Approximate Models For Physically Based Rendering

by Angelo Pesce and Michał Iwanicki

1 Introduction

Physically based rendering has become increasingly popular in recent years. Its ability to provide a coherent and predictable look to materials under different lighting conditions has proven to be invaluable in the fast-paced production environments of today's movies and video games.

This flexibility, however, comes at a price: complete physically based lighting and shading models are typically more complex than previous ad hoc ones, and therefore require more computational power. As such, game developers are often restricted to using them in only the simplest of cases, where the calculations simplify to forms that can be evaluated with minimal performance impact. This, in turn, can lead to inconsistencies or limit artistic expression.

Ideally we'd like to find affordable yet faithful approximations to physically based models, at least for specific scenarios. However, without the right tools or intuition, this can be a time-consuming and fruitless endeavor.

With that in mind, the goal of this document is to provide a set of guidelines on how to approach this process more effectively, drawing from our recent experience. We will also go step by step through a number of case studies, puttings these ideas into practice.

1.1 Motivation

Physically based rendering has the potential to simplify art production: as budgets to produce art assets increase relying on models that constrain production to fewer parameters that are always guaranteed to generate result coherent to a given style (photorealistic in this case) can be quite time saving.

Using these methodologies we are able to shift the focus of production teams from the technical details necessary to achieve realism to higher-level aesthetic considerations, similar to how for example motion capture can shift focus from keyframing to directing, or photography lowered the barrier of entry for picture creation, compared to painting.

If though our models are flawed, approximated, or make the wrong assumptions, the constraints that we place can be detrimental, as we effectively can end up defining a space of possible art creation that doesn't contain the results we seek.

Worse, these issues can be very hard to detect when just observing the final results. Looking at the history of real-time graphics we have often made many even fundamental errors: from not correctly understanding color spaces to ignoring the effect of area light sources, from mistakes in geometric normals to disregarding aliasing effects in shading, and certainly we are still, consciously and not, making many more today [**TODO**].

All these mistakes tend to be "corrected" by artists by compensating in the art assets, often creating other errors that can persist for a long time and subtly manifest as wasted time, unjustified hacks and end results that don't fully achieve the realism we seek.

Effectively artists are "optimizers", we give them a set of parameters with which operate, and if the model we created is not expressive enough they will tune these parameters to be as close as possible to the production goal. This can result in certain effects being artificially muted, or certain properties diverging from what would be physically plausible.

In this context, approximate models can help as they allow us to:

- Not make assumptions: create models from data derived from first principles
- Cheaply approximate effects that we know are needed, but for which no exact runtime model exists
- Find approximated models directly from acquired real-world data

2 Theoretical Background

The problems we're going to solve commonly are expressed as global constrained optimization.

Let's consider a function $f(x, y)$ whose output we can compute numerically and whose inputs are split into two vectors: $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$. Let's consider a model function that we are able to evaluate in runtime $g(x, p)$, we want to find the choice of parameters $p \in \mathbb{R}^k$ such that the model approximates f closely.

$$\forall y \min_{p_y} \int_x |f(x, y) - g(x, p_y)|^n$$

For a n-norm or for infinity (sup norm):

$$\forall y \min_{p_y} \max_x |f(x, y) - g(x, p_y)|^n$$

Note that the solution of this problem is a set of parameter choices p_y, which is equivalent to finding a function $p : \mathbb{R}^m \to \mathbb{R}^k$, which we'll know only numerically at given sample points.

We'll use this formulation in two different, but closely related, scenarios: data projection and analytic modelling.

In the first case we're looking at a high dimensional problem domain that we want to transform into a lower dimensional manifold to make it more tractable. This is achieved either by chosing a model g with $k < m + n$, that's to say its parameter space which we fit numerically has a lower dimensionality than the original function, or by fitting to a model that has equal or higher dimensionality, but that results in parameters that are strongly correlated and can be easily modelled analytically or disregarded entirely.

In the second case we deal with functions of lower dimensionality, often the result of the first projection step, and we want to find a closed-form model, that's to say a $g(x, p)$ where the parameter choice p is constant and can be folded inside the expression of g.

3 Useful numerical algorithms

In this section we'll briefly illustrate some numerical algorithms that we will found useful in our applications. It is not our intent to provide a comprehensive overview of numerical computation topics, as this would be quite daunting, but we hope that this introduction can serve as a starting point for further study.

In general we will avoid too sophisticated algorithms that require expertise to be used and can be tricky to implement. We always prefer erring to the side of a brute force approach when possible, as all the case studies we will present are computed offline just once, we don't care much about efficiency, and prefer to "burn" cycles going wide with mulithreading or GPU compute when needed.

We seek techniques that are:

- Relatively simple and brute force.
- Very established and general, not limited to given classes of functions (no smoothness or continuity constraints).
- Not relying on anything but sampling (gradient-free: no derivatives, jacobians or hessians).
- Not critically dependend on parameter tuning.
- Preferrably robust to noise.

3.1 Integration

Integrals will appear often in these problems, not only as a way to estimate an error measure as detailed above, but also since the rendering equation [**TODO**] is itself an integral. Given that, in general, the rendering equation cannot be solved analytically, frequently we will want to approximate this integral.

In terms of error functions there are two main cases to consider, depending on the nature of the function we will approximate:

- f is given numerically, at fixed sampling locations. This is the case we face with real-world data or with data derived from large offline simulations.
- f is given as an expression cheap enough to be freely evaluated.

In case our source data is the result of an offline simulation, thus we have the choice of sampling scheme to use. In most cases a simple regular (Cartesian grid) sampling is the best choice. Even though it is unlikely to be the most optimal sampling scheme, it is simple to evaluate and has the important properties of being trivial to use to interpolate, project onto planes and slice through (intersect with planes). This will be useful for multidimensional problems when we will need to visualize the data and the approximation errors, or when we will need to make decisions about whether certain variables can be dropped out of the problem.

If we are dealing with problems of very high dimensionality, then more advanced techniques can be used, trading efficiency for ease of implementation. Low-discrepancy sequences (Quasi Monte Carlo) are a standard method to increase convergence of Monte Carlo (sampling based) integration [**TODO**].

Observe that for moderately low dimensions we could use sparse grids [**TODO**], that are sparse but regular sampling schemes that retain the interpolation simplicity of regular grids while not sampling densely all points in the grid.

Importance sampling can also be employed but it has to be used judiciously. Note that in general an importance sampling scheme that considers the values of f alone is not going to distribute samples according to the difference $|f - g|$ we seek to estimate. We can though adaptively generate samples based on f to capture high-frequency areas of the target function, or we can importance sample components that are shared by both f and g. A typical example is with integrals on the sphere or hemisphere, that become easier to evaluate if we importance sample the polar domain [**TODO**]

3.1.1 Numerical Integration

While in general we advocate simple sampling schemes, for integrals in one or two dimensions there are efficient rules (quadrature and cubature formulas) that converge dramatically faster than Monte Carlo or Quasi-Monte Carlo methods [**TODO**].

The simplest way to build an integration rule is to use interpolation: given a number of samples, we use them to construct an interpolating function in a form that we know how to analytically integrate (compute the area under the curve). These formulas, for equispaced abscissas (regular sampling grids), are called Newton-Cotes formulas, and are defined by the order of the interpolating polynomial used. Order zero (considering each sample as a rectangle area) is called the "mid-point" rule, whilst order one (linear interpolation) yields the "trapezoid" rule.

If we do allow non-equispaced abscissas then even better rules can be devised, where "better" usually means they are guaranteed to integrate certain class of functions (usually polynomials) exactly, with a lower number of samples. Gaussian quadrature is a classic example, that can exactly integrate a polynomial of degree $2n-1$ with n sampling points, chosen as the roots of a given class of orthogonal polynomials. Gauss-Kronrod is a notable variant, an example of a nested quadrature rule, which nests the formula precisely once. This adds $n+1$ points to a n point Gaussian rule and yields two integral estimates, which can be useful for adaptive integration.

Other notable integration rules are derived by expanding the integrand in a given space that simplifies integration. For example, the Clenshaw-Curtis method employs Chebyshev polynomials, whilst the related Fejer formula employs a DCT expansion.

In practice we often don't have any guarantee on the smoothness of the integrand (continuity of derivatives)in our problems, thus we can't rely on the theoretical guarantees offered by higher-order integration rules.

Adaptively dividing integration domain

In general, all these quadrature rules are schemes that can be expressed, for a given number of samples, as a simple table of sample point locations and associated weights, in one dimension. For higher dimensional problems the relative quadrature rule can be derived from this one-dimensional table by simply taking the Cartesian product, which makes for a very simple, very efficient implementation.

All these rules can be made adaptive. We can always take any given rule and evaluate it twice, with a different number of points, to yield a coarse and a more refined integral estimate. If the difference is higher than a given threshold, and we have not reached a preset maximum splitting depth, we can then split the integration domain into two or more tiles and evaluate recursively.

Efficient adaptive schemes will use nestable formulas, so that the second estimate "recycles" the samples used in the coarser level. A simple adaptive scheme will always take as the next domain tile to evaluate the first available one, thus can be implemented trivially as a recursive procedure (local adaptation). A much more efficient strategy is instead to keep all the tiles that are still above the target threshold in a heap, and each time remove the tile with the largest error estimate (global adaptation). Note that if no tile hits the maximum recursion limit, both schemes generate identical results.

Finally, note how all these schemes so far describe work on rectangular domains. Any other integration domain needs first to be transformed via a change of variable to a rectangular one, and such changes are well studied for spheres, hemispheres, circles, triangles, and many other domains, including infinite (unbounded) ones. However, there are also integration rules especially crafted for many domains, and just like importance sampling, using these rules is usually more efficient than a change of variable method [**TODO**].

3.2 Optimization

3.2.1 Differential Evolution

Differential evolution [**TODO**] is an optimization method belonging to a family of solutions called “metaheuristic” methods. These techniques are often “nature inspired” and straddle a balance between random search and local exploration to find a global minimum or maximum of large classes of functions, without requiring any constraint on the nature of such functions. They are usually hard to characterize exactly in terms of convergence, but experimentally prove to perform well in real-world optimization problems with nonlinear, multimodal and multivariate functions.

Differential evolution is a population-based method. It seeks an optimum point by evaluating the target function with a number of sampling points, and at each step it generates new coordinates for this population of points based on the current configuration and function estimates. Other notable population-based methods are particle swarm techniques, simulated annealing and genetic algorithms [**TODO**]. Differential Evolution is closely related to Genetic Algorithms as it generates a new population at each step by mixing the coordinates of the individuals at the previous step, but it’s more efficient over the continuous domains we usually encounter than genetic algorithms, whilst also being simpler to implement.

Also, importantly for us and in contrast with many other metaheuristics, it’s a method that does not depend on many complicated control parameters, and for which rule-of-thumb settings do exist [**.**] Note that as for all metaheuristic algorithms the performance on a given function varies dramatically based upon the initial parameters. Yet finding these parameters is itself an optimization problem that we can’t solve a-priori. Some advanced schemes even try to search for the best performing parameters at the same time as optimizing the original problem. /CiteTODO.

Initialization

Initialization for differential evolution is trivial: we just need a population of a given number of individuals: in this case, sampling points in the domain. Usually the population size is determined as $\min(40, 10 * d)$ where d is the dimensionality of the problem domain.

The individuals can be distributed randomly or using any sampling strategy; in low dimensions we can simply jitter a regular grid of samples (simple stratified sampling), but in many dimensions we quickly run out of sampling points to employ such a strategy and we need to resort to random or quasi-random distributions.

We will denote a population as a set of vectors (chromosomes) $\mathbf{x}_i$ of n individuals with $i = (1, 2..., n)$ at iteration (generation) t in d dimensions as:

$$\mathbf{x}_i^t = (x_{1,i}^t, x_{2,i}^t, ..., x_{d,i}^t)$$

Iteration

In common with genetic algorithms, any iteration in the differential evolution metaheuristic is made by genetic crossover (selection). Differential evolution is made of many different strategies (more than ten have been formulated), and mixes of strategies are also possible [**TODO**]. Here we will describe a basic, well proven variant, identified as DE/rand/1/bin.

At each iteration we will replace all the current individuals with new ones. For each individual i we will select three others, all distinct, randomly: $\mathbf{x}_a$, $\mathbf{x}_b$ and $\mathbf{x}_c$. The new individual vector will then be:

$$\mathbf{x}_i^{t+1} = x_a^t + f(x_b^t - x_c^t)$$

The mutation weight f can be either fixed (usually around 0.5) or randomly perturbed (per generator) from a minimum value to a maximum of 1 (a technique called dithering).

For each dimension of each individual we will then use either the mutated component or the original vector by random chance, with a given probability (crossover probability, usually set to 0.5). We always mutate at least one component of each individual, typically by forcing the crossover to always happen for a particular dimension, chosen at random per individual.

From this variant, we can very simply derive a related strategy identified DE/best/1/bin, which forces $\mathbf{x}_a$ to be the best individual (smallest function value for a minimization problem) of the generation, instead of being picked at random.

Constraints on variables (optimization domain) are usually handled by either re-initializing at each generation individuals that violate them (in which case it is acceptable to tolerate a given amount of violation) or by clamping their coordinates to so they respect all the constraints.

Stopping Criteria

The simplest stopping criteria for any optimization algorithm is to end the procedure after a given number of iterations. This method, while wasteful, can be effective as it doesn't incur any issues related to mistakes made by the criteria itself, e.g. stopping too early in cases where the solution is converging slowly. On the other hand, without testing with many different number of iterations, it is impossible to know if convergence happened.

More advanced stopping criterion for differential evolution are based on measuring one or more metrics that are assumed to converge when reaching the optimum: the difference between values of the best individual (current optimum), the movement made by each individual of the population and the distance between all individuals [**TODO**].

Noise and Cooperation

Differential evolution can also deal with noisy functions, usually by sampling the objective function many times and taking the average, then increasing the number of samples as we progress through the generations. The idea is that in the early stages the algorithm is exploring the optimization space coarsely, trying to identify areas of local optima. In these stages we don't need absolute precision in the evaluation of the objective function, while as we progress we begin to isolate and converge to the global optimum, thus more accuracy is needed [**TODO**].

In our problems we can exploit this reasoning in a different way. As our objective function is itself computed numerically we can devise a cooperative optimization-integration scheme where the integration step is tuned to different accuracy as the optimization progresses. This is particularly true in case of randomized algorithms, whilst with deterministic ones we have to be sure that reducing the integration accuracy doesn't skip a given optimum area that would then never be explored by the optimization algorithm. This strategy is applicable even if we deal with error functions based on fixed samples (real world data or a fixed precomputed solution) by randomly selecting points out of an increasingly large subset as the optimization progresses.

3.2.2 Nelder-Mead

The Nelder-Mead, or simplex method [**Nelder65**], is a nonlinear optimization technique that doesn't need explicit derivatives. It can be extremely useful when dealing with complex, non-linear optimization problems, mainly due to its extreme simplicity. Technically, it is a local optimization method as it cannot escape local optima, but in practice it is often effective as a global optimization strategy for functions that don't exhibit strong multimodal characteristics, and can be restarted to effectively sample multiple local optimum points.

The method starts by constructing a $N + 1$ point simplex from vertices x_0, x_1, ... x_N, where N is number of dimensions of the domain. The values of function f are evaluated at the vertices, and the vertices are sorted so that $f(x_0) \leq f(x_1) \leq f(x_2) \leq ... \leq f(x_N)$. Next, a centroid, x_o, of all the points *except* x_N is computed. Depending on the relations between the function values in certain of the points, the simplex is transformed:

- Reflection: the reflected point $x_r = x_o + \alpha(x_o - x_{N+1})$ and the value of the function at the reflected point $f(x_r)$ are computed. If $f(x_0) \leq f(x_r) < f(x_{N-1})$ we replace the worst point (x_N) with x_r and repeat the process.

- Expansion: if the reflected point is better than the best point ($f(x_r) < f(x_0)$) an expanded point $x_e = x_o + \gamma(x_o - x_N)$ is computed. If the value of the function in the expanded point $f(x_e)$ is better than the value at the reflected point ($f(x_e) < f(x_r)$) we replace the worst point (x_N) with the expanded point and restart the process. If the value at the expanded point is worse than the value in the reflected point ($f(x_e) \geq f(x_r)$) we replace the worst point with the reflected one and restart the process. If the reflected point is not better then the best point, we perform contraction.

- Contraction: a contracted point is computed $x_c = x_o + \sigma(x_o - x_N)$. If the value at the contracted point is better then the worst point ($f(x_c) < f(x_N)$) we replace the worst point (x_N) with the contracted point and restart the process.

- Reduction: if none of the above applies, we replace all points *except* the best one (x_0) with $x_i = x_0 + \delta(x_i - x_0)$ and restart the process.

The process is then repeated until the simplex is smaller than a specified threshold. The standard values for α, γ, σ and δ coefficients are 1.0, 2.0, -0.5 and 0.5 respectively.

The method creates the function gradients implicitly, but at each step only a couple of function evaluations are needed. Additionally, the method is extremely easy to implement, and very straightforward to debug. Further details are available on Wikipedia [**NelderMead**].

Starting Point

The Nelder-Mead method is an iterative, local algorithm and the quality of the solution is heavily influenced by the choice of starting point. The probability of finding a global minimum is much higher when the starting point is somewhere near the expected solution. In many cases we can predict where in the domain the solution will be; for example, when modeling a specular BRDF lobe, the peak direction will most likely be close to the reflected view vector. These rules should be taken into account when generating the starting simplex. The simplest way is to calculate one point in the domain, using a simple heuristic to place it close to the expected solution, and generate the initial simplex by performing random perturbations of that point. Care must be taken to generate points that are not coplanar; if the initial simplex is planar, Nelder-Mead will not be able to find any solution outside of this plane.

Multiple Starting Points

Since the quality of the solution is highly dependent on the starting point, a common practice with local optimization algorithms is to use multiple starting points. Instead of generating just one initial simplex, we can generate many of them and perform independent fitting for each one (since they are truly independent, this can be done in parallel). The result of the optimization process is the instance with the lowest error.

Restarts

Another common strategy is to perform a *restart* as soon as a local minimum is found. When the size of the simplex goes below the threshold, we can compute its centroid, generate new a simplex by perturbing the centroid, and perform another set of iterations.

3.3 Putting it all together

Some reccomendations:

- For minimization, combine a global strategy with a local post-process.
  - Global metaheuristics are usually designed to explore efficiently the problem domain, not to zero in accurately on the minima.
  - E.G. Differential Evolution can be post-processed with Nelder-Mead.
- For expensive error measures consider:
  - Running the error estimation at a lower accuracy for early iterations of the minimization algorithm.
  - Running the minimization hierarchially: first on a subset of the parameter space we want to find minima for, then restarting on a finer sampling of the space, using as starting points of the minimization the solution found for the nearest point in the coarser level.
- Use visualizion to assess the quality of the solution.
  - Plot the surface and function distance estimate resulting from running minimization across different parameter choices 1.
  - Plot the error surface that the minimization algorithm is trying to navigate, for a few selected parameters 2.
  - Repeat the plots at different iterations of the algorithms, to verify convergence.

