CEIG'08, Barcelona, Sept. 3–5 (2008)
L. Matey and J. C. Torres (Editors)

Fast Inverse Reflector Design (FIRD)

Albert Mas, Ignacio Martín and Gustavo Patow

Institut d'Informatica i Aplicacions, Girona, Spain

Abstract

This paper presents a new method ofr a GPU-based computation of outgoing light distribution for inverse reflector design. We propose a fast method to obtain the outgoing light distribution of a parametrized reflector, and compare it with the desired illumination, that works completely in the GPU. We trace millions of rays using a hierarchical height-field representation of the reflector. Multiple reflections are taken into account. The parameters that define the reflector shape are optimized in an iterative procedure in order that the resulting light distribution is as close as possible to the user-provided target light distribution. We show that our method can calculate the reflector lighting at least one order of magnitude faster, even with millions of rays, and complex geometries and light sources.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism, I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling - Physically based modeling, I.3.1 [Hardware architecture]: Graphics processors

1. Introduction

This paper presents a new method for a GPU-based computation of outgoing light distribution for inverse reflector design. The manufacturers usually need to produce a desired illumination, but they do not know which the shape of the reflector must be. The usual solution is an iterative process, where a set of reflectors are manufactured and tested. This process is usually carried out in a very empiric way by experienced users that follow a trial and error procedure. This has a high manufacturing cost, both in materials and time.

In recent years, some research has been done in this field. Some works propose local lighting solutions, defining a very restricted set of possible reflectors, such as parabolic reflector families. Other solutions are based on global lighting simulation, but they demand high computational costs, spending hours or days to compute a reflector that produces an illumination distribution reasonably close to the desired one. However, these algorithms are not able to work with complex real world reflector shapes.

We propose a method that computes, from a family of possible reflectors, the best approximation to a given desired illumination distribution. A very fast GPU algorithm to calculate the reflected rays on the reflector is used to speed up an optimization process. We are able to compute reflector outgoing lighting distribution using millions of rays and highly complex reflector shapes in a couple of seconds. The set of reflectors is generated using a parameterizable basis. These parameters are optimized in an iterative process until the best solution is reached.

The rest of the paper is organized as follows. We discuss the previous work in Section 2. We present an overview of our method in Section 3, we present the fast reflection method in Section 4, and in Section 5 the optimization method is explained. Then we show the results in Section 6 and discuss them in Section 7. Conclusions are exposed in Section 8.

2. Previous Work

Our method is based in two main research topics: inverse reflector design and ray tracing on the GPU.

The first problem to solve in this paper can be put in the context of inverse illumination problems, where we know the desired illumination, and we have to compute some of the parameters that produce it. In this case, we have to find the reflector shape that produces the target lighting distribution. This kind of problem can be classified as an inverse geometry problem (IGP) [PP05]. To solve the IGP numerical

© The Eurographics Association 2008.

delivered by EUROGRAPHICS DIGITAL LIBRARY www.eg.org diglib.eg.org

problems, we can use local illumination or global illumination methods. In [CKO99] is used a combination of parabolic reflectors to compute the local illumination. Unfortunately, this method is useful only for really simple configurations. In [PPV04] and [PPV07] it is presented a method that uses global illumination. It starts from an initial reflector mesh and moves the mesh vertices in an iterative process, until the generated lighting distribution is close enough to the desired one. The main disadvantage of this method is the high computational cost, that depends on the number of tested reflectors, the reflector mesh resolution and the number of traced rays in lighting computation. To improve the method we need to calculate in a fast way the ray tracing of millions of rays on a high complexity reflector shape.

There are several GPU methods for calculate the ray tracing. On the one hand, we do not have a complex generic scene, so we do not need full ray tracing engines [CHH02], or environment mapping techniques [UPSK07]. On the other hand, acceleration methods based on space partitioning are more interesting in our case, because we can store the reflector geometry into a hierarchical subdivision structure. Several methods have been proposed to traverse the rays on this kind of structures. A fast algorithm is presented in [RnL00], where the geometry space is subdivided into an octree. This is a top-down parametric algorithm, where the voxel selection is done with simple comparisons with ray parameters. However, this is a CPU based algorithm, and the implementation in GPU would imply the use of a stack for each fragment.

Other GPU approaches in hierarchical structures are presented in [SKU08]. In this case, some techniques are presented to calculate the displacement mapping, where the displacement textures are transformed into hierarchical structures. Related to them, there is the Quadtree Relief Mapping technique [SvG06], based on Relief Mapping [POJ05]. Relief mapping is a tangent space technique that tries to find the first intersection of a ray with a height filed by walking along the ray in linear steps, until a position is found that lies beneath the surface. Then a binary search is used to precisely locate the intersection point. Quadtree Relief Mapping is a variation that takes large steps along the ray without overshooting the surface. This is achieved through the use of a quadtre on a height map. This will be described in more detail in Section 4.2.

