IMPROVEMENTS IN THE RAY TRACING OF IMPLICIT SURFACES BASED ON INTERVAL ARITHMETIC

Jorge Eliécer FLÓREZ DÍAZ

ISBN: 978-84-692-7819-2
Dipòsit legal: GI-1320-2009

Universitat de Girona
Departament d'Electrònica, Informàtica i Automàtica

Improvements in the Ray Tracing of Implicit Surfaces based on Interval Arithmetic

by

Jorge Eliécer Flórez Díaz

Advisors

Dr. Mateu Sbert
Dr. Josep Vehí

DOCTORAL THESIS
Girona, Spain
November, 2008

Universitat de Girona
Departament d'Electrònica, Informàtica i Automàtica

Improvements in the Ray Tracing of Implicit Surfaces based on Interval Arithmetic

A dissertation presented to the Universitat de Girona in partial fulfillment of the requirements of the degree of DOCTOR OF PHILOSOPHY

By

Jorge Eliécer Flórez Dáz

Advisors

Dr. Mateu Sbert

Dr. Josep Vehí

Girona, Spain
November, 2008

Abstract

Implicit surfaces are useful in many areas related to computer graphics. One of their main advantages over other representations is that they can be easily used as primitives for modeling (using Constructive Solid Geometry or blending). However, they are not widely used for this purpose because the models created with implicit surfaces take a long time to be rendered.

When a precise visualization of an implicit surface is required, the best option is to use ray tracing. If simple surfaces are rendered, then the results are suitable, however, thin features can be missed in models that have thin parts, and are not rendered. These problems are caused by the truncation performed in the floating-point representation in the computers: some bits are lost in the mathematical operations during the intersection tests between the rays and the surfaces.

Many authors have used Interval Arithmetic in the intersection test to solve the problems related with point sampling, that cause the loss of intersection points, however, there are still two open problems in the ray tracing of implicit surfaces based on Interval Arithmetic:

- Ray tracing is slow, and Interval Arithmetic is slow too. Many floating-point operations are required for every Interval Arithmetic operation (besides rounding). For that reason, the ray tracing becomes even slower.

- Although Interval Arithmetic has been applied to replace point sampling during the intersection tests, the application of Interval Arithmetic to replace point sampling in the distribution of the rays inside the pixel, has not been studied. This also causes loss of thin parts of the surface in the final image.

In this work algorithms to deal with those problems are presented. The research is based on Modal Interval Analysis, which is a logical completion of classic interval analysis that includes tools for solving quantified uncertainty. Modal interval Analysis gives the mathematical foundations used in the development of these algorithms.

The efficiency of the ray tracing of implicit surfaces is improved by means of the evaluation of groups of rays instead of individual ones, which permits thus saving computational time in the entire ray tracing process. The quality of the visualization is improved by means of the creation of an adaptive anti aliasing algorithm. Using this strategy, it is possible to evaluate areas of the pixels instead of points. The efficiency of animated scenes composed by implicit surfaces is also improved. The improvement is achieved by means of an algorithm that exploits the coherence between frames.

This research also shows how it is possible to reduce the rendering time by means of the use of cluster of computers and also by means of graphic processing cards, where the improvement in efficiency is between two and three orders of magnitude.

Dedicatory

To my family. The distance does not matter, they always take care of me.
To Annie. Thanks to be so supportive, and helping me to find the strength I needed to finish this work.

Acknowledgements

First of all, all my gratitude to my advisors Dr. Mateu Sbert and Dr. Josep Vehí for their advice during the development of this Thesis.

I also would like thank to Dr. Miguel A. Sainz for his advice, specially in the use of the Modal Interval theory during the development of this thesis.

Thanks to professor Brian Wyvill and Erwin de Groot for their valuable ideas and suggestion about my research work during my stay in Victoria (Canada).

Also to Dr. David Defour and Sylvain Collange from Perpignan University (France) for their advice in the numerical issues of GPU.

Also, thanks to Nathan Hayes from Sunfish Studio, who gave me good observations at the begining of this thesis during my stay in Minneapolis (USA).

Finally, thanks to the people from MiceLab and Gilab who share some time with me while I was doing this PhD, specially Oscar, Lucho, Jian, Nico, Xavi, Sergi, Hug, Ruben and Olha.

Thanks to the Catalonian Goverment (Generalitat de Catalunya), for the economical support through the IGSOC grant (one year), and after that, through IQUC-FI grant (three years).

This work has been also partially funded by the European Union (European Regional Development Fund) and the Spanish Government (Plan Nacional de Investigación Científica, Desarrollo e Innovación Tecnológica, Ministerio de Ciencia y Tecnología) through the coordinated research projects TIN2007-68066-C04-01, TIN2004-07451-C03-01, DPI2002-04018-C02-02, DPI2003-07146-C02-02, DPI2004-07167-C02-02, DPI2005-08668-C03-02 and by the government of Catalonia through SGR00296.

Contents

List of Figures

List of Tables

1 Introduction

This chapter presents an overview of the background of the thesis, describing the motivation, the objective, and the contributions to the state-of-the-art in the guaranteed ray tracing of implicit surfaces based on Interval Arithmetic.

It is a challenge to correctly render an implicit surface. Although simple surfaces do not suffer from visualization problems, some surfaces and models in which they are used as primitives could suffer reliability problems.

Efficient methods to solve those problems are required. Although there are many approaches dealing with them, there are still some improvements that can be performed. In this thesis, we focus in the development of improved approaches based on Modal Interval Arithmetic.

1.1 Motivation

Finding methods to perform correct visualizations of implicit surfaces has been the subject of study by several authors in the last two decades. The main objective of these works is to perform reliable intersections tests for ray tracing scenes conformed by implicit surfaces.

When an implicit surface is ray traced, some small and thin features disappear in the final visualization. This problem is related with the use of point sampling in the intersection test. Point sampling fails because floating-point arithmetic in the computer can not represent all the set of real numbers. Such loss of reliability causes the miss of roots in the intersection test, and pixels are classified as empty, although the surface indeed lies inside the pixel.

There are different approaches that can be used to solve this problem, like Lipschitz constants, Affine Arithmetic or Interval Arithmetic. Although these approaches deal with the quality in the visualization, the challenge of achieving both quality and efficiency is not well afforded. One of the causes of the lack of efficiency is ray tracing, which is a slow method, although more reliable than other alternatives like the rendering of polygonized implicit surfaces.

The problem of efficiency is more noticeable when Interval Arithmetic is used. This arithmetic requires to replace the traditional floating-point operations for interval ones. Every interval has bounds composed by two floating-point numbers, and every operation (add, subtract, multiplication, division and so on) requires floating-point operations between those bounds. This implies an extra computational cost, specially when ray tracing is used.

Another problem is related with point sampling at pixel level. Although ray tracing could be reliable, some surfaces are still not correctly rendered because the rays miss the thin parts of the surfaces crossing a pixel. This is still an open research area for guaranteed ray tracing of implicit surfaces using Interval Arithmetic.

1.2 Why Interval Arithmetic?

Interval Arithmetic has been proved to be very helpful in the solution of problems caused by truncation and rounding of real numbers in the computer. Researchers in different areas like structures, control, robotics, chemistry and medicine have solved problems that occur when the floating-point arithmetic of the computer is used (a quick search in internet can return hundreds of results about applications of interval arithmetic).

Also, the application of Interval Arithmetic is very simple and intuitive. In most of the cases, it is used by way of libraries called by the applications when the interval operations are required. There are several public and commercial versions written in programming languages like ALGOL, PASCAL, FORTRAN, MODULA and C++. Generally, the libraries are defined by means of classes which implement the interval operators. This allows independent functioning between the computer graphic applications and the Interval Library.

Moreover, there is an Interval Arithmetic Community which is actively working in the development and improvement of the interval theory. Any improvement in the interval theory can be put in the Interval Library without affecting the algorithm that uses the library (see figure 1.1).

These advantages have made interval arithmetic a good candidate to solve reliability problems in the intersection test for ray tracing of implicit surfaces, although the problems of efficiency and point sampling at pixel level must be studied and solved. Particulary, this thesis is based in Modal Interval Analysis, which is an extended version of the classic interval arithmetic.

Modal Interval Analysis permits to deal with many mathematical problems, defining a logical statement of these problems and giving the tools and theorems to solve them.

1.3 Objective of the Research

The main objective is to develop algorithms to obtain a correct visualization of any implicit surface, including those surfaces with thin parts, when ray tracing with interval arithmetic is used. The proposed solution must take advantage of the potential of the "intervals" to exploit coherence and, in this way, improve the quality of the visualization and also reduce the time spent in the ray tracing process.

In this thesis, the term "quality" will refer to a correct rendering of the surface, that is, if all the parts of the surface appear in the final image. Moreover, an algorithm is "guaranteed" when this kind of quality is assured. The term "efficiency" will refer to

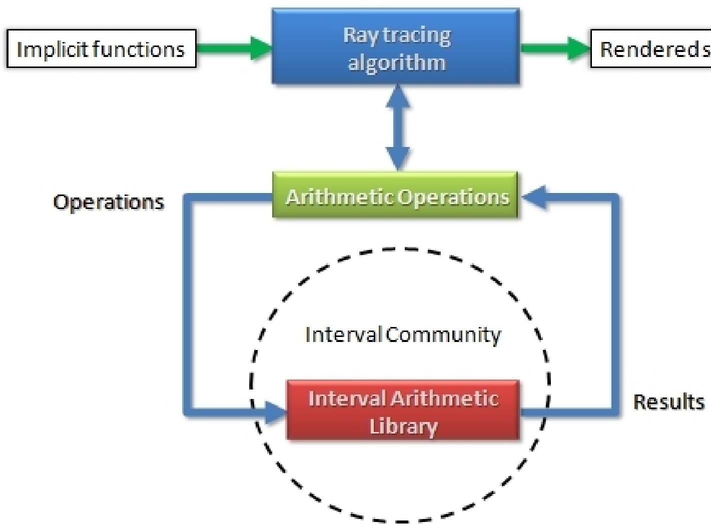

Figure 1.1: The ray tracing using interval arithmetic works performing calls to an independently interval arithmetic library. New versions of interval libraries can be used without affecting the ray tracer.

the rendering time, that is, an algorithm is more efficient than another if it takes less time to render the surface.

To accomplish this objective, this thesis deals with the following specific objectives:

1. To apply Modal Interval Arithmetic to improve the efficiency by means of the exploitation of coherence. Groups of rays can be evaluated with the same computational cost as an individual ray.

2. To improve the quality of the rendered surfaces, creating an adaptive anti aliasing algorithm. Point sampling can be replaced for an adaptive algorithm that evaluate areas of the pixels instead of points, improving the quality of the final image.

3. To prove that the solutions presented can be ported to other environments. The algorithms are tested over two approaches widely used by Computer Graphics researchers to improve efficiency: a parallelized algorithm and an implementation on GPU (Graphic Processing Unit).

1.4 Description of the contents

The thesis has the following chapters:

Chapter 2 is a general introduction to implicit surfaces, explaining its definition, representation and visualization techniques. The objective of this chapter is to give a general knowledge about implicit surfaces. The reader of this thesis can find this chapter useful to understand how the implicit surfaces are used in the context of computer graphics.

Chapter 3 presents a general introduction to interval arithmetic. It goes from the classical interval arithmetic basis, to the concepts of Modal Interval Analysis. This chapter contains the mathematical foundations to develop the algorithms introduced in this thesis.

Chapter 4 introduces the state of the art of the application of interval analysis in computer graphics, focusing on implicit surfaces. It also stresses on different interval algorithms to ray trace the implicit surfaces.

Chapter 5 contains the first contributions of this thesis. It starts explaining the interval arithmetic techniques used to accelerate the ray tracing process. This is done by means of the creation of a structure in image space, in which beams are used instead of individual rays. This chapter also includes algorithms that use interval arithmetic to replace the point-sampling in the adaptive anti aliasing approaches, creating better visualizations of the implicit surfaces.

Chapter 6 presents an application in the rendering of animations composed by implicit surfaces. Here, interval analysis is used to accelerate the rendering time of such scenes. Interval algorithms are used to exploit the coherence in space and time of the animation, improving the time obtained with traditional techniques.

Chapter 7 offers two improvements in efficiency for the algorithms presented in chapter 5. Chapter 7 starts with the description of an algorithm that works in parallel in many computers. The second part is the description of an algorithm working in GPU (Graphic Processing Unit) in which details for the construction of an interval library in GPU are provided.

This thesis concludes with a summary of the results of this thesis and future work. The appendix includes a extensive bibliography about the subject of this thesis.

1.5 Contributions of the research

This major contributions of this research to the state-of-the-art in the ray tracing of implicit surfaces are:

- A method to work with a set of rays simultaneously. In this way, the number of intersection tests required in a traditional ray tracing algorithm are reduced, and consequently, the the computational time is also reduced.
- An efficient and reliable method to trace shadow rays, which is less complex than other techniques like beam tracing for polygonized surfaces.
- An anti-aliasing algorithm, in which beams are used to scan the area of the pixel completely. Using this technique, it is possible to obtain better results than anti aliasing techniques based in point sampling.
- A method to exploit spatial coherence which works with beams and it is completely based on interval arithmetic.

- A method to exploit spatial and time coherence in the animation of scenes composed by implicit surfaces. Consecutive frames can be rendered using the same spatial structure and beam reducing the rendering time of the whole animation.
- A parallel version of the beam tracing of implicit surfaces running on a cluster of computers. Image space coherence is used to reduce the communication between computers.
- An algorithm working on GPU which includes a GPU-based interval arithmetic library where rounding was correctly implemented.

1.6 List of presentations related to this PhD

Visualization of Implicit Surfaces Using Quantified Set Inversion.

Authors:	Jorge Flórez, Pau Herrero, Miguel A. Sainz, Josep Vehí
Congress:	IntCP 2005 (www.epfl.ch/∼sam/IntCP05/)
Place:	Barcelona (Spain), 2005
Proceedings:	Proceedings of the conference

Improving the Interval Ray Tracing of Implicit Surfaces.

Authors:	Jorge Flórez, Mateu Sbert, Miguel A. Sainz, Josep Vehí
Congress:	Computer Graphics International
Place:	Hangzhou (China)
Proceedings:	Lecture Notes in Computer Science. Vol. 4035, Pag. 655-664, 2006

Parallel Implementation of a Ray Tracing of Implicit Surfaces using Interval Arithmetic.

Authors:	Jorge Flórez, Mateu Sbert, Miguel A. Sainz, Josep Vehí
Congress:	GAMM-IMACS International Symposium on scientific computing, computer arithmetic and validated numerics
Place:	Duisburg (Germany), 2006

Guaranteed Adaptive Antialiasing using Interval Arithmetic.

Authors:	Jorge Flórez, Mateu Sbert, Miguel A. Sainz, Josep Vehí
Congress:	International Conference on Computational Science
Place:	Beijing (China)
Proceedings:	Lecture Notes in Computer Science. Vol. 4488, Pag. 166-169, 2007

Efficient Ray Tracing of Implicit Surfaces using Interval Arithmetic.

Authors:	Jorge Flórez, Mateu Sbert, Miguel A. Sainz, Josep Vehí
Congress:	Int. Conference on parallel Processing and applied Mathematics
Place:	Gdansk (Poland)
Proceedings:	Lecture Notes in Computer Science. Vol. 4967, Pag. 1351-1360. 2007

A GPU Interval Library based in Boost Interval.

Authors:	Sylvain Collange, Jorge Flórez, David Defour
Congress:	International Conference on Real Numbers and Computers
Place:	Santiago de Compostela (Spain)
Proceedings:	Proceedings of the conference. Pages: 61-72. 2008

Improved ray tracing of implicit surfaces using interval arithmetic.

Authors:	Jorge Flórez, Mateu Sbert, Josep Vehí
Workshop:	Small Workshop on Interval Methods
Place:	Montpellier (France), 2008

Improved Adaptive Anti aliasing for Ray Tracing Implicit Surfaces.

Authors:	Jorge Flórez, Mateu Sbert, Miguel A. Sainz, Josep Vehí
To appear:	International Journal of Computer Information Systems and Industrial Management Applications, 2008

Also, this work has been referenced in the following documents:

- **Interactive Ray Tracing of Arbitrary Implicits with SIMD Interval Arithmetic.** Aaron Knoll, Younis Hijazi, Charles Hansen, Ingo Wald, H, Hagen. *IEEE Symposium on Interactive Ray Tracing.* 2007.
- **Interactive Ray Tracing of Arbitrary Implicit Functions.** Aaron Knoll, Younis Hijazi, Charles Hansen, Ingo Wald, H, Hagen. *SCI Institute Technical Report, No. UUSCI-2007-002, University of Utah.* 2007.
- **Fast and Robust Ray Tracing of General Implicits on the GPU** Aaron Knoll, Younis Hijazi, Andrew Kensler, Mathias Schott, Charles Hansen, Hans Hagen. *University of Utah Technical Report, No.UUSCI-2007-014.* 2007.

2 Implicit surfaces

2.1 Introduction

Implicit surfaces are used in computer graphics for the modeling of geometric objects. They are useful for represent deformations and blending. They are also combined to generate complex models through Constructive Solid Geometry Operations. Although the mentioned uses of the implicit surfaces, its principal weakness is the amount of time required for its direct visualization (for example, using ray tracing [28]), and also the difficulty of controlling the shape of the surfaces during a quick visualization in an interactive environment [14, 80, 81].

For these reasons, they are not as popular as the parametric representations of the surfaces (for example, polygonal meshes), which can render models in a relatively small time.

But despite of the mentioned weaknesses, implicit surfaces are a flexible way to create complex models, as they offer a great tractability of a set of points by means of simple equations. That is, they allow to classify points to be inside, outside or in a surface defined by an implicit function.

They are also used in the representation of point data, for example, in imaging medical data and reconstructing objects represented by means of sets of points [11, 98, 79, 73].

In this chapter, the mathematical definition of an implicit surface is introduced. Moreover, the different methods to represent and the techniques to visualize implicit surfaces are presented.

2.2 Definition

An implicit surface is the set of solutions of an equation defined by:

$$f(x, y, z) = 0 \tag{2.1}$$

where $f : \Omega \subseteq \mathbf{R}^3 \rightarrow \mathbf{R}$.

Besides, it is possible to classify points "inside" or "outside". Indeed, only the sign must be verified [15, 94], that is, positive results indicate points outside the surface, negative ones indicate points inside. A result equal to zero indicates points in the surface (see figure 2.1).

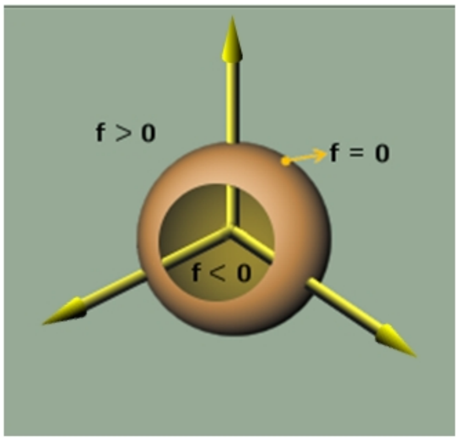

Figure 2.1: An implicit function can be used to define points to be inside or outside the surface.

To apply lighting effects, the normal of the surface must be calculated. The angle between the normal at a point of the surface, and a vector from that point pointing to the light, indicates how much illumination must be applied to that point.

The normal at a point ($\mathbf{p}$) of the surface is defined by means of the gradient of the function:

$$n = \nabla f(\mathbf{p}) = \left(\frac{\partial f(\mathbf{p})}{\partial x}, \frac{\partial f(\mathbf{p})}{\partial y}, \frac{\partial f(\mathbf{p})}{\partial z} \right) \tag{2.2}$$

The gradient point outwards the surface. Negating the surface will invert the normals.

In the case of non-differentiable functions, the gradient can be approximated by the following function:

$$n = \nabla f(\mathbf{p}) = \left(\frac{f(\mathbf{p} + \triangle x) - f(\mathbf{p})}{\triangle x}, \frac{f(\mathbf{p} + \triangle y) - f(\mathbf{p})}{\triangle y}, \frac{f(\mathbf{p} + \triangle z) - f(\mathbf{p})}{\triangle z} \right) \tag{2.3}$$

In this research, this kind of approximation is only used for illumination purposes.

2.3 Illumination model

In this section, the Phong illumination model is explained. This model is one of the most used in computer graphics, and will be applied in all the images generated in this thesis. The vectors involved in the process are: $\overrightarrow{e}$, vector from the point to the eye; $\overrightarrow{l}$, vector to the light; $\overrightarrow{n}$, the surface normal; $\overrightarrow{r}$, the reflection ray (see figure 2.2).

According to this illumination model, the color of a point in the surface must be proportional to the cosine of the angle between the direction of the light $\overrightarrow{l}$ and the normal at the point $\overrightarrow{n}$:

$$c = c_0 \, max(0, \overrightarrow{n} \cdot \overrightarrow{l})$$

in which the *max* function is used to catch those cases in which the dot product is negative (the angle is bigger that 90^o). In those cases, it is supposed that the light does not arrive to the point.

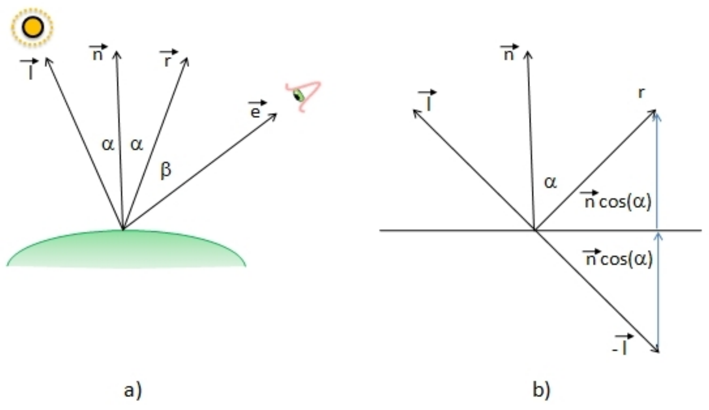

Figure 2.2: Geometry of the Phong illumination model.

The intensity of the light source and the reflectance of the surface can affect the illumination at the point. Thus, the terms c_r and c_l are included:

$$c = c_r\, c_l\, max(0, \overrightarrow{n} \cdot \overrightarrow{l})$$

in which c_r is the *diffuse reflectance*, that indicates the fraction of light reflected by the surface, and c_l is the light intensity.

In a real scene, there is some light surrounding all the objects. It is a common practice to add a term c_a to represent this light:

$$c = c_r\ (c_a + c_l\, max(0, \overrightarrow{n} \cdot \overrightarrow{l})) \tag{2.4}$$

It is also possible to add highlights to the surfaces. The idea is to add more illumination to the points in which the reflection ray $\overrightarrow{r}$ and the vector from the point to the eye $\overrightarrow{e}$ have a small angle (see figure 2.2a).

To obtain a maximum illumination when $\overrightarrow{r} = \overrightarrow{e}$, and that gradually falls when the angle between these vectors grows, the dot product between $\overrightarrow{e}$ and $\overrightarrow{r}$ must be used. That is:

$$c_h = max(0, \overrightarrow{e} \cdot \overrightarrow{r})^p \tag{2.5}$$

in which c_h is a term that indicates the fraction of the spot of the highlight corresponding to the current point. The positive term p is used to control the size of the spot. The *maximum* function is used to catch those cases in which the dot product is negative.

The vector $\overrightarrow{r}$ corresponds to the vector $\overrightarrow{l}$ reflected about $\overrightarrow{n}$ (show figure 2.2b). This vector can be calculated as:

$$\overrightarrow{r} = \overrightarrow{-l} + 2(\overrightarrow{l} \cdot \overrightarrow{n})\overrightarrow{n}.$$

In this model, 2.4 and 2.5 indicate the illumination values corresponding to the diffuse reflectance and the highlights. Those values should be added to obtain the illumination received by the eye (c) in the current point:

$$c = c_r \ (c_a + c_l \, max(0, \overrightarrow{n} \cdot \overrightarrow{l})) + max(0, \overrightarrow{e} \cdot \overrightarrow{r})^p \tag{2.6}$$

2.4 Representation of the Implicit Surfaces

This section describes two of the most common representations: algebraic surfaces and blob surfaces.

2.4.1 Algebraic surfaces

These surfaces are defined by polynomials whose degree indicates the degree of the algebraic surface. The degree indicates the number of intersections between the surface and a line [15]. For example, a plane has degree 1 while a sphere has degree 2. The use of polynomials has the advantage of being less expensive (in rendering time) than any general analytical representation.

The most known algebraic surfaces are the quadrics (polynomials of degree two). Those surfaces are easily rendered with few parameters to control its shape, and also, it is possible to use homogeneous coordinates to apply affine transformations. Typical quadric surfaces are the sphere, cylinder and cone (figure 2.3).

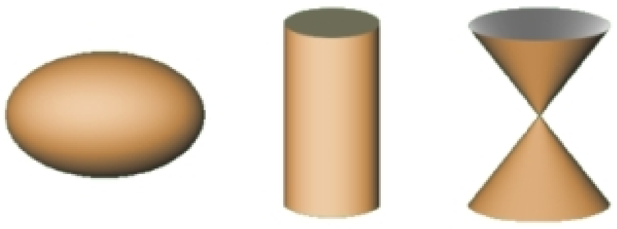

Figure 2.3: Classical quadrics surfaces: ellipsoid, cylinder and cone

2.4.2 Blobs

The blobs are the sum of gaussian distributions inspired in the density distribution of molecules. These surfaces were used for the first time by Blinn [12] to render an animation of the DNA for the Cosmos series by Carl Sagan, in which each atom was approximated by a gaussian field.

The sum of Gaussian fields generates a blend among the surfaces. Blinn proposed the function:

$$f(x, y, z) = \sum\nolimits_{i=1}^{n} b_i e^{-a_i r_i^2} - 1 \tag{2.7}$$

in which every function is centered at term r. The term r is calculated as $r_i(x,y,z) = \sqrt{(x-x_i)^2 + (y-y_i)^2 + (z-z_i)^2}$. The term b represents the height of the function, and a the standard deviation. The effect of the blob function can be changed adjusting those parameters. Figure 2.4 shows an example of a blob function. The exponential

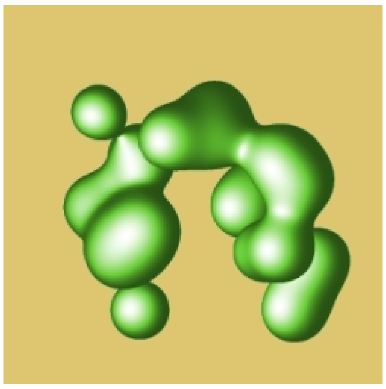

Figure 2.4: An example of the sum of gaussian fields

function defines a gaussian field that tends to infinity while the exponential tends to zero. That means that every gaussian field has influence over the others no matter the distance between them. Soft objects [104] computes field values using a polynomial approximation with a limited distance. Every field is generated by different objects like points, lines or boxes.

2.5 Visualization Methods

This section covers two of the visualization methods more frequently used and referenced in the bibliography about implicit surfaces: polygonization and ray tracing. Ray tracing is covered in more detail, as it is one of the main focuses of the present thesis.

2.5.1 Polygonization of Implicit Surfaces

This method consists in creating polygons that represent the implicit surface. Polygons are easily rendered in modern graphics systems, for that reason, it is the preferred method when interactive visualization is required.