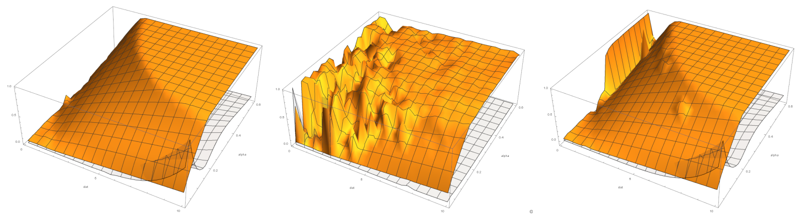


Figure 1: Plot of the surface resulting from running minimization multiple times varying problem parameters in a given domain. The semitransparent surface is the distance between the source data and the model. Left to right: correct surface (note that areas of high distance exist, but these are due the inability of the model to fit the data, not issues with the numerical algorithms), surface resulting from a optimizer that failed to converge to the minimum, surface resulting from an optimizer that on certain points failed to identify the correct global optimum and converged to a different local minimum.

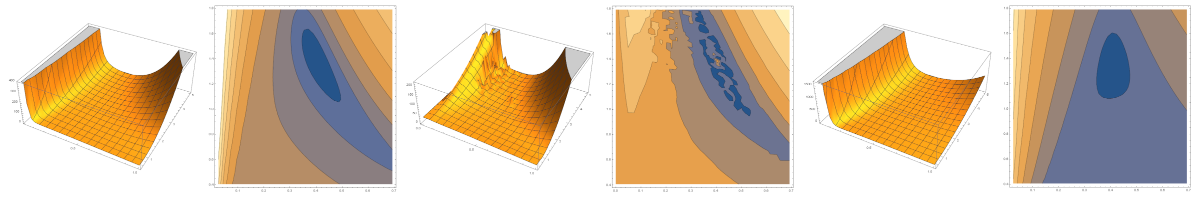


Figure 2: Plot of an error surface (computed via numerical integration) for a minimization problem. 3d surface and logarithmic contour plot (which makes the minimum more evident). Left to right: correct integral, noise from lack of convergence of a randomized integrator, change in surface shape from the lack of convergence of a deterministic integrator. Randomized methods can require more samples to behave well as inputs of a optimizer, but deterministic algorithms can have systematic errors that are much harder to spot than noise.

4 Case Study: Spherical Area Lights

Area lighting (non-puncutal light sources) has been an increasingly popular problem to solve in the realm of real-time rendering [**TODO**]. Its importance stems from the advent of Physically Based methods as one of the main advantages of these approches is that they seek to make, just as in the real world, the final appearance of a scene dependent from a number of decoupled features: materials, lighting, geometry and so on.

This wasn't the case with previous more or less arbitrary, empirical models, where the final results where pretty much painted or dialed in, changing parameters to achieve a specific look without regard for what these parameters meant. This resulted in assets that looked good in certain cases, but were not modular, the same assets under different lighting conditions or even viewing conditions were not guaranteed to look right.

Area lights are a very important component of this decoupling of scene parameters, infinitesimal lights are an ideal model impossible in the real world, and as a consequence, in order to match observed specular and diffuse shading under such illumination, artists often resorted to artificially increasing materials roughness or decreasing the overall prevalence of specular effects.

4.1 Exploring ground truth

The simplest form of area lighting we can integrate into a rendering system are uniform spherical lights. Let's consider the rendering equation for direct lighting at a given surface point [**TODO**]:

$$L_\mathrm{o}(\omega_\mathrm{o}) = \int_\Omega f_r(\omega_\mathrm{i}, \omega_\mathrm{o})\, L_\mathrm{e}(\omega_\mathrm{i})\, (\omega_\mathrm{i} \cdot \mathbf{n})\, \mathrm{d}\,\omega_\mathrm{i}$$

Where $L_\mathrm{o}(\mathbf{x}, \omega_\mathrm{o})$ is the outgoing radiance towards the observer f_r is the BRDF and $L_\mathrm{e}(\mathbf{x}, \omega_\mathrm{i})$ is the emitted radiance from light sources. Then our problem is defined by solving the above integral with L_e constant in a given cone of directions around a light direction vector.

The first step towards an approximate model is then to generate a ground truth solution and explore it, trying to deduce how we could analytically model its shape across the parameter space. For this problem, the parameter space is made by the degrees of freedom fo f_r (roughness and Fresnel assuming a specular GGX model), the light vector direction, the observer vector direction and the light cone radius. We can assume without loss of generality multiplicative constants as the light total intensity or the surface specular albedo to be unitary, so these won't affect the dimensionality of the problem.

The ground truth was generated for simplicity by uniformly sampling all the problem dimensions at fixed intervals 3, computing the area light integral via the Quasi Monte Carlo method, importance sampling only the hemispherical domain. The integrator code was written for GPU execution in OpenCL to allow for interactive exploration.

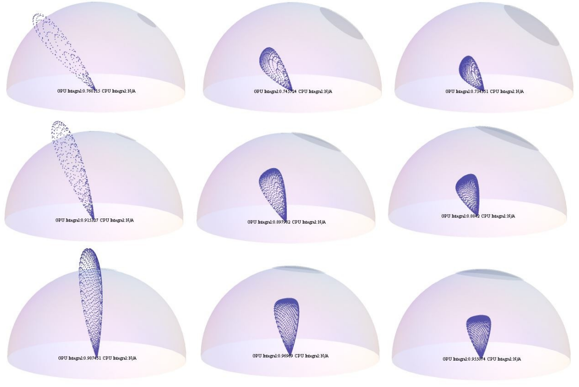


Figure 3: Area light ground truth via numerical integration (QMC), various area light sizes and indicence angles

4.2 Model fitting and dimensionality reduction

Trying to directly approximate the data we obtained with numerical simulation can be daunting, as it is at least a six-dimensional problem. In the following instead we'll try to look for models that can use to project the data into a lower-dimensional space first, and then we'll try approximating that result.

4.2.1 Roughness Modification

Observing the ground truth it is possible to notice how the integrated result still closely resembles a lobe from the original BRDF, with an infinitesimal light. This makes sense intuitively, also considering that as the projected area light size (cone radius) approaches zero, the integral should converge to the results from an infinitesimal light.

As the BRDF itself is parametrized on material roughness, we could try to derive for each incident angle, roughness and area size a new roughness parameter that approximates the intregral as a infinitesimal light on a more matte surface 4. This is also justified by the observation that artists, in absence of an area light model, tend to over estimate the material roughness to compensate, effectively tuning "by eye" what here we try to optimize numerically.

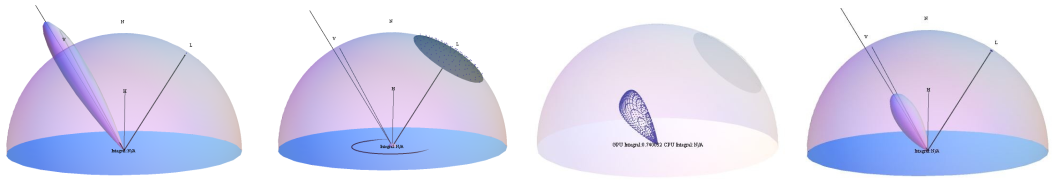


Figure 4: Left to right: a Blinn lobe with an exponent of 500, an area light with a cone of 40 steradians, the results of numerical integration of the previous two, a Blinn lobe with an exponent of 75.

The new roughness parameter is estimated via numerical optimization for every sampled choice of roughness, area light size and incident angle, producing a three-dimensional table $t : (roughness, size, angle) \rightarrow roughness'$ we can then visualize and reason upon.

Immediately after fitting we'll have to verify that our model holds, that indeed we are able to capture the salient charateristics of the numerical solution in a lower-dimensional projection. As it was noted in [**TODO**] such a roughness modification approach, even when driven by numerical fitting is unable to represent a peculiar characteristic of the area light integral, which shows a significant area of plateau in its lobe for wide area lights, a plateau that the original BRDF can't express at any roughness 5.

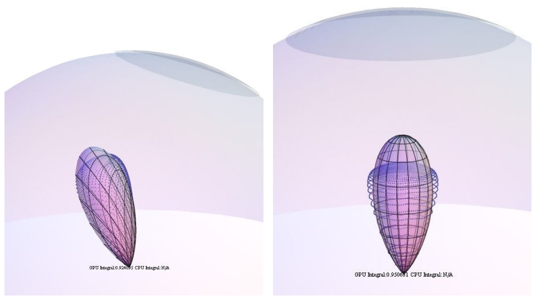


Figure 5: On the left: a case where we managed to find a close fit to the numerical integral. On the right: an example of a failure case, where the inability of a simple fitted BRDF lobe to match the area light integral is evident

A good optimization solution will still produce the best fit possible, but upon inspection is clear that we need a more expressive model.

4.2.2 Adding degrees of freedom: Capped Roughness Modification

A simple way of making a model more expressive is by adding more degrees of freedom to it. This is a technique that should be handled with care, each degree of freedom we add will be more data we'll later have to model analytically, and also increases the complexity of the model to be evaluated at runtime.

On the other hand often identifying an expressive model can yield better projections, whose parameters strongly correlate only with certain dimensions, thus being able to disregard parts of the space and work on a lower dimensionality manifold that simplifies the overall approximation.

In our case, after noting the difficulties of the simple roughness modification model, we tried to augment it with an extra parameter that allowed the representation of a "cap", a maximum value of the BRDF lobe. This cap was modeled as a "soft minimum" function:

$$\text{softmin}(x, y) = (x^s + y^s)^{1/s} \tag{1}$$

With $s < 0$ controlling the smoothness of the resulting function 6.

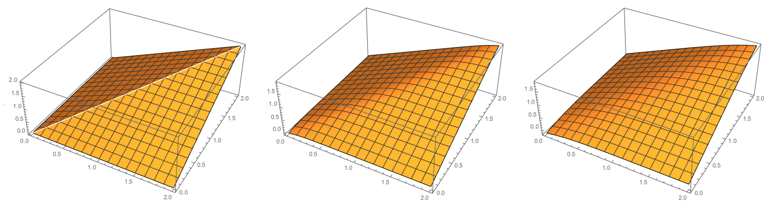


Figure 6: From left to right, graphs of: minimum function, soft minimum with $s = -8$, soft minimum with $s = -4$

After numerical optimization we're left with now three functions, mapping roughness, area light size and incident angle to a new roughness, a maximum cap value and the smoothness of the soft minimum formula employed. This data goes then through another evaluation loop:

- Did the model fit the numerical solution well?
- Which dimensions strongly correlate with the parameters of the model?
- Can we find a simple variable substitution or projection than makes the data simpler?

- Do we need a more expressive or more succint model?

Exploring the data, with both interactive comparisons with the ground truth and by plotting it projected against coordinate axes it's evident how this model is a much better fit for the numerical simulation 7.

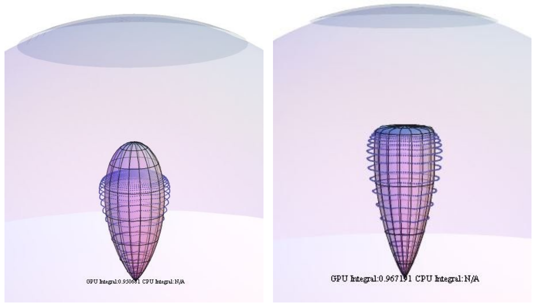

GPU Integral:0.950681 CPU Integral:N/A
GPU Integral:0.967191 CPU Integral:N/A

Figure 7: On the left: the failure case from the previous roughness modification model. On the right: the new, capped model solves the issue.

What might be surprising though is that it ended up not even being more expensive, as it showed a weak dependence to area light incident angle, and a reasonably small variation of the smooth minimum parameter 8.

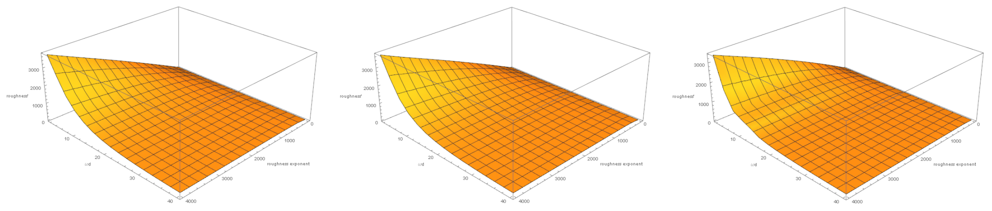


Figure 8: Plots of the fitted roughness parameter as a function of material roughness and area light size, shown at different incident light angles. From left to right: 0 radians, 0.5 radians and 1 radian.

Fixing $s = -10$ and computing an average of the fitted roughness over all incident angles we can derive a much lower dimensional formulation of the problem that still fits the data well. When exploring data is also important to carefully choose the plotting axes and scales, what we are left with is a two-dimensional table 9 $(roughness, area\_light\_size) \rightarrow roughness'$, that actually assumes a very simple shape once we express the roughness parameters as a cone angles (inside which the BRDF has most of its energy, with a threshold arbitrarily set at 90%), following the idea that we should, whenever possible, examine our data on axes with uniform units.

4.2.3 Smoothed Representative Point

While the model presented above is already expressive enough and simple enough to be used for real-time rendering purposes, we kept exploring alternative formulations as we noticed still some slight difficoulties in modeling the ground truth at grazing incidence angles.

A promising model was proposed as an empirical solution to the roughness modification issues by Brian Karis [**TODO**], and we wanted to verify how well it matched our numerical simulations. This model replaces the light vector in the inifinitesimal light solution of the BRDF integral with the direction inside the projected sphere light cone that is closest to the view-normal reflection vector,

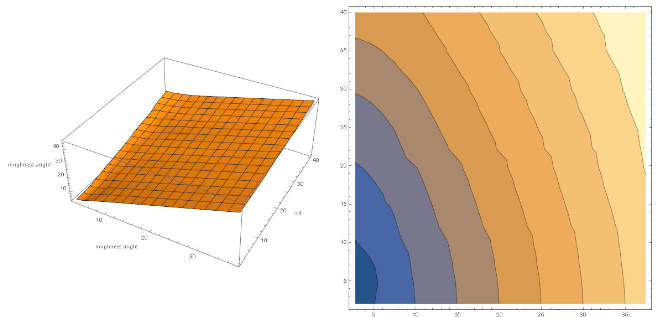


Figure 9: Plot of the fitted roughness parameter as a function of the material roughness and area light size, averaged across incident angles. The roughness has been remapped to be a measure proportional to a cone angle. Right: same data as a contour plot.

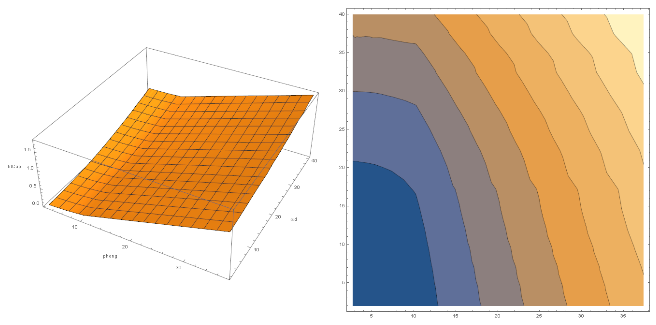


Figure 10: Plot of the fitted cap parameter as a function of the material roughness and area light size, averaged across incident angles. The roughness has been remapped to be a measure proportional to a cone angle. Right: same data as a contour plot.

replacing the area light integral with an evaluation of a single point on the light, the one that would yield the most intense specular reflection.

$$\text{centerToRay} = (\mathbf{L} \cdot \mathbf{r})\mathbf{r} - \mathbf{L} \tag{2a}$$

$$\mathbf{L}_{\text{representative}} = ||\mathbf{L} + centerToRay \cdot \max(0, \min(1, \frac{\text{sourceRadius}}{|centerToRay|}))|| \tag{2b}$$

Where $\mathbf{L}$ is the light vector and $mathbfr$ is the reflection vector. This solution also requires to compute a re-normalization factor to make the resulting modified BRDF respect:

$$\forall \omega_i \int_{\Omega} f_r(\omega_i, \omega_o) d\omega_o \leq 1$$

Which is needed for a physically plausible model [**TODO**] and was Karis main contribution to the technique. As for the roughness modification model, we first tried to fit it unmodified, allowing a varying scale (to account for the normalization, that we won't incorporate a-priori) and a varying roughness as well (that we will eventually disregard if it fits to an identity function, returning always the source material roughness), and then visualize the results plotted against the ground truth.

We immediately notice 11 how this model seems to suffer from the opposite of the issue exhibited by the roughness modification method: it does model a plateau that resembles the behaviour of the simulated data, but such cap is of a flat, uniform intensity, regardless of the shape of the underlying BRDF we convolve with our area light emission.

Again, as we did before, we then seek a more expressive model that has the potential to better fit the data, and we hope that once computed we can achieve a significant dimensionality reduction by

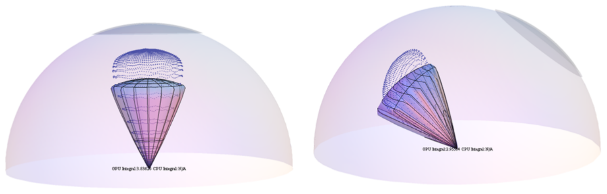


Figure 11: Plots of the fitted "representative point" model, showing its inability to respect the original BRDF shape well.

exploiting weak correlation between the problem dimensions and the fitted parameter values.

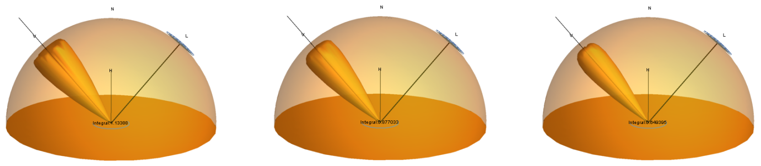


Figure 12: From left to right, tests of representative point solution with varying degrees of smoothness.

Different models for smoothing were tried 12, in the end we settled with a simple modification of the representative point math: replacing in 2b the min again with a soft minimum 1. As usual we first fit the data to this model using a fairly large number of parameters, allowing for changes of roughness, scale and smoothing coefficent s. This yielded a quite accurate match, but we were able to actually drop all the dependencies from the problem dimensions, ignoring roughness modifications and setting the soft minimum smoothness parameter to a constant of $s = -3$ (for our GGX BRDF) while still remaining accurate enough for our needs **??**.