3. Overview

The goal of our method is to obtain a reflector shape that produces a minimum error between the desired and resulting light distributions.

The method has two components. First, we present a fast algorithm to calculate the outgoing lighting distribution from a given reflector. Second, we optimize a set of possible reflectors, obtaining the one that minimizes an error metric.

The input data are the light source, the desired outgoing lighting distribution, and a parametric reflector space. The light source is represented by a set of rays, each composed by an origin and a direction (rayset). The desired outgoing lighting distributions used in this paper are far-field representations, which are lighting distributions measured far enough from the reflector. So, only directional distribution of light matters. However, our algorithm can deal with more complex representations (e.g. near-field) as well.

The reflector lighting calculation has several steps. The first one transforms the reflector geometry into a hierarchical height field, in order to efficiently trace rays in the GPU. This structure is stored into the GPU as a mip-map floating point texture that represents a quadtree, where each node contains its child nodes maximum height. In the second step, the set of rays is traced through the height field, searching for intersection with the reflector. Also, the algorithm considers multiple ray bounces (specular BRDF) inside the reflector. The third step captures all reflected rays and creates a far-field distribution that is compared with desired far-field, and an error value is generated. Note that once the light rays leave the light source, further collisions with it are ignored.

The overall algorithm is implemented using GPU shaders, where each GPU fragment processes a light ray. This results in a very fast algorithm that is able, even for millions of rays and complex reflector geometry shapes, to calculate the reflector lighting in less than 3 seconds, as shown in section 6.

The optimization step searches a set of possible reflectors in an iterative process, where each reflector parameter is optimized between a minimum and maximum value. Then, for each reflector, a far-field light distribution is generated and compared with the desired lighting distribution. After testing all possible reflectors, the best one is choosen.

4. Reflector lighting

The calculation of reflector lighting distribution has four steps. The first is the preprocessing of the input data. The second one transforms the reflector geometry to a hierarchical height field representation. The third step calculates the ray reflections over the reflector. Finally, the results are compared with the desired illumination.

4.1. Preprocessing of the input data

The user-provided data is composed by the desired far-field illumination specification, the light source characteristics and the reflector holder dimensions corresponding to the reflector parametric space. This far-field is given by an IES specification. This specification is established as an industry standard (IESNA [ANS02], EULUMDAT [bCL99]), and assumes large distances from the sources to the lighting environment, so spatial information in the emission of the light can be neglected, considering it as a point light source with a non-uniform directional distribution emittance model. The provided far-field only takes into account the reflected vectors from the desired reflector. The light source specification provides the light source position and dimensions, and the near-field emittance description. Finally, the reflector holder is used to fit the reflector shape into a bounding box.

In this preprocessing step, a rayset is extracted from the light source. Next, we discard the rays that we are sure that do not intersect with the holder bounding box. The non-discarded rays are stored into two textures, one for ray directions and another one for ray origin positions.

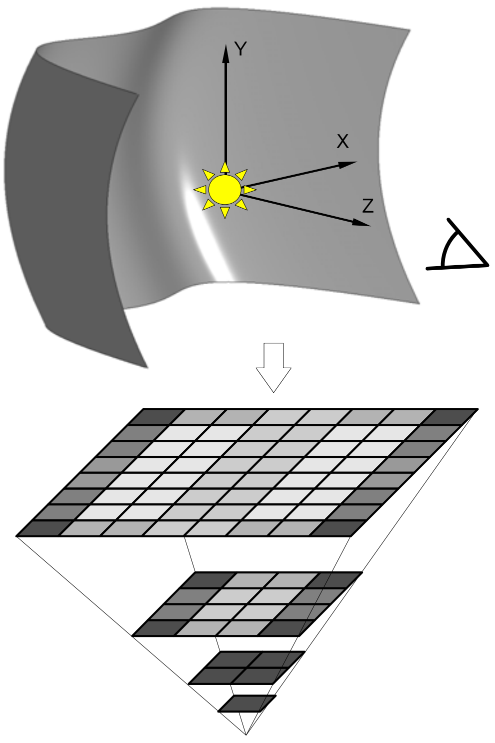


Figure 1: *The reflector mip-map height texture is constructed from the z-buffer, using a view point where all the reflector geometry is visible. Darker texel colors mean higher heights.*

4.2. Reflector geometry transformation

At this step, we need to construct a hierarchical height-field representation of the reflector. The structure used is a quadtree represented by a mip-map height texture. Each quadtree node contains the maximum height of its child nodes (see Figure 1).

