light sources are the two fluorescent lamps; we treat each as an area light source.

Our second set of scenes illustrates our ability to handle complex, highly anisotropic BRDFs, and very complex shaders. These specific shaders are from [Wang et al. 2008] and include captured spatially varying anisotropic reflectance data with rich details.

Pillow. The silk brocade pillow in Figure 4 uses the measured BRDF data from [Wang et al. 2008]. The highly anisotropic shader reads compressed data stored as textures and reconstructs the measured reflectance data using PCA. Eighty-three textures are used.

Plate. The plate in Figure 9 has a shader that implements a highly directional, spatially varying BRDF from [Wang et al. 2008]. The plate stands on a glossy ground plane with a shader generated normal map. The scene is lit by the Kitchen environment map.

Our next scene shows our ability to handle several common primitives used in programmable shaders: the noise function and normal mapping.

Shiny Marble in Cornell Box. The sphere, lateral walls and floor of this scene (shown in Figure 5) are defined by procedural shaders. The right wall shader is highly glossy and modifies its shading normal. The sphere modulates its material from an almost mirror-like surface to diffuse veins using a noise function.

Our last scene is included to compare our results against manually generated bounding functions for shaders.

Tableau. This scene in Figure 6 is taken from [Walter et al. 2006] and shows depth-of-field effects with a range of analytical BRDFs including Phong and Ward. We use this scene to compare against the manually generated bounding functions used in lightcuts.

6.2 Results for Multidimensional Lightcuts

We evaluated the following scenes using multidimensional lightcuts: tableau, big buck bunny, chinchilla, elephants dream, and pillow. Table 1 summarizes the data for these scenes.

To evaluate the performance of our interval shaders we use two main metrics: the average cut size and the total rendering time. Cut size corresponds to the number of representative gather—light pair evaluations needed to compute each pixel. Smaller cut sizes are more efficient and indicate that the interval shaders are being effective at computing bounds that are not overly conservative. We did not place an upper bound on the cut size, so the algorithm runs until it meets the algorithm's stopping criterion.

Tableau. First, we consider the Tableau scene (Figure 6) to evaluate how our automatic interval-derived bounds compare against the hand-crafted bounding functions from lightcuts. This scene includes depth-of-field and is challenging because the gather points are not spatially coherent, causing the s_3 shader to produce looser bounds. Even so, both cut sizes and rendering times are only about a factor 2.7× larger than for the original scene. We expected a performance penalty relative to manually crafted bounds and deem this

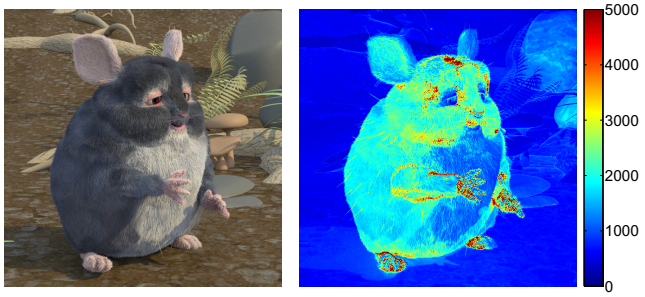


Figure 3: *A chinchilla with over 11 million triangles. The hair uses the Kajiya-Kay shader and needs only a small fraction (less than 0.07%) of all the possible gather-light pairs.*

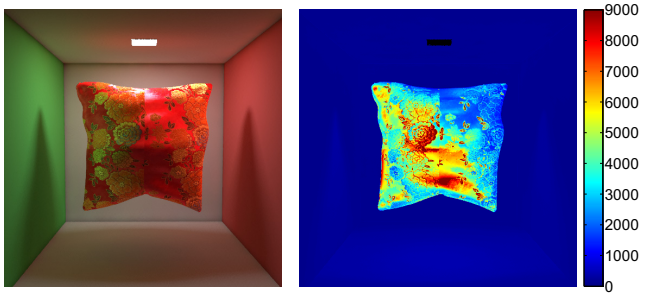


Figure 4: *A pillow with spatially varying anisotropic reflectance rendered with multidimensional lightcuts.*

to be reasonable given the automatic functionality provided by the interval shaders.