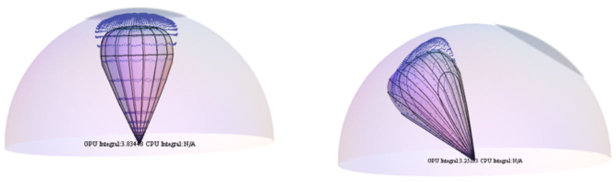


Figure 13: Plot of the corrected "representative point" model, with an added smoothing parameter.

The only data that remained to be approximated is the normalization (scaling factor), which we do recompute both as the model changed from Karis, and also because Karis original normalization formula was itself an approximation.

4.3 Symbolic approximation of low-dimensional data

After we reduced our problem to a manageable dimensionality, we can tackle the issue of finding a symbolic approximation for the remaining data. This is a very similar process to model projection, employing a combination of ingenuity to craft models and numerical fitting to set their constants.

- Visualize the data.
- Formulate an hypotesis (symbolic expression, with free parameters).
- Fit the parameters, numerically.
- Observe the error and the fitted values.
  - Small fitted values can be removed (simplifying the expression).

  - Values fitted near their contraints indicate issues with the model.
  - Large errors require revision of the model, adding degrees of freedom or completely changing its expression.

The initial hypotesis formulation is mostly based on experience and knowledge of the graph shape of polynomials, special functions, rational polynomials and other analytic expressions, rather than domain-specific knowledge.

- Understand the units and ranges of input and output domains.
  - What is the geometrical or physical meaning of the variables involved? Can we make different dimensions use the same units?
  - What is the range of inputs and outputs? Can we normalize it?
  - Should we consider a logarithmic, exponential or reciprocal transform?
  - What is the appropriate error measure? This is important to remember as certain data transforms do make the data easier to model, but are meaningless in terms of the error we want to minimize during the parameter fitting.
- Visualize projecting to two dimensional graphs ($\mathbb{R} \to \mathbb{R}$).
  - Does the data seem to have a similar shape when visualizing one dimension, varying the other?
  - Does the data seem to sweep two simple shapes at the begin and of of its domain, on a given axis?
  - Does the data seem to have a simple shape when intersected with given planes?

Based on these observations we can derive expressions for data in a moderate amount of dimensions by hand, often as sweeps (interpolation) of simpler, lower dimensional expressions, that we were able to model at given slices.

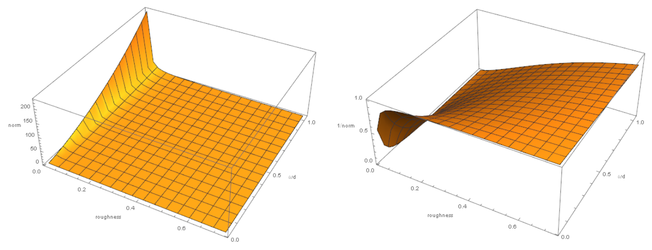
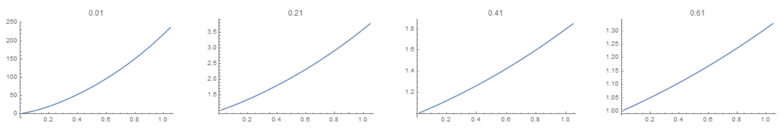


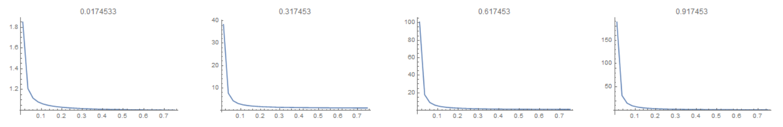
Figure 14: Same data, plotted with interpolation. On the right, the original data, on the left its reciprocal. We will model an expression for the reciprocal, but compute the error based on the original.

In the case of the smoothed representative point area solution we need to find a formula for its normalization, which is a table dependent on light size (expressed as a cone angle) and surface roughness 14. We noted that the normalization data seems to have a complex shape, but actually when viewed as its inverse, which is proportional to the value of the integral over the hemisphere of the non-normalized area integral, it assumes a quite simple behaviour.

In particular this reciprocal seems to have a constant shape when viewed at a constant cone angle, graphed on the roughness, while having a parabolic shape on the other dimension, which varies in what seems an exponential rate. Noting that in our shaders we don't deal with cone angles, but with



(a) Fixed roughness, varying area size.



(b) Fixed area size, varying roughness.

Figure 15: Normlization data.

light radiuses and distance, we fitted a model considering $\tan(\omega/2)$ where ω is the cone angle. The final formulation, where α represents the GGX roughness, was:

$$1/()a + b\frac{\tan(\omega/2)}{\alpha^c})$$

Which was numerically fitted to $a \to 0.962406$, $b \to 0.375892$, $c \to 0.962406$, resulting in the function of figure 16

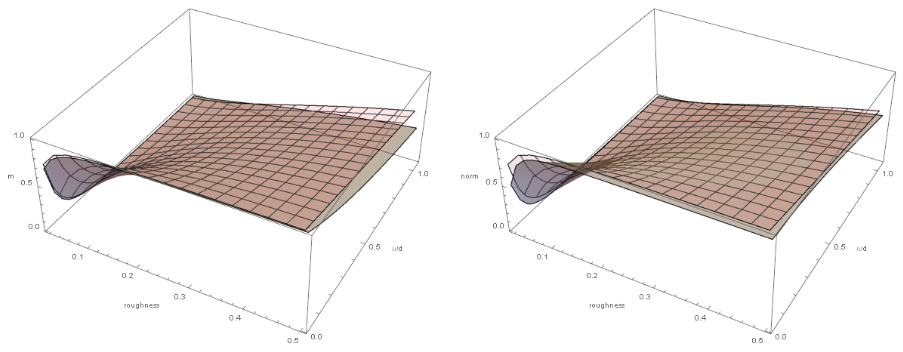
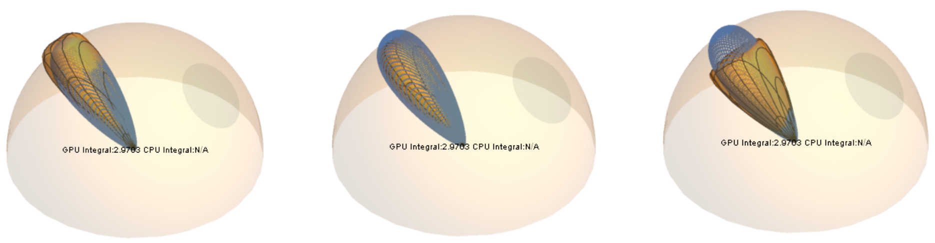


Figure 16: On the left: the computed normalization data and Karis original formulation (in gray); On the right: same data compared with our new analytic approximation.

4.4 Results

Comparisons: 17 18.


GPU Integral:2.9703 CPU Integral:N/A
GPU Integral:2.9703 CPU Integral:N/A
GPU Integral:2.9703 CPU Integral:N/A
Figure 17: Left: fitted smooth representative point with tabled data; Center: tabled data replaced with an analytic function; Right: Original method by Karis.

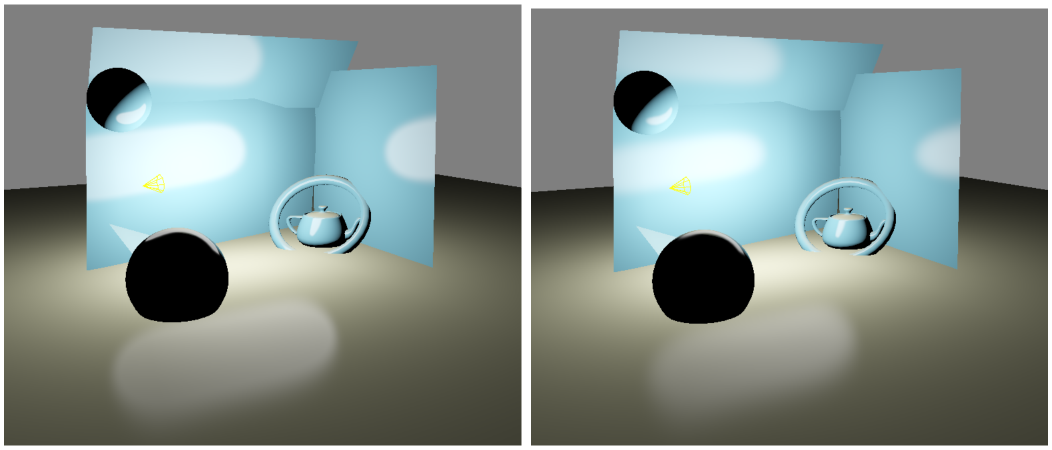


Figure 18: Capsule lights rendered with: on the left the original approximation by Karis, on the right the new smoothed representative point solution.

5 Symbolic Regression

So far we have used automated methods only for projection and fitting: finding the optimal values of certain free parameters to minimize a given error measure. The formulation of the expressions to fit has been though completely manual.

The basic idea behing symbolic regression is to automate the search of good symbolic models to approximate given data sets, or more accurately, models that minimize certain error measures.

A simple symbolic regression procedure could be based for example on random generation of formulas following given grammars which include a set of operators (arithmetic operators and special functions) and a set of inputs (variables in our problems and arbitrary parameters). For each of the generated expressions, we could then try to numerically fit its parameters, if any, and compute the minimum error we can achieve with that form compared to our source data.

This simple approach, albeit not actually as impractical as it might sound, is certainly not very efficient, we are not in any way directing our search of plausible expressions. Most practical symbolic regression methods are instead based on variants of genetic programming [**TODO-Koza**]: genetic algorithms working on individuals that represent encoded program strings.

Genetic algorithms are a class of optimization methods which can work on very generic problem domains, are not limited as the optimizations procedures described in section 3.2 to real, continous domains. They are population based, so a number of individuals is randomly created at initialization, and at each step (generation) of the algorithm such population undergoes two phases:

- Selection: where a subset of the population is considered to breed a new generation. Often the best individuals (least error) are preserved (elitism), but a good genetic variety needs to be maintained to avoid premature convergence (genetic drift, which is non-ergodic)
- Breeding: where a number of "genetic operators" are applied to one or more selected individuals to create new ones in the next generation. Common operators are:
  - Mutation, which randomizes portions of the individual encoding.
  - Crossover, which splits the encoding of two individuals into random substrings, and swaps them.

The actual details vary, and in literature there are many variants [**TODO-overview**], often adapted to specific problems. In general though genetic algorithms have been shown to perform well on problems of moderate complexity, even if, as with many others metaheuristics, their performance is not well understood.

Genetic algorithms operate under the assumption that the "building block hypotesis" holds, which postulates that recombining partial solutions can lead to improved solutions. In other words efficient search happens in a GA when there are certain short substrings in individuals that can be recombined to create higher fit ones, that's to say certain substrings hold a special value, they are expected to be present in any high-fit (low error) individual, they are "partial solutions" or "building blocks".

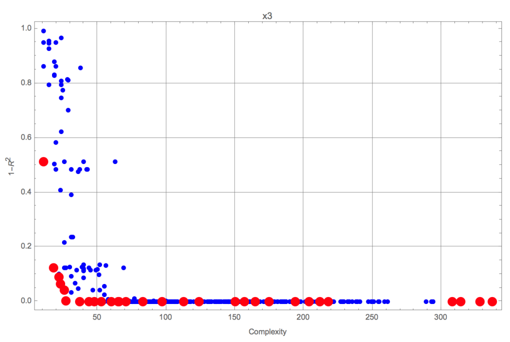
This hypotesis and explaination of the performance of GAs is not without challange [**TODO**] but in the case of symbolic regression it stands to intuition: we see often how certain transformation on input variables do simplify the shape of the data we have to approximate, thus it wouldn't be surprising to see given short expressions, which apply these transforms, in many high-fit individuals, with better solutions expressed as further transforms and combinations of these good subexpressions of input variables.

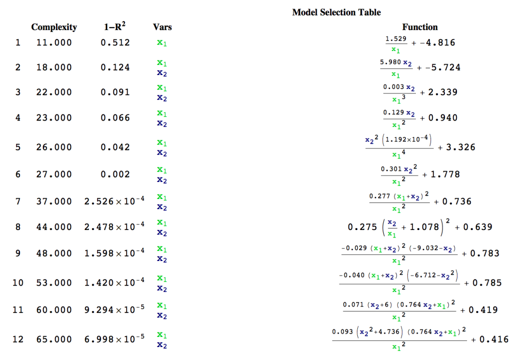
5.1 Pareto Front

Exploring the expression space looking only at fitness in terms of distance from the data that we want to approximate tends to advantage more complex expressions even when there are simpler forms that are acceptable for us. In general we don't know a-priori for a given data set what expression complexity it needs for its approximation, and often we don't even know what error we are willing to accept. A common technique to ameliorate this issue is to design a fitness metric that considers both the approximation error and the expression complexity (parsimony pressure), but the balance between the two would exactly require the kind of the insight on target complexity and error we often don't have.

In practice most of the times, as we are employing these methods for real-time rendering and as the generated expressions commonly end up being used by a shader evaluated for millions of screen-space samples each frame, we would like to explore the different trade-offs between expression complexity and approximation error.

This is what keeping track of a Pareto front of solutions allows us to do. If we have multiple objectives to optimize, a solution is said to be Pareto efficent (or Pareto optimal) if no other solution exists that improves on one objective without being worse at some other. Geometrically if we imagine the solutions to be points in the space of the objective measures, then the Pareto front for a minimization problem is made by the convex hull faces of the point set that are facing the origin 19.





Model Selection Table

	Complexity	1−R²	Vars	Function
1	11.000	0.512	x_1	$\frac{1.529}{x_1} + -4.816$
2	18.000	0.124	x_1 x_2	$\frac{5.980\,x_2}{x_1} + -5.724$
3	22.000	0.091	x_1 x_2	$\frac{0.003\,x_2}{x_1^3} + 2.339$
4	23.000	0.066	x_1 x_2	$\frac{0.129\,x_2}{x_1^2} + 0.940$
5	26.000	0.042	x_1 x_2	$\frac{x_2^2\,(1.192\times10^{-4})}{x_1^4} + 3.326$
6	27.000	0.002	x_1 x_2	$\frac{0.301\,x_2^2}{x_1^2} + 1.778$
7	37.000	2.526×10^{-4}	x_1 x_2	$\frac{0.277\,(x_1+x_2)^2}{x_1^2} + 0.736$
8	44.000	2.478×10^{-4}	x_1 x_2	$0.275\left(\frac{x_2}{x_1} + 1.078\right)^2 + 0.639$
9	48.000	1.598×10^{-4}	x_1 x_2	$\frac{-0.029\,(x_1+x_2)^2\,(-9.032-x_2)}{x_1^2} + 0.783$
10	53.000	1.420×10^{-4}	x_1 x_2	$\frac{-0.040\,(x_1+x_2)^2\,(-6.712-x_2^2)}{x_1^2} + 0.785$
11	60.000	9.294×10^{-5}	x_1 x_2	$\frac{0.071\,(x_2+6)\,(0.764\,x_2+x_1)^2}{x_1^2} + 0.419$
12	65.000	6.998×10^{-5}	x_1 x_2	$\frac{0.093\,(x_2^2+4.736)\,(0.764\,x_2+x_1)^2}{x_1^2} + 0.416$

Figure 19: Left: Complexity/Error plot of individuals of a population for a converged Genetic Programming run, solving the normalization problem of 4.2.3. In red the individuals on the Pareto front. Right: some of the Pareto front solution tabled with their expressions, complexity and error. Graphs generated via DataModeler.

The Pareto front can be used as a distance metric for individual fitness [**TODO**], and keeping track of Parato optimal solutions as the iterations progress allows us to chose a-posteriori the tradeoff between complexity and error that is appropriate for our run-time needs and can give us fundamental insights on the data we are approximating.

This is an important quality of symbolic regression, it's not only a tool for empirical modeling, but also for knowledge discovery as the solutions it finds encode insights in the nature of the underlying problem. After a symbolic regression run on a given data set we can observe for example variables appear often in the population, and which subexpressions appear more often. If certain variables are not used in many solutions they might correlate less with the data, be less important. If certain subexpressions are often present, they might indicate a fundamental transformation that the data implies, maybe a change of variable that is meaningful in the problem domain, that has perhaps a geometrical or physical interpretation.

Note also how this process of extracting sub-expressions and tabulating their frequency is again hinting at the principle of the building block hypotesis, exploited "in reverse".

Lastly note that for our uses the complexity measure, which we didn't define so far, can be obtained as an estimate (or simulation) of the run-time cost of evaluting the expression encoded by the individual on our target run-time platform, which yields the very desiderable property of solutions that for a given error are optimal in terms of execution speed.

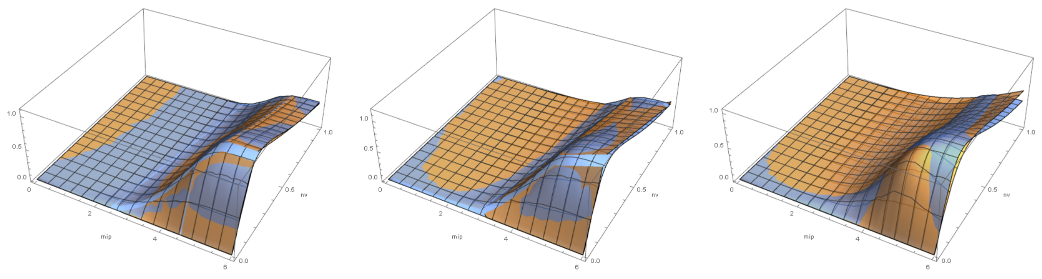


Figure 20: From left to right: an example of symbolic regression surfaces (in yellow) approximating a given data set (blue) obtained with an increasingly high penalty applied in the complexity measure to the use of piecewise functions.

Often though performance is not the only measure though that we have to consider, in many problems for example we might want to avoid discontinous (piecewise) approximations. Albeit this can be achieved simply by not allowing the system to use discontinous operators (e.g. minimum, maximum, absolute value...), often we want just to "strongly discourage" such operators, resorting to them only if otherwise the expression cost would be to high. This can be achieved by artificially penalizing them in the complexity measure, or by tracking them as another objective measure (thus yileding a three dimensional pareto front) 20.

5.2 Linear Genetic Programming

Linear Genetic Programming is a class of variants of Genetic Programming that are fairly interesting. As we already noted, many variants exist and many were created to solve specific problems and are often not extensively tested, and the same consideration apply in general to all optimization metaheuristics. We don't intend to embark in the impossible task of detailing all such variants, and our reccomendation is again to prefer well established, simple algorithms with efficient implementations, but we'd like to mention LGP and why we chose it for our custom symbolic regression code.