As is said before, the method does not depend on reflector geometry complexity. The only restriction is that the reflector must be able to be manufactured with a press-forming process, where the reflector shape is deformed only in the vertical direction. More precisely, the shape must satisfy certain constructive constraints that amount to requiring that the shape of the reflector be the graph of a function defined on a subset of the plane delimited by the reflector's border. That is, in our formulation, for the shape to be "build-able", it must be a function of type $z = f(x,y)$.

We calculate one viewpoint to the reflector from which all reflector geometry is visible. The view direction can be used as the pressing direction. For our experiments, fitting the viewport to the reflector front is good enough.

When the viewport is specified, the reflector is rendered, and then the hardware z-buffer is read, considering the Z component as heights. Then, a GPU shader creates the mip-map texture, where the highest map level is a texture with one texel that contains the reflector maximum height.

Finally, another GPU shader extracts the reflector normal vectors, and stores them into a second texture. These normals will be used later to calculate the reflection vectors.

4.3. Reflectiors computation

The ray tracing on the reflector is based on Quadtree Relief Mapping method (QRM) [SvG06]. It is a variation of Relief Mapping tangent space technique [POJ05], which takes adaptive steps along the view rays in tangent space without overshooting the surface, due to the use of a quadtree on the height map. The goal is to advance a cursor position over the ray until we reach the lowest quadtree level, thus the intersection point is obtained.

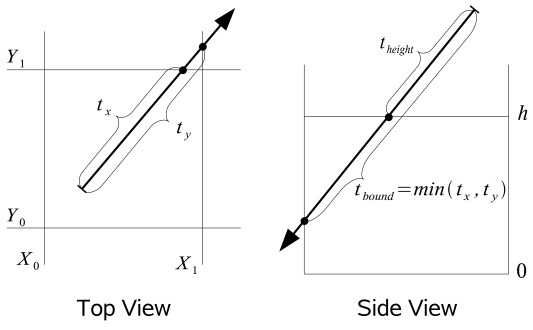


Figure 2: *Two ray steps are calculated for a quadtree node. At left, t_{bound} is the minimum displacement to quadtree node bounds t_x and t_y. At right, t_{height} is the displacement to stored node height h. The final selected step is the minimum between both.*

The method starts at highest quadtree level, where the root node has the maximum height. The ray cursor displacement at this point is $t_{cursor_0} = 0$. To advance the cursor, the