In a polygonization, the intersections between surfaces and a cubic cell enclosing part of the surface are calculated to define the vertices (see figure 2.5). Those vertices have to be ordered to create convex polygons [14]. The quality of the result depends on the size of the cell, that is, if the cells are too large the details of the surface may be lost, and a small size could create too many polygons which make difficult the rendering process. It is possible to solve this problem applying adaptive methods in which the size of the cell depends on the detail of the surface. In this kind of algorithm some cracks can appear in surfaces between the shared faces of the nodes. In order to solve this, it is possible

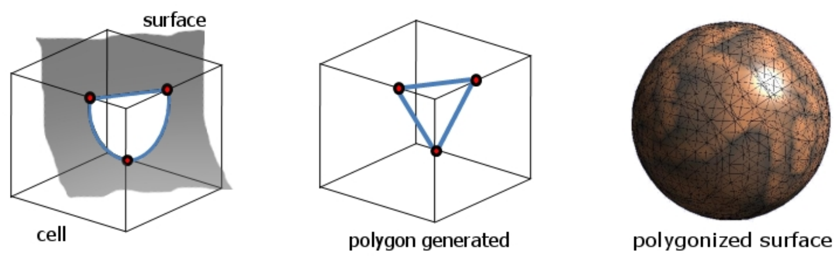

Figure 2.5: An example of the process to generate polygons in a cell. On the right, a polygonized sphere generated using BlobNetStudio 3.1

to replace the edges of the less subdivided cells with the edges of the smallest cells [9]. However, those methods are difficult to implement, for that reason, the methods with fixed cells are preferred [14].

2.5.2 Ray Tracing Implicit Surfaces

In 1980, Whitted [101] proposed a method to generate high quality images using basic geometric models. This method evolved in what is known as ray tracing, where the behavior of rays of light interacting with objects in the nature is simulated: the rays bouncing on the objects arrive to our eyes and we can see the features of the surface (color, transparency, etc.). Indirect light can also rebound from one surface to another and arrive to our eyes (see figure 2.6). Using this rendering technique, it is possible to obtain realistic visualization of different kinds of scenes. In nature, the light rays are emitted by the light sources. After bouncing at the objects, some of the rays arrive at the eye. It could be computationally expensive to try to simulate rays from the light source in the scene, as many of these rays never arrive at the eye. For that reason, it is better to model the inverse process, that is, to trace the rays from the eye and look for further intersections with the objects in the scene.

The Ray Tracing Algorithm

Ray tracing is a method that works pixel by pixel (pixel is short for “picture element”, that is, one element of the rectangular array in which most of the computer graphics images are presented). One or more rays are traced for every pixel in an image plane or screen. The objective is to find intersections among rays and objects. The idea is to check if one object is “seen” through a pixel. Generally, the first intersection with the first object is needed, because the others are occluded.

Figure 2.7 presents the process of constructing a ray. The point c represents the origin or view point. A ray starting at this point is sent through a pixel in screen (in the direction $\overrightarrow{cs}$) or image plane represented by the point s. These rays are referred to

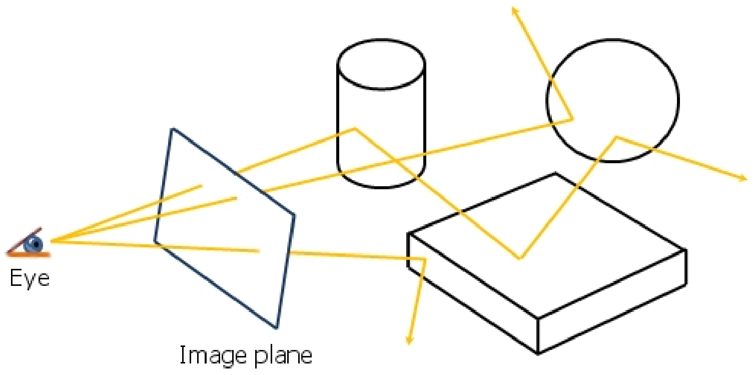

Figure 2.6: Light rays bouncing at different objects arriving at the eyes. The image plane in the figure is crossed by the rays; this plane can contain a representation of the 3D scene in 2D space.

uvw system, while the object has its own system *xyz*. This independence of view rays allows the scene to be viewed from any arbitrary position (in [84] there is an excellent explanation of the geometry of arbitrary view positions in ray tracing).

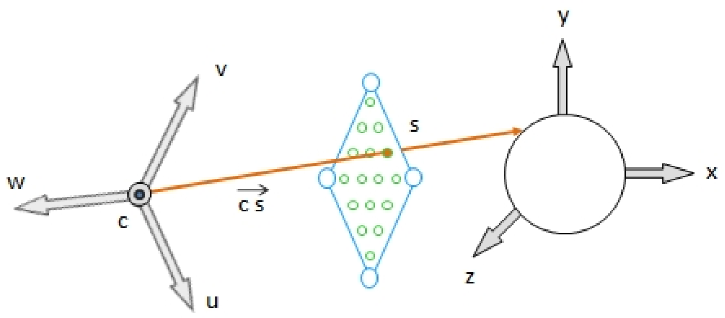

Figure 2.7: Definition of a ray crossing a screen. If the ray intersects any surface, the color is calculated and assigned to the pixel in the screen.

If there are intersections between the ray and an object behind the screen, the shading color is calculated taking into account the direction of the normal at the intersection point of the surface, and the position of the lights. The contribution of indirect light is also calculated. The sum of all the contributions is used to assign the shading color of the pixel.

Intersection tests

A ray is defined by:

$$\begin{aligned} x &= c_x + t(x_s - c_x), \\ y &= c_y + t(y_s - c_y), \\ z &= c_z + t(z_s - c_z) \end{aligned}$$

where (c_x, c_y, c_z) is the origin or view point; (x_s, y_s, z_s) is the point where the ray crosses the screen, and t is the parameter of the ray.

The intersection between an implicit function $f(x, y, z) = 0$ and a ray is defined for the equation:

$$g(t) = 0 \tag{2.8}$$

where $g(t)$ is the *intersection function* defined by:

$$g(t) \equiv f(c_x + t(x_s - c_x), c_y + t(y_s - c_y), c_z + t(z_s - c_z)) \tag{2.9}$$

So, the intersection test for a ray is to find whether the equation 2.8 has roots.

These solutions of t are replaced in the parametric ray to find the intersection point and the normal value.

There are many techniques to find the roots of the equation 2.8 like bisection or Newton method [85]. In [32], fuzzy techniques are used in the intersection test to classify the surfaces in certainty intersecting, certainty non intersecting and potentially intersecting. Interval arithmetic [65], Lipschitz constants [57], or even Corner and Taylor methods [36] can be used to create reliable intersection test.

Efficient Ray Tracing

Ray tracing algorithms lack efficiency. There are many works devoted to solve this problem. Dirk *et al.* performed a classification for the different approaches proposed in this subject [41]. According to this work, there are three major strategies to accelerate the ray tracing process: faster intersections, tracing fewer rays and generalized rays. The category of “faster intersections” covers the techniques that reduce the number of intersection tests performed between rays and surfaces. The majority of the techniques used to reduce the computation time of implicit surfaces are in this category.

The most basic technique to accelerate ray tracing is the bounding volume. In this technique, a volume which contains an object is created. The intersection test with the volume must be less expensive than the intersection with the object itself. The first proposed bounding volume was the sphere [101] because it has the more efficient shape to perform the intersection test. Every object in the scene was covered with a sphere; if any ray intersects the sphere, the ray is then tested for the objects inside it. This method can be improved by the use of a hierarchy of bounding volumes [77]. In this manner, the number of intersection tests is log(n) for n objects [15]. This kind of hierarchical structure is the most used when the bounding boxes are oriented parallelepipeds. In this case, the rays must be transformed to the space of the boxes to make less complicated intersection test.

Another technique is the space subdivision [41], in which the space surrounding the objects is subdivided, discarding the boxes without objects inside. The space can be subdivided in a regular grid (figure 2.8), in which a set of objects is associated with every box. The smaller box which contains objects is known as voxel. The boxes must be ordered, and the ray is tested for the boxes, looking for the first intersection. In this technique, it is important to traverse the ray, in which the output point of the ray from one box must be calculated to identify the next box which should be checked. This technique was first applied by Cleary *et al.* [24], and later by Fujimoto *et al.* [33]

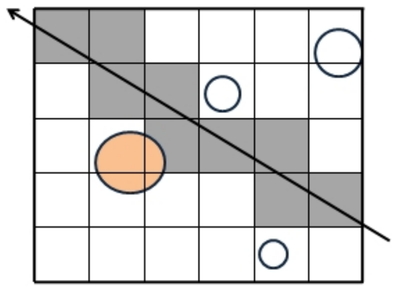

Figure 2.8: Regular Grid, with a ray traversing the cell. Objects must be checked for intersection.

Moreover it is possible to perform a nonuniform subdivision of space, to make finer subdivisions in regions with more objects. In [40] the subdivision is performed in eight parts, and each part is evaluated so as to detect intersection of the objects with any of the faces of the box; in this case, the object is added to a list for the current evaluated node. This technique is known as Octree (see figure 2.9). In [55], regular and adaptive subdivision methods are integrated. The voxels in a regular grid are recursively subdivided depending on object density.

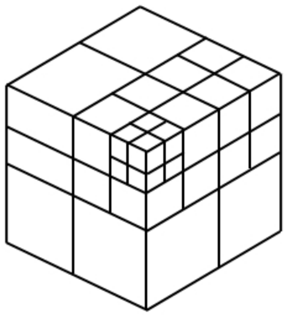

Figure 2.9: Octree subdivision

Although any z-buffer algorithm can be used to render it, the final voxels can be directly ray traced. Every voxel has its own information about position and normals, which facilitates the ray tracing process.

The main problem of this technique is that it requires high resolution (small size of voxels) to obtain good quality images, needing too much memory to store the voxels. Stolte *et al.* [91] proposed an algorithm to select only meaningful voxels and discard empty areas. The flexibility of this technique facilitates the representation of a model in cylindrical coordinates into a rectangular representation [92].

Other techniques deal with the direction of the rays. Those techniques are known as directional techniques. In the majority of these techniques, the scene is still subdivided in voxels but also the directions are discretized in regions called direction cells. Subsequently, the origin of the ray is enclosed in a cube with windows, and every window is checked in order to know which objects could be seen through it (figure 2.10).

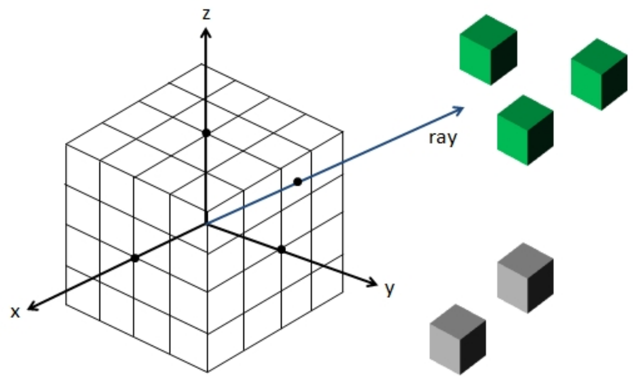

Figure 2.10: The ray is checked against the voxels that can be seen through the window crossed by the ray. In this case, the green voxels will be checked

This technique can be applied for the acceleration of primary rays [6], used in the light sources to accelerate the calculation of shadows [43], or used in the acceleration of general intersection calculation [69].

2.6 Conclusions

In this chapter a general introduction to implicit surfaces, was presented. The importance of implicit surfaces becomes clear when the different possibilities of application are known. In speed, implicit surfaces are not a challenge to surfaces represented by polygons, but many of their properties make them attractive. They have particular applications, like deformations and collision detection [15] or CSG models [72, 10], with more flexibility than parametric surfaces.

3 Interval Analysis

3.1 Introduction

Interval Analysis [67, 66] is a mathematical theory that deals with rounding problems that occur due to the use of floating point arithmetic. The computers have limited floating point registers to represent real numbers. However some real numbers have no finite representation, for that reason, those numbers have to be rounded. That situation causes problems of numerical imprecision that propagate inside the algorithms especially in the recursive ones. Although it is possible to work with a larger number of bits to represent the numbers, this set of numbers is, indeed, a representation of digital numbers that do not have the properties of the set of real numbers [86].

A naive example can demonstrate this problem:

```
a = random()
b = random()
c = a + b
c = c - a - b
```

It is expected that the variable c contains a value equal to zero, though that is not always the case when this short algorithm is tested in a desktop computer.

There are many research areas in which Interval Arithmetic has been applied to keep the numerical precision, like control engineering and supervision [5, 99] and geometric modeling and computer graphics [1, 75].

There are other well documented "catastrophic" examples that could be avoided by using Interval Arithmetic according to the interval community (www.math.psu.edu \ dna \ disasters). For example, a *Patriot* missile battery failed due to the accumulation of rounding errors. Another example is the explosion of the Arianne 5, caused by an overflow error.

In this chapter, the construction, operations and properties of Interval Arithmetic are introduced. There is also an introduction to Modal Interval Analysis which completes the definition of Classical Interval Analysis by means of the application of quantifiers to the intervals.

3.2 Definition

The numeric computation of theoretical problems in a computer requires that real numbers **R** are represented with a limited quantity of decimal places. It is possible to use a large number of decimal places, but still there are real numbers that can not be represented. This means that when a problem is represented from theory to the computer, we are indeed working with a set of digital number(DI), also called machine numbers or floating point numbers.

The real numbers provide a logic support to the models in which continuous magnitudes are used. However, the computer performs truncation of real numbers. It is possible that some part of a real value will be missed (see figure 3.1). The operations

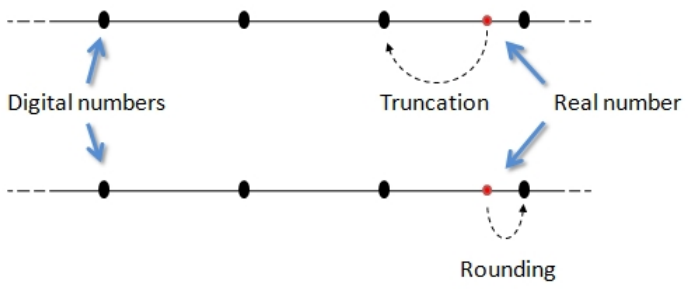

Figure 3.1: The image represents the selection of an upper bound for an interval in the set of digital numbers. If truncation is applied then the actual value is lost. A rounding to the next digital number is required to keep the value.

should be restricted to an interval, obtained by means of rounding, which gives operational identification to the calculated values.

Given two real values $\underline{a}$,$\overline{a}$, an interval A is defined as follows:

$$A = [\underline{a}, \overline{a}] := \{x \in \mathbf{R} \mid \underline{a} \leq x \leq \overline{a}\} \tag{3.1}$$

in which $\underline{a}$ and $\overline{a}$ are known as the *infimum* and the *supremum* of the interval respectively.

To keep the exact representation of the number, rounding must be performed to the next digital number, smaller than x (*infimum*), and to the next digital number, bigger than x (*supremum*). This rounding is performed to accomplish the definition of the intervals (equation 3.1). The use of Interval Arithmetic with digital numbers provides control for rounding errors automatically.

In the construction of the intervals (which are represented as I(**R**)), many of the properties of real numbers are lost (e.g. distributive law) and others like the inclusion relation are obtained.

Computers work with the set DI, for that reason, a set of intervals with bound in DI ($\mathrm{I}(DI)$) must be used for the algorithmic operations. This means that while $\mathrm{I}(DI)$ allows the operations over the set DI to be performed in the computer, an analytic model of the operations and relations of the intervals can be established with the set of classic intervals $\mathrm{I}(\mathbf{R})$.

As intervals are not real numbers anymore, the real algorithms can not be translated without some modifications into interval arithmetic. It should be mentioned that "new algorithms and new ideas have to be provided in order to take full advantage of Interval Arithmetic" [75]. Interval Arithmetic explains how to deal with the set $\mathrm{I}(\mathbf{R})$, defining properties and operations between them. The operations and relations in $\mathrm{I}(\mathbf{R})$ are easily defined between the bounds of the intervals.

3.3 Relations between Intervals

The relations between intervals are equivalent to some relations between the bounds. Although the definition of the relations is not evident in many cases, there is a general assumption of the most useful definition in every case. What is important is to obtain similar relations to the real numbers.

3.3.1 Equal operator

Definition:

$$A = B := (\forall a \in A)(\exists b \in B)(a = b),\ (\forall b \in B)(\exists a \in A)(b = a)$$

In function of the bounds:

$$A = B \Leftrightarrow \underline{a} = \underline{b},\ \overline{a} = \overline{b}$$

3.3.2 Less than

Definition:

$$A < B := (\forall a \in A)(\forall b \in B)(a < b)$$

In function of the bounds:

$$A < B \Leftrightarrow \overline{a} < \underline{b}$$

3.3.3 Less or Equal than

Definition:

$$A \leq B := (\forall a \in A)(\exists b \in B)(a \leq b),\ (\forall b \in B)(\exists a \in A)(a \leq b)$$

In function of the bounds:

$$A \leq B \Leftrightarrow \underline{a} \leq \underline{b},\ \overline{a} \leq \overline{b}$$

3.3.4 Inclusion

Definition:

$$A \subseteq B := (\forall a \in A)(a \in B)$$

In function of the bounds:

$$A \subseteq B \Leftrightarrow \underline{a} \geq \underline{b},\, \overline{a} \leq \overline{b}$$

3.3.5 Incidence

Definition:

$$A =_{\text{и}} B := (\exists a \in A)(\exists b \in B)(a = b)$$

The definition is equivalent to $A \cap B \neq \emptyset$.

In function of the bounds:

$$A =_{\text{и}} B \Leftrightarrow max(\underline{a}, \underline{b}) \leq min(\overline{a}, \overline{b})$$

3.4 Interval Arithmetic Operations

The Interval operations are based on the theory of sets. In this way, it is possible to define the operations by means of the bounds of the intervals.

There are some conditions that must be accomplished by interval operations:

- The result of an interval operation must be another interval.
- The constraint of an interval operation between particular intervals must coincide with the same operation between reals.
- All the operations between particular elements of both intervals must be contained in the interval result. This is known as the *inclusion principle of Interval Arithmetic*. This principle allows the theory of interval arithmetic to implemented into a machine arithmetic working with digital numbers.

According to these conditions, the operations are defined by the expression:

$$AwB = \{awb \;:\; a \in A,\; b \in B\}$$

in which w can be any of the operations: $+, -, *, /$.

The general equation of the Interval Arithmetic operations is:

$$AwB = [min(\underline{a}w\underline{b}, \underline{a}w\overline{b}, \overline{a}w\underline{b}, \overline{a}w\overline{b}), max(\underline{a}w\underline{b}, \underline{a}w\overline{b}, \overline{a}w\underline{b}, \overline{a}w\overline{b})] \tag{3.2}$$

According to definition 3.2, the basic four operations are defined as:

$$\begin{array}{lclcl}
[\underline{a}, \overline{a}] & + & [\underline{b}, \overline{b}] & = & [\underline{a} + \underline{b}, \overline{a} + \overline{b}] \\
[\underline{a}, \overline{a}] & - & [\underline{b}, \overline{b}] & = & [\underline{a} - \overline{b}, \overline{a} - \underline{b}] \\
[\underline{a}, \overline{a}] & * & [\underline{b}, \overline{b}] & = & [min(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b}), max(\underline{a}\underline{b}, \underline{a}\overline{b}, \overline{a}\underline{b}, \overline{a}\overline{b})] \\
[\underline{a}, \overline{a}] & / & [\underline{b}, \overline{b}] & = & [\underline{a}, \overline{a}] * \left[\frac{1}{\overline{b}}, \frac{1}{\underline{b}}\right] \;\; if\ 0 \notin [\underline{b}, \overline{b}]
\end{array}$$

3.5 Modal Intervals

Modal Interval Analysis (MIA) [35] is a logical completion of classic interval analysis that includes tools for solving quantified uncertainty. To accomplish that objective, the classic interval is associated with a quantifier $(\forall, \exists)$.

A modal interval is defined as a pair (I, Q) where "I" is a classic interval and "Q" is a modal quantifier, either universal $(\forall)$ or existential $(\exists)$. The set of the modal intervals is represented by $I^*(\mathbf{R})$. If the modal interval is associated with an existential quantifier it is called "proper" interval. In the case where the interval is associated with a universal quantifier, it is called "improper". The canonical representation of a modal interval is:

- Proper interval: $X = [a, b] = ([a, b]', \exists)$ if $a \leq b$
- Improper interval: $X = [a, b] = ([b, a]', \forall)$ if $a \geq b$
- Point-wise interval: $X = [a, b] = ([a, b]', \{\exists, \forall\})$ if $a = b$

in which the quotation mark in $[a, b]'$ indicates a classic interval.

A point-wise interval can have universal or existential quantifier, that is, the interval can be considered as proper or improper.

The process of construction of modal intervals is completed with the concept of modal quantifier Q defined by:

$$Q\,(x, X)\,P\,(x) :\Leftrightarrow \begin{cases} (\exists x \in X')P(x) & \text{if } X = (X', \exists) \\ (\forall x \in X')P(x) & \text{if } X = (X', \forall) \end{cases}$$

which defines the set of real predicates accepted by a modal interval $A = (A', Q_A)$:

$$Pred((A', Q_A)) := \{P(.) \in Pred(\mathbb{R}) \mid Q(x, (A', Q_A))\ P(x)\}.$$

This definition of the modal quantifier Q compels a change to the classical notation for the quantifiers. Hereafter,

$$\exists(x, X') \text{ will be used instead of } (\exists x \in X')$$

$$\forall(x, X') \text{ will be used instead of } (\forall x \in X')$$

Through the identification of a modal interval with the set of those real predicates which it accepts: $X \leftrightarrow P(X)$ arises the inclusion of two intervals as the inclusion of the set of predicates that they accept, that is to say, if $X, Y \in I^*(\mathbb{R})$

$$X \subseteq Y :\Leftrightarrow Pred(X) \subseteq Pred(Y)$$

Using their canonical coordinates $X = [x_1, x_2]$ and $Y = [y_1, y_2]$, this inclusion maintains the traditional *modus operandi*; that is to say,

$$[x_1, x_2] \subseteq [y_1, y_2] \Leftrightarrow (x_1 \geq y_1, x_2 \leq y_2) .$$

Figure 3.2 shows the geometrical representations of modal intervals and the inclusion relation.

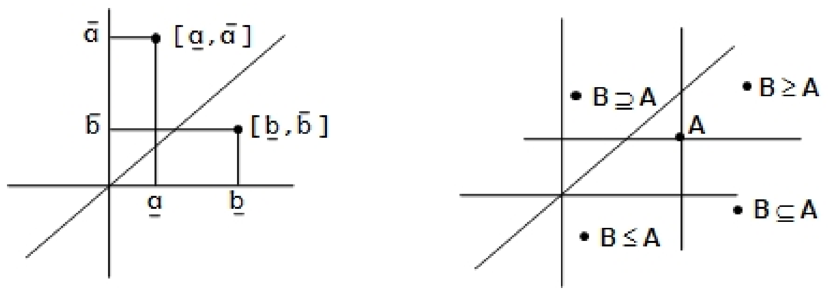

Figure 3.2: a) Geometrical representation of modal intervals. b) Inclusions and inequalities.

The lattice operations “meet” and “join” on $I^*(\mathbb{R})$ for a bounded family of modal intervals $A(I) := \{A(i) = [a_1(i), a_2(i)] \in I^*(\mathbb{R}) \mid i \in I\}$ (I is the index’s domain) are defined by

$$\wedge(i, I)\ A(i) = A \in I^*(\mathbb{R}) \text{ is such that } \forall(i, I)\ X \subseteq A(i) \Leftrightarrow X \subseteq A,$$
$$\vee(i, I)\ A(i) = B \in I^*(\mathbb{R}) \text{ is such that } \forall(i, I)\ X \supseteq A(i) \Leftrightarrow X \supseteq B,$$

annotated $(A \wedge B)$ and $(A \vee B)$ for the corresponding two-operands’ case. The result, as function of the interval bounds, is

$$\bigwedge_{i \in I} A(i) = [\max_{i \in I} a_1(i), \min_{i \in I} a_2(i)]$$
$$\bigvee_{i \in I} A(i) = [\min_{i \in I} a_1(i), \max_{i \in I} a_2(i)]$$

With these operations the set of modal intervals is a reticle for this $\subseteq$-relation, while the classic intervals are not, therefore, modal intervals are a reticular completion of the set of classic intervals. Both operators are isotonic, i.e., if $A_i \subseteq B_i$ for every $i \in I$, then

$$\bigwedge_{i \in I} A_i \subseteq \bigwedge_{i \in I} B_i \qquad \text{and} \qquad \bigvee_{i \in I} A_i \subseteq \bigvee_{i \in I} B_i$$

In the set of the real numbers there are two relationships: $\leq$ and $\geq$ and the extension of these relationships to intervals is defined by

$$[x_1, x_2] \leq [y_1, y_2] :\Leftrightarrow (x_1 \leq y_1, x_2 \leq y_2) \,.$$

which leads to the lattice operators “min” and “max”: for a bounded family of modal intervals $A(I) := \{A(i) \in I^*(\mathbb{R}) \mid i \in I\}$

$$\min_{i\in I} \ A(i) = A \in I^*(\mathbb{R}) \text{ is such that } \forall(i, I) \ X \leq A(i) \Leftrightarrow X \leq A;$$
$$\max_{i\in I} \ A(i) = B \in I^*(\mathbb{R}) \text{ is such that } \forall(i, I) \ X \geq A(i) \Leftrightarrow X \geq B.$$

and computationally

$$\min_{i\in I} A(i) = [\min_{i\in I} a_1(i), \min_{i\in I} a_2(i)]$$
$$\max_{i\in I} A(i) = [\max_{i\in I} a_1(i), \max_{i\in I} a_2(i)].$$

The set of the modal intervals is also a reticle for this $\leq$-relation. Figure 3.3 shows geometrical representations of the meet, join, min and max operators for two intervals.

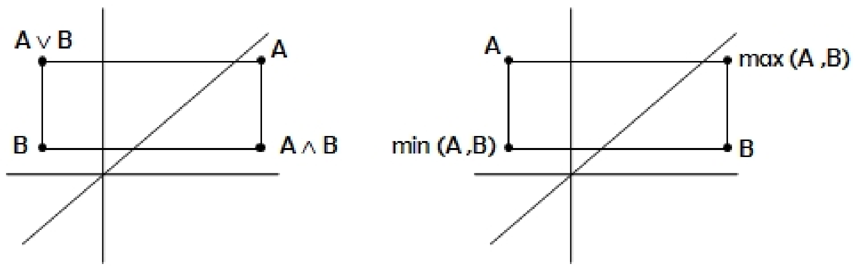

Figure 3.3: Meet, join, max and min lattice operators

3.5.1 Semantic extensions

In the classic set-theoretical interval analysis, one extension of a $\mathbf{R}^n$ to $\mathbf{R}$ continuous function $z = f(x_1, ..., x_n)$ is the *interval united extension* R_f of f. For the interval argument $X' = (X'_1, ..., X'_n) \in I(\mathbb{R}^n)$ it is defined as the range of f-values on X'

$$\begin{aligned} R_f(X'_1, \ldots, X'_n) \ &:= \{f(x_1, \ldots, x_n) \mid x_1 \in X'_1, \ldots, x_n \in X'_n\} \\ &= [\min\{f(x_1, \ldots, x_n) \mid x_1 \in X'_1, \ldots, x_n \in X'_n\}, \\ &\quad \max\{f(x_1, \ldots, x_n) \mid x_1 \in X'_1, \ldots, x_n \in X'_n\}] \end{aligned}$$

To obtain the estimates for the united extension, the set-theoretical interval rational extensions $fR(X'_1, \ldots, X'_n)$ are defined like their corresponding real-rational functions $f(x_1, \ldots, x_n)$ replacing

1) their numerical arguments $x_1, \ldots, x_n$ by the interval arguments $X'_1, \ldots, X'_n$ and

2) their "real" arithmetic operators ω by their corresponding interval operations which, in the common case of the truncated computations of any actual arithmetic, must be the outwards directed ω^R because of the inclusion

$$X'\omega Y' \subseteq X'\omega^R Y' := Out(X'\omega Y').$$

where Out represents the outer rounding of the interval $X'\omega Y'$

Rational interval functions have the property, fundamental to the whole body of Interval Analysis, of being "inclusive", that is, for $X'_1 \subseteq Y'_1, \ldots, X'_n \subseteq Y'_n$ the relation

$$fR(X'_1, \ldots, X'_n) \subseteq fR(Y'_1, \ldots, Y'_n)$$

holds, assuming that no division by intervals, which contains zero, occurs.

