Volume 0 (1981), Number 0 pp. 1–12 COMPUTER GRAPHICS *forum*

Fast Inverse Reflector Design (FIRD)

Albert Mas, Ignacio Martín and Gustavo Patow

IIiA, UdG, Girona, Spain

Abstract

This paper presents a new inverse reflector design method using a GPU-based computation of outgoing light distribution from reflectors. We propose a fast method to obtain the outgoing light distribution of a parameterized reflector, and then compare it with the desired illumination. The new method 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 for the resulting light distribution to be as close as possible to the desired, user-provided one. We show that our method can calculate reflector lighting at least one order of magnitude faster than previous methods, even with millions of rays, 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. When manufacturers need to produce a desired illumination, they sometimes do not know what shape 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 empirical 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, or define a very restricted set of possible reflectors, such as the families of parabolic reflector. Other solutions are based on global lighting simulation, but they have high computational costs, requiring 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 the optimization process. We are able to compute reflector outgoing light 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 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 explain the optimization method in Section 5. Then we show the results in Section 6 and discuss them in Section 7. Conclusions are presented in Section 8.

2. Previous Work

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

The first problem to solve in this paper can be explained 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 desired light distribution. This kind of problem can be classified as an inverse geometry problem (IGP) [PP05]. To solve the IGP numerical problems, we can use local or global illumination methods. In [CKO99] a combination of parabolic reflectors is used to compute the local illumination. In [Neu97] a simple spline combined with local illumination is used to perform a local optimization. Unfortunately, these methods are useful only for really simple configurations. In [PPV04] and [PPV07] a method that uses global illumination is presented. It starts from an initial reflector mesh and moves the mesh vertices in an iterative process, until the generated light distribution is close enough to the desired one. The main disadvantage of this method is the high computational cost, which depends on the number of tested reflectors, the reflector mesh resolution and the number of rays traced for lighting computation. To improve the method we need to calculate the ray tracing of millions of rays on a highly complex reflector shape in a fast way.

Whe can employ several GPU methods to calculate ray tracings. On one hand, we do not have a complex generic scene, so we do not need a full engine [CHH02] or approximated ray tracing techniques [UPSK07]. On the other, 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 through this kind of structure. A fast algorithm is presented in [RUL00], where the geometry space is subdivided into an octree. This is a top-down algorithm where the voxel is selected according to ray parameters in tangent space. However, this is a CPU based algorithm, and its implementation in GPU would require the use of a stack for each fragment.

Other GPU approaches in hierarchical structures are presented in [SKU08], where some techniques are presented to calculate displacement mapping, and the displacement textures are transformed into hierarchical structures. Related to them, there is the quadtree relief mapping technique [SG06], 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 field by walking along the ray in linear steps, until a position is found that lies beneath the surface. Then, a binary search is conducted to precisely locate the intersection point. Quadtree relief mapping is a variation that takes larger steps along the ray without overshooting the surface. This is achieved through the use of a quadtree 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 light distribution from a given reflector. Second, we optimize a set of possible reflectors, obtaining the one that minimizes the error metric.

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

Reflector light calculation occurs in three 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 in the GPU as a mip-map of floating point textures that represents a quadtree, where each node contains the maximum height of its child nodes. In the second step, the set of rays is traced through the height field, searching for intersections with the reflector. The algorithm also 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 the 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 procedure searches for a minimum error among a set of possible reflectors in an iterative process, where each reflector parameter is optimized between two bounding values. Then, for each reflector, a far-field light distribution is generated and compared with the desired light distribution. After testing all possible reflectors, the one which provides the most similar light distribution to the desired one is chosen.

4. Reflector lighting

Reflector light distribution can be calculated in three steps. After the input data is preprocessed, the first step transforms the reflector geometry to a hierarchical height field model. The second one calculates the ray reflections over the reflector. Finally, the results are compared with the desired illumination. Figure 1 shows the general scheme of reflector light calculation.