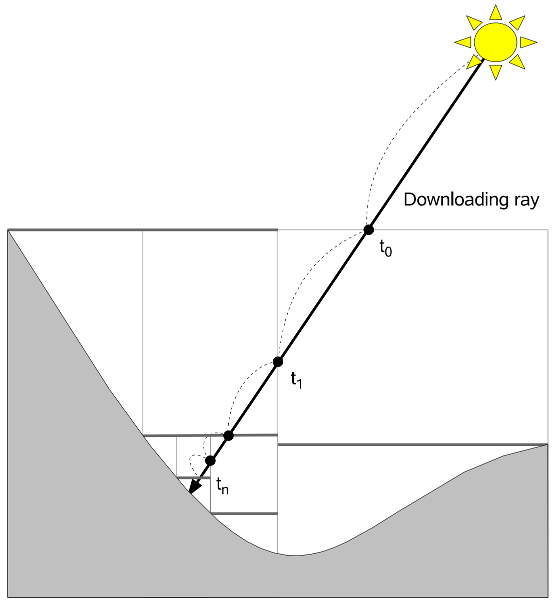


Figure 3: *Intersection search going down the quadtree hierarchy.*

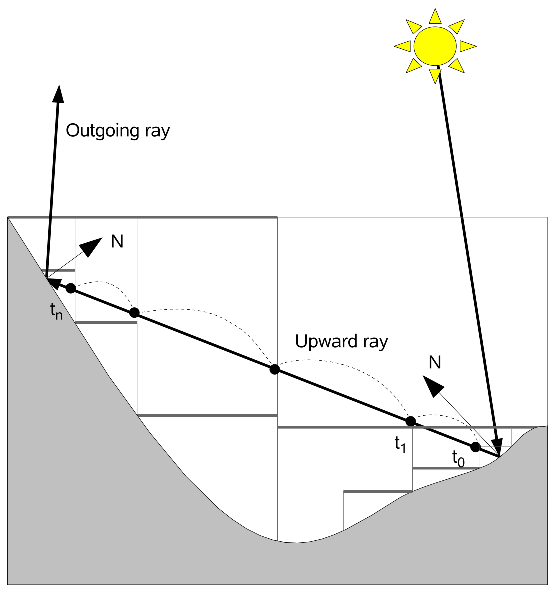


Figure 4: *Intersection search going up the quadtree hierarchy.*

ray is intersected with quadtree node bounds (see Fig. 2 left), and with the quadtree node stored height (see Fig. 2 right). There are two possible node bound intersections in tangent space: t_x and t_y. From them, we only use the nearest, called t_{bound}. Also, an intersection called t_{height} is obtained intersecting the ray with the height value stored in node. If t_{bound} is greater than t_{height}, means that the ray intersects with the current quadtree cell. So, the quadtree level is decreased, and the process starts again with one of the four child nodes. In this case, the cursor does not advance, so $t_{cursor_{i+1}} = t_{cursor_i}$. Otherwise, the cursor advances to the cell bound, $t_{cursor_{i+1}} = t_{bound}$, and the process starts again with the neighbour cell. This process stops when the minimum quadtree level is reached, or when the cursor position is out of texture bounds. In Figure 3 there is an example of this algorithm.

In the QRM algorithm, the first cursor position is found intersecting the view ray with the geometry bounding box. In our case, the first cursor position is the light ray origin (see Figure 3). This means that one more step is processed in comparision with QRM, because we need to intersect the root quadtree node in an initial step. However, we avoid the ray-bounding box intersection calculations that QRM performs.

On the other hand, QRM only process rays going down the quadtree hierarchy, being unable to process the going up rays. This is the case when the light source is inside reflector, or when more than one ray bounce inside the reflector is considered. We propose an intersection search algorithm going up the quadtree hierarchy. The original algorithm asumes that the cursor advances always on the opposite direction of the height map direction. Otherwise, QRM discards the ray because it does not intersect with the surface. To solve this case, we start the algorithm from the highest quadtree level using the new ray, composed by current intersection point and reflection direction. A small offset is applied as initial cursor displacement to avoid the self-intersection, thus $t_{cursor_0} = \varepsilon$. Then, we go down through the quadtree until $t_{cursor_i} > t_{height}$, which means the current cursor position height is over current node height. Now, we are sure that there is not any node under the current one that has a height that intersects with the ray. Hence, we jump to the neighbour node, so $t_{cursor_{i+1}} = t_{bound}$, and increase the quadtree level. If $t_{cursor_i} < t_{height}$ then there is not any possible intersection under current level. Thus, we decrease the current quadtree level, and do not update t_{cursor_i}. The process stops when the intersection is reached, or when the cursor position falls out of texture bounds. In the second case, it is a reflected ray with no more bounces, and it does not have to be discarded. In Figure 4 there is an example of this algorithm.

The algorithm is implemented into a GPU fragment shader. The rayset data is provided by previously stored rayset textures. The textures are mapped into a quad, so each ray corresponds to a fragment. Each fragment runs an iterative process that ends with an intersection point and a reflection vector. These values are stored into two output textures. The first texture stores the intersection position and a bounce counter. The second texture stores the reflected direction and a control code. This control code is used to identify when a ray falls out of the texture bounds. In this case, if bounce counter is 0, it means that the ray must be discarded. Otherwise, the ray does not need more bounces. This shader is executed as many times as the maximum number of allowed bounces. The resulting textures are used as input textures for the next excecution, thus a GPU ping-pong approach is used.

4.4. Comparision to a desired distribution

At this step we compare the obtained light distribution with the target one. Both distributions are converted to far-fields to be compared in the same domain (see Fig. 5). .

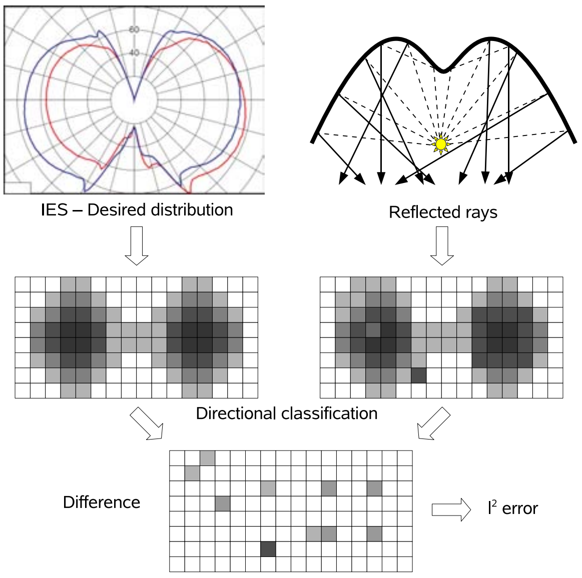


Figure 5: *Both desired distribution and reflected rays are classified into histograms. Next, the histograms are compared using the l^2 metric*

To convert reflected rays to a far-field distribution, a regular grid is used to classify the ray directions. Each grid cell represents a pair of azimuth and altitude vector directions in horizontal coordinate system, and contains the number of rays in this direction. The azimuth and altitude ranges are $[-\pi...\pi]$ and $[\pi/2... - \pi/2]$ respectively. The grid size depends on the specified far-field directional space discretization. We use two textures to store both grids, where each texel represents a grid cell.