Linear Genetic Programming [**TODO**] encodes individuals not as trees or graphs, but as strings or arrays. In our case we chose an encoding where individuals are made of a fixed maximum number of opcodes, and each opcode admits a maximum fixed numbers of inputs and generates a single output.

This representation is similar to an assembly program, but instead of using registers we allow operators to fetch the result of any preceeding opcode in the string sequence. An opcode input can thus be of three types: a data input dimension, a floating or integer constant, or an index of a preceeding opcode. The expression result is considered to be the the output of the last opcode in the string. Note that in this representation certain opcodes can be "dormant", that's to say, not connected to the output and thus nor part of the actual encoded expression. These non-expressive parts of an individual are called "introns" and can be optionally pruned to keep the strings as small as possible.

There are a few advantages of this representation we exploit:

- Expressiveness: as the output of any opcode can be used as an input, the string represents a directed acyclic graph, allowing sub-expressions to be reused multiple times, creating an encoding that is more powerful than the basic tree-based genetic programming method.
- Efficiency: fixed strings are faster and easier to manipulate than variable length trees. The encoding is also close to how a given expression could be evaluated in run-time, allowing for better complexity estimates.
- Secondary encodings: we can easily keep a secondary representation of an individual if it's made of fixed-length components each encoding a given opcode. In this case the cross-over operator only needs to blindly copy regions of memory, without even knowing their meaning, just the length we need per opcode. Mutation does need to re-generate the secondary encoding but only for the opcode that changed. This allows to efficiently handle just-in-time assembly traslated versions of the individual for direct execution on the CPU, or source-code lines of a language we want to use to compile the individual associated expression.

Efficient evaluation of the individuals expressions on the coordinates of the data points is crucial for the performance of the regression. Most regressions were let run overnight to insure convergence, on a system capable of millions of expression evaluation per second.

The key to efficient evaluation is to exploit the data-parallel nature of the problem; On modern CPUs a bytecode interpreter has a considerable overhead as each function call or jump dependent on the bytecode type incours typically in a branch misprediction and large stalls [**TODO**] but as know we will need to evaluate the same expression over large numbers of data points, we can devise an interpreter that executes a given bytecode operation for multiple points at once, thus reducing the cost of interpretation to a vanishingly small amount [**TODO**]

This is not to say that a just-in-time compiler could not yield even more efficient code, an data-parallel interpreter still can't perform most optimizations that a compiler could, and it's particularly limited in the way it can use CPU registers, being forced instead of storing the results of each bytecode execution in memory. This also implies one can't do very wide parallel evaluations, we want to keep all the data we are working on in cache, so the data-set still has to be split in sub-groups. That said, a good data-parallel interpreter is much easier to write than a JIT compiler, and it can yield good performance.

Lastly, consider for very large data sets to adopt a pruning pre-pass, culling points in areas of less variation (frequency) 21. When this is done, care has to be taken to account for subsampled regions by increasing the weights of the points in them to be proportional to the sampling density. Note also that albeit we described a system that works on sampled data, the same system can be adapted to directly approximate an analytic function using numerical integration for error estimation. As the globally adaptive integrator described in 3.1.1 works on tiles, each using a fixed number of samples, the data-parallel interpreter can run per tile.

5.3 Case Study: Approximation of Split Integral for GGX Environment Lighting

As a sample problem for symbolic regression we consider approximating a data table arising from a common solution to the pre-integrated image based lighting equation. Our objective is to find a

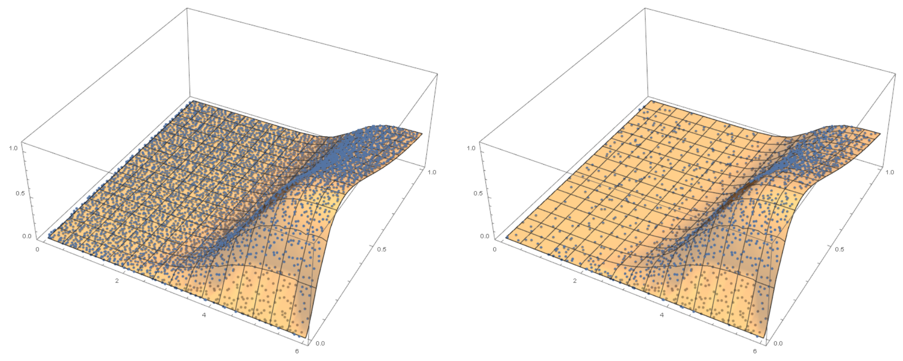


Figure 21: On the left: regular, jittered stratified sampling. On the right: sampling density proportional to the gradient.

way to cheaply compute surface radiance emitted from a given illumination environment around the hemisphere of the surface. Assuming that the environment is infinitely far we can then note how this solution is not dependent on the surface position, but only on the considered hemisphere (surface normal) and material characteristics. The corresponding rendering equation would be:

$$\int_{\Omega^H} f_r(\omega_\mathrm{i},\, \omega_\mathrm{o})\, L_\mathrm{e}(\omega_\mathrm{i})\, \cos\theta_i \, \mathrm{d}\,\omega_\mathrm{i} \tag{3}$$

Where Ω^H is the hemisphere we consider, L_e is the emitted lighting and f_r is our BRDF. This is a multi-dimensional problem that can't be tabulated easily, as it depends on the hemisphere normal, the viewing direction and the surface parameters.

Karis in [**TODO**] suggested a cheap but effective approximation based on "split sums". Considering assuming in our BRDF that view view vector is equal to the hemisphere normal we can solve 3 via importance-sampled Monte Carlo integration:

$$\frac{1}{N}\sum_{j=1}^{N}\frac{L_\mathrm{e}(\omega_j)\cdot f_r(\omega_\mathrm{j},\mathbf{n})\cdot\cos\theta_k}{p(\omega_j,\mathbf{n})} \tag{4}$$

Where $p(\omega_j, \mathbf{n})$ is the probability of the sample k. This equation was in turn approximated by splitting it in two sums, effectively representing respectively the integral of the environment lighting with a GGX lobe (made radially symmetric) multiplied by the integral of the BRDF assuming an uniformly white (unitary intensity) lighting environment:

$$\left(\frac{1}{N}\sum_{j=1}^{N}L_\mathrm{e}(\omega_j)\right)\left(\frac{1}{N}\sum_{j=1}^{N}\frac{f_r(\omega_\mathrm{j},\mathbf{n})\cdot\cos\theta_k}{p(\omega_j,\mathbf{n})}\right) \tag{5}$$

The first sum was then stored in the mip chain of a cubemap, each level representing a different roughness (GGX lobe), the second was approxiamted as a two-dimensional table parametrized on roughness and $\mathbf{n}\cdot\mathbf{v}$ with each element being a two-dimensional vector comprising of a scale and a bias which would then be applied to the Fresnel term of GGX.

The first attempt at analytically modeling the latter table was published by Lazarov [**TODO**], a subsequent improvement found by Krzysztof Narkowicz via hand-crafted curve fits [**TODO**]. His polynomial fit required ten multiplies and eight additions (five of which could be fused into a multiply-add):

```
float2 EnvDFGPolynomial( float gloss, float ndotv )
{
    float x = gloss;
    float y = ndotv;
    float b1 = -0.1688;
    float b2 = 1.895;
    float b3 = 0.9903;
    float b4 = -4.853;
    float b5 = 8.404;
    float b6 = -5.069;
    float bias = saturate( min( b1 * x + b2 * x * x, b3 + b4 * y + b5 * y * y + b6 * y * y * y
        ) );
    float d0 = 0.6045;
    float d1 = 1.699;
    float d2 = -0.5228;
    float d3 = -3.603;
    float d4 = 1.404;
    float d5 = 0.1939;
    float d6 = 2.661;
    float delta = saturate( d0 + d1 * x + d2 * y + d3 * x * x + d4 * x * y + d5 * y * y + d6 *
        x * x * x );
    float scale = delta - bias;
    bias *= saturate( 50.0 * specularColor.y );
    return float2(scale, bias);
}
```

This problem is particularly hard to approximate via genetic programming as in general multiple outputs greatly increase the search space, especially as we want to exploit correlations between them to devise cheaper solutions, deriving from the same building blocks.

In order to sidestep this issue we did the regression in two steps. First we seeked for symbolic formulas separately for scale and bias, each time allowing the other output as an input (deriving so a scale expression dependend on the bias value and viceversa). Then we used these regression runs to observe which sub-expressions recurred more often and which of the two formulas was more advantaged by the knowledge of the value of the other. With this analysis done, we could finally do a regression of one component in isolation and the second which retained the dependency from the first (or a transformation of it).

The resulting formula we selected was quite competitive with the hand-crafted solutions even in this problematic scenario, being more accurate and not more expensive 22:

$$\text{bias} = 2^{-(7(\mathbf{n}\cdot\mathbf{v})+4\text{roughness}^2)}$$

$$\text{scale} = 1 - \text{bias} - \text{roughness}^2 \cdot \max(\text{bias}, \min(\text{roughness}, 0.739 + 0.323(\mathbf{n}\cdot\mathbf{v})) - 0.434)$$

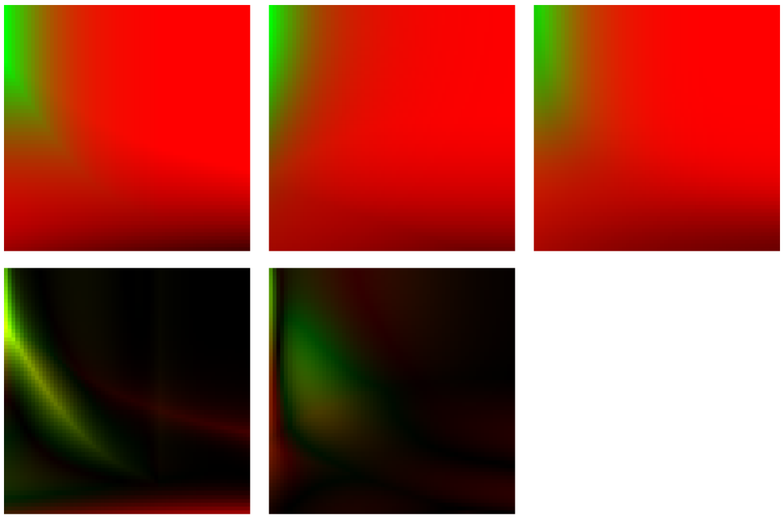


Figure 22: From left to right: Narkowicz formula, approximation via symbolic regression, original data. Bottom row shows absolute difference with the original data times 3.

6 Case Study: Environmental Lighting with the Disney Diffuse Model

In 2012, as a part of this course, Brent Burley presented a physically plausible diffuse shading model, which we shall refer to here as the Disney Diffuse BRDF. It exhibits some of the characteristics of real-life materials, such as dependence on roughness and retro-reflection near grazing angles. Many real-time applications do feel the need for more complex diffuse lighting, and while model like this can be used directly for punctual light sources, it's not obvious how to utilize it in the calculations of image-based lighting, which is commonly used to simulate environment lighting.

We will use Disney model as an example, but similar approximations can be performed for other low-frequency BRDFs.

The Disney Diffuse BRDF has the following form[1]:

$$f_d = \frac{baseColor}{\pi}\left(1 + (F_{D90} - 1)(1 - \mathbf{n}\cdot\mathbf{l})^5\right)\left(1 + (F_{D90} - 1)(1 - \mathbf{n}\cdot\mathbf{v})^5\right)$$

with

$$F_{D90} = 0.5 + 2\,roughness(\mathbf{l}\cdot\mathbf{h})^2.$$

Figure 23 (left) shows a polar plot of the Disney Diffuse BRDF for a fixed roughness and view angle. Looking at the plot like this might be slightly misleading. It suggests that lighting coming from the angles near the horizon contributes a lot to the final outgoing radiance. Also, it exhibits some high-frequency features (the 'foot' of the plot). The plot, however, doesn't include the cosine term—present in the lighting integral—that reduces the amount of incoming lighting based on the angle it's coming from.

If we multiply the BRDF by the cosine term, the resulting plot will look like the one in Figure 23 (right). The resulting function is much smoother, and while it still shows some skew toward the viewer, it no longer has a flat, high-frequency 'foot' near the horizon. Since we actually want to calculate the final value of the lighting integral, when trying to approximate the BRDF, it's much more useful to think of the BRDF multiplied by the cosine term, not just the BRDF itself. The spherical function—a product of the BRDF and the cosine—acts as a weight for the incoming lighting. We'll focus on

[1]Where $\mathbf{n}$ is the surface normal, $\mathbf{v}$ is the view vector, $\mathbf{l}$ is the light vector, and $\mathbf{h}$ is the half vector.

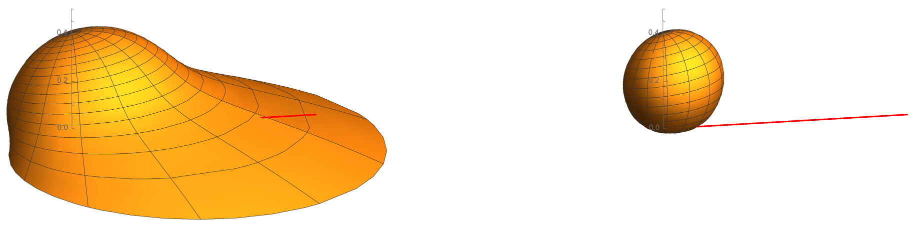


Figure 23: Polar plot of the Disney Diffuse BRDF (left) and the product of BRDF and the cosine term (right) (roughness = 1.0, viewing angle = 1.4; view vector marked in red).

approximating that product, not the BRDF alone. We will refer to that cosine-weighted BRDF as CBRDF. If we manage it, calculating the actual lighting will require calculating the integral of the product of the two components: the lighting and the CBRDF.

6.1 Spherical Harmonics Lighting

When dealing with diffuse lighting, Spherical Harmonics (SH) are often used to represent incoming radiance. Spherical Harmonics are an orthonormal basis defined over the unit sphere, analoguous to a Fourier basis, with the subsequent coefficients representing features of increasingly higher frequencies.

One of the most useful properties of SH is that it's very easy to calculate integrals of products of the functions projected to SH. Given functions A and B and SH basis functions Y_i

$$A(\omega) \approx \sum_i^N a_i Y_i(\omega)$$
$$B(\omega) \approx \sum_j^N b_j Y_j(\omega),$$

we have:

$$\begin{aligned}\int_\Omega A(\omega)B(\omega)d\omega &= \int_\Omega \sum_i^N a_i Y_i(\omega) \sum_j^N b_j Y_j(\omega) d\omega \\ &= \sum_i^N \sum_j^N a_i b_j \int_\Omega Y_i(\omega)Y_j(\omega)d\omega.\end{aligned}$$

Since SH form an orthonormal basis, $\int_\Omega Y_i(\omega)Y_j(\omega)d\omega = 1$ when $i = j$ and 0 otherwise, so the above reduces to

$$\int_\Omega A(\omega)B(\omega)d\omega = \sum_i^N a_i b_i, \tag{6}$$

which is just a dot product of the coefficients.

This property is often used in real-time applications to provide diffuse, environmental lighting. The incoming radiance is calculated and projected to SH, and given a normal direction, an SH projection of the cosine lobe oriented in that direction can be trivially constructed. Calculating the diffuse lighting is just a matter of computing the dot product between the two.

Given the low-frequency nature of diffuse lighting, we typically only need 3rd-order SH (nine coefficients) [**Ramamoorthi01**]. Since the plot of the Disney Diffuse CBRDF is fairly low-frequency, we can use similar reasoning. If we can find a simple way to generate an SH representation of the CBRDF for all the possible material roughness values and view angles, we can use the above rule of double product integrals of SH-projected functions to calculate the lighting.

6.2 Zonal Harmonics

Looking at the plots of the CBRDF, one can notice that the lobe is fairly radially symmetric for all the possible roughness values and viewing angles. When dealing with signals that are radially symmetric, we can use a simplified basis. When projecting signals that are radially symmetric around the z axis to SH, only one coefficient per band is non-zero. The set of functions with non-zero coefficients forms a reduced basis called Zonal Harmonics (ZH). Rotation of the ZH (which in general case produces an SH) is greatly simplified compared to general SH rotations (see [**Sloan08**] for details) which makes them a good choice for modeling low-frequency, radially symmetric signals.

We decided to model the CBRDF using Zonal Harmonics.

6.3 Approximation

The CBRDF was computed and projected into SH, for all combinations of viewing angles and roughness values. From the SH representation, an *optimal linear direction* was extracted (again, see [**Sloan08**] for details) and it was assumed to be the lobe direction. The SH projections were rotated to align the lobe axis with the z axis, and the ZH coefficients were extracted, as an approximation of the lobe shape.

Figure 24 shows the plot of ZH_0, ZH_1 and ZH_2 coefficients of the resulting lobes across the domain. All three coefficients change in a very similar manner across the domain, so we decided to perform a more complicated fit to one of them and approximate the remaining ones as multiples of the first one.

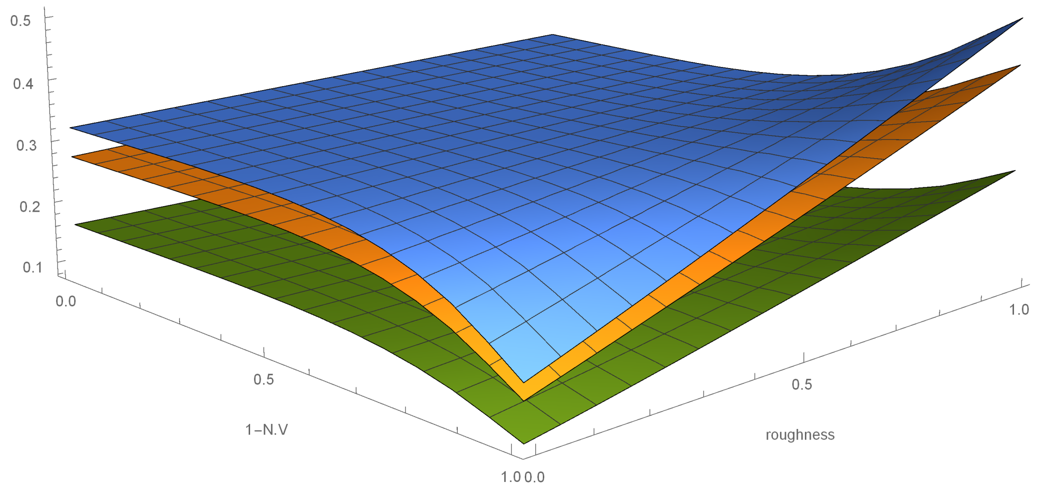


Figure 24: ZH coefficients of the diffuse CBRDF lobe across the *roughness* $\times$ $\mathbf{n} \cdot \mathbf{v}$ domain (ZH_0 in orange, ZH_1 in blue, ZH_2 in green).