4.1. Preprocessing of the input data

The user-provided data is composed of the desired far-field illumination specification, the light source characteristics and the reflector holder dimensions. The far-field is given by an IES specification. This specification is established

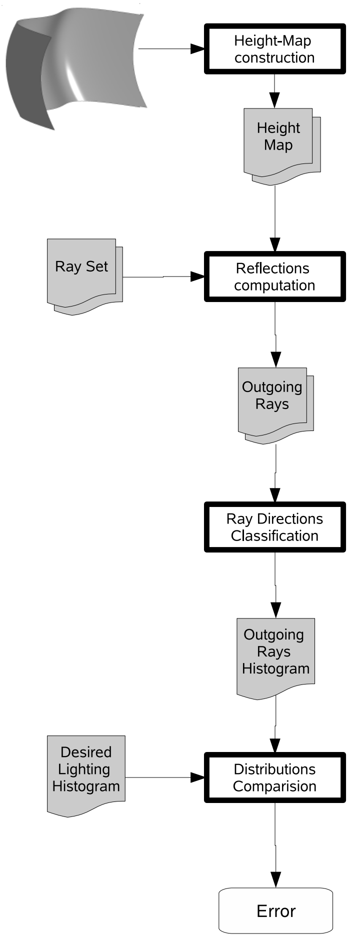


Figure 1: *Overall scheme of reflector lighting pipeline and the used shaders and textures.*

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 vectors reflected from the desired reflector, discarding the direct rays from the light source. 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 obtained from the light source. Next, we discard the rays we are sure will not intersect with the reflector bounding box. The non-discarded rays are stored in two textures, one for ray directions and another for ray origin positions.

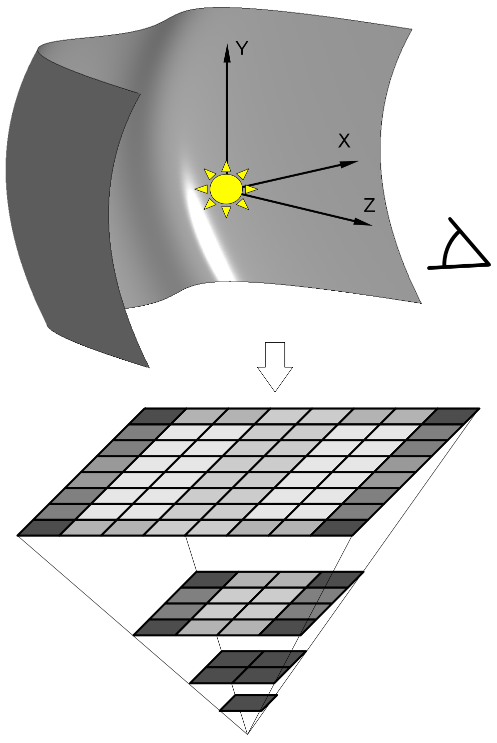


Figure 2: *The reflector mip-map height texture is constructed from the z-buffer, using a view point where all the reflector geometry is visible. Darker texel colours mean greater heights.*

4.2. Reflector geometry transformation

At this stage, we need to construct a representation of the hierarchical height-field 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 2).

As mentioned previously, the method does not depend on reflector geometry complexity. The only restriction is that the reflector must be able to be manufactured through 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 require the shape of the reflector to 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 orthogonal projection view from which all reflector geometry is visible. The view direction can be used as the pressing direction, so in our case, the Z axis matches the press-forming vertical direction. For our experiments, just 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 maximum reflector height.

To avoid an excessive number of texels representing the background, we fit the mip-map texture into a tight bounding rectangle around the reflector. Therefore the mip-map texture is non-power-of-two size, which means the number of height-field levels will depend on the largest texture dimension.

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

4.3. Reflections computation

Ray tracing on the reflector is based on quadtree relief mapping method (QRM) [SG06], which it is a variation of relief rapping in tangent space [POJ05]. QRM 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, thereby obtaining the intersection point.

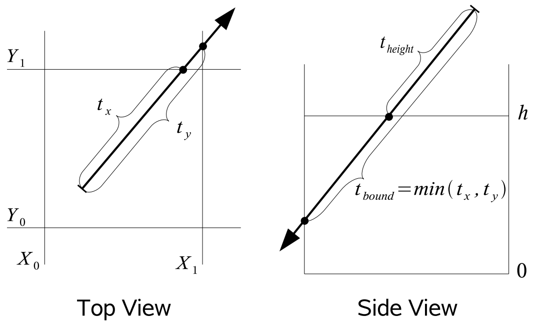


Figure 3: *Two ray steps are calculated for a quadtree node. At the left, t_{bound} is the minimum displacement to quadtree node bounds t_x and t_y. At the right, t_{height} is the displacement to the stored node height h. The final selected step is the minimum between both.*

The method starts at the 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 ray is intersected with the quadtree node bounds (see Fig. 3, left), and with the stored quadtree node height (see Fig. 3 right). There are two possible node bound intersections in tangent space: t_x and t_y. From them, we use the nearest one,