We classify the reflected directions calculating a 3D histogram, where each interval represents a grid cell. The algorithm, based on [SH07], has two steps: First, after the last reflection step the results are stored into a vertex buffer object. Next, this vertex buffer is rendered, and a vertex shader classify de directions calculating the fragment coordinates for each reflected direction. Then, the number 1 is sent to all fragments. If the hardware blending is activated, the result is a counter for each fragment.

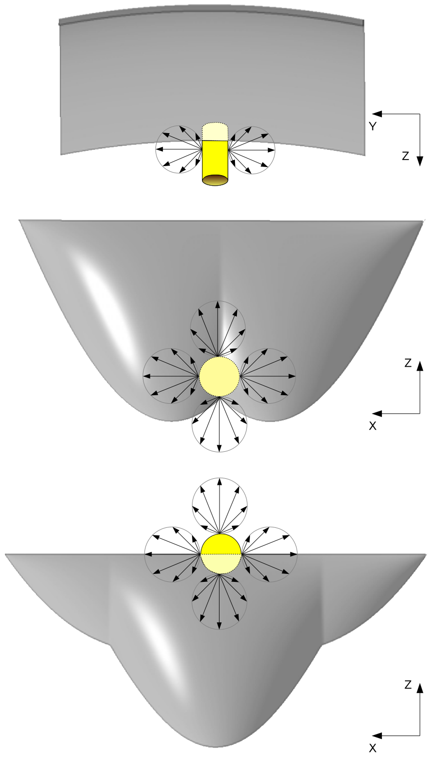


Figure 6: *Cross section views of reflectors and their associated light sources used to test our method.*

We use the same algorithm to classify the target distribution. In this case we don't have to use a counter, because each far-field directional component has the respective emitted energy. To use the same measurement units, both the number of reflected rays and energy (usually in candelas) for each cell, are transformed to lumens.

The comparision between both textures is done with a shader that calculates for each fragment the l^2 error metric:

$$D_{l^2}(a,b) = \sqrt{\sum_i^N (a_i - b_i)^2}$$

In addition, a reduction shader is used to calculate the summation part of the formula.

5. Optimization

To obtain a reflector shape that produces a light distribution close to the desired one, we optimize the parameters used in the parametric reflector shape definition. For each opmization step, a new reflector shape is obtained, and the outgoing light distribution is compared with the target distribution. If

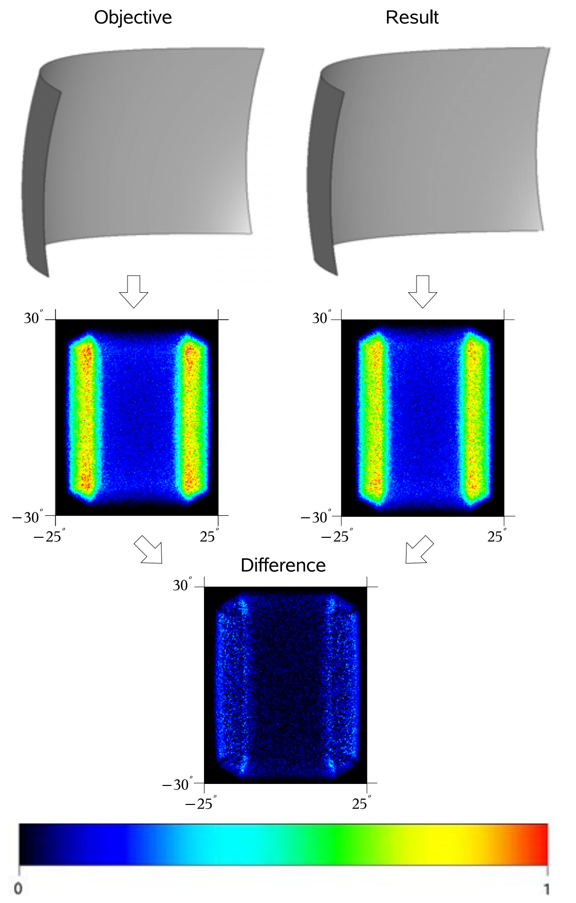


Figure 7: *Results for our Model A. At top, the desired and obtained reflectors. At middle, the desired and obtained far-field histograms, indicating the respective angle domains. At bottom, the histogram difference between both*

