Reliable Soft Shadows over Implicit Surfaces

Jorge Flórez-Díaz
Institut d'Informàtica i Aplicacions
Campus Montilivi
Universitat de Girona
Spain (17071), Girona, Catalonia
jeflorez@ima.udg.edu

Mateu Sbert
Institut d'Informàtica i Aplicacions
Campus Montilivi
Universitat de Girona
Spain (17071), Girona, Catalonia
mateu@ima.udg.edu

ABSTRACT

This paper introduces an efficient and reliable algorithm to create soft shadows for ray traced implicit surfaces. In a classical soft shadow creation process, many rays have to be traced to calculate how much of the area light is visible from a point. In our approach, we trace from a point a beam that covers the area light. If the beam is not occluded, the point can be safely discarded because the use of Interval Arithmetic guarantees that the point is fully lit by the area light. In this case, extra rays are not traced, thus extra intersection tests are not required. Our soft shadow algorithm is then 38% to 59% faster than an algorithm based in a regular number of beams traced for each point. Besides, the using of beams based on Interval Arithmetic guarantees that thin details of the surfaces are not lost during the process.

Keywords

Soft shadows, Implicit Surfaces, Beam tracing, Interval Arithmetic.

1. INTRODUCTION

An implicit surface can be defined as the set of points from the equation defined by $f(x,y,x)=0$, where $f : \Omega \subseteq \boldsymbol{R^3} \rightarrow \boldsymbol{R}$. Finding methods to perform a reliable ray tracing process of these surfaces has been the subject of study by several authors in the last two decades [Kal89a,Mit90a,Har97a]. However, the most extended technique to perform such work is a root finding process using Interval Arithmetic [Mit90a,Cap00a,San03a].

Ray tracing of implicit surfaces is a slow process. Furthermore, the use of Interval Arithmetic decreases the performance of such algorithms. This is because every interval operation requires many floating point operations. For that reason, the creation of effects like soft shadows are often avoided due to the vastly increased number of rays required.

Another problem is that some special implicit surfaces are still not correctly rendered. This occurs because the rays miss the thin parts of the surfaces crossing a pixel, causing aliasing problems. This is more noticeable in the shadows produced by the implicit surfaces.

In this paper, we propose an efficient and reliable algorithm to create soft shadows for implicit surfaces using Interval Arithmetic. Our method is based on the use of shadow beams instead of rays, having the following advantages:

- A shadow beam can be used to replace the process of many shadow rays. If the beam is not occluded by any object in the scene, then the trace of the single rays is avoided, saving the computational time required for such intersection test.
- Our reliable beam can detect any feature (even the smallest ones), avoiding aliasing problems in the shadows.

Previous Works

There are a huge number of articles related with soft shadows in literature. The most widely used algorithms are based in rasterization techniques like shadow maps [Kir03a,Arv04a,Ann08a], in which a pre-filtered depth map of the shadows are created, and shadow volumes [Asr03a,Leh06a] which defines the shadow boundary using a geometric algorithm for a silhouette extraction. Guennebaud et al [Gue06a] developed an algorithm which computes a percentage of light seen from every pixel in the image. This is done by backprojecting the occluders in a shadow map onto the light plane and determining the magnitude of the occlusion. These techniques can run very fast in graphics hardware.

However, they can not be easily adapted to work with a pure ray tracing process for implicit surfaces.

An approach that can be easily applied to create soft shadows for ray tracing is Monte Carlo sampling [Coo84a]. Here, many rays have to be traced against the area light, which is inefficient for implicit surfaces where the root finding process is slow. To solve that problem, coherence approaches can be applied.

In pencil tracing [Shi87a], the rays nearby to a special ray (called the axial ray) are grouped. These rays (called paraxial rays) are represented by a 4D matrix that represents the deviations in position and direction from the axial ray. This matrix system can be combined with the matrices for every surface in the environment to represent the propagation of the rays. The disadvantage of this method is that the surfaces have to be smooth, because the system can not deal with discontinuities. In those cases, the method requires of individual rays instead of pencils.

In cone tracing [Ama8a], the rays are represented by a cone conformed by an apex, a center line and a spread angle. To calculate reflections and refractions, the new center line is calculated using a standard ray tracing technique. This technique can deal with the aliasing problems in the soft shadows by means of the calculation of the part of the cone blocked by the objects. However, due to the complexity in the calculation of intersections and parts of the cone covered by the objects, the method only deals with spheres, planes and polygons.