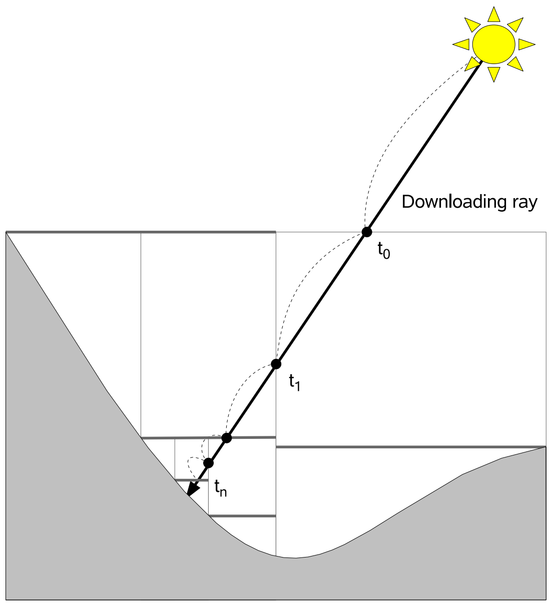


Figure 4: *Intersection search going down the quadtree hierarchy.*

called t_{bound}. Also, an intersection called t_{height} is obtained by intersecting the ray with the height value stored in the node. If t_{bound} is greater than t_{height}, it 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 the texture bounds. In Figure 4 there is an example of this algorithm.

In the QRM algorithm, the first cursor position is found by intersecting the view ray with the geometry bounding box. In our case, the first cursor position is the light ray origin (see Figure 4). This means that one more step is processed in comparison with QRM, because we need to intersect the root quadtree node in an initial step. However, we avoid the ray-bounding box intersection calculation that QRM performs.

On the other hand, QRM only processes rays going down the quadtree hierarchy, being unable to process the rays going up. This is the case when the light source is inside the reflector, or when more than one ray bounce inside the reflector are considered. We propose an intersection search algorithm going up the quadtree hierarchy. The pseudo-code for the new algorithm, called RQRM, is presented in Algorithm 1.

The original algorithm assumes that the cursor always ad-

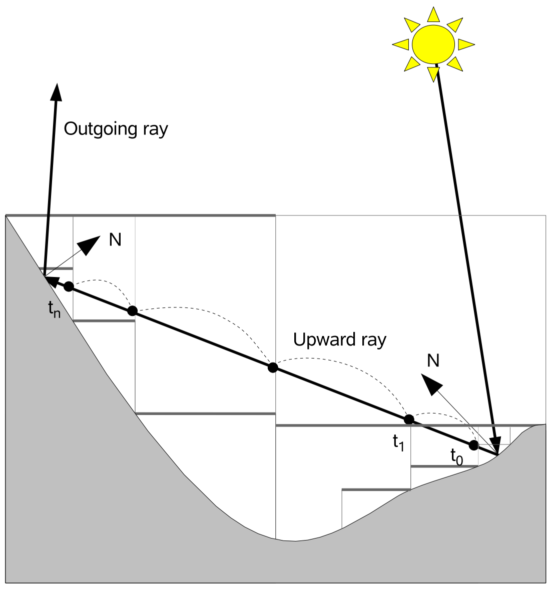


Figure 5: *Intersection search going up the quadtree hierarchy.*

Algorithm 1 $RQRM(texCoord)$

```
1:  $RQRMInitialization(texCoord)$
2:  while $level \leq log(max(reliefMapSize_{xy}))$ do
3:      $RQRMCalculateTangentSpaceBounds$
4:      $RQRMStep$
5:      if $OutOfLimits(cursor)$ then
6:          if $FirstBounce$ then
7:              return $DISCARDED$
8:          else
9:              return $FINISHED$
10:         end if
11:     end if
12: end while
13: $finalPos \leftarrow reflectorTex[cursor]$
14: $finalNormal \leftarrow reflectorNormTex[cursor]$
15: $reflectRay \leftarrow reflect(rayDir, finalNormal)$
16: return$(finalPos, reflectRay)$
```

vances in the opposite direction to the height map direction. Otherwise, QRM discards the ray because it does not intersect with the surface. To solve this case for reflections, we start the algorithm from the highest quadtree level using the new ray, composed of the current intersection point and reflection direction. A small offset is applied as initial cursor displacement to avoid self-intersections, thus $t_{cursor_0} = \delta$ (see lines 4-7 of Algorithm 2). Then, we go down through the quadtree until $t_{cursor_i} > t_{height}$ (see Algorithm 3 for tangent space bound calculations), which means the height of the current cursor position is above the current node height. Now, we are sure there are not any nodes under the current one that have 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} (see lines 8-14 of Algorithm 4). The process stops when the intersection is reached, or when the cursor position falls out of the texture bounds (see lines 6-10 of Algorithm 1). In the second case, it is a reflected ray with no more bounces, and it is stored as an outgoing ray. In Figure 5 there is an example of this algorithm.