The relation between both extensions is

$$R_f(X'_1, \ldots, X'_n) \subseteq fR(X'_1, \ldots, X'_n),$$

where $fR(X'_1, \ldots, X'_n)$, is computable from the bounds of the intervals $X'_1, \ldots, X'_n$, and usually represents an overestimation of $R_f(X'_1, \ldots, X'_n)$.

In Modal Interval Analysis, the similar role to R_f is played by the semantic $*$ and $**$-functions, denoted by f^* and f^{**} (star and double-star functions), and defined by

$$\begin{aligned} f^*(X) &:= \bigvee_{x_p \in X'_p} \bigwedge_{x_i \in X'_i} [f(x_p, x_i), f(x_p, x_i)] = \\ &= [\min_{x_p \in X'_p} \max_{x_i \in X'_i} f(x_p, x_i), \max_{x_p \in X'_p} \min_{x_i \in X'_i} f(x_p, x_i)] \end{aligned}$$

and

$$\begin{aligned} f^{**}(X) &:= \bigwedge_{x_i \in X'_i} \bigvee_{x_p \in X'_p} [f(x_p, x_i), f(x_p, x_i)] = \\ &= [\max_{x_i \in X'_i} \min_{x_p \in X'_p} f(x_p, x_i), \min_{x_i \in X'_i} \max_{x_p \in X'_p} f(x_p, x_i)] \end{aligned}$$

which have the property of inclusion $f^*(X) \subseteq f^{**}(X)$. Also, $X \subseteq Y \Rightarrow (f^*(X) \subseteq f^*(Y), f^{**}(X) \subseteq f^{**}(Y))$.

In some cases, it may occur that $f^* = f^{**}$. Classic examples are the arithmetic operators, which according to the previous definitions, can be calculated by means of operations between the bounds of the intervals [86].

The following semantic theorem, give logical interpretation to the semantic extensions.

$*$-semantic theorem: Let $X \in I^*(\mathbf{R}^n)$ and $Z \in I^*(\mathbf{R})$, then

$$f^*(X) \subseteq Z \Leftrightarrow \forall(x_p, X'_p)\ Q(z, Z)\ \exists(x_i, X'_i)\ z = f(x_p, x_i)$$

$**$-semantic theorem: Let be $X \in I^*(\mathbf{R}^n)$ and $Z \in I^*(\mathbf{R})$, then

$$f^{**}(X) \supseteq Z \;\Leftrightarrow\; \forall(x_i, X_i')\; Q(z, Dual(Z))\; \exists(x_p, X_p')\; z = f(x_p, x_i)$$

This means that it is possible to reduce a logical expression into interval inclusions. Both semantic theorems make equivalent a logical formula, with intervals and functional predicates where the universal quantifiers precede the existential ones, to an interval inclusion.

For example, for the real function $f(x, y) = x + y$, having $X = [1, 3]$ and $Y = [3, 2]$, the result is $[1, 3] + [3, 2] = [4, 5]$. According to $*$-semantic theorem we obtain:

$$\forall(x, [1, 3]')\exists(z, [4, 5]')\exists(y, [2, 3]')x + y = z$$

and according to $**$-semantic theorem we obtain:

$$\forall(y, [2, 3]')\forall(z, [4, 5]')\exists(x, [1, 3]')x + y = z$$

.

Even though functions f^* and f^{**} are optimal in the semantic sense, these theorems do not explicit the process of computation of the interval Z which fulfills $f^* \subseteq Z$ or $f^{**} \supseteq Z$, i.e., intervals which are an outer estimate of f^* and an inner estimate of f^{**}. In fact, excepting the arithmetic operators, in general the calculation of f^* and f^{**} can not be reached for any direct computation. If the continuous function f is a rational function, there exist modal rational extensions which are obtained by using the computing program defined by the syntax tree of the expression of the function: if f is a $\mathbb{R}^n$ to $\mathbb{R}$ rational function, its rational extension to the modal intervals $X_1, \ldots, X_n$, represented by $fR(X_1, \ldots, X_n)$, is the function fR from $I^*(\mathbb{R}^n)$ to $I^*(\mathbb{R})$ defined by the computational program indicated by the syntax of f when the real operators, supposed JM-commutable functions, are transformed into their semantic extensions. Modal rational interval functions are not interpretable but they also have the property of being isotonic, i.e., for $X_1 \subseteq Y_1, \ldots, X_n \subseteq Y_n$ the relation

$$fR(X_1, \ldots, X_n) \subseteq fR(Y_1, \ldots, Y_n)$$

holds assuming, that no division by intervals containing zero does occur.

3.5.2 Interpretability and optimality.

The solution to the problem of computing the semantic extensions f^* and f^{**} consists in relating them by means of inclusion relations to some rational extensions. Computations with $fR(X)$ must be done with external truncation of each operator to obtain inclusions $f^*(X) \subseteq fR(X)$, and with inner truncation to obtain inclusions $fR(X) \subseteq f^{**}(X)$. In many cases the rational extension $fR(X)$ is *optimal*, i.e.,

$$f^*(X) = fR(X) = f^{**}(X),$$

and, except rounding, both semantic theorems are applicable to the computed interval $fR(X)$ providing a logical meaning to it.

MIA provides a collection of results about inclusions or equalities which solve part of the double problem of interpretability of modal rational extensions and computability of semantic extensions. Important results about the interpretability of rational extensions are the following theorems:

Theorem 1. *-interpretability of modal rational functions: If the improper components of X are uni-incident in $fR(X)$, and if $Out(fR(Prop(X)))$ does exist, then

$$Out(fR(X)) \supseteq f^*(X),$$

where Out represents the outer rounding of the interval $fR(X)$.

Theorem 2. **-interpretability of modal rational functions: If the proper components of X are uni-incident in $fR(X)$, and if $Out(fR(Prop(X)))$ does exist, then

$$Inn(fR(X)) \subseteq f^{**}(X),$$

where Inn represents the inner rounding of the interval $fR(X)$.

A real function f is called *x-totally monotonous* for a multi-incident variable $x \in \mathbb{R}$ if it is uniformly monotonous for this variable and for each one of its incidences, considered as independent variables.

Theorem 3. *-interpretability with total monotony: Let X be an interval vector, and fR defined in the domain $Prop(X)$ and totally monotonous for a subset Z of multi-incident components. Let XDt^* be the enlarged vector of X, so that each incidence of every multi-incident component of the subset with total monotonicity is included in XDt^* as an independent component, but transformed into its dual if the corresponding incidence-point has a monotony-sense contrary to the global one of the corresponding Z-component; for the rest, the multi-incident improper components are transformed into point-wise intervals defined for any of their points. Then

$$f^*(X) \subseteq fR(XDt^*).$$

Theorem 4. **-interpretability with total monotony: Let X be an interval vector, and let fR be defined on the domain $Prop(X)$ and totally monotonous for a subset Z of its multi-incident components. Let XDt^{**} be the enlarged vector of X, such that each incidence of every multi-incident component of the subset with total monotonicity is included in XDt^{**} as an independent component. It should be transformed into its dual if the corresponding incidence-point has a monotony-sense contrary to the global one of the corresponding Z-component; for the rest, the multi-incident proper components are transformed into point-wise intervals defined for any of their points. Then,

$$fR(XDt^{**}) \subseteq f^{**}(X).$$

Theorem 5. Interpretability in multi-incidence case (general case): if $fR(X)$ has improper multi-incident components and Xt^* is obtained replacing those components by point-wise intervals defined by any of the points of their domains, then,

$$f^*(X) \subseteq fR(Xt^*).$$

This theorem is useful when it is not possible to perform monotonicity tests.

It is possible to obtain better results if the concept of tree-optimality is applied. A modal rational function $fR(X)$ is tree-optimal if, in its syntax tree, any of its non-uniformly monotonous operators is follow downwards only by one variable operators.

Theorem 6. Coercion to optimality: Let X, fR and XD be defined under the conditions of previous theorems 3 and 4, and let fR be tree-optimal on the domain Prop(X). In this case,

$$f^*(X) = fR(XD) = f^{**}(X).$$

This theorem is very useful for solving the problem especially in the case when the function involved in the logical formula verifies the optimality conditions because, in this case, the rational computation $fR(XD)$ is equal to $f^*(X)$, except rounding, and the *-semantic theorem makes it equivalent to the logical formula.

Example: Let us consider the continuous function f from $\mathbf{R}^2$ to $\mathbf{R}$ defined by $f(x, y) = \frac{xy}{x+y}$ with $X = [2, 3]$ and $Y = [4, 3]$.

The function f is totally monotonous respect to x and y, because $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ are bigger than zero for the domain of the variables. Taking the multi-incident components as independent components:

$$f(x_1, y_1, x_2, y_2) = \frac{x_1 y_1}{x_2 + y_2},$$

the partial derivatives are $\frac{\partial f}{\partial x_1} > 0$, $\frac{\partial f}{\partial x_2} < 0$, $\frac{\partial f}{\partial y_1} > 0$ and $\frac{\partial f}{\partial y_2} > 0$ for the domain of the variables.

With these conditions, we will show the application of three theorems:

1. According to Theorem 5, the improper multi-incident components are replaced by point-wise intervals, $Xt^* = ([2, 3], [4, 4], [2, 3], [4, 4])$. The result is

$$fR(Xt^*) = \frac{X\ [4, 4]}{X + [4, 4]} = [1.1428, 2].$$

2. According to Theorem 3, taking into account the monotonicity of the function presented in the last paragraph, $XDt^* = ([2, 3], [4, 4], [3, 2], [4, 4])$. In this case,

$$fR(XDt^*) = \frac{X\ [4, 4]}{Dual(X) + [4, 4]} = [1.3333, 1.7144].$$

3. According to Theorem 6, and taking into account the monotonicity of the function presented in the last paragraph, $XD = ([2,3],[4,3],[3,2],[3,4])$. The result is

$$f^*(X) = fR(XD) = \frac{XY}{Dual(X) + Dual(Y)} = [1.3333, 1.5].$$

The best approximation is obtained with $fR(XD)$, which is the theorem that better fits the monotonicity conditions of the function. The worst approximation is obtained for $fR(Xt^*)$. In this example, $f^*(X) = fR(XD) \subseteq fR(XDt^*) \subseteq fR(Xt^*)$.

3.6 Conclusions

In this chapter, the different properties and operations of Interval Analysis were introduced. As it was explained, the Interval Arithmetic can be applied in many areas of research to control rounding problems, and Computer Graphics is not an exception.

There is also an overview of Modal Interval Analysis, which completes the definition of the Classical Interval Analysis by means of the application of Quantifiers to the definition of Intervals. The Modal Interval theory gives meaning to the improper intervals; in classical intervals, an improper interval has no meaning, and in the most of the cases such intervals are converted to proper intervals without any logical explanation. Modal Interval Analysis gives an explanation for such cases by means of application of the different theorems included in this chapter. These theorems set the basis bricks to hold the theory developed and to improve the ray tracing of implicit surfaces (chapter four).

4 Applications of Interval Analysis to Implicit Surfaces - State of the Art

4.1 Introduction

As was explained in the previous chapter, Interval Analysis is a reliable tool that deals with rounding problems on computers. It has been used to solve problems in areas like structural assessment [34] and control [99]. There are also many researchers in the area of computer graphics working in different applications of Interval analysis like in the design of geometric models [1], in collision detection [87], to carry a description of an approximation error in the transfer of data between CAD/CAM systems [82], the ray tracing of parametric surfaces [4, 97] and in the ray tracing of implicit surfaces [65].

Interval Analysis has been used mainly in computer graphics for the creation of reliable subdivision algorithms. Those algorithms can evaluate areas or space to detect the "existence" of surfaces or curves. This allows the creation of structures like octrees or quadtrees in a reliable way. Moreover, the majority of these applications are devoted to implicit surfaces.

This chapter starts with an introduction of different interval subdivision algorithms used in computer graphics. The second section presents a survey of the different techniques applied to perform Reliable Ray Tracing, stressing on Interval Arithmetic (The improvement of those techniques is the main subject of this thesis). The last section also includes a comparison between the Interval approaches applied to find the roots during the intersection tests to know the efficiency in each case.

4.2 Recursive subdivision algorithms

There are many techniques to represent geometric objects, like polygonal representations, space subdivision techniques, bicubic parametric patches and constructive solid geometry [100]. The visualization process using interval analysis involves characteristics of more than one of those techniques, although the most used approach is the space subdivision. This is used to represent the object geometrically, and then apply any rendering strategy like z-buffer or ray tracing.

Space subdivision methods (like octrees) are intended to subdivide the space enclosing the object. In the octree method, if a cubic region is enclosing an object, and we cut down every face of the cube trough the middle, we will obtain eight new cubes. Each new cube (also called octant) must be evaluated to confirm if it contains any part of the object. This subdivision process continues until a small size of the octants or a predefined subdivision level is achieved. This process creates an octree structure with the spatial information of the object that can be used to create a direct visualization, or to accelerate other visualization techniques.

4.2.1 Space Subdivision using Interval Arithmetic

This technique can be applied either in the object space or in the viewing volume [93]. The subdivision algorithm performs recursive subdivisions in the object space to generate eight boxes called octants. An octant consists in a cubic region defined by three intervals, each one representing values for the bounds of the octant for every dimension (x,y,z). The octants that do not contain any point of the surface are discarded.

This process allows the generation of structures that describe the object geometrically. The creation of structures like octrees is straightforward. An octree is a structure that represents the occupancy of many objects in a three-dimensional scene (see figure 4.1). The two dimensional version of this structure is called quadtree. An octree is a tree in which the nodes represent regions and its leaves are smaller regions that contain part of the objects.

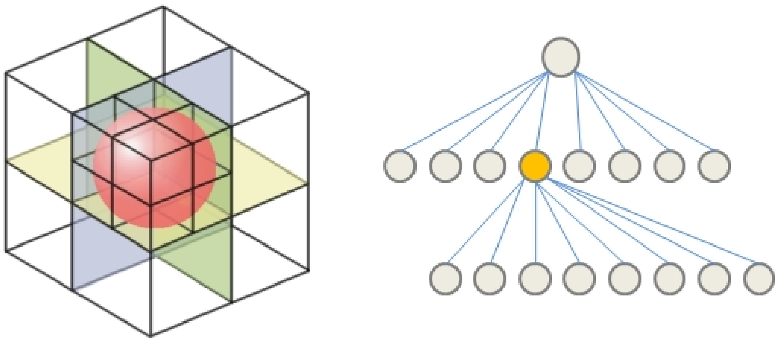

Figure 4.1: The recursive subdivision of the space into octants (left) is described by means of an octree (right).

Bloomenthal [13] used octrees to approximate and implicit surface with a polygonal representation. Stolte *et al.* [93] created an octree structure to store the voxels generated by the subdivision algorithm. In that work, every voxel contains the information about color and a normal vector, which facilitates the process of rendering.

It is possible to use this algorithm to represent operations between many implicit functions. Suffern *et al.* [94] introduced a technique to render the intersection of two implicit surfaces based on an octree. They used interval arithmetic to discard regions that do not contain one of both surfaces. If the region contains both surfaces it is further subdivided and the same analysis is made in the new two regions. In [54] a similar process is applied using parametric functions.

The octants that contain any part of the surface must be subdivided into another eight parts that have to be evaluated to know if they contain part of the solution set. These verifications are accomplished by means of an inclusion function using interval arithmetic.

Given an implicit surface defined by $f(x, y, z) = 0$, and a cubic region or box defined by three interval values X, Y, Z, to know if the surface intersects the box, the united extension (section 3.5.1) of f to the box (X,Y,Z),

$$F(X, Y, Z)$$

is used. The algorithm works as follows: if $0 \in F(X, Y, Z)$, the region may include part of the surface. Otherwise, the region can be definitively discarded.

The regions of space that contain part of the surface must be subdivided recursively and computed again until a determined size is achieved, which has to be selected according to the level or resolution required. At the end of the algorithm, a list of boxes that contain part of the surface are obtained, but part of these boxes might contain regions that are not solution of the problem. The subdivision algorithm using inclusion functions is illustrated in the figure 4.2.

This algorithm has many weak points as was exposed by Bühler [17]. The most relevant defect of this algorithm is related to the number of subdivisions required to arrive at a high resolution, which results in an expensive computing time.

Although the process is useful when it is used for three dimensional cases, the same principle can be applied for two dimensions [51, 70]. A 2D version of the algorithm can be used to rasterize algebraic curves [96, 51, 70, 61].

4.2.2 Improving the subdivision process

There are many techniques to improve the subdivision process. The principal objective of those techniques is to allow the design of fast and robust subdivision algorithms.

An adaptive version of the subdivision algorithm deals with the curvature of the surface during subdivision processes. Balsys *et al.* [9] developed an adaptive algorithm to improve the creation of an octree structure. This work solved the problem of cracks that occurred for differences in depth between adjacent nodes. In [70, 94, 52], the adaptive techniques are used to guarantee that subdivisions are small enough to contain parts of a curve with high curvature. This is performed by evaluating the gradient of the curve in every subdivision; a big gradient means that the curve changes so much inside the region, hence small boxes to represent the curvature have to be generated [51]. Carvalho *et al.* [20] provided some conditions to stop the recursive subdivision process only when the cells are small enough and do not contain closed loops.

Bühler [17] developed an Implicit Linear Interval Estimation (ILIE) which consists in a linear enclosure of an object adapted to its topology. Every cell is reduced to parts containing only the corresponding ILIE, which reduces the number of subdivisions for the entire process, decreasing thus the computational time.

Stander and Hart [89] showed how the critical points of a function affect the topology of an implicit surface. They used interval arithmetic to find those points, and afterwards, they modified the polygonization to accommodate the changes in the topology. Martin *et al.* [64] presented the comparison of many interval methods to plot algebraic curves. It is a very interesting work that compares the advantages and drawbacks of different interval methods applied to create subdivision structures in 2D.

Duff [30] developed an Interval application of constructive solid geometry using a tree in which the leaves are implicit functions. The algorithm takes into account the information of the whole tree in every subdivision, obtaining a complete scene with many objects correctly rendered in the same scene.

4.3 Reliable Ray Tracing of Implicit Surfaces

This section covers the improvements developed by many authors to perform a reliable ray tracing of implicit surfaces. These works are focused in the creation of reliable

```
Evaluate(intervals $X, Y, Z$) {
 If ($0 \in$ F(X,Y,Z)) {
     If ($X$ or $Y$ or $Z \leq$ threshold)
         add ($X$,$Y$,$Z$) to solution List
     Else
         subdivide $X$ into $X_1$ and $X_2$
         subdivide $Y$ into $Y_1$ and $Y_2$
         subdivide $Z$ into $Z_1$ and $Z_2$
         Evaluate($X_1$,$Y_1$,$Z_1$)
         Evaluate($X_1$,$Y_1$,$Z_2$)
         Evaluate($X_1$,$Y_2$,$Z_1$)
         Evaluate($X_1$,$Y_2$,$Z_2$)
         Evaluate($X_2$,$Y_1$,$Z_1$)
         Evaluate($X_2$,$Y_1$,$Z_2$)
         Evaluate($X_2$,$Y_2$,$Z_1$)
         Evaluate($X_2$,$Y_2$,$Z_2$)
     Endif
  Else
     // The octant is rejected
  Endif
```

Figure 4.2: Interval algorithm for the evaluation of the space occupied by an implicit surface.

intersection test. The authors applied different numerical techniques to control the loss of roots caused by the use of point sampling algorithms based in floating point arithmetic on the computers. This kind of problem causes that some thin parts of the surfaces disappear during the rendering process [18, 65]. Although those surfaces are indeed special cases, it is desirable to obtain a complete and reliable ray tracer, in which those problems will not appear during the visualization. Also, rounding problems cause some defective frames during the generation of animations [57].

Another problem of ray tracing is related with the efficiency of the algorithms. Whitted [101] reported that 95% of time is spent in the intersection tests for complex scenes. Ray tracing is slow because it requires many intersection tests for every pixel in the screen. This time is higher when implicit surfaces are rendered (because of the more expensive intersection tests), an even higher when Interval Arithmetic is used to keep the reliability in the intersection test.

4.3.1 Point Sampling Approach

The intersection test can be performed in two different steps: root finding, in which the first interval containing a root is selected, and root refinement, in which the interval having the root is reduced until a small size is achieved [12]. The key in the first step is to guarantee that the interval contains only one root for the implicit function. This step is usually more complicated and can be solved using fast methods, like classical bisection (figure 4.3a). However, this method can cause problems (figure 4.3b), like convergence to a wrong root or miss the roots.

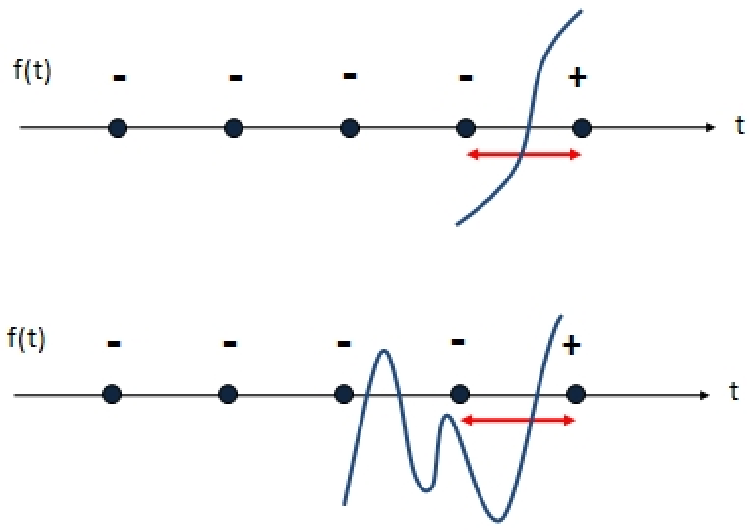

Figure 4.3: (Top) Root finding using a classical bisection. (Bottom) The algorithm could converge to another root or even miss the root.

In chapter 1, the general equation for the intersection test between a ray and an implicit surface was introduced (equation 2.9). Bisection is an algorithm that performs a binary search in the parameter of the ray (t) looking for roots. This is the most classical approach that can be applied in both root finding and root refinement (see figure 4.4).

```
Function Bisection($[t_1, t_2]$)
    $tm = (t_1 + t_2)/2$
    If $f(tm) = 0$
        $tm$ is a root
    Else
        If $(t_2 - t_1) < threshold$
            There are not roots
        Endif
        if $f(t_1) * f(tm) < 0$
            Bisection($t_1$,$tm$)
        Else
            Bisection($tm$,$t_2$)
        Endif
    Endif
```

Figure 4.4: Bisection algorithm.

An alternative to Bisection is the Newton Method. In this method the refinement is performed using the form: $t_{i+1} = t_i - \frac{f(t_i)}{f'(t_i)}$. The advantage of Newton method is that it converges quadratically, while Bisection converges linearly. However, Newton method may converge to any of the roots or in some cases diverge as was explained by Hart [47]. Bisection does not diverge, but may converge to any of the roots.

An example of the problem with the use of any method based in point sampling is presented in figure 4.5a&b. If the initial intervals are big (figure 4.5a) then many roots are lost. Using a smaller size in the intervals, the problem is not totally solved as can be seen in figure 4.5b. Moreover, small intervals mean that the algorithm lost efficiency (it has to evaluate more intervals). The problem is solved using a reliable approach like Interval Arithmetic (figure 4.5c&d). Even using a small precision in the bisection, the result is better (figure 4.5c). The result is correct when the bisection arrives to machine precision (figure 4.5d).

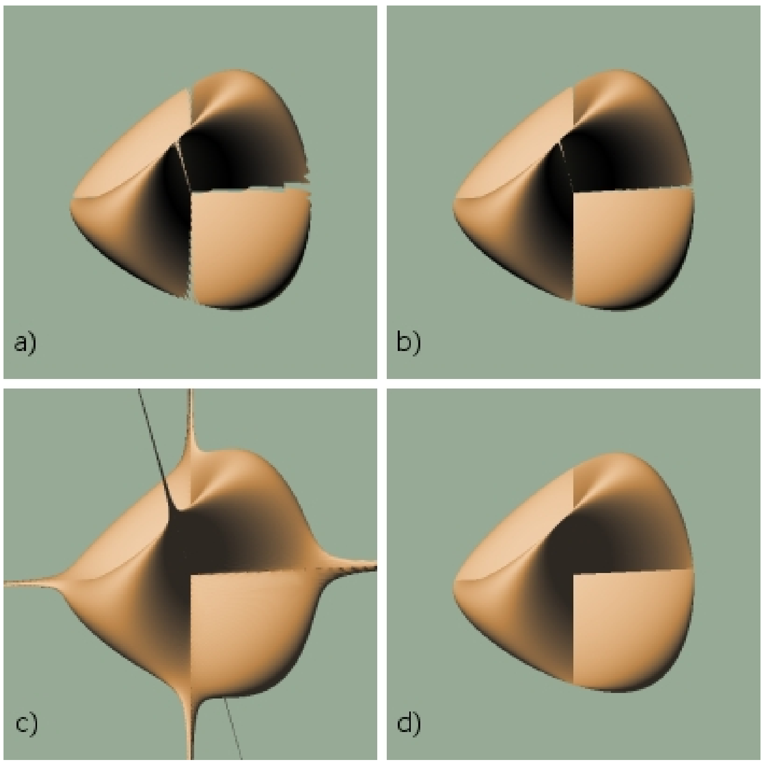

Figure 4.5: Images obtained with point sampling in root finding for a crosscap surface, using an interval size of 0.001 in a) and 0.0001 in b). Images c) and d) are obtained using Interval Arithmetic Bisection. In c) the subdivision arrives to 0.0001; in d) the bisection arrives to machine precision.

4.3.2 Interval Arithmetic Approach

Mitchell [65] proposed the use of Interval Arithmetic to perform the root finding in a reliable way. The united extension of the intersection function f is (see equation 2.9):

$$F(T) = f(c_x + T(x_s - s_x), c_y + T(y_s - s_y), c_z + T(z_s - s_z)) \tag{4.1}$$

The values c_x, c_y and c_z indicate the view point and the values x_s ,y_s and z_s indicate the point in which the ray crosses the screen, and T is an interval parameter. The difference with the classical definition is in the use of the interval variable T, which takes interval values instead of single real numbers.