Übershaders in complex scenes. The scenes in Figures 3, 7 and 8 use übershaders. For all these scenes our interval shaders let us add indirect illumination and thus eliminate the need for fill lights to fake global illumination. Average cutsizes stay reasonable, ranging on average from 700 to 1,200 gather–light pair interactions computed per pixel (0.02% to 0.07% of all possible pair interactions).

Big Buck Bunny and Chinchilla are particularly complex scenes (10+ million polygons) and demonstrate how lightcuts amortizes the cost of rendering highly detailed geometry which is otherwise notoriously prone to aliasing artifacts under complicated lighting.

Measured BRDF. Using programmable shaders and our techniques automatically handles novel material models for which no simple parameterized formula or sampling strategy is known. Figure 4 shows a pillow with a measured BRDF stored in a complex compressed format and includes the indirect illumination it casts onto the surrounding box. Average cutsize is less than 1,000.

6.3 Importance Sampling

Our interval-based shaders enable importance sampling of the programmable shaders. However, our interval shaders incur an additional cost to build the importance function using several shader evaluations as described in Section 5. We now demonstrate how effective importance sampling is (despite its cost) in a couple of scenes: the shiny marble, and the plate scenes.

Shiny Marble. We render this scene using photon mapping and generate the photons in two ways: using cosine-weighted uniform hemispherical sampling, and using importance sampling derived from our interval-based shaders. Using roughly the same computation time we can generate a higher quality result with less variance

	Triangles	Eye rays	Direct	Indirect	Cut Size	VPL	Render	Total	Speed-up
Tableau (reference)	630 843	256	5 000	50 000	850.1	0.5 s	174.5 s	175.8 s	1 475 x
Tableau (shaders)	630 843	256	5 000	50 000	2 376.6	20.3 s	438.9 s	459.2 s	601 x
Big Buck Bunny	10 218 262	32	5 000	50 000	1 087.7	167.7 s	480.6 s	647.6 s	311 x
Chinchilla	11 020 943	32	5 000	50 000	1 223.1	168.8 s	347.5 s	516.3 s	282 x
Elephants Dream	928 557	32	37 753	50 000	759.6	85.7 s	378.3 s	464.0 s	834 x
Pillow	41 000	4	200	50 000	967.7	75.8 s	2 336.6 s	2 412.5 s	8 x

Table 1: *Results for the multidimensional lightcuts scenes. The statistics show: number of triangles, number of rays per pixel, direct lights, number of indirect VPLs. The results are: average cut size, time to generate VPLs, time to render, total time to render image. The last column shows the speed-up compared to a brute force rendering which uses all gather-light pairs without the bounding overhead.*

(Figures 5a and 5b). The benefits of the improved sampling lead to faster convergence using fewer samples, compensating for the cost of building the importance function. Even with more rendering time cosine-weighted sampling alone cannot match the importance sampling quality (Figure 5c).

Plate. This scene is also rendered with photon mapping using both cosine-weighted uniform sampling and our shader-derived importance sampling. Similar to the shiny marble scene, equal-time and equal-quality comparisons are shown in Figure 9.

Note that in scenes with diffuse-only shaders the cost of generating the importance function might not be worthwhile, since cosine-weighted uniform sampling (our default) is the best one could do. Knowledge from a modeler indicating if shaders are mostly diffuse could be useful. In our system we assumed that shaders are arbitrary, and so we conservatively incur the cost of importance sampling for all shaders.

7 Conclusions

This paper takes a first step towards automatically combining the disparate worlds of programmable shading and physically-based global illumination algorithms. Our compiler automatically generates interval versions of a programmable shader that compute bounds on shading values over sets of directions or sets of spatial and directional parameters. These bounding functions are then used to automatically generate low variance importance sampling functions even for highly directional procedural shaders. These additional type of queries on procedural shaders, sampling and bounds, then enable the use of efficient, robust global illumination rendering algorithms, such as multidimensional lightcuts and photon mapping, for models with procedural shaders.