We start by finding a good model for the base coefficient ZH_0. Looking at the cross section of the plot (Figure 25), one can see that it's flat up to around half of the domain and then bends. After several experiments, we found that the bent part can be approximated really well with a quadratic curve. To model the flat part, we simply clamp the argument to the quadratic function. The parameters of

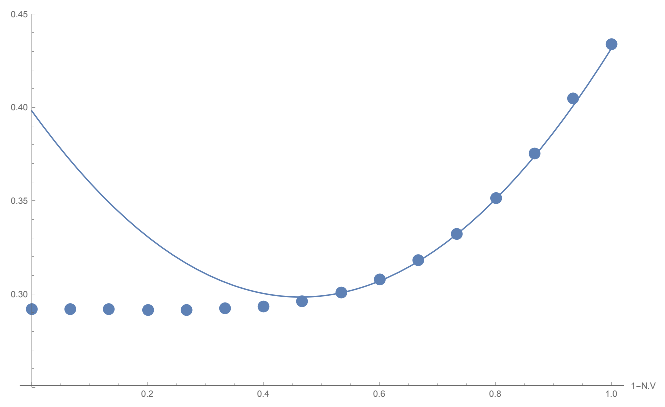


Figure 25: First ZH coefficient of the diffuse CBRDF lobe, for a fixed roughness (1.0), as a function of $1 - \mathbf{n} \cdot \mathbf{v}$ together with a quadratic approximation of the curved section.

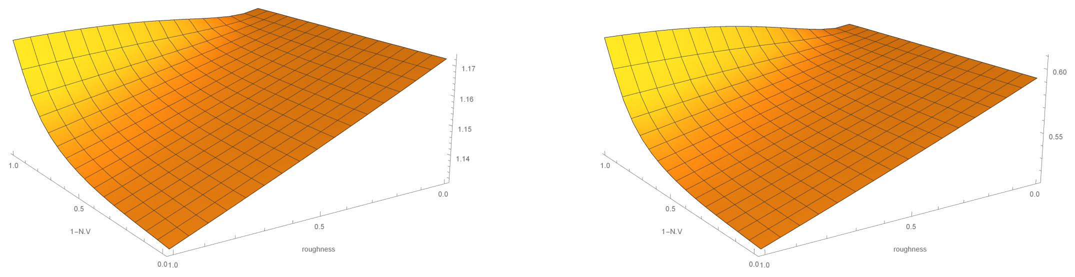


Figure 26: The ratio of first two ZH coefficients (ZH_1/ZH_0) of the diffuse CBRDF lobe(left) and third and first coefficients (ZH_2/ZH_0) (right).

the quadratic functions change in linear fashion with roughness, so we fit linear functions to these. The parameters for both the quadratic and linear functions were obtained using a simple, linear least-squares fit. As a result, we obtained a function that is linear with respect to surface roughness and quadratic with respect to $\mathbf{n} \cdot \mathbf{v}$, that provides a good fit for the first ZH coefficient (see Figure 27).

Looking at the ratios of the ZH_1 and ZH_2 coefficients to ZH_0 one can notice that they change slowly across the domain (note the range of the plots on Figure 26). While the change is not strictly linear and we could perform similar reasoning to model it as we did for the ZH_0 coefficient, we decided that doing a simple linear fit for the ratios should be enough. If runtime cost of the approximation is an issue, the ratios could be simplified even further, to constants.

To fit the lobe direction, $\mathbf{d}$, we express it as a linear combination of normal and view vectors:

$$\mathbf{d} = (1 - \beta)\mathbf{n} + \beta\mathbf{v}$$

For each point in the domain, we calculate a β coefficient, that generates $\mathbf{d}$ direction matching the extracted optimum linear direction from the SH representation of the CBRDF. Next, we perform a fit of the β coefficient. It's close or equal to zero on most of the domain and changes in relatively linear fashion in the remaining areas. We fit a plane to this change and clamp it to zero. Figure 28 shows the β coefficient together with its fit.

6.4 Results

All the parameters needed for rendering were approximated with simple functions that can be very efficiently computed on the modern GPUs.

Figure 29 shows the comparison between the simple Lambertian BRDF, the importance-sampled reference implementation of the Disney Diffuse BRDF, and the described approximation.

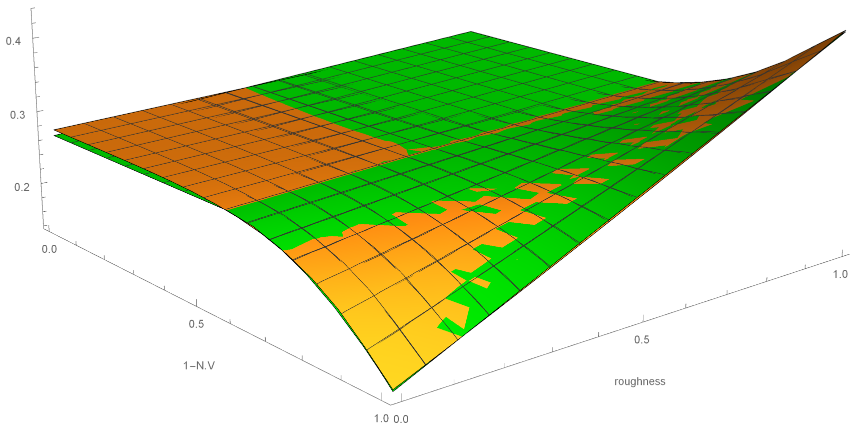


Figure 27: First ZH coefficient of the diffuse CBRDF lobe as a function of roughness and $\mathbf{n} \cdot \mathbf{v}$ (orange) and its analytical fit (green).

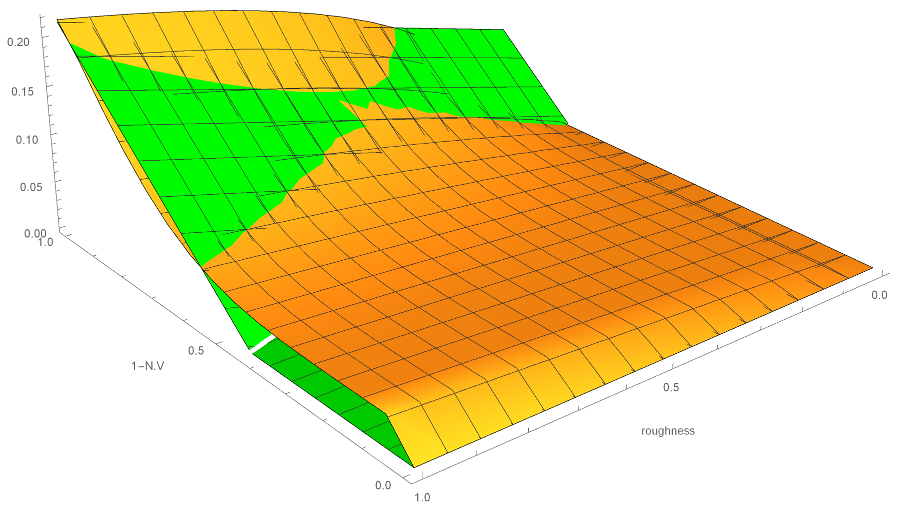


Figure 28: The β coefficient for calculating lobe axis (orange) as a function of roughness and $\mathbf{n} \cdot \mathbf{v}$ and its analytical fit (green).

Below are the actual analytical expressions that approximate the parameters of the diffuse CBRDF:

$$\begin{aligned}
\beta &= \max\left(0, -0.2988 + 0.3775\,(1 - \mathbf{n} \cdot \mathbf{v}) + 0.1372\, roughness\right) \\
ZH_0 &= (\quad 0.0913 + 0.3067\, roughness) + \\
&\quad (\quad 0.6502 - 1.0803\, roughness) * \max\left(1 - \mathbf{n} \cdot \mathbf{v}, 0.5637\right) + \\
&\quad (-0.5993 + 1.0632\, roughness) * \max\left(1 - \mathbf{n} \cdot \mathbf{v}, 0.5637\right)^2 \\
ZH_1 &= (\quad 1.1635 - 0.0304\, roughness + 0.0162\,(1 - \mathbf{n} \cdot \mathbf{v}))\ ZH_0 \\
ZH_2 &= (\quad 0.5725 - 0.0569\, roughness + 0.0386\,(1 - \mathbf{n} \cdot \mathbf{v}))\ ZH_0
\end{aligned}$$

Since the multipliers for ZH_1 and ZH_2 change very slowly across the domain, they can be simplified

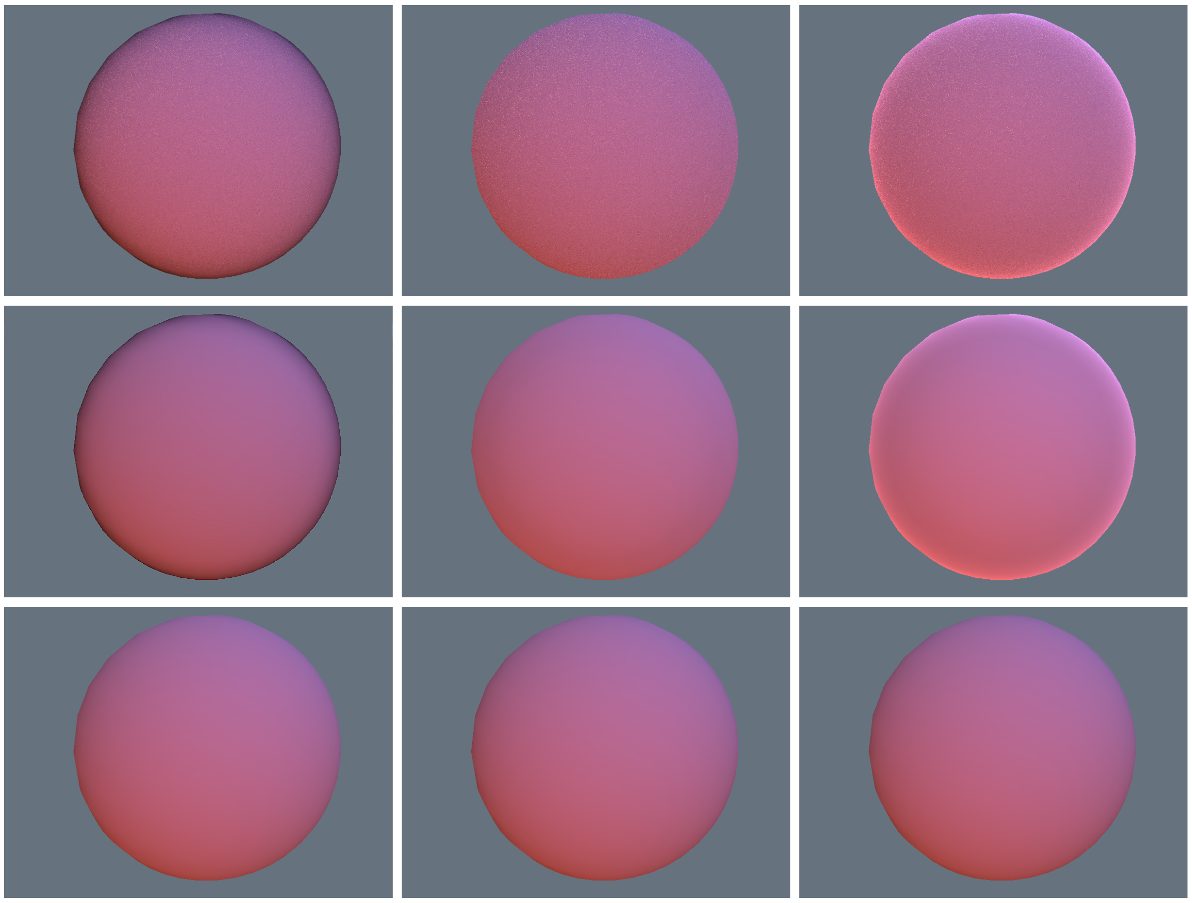


Figure 29: Comparison of the ground truth (importance sampled) version of the Disney Diffuse BRDF (top row), the proposed approximation (middle row) and the regular Lambertian model (bottom row) for different roughness values (left to right: 0.0, 0.5, 1.0).

even further, to constants:

$$ZH_1 = 1.1564\,ZH_0$$
$$ZH_2 = 0.5634\,ZH_0$$

7 Case Study: Environmental Lighting with a Microfacet BRDF

A popular solution for approximating environmental lighting with physically based models was presented in an earlier incarnation of this course [**Lazarov13**; **Karis13**]. It relies on two main simplifications:

- The integral of the product of the lighting and BRDF is approximated as the product of two separate integrals
- The integral containing the lighting (a preconvolved cubemap) assumes a fixed, 'top-down' view direction ($\mathbf{v} = \mathbf{n}$)

The main problem with the latter assumption is that shape of microfacet-based BRDFs can change significantly with viewing angle, from totally symmetric when view vector is parallel to the normal direction, to highly anisotropic and stretched at grazing angles. We wanted to develop a model capable of capturing these features.

As with the diffuse model covered earlier, we'll fold the cosine term into the approximation. Also, for simplicity, we will assume a fixed value for the F0 parameter and restrict ourselves to the isotropic version of the BRDF.

Simple, visual analysis of CBRDF we want to model doesn't show any obvious pattern. The function changes significantly with both roughness and view direction. Also, automatic solutions fail on such a broad domain.

7.1 Coordinate System

Before digging into the details of the various approaches we tried, let's first establish some basic terminology and conventions that are used in the following sections.

All of the fitting is performed in the local, tangent space of the point being shaded. The viewer is assumed to be looking along the $-\mathbf{v}$ direction that lies on the x-z plane. Since the CBRDF lobe lies on the same plane and we can remove one degree of freedom and restrict our reasoning to that plane only. The symbol θ will be used to denote angles on that plane (with $\theta = 0$ indicating the "up" direction, parallel to the normal). We assume that the BRDF lobe is in the quadrant across the normal from the viewing direction, and we will implicitly place the approximated model there. Figure 30 shows the described coordinate system.

7.2 Mixture of Spherical Gaussians

Our initial idea for the model was to use a mixture (weighted sum) of Spherical Gaussian (SG) lobes. Gaussian lobes are radially symmetric, so an environment preconvolved with a whole set of them can be stored in a cubemap, with higher MIPs storing results of convolutions with sharper lobes. Also, the math of Spherical Gaussians is relatively simple: some integrals can be evaluated in closed form, it's easy to sample (which we will take advantage of later) etc.

A Spherical Gaussian has the form:

$$\mathrm{SG}(\omega, a, \lambda, c) = c \cdot e^{-\lambda(1-\omega \cdot a)}$$

We started with a fixed number N of SG lobes. Each lobe had a separate amplitude (c_i), bandwidth (λ_i - which controls the width of the lobe) and direction (a_i):

$$\mathrm{SGMix}(\omega) = \sum_{i=1}^{N} \mathrm{SG}(\omega, a_i, \lambda_i, c_i)$$

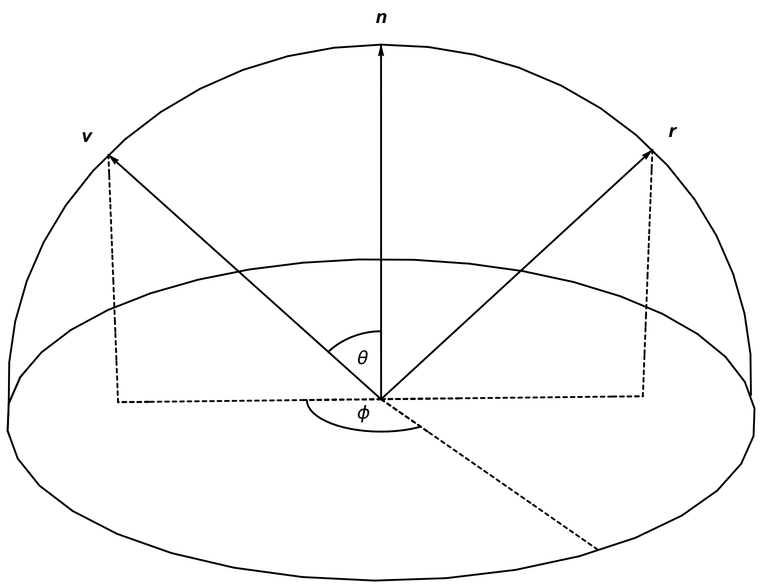


Figure 30: Coordinate system used.

Given the presented model, we tried performing the fit to the BRDF for a fixed view direction and roughness value. The spherical domain was sampled uniformly, and the error was computed as the sum of squared differences between the value of the CBRDF and the model over all sample directions. The L-BFGS algorithm was used to find the minumum of that error function, by choosing the optimal placement of the Gaussian lobes on the sphere and their parameters.

Unfortunately, the problem of this method is that we weren't be able to properly calculate the error in situations where the BRDF response or some of the Gaussian lobes fell between the sampling directions. Sharper BRDF responses require higher sampling rates and so the cost of calculating the error quickly becomes prohibitive. Also, an solver can often find local minima where some of the lobes are placed between the sampling directions, effectively removing them from the solution.

Despite these problems, we were able to find a resonable to a set of CBRDF responses for a fixed roughness value and varying viewing angle. This is when we discovered another problem. The fit for each viewing angle was performed independently, so corresponding lobes in neighboring fits were not consistent. When interpolating, this caused very distracting artifacts. As a post process, we tried sorting the lobes, which helped a bit with the interpolation, but the results were still not perfect. It became obvious that interpolating over both roughness *and* viewing angle, would only make the problem more apparent. One possible solution to this, which is used in similar applications ([**Green06**], [**Han07**]), would be to introduce additional terms to the error function that penalize variation between corresponding lobes in neighboring sampling points. This, however, requires solving for all the viewing angles and roughness values simultaneously, which means there are many more degrees of freedom in the error function and many more local minima. Searching for the global minima thus becomes time consuming and ineffective. In our experiments, we weren't able to find a resonable fit for the whole roughness $\times$ viewing angle domain this way.

7.3 A Better Way of Calculating the Approximation Error

The solution for the first problem - sampling - is based on Monte Carlo integration methods.

The error of approximating the BRDF response for a given model M, with parameters P in some direction ω, can be defined as

$$E\left(\mathrm{P},\omega\right) = \left(\mathrm{BRDF}\left(\omega\right) - M\left(\mathrm{P},\omega\right)\right)^2. \tag{7}$$

The total error of the approximation is the integral of Equation 7 over the entire unit sphere[2]:

$$E\left(\mathrm{P}\right) = \int_\Omega E\left(\mathrm{P},\omega\right) d\omega = \int_\Omega \left(\mathrm{BRDF}\left(\omega\right) - M\left(\mathrm{P},\omega\right)\right)^2 d\omega. \tag{8}$$

Since the error function is an integral, its value can be computed using Monte Carlo (or Quasi-Monte Carlo) methods. Monte Carlo integration relies on repeatedly sampling the function over the domain to compute the integral. (Q)MC methods have been applied in computer graphics for a while now, and the standard tricks can be used to compute the value of the error function.