Beam tracing [Hec08a] replaces individual rays with beams. A beam consists in a set of rays with a common apex crossing planar polygons. When a beam intersects an object, a new beam with a new polygonal section is generated. The remainder of the beam can be complex forms that require a method that can operate with arbitrary polygons. However, this technique is slow, only works for planar polygons facets and require complex intersections test.

There are some techniques that improve the beam tracing process, which can be used to accelerate the soft shadow generation. [Over07a] introduced a technique based in algorithms for efficient beam-triangle intersection, and beam-kd-tree traversal to generate the soft shadows. In [Car09a] a frustum tracing technique is introduced. Here, shadow rays are generated in every frustum on demand. This technique is well suited for SIMD architectures, and was specially designed for the Larrabee architecture of Intel. These techniques represent the beam by means of four rays, which are traversed in the acceleration structures. Those techniques can be easily adapted to work with implicit surfaces. However, they could miss small features of the implicit surfaces that lie inside the beam represented by four unconnected rays.

In [Over99a] there is a proposal to create an approximation of the soft shadow for implicit surfaces. However, it only works for simple implicit surfaces and generates only approximations of the shadows.

2. PRELIMINARIES

Our proposal is based in the use of beams and Interval Arithmetic to accelerate the creation of the soft shadows, as well to guarantee that any part of the surfaces is not missed during the beam traversing. Besides, our algorithm was implemented for GPU using the Nvidia's CUDA architecture to improve the rendering time. Using a CPU, a ray tracer using Interval Arithmetic could take minutes for naïve implicit surfaces [Cap00a,San03a]. Other approaches based on GPU have demonstrated that the rendering time for the ray tracing of implicit surfaces could be decreased up to 3 orders of magnitude [Col08a,Woo04a].

Beam Definition

Using interval arithmetic, a beam is defined by:

$$X = c_x + T(X_s - c_x)$$

$$Y = c_y + T(Y_s - c_y)$$

$$Z = c_z + T(Z_s - c_z)$$

Where X, Y, Z; X_s, Y_s, Z_s; T are intervals (in this paper, intervals are defined as capital letters). (c_x, c_y, c_z) is the origin or view point; X_s, Y_s, Z_s are the intervals defining the direction of the beam, and T is the interval parameter of the beam, which determines how much the beam is advancing in the direction X_s, Y_s, Z_s.

The intersection between a beam and an implicit surface defined by $f(x, y, z) = 0$, is defined by:

$$0 \in F(X_s, Y_s, Z_s, T)$$

where $F(X_s, Y_s, T)$ is called the "*inclusion function*" which is equivalent to:

$$f(c_x + T(X_s - c_x), c_y + T(Y_s - c_y), c_z + T(Z_s - c_z))$$

So, the intersection test for the beam consists in finding a space of solutions (T) for all the rays that conforms the beam. The value T can be found using a classical Interval Bisection method or an Interval Newton method [Cap00a].

Interval Arithmetic guarantees that for any value $x \in X$, $f(x) \subseteq F(X)$, where $F(X)$ is the inclusion function. Hence, the advantage of the beam definition using Interval Arithmetic is that any part

of the surface crossed by the pixel is not missed (see figure 1).

If $0 \notin F(X_s, Y_s, Z_s, T)$ then it is sure that there is not an intersection between the beam and the surface for any value $t \in T$.

If $0 \in F(X_s, Y_s, Z_s, T)$ then it is possible that there are some intersections between any of the rays included in the beam and the implicit surface. In this case, if a subdivision process is performed over the parameter of the beam T, the current value of T must be subdivided and every new interval evaluated again. The subdivision algorithm needs to arrive to machine precision because in that case $0 \in F(X_s, Y_s, Z_s, T)$ is often achieved [Cap00a].

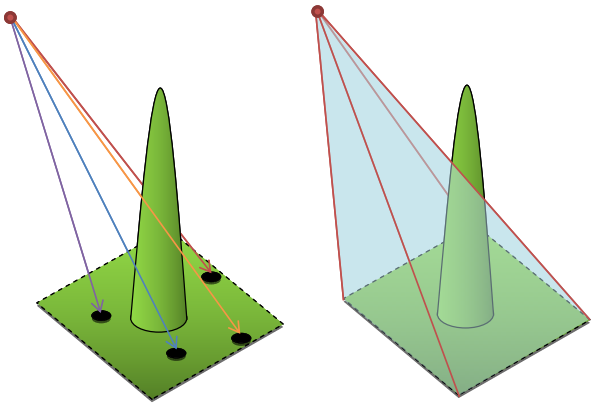

Figure 1. (Left) Some rays can miss thin parts of an implicit surface. (Right) Using beams, it is possible to discover such thin details.