Function 4.1 is known as "inclusion function" for the Implicit Surface. This function can be evaluated using any interval value of T. If the result of the evaluation is $0 \notin F(T)$, it is sure that there are no roots for the current value of T. However, this function can not be used to perform the opposite test, that is, to know if for any value of T then a root exist. This occurs because in every interval operation, the bounds of the interval are rounded up and down to guarantee that any possible result is not missed. This rounding includes values that are not part of the evaluation of the function, for that reason, the zero value could be included in one of the rounding operations. The inclusion function is then used only to perform rejection test. To know if there is only one root in the interval, then the united extension F' of the derivative f' is used. The algorithm proposed by Mitchell is presented in figure 4.6. Mitchell's algorithm starts performing the evaluation

```
Function Mitchell($T$ as $[t1, t2]$)
    If $0 \in F(T)$
        If $0 \notin F'([T])$
            If $f(t1) * f(t2) \leq 0$
                Root refinement over T
                  using Bisection or Newton method
            Endif
        else
            $T1$=$[t1,\ (t1+t2)/2]$
            $T2$=$[(t1+t2)/2,\ t2]$
            If width($T1$) $\geq$ threshold
                Mitchell($T1$)
            Else
                Root refinement over $T1$
                  using Bisection or Newton method
            Endif
            If width($T2$) $\geq$ threshold
                Mitchell($T2$)
            Else
                Root refinement over $T2$
                  using Bisection or Newton method
            Endif
        Endif
    Else
        reject $T$
    Endif
```

Figure 4.6: Mitchell's algorithm for root finding.

of an interval T using the inclusion function. In the case that zero is contained in the result, the derivative of the implicit function is evaluated using the interval value of T (this derivative can be another inclusion function). If zero does not belong to the result of the evaluation of the derivative, the signs of the evaluation of the implicit function in the bounds of the interval T are checked to see if they are different. If the conditions are satisfied, then it is sure that the interval T only contains one root, and can be used in a root refinement process. If the result of the evaluation of the inclusion function and also the derivative are both zero, the interval is bisected and the process must start over with the new intervals.

Another consideration is the width that the intervals can achieve before finishing the root finding and start the root refinement. For example, when the roots are in the tangent, the convergence will be always to these multiple roots. In that case, the derivative is zero in all the cases, and the algorithm only finishes when the small value allowed for an interval is achieved.

In the other cases, there is not mathematical guarantee that the small allowed interval contains roots. To solve that, the threshold can be set as machine precision, in which the probability of have roots is bigger [18].

Mitchell did not use rounding up and down in his interval operations (rounding is described in section 3.2). He considered that the use of floating point arithmetic in single precision was enough to render the surfaces he proposed. The reason Mitchell did not use this feature of Interval Arithmetic is that it increases the computational time in the rendering process.

Finally, this algorithm proposes to solve the root refinement using any classical point sampling method, like bisection, Newton or another fast enough method to render the surfaces. This is done because, according to Mitchell, Interval Arithmetic is slow to perform this process.

4.3.3 Improving the Interval Arithmetic Approach

Reliable techniques can be used in all the operations involved in the intersection test. If the root finding algorithm is performed using a classical interval bisection (the same algorithm of the previous section without derivative verification), the efficiency of the algorithm can decrease drastically.

Capriani *et al.* [18] show that there are interval algorithms for root finding, both reliable and fast enough to render an implicit surface.

The Interval Newton method is a reliable approach, which is faster than a classical interval bisection. Interval Newton method has often quadratical convergency [67], but requires derivatives. The Interval Newton method for root finding is presented in figure 4.7.

Capriani *et al.* proposed an important improvement of the Interval Newton method by means of an Alefeld-Hansen operator. The operator is obtained from the case when $0 \in F'(T)$. If $T = [t_1, t_2]$ it is possible to define [44, 2]:

$$\frac{1}{F'(T)} = \begin{cases} [1/t_2, \infty] & if\ t_1 = 0 \\ [-\infty, 1/t_1] & if\ t_2 = 0 \\ [-\infty, 1/t_1] \cup [t_2, \infty] & otherwise \end{cases}$$

Using extended arithmetic [45], the operator is the following:

$$AH(T) = \left[midpoint(T) - \frac{f(midpoint(T))}{F'(T)} \right] \cap T \tag{4.2}$$

The Alefeld-Hansen operator can be added to the Interval Newton method to test the cases where $0 \in F'(T)$. As appointed in [18], the result for the evaluation of this operator can be an empty set, a single interval or two intervals (T_1, T_2), in which $width(T_1)$, $width(T_2) \leq 1/2\, width(T)$. This means that the operator has quadratic convergence in many cases. When the operator returns two intervals, both of them must be used to evaluate the function recursively. This algorithm is presented in figure 4.8.

The previous algorithms require the use of the derivative of the function. This is a disadvantage when non-derivable functions have to be rendered. In those cases, a classical interval bisection is an alternative that works fine, although it is not efficient.

San Juan-Estrada *et al.* [78] proposed an algorithm to optimize the classical branch-and-bound strategy which is based on the classical Interval bisection algorithm. Basically, they propose a set of conditions that must be achieved during the bisection process. They stated that for an interval $[t_1, t_2]$, and its midpoint tm, if $\overline{F([t_1, t_1]) * F([tm, tm])} \leq 0$ then the interval $[tm, t_2]$ is rejected. This algorithm is called MRF.

```
Function Newton($T$ as $[t1, t2]$)
    If $0 \in F(T)$
        If $0 \notin F'([T])$
            $tm = t_1 + (t_1 + t_2)/2$
            $NT = tm - \frac{f(tm)}{F'(T)}$
            If $NT \cap T$ is empty
                There is no root
            Else
                Newton($NT \cap T$)
            Endif
        else
            $T1$=$[t1,\ (t1 + t2)/2]$
            $T2$=$[(t1 + t2)/2,\ t2]$
            If width($T1$) $\geq$ threshold
                Newton($T1$)
            Else
                $T1$ is the root
            Endif
            If width($T2$) $\geq$ threshold
                Newton($T2$)
            Else
                $T2$ is the root
            Endif
        Endif
    Else
        reject $T$
    Endif
```

Figure 4.7: Interval Newton method for root finding.

```
Function Alefeld-Hansen($T$ as $[t1, t2]$)
    If $0 \in F(T)$
        If $0 \notin F'([T])$
            $tm = t_1 + (t_1 + t_2)/2$
            $NT = tm - \frac{f(tm)}{F'(T)}$
            If $NT \cap T$ is empty
                There is not root
            else
                Alefeld-Hansen($NT \cap T$)
            Endif
        Else if $0 \notin f(tm)$
            $AH(T) = (tm - \frac{f(tm)}{F'(T)}) \cap T$
            If $AH(T) = \emptyset$
                There is no root
            Else
                If $AH(T)$ generates one interval T1
                    Alefeld-Hansen(T1)
                Else if $AH(T)$ generates two intervals T1,T2
                    Alefeld-Hansen(T1)
                    Alefeld-Hansen(T2)
                Endif
            Endif
        Else
            $T1$=$[t1,\ (t1 + t2)/2]$
            $T2$=$[(t1 + t2)/2,\ t2]$
            If width($T1$) $\geq$ threshold
                Alefeld-Hansen($T1$)
            Else
                $T1$ is the root
            Endif
            If width($T2$) $\geq$ threshold
                Alefeld-Hansen($T2$)
            Else
                $T2$ is the root
            Endif
        Endif
    Else
        reject $T$
    Endif
```

Figure 4.8: Interval Newton method with Alefeld-Hansen operator.

In another algorithm called $MRFro$ [78], they introduced some conditions to improve the bisection test. This algorithm evaluates $F[t_1, t_2]$ only when $\underline{F([t_1, t_1]) * F([t_2, t_2])} > 0$, because $\overline{F([t_1, t_1]) * F(t_2, t_2)} \leq 0$ assures that $0 \in F([t_1, t_2])$, which is not true if rounding is taken into account. Rounding can introduce values that can fulfill the equation, but this is not true for the original values considered in the interval. However, this assumption was enough for the surfaces rendered in the work presented by San Juan-Estrada *et al.*

4.3.4 Comparison of different Interval approaches

As can be seen in the previous sections, there are many different Interval approaches to guarantee the intersection tests in ray tracing implicit surfaces. This section covers a comparison of the efficiency of the methods. The surfaces used in the test are presented in figure 4.9. These surfaces were rendered by means of a ray casting method (only primary rays) using one ray per pixel.

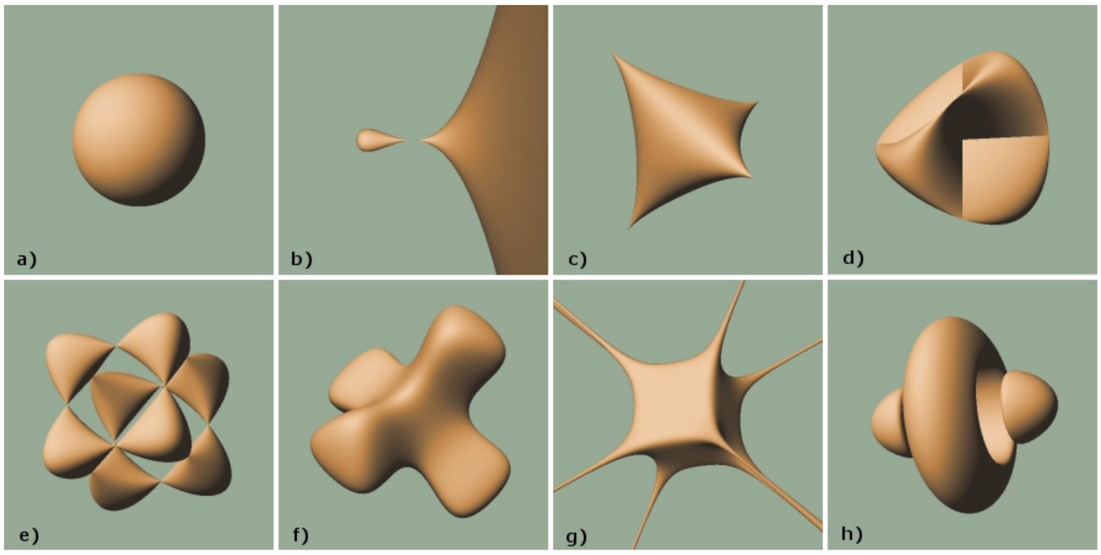

Figure 4.9: Surfaces used to compare the interval methods for ray tracing implicit surfaces. a) Sphere. b) Drop surface. c) Kusner-Shmitt. d) Crosscap. e) Chubs surface. f) McMullen K3 Model. g) Horned Cube. h) Gumdrop Torus.

Five methods to find roots were tested: the algorithm proposed by Mitchell, the Interval Newton method, the Interval Newton method using the Alefeld-Hansen operator, the $MRFro$ algorithm and an algorithm that combines the conditions of $MRFro$ algorithm and the Interval Newton method. The images have a resolution of 300x300 pixels, and were rendered in a Pentium 4 computer. The results are presented in table 4.1. In the methods that require the use of derivatives to evaluate the intersection test (Interval Newton method, and with Alefeld-Hansen operator), we use the derivative of the function in closed form to avoid possible precision errors.

	Sphere	Drop	Chubs	Crossc.	Gumd. Torus	H. cube	McMul. K3	Kusn.-Schm.
Mitchell	20	36	98	80	240	99	102	75
Newton	28	47	94	96	260	180	123	94
N.+Al.-Han.	27	46	92	99	202	166	110	86
MRFro	48	103	188	105	541	272	276	210
MRFro+Newt.	55	96	174	151	380	283	221	173

Table 4.1: Comparison of different Interval methods for the intersection test: 1. Mitchell algorithm, 2. Newton method, 3. Newton + Alefeld-Hansen, $MRFro$ and $MRFro$ + Newton method (time in seconds).

The time results are better for the sphere, and the worst times are for the Gumdrop torus for all the tested methods. The algorithm proposed by Mitchell is faster for a naive surface like the sphere, but when the complexity is high, the Newton+Alefeld-Hansen method achieve a better time. Also the Newton method is faster, but not as fast as the Newton + Alefeld-Hansen one. This means that the extra check performed in the Alefeld-Hansen operator is compensated with the better convergence obtained for complex surfaces. In the case of naive surfaces, it is better to apply a naive method (like Mitchell's one).

The $MRFro$ algorithm is the slower in all the cases. This occurs because this method does not use extra information of the surface, like the derivative. When extra information is added to $MRFro$ (like a Newton subdivision), the method improves the efficiency for complex surfaces like Gumdrop torus but not for naive ones like the sphere.

Figure 4.10 shows graphically the results. The surfaces were ordered intentionally from the less computationally expensive to the more expensive ones. This indicates a kind of complexity that increases from left to right. Also, a high time indicates less efficiency and viceversa. In this case it is possible to say that the figure represents a complexity vs. efficiency test, in which a point in the top-left of the graphic indicates less complexity of the implicit function and high efficiency of the method, and a bottom-right point indicates high complexity of the function and low efficiency in the method.

The Mitchell, Newton and Alefeld-Hansen methods have an efficiency with almost linear behavior when the complexity grows. The MRFro and MRFro-Newton algorithms tend to lose efficiency when the complexity is increasing. Also, the behavior in all the methods based in the derivative of the ray parameter (Mitchell, Newton,Newton+Alefeld-Hansen) is similar. It is remarkable that Alefeld-Hansen, and also Newton operators are as fast as the reliable version of Mitchell's algorithm which use intervals only for root finding. This means that it is possible to use interval operations in all the process (root finding and root refinement) without losing efficiency. Also, we have not found any visual difference in the different approaches for the tested models. This indicates that all the methods converge to the same first root during intersection test. Note that, as we will see in next chapter, this first root is not always correct.

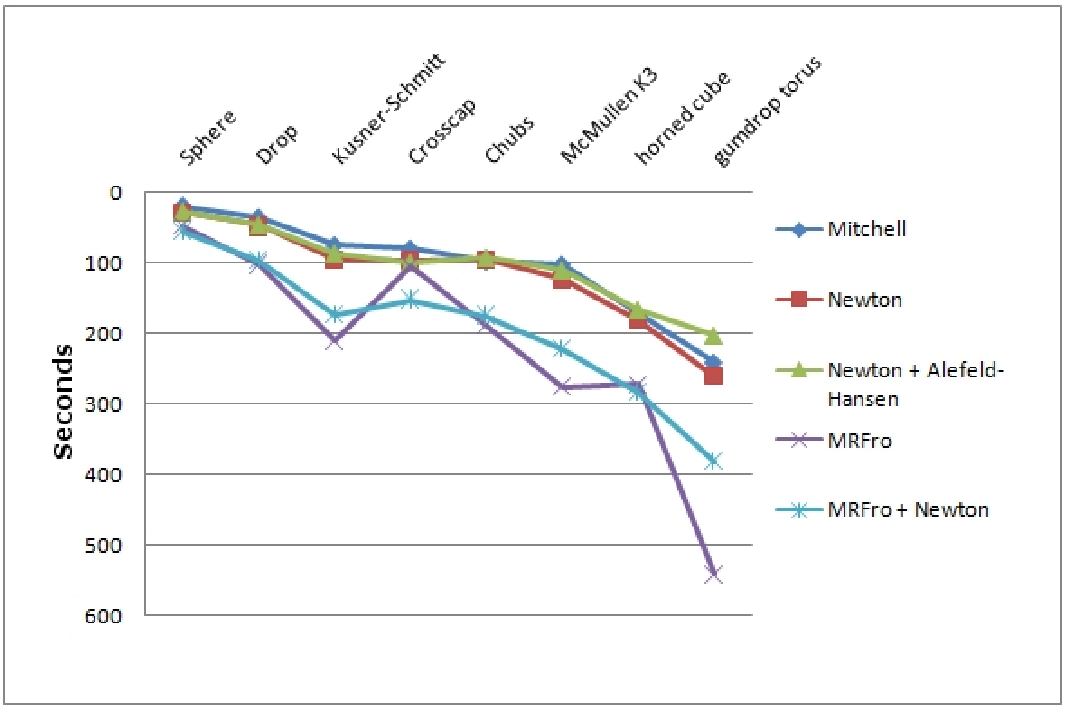

Figure 4.10: Results for the comparison of the Interval methods to ray tracing implicit surfaces for the different tested surfaces. Differences in quality are not noticeable for using the different methods.

4.3.5 Other reliable approaches

Lipschitz Constants based methods

Given an implicit function $h(t)$, if there exists a constant L such that for any values t_1, t_2 accomplish ([15]):

$$|h(t_1) - h(t_2)| < L|(t_1 - t_2)|$$

then L satisfies the Lipschitz condition and it is known as Lipschitz constant. If t_1 becomes too similar to t_2, then

$$\frac{|h(t_1) - h(t_2)|}{|(t_1 - t_2)|} < L$$

represents the derivative of the function, in which L measures the rate of change of the function between t_1 and t_2.

The L-G surfaces method [57] is based in the Lipschitz constants to develop a reliable intersection test in the ray tracing of implicit surfaces. The method works using two steps (using Lipschitz constant L in the first step and G in the second one, hence the name of the method). First, an octree structure is created for the implicit function $f(x)$, where $x \in \mathbf{R}^3$, for which the Lipschitz constant L exists:

$$\|f(x_1) - f(x_2)\| = L\|(x_1 - x_2)\| \tag{4.3}$$

The rate change of the function can be used to know which boxes contain part of the surface. Let x_0 be the center of a box, and d be the distance from the center to any vertex, if $|f(x_0)| > Ld$ then the box can be discarded. Otherwise, the box is subdivided in eight parts. Some boxes can be accepted or rejected checking the vertices (if some vertices are inside and some are outside then the box contains part of the surface) but some case are conflictive as is shown in figure 4.11.

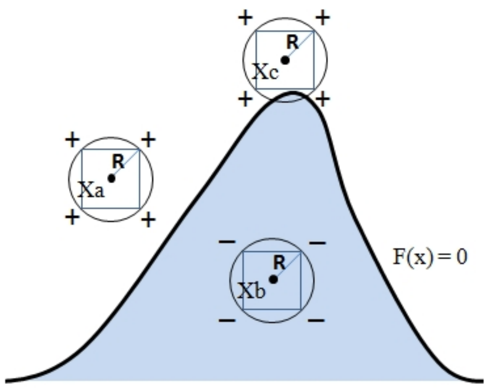

Figure 4.11: Spheres with center x_a and x_b do not intersect the surfaces because $f(x_a)/La < R$ and $f(x_b)/Lb < R$. Sphere with center xc may intersect the surface because $f(x_c)/Lc > R$

In a second step, the surface is ray traced using the structure created in the first step. A Lipschitz constant G that fulfills:

$$\|g(t_1) - g(t_2)\| = G\|(t_1 - t_2)\| \tag{4.4}$$

is used to know the rate of change of the function $g(t)$, which is the function used in the intersection test to find the roots for the parameter of the ray.

Let t_1 and t_2 be the entry and exit points of a ray in a voxel, $tm = (t_1 + t_2)/2$ and $d = (t_2 - t_1)/2$. If $|g(tm)| > Gd$ then there is only one root between t_1 and t_2. The algorithm for the intersection test is presented in figure 4.12 ([15]).

Another method based in Lipschitz constants is Sphere Tracing [48]. This method is similar to L-G Surfaces, but uses Lipschitz bounds instead of Lipschitz constants. A Lipschitz bound is any constant that satisfies the function, not always the smallest one. In this method, a Lipschitz bound for a function giving the distance to a surface, must be found. The Lipschitz bound for an implicit function $f(x)$ must fulfill: $|f(x)| < L * d(x, f^{-1}(0))$, in which $d(x, f^{-1}(0))$ is the signed distance bound.

Affine Arithmetic

De Cusatis *et al.* [27] introduced the use of affine arithmetic to ray casting implicit surfaces. Interval Arithmetic can generate overestimation in some interval operations, because boxes are considered in the evaluations of interval variables. Affine arithmetic is proposed to control the overestimation, because the approximation is performed over a parallelogram (figure 4.13). Affine arithmetic converges quadratically while interval bisection using Interval Arithmetic converges linearly.

In Affine Arithmetic, a quantity x is defined by the affine form:

$$\hat{x} = x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n$$

which is a polynomial of degree 1. The noise symbols ε_i are unknown but lies between -1 and 1. The same noise symbol may contribute in the uncertainty of two or more quantities. This indicates some partial dependency between the underlying quantities.

The algorithm to use Affine Arithmetic is basically the same than a classical bisection using Interval Arithmetic. The difference is the use of an affine Arithmetic Library instead of an interval one, in the calculation of united extensions. As happens with Interval Arithmetic, the basic arithmetic operations and functions can be extended to handle affine forms [90].

However, the improvements reported in the use of Affine Arithmetic in the ray casting of implicit surfaces [27] were against the basic algorithm proposed by Mitchell, but not against a faster method like Interval Newton method. Also, it has been proved that Affine Arithmetic is a special case of the centered form of Interval Arithmetic [36].

```
Intersection_Lipschitz($t1$,$t2$)
    Compute $G$ for $t1, t2$
    $t_m = (t1 + t2)/2$
    $d = (t2 - tm)/2$
    If $|g(t_m)| > Gd$
        If $F(t1) * F(t2) < 0$
            Compute using Newton method
        Else
            There is not intersection
        Endif
    else
        Intersection_Lipschitz($t1$,$t_m$)
        Intersection_Lipschitz($t_m$,$t2$)
    Endif
```

Figure 4.12: Intersection test using Lipschitz constants.

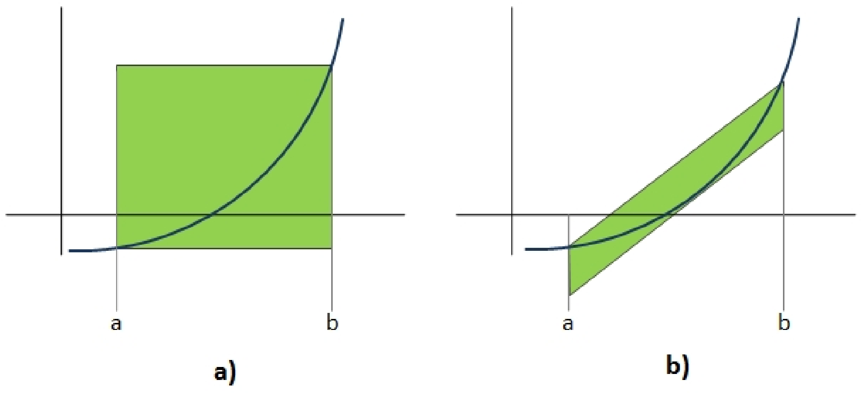

Figure 4.13: Approximation of a function in an interval [a,b]. a) Using Interval Arithmetic. b) Using Affine Arithmetic.

4.4 Conclusions

In this chapter, the different applications of Interval Arithmetic in the creation of subdivision structures and also in the creation of reliable intersection tests for ray tracing implicit surfaces were introduced. Interval Arithmetic provides an easy way to test the three axis of a box to know if it intersects the surface. This technique can be improved by means of an study of the surface to perform more subdivisions in critical section of it, like loops or parts with high curvature.

Moreover, different techniques to perform reliable intersection tests between rays and implicit surfaces were introduced. The majority of methods for reliable intersection test are intended to work with Interval Arithmetic, which replaces the operations over real numbers using interval operations. There are different approaches, for example, the based in the derivatives (like Newton methods) and also others that create some conditions to improve the bisection process. Although the use of derivatives improves the efficiency, a derivative can not be easily obtained for all kinds of implicit surfaces. A comparison was presented to know the behavior of different methods against a set of varying complexity surfaces.

There are other reliable techniques that do not use Interval Arithmetic. One of them is Affine Arithmetic, which is similar to Interval Arithmetic but implements some affine operations. A different approach is the use of Lipschitz constants, which are reliable enough to perform ray tracing of implicit surfaces. However, the main disadvantage of Lipschitz techniques is that they require the calculation of the Lipschitz constants for every surface to be considered. This loss of generality makes these methods less attractive when general ray tracers are required.

5 Beam tracing implicit surfaces

5.1 Introduction

In the previous chapter, different Interval approaches to create reliable intersection tests in the ray tracing of implicit surfaces were presented. When point sampling is used, some surfaces are not well rendered, that is, some parts of the surfaces do not appear in the final image. Ray tracing algorithms which work with Interval Arithmetic solve those problems, but unfortunately, they are not efficient, specially when they are compared with non-reliable algorithms based in floating point arithmetic.

According to Arvo and Kirk [41], ray tracing can be accelerated by: 1) the reduction of the average cost of the intersection test, or 2) the reduction of the total number of rays intersected, or 3) by replacing individual rays with groups of rays. Because the nature of Interval Arithmetic is to evaluate "intervals" which contain "set of values", the third kind of acceleration approach could be applied directly using the interval arithmetic. Here, the concept of ray coherence can be considered, that is, similar rays are likely to intersect the same objects in the scene.

This chapter shows a new method to exploit the coherence of rays to accelerate the process of ray tracing implicit surfaces. Interval theory can be adapted to study the behavior of many rays simultaneously, which means that areas of the image space can be studied to know the coherency of objects "seen through it".

Coherency is obtained by means of two algorithms called *rejection test* and *inclusion test*. They permit the identification of sets of rays in which either all the rays intersect the surface, or all the rays miss the surface. The results of these tests are used to create a new subdivision structure that improves the ray tracing process. Modal Interval theory (summarized in chapter 3) gives the set of theorems that support the algorithms used in this chapter.

The rejection and inclusion test are also used to create an efficient anti aliasing algorithm for ray tracing. This algorithm allows the creation of images with better quality than images obtained with classical interval techniques using less computational time.

5.2 Types of Coherence

There are many types of coherence [95], four of which are applied in ray tracing algorithms: object coherence, image coherence, frame coherence and ray coherence.

Object coherence occurs because objects in space tend to be connected or to be in close positions in space. All the techniques that subdivide the space of the objects take advantage of this type of coherence.

The subdivision methods have two different approaches: subdivision by objects, and subdivision by space [62]. In the first type, the scene is subdivided taking into account the position of the nearby objects, creating bounding volume hierarchies. In the second type of subdivision, the scene is subdivided in cells. Every cell is tested to know which objects cross it, and the intersection test is performed only in these objects. The subdivision can be either performed in cells with the same size or can be adapted according with the distribution of the objects in space.

The *image coherence* is based in the same concept of Object coherence, but taking into account that the coherence in this case is studied in the 2D image plane. That is, the same coherence of the 3D objects can be projected in the 2D plane.

The *frame coherence* is the image coherence with a temporal dimension. This means that during animation, successive frames tend to be similar in small changes of time. This requires not chaotic changes in eye and light positions [41].

Ray coherence occurs when similar rays have similar behavior. Rays starting in similar points and having similar directions tend to intersect the same objects. In this case, it should be more efficient to process groups of rays instead of individual rays. However, this is not always the case, due to the cost of calculating the intersection between groups of rays and objects is higher than the use of individual rays [62]. One of the most characteristics works using this type of coherence was performed by Speer *et al.* [88]. In this work, the ray tree generated for the first-generation ray is used in the construction of the secondary ray trees.

5.3 Generalized Rays

This section covers the different techniques used to take advantage of ray coherence. This kind of coherence is hard to exploit [41], although there are some algorithms that use it successfully. The idea is that similar rays can intersect the same objects, being possible to deal with many rays simultaneously (see figure 5.1).

These techniques are based in the generation of 3D structures that keep the information about the set of rays. Using those structures it is possible to trace the set of rays simultaneously. However, those techniques require some restrictions, for example, in the type of object to use [62] or in the use of different concepts to approximate the exact intersection test [41].

There are three main approaches: cone tracing [3], pencil tracing [83] and beam tracing [50].

In cone tracing, the generalized 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. The new virtual origin and angle is calculated taking into account the surface curvature. This technique treats the aliasing problems 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 that are covered by the objects, the method only deals with spheres, planes and polygons. This method also facilitates some light effects like the representation of penumbrae.

Another proposal is pencil tracing. In this case, the rays nearby to a special ray (called the axial ray) are grouped. These rays (called paraxial rays) are represented by a 4D matrix which 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 on pencils.