While sampling the domain uniformly and summing up the square differences between the approximated and the actaul value, as descibed previously, is in fact a variation of numerical integration, is not the most effective one by any means. We can get much more accurate results using fewer samples if we replace the uniform sampling with importance sampling. Importance sampling tries to sample the function being integrated with a probability distribution that is similar to the function being integrated—so that the areas with high values are sampled more densely. We don't know the exact shape of the error function, but we can safely assume it's going to have some multi-modal distribution, with one of the modes representing the BRDF term and the other ones representing the approximation.

In cases with multi-modal integrands, we can use Multiple Importance Sampling (MIS) to create a sampling scheme that effectively combines multiple estimates (see [**Veach97**]). The simplest case involves only two distributions, p_1 and p_2 (with ξ_1 and ξ_2 being samples drawn from those distributions). Drawing the same number of samples from each of them, MIS boils down to drawing a sample according to one distribution, calculating its probability according to the other one and weighting the samples by the inverse of the sum of the two PDFs:

$$\int_\Omega F(x) \approx \frac{1}{N}\sum_{i=1}^{N} \frac{F(\xi_1^i)}{p_1(\xi_1^i) + p_2(\xi_1^i)} + \frac{F(\xi_2^i)}{p_1(\xi_2^i) + p_2(\xi_2^i)}.$$

Importance sampling of the microfacet BRDF with GGX distribution is described in detail in a number of sources, including [**Walter07**].

One concern with using probabilistic methods for computing the integrals is the noise in the result. If not enough samples are used, the function landscape is going to be noisy, with many peaks and valleys caused not by actual function minima but by the noise. Using low discrepancy sequences (such as the Halton sequence), a large number of samples (around 30k-50k is sufficient in this application) and performing all the calculations using double-precision arithmetic leads to practically noise-free results (see Figure 31).

Computing error values using probabilistic methods doesn't automatically generate the partial derivatives of the error functions. While they could be computed, the evaluation of the error function is pretty costly by itself and avoiding additional calculations is preferable. We decided to use the Nelder-Mead simplex method for finding the parameters for the model, as it doesn't require explicit partial derivatives. Whilst there are variations of the Nelder-Mead method that can deal with uncertainty of the function value at simplex vertices, the function landscape was smooth enough, so they weren't necessary in this case.

[2]Even though the BRDF response is only non-zero in the upper hemisphere, integration is performed over the entire sphere because we want to penalize models that generate non-zero values in the lower hemisphere.

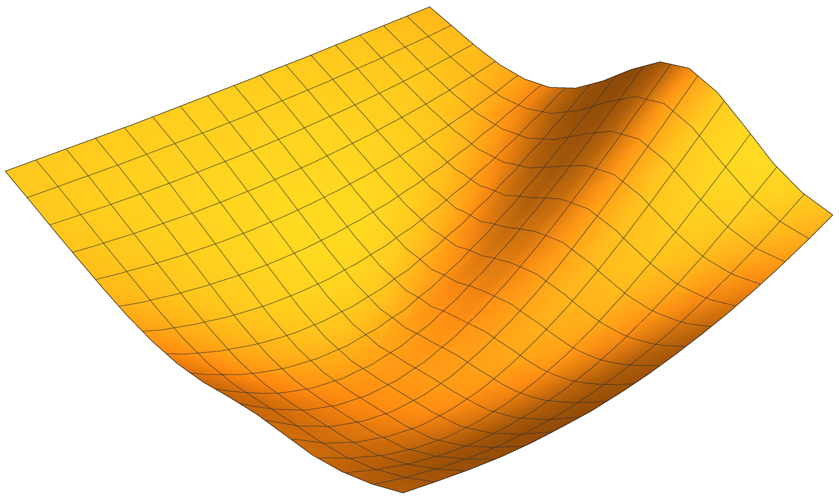


Figure 31: Noise-free function landscape. Function values were calculated using QMC methods.

7.4 Anisotropic Spherical Gaussian

At some point during our research, colleague Paul Edelstein suggested using an anisotropic texture lookup to produce elongated highlights on cubemap reflections. Instead of manually sampling the environment multiple times, hand-calculated derivatives would be passed to the texture functions, and the hardware would automatically figure out the appropriate number of samples, locations and weights. This seemed like an elegant utilization of existing filtering hardware, and would greatly simplify the fit. Instead of having to determine parameters for multiple lobes, we only only need to fit a single, anisotropic lobe. At runtime, the lobe anisotropy parameters would be passed to the texture lookup as texture coordinate derivatives. This would eliminate all problems with inconsistent lobe directions, because we would be only interpolating parameters for a single lobe, and they should be consistent.

As the underlying mathematical model for the shape of the filtering kernel of the anisotropic lookup from the cubemap, the Anisotropic Spherical Gaussian (ASG) was chosen. It was introduced by [**Xu13**]. It has the form

$$\mathrm{ASG}(\omega, \lambda, \mu, c) = c \cdot \max(0, \omega \cdot \mathbf{z}) \cdot e^{-\lambda(\omega\cdot\mathbf{x})^2 - \mu(\omega\cdot\mathbf{y})^2}.$$

$\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ define a local frame for the lobe, λ and μ parameters control the tightness of the lobe on $\mathbf{x}$, $\mathbf{y}$ axes, c is the amplitude of the lobe.

Fitting the ASG to a given BRDF response means finding the θ angle for the lobe axis and the λ and μ parameters. (We assume θ will lie on the plane formed by the normal and view vectors, so fitting one angle is sufficient.) The amplitude should be set so that the integral of the ASG is the same as the integral of the BRDF response.

To efficiently perform the error calculations, we also need a way to importance sample the lobe. With importance sampling, the closer the PDF approximates the integrand, the more effective the process, but they don't have to match exactly. In our case the chosen function only roughly approximates the lobe.

As with SGs, importance sampling of the ASG is performed in the local space of the lobe, without loss of generality. The values of the Gaussian are generally positive for the whole hemisphere, but for

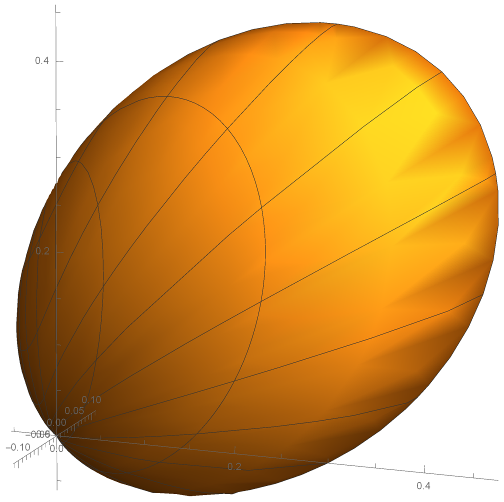


Figure 32: Anisotropic Spherical Gaussian.

higher values of λ and μ they fall very quickly with the angle from the z axis. For the purpose of sampling, we clamp the values below a certain threshold to 0. The remaining values create an ellipse on the surface of the unit sphere, which we project down onto the unit disc. To sample the ASG, we perform uniform sampling of the unit disc, scale the sample positions along the **x** and **y** directions so they lie within the ellipse formed by the clamped, projected lobe, and project them up to the surface of the unit sphere. This creates a distribution of points within the interesting region of the lobe, with a slightly higher density of the samples near the pole of the lobe. While such a distribution is not perfect—it doesn't sample areas below the prescribed threshold and it isn't strictly correlated with the lobe—it's extremely simple and can be successfully used to perform importance sampling for MC integration of the error function.

To perform importance sampling using directions generated in the above way, we need to know the PDF of this distribution. We start from a uniform distribution on the unit disc (so the uniform PDF equals $\frac{1}{\pi}$) and we apply a series of transformations. To get the PDF of the final distribution, we need to multiply the initial PDF by the ratio of the differential area resulting from those transforms (the Jacobian of the set of transforms). Given x and y on the unit disc, we first perform scaling:

$$\begin{aligned} x' &= s_x x \\ y' &= s_y y \end{aligned}$$

and later project up, onto the surface of the hemisphere

$$\begin{aligned} x'' &= x' \\ y'' &= y' \\ z'' &= \sqrt{1 - x''^2 - y''^2} \end{aligned} \tag{9}$$

The differential area dA on the unit disc is transformed to dA''. We can compute the dA'' as the length of the cross product of the partial derivatives of the $[x'', y'', z'']$ with respect to x and y (those partial derivatives will be vectors tangent to the surface of the unit sphere; the length of the cross product will be the area of the parallelogram spanned by them).

$$
\begin{aligned}
\frac{\delta[x'', y'', z'']}{\delta x} &= \left[s_x, 0, -\frac{x s_x^2}{\sqrt{1-(x s_x)^2-(y s_y)^2}} \right] \\
\frac{\delta[x'', y'', z'']}{\delta y} &= \left[0, s_y, -\frac{y s_y^2}{\sqrt{1-(x s_x)^2-(y s_y)^2}} \right] \\
\vec{N} &= \frac{\delta[x'', y'', z'']}{\delta x} \times \frac{\delta[x'', y'', z'']}{\delta y} \\
&= \left[\frac{s_x^2 s_y x}{\sqrt{1-(x s_x)^2-(y s_y)^2}}, \frac{s_x s_y^2 y}{\sqrt{1-(x s_x)^2-(y s_y)^2}}, s_x s_y \right] \\
\text{length}(\vec{N}) &= \frac{s_x s_y}{\sqrt{1-(x s_x)^2-(y s_y)^2}} \\
&= \frac{s_x s_y}{z''}
\end{aligned}
$$

So the PDF for the direction sampled according to (9) will be the PDF for uniform sampling of the disc ($\frac{1}{\pi}$) divided by the Jacobian:

$$\text{PDF}([x'', y'', z'']) = \frac{z''}{s_x s_y \pi}$$

The sampled direction is later transformed to the coordinate system aligned with the surface normal. Since this transformation is just a rotation, it doesn't change the calculated PDF.

7.5 Problems with Anisotropic Spherical Gaussian Lobes

The fit of the anisotropic Gaussian lobes to the GGX-based BRDF was performed over the domain of view angles and roughness values. The results were very encouraging. The fit was showing the anisotropic characteristics of the GGX distribution near grazing angles and interpolating properly over the whole domain. The example fit is shown in Figure 33.

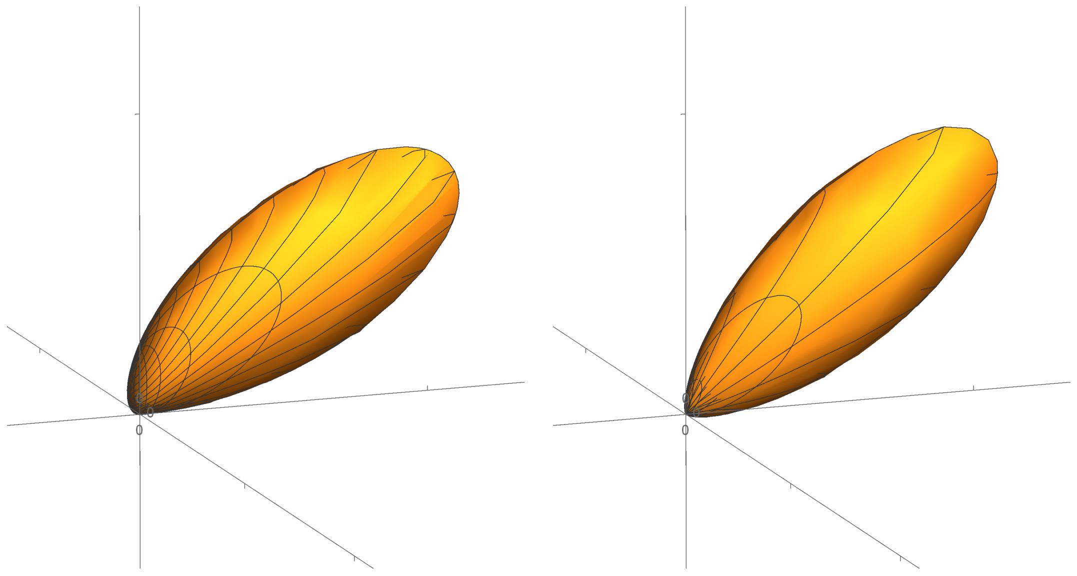


Figure 33: Polar plot of an ASG (right) fit to a response of the BRDF with GGX NDF(left).

Unfortunately, hardware anisotropic filtering of the cubemaps turned out to be unusable as an approximation of the ASG. First of all, hardware vendors are reluctant to disclose any information about the details of their anisotropic filtering kernels, for competitive reasons[3]. Secondly, whilst those details could be reverse engineered, there is a limitation common to NVIDIA and AMD hardware: all of the anisotropic taps are restricted to the dominant cube face.

Figure 34 compares the results of performing an anisotropic lookup with overridden gradients into a cubemap with false-colored MIP levels, with reference DirectX rasterizer (which we consider to be the gold standard), NVIDIA and AMD hardware. This experiment highlights some further issues as well as hardware differences. NVIDIA hardware seems to fall back to trilinear filtering if the filtering kernel overlaps cubemap edges. There also appear to be approximations in the calculation of the filter footprint, as indicated by the weird, flower-like patterns. The behavior of the AMD hardware is also surprising: the results of the lookup are not symmetric.

In conclusion, the insufficient quality and inconsistencies between vendors makes the use of anisotropic hardware lookup impractical for our purpose.

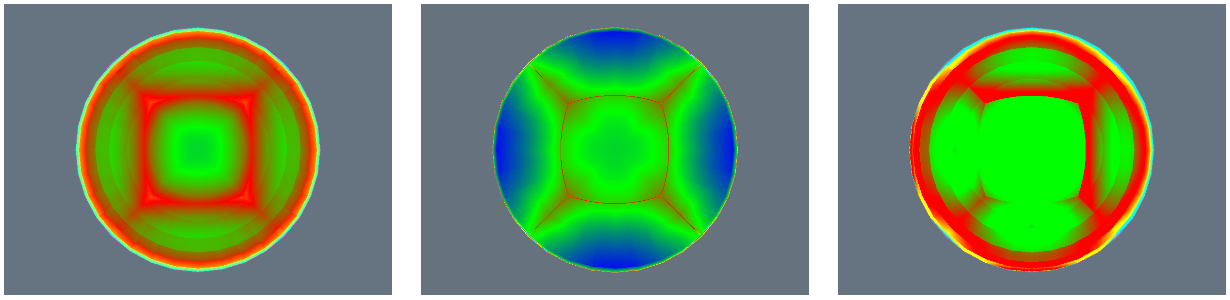


Figure 34: Results of the anisotropic cubemap lookup with overriden gradients on the reference rasterizer (left), NVIDIA (center), and AMD (right) hardware.

7.6 Multiple Gaussian Lobes

Using multiple Gaussian lobes benefits from robust hardware support, but suffers from problems with interpolation. In contrast, anisotropic lobes interpolate very well, but we cannot utilize them due to hardware constraints. The logical solution was to combine these two ideas: create a two-layered model, consisting of multiple SG lobes internally, but with their individual parameters being controlled by a small set of external parameters that would describe the overall shape of the kernel.

The new model was inspired by the model used for constructing the anisotropic filtering kernel in [**McCormack99**]. It distributes the lobes uniformly, along the θ angle, over some range of angles. All of the lobes share the same λ value, and their amplitudes follow a Gaussian curve, with the standard deviation based on how much the lobes are spread. We performed fitting in two variants. In the first one we uniformly scale the amplitudes of all the lobes so that the integral of the whole approximated lobe over the sphere is equal to the integral of the GGX lobe we want to approximate. This effectively removes one of the variables from the fit and forces the fit to preserve the 0^{th} moment of the original signal, which is sometimes practiced in similar applications. The second variant leaves the amplitude as a free variable. Assuming some number of lobes, this gives us either three or four degrees of freedom - θ angle for the center direction of the lobe set, a spread value (describing the spacing between individual lobes) a common λ parameter for all the lobes, and optionally the amplitude.

An example of the model is shown in Figure 35.

[3]Behavior could also change significantly across hardware generations, since DirectX and OpenGL allow vendors considerable latitude in how anisotropic filtering is implemented.

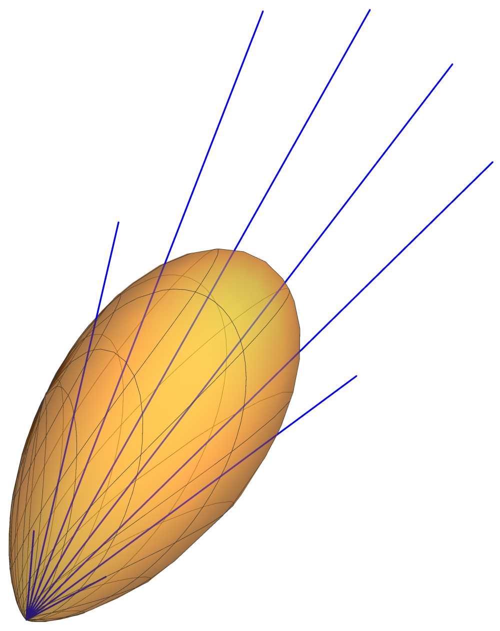


Figure 35: Example of spherical function generated by the simplified, multi-lobe model. The lobe is constructed out of 8 SG lobes. They share the λ value. The directions of individual SG lobes are depicted as blue lines, with lengths proportional to lobe amplitudes

The θ and amplitude parameters for individual lobes are calculated as follows:

$$\theta_i = \left(\frac{2i}{N-1} - 1\right) \text{spread} + \theta_{\text{center}} \tag{10}$$

$$\text{amplitude}_i = \text{norm} \cdot e^{\frac{-\text{spread}}{3}\left(\left((i+1) - \frac{N+1}{2}\right)\left(\frac{2i}{N-1} - 1\right)\right)^2},$$

where N is the number of lobes, θ_{center} is the θ angle of the center direction, and *norm* is the normalization factor that causes the whole model to integrate to the same value as the BRDF response.