Acceleration Structure

In our implementation we use a regular grid to accelerate the beam traversal. The nature of the rendered scenes, composed by arbitrary implicit surfaces uniformly distributed, makes a regular grid a good candidate to accelerate the beam tracing process [Fuj01a]. Moreover, using a regular grid the complexity of the beam traversing is reduced.

We found that a regular grid is an approach that fits perfectly in the SIMT architecture (Single-Instruction, Multiple-threads) of CUDA. Here, each independent thread can perform the evaluation of a cell of the grid, so the evaluation of many cells can be performed very fast. In our implementation this time is meaningless compared with the time spent in the rendering process.

To create the regular grid, the object space in the scene is subdivided in a predefined number of boxes or cells, and each one is scanned to look for the implicit surfaces crossing them.

Having an implicit surface $f(x,y,z)=0$, to know if the surface crosses a cell defined by the intervals (X,Y,Z), the united extension $F(X,Y,Z)$ is used. If $0 \in F(X,Y,Z)$, the region may be crossed by the surface. In the other case, the region can be definitively discarded.

Beam Traversal

To determine the boxes intersected by the beam, the direction of the beam in x, y or z (in space coordinates) is selected. This is done taking the normalized directions of the four vectors composing the corners of the beam and selecting the one with the bigger absolute value. The boxes are scanned advancing in the coordinate selected as direction, but only for the boxes covered by the other two coordinates of the beam.

Figure 2 represents a 2D scenario to facilitate the understanding of the process. In this case, the output is a line instead of an area. The propagation direction is supposed to be in the z axis. The adaptation of this process to a 3D case is straightforward.

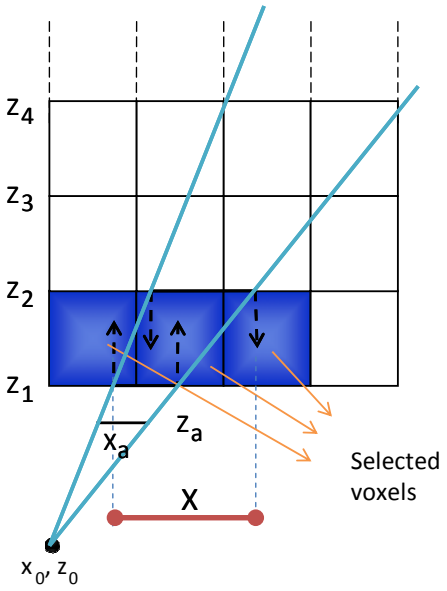

Figure 2. Traversing of a beam inside a regular grid structure.

Every row in direction z is checked, but only the cells covered by the interval ***X*** are considered for the evaluation of the intersection test (cells in blue). The interval covering the cells in x coordinate is calculated as follows:

$$X = x_0 + T(X_a - x_0)$$

having:

$$T = (Z - z_0) / (Z_a - z_0)$$

in which X_a and Z_a are the intervals representing the direction of the beam, and x_0, y_0 are real values representing the viewpoint. The value of Z is obtained taken the minimum and maximum values of z for the current row of cells in the grid.

The algorithm continues looking for cells covered by ***X*** in every row in the grid.

In every cell, the corresponding interval value of the parameter of the beam (T_C) must be calculated. This value represents the minimum and maximum value in which the roots are searched for the parameter *T*. This value is calculated as:

$$T_C = [\max[\underline{T_z}, \underline{T_x}], \min[\overline{T_z}, \overline{T_x}],$$

where $\underline{T_z}, \underline{T_x}$ are the lower bounds, and $\overline{T_z}, \overline{T_x}$ are the upper bounds of the intervals T_z, T_x. Moreover:

$$T_z = (V_z - z_0) / (Z_a - z_0), \quad T_x = (V_x - x_0) / (X_a - x_0),$$

where the cell is defined by the intervals V_x and V_z.