Algorithm 2 $RQRMInitialization(texCoord)$

```
1:  $rayPos \leftarrow rayPosTex[texCoord]$
2:  $rayDir \leftarrow rayDirTex[texCoord]$
3:  $cursor \leftarrow ReflectorMapProjection(rayPos)$
4:  if $FirstBounce$ then
5:      $tcursor \leftarrow 0$
6:  else
7:      $tcursor \leftarrow \delta$
8:  end if
9:  $cursor \leftarrow cursor + rayDir \cdot tcursor$
10: $startPoint \leftarrow cursor$
11: $quadrant \leftarrow (sign(rayDir)+1)\, div\, 2$
12: $level \leftarrow 0$
```

Algorithm 3 $RQRMCalculateTangentSpaceBounds$

```
1: $bound \leftarrow \lfloor (cursor \cdot 2^{level}) + quadrant \rfloor$
2: $tbound \leftarrow \frac{\frac{bound}{2^{level}} - startPoint}{rayDir_{xy}}$
3: $tmin \leftarrow min(tbound_x, tbound_y)$
4: $height \leftarrow reliefMap[cursor, level]$
5: $heightNorm \leftarrow (height - rayPos_z) \cdot \alpha$
6: $t \leftarrow \frac{heightNorm}{rayDir_z}$
```

Algorithm 4 $RQRMStep$

```
1:  if $rayDir_z \leq 0$ then
2:      $tcursor \leftarrow max(tcursor, min(t, tmin+\delta))$
3:      $cursor \leftarrow startPoint + (rayDir_{xy} \cdot tcursor)$
4:      if $t < (tmin+\delta)$ then
5:          $level \leftarrow level + 1$
6:      end if
7:  else
8:      if $t > tcursor$ then
9:          $level \leftarrow level + 1$
10:     else
11:         $tcursor \leftarrow tmin + \delta$
12:         $cursor \leftarrow startPoint + (rayDir_{xy} \cdot tcursor)$
13:         $level \leftarrow level - 1$
14:     end if
15: end if
16: return $level$
```

The algorithm is implemented in a GPU fragment shader. The rayset data is provided by the previously stored rayset textures. The textures are mapped into a quad, so each ray corresponds to a fragment. Each fragment program runs an iterative process that ends with an intersection point and a reflection vector. These values are stored in two output textures, one for the intersection positions, and the another one for the reflected directions. 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 execution, thus a GPU ping-pong approach is used.

4.4. Comparison with a desired distribution

At this step we compare the obtained light distribution with the desired one. Both distributions are converted to far-field to be compared in the same domain (see Fig. 6).

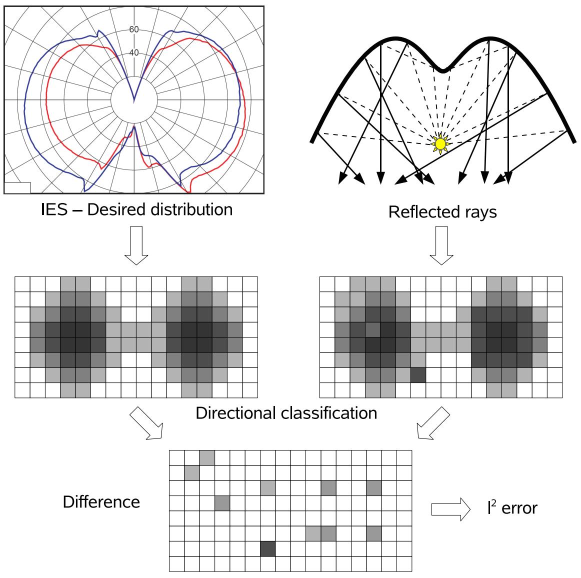


Figure 6: *Both the desired distribution and the reflected rays are classified into histograms. Next, the histograms are compared using the l^2 metric.*

To convert the 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 total 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 by calculating a 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 classifies the directions by calculating the fragment coordinates for each reflected direction. Then, the fragment shader gathers the directions using counters and the hardware blending.