We have demonstrated our system over a wide range of scenes including complexity in geometry, shaders, and illumination. We show that our automatically generated sampling and bounding, though not as fast as manually tuned versions, can be more efficient than existing techniques such as cosine-weighted sampling for arbitrary shading code.

Limitations and Future Work. Our sampling and interval bounding could be improved in many ways. Creating good importance functions currently is expensive and requires many subdivisions largely due to looseness in the interval bounds, and our simplistic refinement heuristic which does not always accurately reflect cost/benefit of refining importance regions.

The shading language is currently somewhat limited and does not support pruning of conditionals, arbitrary looping, or partial evaluation. Use of higher level type information such as unit vectors could be used to produce tighter bounds. Optimizing performance of the interval shaders and opportunities to create tighter bounds by specializing commonly used graphics operations must also be investigated. Automatic use of SIMD operations could significantly speed up some computations. We would also like to add support for other types of shaders such as light shaders.

Acknowledgments

This work was supported by NSF CAREER 0644175, NSF CPA 0811680, NSF CNS 0403340, and grants from Intel Corporation and Microsoft Corporation. The first author was also supported by CONACYT-Mexico 228785. The third author was partly funded by the NVIDIA Fellowship. The Eucalyptus Grove and Kitchen environment maps are courtesy of Paul Debevec (`http://www.debevec.org/probes/`).

References

Apodaca, A. A., and Gritz, L. 1999. *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann.

Arikan, O., 2008. Pixie – Open Source RenderMan. `http://www.renderpixie.com/`.

Bala, K., Dorsey, J., and Teller, S. 1999. Radiance Interpolants for Accelerated Bounded-error Ray Tracing. *ACM Transactions on Graphics 18*, 3, 213–256.

Blender Foundation, 2006. Project Orange – Elephants Dream. `http://www.elephantsdream.org`.

Blender Foundation, 2008. Project Peach – Big Buck Bunny. `http://www.bigbuckbunny.org`.

Comba, J. L. D., and Stolfi, J. 1993. Affine Arithmetic and Its Applications to Computer Graphics. In *Anais do VI Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens (SIBGRAPI'93)*, 9–18.

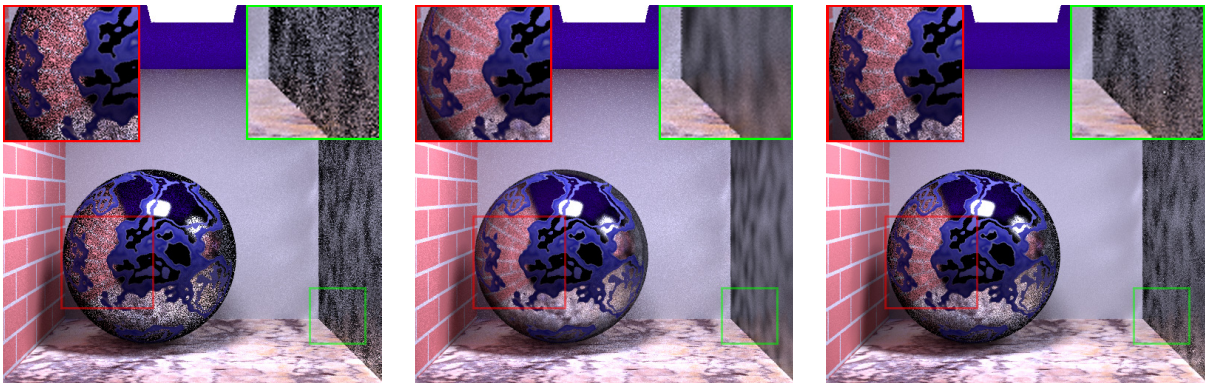
Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13*, 4 (Oct), 451–490.

Flórez, J., Sbert, M., Sainz, M. A., and Vehí, J. 2006. *Improving the Interval Ray Tracing of Implicit Surfaces*. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 655–664.