Root Searching

When a beam intersects a cell crossing a part of a surface, a root must be searched in the space T_c corresponding to the current cell. Usually, recursion is used for the root searching algorithms. However, in the GPU the recursion is not allowed. For that reason, we made an algorithm which work making bisections over the value T_c and evaluating the inclusion function in the two sections created in every step. We use two variables to control the bisection process: one variable, *ns,* keeps the number of subdivisions currently performed over the parameter T_c in power of two (number of bisections = 2^{ns}), and the other, *cs,* is pointing to the section currently scanned. For instance, if *ns* = 2 and *cs* = 3, then *Tc* is currently subdivided in four sections, and the section in which the inclusion function is evaluated is the third one. This means that $cs < 2^{ns}$ during the bisection process.

The subdivision process must start with ns = 1 (T_c is subdivided in two sections), and *cs* = 1 (pointer to the first section). If we find that $0 \in F(X_s, Y_s, T)$ (the root could be in the current section), the current section is bisected, the level of subdivision is incremented (*ns* = *ns* + 1) and the *cs* is set to the first section of the current subdivision level (*cs* = *cs**2 - 1).

If for the current section $0 \notin F(X_s, Y_s, T)$ (there are no roots in this section), then we must check if the pointer is in the first or the second section. If the remainder of *cs*/2 is different than 0, then *cs* is pointing to the first section. In this case, *cs* is set to the next section (*cs* = *cs* + 1). If the remainder of *cs*/2 is equal than 0, then *cs* is in the second section of the current subdivision. In that case, the two sections are rejected, the level of subdivision is reduced (*ns* = *ns* - 1), and two new sections are selected, just following the two evaluated sections in the last step (*cs* = *cs*/2 + 1).

The process continues evaluating pairs of sections until a section reached a predefined small size. In our implementation, we finish the process when the width of the currently evaluated section is less than 10^{-5}. The process is also finished if $cs > 2^{ns}$. This occurs when the bisection process did not find any intersection between the beam and the surface.

3. ALGORITHM SPECIFICATION

In our implementation, a PBO (Pixel Buffer Object) is created and sent to the GPU. We use the PBO to keep the final color values corresponding to every pixel of the window. After that, the surface to be rendered can be defined by the user. The surfaces were previously loaded, so the user only has to define an index indicating which surface has to be rendered. Once the surface is defined, the CUDA programs are executed, and the CPU waits until the results are collected from the GPU. Finally, OpenGL is used to show the contents of the PBO in a window.

Creation of the Grid

In our algorithm, the first step performed in the GPU is the definition of a grid to accelerate the beam tracing process. This task can be performed directly in CUDA or can be performed in CPU. For the last case, the data of the grid must be saved in an array to be sent to the GPU.

Beam Casting Process

Having the grid in the GPU, a beam casting process is performed, tracing a beam for every pixel in the screen. The objective is to trace a beam covering all the area of the pixel, and traverse the beam to find the intersections with the surfaces. For this task, we divided the screen in many tiles, which are sent to the GPU to be processed in one block each one. The result of this task is saved in an array with size equal to the number of pixels in the screen. The array has two floats for each position. Those values are used to keep the lower and upper bounds of the space of solutions (*T*) for the corresponding pixel. This value can be replaced in the definition of the beam to obtain the corresponding intersection values for *X, Y* and *Z*.

Because the intersection points are intervals, we use the midpoints to obtain the initial point of the shadow beam.

Soft Shadow Beam Tracing

Having the initial point, a shadow beam is traced from the initial point at a distance $\propto$ to avoid self intersections. The beam must cover all the area of the light.

In our implementation, we define the area lights as squares that are parallel to the x,y or z axis. We use three intervals to define the light (L_x,L_y,L_z), but one

of them is a point-wise interval. The intervals defining the light are used as the direction of the beam.

We use the shadow beam to analyze what happens with the rays included in the beam. There are two possible situations:

- Case 1: An occluder is not found between the initial point and the light. It means that all the rays inside the beam are not occluded by any surface in the scene.
- Case 2: the beam intersects a surface in the scene. It means that some or all the rays that conform the beam are occluded by an object.

In the first case, the point is not covered by a shadow. This means that further rays are not needed; so we finish for this point and continue with the next. This represents a lot of time saved in the calculation of the intersection test for new rays, which is the more expensive step in a ray tracing process (see figure 4).