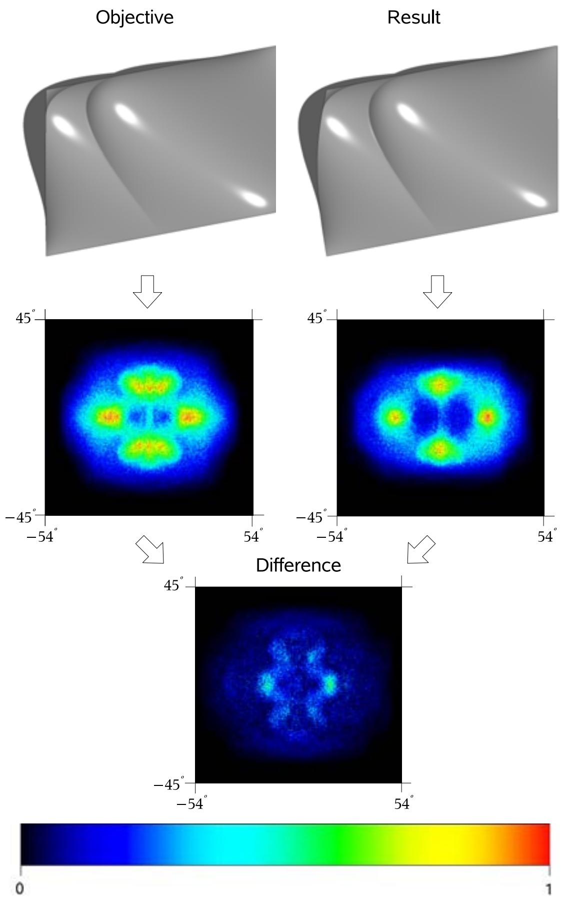


Figure 8: *Results for our Model B. At top, the desired and obtained reflectors. At middle, the desired and obtained far-field histograms, indicating the respective angle domains. At bottom, the histogram difference between both*

difference value is under a user-specified threshold, the process stops. If no reflector produces a light distribution close enough to the objective, the best one is choosen.

We use a standard optimization method, where for each parameter, a range and a constant step are specified. The algorithm is an iterative process that, for each parameter it is increased inside its range by its offset value [PPV04]. The mip-map height texture must be regenerated at each iteration, due to the reflector geometry changes. Hence, for each iteration we have to recalculate the outgoing light distribution. However we do not have to recalculate the rayset for each reflector, so the initial ray intersection step on the reflector bounding box assures us that the rayset is valid for any reflector inside this box.

6. Results

We have tested our method with three cases. The first one, called *Model A*, uses a cylindrical light source with a cosinus emittance along his surface, except for the caps, that do not emit light. The cylinder dimensions are 4.1mm length and 0.65mm radius. It is placed at (0,0,0), inside a holder bounding box located between (-30, -20, -20) and (30, 20, 0), also in mm. The second one, called *Model B*, uses a spherical light source with a cosinus emittance. Its dimensions are 0.5mm of radius, and it is placed at (5, -5, -5), inside a holder bounding box located between (-10, 0, -6) and (10, 0, 0). The third one, called *Model C*, uses a spherical light source with a cosinus emittance. Its dimensions are 1mm of radius, and it is placed at (5, 5, 0), inside a holder bounding box located between (0, 10, -6) and (0, 10, 0). The cross section of three cases, and light source relative positions, are shown Figure 6. For models A and C, the light sources emit 10 millions of

Model	Effective rays	Max. bounces	Reflector lighting mean time (sec.)	Optimization time (hours)	Tested Reflectors	Optimized parameters	Best l^2 error
A	$7.38x10^6$	1	1.3	0.63	1728	3	0.599456
B	$5x10^6$	5	3.2	2.2	2401	4	0.975587
C	$6.05x10^6$	6	2.7	4.9	6561	4	0.245821

Table 1: *Results for our three configurations: From left to right, the left column is the number of traced rays, maximum number of bounces inside the reflector, mean time of reflector lighting computation, total time of optimization, number of tested reflectors, number of optimized parameters and resulting error*