Hasselgren, J., and Akenine-Möller, T. 2007. PCU: The Programmable Culling Unit. *ACM Transactions on Graphics 26*, 3 (August), 92:1–92:10.

Hasselgren, J., Munkberg, J., and Akenine-Möller, T. 2009. Automatic Pre-tessellation Culling. *ACM Transactions on Graphics 28*, 2 (April), 19:1–19:10.

Heidrich, W., and Seidel, H.-P. 1998. Ray-tracing Procedural Displacement Shaders. In *Proceedings of Graphics Interface 1998*, Canadian Human-Computer Communications Society, W. A. Davis, K. S. Booth, and A. Fournier, Eds., 8–16.



(a) *Cosine-weighted sampling (2 031.4 s)* **(b)** *Importance sampling (1 829.5 s)* **(c)** *Cosine-weighted sampling (6 134.4 s)*

Figure 5: *Images rendered with photon mapping. (a) and (b) are equal time comparisons, and demonstrate the higher quality achieved using our automatic importance sampling. (c) is rendered using cosine-weighted sampling and significantly more time, without achieving the quality of (b).*

Heidrich, W., Slusallek, P., and Seidel, H.-P. 1998. Sampling Procedural Shaders Using Affine Arithmetic. *ACM Transactions on Graphics 17*, 3, 158–176.

Jensen, H. W., and Christensen, P., 2007. High Quality Rendering Using Ray Tracing and Photon Mapping. ACM SIGGRAPH 2007 Course #8 Notes, August.

Jensen, H. W. 1996. Global Illumination Using Photon Maps. In *Proceedings of the eurographics workshop on Rendering techniques '96*, Springer-Verlag, London, United Kingdom, 21–30.

Keller, A. 1997. Instant Radiosity. In *Proceedings of SIGGRAPH 1997*, ACM Press / Addison-Wesley Publishing Co., T. Whitted, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 49–56.

Latta, L., and Kolb, A. 2002. Homomorphic Factorization of BRDF-based Lighting Computation. In *Proceedings of SIGGRAPH 2002*, ACM Press, T. Appolloni, Ed., Computer Graphics Proceedings, Annual Conference Series, ACM, 509–516.

Lawrence, J., Rusinkiewicz, S., and Ramamoorthi, R. 2004. Efficient BRDF Importance Sampling Using a Factored Representation. *ACM Transactions on Graphics 23*, 3 (August), 496–505.

Moore, R. E., and Bierbaum, F. 1979. *Methods and Applications of Interval Analysis (SIAM Studies in Applied and Numerical Mathematics) (SIAM Studies in Applied Mathematics, 2.)*. Soc for Industrial & Applied Math.

Moule, K., and McCool, M. D. 2002. Efficient Bounded Adaptive Tessellation of Displacement Maps. In *Proceedings of Graphics Interface 2002*, 171–180.

Olano, M., Kuehne, B., and Simmons, M. 2003. Automatic Shader Level of Detail. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, Eurographics Association, Aire-la-Ville, Switzerland, Switzerland, 7–14.

Parker, S. G., Boulos, S., Bigler, J., and Robison, A. 2007. RTSL: A Ray Tracing Shading Language. *Symposium on Interactive Ray Tracing 0*, 149–160.

Pellacini, F., Vidimče, K., Lefohn, A., Mohr, A., Leone, M., and Warren, J. 2005. Lpics: A Hybrid Hardware-accelerated Relighting Engine for Computer Cinematography. *ACM Transactions on Graphics 24*, 3 (July), 464–470.

Pellacini, F. 2005. User-configurable Automatic Shader Simplification. *ACM Transactions on Graphics 24*, 3 (July), 445–452.

Ragan-Kelley, J., Kilpatrick, C., Smith, B. W., Epps, D., Green, P., Hery, C., and Durand, F. 2007. The Lightspeed Automatic Interactive Lighting Preview System. *ACM Transactions on Graphics 26*, 3 (August), 25:1–25:12.

Snyder, J. M. 1992. Interval Analysis for Computer Graphics. In *Computer Graphics (Proceedings of SIGGRAPH 92)*, vol. 26, ACM, 121–130.