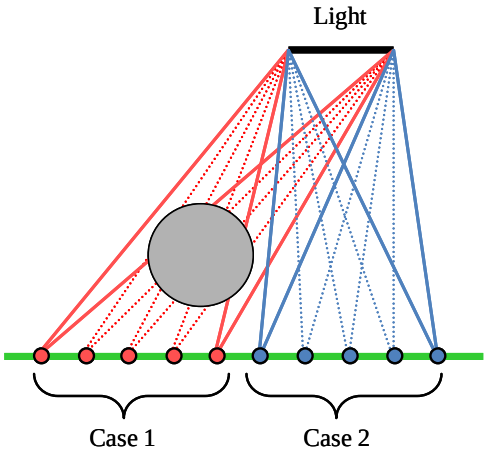

Figure 4. Two cases detected during the beam tracing process. Case 2 can represent a lot of points in a scene, saving the tracing of new rays.

For the second case more rays are needed. It is possible to apply a classical Monte Carlo sampling or any other method based in rays. In our implementation, we opt to trace more sub-beams instead or rays to guarantee that small features are not missed. Our idea is to avoid aliasing problems in the visualization of the shadow (see figure 5).

If the beam intersects any of the surfaces in the scene, the area light is subdivided in many tiles. Many sub-beams starting in the same point are traced against these tiles. The count of sub-beams hitting surfaces over the total number of traced beam is used to create an index of the percent of the area of the point under the shadow. The value of this index varies from 0 for points not covered by shadow, to 1 for surfaces in the umbra

In our implementation, the light is subdivided in a regular number of tiles, tracing new beams against them. Initially, we thought that subdivision process like a quadtree could reduce the number of beams needed, and hence reducing the processing time. However, we found that our implementation using a quatree for the subdivision of the light, was up to 1.5 times slower than an implementation using a regular subdivision. This occurs because GPU is faster with a completely parallel approach, in which every beam can be traced independently of the others. A quadtree requires more conditionals, and its irregular structure makes this approach slower than the tracing of a regular and predefined number of sub-beams against the light.

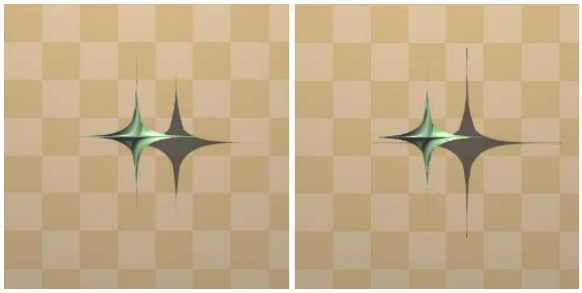

Figure 5. A twister surface with thin features. Left: using rays, the shadow is not well defined. Right: using beams, the thin details are revealed in the shadow.

4. EXPERIMENTATION

To test our algorithm, we used four surfaces: Orthocircle, Chubs, Kusner Schmitt and McMullen (figure 6).

The soft shadows were generated using 49 sub-beams. We found that this number of beams is good enough to obtain the results in figure 6. We use an area light of *0,1* of the size of the area of the screen. The images were generated at a resolution of *512x512* pixels in a GPU NVIDIA 9600M GT.

Table 1 shows the results of our algorithm compared to an algorithm in which 49 sub-beams are traced for every intersection found during the beam casting process. Our method can drastically reduce the number of beams needed for the soft shadow creation. Time reduction in all the cases is over 38% (arriving up to 59% for the Orthocircle) for the shadow creation process.

In Table 1, the column “initial beams occluded” indicates the number of initial beams traced, for which an occlusion was detected. Note that the number of “pixels in shadow”, which are detected when the sub-beams are traced, is smaller than the pixels detected by the “initial beam occluded”. This occurs because the overestimation of the intervals produces a beam that increases its real size. For that

reason, some pixels are detected to be in a shadow but they are not (see figure 7).

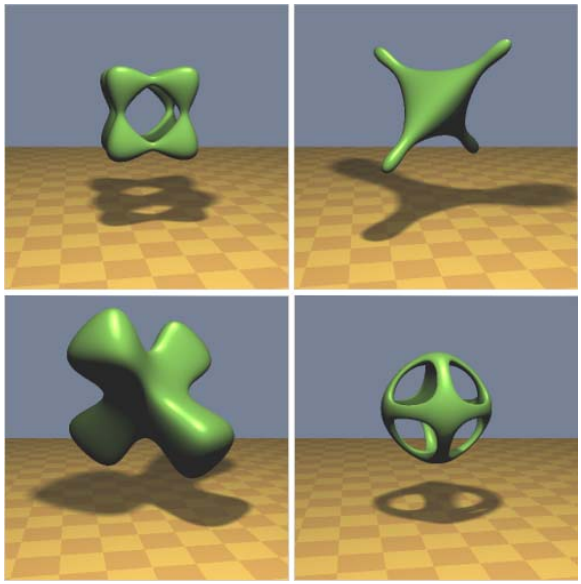

Figure 6. Surfaces used in our experimentation, from left to right and top to bottom: Chubs, Kusner-Schmitt, McMullen and Orthocircle.