We use the same algorithm to classify the desired distribution. In this case, we do not have to use a counter because each far-field directional component has its 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 comparison 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 optimization step, a new reflector shape is obtained, and the outgoing light distribution is compared with the desired one. If the difference value is below a user-specified threshold, the process stops. If no reflector produces a light distribution close enough to the objective, the best one is chosen. Figure 7 shows the overall scheme of the optimization algorithm.

We use a standard optimization method, where for each parameter, a range and a constant step are specified. The algorithm is an iterative process where each parameter is increased inside a given range by its step value [PPV04]. The mip-map height texture must be regenerated at each iteration, due to 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 guarantees that the rayset is valid for any reflector inside this box (see Section 4.1).

5.1. Method calibration

To test the accuracy of method, we performed a preprocessing step, where the lighting simulation was calculated several times using the desired reflector and combining different rayset sizes. For each test the light source was resampled. Table 1 presents the results for the *Model A* example (see Section 6), showing the means and variances of the l^2 errors from the difference between each result and the desired light distribution. For each rayset size, 100 lighting simulations have been calculated.

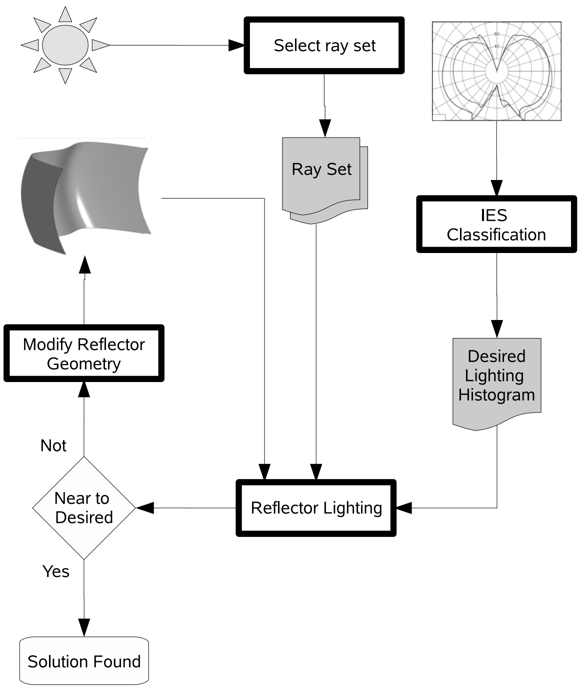


Figure 7: *Overall scheme of optimization algorithm.*

Rays	Mean l^2 Error	Variance l^2 Error
1000	271.68	15078.60
10000	27.97	20.97
100000	2.91	0.04
1000000	0.38	$6x10^{-4}$
10000000	0.13	$3x10^{-7}$

Table 1: *Results of several lighting simulations on the* Model A *using different rayset sizes.*

We observe that the variance error decreases when the rayset increases. On the rayset of 1 million rays, the mean error is quite good, so we can use this rayset to perform the optimizations. The last row shows the calibration values to consider the goodness of our method.

Moreover, we are interested in knowing the minimum optimization parameter step required to consider that two consecutive measures are different. For this, we use the semivariogram [Ole99], a statistical measure that assesses the average decrease in similarity between two random variables as the distance between the variables increases. The measure defines a lag called range, at which the semivariogram reaches a constant value. We can consider that two measures separated by a distance larger than the range are stochastically independent, so the range is equivalent to the notion of influence of an observation. That is, if we want to get significant measurements without being influenced by statistical noise problems, we can not take measurements that are closer than the range. From this, we can find a lower bound for the step size in the optimization process.

The semivariogram is defined as follows: Given two locations x and $x+h$ inside the domain of a random function Z, the semivariogram is:

$$\gamma(h) = \frac{1}{2n(h)}\sum_{i=1}^{n(h)}[Z(x_i+h)-Z(x_i)]^2$$

where $n(h)$ is the number of pairs of measurements at a distance h apart. Figure 8 presents the semivariogram for one of the parameters of *Model A*. From this graph we can see that the range of the semivariogram is about 0.1. So, we have to use values larger than 0.1 as the step size for this parameter in the optimization process. We computed this lower bound for every degree of freedom included in the optimization process.