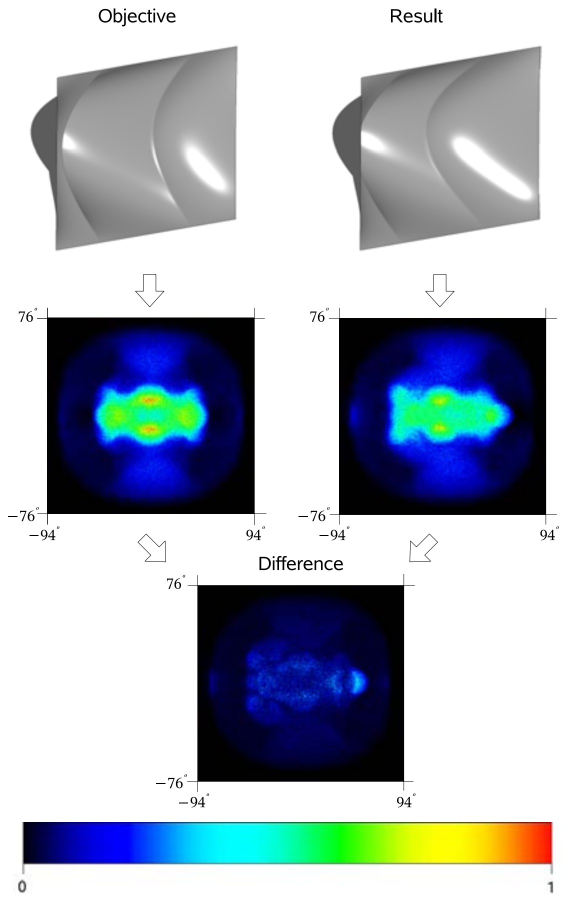


Figure 9: *Results for our Model C. At top, the desired and obtained reflectors. At middle, the desired and obtained far-field histograms, indicating the respective angle domains. At bottom, the histogram difference between both*

rays, and 5 millions of rays for model B. All of them have an overall energy of 1100 lumens. Also for all cases, the mip-map height texture resolution is 1200×800, thus a quadtree is created with 9 levels of subdivision.

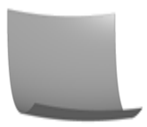
The optimization results for each case are shown in Figures 7, 8 and 9. The desired and obtained reflectors are shown, with the respective far-field distributions and difference images. Both far-field and difference images are represented by false-color histograms. These histograms are defined like far-field textures, thus the columns of the texture grid correspond to horizontal angles, and the rows correspond to vertical angles. The directional space resolution is 1800×900 for horizontal and vertical angles range respectively. Thererfore, each histogram cell represents an angle range of 0.2×0.2 degrees. The color scale represents the amount of energy for each histogram cell.

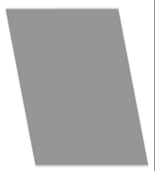
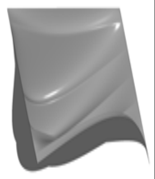
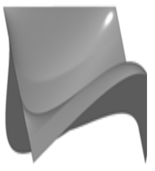
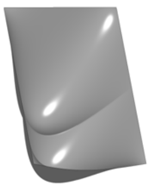
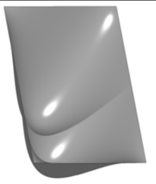
In table 1 there is a summary of data for the overall inverse reflector searching process for each model. The number of effective rays means the number of non-discarded rays from the initial rayset. For *Model B* there are not any discarded rays, because the light source is inside the reflector bounding box, and all rays intersect the height map. The time needed to compute the reflector lighting depends on the number of effective rays and number of maximum allowed bounces. Since all models have a similar number of effective rays, the *Model A* has the lower reflector lighting computation time because only one bounce is specified. The optimization time depends on reflector lighting time and the number of tested reflectors, and the number of tested reflectors depends on the number of optimizable parameters and on the range and offsets applied in the optimization procedure,

Model	Heigh map construction	Intersection search	Error calculation
A	56	976	277
B	34	2963	278
C	86	2406	263

Table 2: *Mean times (in miliseconds) broken down into the three main algorithm sections.*

In table 2 there is a summary of the broken down times for each reflector lighting step. The height map creation times are similar because of all models use the same mip-map height texture resolution. The intersection search time depends on the number of traced rays, on the maximum number of allowed bounces, and on the height map levels.

Model A							
# Step	0	20	56	75	182	298	Desired
l² error	2.63456	1.95119	1.90016	1.32525	0.907986	0.549456	

Model B							
# Step	0	362	429	767	1459	2257	Desired
l² error	6.0011	2.56208	2.52433	2.27985	1.27989	0.975587	

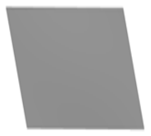
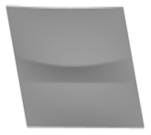
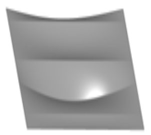
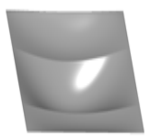
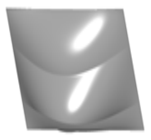
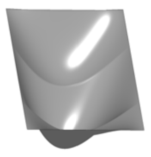
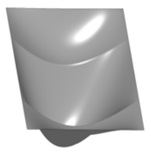
Model C							
# Step	0	1636	2446	3859	3886	3891	Desired
l² error	1.7105	0.980502	0.802349	0.485431	0.411784	0.245821	