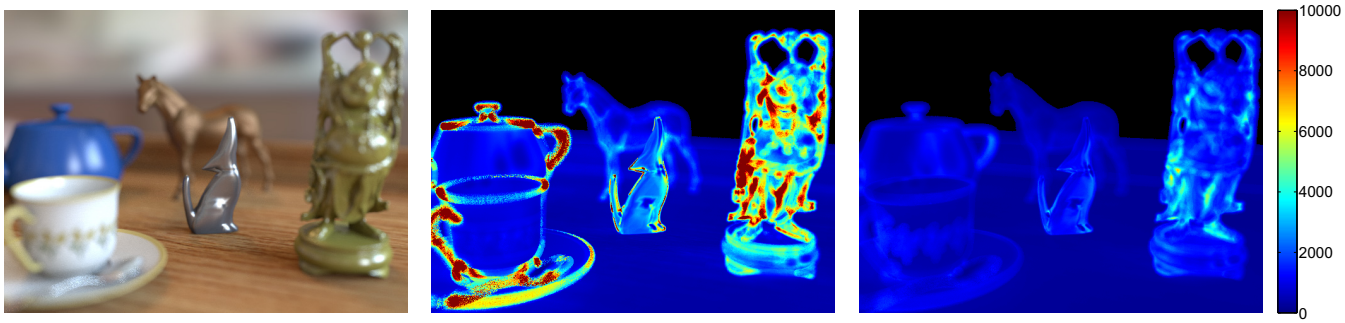
Tabellion, E., and Lamorlette, A. 2004. An Approximate Global Illumination System for Computer Generated Films. *ACM Transactions on Graphics 23*, 3 (August), 469–476.

Tupper, J. A. 1996. *Graphics Equations with Generalized Interval Arithmetic*. Master's thesis, University of Toronto.

Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., and Greenberg, D. P. 2005. Lightcuts: A Scalable Approach to Illumination. *ACM Transactions on Graphics 24*, 3 (July), 1098–1107.

Walter, B., Arbree, A., Bala, K., and Greenberg, D. P. 2006. Multidimensional Lightcuts. *ACM Transactions on Graphics 25*, 3 (July), 1081–1088.

Wang, J., Zhao, S., Tong, X., Snyder, J., and Guo, B. 2008. Modeling Anisotropic Surface Reflectance with Example-based Microfacet Synthesis. *ACM Transactions on Graphics 27*, 3 (August), 41:1–41:9.



(a) *Rendered image* **(b)** *Interval shaders* **(c)** *Analytic materials*

Figure 6: *Tableau scene with high depth-of-field rendered with multidimensional lightcuts. The cut sizes are comparable in most of the scene, but are higher on the glossy regions (as expected). Performance remains competitive.*

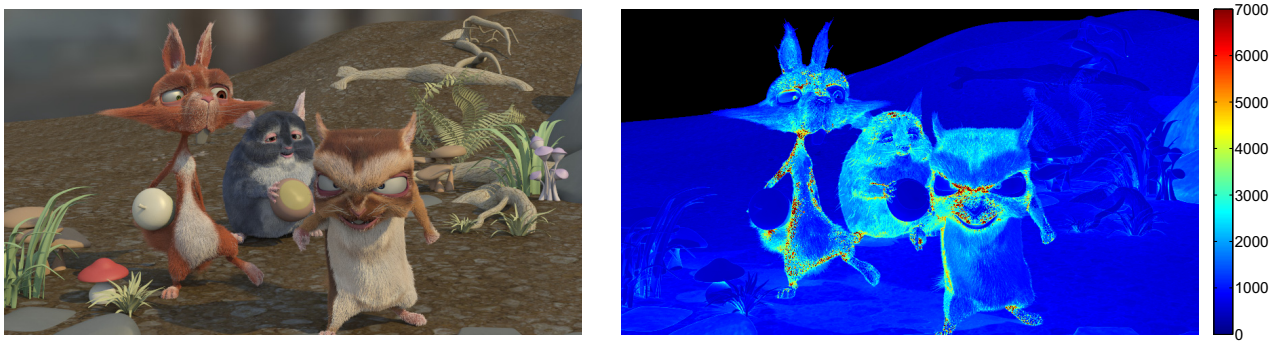


Figure 7: *A frame from the movie Big Buck Bunny rendered with multidimensional lightcuts (left), and associated cut size image (right).*