Importance sampling of an SG is straightforward. First of all, we can derive a Probability Density Function (PDF) directly from the SG. We can achieve this by multipling the SG lobe by a normalization factor, equal to the reciprocal of the integral of the SG over the sphere. The integral is equal to:

$$\begin{aligned}\int_{\Omega} c \cdot e^{-\lambda(1-\omega.[0,0,1])} d\omega &= \int_0^{2\pi} \int_0^{\pi} c \cdot e^{-\lambda(1-\cos\theta)} \sin\theta \, d\theta d\phi \\ &= c \cdot \frac{2\pi \left(1 - e^{-2\lambda}\right)}{\lambda}\end{aligned}$$

which gives us a PDF of:

$$\text{PDF}_{\text{SG}}(\omega) = \frac{\lambda}{2\pi \left(1 - e^{-2\lambda}\right)} e^{-\lambda(1-\omega.[0,0,1])}$$

$$\text{PDF}_{\text{SG}}(\theta, \phi) = \frac{\lambda}{2\pi \left(1 - e^{-2\lambda}\right)} e^{-\lambda(1-\cos\theta)}$$

To derive the sampling rules given the PDF we will use marginal and conditional PDFs:

$$\begin{aligned}
\mathrm{PDF}_{\mathrm{SG}}(\theta, \phi) &= \frac{\lambda}{2\pi\left(1-e^{-2\lambda}\right)} e^{-\lambda(1-\cos\theta)} \\
\mathrm{PDF}_{\mathrm{SG}}(\theta) &= \int_0^{2\pi} \mathrm{PDF}_{\mathrm{SG}}(\theta, \phi)\, d\phi \\
&= \frac{\lambda}{2\pi\left(1-e^{-2\lambda}\right)} e^{-\lambda(1-\cos\theta)} \left(x\right)\Big|_0^{2\pi} \\
&= 2\pi \frac{\lambda}{2\pi\left(1-e^{-2\lambda}\right)} e^{-\lambda(1-\cos\theta)} \\
&= \frac{\lambda}{1-e^{-2\lambda}} e^{-\lambda(1-\cos\theta)} \\
\mathrm{CDF}_{\mathrm{SG}}(\theta) &= \int_0^{\theta} \mathrm{PDF}_{\mathrm{SG}}\left(\theta'\right) \sin\theta\, d\theta' \\
&= \frac{e^{2\lambda} - e^{\lambda(1+\cos\theta)}}{e^{2\lambda}-1} \\
&= \frac{1-e^{-\lambda(1-\cos(\theta))}}{1-e^{-2\lambda}} \\
\mathrm{PDF}_{\mathrm{SG}}\left(\phi|_{\theta}\right) &= \frac{\mathrm{PDF}_{\mathrm{SG}}(\theta, \phi)}{\mathrm{PDF}_{\mathrm{SG}}\,\theta} \\
&= \frac{1}{2\pi} \\
\mathrm{CDF}_{\mathrm{SG}}\left(\phi|_{\theta}\right) &= \int_0^{\phi} PDF_{\mathrm{SG}}\left(\phi'|_{\theta}\right) d\phi' \\
&= \frac{\phi}{2\pi} \\
\cos\theta &= \frac{\lambda + \log\left(1 + \left(e^{-2\lambda}-1\right)\xi_0\right)}{\lambda} \\
\phi &= 2\pi\xi_1
\end{aligned}$$

Directions sampled in the local space of the Gaussian are next transformed to the appropriate coordinate system.

This model performs reasonably well (see Figure 36 for an example fit). It is able to express the elongation of the BRDF near grazing angles, although it puts more energy to the ends of the lobe, compared to the ASG approximation (this is caused by the same λ being used for all the lobes). The degree of anisotropy the model is able to express depends on the number of internal lobes. In the case of the GGX distribution, eight lobes are able to approximate the responses across the domain without noticeable artifacts. If the anisotropy of the BRDF response is too large for the number of SGs we're using, it will manifest itself as 'wobbly' highlights with multiple peaks.

When evaluating the error integral, each lobe is treated as a separate distribution and the lobes are combined using multiple importance sampling.

Numerical solvers tend to be more stable when all of the variables are within the same order of magnitude and change at similar rates. In the case of the presented model, the overall amplitude and the λ parameters tend to change in a nonlinear fashion. To overcome this, the fit is performed in the logarithmic domain for those two parameters. The coordinates of the simplex vertices are exponentiated to obtain the actual values. The θ angle of the fit model changes according to $\cos\theta$, because fitting the angle directly creates worse distribution of samples when doing the runtime rendering. Also, all the coordinates are scaled and restricted to the $[0, 1]$ range.

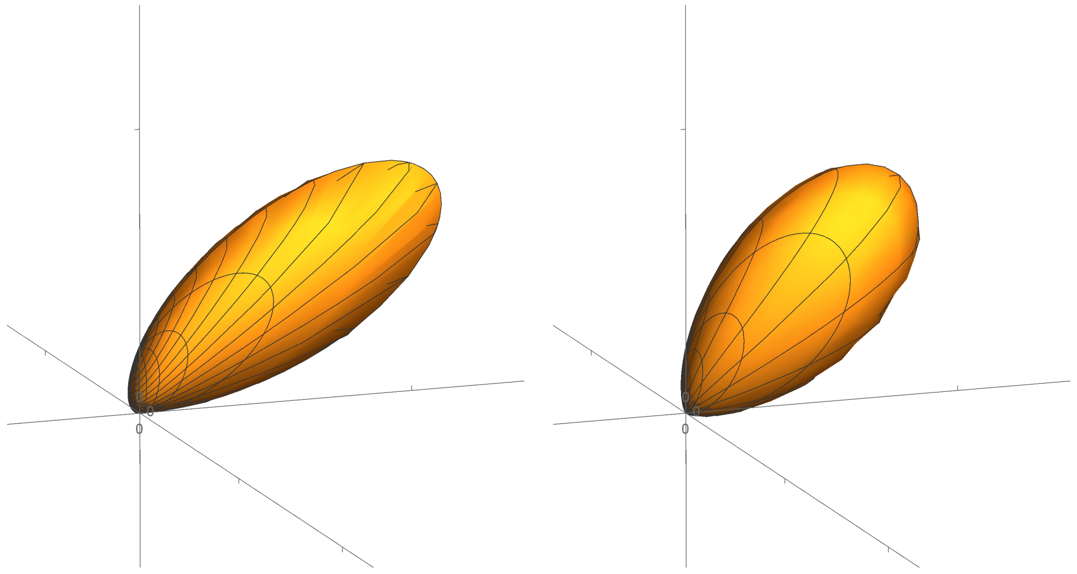


Figure 36: Polar plot of a SG mixture with 4 lobes (right) fit to a response of the BRDF with GGX NDF (left).

One last problem that we had to deal with were the visual artifacts observable for high roughness values when **v** was close to **n** (see Figure 37, left). As it turned out, those were visibile when the spread value for the approximate model was larger than the θ value. In such cases, the fast change of the local coordinate system around the **n** pointing towards the camera causes the samples for the individual lobes to be taken from vastly different positions in the cubemap, even for neighboring pixels, which creates radial artifacts. This was solved by restricting the spread value to be *no greater* than the θ angle. This can be done trivially when using Nelder-Mead, by simply clamping the spread value before computing the error. This effectively flattens the error function landscape to the value of the error when spread equals θ and causes the solver not to explore that area. Since the error function landscape for the high roughness values is fairly flat anyway, this doesn't increase the approximation error in any significant way. The fit for low roughness values is unaffected, since the global minima have spread values siginificantly lower than θ values.

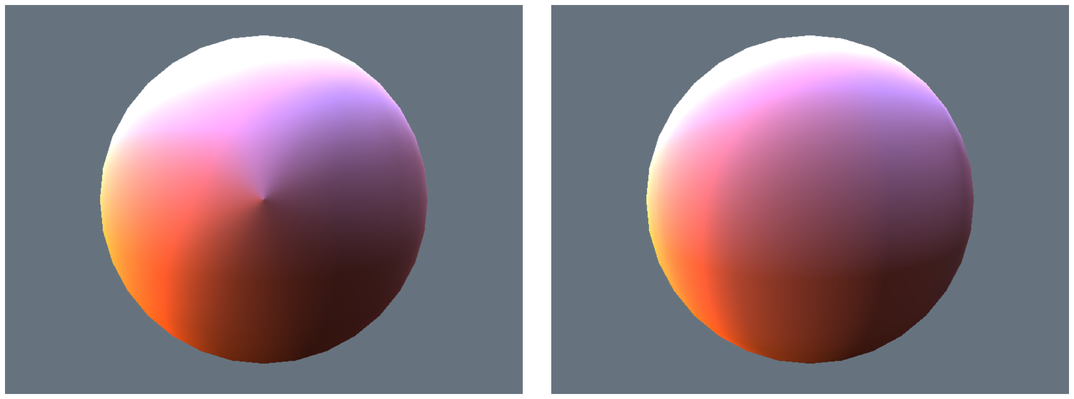


Figure 37: Artifacts visible for low roughness valueswhen no constraints are placed on the spread value (left). They are elimineted by ensuring that spread values is lower than θ

The parameters fitted for the whole domain are tabulated in a form of small textures (16^2 in our case) and passed to the shaders. The fits for individual parameters are shown in Figure 38. The pixel shader fetches the parameters for the approximation based on viewing angle and the roughness of the surface, then reconstructs the sample directions and performs lookups into a prefiltered cubemap. The

λ parameter is used to determine the MIP level to sample from, and the individual taps are weighted according to Equation (10) and summed to get the final result.

There is very little difference between the fit with the amplitude being free variable vs. forcing it to make the integrals of approximation and the base mode equal. Allowing amplitude to change freely makes it approximate the GGX lobe for grazing angles slightly better. The attached Mathematica notebook contains the results for both variants, together with some simple tools to visualize them.

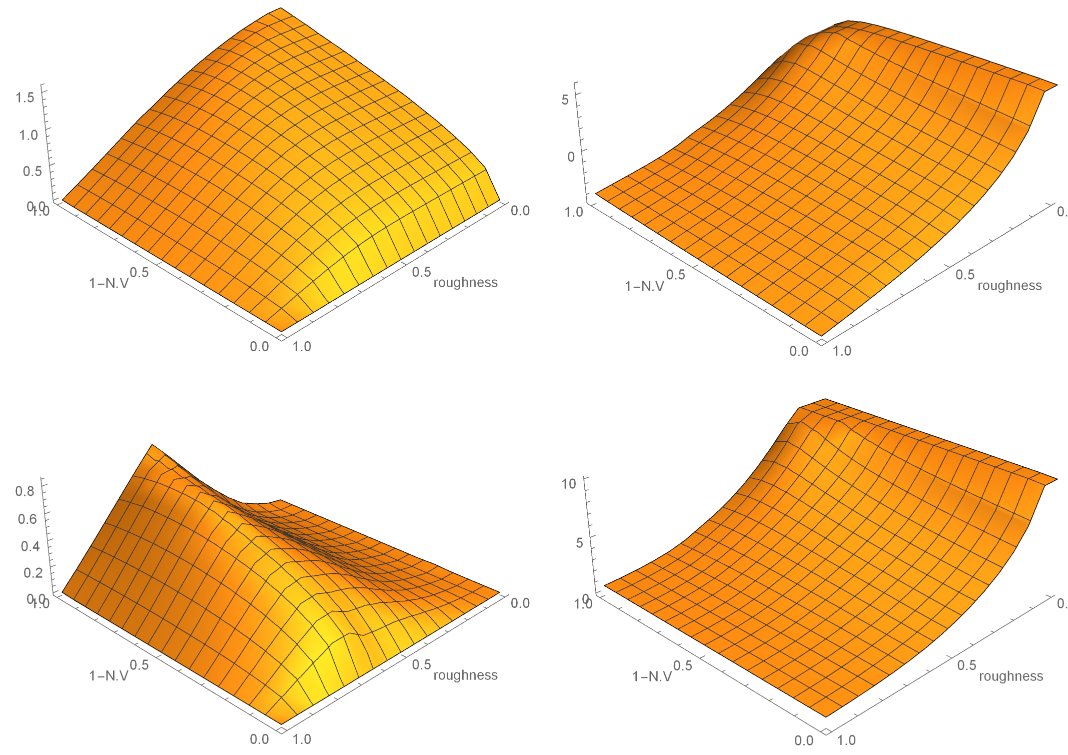


Figure 38: Fit of the parameters across view angles $\times$ roughness domain. Clockwise, from top-left: θ angle, log(base amplitude), log(λ), spread.

Figure 39 shows the comparison of the importance sampled, ground truth rendering, the approximation with multiple lobes, and the most common approximation with a symmetrical integration kernel.

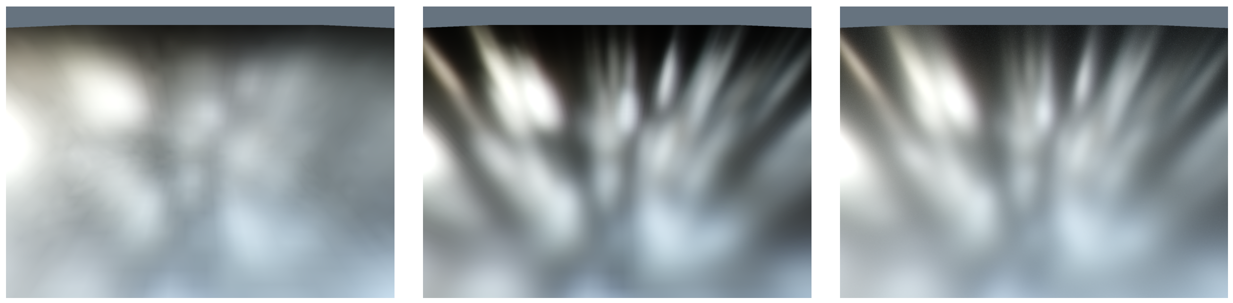


Figure 39: Comparison of the common approximation (left), the multi-lobe approximation (center), and the ground truth (right) rendering of cubemap reflections. Material roughness was set to 0.3

7.7 Issues/Ideas for the Future

The ideas presented here form some basic building blocks for getting more accurate image-based lighting for the more complex BRDFs used in modern applications. They can be further extended to provide more specialized solutions.

One straightforward improvement would be to use the GGX distribution instead of a Spherical Gaussian as the basic building block. A Spherical Gaussian was used due to its simplicity, but the final model doesn't rely on it in any way. Because the only operation we actually perform on the lobes is importance sampling, we can easily replace them with other functions. SGs don't have the characteristic 'long tail' of GGX, so using the GGX distribution directly as a basis function is a good idea if simulating this effect is important. And while hardware anisotropic filtering is unreliable with high degrees of anisotropy, it could be utilized to provide some, so that final kernel is composed of a number of slightly anisotropic base kernels.

The described model assumes a Gaussian distribution of amplitudes of individual lobes and a uses a constant λ value for all of them. Instead, a slightly more sophisticated model could be used that is closer to the characteristics of the BRDF response we are trying to model. For example, the BRDF response narrows and gets stronger at grazing angles, which could approximated. The resulting model would still have only a few external parameters generated during the fit, so it would interpolate smoothly and not create visual artifacts.

In the presented example, we use a constant number of lobes for the whole domain. The higher numbers of lobes are needed in cases when the BRDF response has a high level of anisotropy, but in many cases are unnecessary—actually, the fits of the presented model done for different number of lobes differ only in the areas of the high anisotropy! Changing the number of lobes used can be incorporated into both fit and the runtime part.

The fit results are stored in small textures, but those 2D tables could be approximated with analytic functions. Symbolic regression could be used to find reasonable approximations.

While we perform the fit to the response of the GGX-based BRDF, we're not limited by the choice of BRDF in any way. In fact, the interesting elongated highlights occur on a relatively limited range of roughness values (for low roughness, the reflection is mirror-like; at high roughness it's too blurry to notice the details) and are most visible on relatively big, flat surfaces. The same methodology can be used to perform a fit to anisotropic BRDFs. Instead of fitting over the θ × roughness domain, we would fit across $\theta \times \phi$, with a fixed roughness value (or store multiple values in slices of a small volume texture).

The described BRDF approximation can also be used in other situations. If we precompute the response of a single lobe to the illumination of a spherical light source, we can use the approximation to perform area lighting. Expanding the lobes based on the distribution of normals covered by a screen pixel, can provide a way of performing anti-aliasing of the IBL lighting (similarly to what [**Olano10**] or [**Han07**] do).

The approximation could also be combined with some screen-space filtering scheme to reduce the cost. Neighboring pixels could sample only a single lobe, and a surface-aware screen-space filter could gather samples along the lines defined by view vectors and surface normals and combine them to get more accurate results.

8 Case Study: Environment Lighting with Parallax Correction

We illustrated solutions to two of the main issues of cubemap-based, precomputed environment lighting, namely its limitation to radially-symmetric lobes and the effects of masking-shadowing and Fresnel in sections 7 and 5.3. Albeit still improvements are possible on these, we so far completely ignored other sources of error. We still not consider the surface variance (in the footprint of the camera sample) for example (an error common in many implementations), moreover we run under the assumption that our lighting environment to be infinitely far which of course is never true. The latter error can be fairly large as in practice preconvolved cubemap probes are used for indirect specular highlights even in very confined interior environments.

Recently, to partially address this problem and adapt the irradiance captured at a given fixed location in the scene to specular reflections on any point of the scene surfaces, an approach based on reprojection has become very popular [**TODO**]. The basic idea of this method is to approximate the scene geometry with a simple convex shape (proxy) with which is easy to compute a ray-intersection: usually a collection of planes or slabs (a k-DOP). We can then replace the reflection vector when doing the pre-convolved cubemap lookup with a vector that connects the cubemap probe position with the intersection between the reflection ray and the convex shape.

This technique adds to the errors and approximations of environment lighting we've seen a few more complications:

- The reprojection proxy we use has to be convex as we can't account for discontinuities. Our prefiltered irradiance incorporates the visibility from the probe perspective so if the visibility of the reprojection geometry has discontinuities these won't be filtered across at all, resulting in sharp edges visibile in otherwise blurry reflections. In fact, in our implementation the geometric proxy is made smooth proportionally to the desidered surface roughness.
- As our reprojection proxy has to be simple, for most scenes it ends up also being substantially different from the actual geometry. We typically can model the walls of a room, but all the objects inside the room are completely lost, as they are their reflections and occlusions.
- The reprojection effectively warps the cubemap into a different one, stretching some areas and expanding others. This process distorts the kernel we applied in the pre-filtering, effectively roughening or smoothing the resulting reflection inconsistently across surfaces. This is another reason why we prefer smooth proxies, where the distortion does at least vary without discontinuities that are visually very easy to spot in reflections.

The first two errors can't be solved without reprojections that can allow for discontinuties in the scene proxy, which can be achieved only with multiple samples from different pre-filtered layers. The latter error could instead be ameliorated if we can estimate the difference between the prefiltered kernel we have baked in the mip chain of the cubemap, and the kernel we need, as both project onto the proxy surface40.

We approach this problem in two steps. First we'll estimate for the projection of a specular lobe of a given roughness into a perpendicular plane at a given distance, the roughness of a second lobe perpendicular to the plane at a fixed unitary distance that best approximates the first. This will be used as a roughness correction based on the ratio of the distances of the probe from the plane and the shaded surface point from the plane.