Finally, the beam tracing replaces individual rays with beams which consist in a set of rays with a common apex crossing planar polygons. This means that this method is restricted to objects with planar polygonal facets. When a beam intersects an object, a new beam with a new polygonal section is generated (see figure 5.2). The remainder of the beam can be complex forms that require a method that can operate with arbitrary polygons.

The reflected beams have a new virtual eye which is calculated using a linear transformation (see figure 5.3). This preserves the nature of the reflections as beams.

A beam tree to keep the information of reflected and refracted beams can be constructed in a similar way that standard ray tracing. In this case, a previous ordering of

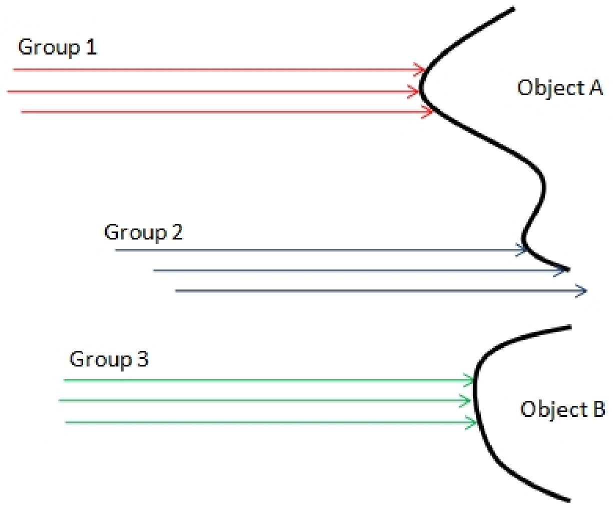

Figure 5.1: Similar rays can intersect similar objects. However, there are some exceptions as it happens with group 2. In this case, two rays have intersections while one of them misses the object.

the objects is needed to assure that the operations were performed over the near objects.

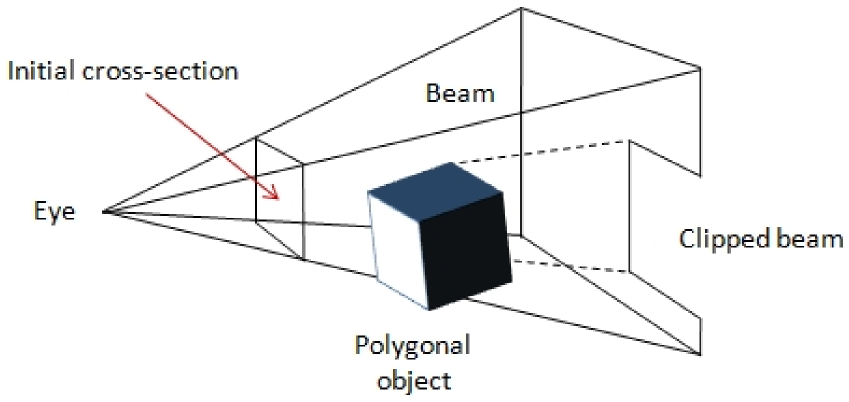

Figure 5.2: A new complex polygonal section is created when a beam is occluded by an object.

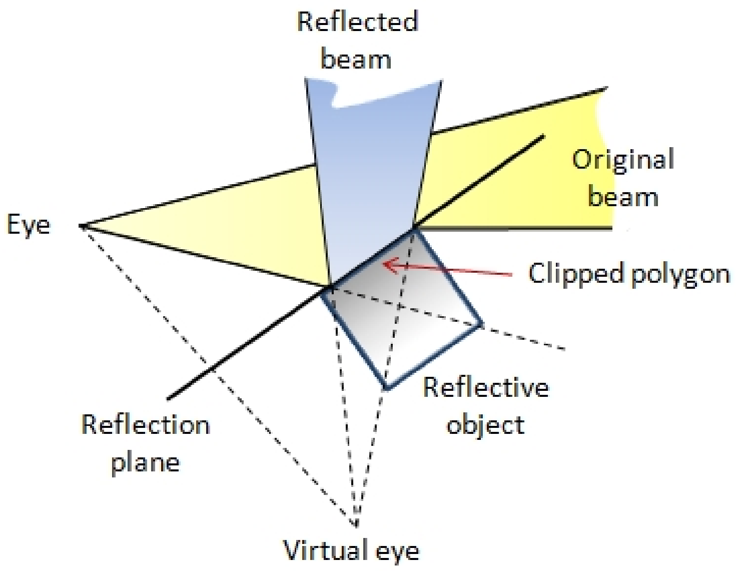

Figure 5.3: A new reflected ray is generated by means of the calculation of a new eye position.

5.4 Exploiting coherence for Implicit Surfaces

The current methods for ray coherence work very well for a small set of objects: in the majority of cases, those methods require objects with polygonal faces to work.

The coherence over implicit surfaces is a different problem, in which, due to the different types of surfaces, the only available information is the definition of the surface. Although ray coherence is hard to achieve [41], using Interval Arithmetic the problem can be solved using only the definition of the surface and the Interval theory to deal with many rays simultaneously.

In this section a new beam definition for implicit surfaces is used to create an acceleration structure for ray tracing. Interval Analysis works with sets of numbers instead of single ones, replacing the floating point arithmetic for an Interval Arithmetic, which naturally replaces the operations over real numbers. This means that using an interval library for the interval operations, and using the properties of intervals, it is possible to deal with a set of rays simultaneously (see figure 5.4).

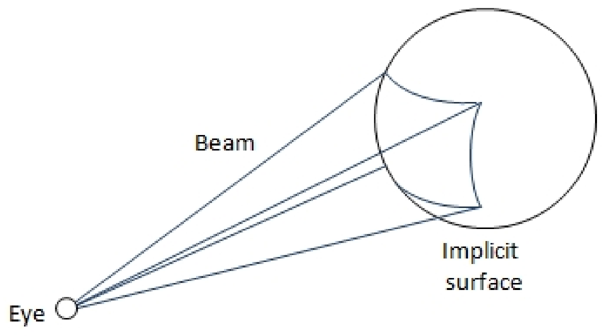

Figure 5.4: A beam intersecting a naive implicit surface.

5.4.1 Definition of the Beam

Using Interval Arithmetic it is possible to define a beam crossing many pixels inside a unique structure. This is performed defining a beam which consists of a set of rays. These rays are enclosed in a pyramid in which the apex is the eye position and the base is an area in image space.

The objective is to use the beam to classify different regions in image space. Because the coherence of 3D objects is preserved when those objects are projected over a 2D plane, this method can use this coherence to create cells or boxes in the screen. The information of the beam crossing the screen is saved in the box formed by the intersection of the beam and the screen; these boxes will represent the direction of the beam.

Let us suppose that the screen is inside an horizontal plane at a distance z_s from the origin of coordinates. An interval (X_s, Y_s) contained in the screen defines, together with the eye position, a beam. This interval (X_s, Y_s) can be outside the projection of the surface on the screen, it can be inside this projection or it can partially intersects it. These three cases are equivalent to:

1. The beam misses the implicit surface completely.

2. The beam completely intersects the implicit surface.

3. The beam partially intersects the implicit surface.

The two first classes correspond to a specific semantic of Modal Interval Analysis which are solved using Modal Interval theorems. The first case will be called *rejection test* and the second one *inclusion test* (see figure 5.5). The last case occurs when the two previous tests fail, and usually means that the beam has rays intersecting the surface, but some rays do not have intersection.

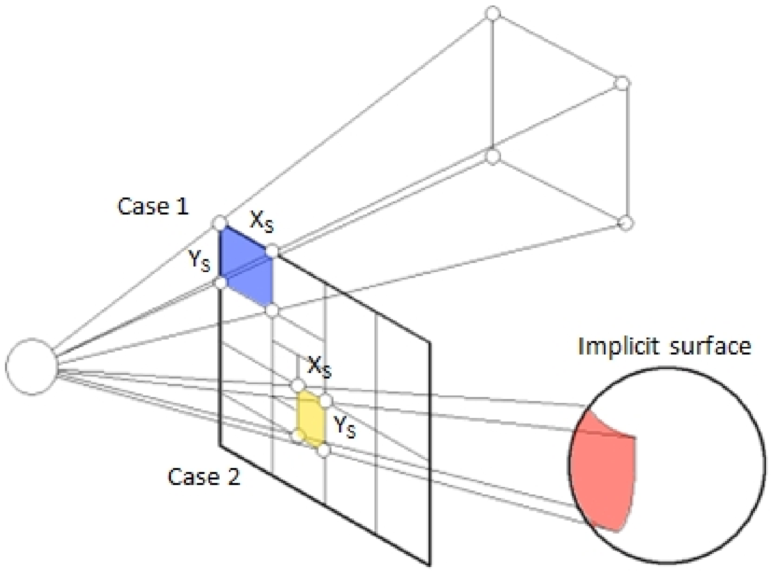

Figure 5.5: Two cases of intersections between a beam and a surface. The first case occurs when the beam misses the surface. The second case represents an intersection among the surface and all the rays that compose the beam.

Once different areas or regions in the screen are obtained, different criteria can be selected to ray trace these regions. It is obvious that rejected areas will not be ray traced, but areas in which the beam completely intersects the surface can be ray traced using less rays. Some examples will be introduced in further sections to show the advantage of these strategies.

For a simple ray the intersection test works to determine whether the equation 2.8 has some solutions. From any interval point of view, this fact means

$$0 \in fR(x_s, y_s, T')$$

where T' is the interval of variation of the ray parameter t. The subdivision algorithms need to arrive to machine precision because in that condition $0 \in F(X)$ is often achieved [18].

For a beam, as x_s, y_s become intervals in the screen, the intersection test must be split between a rejection test and an inclusion test. Modal Interval Analysis (introduced in chapter 3) will give the tools to perform the evaluation of the two test presented in this chapter, because the required semantics are not allowed by classic interval theory. Although the rejection test can be achieved using the classical approach, it is presented using modal interval semantics to keep the coherence in all the algorithms presented.

5.4.2 Rejection test

The objective of this test is to detect when a beam, crossing a region of the screen defined by two intervals X and Y, completely misses an implicit surface. This is equivalent to say that all the rays inside the beam miss the implicit surface. In this case, for all the values of X_s, and for all the values of Y_s, there are no roots for any value of the parameter $t \in T$:

$$(\forall x_s \in X_s)\ (\forall y_s \in Y_s)\ (\forall t \in T)\ f(c_x + t(x_s - c_x), c_y + t(y_s - c_y), c_z + t(z_s - c_z)) \neq 0 \quad \boxed{5.1}$$

which is equivalent to

$$\neg((\exists x_s \in X_s)\ (\exists y_s \in Y_s)\ (\exists t \in T)\ f(c_x + t(x_s - c_x), c_y + t(y_s - c_y), c_z + t(z_s - c_z)) = 0). \quad \boxed{5.2}$$

In accordance with the semantic theorems (see section 3.5.1), this logical formula is equivalent to

$$[0,0] \nsubseteq f^{**}(X_s, Y_s, T) = f^{*}(X_s, Y_s, T)$$

with X_s, Y_s and T proper intervals. From the theorem 1 of Section 3.5.2, for this relation it is sufficient that

$$[0,0] \nsubseteq Out(fR(X_s, Y_s, T)) \quad \boxed{5.3}$$

where Out means the external rounding of the rational extension fR of the function f to the proper intervals X_s, Y_s and T. This indicates that it is enough to know if zero is not contained in the rational calculation to accomplish the semantic of the expression.

Using this, it is possible to implement the test to perform a fast trimming of screen regions that do not contain intersections with the implicit surface. That is, an algorithm using this test can identify regions with pixels that should be shaded with the background color.

The algorithm explained in this section can reject regions image space that correspond to beams missing the surface. This algorithm is presented in figure 5.6.

The algorithm performs a branch-and-bound process over the parameter of the ray, evaluating the function $fR(T, X_s, Y_s)$ with every interval T. The Intervals X_s, Y_s are fixed to an area of the screen to be scanned. The intervals in which $0 \notin fR(T, X_s, Y_s)$ are rejected.

The criterion used to stop the bisection process is:

$$Width(T) < \epsilon_T \equiv (T.Upper\_bound - T.Lower\_bound) < \epsilon_T$$

To select ϵ_T, the user has to take into account that a coarse precision could make that the algorithm does not eliminate enough regions without intersections. Using a high precision, the first root will be found, but the time of the bisection process is increased. Because we only want to discard empty regions, we do not need to use a high precision. Using an ϵ_T value of 10^{-3}, the results obtained are good (see the results at the end of this section).

To perform a trimming algorithm over image space, a branch-and-bound process must be performed in the screen over X and Y values. For every generated box, the rejected test can be used to reject boxes without rays intersecting the surface. When a box of the screen is evaluated and the test fails, the box is bisected and the test must be performed for the new two boxes.

```
Function Reject(Intervals $X_s$,$Y_s$,$Z_s$)
$T$=Interval(0,t)
add $T$ to front of ListT
While ListT is not empty
  $T$ = front of ListT
  If Width($T$) < $\epsilon_T$
    Exit While
  EndIf
  If $0 \notin fR(X_s, Y_s, T)$
    Drop $T$ from ListT
  Else
    Bisect $T$ into $T_1$,$T_2$
    Add $T_1$,$T_2$ to front of ListT
  EndIf
End While
If empty(ListT)
  Box does not have Intersections
Else
  Box can have Intersections
EndIf
```

Figure 5.6: Rejection test to detect if a beam crossing a box misses the surface completely.

Some examples of the rejection test are presented in figure 5.7. The blue boxes were rejected using only one rejection test that has the same complexity of an intersection test using a single ray. In this case, it is only necessary to ray trace the yellow boxes that represent boxes that could have rays intersecting the implicit surface.

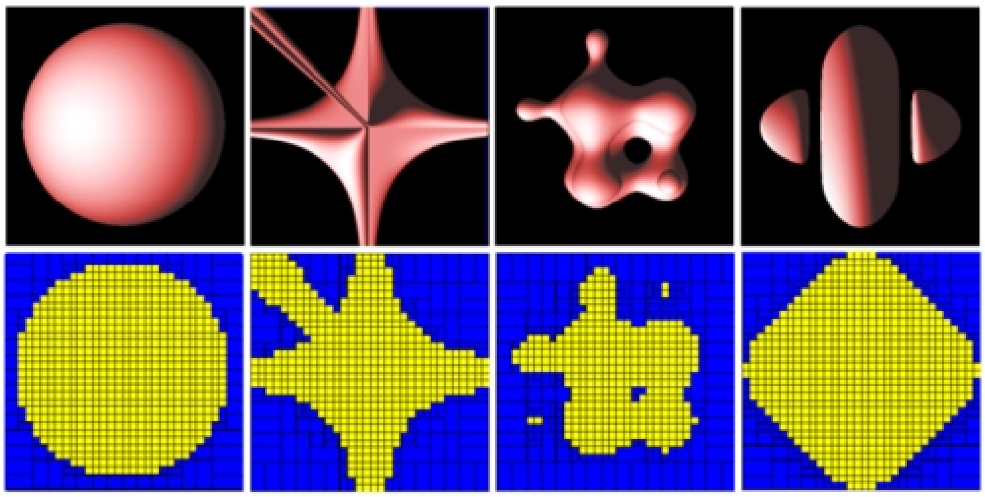

Figure 5.7: Some examples of the rejection test. At the top of the image, there are some implicit surfaces: Sphere, Blobby, Steiner and Gumdrop torus. At the bottom, the results of the rejection test in screen space for every surface.

Table 5.1 summarizes the results for a ray tracing process combined with the trimming strategy, and a classic ray tracing using individual rays. Although the trimmed area of the screen is almost 50% of the area in all the cases, the time saved is not necessary 50%, because the intersection test is faster in pixels in which the rejection test is true. To illustrate this, the time that a ray casting algorithm takes over the rejected boxes is shown in table 5.2. The table presents the average time per pixel. The times for a ray casting over the non-rejected boxes is given in table 5.3.

Surface	trimming time	trimming + ray tracing	Classic ray tracing	% Saved time
Sphere	0.79	41.46	53.22	22.07 %
Blobby	1.33	64.17	107.59	40.34 %
Steiner	1.59	94.43	152.63	38.13 %
Gumdrop torus	3.6	381.62	544.95	29.97 %

Table 5.1: Comparison between a classical Interval ray tracing and a ray tracing using the trimming strategy (time in seconds).

As can be seen in tables 5.2 and 5.3, the intersection test takes more time in pixels corresponding to non-rejected boxes than rejected boxes. This occurs because the intersection test for rays intersecting the implicit surface must arrive at a smaller precision to obtain the value for the intersection. Rays that do not intersect the implicit surface are rejected before a small precision is achieved.

Figure 5.8 shows a color map for the Steiner surface in which the time that the intersection test takes at every pixel is color encoded. The pixels in which the intersection test fails take less time than pixels inside the implicit surface. Also note that the intersection tests take the maximum time on the borders of the surface.

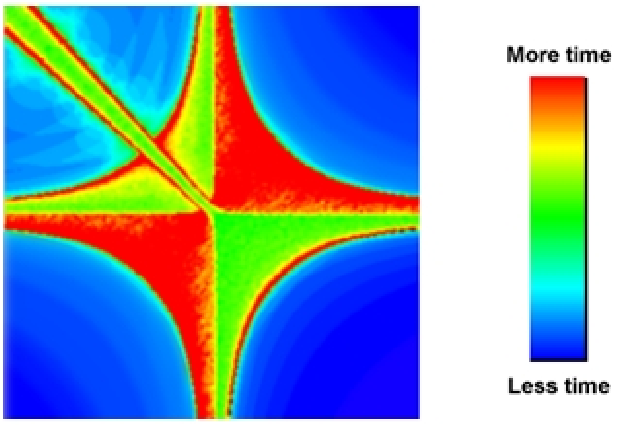

Figure 5.8: Color map to represent the time that the intersection test takes for every pixel in a Steiner surface.

Surface	Boxes Outside	Pixels	Time	Time/pixel
Sphere	92	33105	12.54	0.00038
Blobby	171	59204	44.74	0.00076
Steiner	132	46958	59.79	0.00127
Gumdrop torus	156	33087	166.93	0.0050

Table 5.2: Time in seconds for ray tracing rejected boxes.

Surface	Boxes Inside	Pixels	Time	Time/pixel
Sphere	648	57496	40.67	0.00071
Blobby	350	31397	62.84	0.00200
Steiner	488	43643	92.84	0.00213
Gumdrop torus	648	57514	378.02	0.00657

Table 5.3: Time in seconds for ray tracing boxes in which the rejection test fails.

5.4.3 Inclusion Test

The objective of this test is to detect those beams that completely hit an implicit surface. The test allows to know if for a box in the screen all the rays crossing it intersect the implicit surface. That is, for all the values of X_s, and for all the values of Y_s, must exists a root for the parameter T. Formally:

$$(\forall x_s \in X_s)\ (\forall y_s \in Y_s)\ (\exists t \in T)\ f(c_x + t(x_s - c_x), c_y + t(y_s - c_y), c_z + t(z_s - c_z)) = 0 \tag{5.4}$$

which in accordance to the *-semantic theorem of the Modal Interval Analysis it is equivalent to the modal interval inclusion:

$$f^*(X_s, Y_s, T)\ \subseteq\ [0, 0] \tag{5.5}$$

in which X_s,Y_s are proper intervals, and T is an improper one.

To get a good approximation of f^*, the rational extension of the function is used:

$$f^*(X_s, Y_s, T)\ \subseteq\ f^*(X_s, Y_s, t)\ \subseteq\ Inn(fR(X_s, Y_s, t))$$

where *Inn* means the inner rounding of the rational extension fR. This last expression is true because the improper interval $T\ \subseteq\ t$ in accordance with Theorem 1 of section 3.5.2. Also, for every $t_1, ..., t_n \in T'$

$$f^*(X_s, Y_s, T)\ \subseteq\ Inn(fR(X_s, Y_s, t_1))\ \wedge\ ...\ \wedge\ Inn(fR(X_s, Y_s, t_n))$$

which shows that if the meet of the intervals obtained by means of the rational calculations for different values of $t \in T$ is contained in $[0, 0]$, then,

$$Inn(fR(X_s, Y_s, t_1))\ \wedge\ ...\ \wedge\ Inn(fR(X_s, Y_s, t_n)) \subseteq [0, 0], \tag{5.6}$$

then 5.5 is true. The values of t used in the rational calculation can be the bounds of the intervals obtained by means of subdivisions of the interval T'. The inclusion test is presented in figure 5.9.

The algorithm in table 5.9 performs a branch-and-bound process over the parameter T. The lowest and highest values of the interval (infimum and supremum) at every subdivision step over T, are used to evaluate the inclusion function. After that, the *meet* operation of those two intervals is calculated. The *meet* operation between two intervals generates a new interval, in which the new infimum value is the maximum of the two infimum, and the new supremum is the minimum of the two supremum. This operation is also performed with the *meet* obtained in the previous subdivisions steps. If the value of the *meet* contains zero, the box currently evaluated represents an inside region.

To optimize the process, both tests (rejection and inclusion) must be evaluated in the same algorithm to obtain a 2D structure to take advantage of coherence in image space (see figure 5.10). The inclusion test is only true when $0 \in meet$. Otherwise, the list of values of T must be evaluated at the end of the inclusion test. If that list is empty, the beam misses the surface (the rejection test is true). If that list is not empty then both tests failed. This means that the region is undefined, that is, it can contain rays intersecting and missing the surface.

To further improve the ray tracing process over included boxes (boxes in which the inclusion test is true), it is necessary to calculate a small space of solutions for the parameter T. The rays must locate the roots in this space instead of a big value of the parameter T. This is similar to the calculation of the entry and exit values of a ray in a bounding box. The entry and exit point calculated for every ray conform to an interval in which the intersection test operates for the surfaces contained in the box. In the case presented here, the calculation of these values is performed previously for all the beam, and not for every ray. The new value of T for every box in screen is obtained taking the nearest and the farthest intersection between the beam and the surface (see figure 5.11). This interval is the initial space to search roots for every ray traced in the box.

The nearest and the farthest intersection points are calculated during the subdivision process using the result of the rational calculation performed in the bounds of interval T. If a result is bigger than 0, the infimum of the result is saved in a vector. If the result is less than 0, the supremum value is saved in another vector. After that, the maximum of the minimums and the minimum of the maximums are used to create the

```
Function Inclusion(Intervals $X_s$,$Y_s$)
    $Meet$=$[-\infty, \infty]$
    $T$=Interval(0,t)
    add $T$ to front of ListT
    While ListT is not empty
        $T$ = front of ListT
        If Width($T$) $< \epsilon_T$
            Exit While
        End If              If $0 \in fR(X_p, Y_p, T)$
            $R_a = fR(X_p, Y_p, T.Inf)$
            $R_b = fR(X_p, Y_p, T.Sup)$
            $Meet = Meet \wedge R_a \wedge R_b$
            If $0 \in Meet$
                The Beam Intersects completely the surface
            Else
                Bisect $T$ into $T_1$,$T_2$
                Add $T_1$,$T_2$ to front of ListT
            Endif
        EndIf
    End While
```

Figure 5.9: Algorithm to perform the inclusion test. This test determines if a beam completely intersects the implicit surface.

final interval value of T called T_{Final}. This value, which represents the set of values of t for all the rays traced in the set, is used to determine the regions covered by shadows (see the next section).

```
Function EvaluateBox(Intervals $X_s$,$Y_s$)
    $Meet$=$[-\infty,\infty]$
    $T$=Interval(0,t)
    add $T$ to front of ListT
    While ListT is not empty
      $T$ = front of ListT
      If Width($T$) $< \epsilon_T$
        Exit For
      EndIf
      If $0 \in fR(X_p, Y_p, T)$
        $R_a = fR(X_p, Y_p, T.Inf)$
        $R_b = fR(X_p, Y_p, T.Sup)$
        $Meet = Meet \wedge R_a \wedge R_b$
        If $0 \in Meet$
          The Beam Intersects completely the surface
          End Function
        Else
          Bisect $T$ into $T_1$,$T_2$
          Add $T_1$,$T_2$ to ListT
        EndIf
      Else If $0 \notin fR(X_p, Y_p, T)$
        Drop $T$ from ListT
      Else
        Bisect $T$ into $T_1$,$T_2$
        Add $T_1$,$T_2$ to front of ListT
      EndIf
    EndFor
    If empty(ListT)
      Box does not have Intersections
    Else
      Box can have Intersections
    EndIf
```

Figure 5.10: Combined algorithm to apply the reject and inclusion tests in a box.

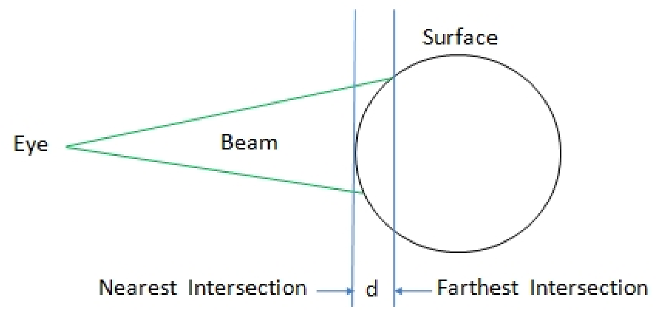

Figure 5.11: The distance between the nearest and the farthest point of Intersection is used to give a final value for T.

5.4.4 Identification of boxes inside a shadow

The structure of the previously generated boxes works with primary rays. However, it is possible to add information to identify which boxes of the structure may be in a shadow.

The algorithm presented in this section will be used to detect in which boxes the shadow rays must be traced, and other boxes in which the trace of shadow rays is not necessary. If the boxes that do not require the trace of shadow rays are detected, then this time will be saved.

The algorithm proposed in this section rejects the boxes that are directly illuminated. The boxes that remain when this algorithm is used, may be in shadow or not, but the test cannot decide. Shadow rays must be traced in those boxes.

The problem of identifying the intersection area of a beam over an implicit surface is not easy to solve, because that area can have any shape. This problem is less complicated for polygonal objects because the intersection is always performed over plane objects.

In the previous section, we found the range of intersection points for which the individual rays of the beam intersect the implicit surface (T_{Final}). This interval indicates the range of final values of t for all the rays in the beam. Using this interval, it is possible to calculate the set of intersection points between the beam and the surface, and trace an arbitrary beam from the light source to this set of points (see figure 5.12).

The shadow beam is traced only for the boxes for which the inclusion test for the primary beam is true (that is, for boxes in which the corresponding beam completely intersects the implicit surface). Given the final value of the interval T_{Final} calculated for the box, the intersection values are calculated as follows:

$$X_{sh} = c_x + T_{Final} * (X_s - c_x) \tag{5.7}$$

$$Y_{sh} = c_y + T_{Final} * (Y_s - c_y) \tag{5.8}$$

$$Z_{sh} = c_z + T_{Final} * (z_s - c_z) \tag{5.9}$$

In this case, (c_x, c_y, c_z) represent the coordinates of the origin of the primary ray and X_s, Y_s define the area of the screen in which the primary ray was traced, and z_s defines the distance from the origin of coordinates to the plane that contains the screen.

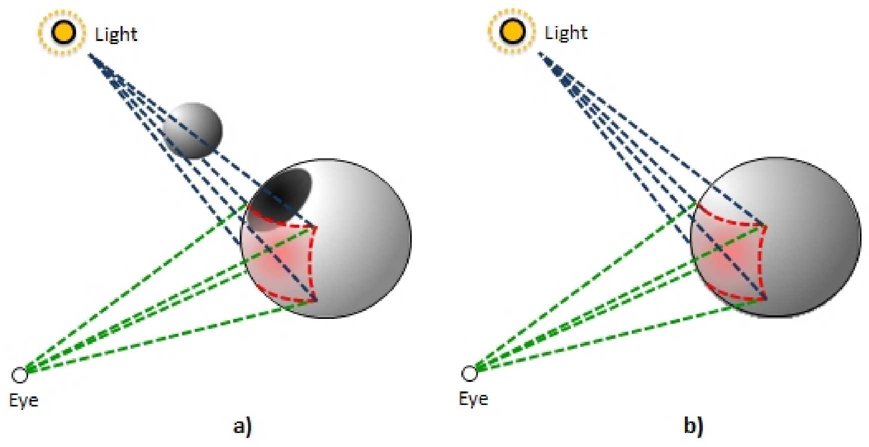

Figure 5.12: A beam from the light source is traced to detect boxes covered by shadows. a) If the shadow beam intersects an object, the area covered by the beam can be under a shadow. b) If the shadow beam is not intersecting an object, the area is not under a shadow.