The overestimation is proportional to the width of the used intervals. In this algorithm, this effect depends of the size of the light: for a bigger light, more pixels will be detected to be in a shadow when they really are not. However, if the intervals are subdivided, and each one is computed independently, the effect of overestimation is reduced [rev05a]. In our algorithm, when the small beams are traced (the size of the intervals is smaller) a correct representation of the soft shadow is found. The overestimation represents an extra cost in the rendering time, because 49 rays are traced from some points fully lit by the light. However, this matter does not affect the quality of the image. In the case of a really big light, instead of tracing an initial beam, it is possible to trace four, and perform the same analysis presented in this paper.

The efficiency of the algorithm is related to the number of pixels covered by a shadow. For example, the Orthocircle have a 5% of the pixels under a shadow, and the improvement is 59% over the approach based in a regular number of beams. For the McMullen surface, the shadow covers the 21% of the pixels, with an improvement of 38%.

Another issue to be taken into account is that a beam detects pixels under a shadow where a ray can miss thin features of the surface. The same occurs for the borders of the surface: the beam detects more pixels under a shadow than an approach based in rays. Indeed some of these pixels could be fully lit. The overestimation contributes to obtain this effect. However, this issue does not affect the quality of the images obtained (see figure 6).

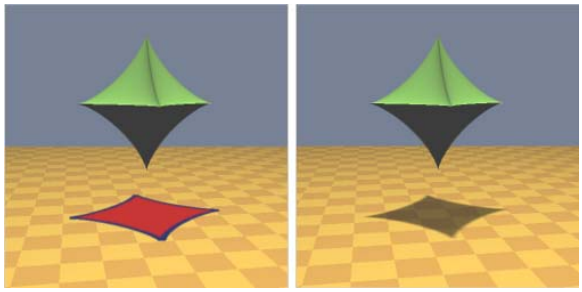

Figure 7. Left: The blue and red regions indicate the pixels detected to be in a shadow by the initial beams. The red region indicates the pixels that are indeed under the shadow. Right: the surface and its soft shadow.

5. CONCLUSIONS

In this paper we introduced a reliable soft shadow algorithm for implicit surfaces. Here, we propose the use of beams based on interval beam arithmetic to accelerate the rendering process and to detect thin features of the implicit surfaces.

We prove that using beams, the creation of soft shadows can be accelerated. Using only one beam it is possible to detect if a pixel is under a shadow. If occlusions are not detected, then the tracing of more rays for the same point are avoided, thus saving computational time. This is more efficient than tracing many rays to detect the soft shadows in every intersection found during a ray casting process.

Besides, Interval Arithmetic allows the creation of convex hulls in which none of the solutions is lost. A beam based on Interval Arithmetic can guarantee that either small or thin parts of the surfaces are not missed in the final image.

As a future work, we are going to study the application of Affine Arithmetic in order to improve the computational time. Affine arithmetic promises to reduce overestimation problems, and to keep all the advantages of the interval arithmetic.

6. ACKNOWLEDGEMENTS

This work has been partially funded by the Spanish Government (Plan Nacional de Investigación Científica, Desarrollo e Innovación Tecnológica, Ministerio de Ciencia y Tecnología) through the co-ordinated research project MEC TIN2007-68066-C04-01 and by the government of Catalonia through 2009 SGR643.

7. REFERENCES

[Ama8a] Amanatides J. Ray Tracing with Cones. Computer Graphics 1984, 18, 3, 129-185.

			Our method (using initial beams)			Using 49 beams		
Surface	Beam casting (secs.)	Pixels in shadow	Initial beams occluded	Time (secs.)	No. Beams	Time (secs.)	No. Beams	%time saved
Orthocircle	1,23	13.328	14.961	1,66	719.859	4,08	7.101.962	59%
Chubs	1,53	21.932	27.357	3,82	1.340.493	6,57	7.343.091	42%
Kusner Schmitt	2,24	22.374	44.979	5,65	2.203.971	9,44	7.395.374	40%
McMullen	2,28	55.231	73.386	6,84	4.085.914	11,06	8.749.244	38%

Table 1. Results of the experimentation.