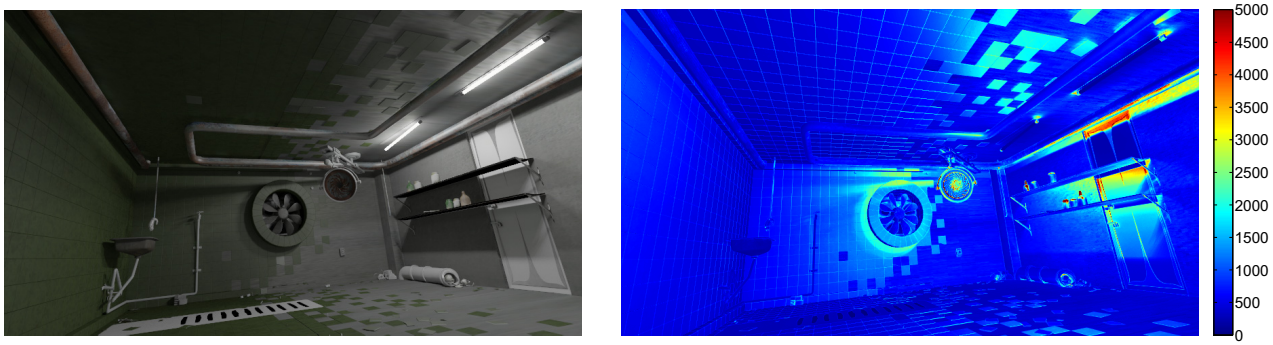
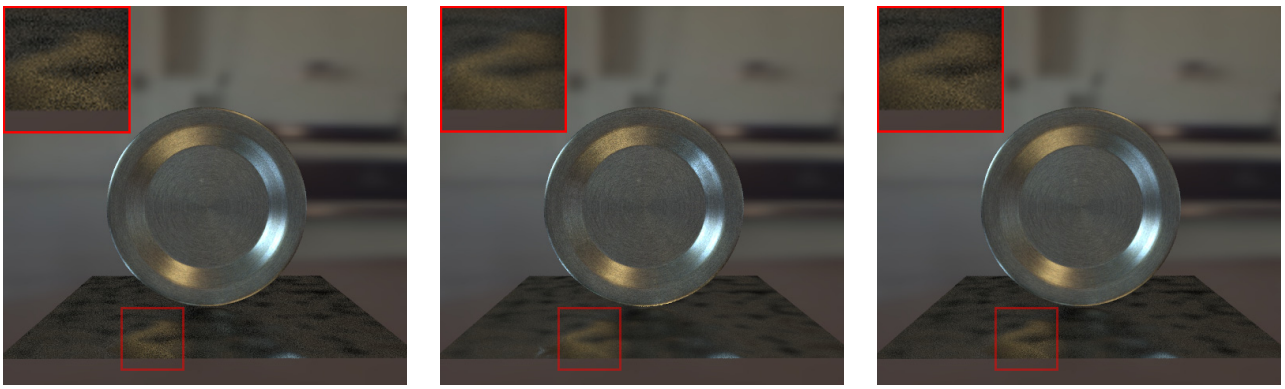


Figure 8: *A frame from the movie Elephants Dream rendered with multidimensional lightcuts (left), and associated cut size image (right).*



(a) *Cosine-weighted sampling (11 020.6 s)* **(b)** *Importance sampling (10 836.7 s)* **(c)** *Cosine-weighted sampling (19 670.6 s)*

Figure 9: *A plate with spatially varying anisotropic BRDF standing on a glossy surface with a shader-generated normal map, rendered using photon mapping with different sampling configurations. (a) and (b) show equal time comparisons, while (c) requires more time but still does not achieve the quality of (b).*