Using the position of the light, and the calculated values of X_{sh}, Y_{sh}, Z_{sh} it is possible to define a shadow beam. The problem is then reduced to perform a rejection test, in which is sufficient that:

$$[0,0] \not\subseteq Out(fR(X_{sh}, Y_{sh}, Z_{sh}, T))$$

For this rejection test, the definition of any implicit surface is used, replacing the values of X,Y,Z of the shadow beam in the corresponding values of the implicit function. The application of a rejection test was already explained in section 5.4.2

The rejection test for the shadow is evaluated only for those boxes in which the corresponding primary beam (the beam traced from the view point) completely intersects the surface. Moreover, this rejection test must be performed for all the surfaces in the scene.

The objective is to identify if between the light and the box there is any surface blocking the beam emanating from the light. In that case, every pixel corresponding to the box must be checked for shadows. In the other case, that is, if there is not any object blocking the light that arrives at a box, the pixels corresponding to the box are not checked for shadows.

Figure 5.13 shows the structure generated for a set of spheres. The rejection test for primary rays was true for the blue boxes. For red and yellow boxes, the inclusion test is true. The yellow boxes represent the parts of the surfaces that may be covered by a shadow. Those boxes are found using the rejection test for shadow beams (remember that this test is only used in boxes in which inclusion test was true). Boxes that fail both test generally correspond to the borders of the surfaces. Figure 5.13 also shows the results of the ray tracing process over the structure previously generated. Using a classic interval ray tracing, this image is obtained in 38 minutes (using 9 rays per pixel; 394045 view and shadow rays). Using the structure of boxes and, tracing 9 rays in undefined areas, 1 ray in inside areas and tracing shadow rays only in shadow boxes, the visualization process takes 16 minutes (using 188641 primary and shadow rays).

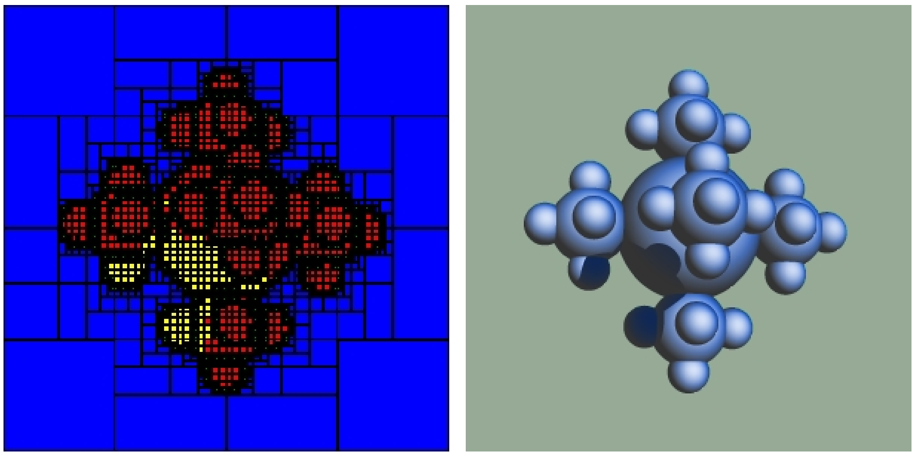
Figure 5.13: Some boxes in shadow in a scene composed by a set of spheres.

5.4.5 Experimentation and results

Three surfaces are used to measure the efficiency of a ray tracing process using the structure of boxes in image space: an Orthocircle surface, a Blobby surface and a Drop surface (see figure 5.14). The surfaces are rendered at 300x300 pixels, using 9 rays per pixel. For this test, a Pentium 4 (2,4Ghz) computer was used.

A structure of boxes, as was explained in the previous sections, is generated for each surface. Only one ray is traced for boxes that correspond to beams intersecting the surface, and nine rays for undefined boxes, that correspond to the borders of the surfaces. For the classic ray tracing, a uniform grid was generated. The results are compared with a classic ray tracing process using interval arithmetic (see table 5.4).

In this section we are using beams only to generate the acceleration structure. In the next section, we will explain an anti aliasing strategy using beams instead of single rays to compute the intersection tests, in order to improve the quality of the images obtained.

According to table 5.4, the improvement is more than twice for the surfaces tested. The quality of the images obtained by means of our acceleration structure, tracing only one ray in inside boxes, and using the classic ray tracing approach with nine rays per pixel, is the same. However, note that the the thin part connecting the drop with the main body in the figure 5.14c is still not perfect. In the next section, this problem is solved with an anti aliasing approach using beams instead of rays.

5.5 Removing Aliasing using Interval Arithmetic

5.5.1 Aliasing

In computer graphics, aliasing refers to the jagged edges that appear in the visualization of surfaces. Its principal manifestation is high contrast of the border color over the background color. This occurs because a sampling process is needed to work with pixels which are discrete entities.

When frequencies are greater than the Nyquist Limit (one cycle every two pixels) aliasing effects appear in the visualization. This limit is determined by the pixel size [41]. This problem is shown in figure 5.15. If an inadequate number of samples is applied then alias can appear as low frequencies [16].

Ray tracing is a discrete algorithm, for that reason, the problem of aliasing is inherent to it. There are three main approaches to solve aliasing problems in ray tracing: super sampling, adaptive sampling and distributed sampling (see figure 5.16).

Super sampling is based on a uniform distribution or rays inside the pixel. Indeed, lots of rays are traced for every pixel, and the final color of the pixel is obtained by means of the average of the colors for every ray traced [41]. To avoid aliasing problems,

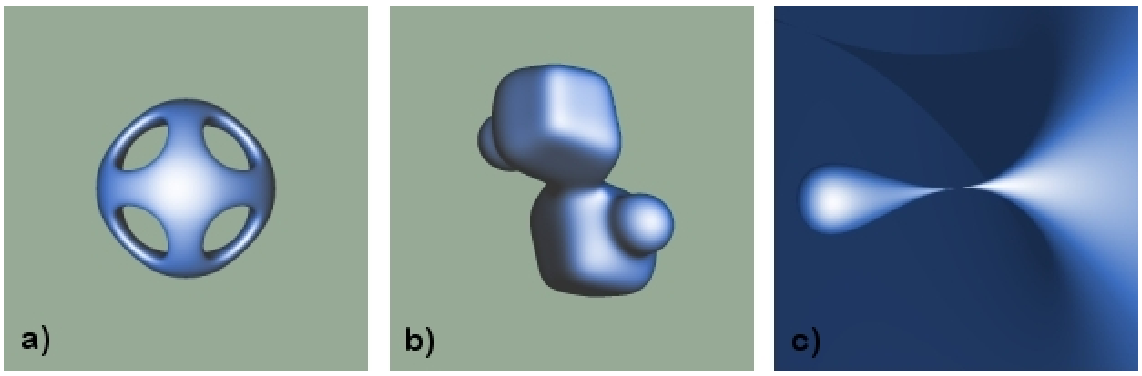

Figure 5.14: Surfaces used to test the efficiency of the structure of boxes. a) Orthocircle, b) Blobby surface, and c) Drop surface.

	Presented Method		Previous technique		
Surface	Time	Rays Traced	Time	Rays Traced	% Improvement
Orthocircle	8,8	68631	21,1	375003	139%
Blob	11,6	58273	35,6	414406	205%
Drop	35	460495	106	1620000	202%

Table 5.4: Comparison between a classic Interval Ray Tracing process and a ray tracing using the structure of boxes generated by means of the inclusion and rejection tests (time in minutes).

a big number of rays should be traced, but there could be some frequencies that can produce alias. Adaptive sampling [101] can solve these problems reducing the aliasing in the edges. In this approach, the pixel is recursively subdivided in small areas and new rays are traced in the edges. If the colors for the neighbor rays traced in the edges of an area of the pixel are too different, the area is again subdivided until a threshold for the size of the area is achieved. If the colors are similar, the subdivision is finished and the different values are weighted to obtain the final shade color of the pixel. The disadvantage of this method is that it requires a large number of rays which decrease the efficiency of the ray tracing process.

A variation of super sampling is distributed sampling, in which a non-regular distri-

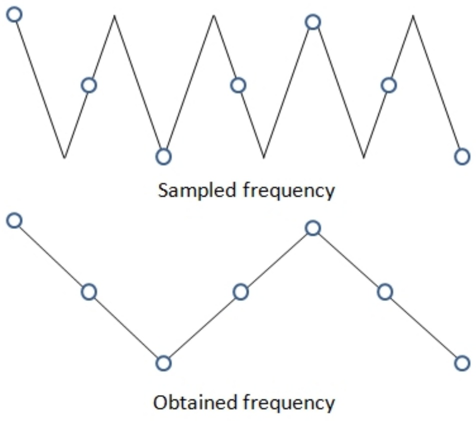

Figure 5.15: An incorrect sampling over a signal with high frequency (top) can generate a low frequency alias (bottom).

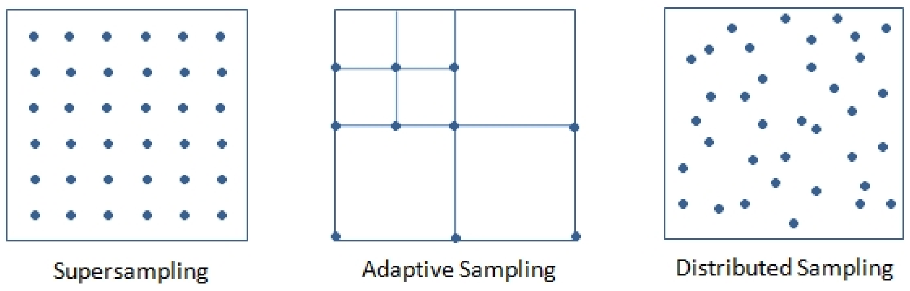

Figure 5.16: Aliasing techniques for ray tracing in a pixel. Supersampling (left), Adaptive sampling (center) and Distributed sampling (right).

bution of the rays is used to replace aliasing for noise, which is less noticeable for the human eye [41].

Using the boxes structure presented in the previous sections, an efficient anti aliasing process can be implemented. As was pointed out in the last section, undefined boxes correspond to border areas that always need an anti aliasing process. However, boxes that are completely included can also have aliasing due to the variation of the surface inside the box. In the example of figure 5.17, the blobby surface has a small sphere that is part of the surface. That sphere causes aliasing in its borders.

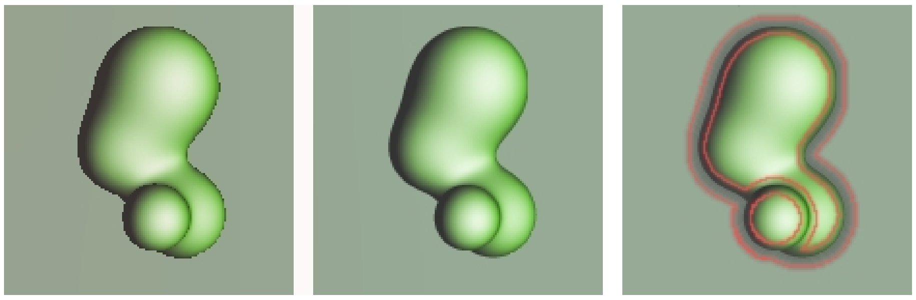

Figure 5.17: A blobby surface with aliasing (left). The same surface without aliasing (middle). The aliasing must be corrected in the borders of the surface but also in the small sphere that is inside the surface (right).

To use the structure of boxes to improve the efficiency of an anti aliasing process, a new constraint can be added during the creation of the structure. The objective is to find those boxes included in the surface that can produce aliasing. The constraint must be the size of the final width of the value of T_{Final}. The width of T_{Final} represents the distance between the first and the last ray intersecting the surface in the beam; if that distance is too big, the box must be subdivided and the new boxes evaluated. Using this strategy, the different aliasing regions are identified as seen in figure 5.18. The borders of the small sphere are identified as aliasing regions because the width of the final value of T for those boxes is big, this is, there is a big distance between the first and the last intersection of the beam. After that, many rays must be traced in the undefined boxes (green color), and also in the boxes in a shadow; in the red boxes, only one ray needs to be traced. It is possible to use super sampling, adaptive sampling or distributed sampling in aliasing boxes.

5.5.2 Improving the point sampling process

The anti aliasing approaches presented in the last section are based in point sampling. The disadvantage of this approach is that thin features of the surfaces can be missed by the rays (see figure 5.19).

There are many approaches to solve aliasing caused by missing objects. Whitted [101] used bounding boxes for small objects. If a ray intersects a bounding box, the sampling rate is increased to guarantee that view rays do not miss the object. Although effective in most of the cases, this technique does not work very well with long thin objects [37].

The approaches that are based on gathering information of a continuous set of rays as cone tracing, beam tracing and pencil tracing, can replace the use of infinitesimal rays. The main disadvantage of these proposals is that they require computationally complex intersection tests.

In this section, an Interval Arithmetic Anti aliasing strategy (IAA) based in an adaptive anti aliasing process is presented. This method uses the machinery developed to create the structure of boxes in image space to accelerate the ray tracing process; that is, the rejection and inclusion test are used to evaluate areas of the pixel instead of areas of the screen. Interval arithmetic guarantees that any part of the surface is not missed during the process.

The IAA algorithm is performed by means of a subdivision process over the pixel

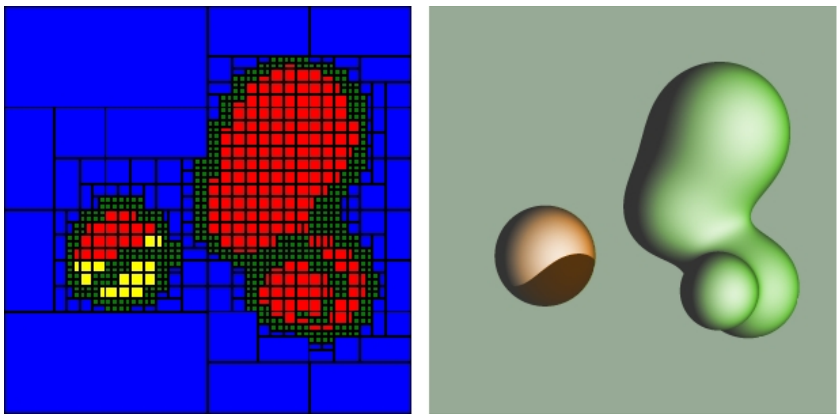

Figure 5.18: The structure of boxes can be used to identify aliasing areas in image space.

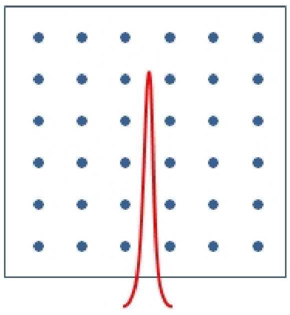

Figure 5.19: When point sampling is used, some thin features of the surfaces can be missed.

area. At the beginning of the process, a beam is traced for the whole area of the pixel. If the rejection test is true, then the pixel is shaded with the background color.

If rejection test fails but the inclusion test is true, then the dot product between the beam and the surface normal is calculated as follows: if a beam completely intersects the surface, the value of T_{final} is returned by the inclusion test. This value is used to calculate the interval values of X, Y and Z, corresponding to the intersection between the beam and the surface (see equation 5.7). Using an interval version of the derivatives of the implicit function in x, y, z, it is possible to obtain the vector (N_x, N_y, N_z) which represents the set of normals of the surface in the intersected area. This vector can be obtained replacing the values of X, Y, Z in the derivatives obtained in last step. In the case of a ray, the vector from the view point (c_x, c_y, c_z) to an intersection point (x, y, z) is defined as $(x_s - c_x, y_s - c_y, z_s - c_z)$. For a beam, the intersection is composed by intervals, then replacing the real values for interval values we obtain $(X - c_x, Y - c_y, Z - c_z)$. The estimator is obtained by means of the interval dot product between the vectors $(N_x, N_y, N_z))$ and $(X - c_x, Y - c_y, Z - c_z)$, which is equivalent to:

$$(N_x(X - c_x), N_y(Y - c_y), N_z(Z - c_z)) \tag{5.10}$$

Although other estimators could be used, this is chosen because it is simple to implement and gives enough information about the variation of the curvature of the surface to detect regions with potential aliasing.

If the width of the interval dot product between the beam and the normals is bigger than a predefined threshold, the pixel is subdivided in four sections or sub pixels. Also if both tests fail, then the pixel is subdivided in four sections.

A beam must be traced for every sub pixel and the process started in every new area. When the estimator in a sub pixel is equal or less than the threshold, the average of the normals is used to calculate the Phong shading of the sub pixel. Also, the subdivision process continues until the area of the sub pixel is less or equal to another threshold. This threshold determines the number of samples allowed per pixel. In the examples of the next section, the maximum threshold allowed is a depth of 4 in the subdivision tree. When this threshold is achieved, the first surface intersected by the beam is used to calculate the shade color for the current area.

Finally, the pixel is shaded using the average of the areas and the shading values of every sub pixel.

Figure 5.20 shows an example of the IAA algorithm over a pixel. First, all the area of the pixel is scanned (Figure 5.20a). Because the surface is not completely covering the pixel, the area of the pixel must be subdivided (Figure 5.20b). For the new four sections, only the right-bottom section is completely covered by the surface. If the estimator is accomplished for this section of the pixel, the section is not subdivided and Phong shading can be calculated (Figure 5.20c). The other sections are subdivided and evaluated. Some of the new sections are covered by the surface (red color). Blue boxes must be shaded with background color, while yellow regions are undefined (Figure 5.20d).

The IAA algorithm can visualize features that are hard to detect with ray tracing using point sampling. This is the case with the Teardrop surface (Figure 5.21). According to

Paiva *et al.*, even when a robust polygonal approximation of this surface is used, it is not possible to render the thin feature that joins the drop with the main body [71].

A Chub's surface (Figure 5.22) has many sections connected in just a few pixels which are correctly rendered using the IAA algorithm.

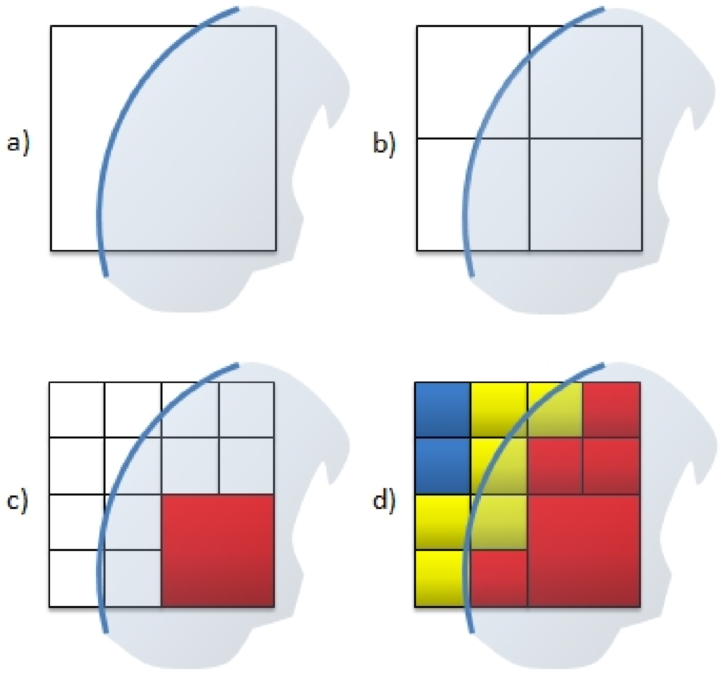

Figure 5.20: An example of the IAA algorithm over a pixel.

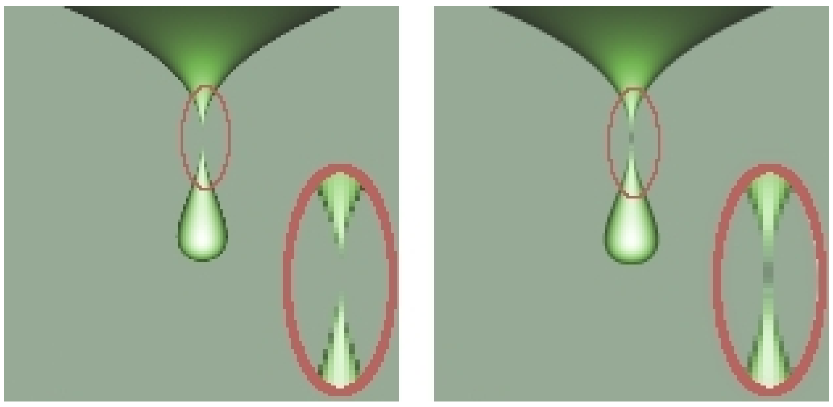

Figure 5.21: A drop surface rendered using point sampling (left) and IAA algorithm (right).

Figure 5.23 shows the advantage of IAA over an adaptive anti aliasing method based on point sampling. When point sampling is used, nothing is known about variations of the surface between neighboring rays. Using IAA, the details inside the whole pixel are taken into account. In this case, the beam is subdivided because the surface has too much variation in the area covered by the beam. If in the new set of beams the smaller and the bigger values of the estimators are too different, then those beams must be subdivided again.

5.5.3 Experimentation and results

The IAA method was tested on the surfaces presented in figure 5.24. The comparisons have been performed against an adaptive anti aliasing algorithm based on point sampling (using a reliable algorithm based on interval arithmetic for the intersection test). Both techniques were applied to the classical interval ray tracing algorithm based on a branch-and-bound strategy. All the examples were generated using a resolution of 500x500

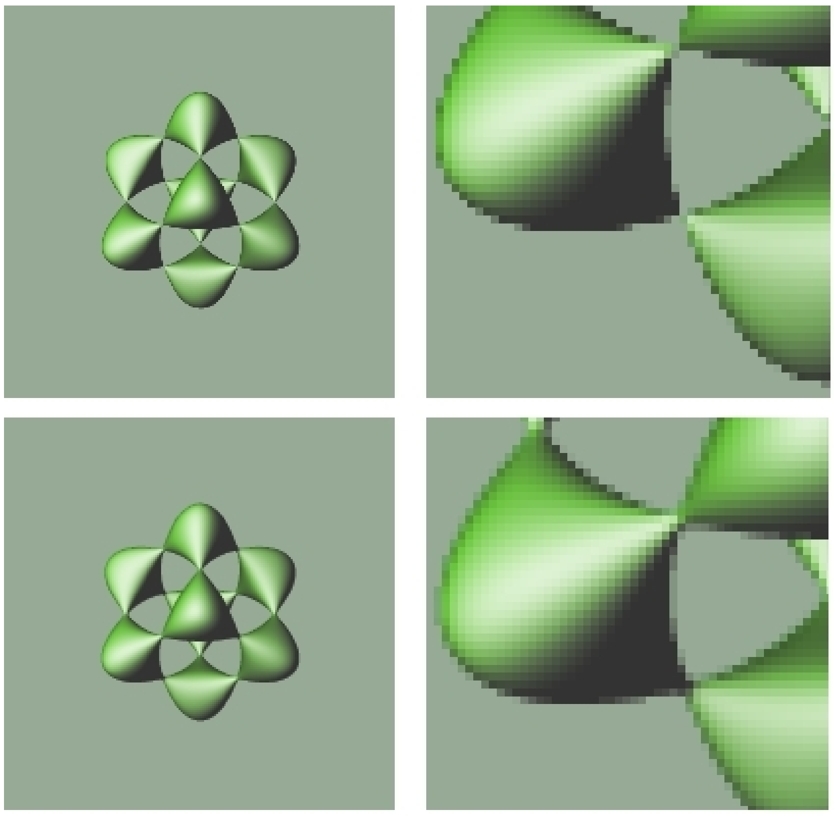

Figure 5.22: When anti aliasing strategies based in point sampling are used, some details of the surface disappear (top-right). Using IAA, these details are well rendered (bottom-right).

pixels. This resolution is enough to show the improvements in the quality of the final visualization. Higher resolution images will be shown in the final chapter of this thesis, where we use acceleration techniques to avoid the high rendering times.

Figure 5.24 (a and b) shows a Twist with shadows. Fine details of the shadow are not well visualized using point sampling (a). Using IAA, those details are better visualized (b). Problems in the visualization of the point sampling example occur because shadow rays miss the thin details of the Twist. The visualization of figure 5.24(a) takes 27 minutes; figure 5.24(b) takes 20 minutes. The time difference is due to the fact that IAA detects pixels without much variations inside, using one single test. In the point sampling test, at least four rays are traced for every pixel. Figure 5.24(c) shows an example of a blobby surface visualized with the IAA algorithm in 15 minutes. Figure 5.24(d) represents the Tri-trumpet. Details of that surface are presented in figure 5.24(e) for adaptive point sampling and figure 5.24(f) for IAA. Using our method, the borders look better and the image is created in 8 minutes. Using point sampling, the sections appear separated although interval arithmetic is used for the intersection tests. Using point sampling, the algorithms takes 7 minutes.

5.6 Conclusions

This chapter introduced an alternative algorithm to improve the efficiency of the ray tracing process when interval arithmetic is used. The idea is to combine many rays in one beam, and evaluate it as a unique ray. Interval arithmetic has been demonstrated

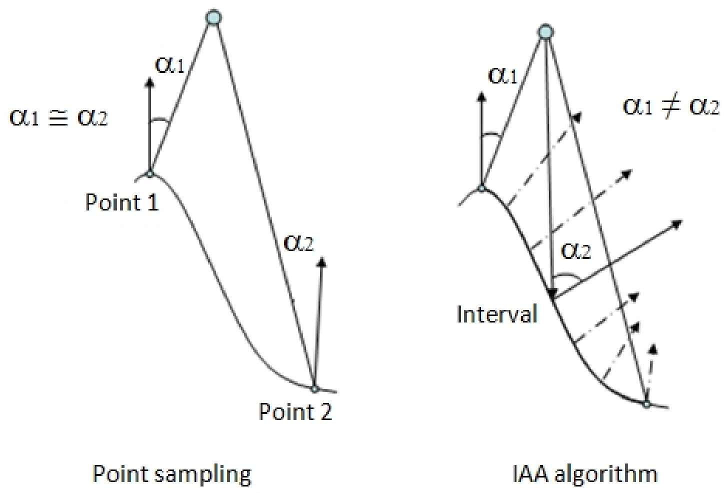

Figure 5.23: Advantage of IAA to estimate the variation in the implicit surface. Using point sampling, nothing is known about the variation between two points. Using IAA the variation is detected.

to be a good tool to exploit the coherence when many rays are analyzed simultaneously. The results demonstrate that this technique can improve the efficiency even twice in some cases.

An algorithm that improves the quality of the images obtained from the ray tracing of implicit surfaces was also introduced. We have shown that point-sampling anti aliasing techniques can generate some noticeable problems in the visualization of the implicit surfaces, even when interval arithmetic is used in the intersection test. The algorithm provided solves those problems, and also it is possible to combine it with the beams to improve the overall efficiency of the rendering process.