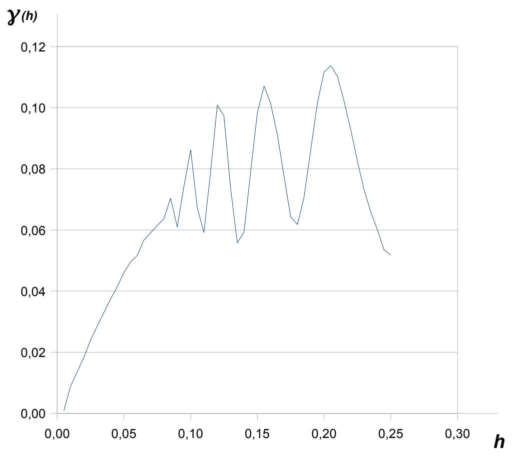


Figure 8: *Semivariogram when changing one parameter of* Model A *using* 10^6 *rays.*

6. Results

We have tested our method with three cases. The first one, called *Model A*, uses a cylindrical light source with a cosine emittance along its surface, except for the caps that do not emit light. The cylinder dimensions are 4.1mm in length and a 0.65 mm 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 case, called *Model B*, uses a spherical light source with a cosine emittance. It has a radius of 0.5mm, and it is placed at (5, -5, -5) inside a holder bounding box located between (0, -10, -6) and (10, 0, 0). The third one, called *Model C*, uses a spherical light source with a cosine emittance. It has a radius of 1mm, and it is placed at (5, 5, 0) inside a holder bounding box located between (0,

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 2: *Results for our three configurations: From left to right, we find 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.*

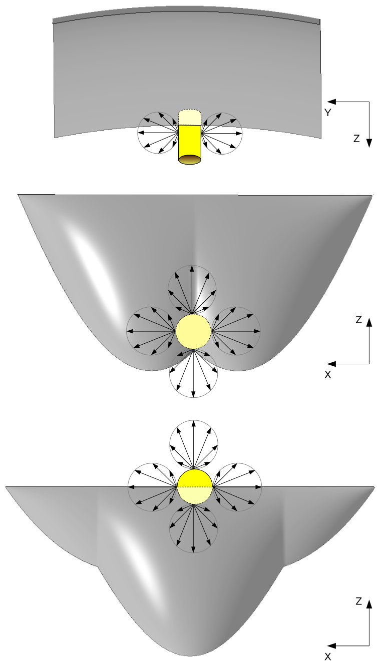


Figure 9: *Cross section views of reflectors and their associated light sources used to test our method.*

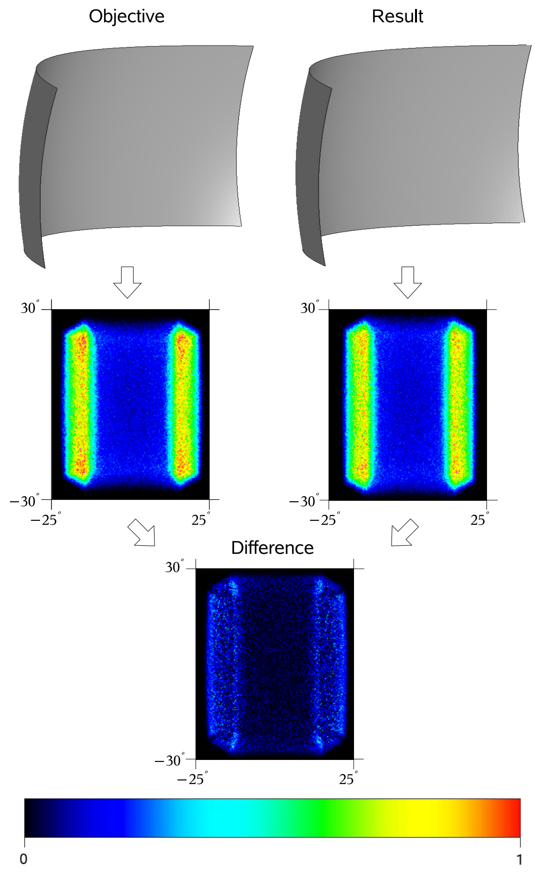


Figure 10: *Results for our Model A. At the top, the desired and obtained reflectors. In the middle, the desired and obtained far-field histograms in false-colour, indicating the respective angle domains. At the bottom, the histogram difference between both*

0, -6) and (10, 10, 0). The cross sections of the three cases and light source relative positions are shown in Figure 9. For *Models A* and *C*, the light sources emit 10 million rays, and 5 million 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, and a quadtree is created with 9 subdivision levels.