Figure 10: *Reflector searching progress, from an initial shape (left), to desired one (right). From top to bottom, model A, B and C. Below each reflector, there are the current number of steps in the optimization process and the l^2 error*

However, the results are very similar between the models, because they have the same height map texture sizes (thus, the same number of quadtree levels), and the number of traced rays is similar between them. The GPU parallel processing allows us to get a non-linear computational cost on rayset size. Therefore, the maximum number of allowed bounces is the most important factor in the intersection search procedure. Finally, the calcualtion of the error has similar times for all cases, since the outgoing textures have the same size.

7. Discussion

As is shown in previous section, we never obtain the desired reflector with 0 error. This is because the optimization algorithm tests different parametrized reflectors changing the parameters values in a constant step size and in a floating space. On the other hand, we can improve the results optimizing at very small steps, also guaranteeing conergence to a more usable solution, but this would affect strongly the processing times.

The most time consuming part of our method is the intersection search algorithm. If we use a very refined height map, we need more time to traverse the ray through the quadtree. If we want to manage very complex reflector shapes, we need height maps with high resolutions. Therefore, we need to achieve a comprimise between time costs and quality of results.

8. Conclusions and Future Work

We have presented a method for the inverse reflector design problem that improves on previous approaches. From a wide set of parametrized reflectors, the best approximation to a given desired illumination distribution is found. The method is based on a very fast GPU algorithm that calculates the reflected rays on the reflector (with one or more bounces) in 2 to 3 seconds, even using millions of rays and highly complex reflector shapes. The reflector parameters are optimized in an interative process until the generated light distribution is close enough to the desired one.

We consider, as future work, the use of better optimization methods to reach the solution in a non-constant step size, thus the desired reflector can be obtained faster, e.g. using adaptive methods. Another future research line is the optimization based on combination of predefined complex reflector shapes stored as texture masks.

9. Acknowledgments

This work was done under grant TIN2007-67120 from the Spanish Government.

References

[ANS02] ANSI/IESNA: Lm-63-02. ansi approved standard file format for electronic transfer of photometric data and related information, 2002.

[bCL99] BYHEART CONSULTANTS LIMITED: Eulumdat file format specification, 1999. http://www.helios32.com/Eulumdat.htm.

[CHH02] CARR N. A., HALL J. D., HART J. C.: The ray engine. In *HWWS '02: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (Aire-la-Ville, Switzerland, Switzerland, 2002), Eurographics Association, pp. 37–46.

[CKO99] CAFFARELLI L. A., KOCHENGIN S. A., OLIKER V. I.: On the numerical solution of the problem of reflector design with given far-flield scattering data. *Contemporary Methematics 226* (1999).

[POJ05] POLICARPO F., OLIVEIRA M. M., JO A. L. D. C.: Real-time relief mapping on arbitrary polygonal surfaces. *ACM Trans. Graph. 24*, 3 (2005), 935–935.

[PP05] PATOW G., PUEYO X.: A survey of inverse surface design from light transport behaviour specification. *Computer Graphics Forum 24*, 4 (2005), 773–789.

[PPV04] PATOW G., PUEYO X., VINACUA A.: Reflector design from radiance distributions. *International Journal of Shape Modelling 10*, 2 (2004), 211–235.

[PPV07] PATOW G., PUEYO X., VINACUA A.: User-guided inverse reflector design. *Comput. Graph. 31*, 3 (2007), 501–515.

[RnL00] REVELLES J., NA C. U., LASTRA M.: An efficient parametric algorithm for octree traversal. In *Proc. WSCG* (2000), pp. 212–219.

[SH07] SCHEUERMANN T., HENSLEY J.: Efficient histogram generation using scattering on gpus. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games* (New York, NY, USA, 2007), ACM, pp. 33–37.

[SKU08] SZIRMAY-KALOS L., UMENHOFFER T.: Displacement mapping on the GPU - State of the Art. *Computer Graphics Forum 27*, 1 (2008).

[SvG06] SCHRODERS M. F. A., V. GULIK R.: Quadtree relief mapping. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/Eurographics symposium on Graphics hardware* (New York, NY, USA, 2006), ACM, pp. 61–66.

[UPSK07] UMENHOFFER T., PATOW G., SZIRMAY-KALOS L.: *GPU Gems 3*. GPU Gems 3. Addison-Wesley, 2007, ch. Robust Multiple Specular Reflections and Refractions, pp. 387–407.