The algorithms presented also include the creation of secondary beams, in particular of shadow beams. The definition of shadow beams involve of complicated intersection tests when algorithms based on beam tracing are applied. This can be even more complicated when applied to implicit surfaces, because the intersection between the primary beam and the surface can have any shape. In the algorithm for shadows presented in this chapter, the problem is solved in an easy and reliable way. It is even more efficient

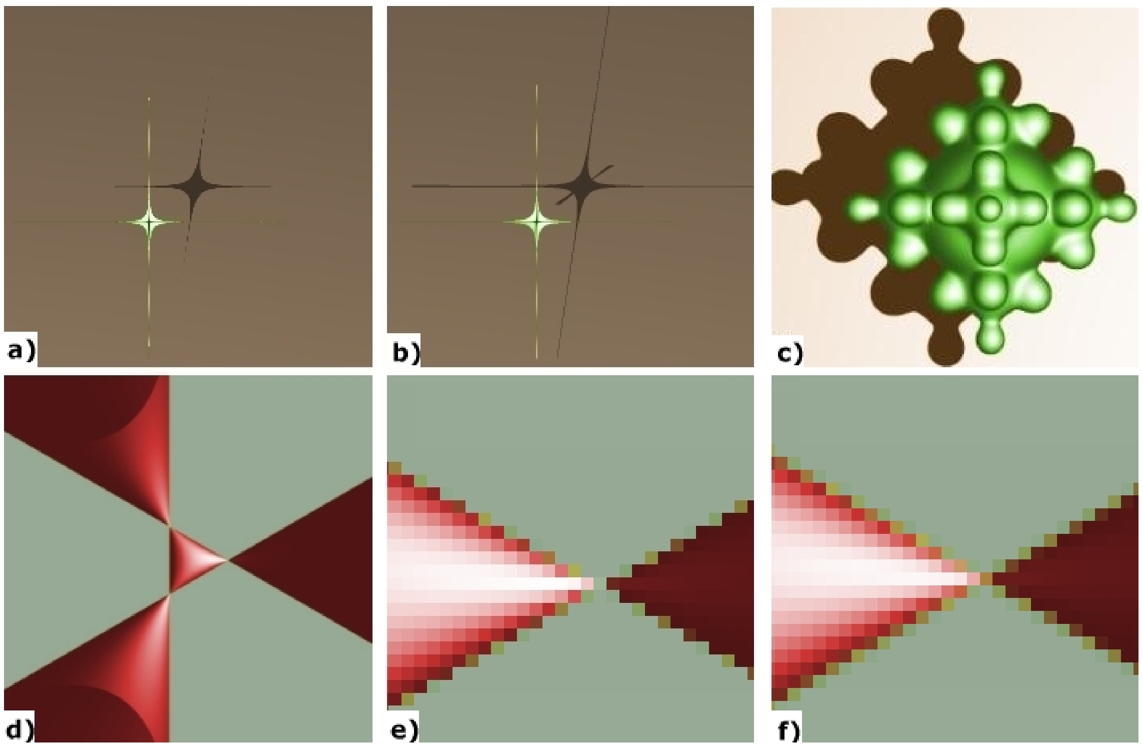

Figure 5.24: Experimental images. An image with thin features is not correctly rendered using point sampling (a). Using IAA the result is better (b). The method can be used for general surfaces, like a set of blended spheres (c). Also, a surface with critical points like the Tri-trumpet (d) is not correctly rendering using point sampling (e), but the critical points are detected using IAA (f).

than other approaches, because many areas without shadows can be discarded at the beginning of the process; the algorithms do not have to look for shadows in those areas.

The algorithms presented are faster than other reliable algorithms based on Interval Arithmetic. However, these algorithms are still slower than non reliable ones. This is the cost to pay if a correct representation of the surface is needed.

6 Exploiting coherence to improve the efficiency in Animated Scenes

6.1 Introduction

This chapter presents an application of the contributions introduced in the previous chapter, to the animation of scenes composed of implicit surfaces.

In the previous chapter, the algorithms were devoted to improve the quality of the images and the efficiency of the rendering process, but over individual images. To improve the efficiency of animated scenes, a spatial structure is needed. This structure will be used to exploit space coherence and reduce the rendering times. For that reason, this chapter starts with the explanation of the creation of such structure, in this case, a regular grid. Moreover, a new traversal method to work with beams instead of rays will be described. Interval arithmetic is used to maintain the numerical reliability in all the process.

To exploit temporal coherence, the position of the surface in a fixed number of frames is added as an interval variable in the transformations applied to the implicit surface. This process creates an special surface that contains all the positions of the surface for a desired number of frames.

6.2 Creation of space structured scenes

In the previous chapter, image space coherence was used in the interval approach to accelerate the tracing of primary and shadow beams. This section explains that it is also possible to apply acceleration techniques to exploit space coherence by means of space structured scenes. The scenes will be structured by means of a regular grid, which can be easily adapted to work with beams.

In a previous work, Gonzalez [62] presented a technique to work with octrees and beams. In that work, the structure is used to accelerate the tracing of beams only to primary rays, although the same octree is used to trace shadow and reflection rays. The initial beams are generated by means of the projection of the octants in the screen. Every face of the octant seen through the screen is used to trace a beam. The other parts of the screen are supposed to be empty, so rays are not traced there and the background color is set in the pixels corresponding to these areas.

The main differences between the technique presented in [62] and the work presented in this chapter is the use of interval arithmetic. This helps to keep the reliability in the creation of the structure and the traversal of the beams.

6.2.1 Creation of the regular grid structure

In chapter 4, an explanation of the process of creation of an octree structure using interval arithmetic was presented. The creation of a grid is not so different, because the evaluation of the space in x,y and z is performed in the same way.

Duff [30] described the creation of structured scenes by means of CSG (Constructive Solid Geometry) operations, using Interval Arithmetic. He subdivided the space of the scene until a precision equal to the pixel size is achieved. During subdivision, CSG operations are tested in every cell. In other works, Lipschitz constants are used to perform the evaluation of CSG operations [29].

In the algorithm presented in this chapter, a regular grid is used to create a structure of the scene for further beam tracing. The object space in the scene is subdivided in a predefined number of boxes or cells, and every one is scanned to look for the implicit surfaces crossing it. Having an implicit surface $f(x, y, z) = 0$, to know if the surface crosses a cell defined by (X,Y,Z), the united extension of f to the cell, $F(X, Y, Z)$ 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.

Moreover, the CSG operations are evaluated in every cell. The allowed operations are *union* ($f_1 \cup f_2$), *intersection* ($f_1 \cap f_2$), and *difference* ($f_1 - f_2$). The CSG tree is only evaluated for the surfaces crossing the cell. A surface that is not crossing the cell is represented as an empty set in the CSG tree.

6.2.2 Traversing of the regular grid

The objective of the spatial structure is to identify the areas of the object space that need to be ray traced (space coherence). When a ray is traced, a test to determine the intersection between a cell and a ray is performed. Only the cells intersected by the ray are considered in the intersection test between the surfaces crossing this cell and the ray. This intersection test is only performed with cells having at least one surface crossing it [25].

However, the traversing of a beam containing a set of rays is different because the same beam can intersect many cells at the same time. 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 vector 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 6.1 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. Every

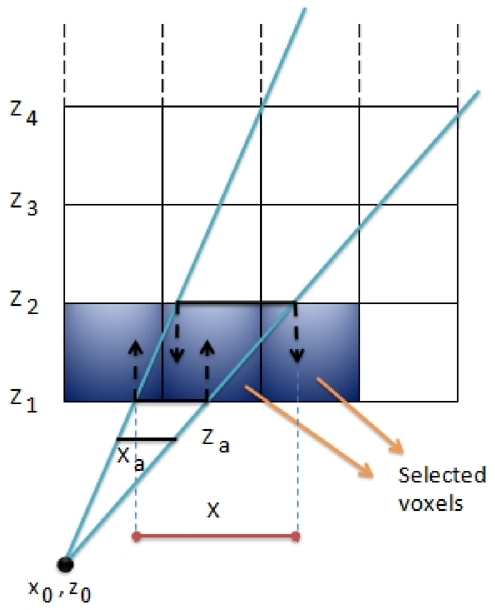

Figure 6.1: A propagation of a beam inside a regular grid structure.

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:

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

having:

$$T = \frac{Z - z_0}{Z_a - z_0}$$

in which X_a, Z_a are the intervals representing the direction of the beam and x_0, y_0 are real values representing the view point. 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 of cells in the grid.

In every cell, the corresponding interval value of the parameter of the beam (T_V) 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 follows:

$$T_V = T_z \wedge T_x$$

where

$$T_z = \frac{V_z - z_0}{Z_a - z_0}$$

and,

$$T_x = \frac{V_x - x_0}{X_a - x_0}$$

in which the cell is defined by the intervals V_x and V_z. Once the first surface intersecting the beam is reached, the algorithm calculates the shading value for the corresponding point in the screen.

6.3 Constructive Solid Geometry among surfaces in a Cell

The creation of scenes composed by implicit surfaces can be performed by means of Constructive Solid Geometry Operations (CSG) or other approaches as in [103], where a method that allows arbitrary compositions of models that make use of blending, warping and Boolean operations is proposed.

There are two ways to perform CSG operations in ray tracing over implicit surfaces. First, the *maximum* and *minimum* operations defined by Ricci [76]. And second, the operations over implicit functions defining volumes as point-sets.

Let $f_1(x, y, z)$ and $f_2(x, y, z)$ be two functions intersected by the same ray. Let t_1 and t_2 be the roots found for the intersection functions for f_1 and f_2 respectively.

For every operation, one of the roots t_1 or t_2 is taken:

- Union : $max(t_1, t_2)$
- Intersection : $min(t_1, t_2)$

In the case of $f_1 - f_2$, the intersection between $f_1(x, y, z)$ and $-f_2(x, y, z)$ is calculated. This is, if t_1 is the root found for f_1 and t_2 is the root found for $-f_2$, then the root taken is $min(t_1, t_2)$.

Ricci operations require an intersection test for every surface to perform *maximum* and *minimum* operations between the roots found for the parameter of the ray.

On the other hand, when volumes are defined, the CSG tree can be evaluated during the intersection test for every surface in the cell. In the algorithm presented in this chapter, the operations will be performed using this approach, although Ricci operations can be easily implemented.

In every subdivision step of the parameter T, the CSG operations must be tested. Having the functions f_1 and f_2, the CSG operations are tested in every subdivision step in the following way:

- Union: $\underline{F_1(T)} \leq [0, 0]$, $or\, \underline{F_2(T)} \leq [0, 0]$
- Intersection: $\underline{F_1(T)} \leq [0, 0]$, $and\, \underline{F_2(T)} \leq [0, 0]$
- Substraction: $\underline{F_1(T)} \leq [0, 0]$, $and\, \overline{F_2(T)} \geq [0, 0]$

Note that the negation operation over a function is represented by the set of points "outside" the surface. This means that the *supremum* of the results of the evaluation of the function, must be checked for values either equal or bigger than zero. Figure 6.2 presents some images generated by means of the CSG operations using volumes as point-sets.

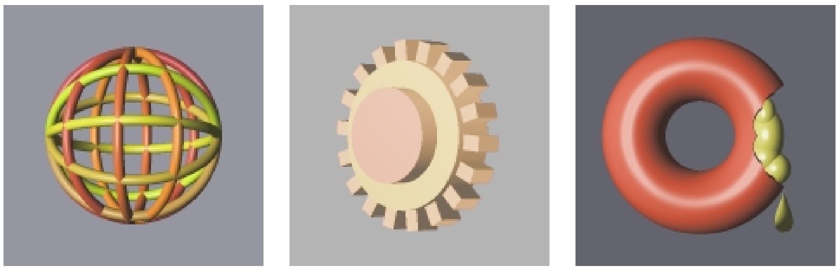

Figure 6.2: Some images created by means of CSG Operations.

6.4 Improvement of the efficiency in animations

As was explained earlier in this thesis, the visualization time of ray traced scenes involving implicit surfaces is high, especially when interval arithmetic is involved. Even without Interval arithmetic, the animation of implicit surfaces suffers of the same problem.

Many techniques to improve the efficiency of animation, exploiting the temporal coherence, have been proposed. Chapman [21] assumes that if a pixel has the same color in the frames a and b, all the frames between them must have the same color. Otherwise, the frame at which the pixel changes the color is searched in between frames. This method could fail in some cases because a pixel could have the same color in two non-consecutive frames, but have a different color between them. Chen *et al.* [22] used morphing techniques to interpolate scenes of a stationary scene in which the view point is moving.

Another way to reduce time in recalculating intersections between rays and objects is the reprojection [8, 49]. In this technique, the intersection points of primary rays in a frame are stored, and then projected to the camera in the following frames. Rayskip approach [28] compares a ray with the neighbor rays and also with rays from the previous frame to search for the portion of the ray that is not likely to hit the surface.

Glassner [39] suggest to convert a 3D dynamic scene to a 4D static one. To achieve this, an acceleration structure is created adding the time as fourth dimension. This is more efficient than calculating a structure for every frame. This method requires a previously knowledge of the changes of the objects during animation.

6.4.1 Using Interval Analysis to exploit temporal coherence

In this section, interval arithmetic is used to define a set of positions for an object in a defined number of frames. The changes over an object that are previously known at every frame r, will be defined using the following notation:

$$(x, y, z) = TR(x_1, y_1, z_1, r)$$

in which x_1, y_1, z_1 are the original coordinates of the object, and TR is the transformation applied to the frame r to obtain the new coordinates x, y, z. A transformation is defined by a 4x4 matrix that can represent a rotation, a scale, a translation or any other transformation that could be represented by a 4x4 matrix. Indeed, this is equivalent to:

$$\begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \mathbf{M} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix}$$

in which M represents the transformation matrix.

The transformation in a range of frames r and $r + n$, in which n represents a defined number of frames, can be defined as:

$$(X, Y, Z) = TR(x_1, y_1, z_1, [r, r + n])$$

in which X, Y, Z are intervals representing all the values of x, y, z obtained in the range of frames defined by the interval $[r, r + n]$.

This allows the creation of a new structure that contains all the spatial positions that the surface will achieve between the frames r to $r + n$.

To apply this technique, the interval $[r, r + n]$ is replaced in the parameter defined for the transformation. That is, in a rotation, $[r, r + n]$ will represent an angle; in a scaling, $[r, r + n]$ will represent a scale factor, and so on. For example, if a surface is rotated 30^o in x in frame r, and 35^o in x in frame $r + 1$, then the corresponding values for a rotation in the transformation matrix are replaced by the interval values:

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \text{cos([30,35])} & \text{-sin([30,35])} & 0 \\ 0 & \text{sin([30,35])} & \text{cos([30,35])} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The surface obtained with this kind of transformation will be called n-surface, that is, a surface defining the spatial positions or changes of a surface in n animation frames simultaneously. This means that it is not necessary to interpolate either the positions of the objects or pixel color between frames, because the method guarantees that the coordinates obtained correspond to the changes performed by the object in the range of frames. For example, if an object performs a fast movement between frames (figure 6.3), then other methods can produce cracks because that behavior is not detected. Using intervals, a fast movement between frames is registered in the interval coordinates. A similar approach is proposed in [39], in which the acceleration structure is converted to 4D, adding the time as another dimension. The main difference with our approach is the application of interval arithmetic and beam tracing to exploit temporal coherence, as is explained in the following section.

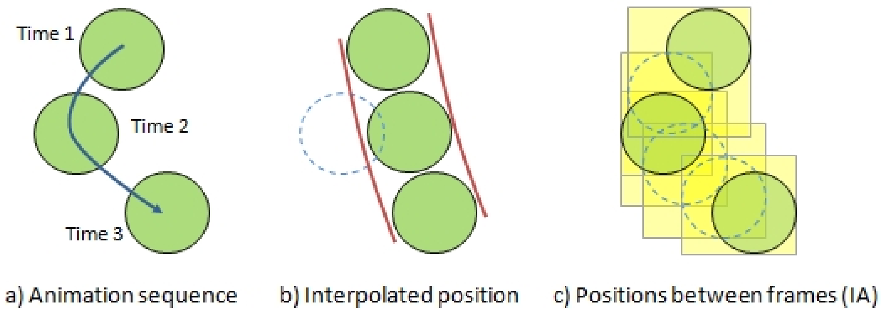

Figure 6.3: Difference between the interpolation of frames using interval arithmetic and other interpolation methods.

6.4.2 Use of a regular Grid for n frames

The algorithm that will be presented in this section requires a regular grid to exploit spatial coherence for the n-surfaces. For a defined number of frames n, only one grid must be used.

Figure 6.4 represents three frames of an animation. In a regular animation, three grids are needed. If interval arithmetic is applied, only one grid must be created for the n-surfaces.

The process starts evaluating the implicit surface with the cell X_c, Y_c, Z_c, but applying the transformation for the interval $[r, r+n]$ to define a n-surface. The transformation must be applied for the values X_c, Y_c, Z_c as follows:

$$(X, Y, Z) = TR(X_c, Y_c, Z_c, [r, r+n])$$

Given an implicit surface $f(x, y, z)$, the cell may contain a part of the n-surface if $0 \in F(X, Y, Z)$, in which

$$F(X, Y, Z)$$

is the united extension of f.

This will create a grid that contains the spatial distribution of cells for the n-surfaces defined in the interval $[r, r+n]$. Once the grid is created, the traversal is performed against the cells containing the n-surfaces. When a beam intersects a cell, the intersection test is performed for every frame, and for the implicit surfaces represented by the n-surfaces crossing the cell. Also, the transformations for the surfaces are performed only

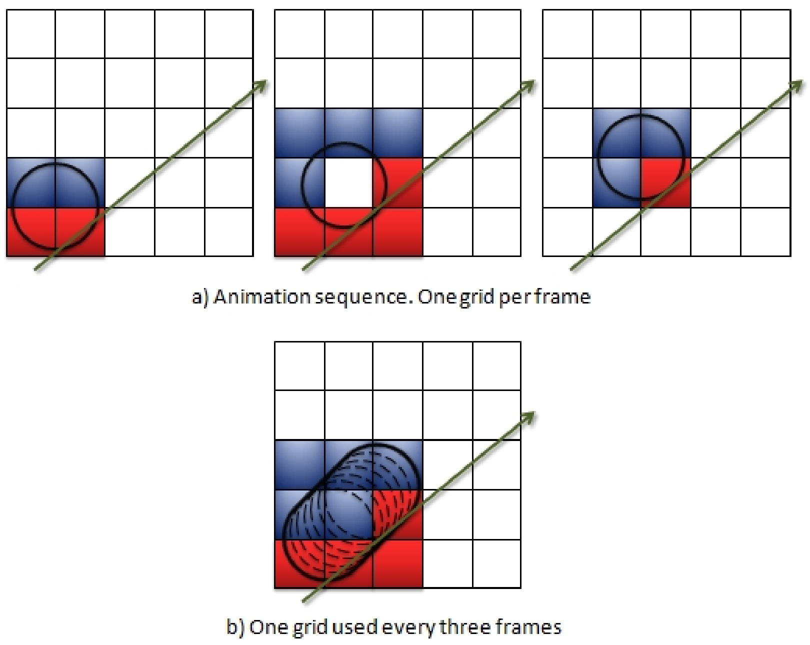
a) Animation sequence. One grid per frame
b) One grid used every three frames

Figure 6.4: Creation of a regular grid for an animated sequence.

for the corresponding frame. The results obtained for every frame are stored in a buffer. At the end of the traversal, the buffers are used to generate the number of frames of the animation represented by the n-surfaces.

One problem of this approach is that a fast movement of a surface can create a big n-surface, for which the beam tracing will decrease the efficiency of the algorithm. To solve this, the algorithm tests the size of the values X, Y, Z for the n-surface, against the values for one of the frames. If the difference is bigger than a desired threshold, the n-surface is rejected for the current number of frames. In this case, the frames corresponding to the n-surface are rendered one by one. However, this problem does not appear in the experimentation presented in this chapter, because a random movement of the objects was not represented.

An example of the traversal process is presented in figure 6.5. The scene is composed of two objects and a light point with different positions in three animation frames. A ray for a pixel (x_p, y_p) of the screen is traced and traversed against the n-surfaces (figure 6.5a). The ray hits the first n-surface, and the intersection test is performed with the surface defined for the first frame (figure 6.5a). Because the ray hits the surface, a shadow ray (with direction to the position of the light for the first frame) is traced against the n-surfaces to look for an intersection. In this case, the point is not in a shadow. Because the point was already found, it is used to calculate the illumination that must be applied to the pixel (x_p, y_p) in the buffer corresponding to the first frame.

The intersection test is then performed for the first surface but in the position defined

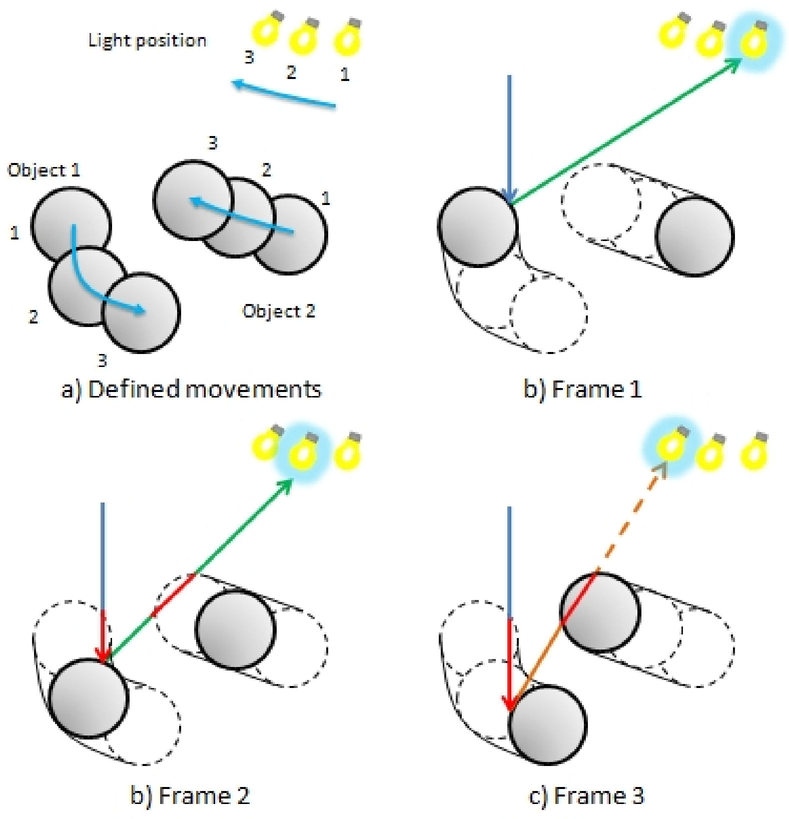

Figure 6.5: A ray traversing a scene (three frames).

for the second frame (figure 6.5b). Because the ray also hits the surface in the second frame, a shadow ray is traced against the light in the position defined for the second frame. In this case, the shadow ray hits the second n-surface, for that reason, the intersection test is performed over the second surface in the position defined for the second frame. This intersection test fails (the shadow ray does not hit the surface). As in the first frame, the pixel (x_p, y_p) of the buffer of the second frame is shaded using the information obtained from the current intersection point.

The case of the third frame is similar to frame two: the ray hits the surface for the third frame and a shadow ray is traced against the light in its position for this frame (figure 6.5c). In this case, the shadow ray intersects the second n-surface, and also the second surface for the position defined for this frame. The pixel (x_p, y_p) of the buffer of the second frame is shaded using the information obtained from this point.

The previous method is efficient because:

1. the regular grid is not created for every frame

2. only one beam is traced for all the corresponding pixels of the frames defined in the interval $[r, r+n]$

With 1 we save the time required to create the grid, which is negligible if the range of frames n is small. With 2 we avoid the time required to create a new beam and to traverse it in the grid. To achieve the full efficiency of the algorithm presented, the view point must be static. This is not a problem because the movement of the view point can be applied directly to the scene (OpenGL works the same way). The effect obtained is the same in both cases (figure 6.6).

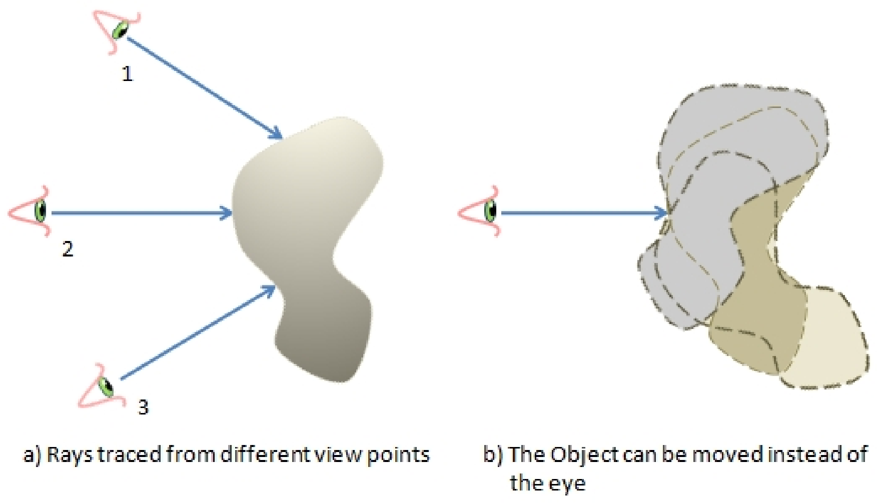
1
2
3
a) Rays traced from different view points
b) The Object can be moved instead of the eye

Figure 6.6: Representations of the movement of the view point in a scene.

6.5 Experimentation

The algorithm was tested for an animation of an orthocircle surface defined as:

$$((x^2+y^2-1)^2+z^2)((y^2+z^2-1)^2+x^2)((z^2+x^2-1)^2+y^2)-(\mathbf{a}^2)(1+\mathbf{b}(x^2+y^2+z^2))=0$$

in which the parameters **a** and **b** can take different values in every frame to change the shape of the surface.

Also, both the position of the surface and the camera are changed during visualization (see figure 6.7).

The test was performed using 5 frames of the animation to create the n-surface. Every 5 frames the algorithm creates a grid to perform the ray tracing for the five images corresponding to the frame.

The method was tested against the same animation, using one grid for every frame. In both cases, ray tracing is performed by means of beams, and Interval Arithmetic to keep reliability (see chapter 5). Results of the comparison are presented in figure 6.8. The rendering time is improved between 40% and 54%. The quality of the images obtained is the same, using acceleration and rendering the frames independently.

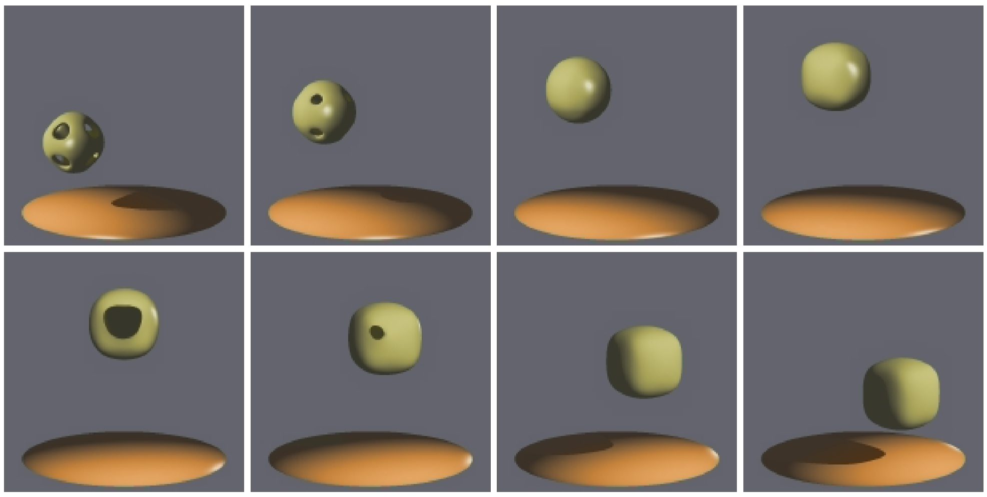

Figure 6.7: Eight frames of the animation of the orthocircle surface (from left to right and top to bottom).