The optimization results for each case are shown in Figures 10, 11 and 12. The desired and obtained reflectors are shown, with the respective far-field distributions and difference images. In the figures, both far-field and difference images are represented by false-colour 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 respectively. Therefore, each histogram cell represents an angle of 0.2×0.2 degrees. The colour scale represents the amount of energy for each histogram cell.

Table 2 provides a summary of the data for the overall inverse reflector search process for each model. The num-

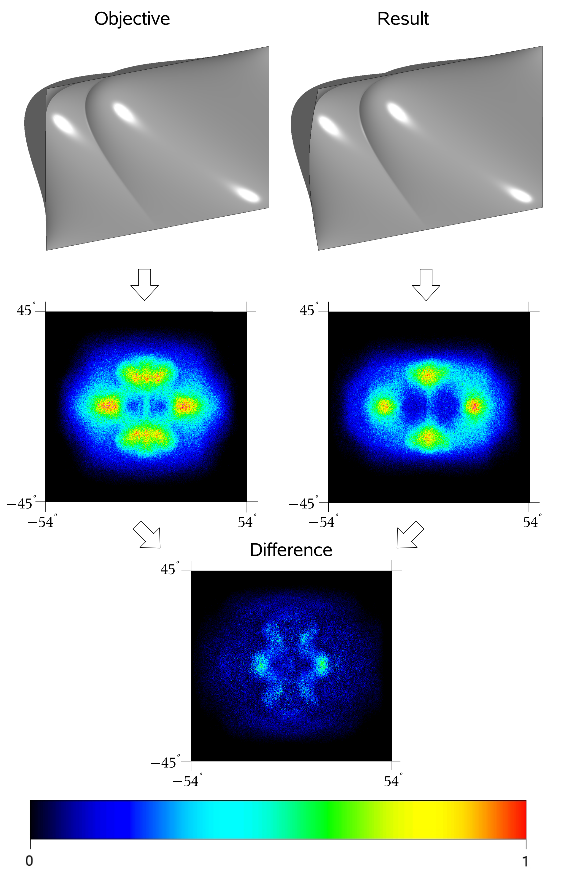


Figure 11: *Results for our Model B. At the top, the desired and obtained reflectors. In the middle, the desired and obtained far-field histograms in false-colour, indicating the respective angle domains. At the bottom, the histogram difference between both.*

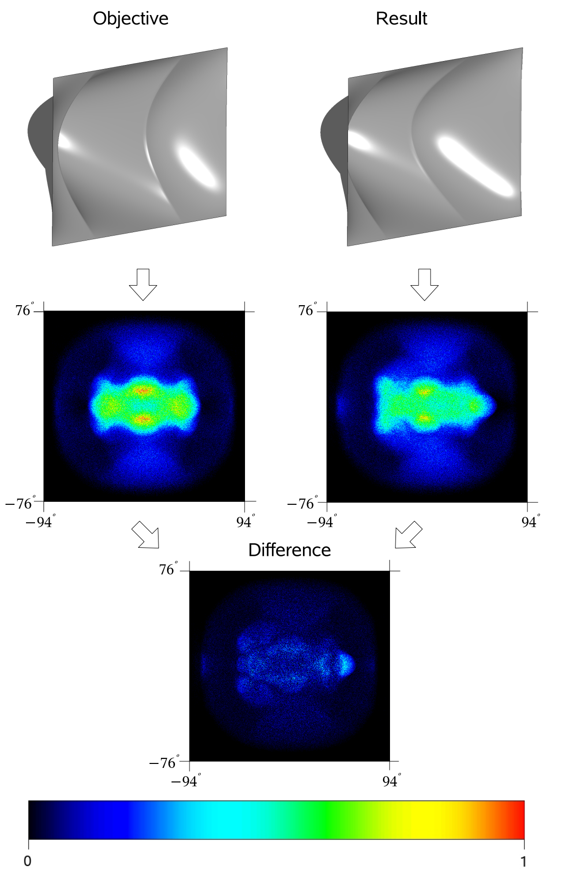


Figure 12: *Results for our Model C. At the top, the desired and obtained reflectors. In the middle, the desired and obtained far-field histograms in false-colour, indicating the respective angle domains. At the bottom, the histogram difference between both.*

ber of effective rays is 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 the rays intersect the height map. The time needed to compute the reflector lighting depends on the number of effective rays and the number of maximum allowed bounces. All models have a similar number of effective rays, but *Model A* has the lower computation time because only one bounce is specified. The optimization time depends on the 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.