Then we'll estimate for the projection of a specular lobe at a given angle and roughness from a plane, the roughness of a second lobe at the same distance but perpendicular to the plane that best approximates the first. This can be used as a roughness correction based on the angles of the probe and surface specular lobes to the intersected proxy plane. It is quite expensive though and so far we haven't included it in a game.

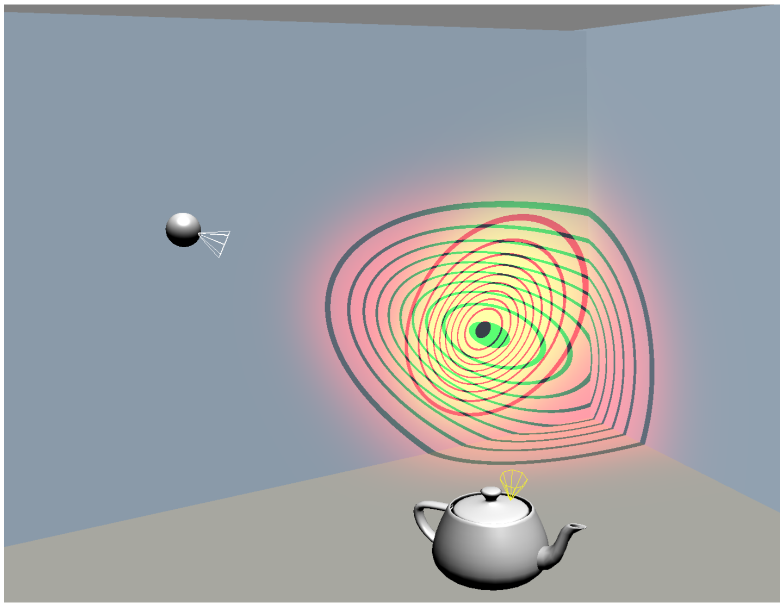


Figure 40: Visualization of two Phong lobes of the same exponent, projected on the planes of a reflection proxy. The red lobe comes from the specular probe on the left, the green lobe comes from a reflection off the teapot at the bottom.

Note that we consider a single plane; If the proxy geometry is made of multiple planes, or of other surfaces, we'll approximate the surface near the intersection as a plane with the same normal as the proxy surface at the intersection point. This is per se a source of error that can be quite large for wide lobes.

Both problems result in functions that map from two input dimensions (respectively: roughness and distance, roughness and incident angle) into a single output (fitted roughness of a perpendicular lobe at unitary distance) and are thus straightforward applications of the methodology we have so far described, with analytic approximations found via symbolic regression, tailored for the encoding of surface roughness we employ in our games. We'll thus only highlight here the major steps:

- First we visualized the projections of a GGX lobe to a perpendicular plane at varying roughness and distance 41. Note how it's possible to visually confirm a match along the diagonal of the graph between the projections at increasing distances and the projections of rougher lobes.
- From the same graph we note also that there appears to be a constant tail, off the lobe peak (typical of GGX) that can't be easily reproduced by the varying distances. We decided to run the minimization ignoring this tail by allowing the fit to vary an additive and multiplicative constant, together with the lobe roughness42. These extra parameters were used in the fit but then discarded. We could have considered the multiplicative constant though as it doesn't change the pre-filtering.
- Similarly we visualized the effects of angle of incidence on the shape of the projected kernel, and determined that a good fit could be done by allowing other than a roughness change an extra parameter which controlled the shift of the fitted perpendicular lobe on the plane. Assuming that the center of the projection is our ray intersection we computed a second point on the plane corresponding to the projection of the ray origin onto it. The shift parameter allows the fitted lobe to lie anywhere on the line between these two points43, and it was used only during the optimization phase, the resulting function 44 was then discarded.

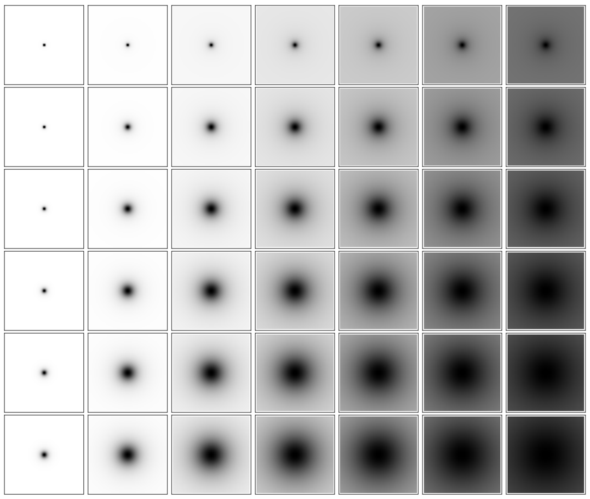


Figure 41: Projection of a GGX lobe into a perpendicular plane. From left to right: increasing roughness. From top to bottom: increasing distance.

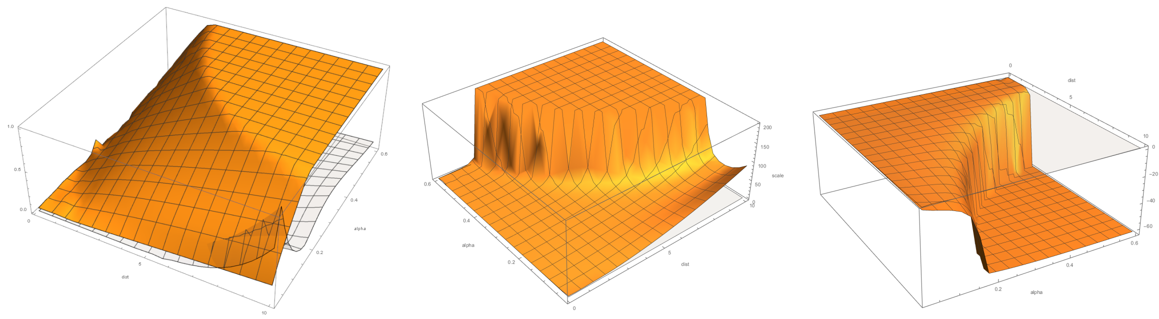


Figure 42: From left to right: fitted roughness, scale and constant offset for varying source lobe distance and roughness. The plateau doesn't represent an error in the fitting, but the limits of our prefiltered kernel.

- We can then use the angle and roughness to perpendicular lobe roughness function to find the cubemap mipmap than when projected yields a perpendicular roughness similar to the perpendicular roughness that the surface specular BRDF projects onto the proxy plane. In both cases we have to adjust roughness for distance first.
- Note that for this angle compensation to work is imperative to have a proxy shape with smooth normals.

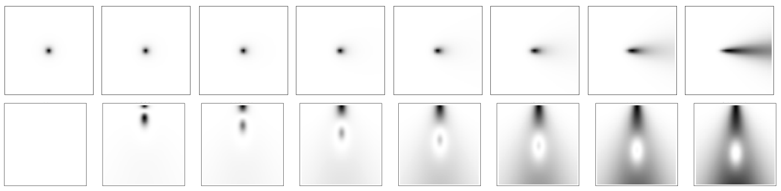


Figure 43: Top: the projection of a specular lobe of a fixed roughness into a plane at different angles of incidence. Bottom: the difference between a lobe projected at different angles and a fitted perpendicular lobe allowed to lay anywhere on the line between ray intersection and ray origin projection.

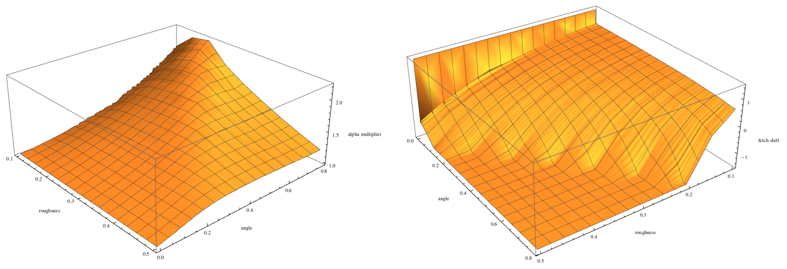


Figure 44: Left and right respectively: fitted roughness, and shift factor for varying source lobe angle and roughness.

9 Some notes on Visualization

- scalar field visualization in many dimensions is hard - discontinuities are hard! robust graphs... - more than 4d is very hard. 4d-¿volumetric, point cloud, contours (similar to 3d...) - multiple projections - cite some interesting papers... - interactive exploration is the easiest!

10 Conclusions

We illustrated a very general methodology useful to approximate empirically derived data and analytic functions with symbolic expressions. Its main steps are:

- Data gathering and/or computation.
- Low dimensional data.
  - Can we store it in a lookup table?
  - Can we can derive a simple expression (symbolic regression)?
- High-dimensional data.
  - Data exploration via visualization.
  - Dimensionality reduction via projection/fitting to convenient models.
- Error analysis.
  - Is our model valid enough?
  - Should we simplify it or extend it?
  - What insights did we derive from it?

This allows us to bring to real-time rendering more phenomena derived from simulations or real data, using arbitrarily complex error measures and without relying on simplifying assumptions. The use of empirical data, acquired or simulated, does not negate the benefits of theoretical approaches formulated "ab initio" from first principles and derivations, but it does help verify that the underlying often assumptions made to make the derivations possible are indeed sound.

Also note that approximation techniques can still create considerably difficult, and should not be adopted blindly. In particular:

- It's easy to generate plausible, but wrong models. This is problematic as each error in our models potentially can propagate and be compensated in the art assets, and when the end results still are more or less subtly wrong having to change assets usually is an expensive fix.

  - Every step has to be checked, from the numerical algorithms to the final image.
  - Preparing multiple ground truths, via both simulation and acquisition.
  - Visualization helps a lot, from verifying model fitting to understanding convergence of numerical algorithms, but care has to be taken to property chose the scales of each axis. Visualization can itself be misleading.
  - One-off global approximations it's easier to verify their correcness, but local, asset-dependent approximations need actual guarantees of robustness.

- It's easy to generate models that dont respect key constrains or misbehave at certain points on the domain.
  - Check extrapolation, overfitting, discontinuities.
  - Check if fitted parameters are hitting constrain bounds.
  - Check error metrics employed.
  - A good understand of the problem that we are approximating is necessary to select which contraints are needed.

- Fitted models are specific to a given problem. Most changes (e.g. going from a Blinn BRDF to GGX) require refitting of all the models that depend on it in the best case, in the worst it might require finding entirely new models to fit.

In the future we would like to see more work dedicated to the verification of all the methods used in real-time physically based rendering, with more emphasis on acquired data beyond simple comparison of photographs and simulations. We would also like to explore more the error metrics we employ, especially regarding perceptual error estimators.

We also presented in this only case studies that deal with offline global approximations, deriving formulas that are typically to be coded in shaders and used everywhere, but there is lot of potential in local approximations specially crafted to the statistics of our scenes, e.g. approximations done per ambient or computed per surface element or per voxel in a volumetric discretization of our environments. These models will likely require faster and more stable automated methods, thus a compromise between the expressiveness of the system and its ability to be automatically trained will probably be needed.

Lastly there are some related techniques that we would like to see applied to real-time rendering more often, from auto-tuning [**TODO**] to super-optimization [**TODO**] to automatic simplification [**TODO**]

11 Acknowledgements

- Paul Edlestein...
- XXXX [for suggesting the use of GGX instead of SGs].
- Demetrius Leal...

12 Appendix - HLSL code for Disney Diffuse BRDF

Below is a simple implementation of the presented approximation for the Disney Diffuse BRDF

```
float3 SHZHProductIntegral( float3 dir, float3 zh, float3 shLighting[9] )
{
    // VERY explicit evaluation of the integral of a product of a rotated ZH and SH
    float3 result = float3( 0, 0, 0 );

    result += shLighting[0] * zh.x * 3.544908 *  0.2820948;
    result += shLighting[1] * zh.y * 2.046653 * -0.4886025 * dir.y;
    result += shLighting[2] * zh.y * 2.046653 *  0.4886025 * dir.z;
    result += shLighting[3] * zh.y * 2.046653 * -0.4886025 * dir.x;
    result += shLighting[4] * zh.z * 1.585331 *  1.0925484 * dir.y * dir.x;
    result += shLighting[5] * zh.z * 1.585331 * -1.0925484 * dir.y * dir.z;
    result += shLighting[6] * zh.z * 1.585331 *  0.3153916 * ( 3.0f * dir.z * dir.z - 1.0f );
    result += shLighting[7] * zh.z * 1.585331 * -1.0925484 * dir.x * dir.z;
    result += shLighting[8] * zh.z * 1.585331 *  0.5462742 * ( dir.x * dir.x - dir.y * dir.y );

    return result;
}


float3 DiffuseBrdfApprox( float3 worldPosition, float3 worldNormal,
                          float3 cameraPosition, float roughness,
                          float3 shLighting[9] )
{
    float3 N            = normalize( worldNormal );
    float3 V            = normalize( cameraPosition.xyz - worldPosition );

    float NdotV         = saturate( dot( N, V ) );

    float xParamLobe    = 1.0 - NdotV;
    float yParamLobe    = roughness;

    float lobeDirAlpha  = max( 0, -0.2988 + 0.1372 * yParamLobe + 0.3775 * xParamLobe );
    float3 lobeDir      = normalize( lerp( N, V, lobeDirAlpha ) );


    float xParamZh0     = max( 1.0 - NdotV, 0.5637 );
    float yParamZh0     = roughness;
    float zh0           = (  0.0913 + 0.3067 * yParamZh0 ) +
                          (  0.6502 - 1.0803 * yParamZh0 ) * xParamZh0 +
                          ( -0.5993 + 1.0632 * yParamZh0 ) * xParamZh0 * xParamZh0;

    float xParamZh12    = 1.0 - NdotV;
    float yParamZh12    = roughness;

    float zh1Mult       = 1.1635 - 0.0304 * yParamZh12 + 0.0162 * xParamZh12;
    float zh2Mult       = 0.5725 - 0.0569 * yParamZh12 + 0.0386 * xParamZh12;

    // constant fit
    // float zh1Mult    = 1.1564;
    // float zh2Mult    = 0.5634;

    float zh1           = zh0 * zh1Mult;
    float zh2           = zh0 * zh2Mult;

    return SHZHProductIntegral( lobeDir, float3( zh0, zh1, zh2 ), shLighting ) / Pi;
}
```

13 Appendix - HLSL code for multilobe approximation to GGX

Below is a simple implementation of the presented multi-lobe approximation for the GGX based microfacet BRDF

```
static const int kNumSgMixLobes = 8;
static const int kNumMipsUsed   = 7;

// cubemap prefiltered using SG kernel
TextureCube g_sgEnvMap;

// approximation parameters
Texture2D   g_lobeParams;

SamplerState g_linearSampler;

float4 GetSgMixLobeParams( float NdotV, float roughness )
{
    static const int kLookupTextureSize = 16;
    static const float kLookupSizeMult  = (float)(kLookupTextureSize - 1) / kLookupTextureSize;
    static const float kLookupSizeShift = 0.5 / kLookupTextureSize;

    float2 uvs    = float2( saturate( NdotV ), saturate( roughness ) ) * kLookupSizeMult +
                    kLookupSizeShift;

    float4 lobeParams = g_lobeParams.Sample( g_linearSampler, uvs );

    return lobeParams;
}

float GetSgMix_FirstLobeThetaCos( float4 lobeParams )
{
    // this stores baseTheta + spread
    return lobeParams.x;
}

float GetSgMix_Spread( float4 lobeParams )
{
    return lobeParams.y;
}

float GetSgMix_BaseAmplitude( float4 lobeParams )
{
    return lobeParams.z;
}

float GetSgMix_Amplitude( float baseAmplitude, float lambda, float spread, int lobe )
{
    // lot of this can be folded and reformulated
    // to utilize underlying hardware
    // (e.g AMD GCN provides exp2 instruction, but no exp)
    // but we have left it like this for clarity
    float N = ( (float)lobe + 1.0 ) * 2.0 - ( kNumSgMixLobes + 1.0  );
    float D = N / 2.0  * ( ( (float)lobe / ( kNumSgMixLobes - 1.0 ) ) * 2.0 - 1.0 );
    float lobeAmpl = exp( -( spread / 3.0 ) * D * D );

    float lobeInt = 2.0 * PI * ( 1.0 - exp ( -2.0 * lambda ) ) / lambda;

    return baseAmplitude * lobeInt * lobeAmpl;
}

float GetSgMix_Lambda( float4 lobeParams )
{
    return exp( lobeParams.w );
}

float GetLambdaMip( int numMips, float lambda )
{
    // some simple mapping between lambda and mip
    float mipAlpha = log( lambda ) * 0.1 + 1.0;

    return numMips * mipAlpha * mipAlpha;
}

float3 SpecularBrdfApprox( float3 worldPosition, float3 worldNormal,
                           float3 cameraPosition, float roughness )
{
    float3 N                = normalize( worldNormal );
    float3 V                = normalize( cameraPosition - worldPosition );

    // construct local frame
    float3 localY           = -normalize( V - dot ( V, N ) * N );

    float4 lobeParams       = GetSgMixLobeParams( dot( N, V ), roughness );

    float lobeThetaCos      = GetSgMix_FirstLobeThetaCos( lobeParams );
    float lobeThetaSin      = sqrt( 1.0 - lobeThetaCos * lobeThetaCos );

    float spread            = GetSgMix_Spread( lobeParams );
    float spreadStep        = 2.0 * spread / ( kNumSgMixLobes - 1.0 );

    float spreadStepSin, spreadStepCos;
    sincos( spreadStep, spreadStepSin, spreadStepCos );

    float baseAmplitude     = GetSgMix_BaseAmplitude( lobeParams );

    float lambda            = GetSgMix_Lambda( lobeParams );
    float mipLevel          = GetLambdaMip( kNumMipsUsed, lambda );

    float lobeFloat         = 0;

    float3 result           = float3( 0, 0, 0 );

    for( int i=0; i<kNumSgMixLobes; ++i )
    {
        // figure out sampling direction and amplitude for current lobe
        float3 R                = lobeThetaCos * N + lobeThetaSin * localY;
        float lobeAmpl          = GetSgMix_Amplitude( baseAmplitude, lambda, spread, i );

        float3 env              = g_sgEnvMap.SampleLevel( g_linearSampler, R, mipLevel ).rgb;

        // add contribution of current lobe
        result                 += env * lobeAmpl;

        // move to the next previous lobe (actaully previous ;)
        float nextLobeThetaSin  = lobeThetaSin * spreadStepCos -
                                  lobeThetaCos * spreadStepSin;

        float nextLobeThetaCos  = lobeThetaCos * spreadStepCos +
                                  lobeThetaSin * spreadStepSin;

        lobeThetaSin            = nextLobeThetaSin;
        lobeThetaCos            = nextLobeThetaCos;
    }

    return result;
}
```