[Ann08a] Annen, T., Dong, Z., Mertens, T., Bekaert, P., Seidel, H., and Kautz, J. 2008. Real-Time All-Frequency Shadows in Dynamic Scenes. ACM Transactions on Graphics. 27, 3, 1–8. 2008

[Arv04a] Arvo J. and Westerholm J. Shadows Using a Single Light Sample Hardware Accelerated Soft Shadows using Penumbra Quads. Journal of WSCG, Vol.12, No.1-3, 2004

[Asr03a] Assarsson U., Akenine-Möller T.: A geometry based soft shadow volume algorithm using graphics hardware. ACM Transactions on Graphics. 22, 3. 511–520. 2003

[Cap00a] Capriani O., Hvidegaard L., Mortensen M. and Schneider T. Robust and efficient ray intersection of implicit surfaces. Reliable Computing. 1, 6,, 9-21, 2000.

[Car09a] Carsten B. and Wald I. Efficient Ray Traced Soft Shadows using Multi-Frusta Tracing. Proceedings of the Conference on High Performance Graphics. 135-144. 2009.

[Col08a] Collange S., Flórez J. and Defour D. Interval Library based in Boost Interval. Proceedings of the International Conference on Real Numbers and Computers, 61-72. 2008

[Coo84a] Cook, R., Porter, T., and Carpenter, L. Distributed Ray Tracing. Computer Graphics (Proc. of SIGGRAPH ’84) 18, 3, 137–144.

[Fuj01a] Fujimoto, A. Tanaka, T. Iwata, K. ARTS: Accelerated Ray-Tracing System. IEEE Computer Graphics and Applications. 6,4,16-26, 1986.

[Gue06a] Guennebaud G., Barthe L., and Paulin M. Realtime soft shadow mapping by backprojection. In Eurographics Symposium on Rendering, 227–234, 2006.

[Har97a] Hart J. Sphere tracing: A geometric method for the antialiased ray tracing of Implicit Surfaces. The Visual Computer, 12, 10:527–545, 1997.

[Hec8a] Heckbert P. and Hanrahan P. Beam Tracing polygonal Objects. Computer Graphics 1984, 18 (3), 119 – 127.

[Kal89a] Kalra D. and Barr A. Guaranteed ray intersection with implicit surfaces. Computer Graphics (Siggraph proceedings), 23:297–206, 1989.

[Kir03a] Kirsch F. and Döllner J. Real-Time Soft Real-time soft shadows using a single light sample. Journal of WSCG (Winter School on Computer Graphics 2003), 11, 1, 2003.

[Leh06a] Lehtinen, J., Laine, S., and Aila, T. 2006. An Improved Physically-Based Soft Shadow Volume Algorithm. Computer Graphics Forum (Proceedings of Eurographics ’06) 25, 3.

[Mit90a] Mitchell, D. Robust ray intersection with interval arithmetic. Proceedings on Graphics Interface ’90, 68–74, 1990.

[Over07a] Overbeck R., Ramamoorthi R.,and Mark W. A Real-time Beam Tracer with Application to Exact Soft Shadows. Eurographics Symposium on Rendering 2007.

[Over99a] Overveld K., Mark T. and and Wyvill B. Soft Shadows for Soft Objects. Proceedings of Fourth Eurographics Workshop on Implicit Surfaces, Bordeaux, France. 1999.

[Rev05a] Revol, N. and Rouillier, F. Motivations for an Arbitrary Precision Interval Arithmetic and the MPFI Library. Reliable Computing, 11, 4, 275-290, 2005

[San03a] Sanjuan-Estrada J., Casado L. and García I. Reliable algorithms for ray intersection in computer graphics based on Interval Arithmetic. XVI Brazilian Symposium on Computer Graphics. 35 – 44, 2003.

[Shi87a] Shinya M., Takahashi T. and Naito S. Principles and applications of pencil tracing. Computer Graphics 1987, 21, 4, 45-54.

[Wil78a] L. Williams. Casting curved shadows on curved surfaces." SIGGRAPH '78, pp. 270-274. New York, USA, 1978.

[Woo04a] Wood A, Brendan M. and Scott K. Ray tracing arbitrary objects on the GPU. Proceedings of Image and Vision Computing, 21-23, 2004.A GPU