Table 3 summarizes the broken down times for each reflector lighting step. The height map creation times are similar because all the models use the same mip-map height texture resolution. The intersection search time depends on the

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

Table 3: *Mean times (in milliseconds) broken down into the three main algorithm sections.*

number of traced rays, on the maximum number of allowed bounces and on the height map levels.

Figure 13 shows the progress in the optimization process The results are very similar between the three different models, because they have the same height map texture sizes (thus, the same number of quadtree levels), and the number of traced rays is also similar. The GPU parallel processing involves a linear computational cost on rayset size. Therefore, the most important factor in the intersection search procedure is the maximum number of allowed bounces. Finally, the error calculation has similar times for all cases, since the outgoing textures have the same size.

7. Discussion

As is shown in the previous section, we cannot obtain the desired reflector with zero error. This is because the optimization algorithm tests different parameterized reflectors by changing the parameter values in a constant step size and in a floating point space. On the other hand, we can improve the results by optimizing in very small steps, thereby guaranteeing convergence to a better solution, but this would strongly affect the processing times. Also, the semivariogram gives us a lower bound to the step size for each parameter in the optimization.

The most time consuming part of our method is the intersection search algorithm. If we use a very refined height map, we will need more time to traverse the ray through the quadtree. If we wanted to manage very complex reflector shapes, we would need height maps with high resolutions. Therefore, we need to work to find a compromise 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 parameterized reflectors, the one that best approximates 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, using millions of rays and highly complex reflector shapes. The reflector parameters are optimized in an iterative process until the generated light distribution is close enough to the desired one.

We consider, as future work, the use of better optimization methods, e.g. adaptive methods, so the desired reflector can be obtained faster. Another line of research is optimization based on a combination of predefined complex reflector shapes, which can be stored as texture masks.

9. Acknowledgments

This work was done under grant TIN2007-67120 from the Spanish Government, and under grant 7th Framework Programme-Capacities (grant No 222550, project EcoStreetLight) from the UE.

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, 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-field scattering data. *Contemporary Mathematics 226* (1999).

[Neu97] NEUBAUER A.: *Design of 3d-reflectors for near field and far field problems*. Springer, 1997.

[Ole99] OLEA R.: *Geostatistics for Engineering and Earth Scientists*. Kluwer Academic Publishers, 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.

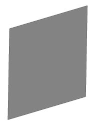
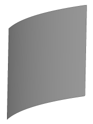
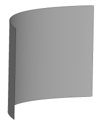
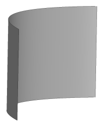
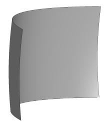
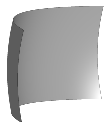
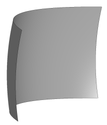
[RUL00] REVELLES J., URENA C., LASTRA M.: An efficient parametric algorithm for octree traversal. In *Journal of WSCG* (2000), pp. 212–219.

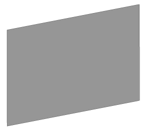
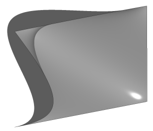
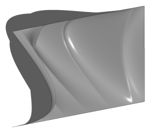
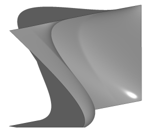
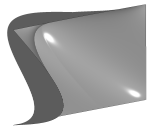
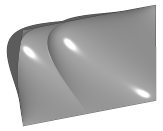
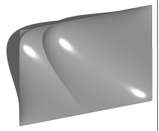
[SG06] SCHRODERS M. F. A., GULIK R. V.: Quadtree relief mapping. In *GH '06: Proceedings of the 21st ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware* (New York, NY, USA, 2006), ACM, pp. 61–66.

[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).

[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.

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	

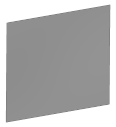
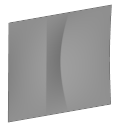
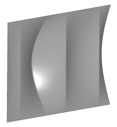
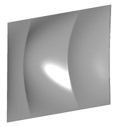
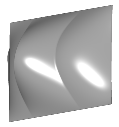
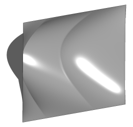
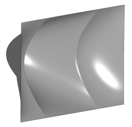
Model C							
# Step	0	1636	2446	3859	3886	3891	**Desired**
l² error	1.7105	0.980502	0.802349	0.485431	0.411784	0.245821	

Figure 13: *Reflector searching progress, from an initial shape (left), to the desired one (right). From top to bottom, models A, B and C. Below each reflector, there are the current number of steps in the optimization process and the l^2 error*