In a second example, a set of spheres moving out from the center of the scene was rendered (see figure 6.9). The conditions were the same that in the last experiment. In this case the improvement was between 32% and 66%. According to the time results (figure 6.10), the improvement is not as regular as the previous experiment, because more objects are rendered in the last frames.

For this experiment, we measured the time used for the grid creation. For an animation frame to frame it was between 14 and 18 seconds for each frame. For a n-surface it was between 15 and 22 seconds for n equal 5. This means that we can save at least 65 seconds every 5 frames for the grid creation process.

The visual results are the same when the animation is created frame by frame or using n-surfaces.

6.5.1 An extreme experiment

In this experiment, an animation with a fast change between frames is tested. In this case, a morph between a chubs surface and a Mc Mullen k3 surface is performed. The color is also changed in a small value for every frame. In figure 6.11 it is possible to see the big rate of change in only eight consecutive frames. Also, the surface is rotated in x and y and the point of light moved. This experiment fails in approaches based in interpolation, because the color of every pixel is different between frames (although it is hard to see it directly, the parameter is changed in a small value in every frame). Also, the change of almost every part of the surface between frames makes difficult to use any interpolation approach.

Figure 6.12 shows the result of the application of the n-surfaces compared by a frame-to-frame approach. For this case, the efficiency obtained with the n-surfaces is 10%.

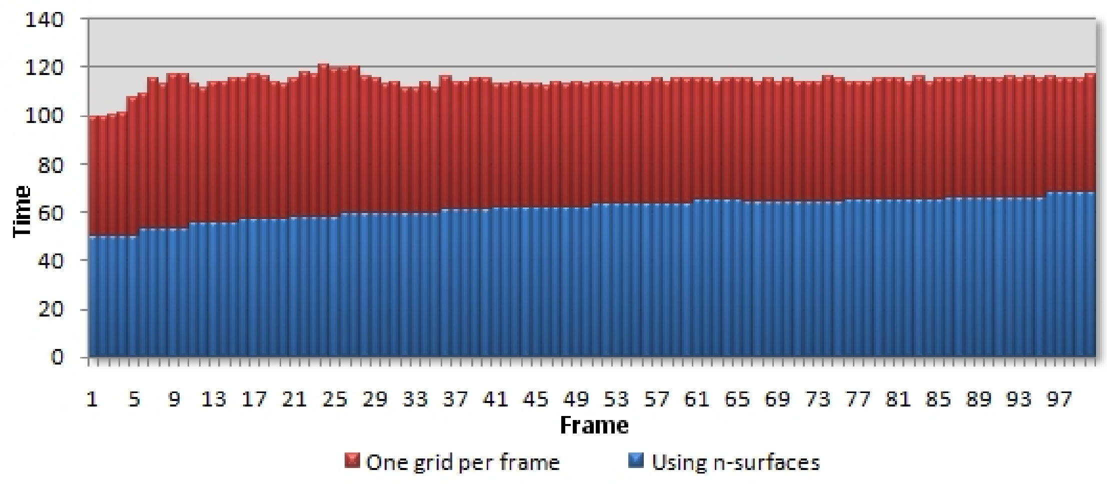

Figure 6.8: Results for the animation of the orthocircle (100 frames).

This means that even in this complex example, the method presented here obtain some improvement over the frame-to-frame version.

6.6 Conclusions

This chapter introduce an application of the concepts introduced in this thesis in the animation of scenes composed of implicit surfaces. This method has two drawbacks: the movement of the surfaces must be known in advance, to compute the n-surface, and the efficiency decreases when the change between surface positions is high.

Interval arithmetic can be used to exploit coherence between consecutive frames, reducing the rendering time. Although the method was tested for beams, it is expected to obtain similar results using rays.

Also, a technique to exploit spatial coherence was introduced. This was explained by means of a regular grid, although it can be applied to any other kind of structure, like an octree or bounding boxes.

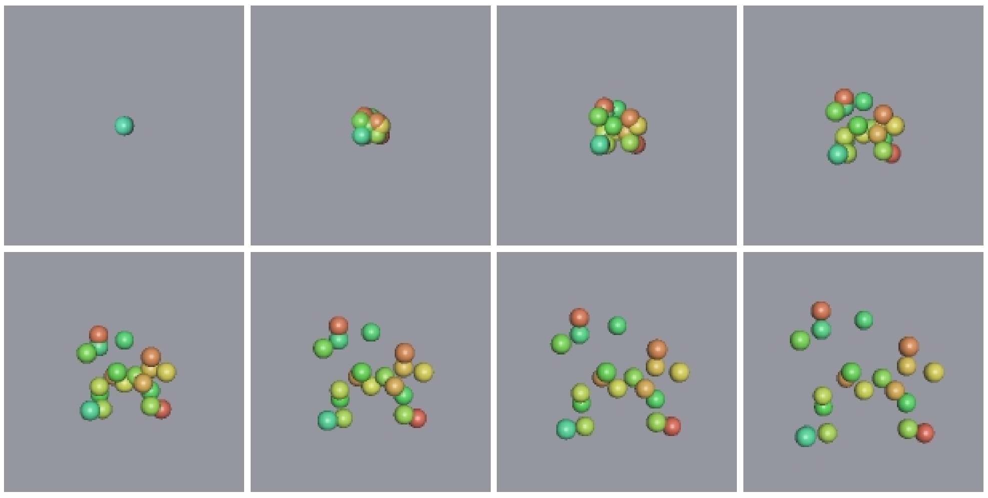

Figure 6.9: Eight frames of the animation of the spheres emerging from the center of the scene (from left to right and top to bottom).

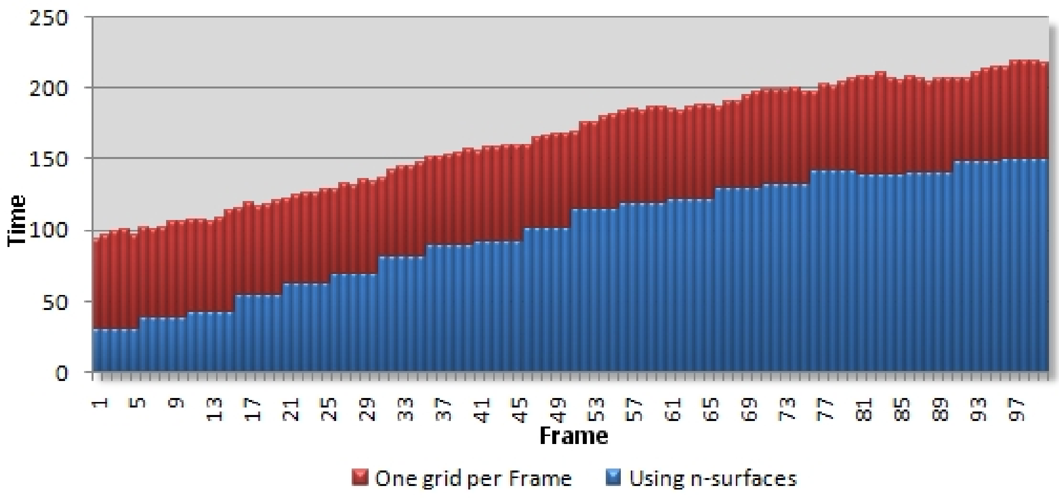

Figure 6.10: Results for the animation of the spheres (100 frames).

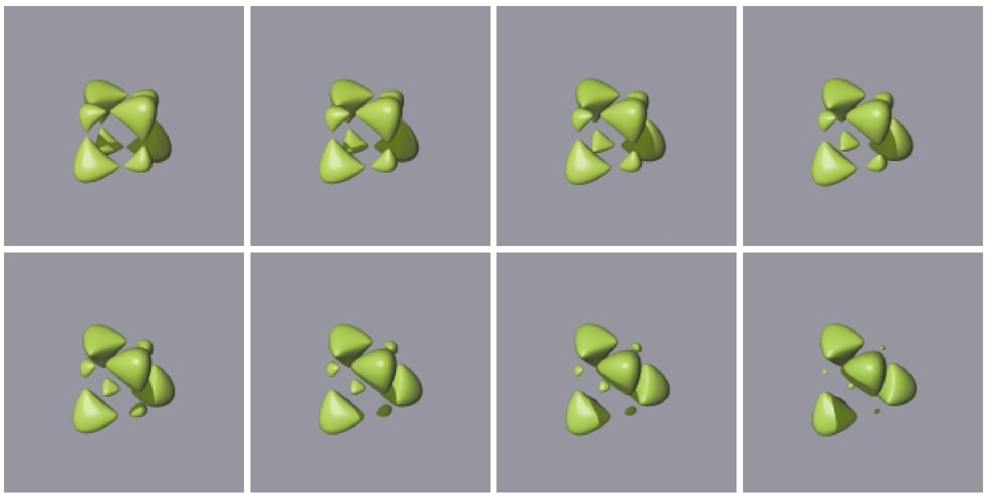
Figure 6.11: Eight consecutive frames of the morph (from left to right and top to bottom).

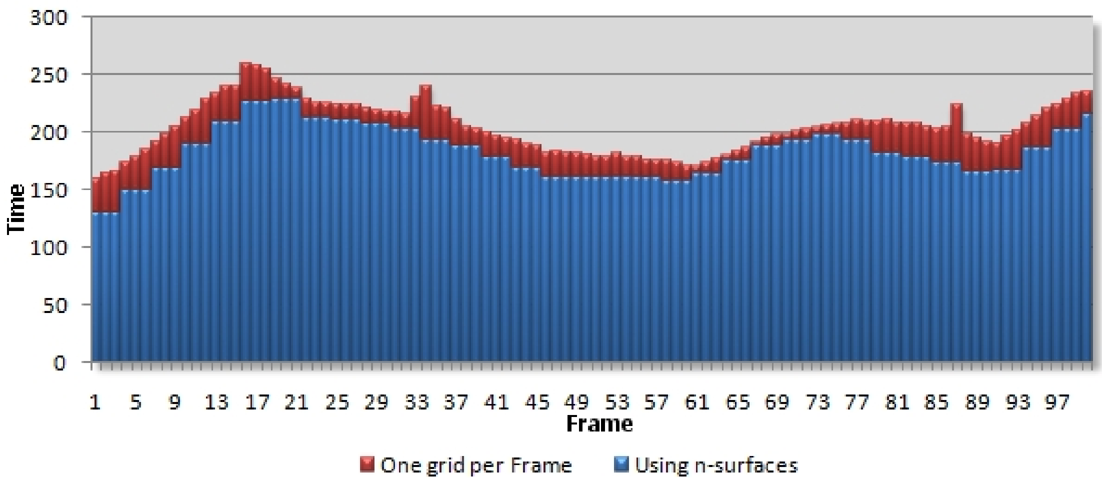

Figure 6.12: Results for the morph animation (100 frames).

7 Parallel and GPU Improvements for the Interval Ray Tracing of Implicit Surfaces

7.1 Introduction

This chapter presents two new versions of the reliable algorithms for ray tracing implicit surfaces. The first is a parallel version running in a pc cluster. There are lots of ray tracing versions for parallel systems in the literature. However, the contribution here is a reliable version using interval analysis to take advantage of coherency, which permits the evaluation of many rays simultaneously.

The second version is implemented over a Graphics Processing Unit (GPU). The GPUs are hardware devoted to mathematical-intensive operations related to 3D graphics. The modern ones have special units that allow programmable capabilities within the GPU. This feature has been used to create general purpose applications, as ray tracing. There are many recent applications of ray tracing performed on a GPU, but few devoted to implicit surfaces, and none of them takes care of reliable things like interval arithmetic using correct roundings. In this chapter, a fully reliable version is introduced, being a main contribution of the GPU implementation.

7.2 Parallel and reliable ray tracing of Implicit Surfaces

Ray tracing is an algorithm that can be easily parallelized because every ray can be treated independently of the others. Ray tracing requires a high computation cost, for that reason, many parallel approaches have been proposed to distribute the work in the processors, reducing the entire computation time.

As was pointed out in previous chapters, ray tracing of implicit surfaces based on Interval Arithmetic is computationally expensive. Although a well designed sequential algorithm can improve the efficiency, it is desirable to take also advantage of a parallel architecture to improve even more the computation time.

This section includes a parallel version of a ray tracing algorithm for implicit surfaces which takes advantage of the image coherence to reduce communication among processors. Although the idea of parallelism implies certain independence of rays, we extend this idea by means of the sharing of beams using sets of rays instead of individual ones.

In this way, a more homogeneous task is sent to every node, which guarantees an efficient use of the processors in the system.

7.2.1 Approaches to create a parallel ray tracing

There are two main approaches to create a parallel version of a ray tracing process: the image space subdivision in which the pixels are distributed among the processors and object space subdivision, in which the different objects composing the scene are distributed.

In the *image space subdivision*, the pixels of the screen are subdivided in many independent domains which are distributed in every processor available in the system. One characteristic of this approach is that every processor must have access to all the objects of the scene to trace the rays over them. A solution is to store the information of the scene in every node of the system. Due to the high memory capacity of the modern computers and the use of hardware systems with shared memory, the copy of the scene can be accessed by every computer in the system [58, 7, 63]. The image can be distributed in connected areas [53] or unconnected like sequences of lines of the screen [38].

The *Object space subdivision* tries to solve the problem of distributing and sharing a complex scene in all the processors. If there is a big enough scene that it is not possible to be saved in the memory of one computer, the scene must be distributed between the different processors. The scene is subdivided in small parts which are processed individually an the results are joined in a final image [24, 60, 74, 68, 46].

The objective here in this chapter is to implement a parallel ray tracing algorithm based on the ideas presented in the previous chapter, for that reason, the parallel approach must facilitate the distribution of areas of the screen instead of individual rays or parts of the scene. In that sense, it seems obvious to choose an image space subdivision approach. That method supposes that the scene is distributed and accessible for all the processors in the system. We will suppose that every workstation involved in the process has enough memory to store all the scene, which consists in complex surfaces or groups of them.

7.2.2 Algorithm specification

The data to be processed consists in areas of the image plane (in this case the computer screen) which has to be subdivided recursively, and every box generated must be evaluated to know if the rays intersect the implicit surface.

The algorithm that performs image space subdivision was parallelized over a cluster with four workstations. Every workstation has two processors, that is, eight processors are available in the virtual network. In this approach, every processor will share areas of screen representing sets of rays. Also, the surfaces must be replicated in every computer in the network. There are processors distributed in different computers, for that reason, a dynamic load balancing strategy is proposed to distribute work in different computers, and a multi-thread approach is applied in every computer, in which the processors share memory.

The evaluation of areas is performed in every processor in which some regions without rays intersecting the implicit surface are quickly eliminated reducing the time required by an evaluation pixel by pixel. The boxes that still contain pixels with rays intersecting the surface are recursively subdivided and shared between processors until a desired precision is achieved. The final boxes which contain the pixels that represent the surface, must be ray traced pixel by pixel using an interval ray tracing algorithm.

7.2.3 Communication strategy

The model introduced is entirely distributed. In the selected model, every workstation is connected with the others (see figure 7.1). This means that the messages are delivered directly between workstations instead of processors. This configuration reduces the distance path of the messages among processors, which facilitates the balancing of work. On the other hand, this kind of configuration could increase the amount of messages in the network. However, due to the hardware availability in which the communication is performed between four workstations, this situation does not affect the load balancing performance at all.

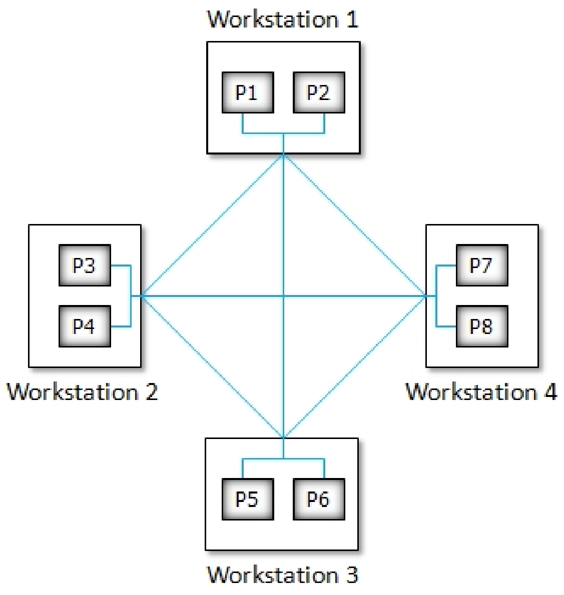

Figure 7.1: System connection used in the communication strategy.

Every workstation manages a list of boxes in memory which is shared between the two processors. This unique list facilitates the distribution of the work among the workstations. The process starts with some set of the rays assigned to every workstation. The set is determined by a rectangular area or box of the screen. This initial box is subdivided and the new boxes added to the list in memory. Although the term “box” generally implies a 3D structure, in this chapter box will imply a rectangle in the screen. This assumption is taken from interval analysis, in which the therm “box” can imply 2D or 3D.

There is one thread running on each processor. Every thread takes boxes from the back of the list and starts a bisection process to generate small boxes. This process continues until the boxes achieve a small size; the size allowed for a box in this algorithm is 0.025 of the initial screen size. During the process, every box is evaluated to know if that box has rays intersecting the surface. If that is not the case then the box is not put in the list, that is, the box is rejected. This means that only boxes with rays intersecting the surfaces are shared between processors.

To know which boxes have to be rejected, the rejection test 5.1 is used. With the values of X and Y of every box, a branch-and-bound over parameter T is started until a small precision is achieved. The algorithm as works in each workstation of the cluster is presented in figure 7.2.

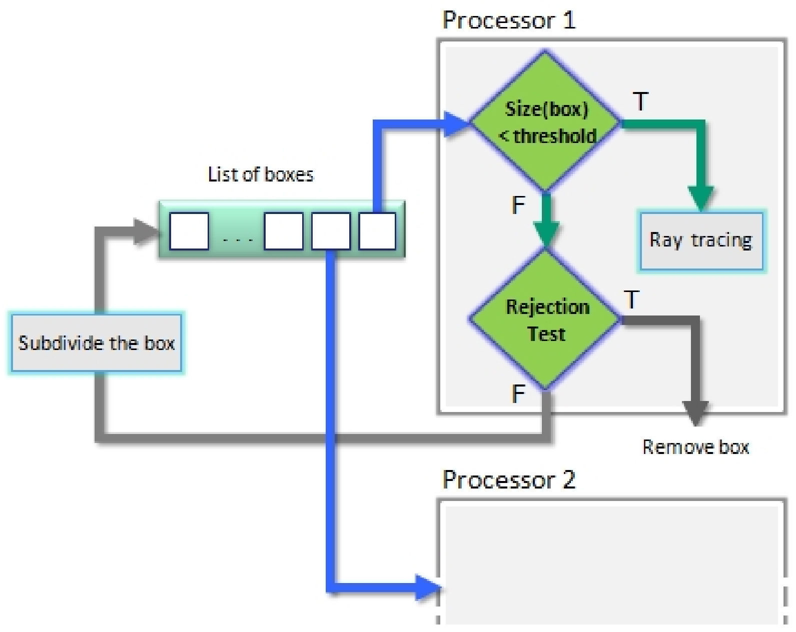

Figure 7.2: Algorithm working in each workstation.

7.2.4 Dynamic load balancing

Transferred data

The initial data of the process are the entire set of pixels in the screen. To share the work among the processors, the screen is subdivided in four equal parts and every part sent to any of the available workstations. The data shared with the processors are the boxes of the list in memory of every workstation. The boxes contain information about x and y coordinates, and the current minimum value of the parameter T for the set of pixels of the screen that the box represents.

Workload placement strategy

Both processors in each workstation have the information about the state of the list of boxes, but they do not know the state of the other processors in other workstations. The only available information about the other processors is the route to communicate with the other workstations. For that reason, the only decision that a processor could take is to offer help to the other processors when it knows that it will soon become idle.

The length of the list is the index used by the algorithm to detect when the processor is close to being idle. This occurs when a list has less boxes than some predefined threshold. In this study case, the threshold is two boxes, which is enough to guarantee that the processor does not achieve the idle state. This index is good because it does not need to be recalculated, so the computation involved in this task is inexpensive.

Also, it is not necessary to keep equal amounts of work in all the processors, because the only need is that they do not become idle at any moment to obtain full occupation during all the process. For that reason, it is not necessary to calculate a complex index.

When a pair of processors in a workstation are approaching the idle state, the first processor in detecting the situation sends a request for work to the other workstations and waits for some boxes. When a processor in a workstation receives this request, it sends the 40% of its boxes. This value was obtained by experimentation: the best efficiency was obtained in the majority of the tests with this value.

Global termination

If the list of boxes of a given workstation is empty, and this workstation does not receive any additional work from the other ones, then we suppose that all workstations are close to finishing their work. The given workstation stops the request process, and sends a message to inform that it has terminated its work.

In this situation, it could occur that the processor is idle while others have some work to do, but a distribution of work when all processors are finishing their works is more expensive than to make the processor wait for the others. In the experiments, the idle state does not take more than one second using this approach.

The message to inform the termination of the work is sent to one of the workstations, because only one workstation is defined to collect the final results. When this workstation has finished its work and has received the messages of termination from the other workstations, the independent results are received and joined to shape the final complete visualization.

7.2.5 Dynamic load balancing performance

To evaluate the performance of the Load Balancing algorithm, the Steiner surface has been selected to develop the test. The screen was subdivided into four sections, and each one was sent to a workstation: left-up for workstation 1, right-up to workstation 2, left-down to workstation 3 and right-down to workstation 4. The rendering process was started over the four workstations independently, that is, the workstations do not communicate with each other.

The times obtained in the evaluation of separate sections were different as was expected, even when the surface was symmetrically subdivided. The left-up section takes the more computation time because it contains a ray that emerges from the center of the surface that have to be ray traced, which means this section has more pixels to be processed. The right-down section spends the least time in all the process. The distribution of the boxes after this preliminary test is presented in figure 7.3.

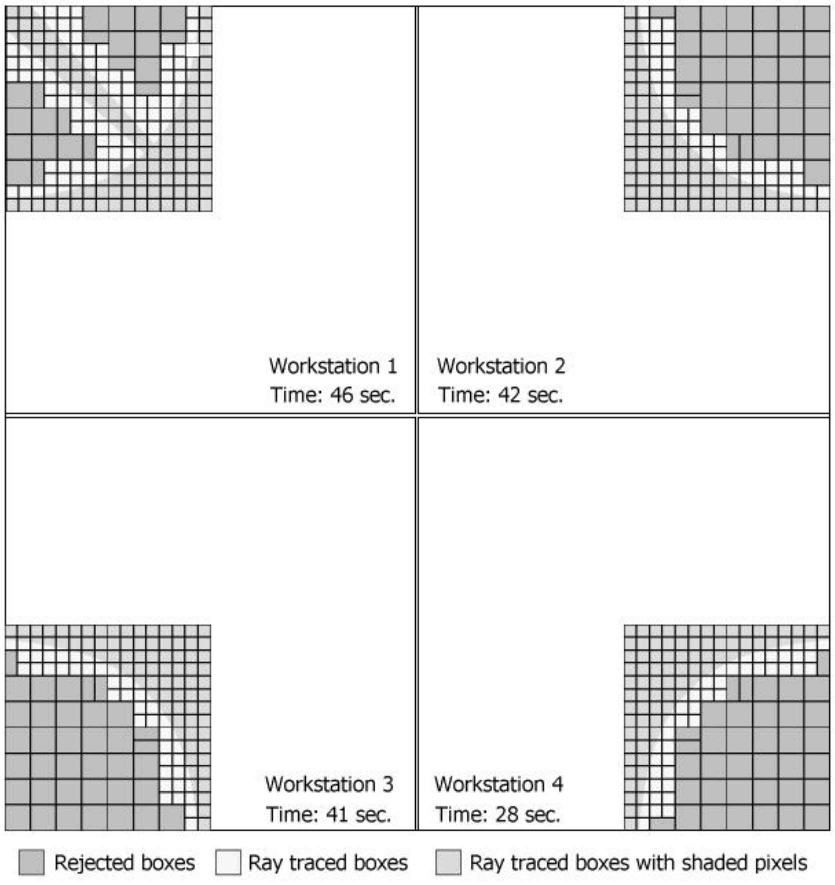

Figure 7.3: A parallel work without Dynamic Load Balancing.

The Steiner surface is visualized in 318 seconds using a sequential version of the algorithm. Using the Load Balancing strategy, with four workstations with 2 processors each one, the time to process the surface is 39,5 seconds. The four sections take the same time in the process. The workstation 4 processed all the boxes assigned to its section, and even some boxes from the other three sections of the screen. Workstations 2 and 3 processed only a part of its assigned boxes, but also some boxes of the workstation 1, which was the busiest one (see figure 7.4).

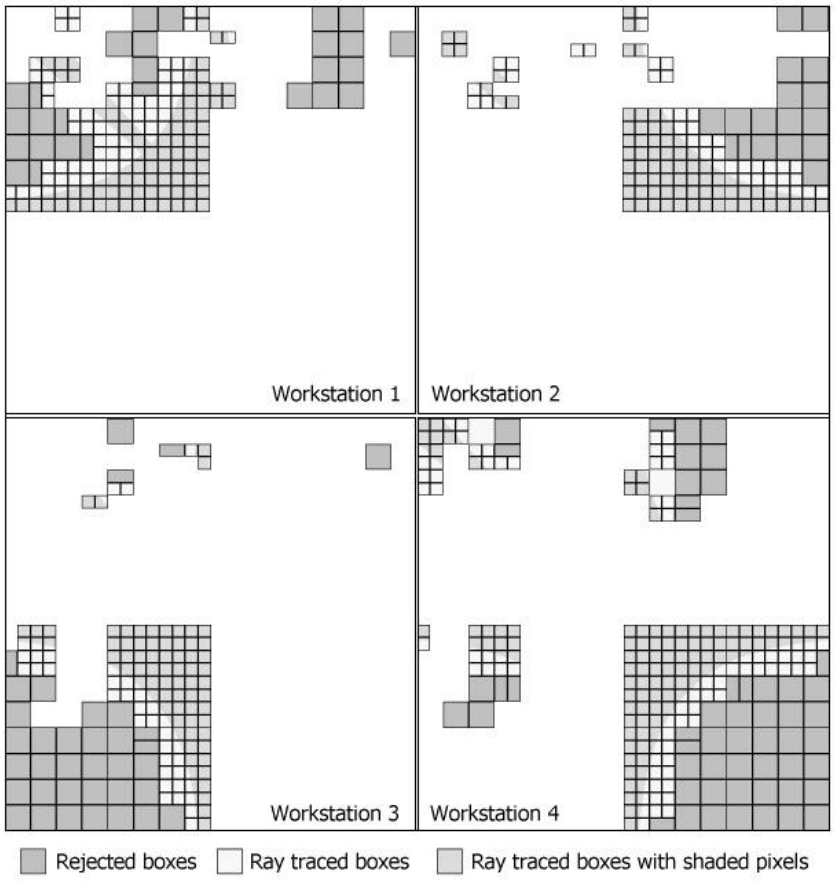

Figure 7.4: Distribution of the boxes using a Dynamic Load Balancing strategy.

7.2.6 Results

The algorithm was implemented in C++ using socket programming. The algorithm runs in four HP Proliant DL145 workstations with two AMD Opteron processors of 2.6 GHz. each one. The computers are connected by a switched 100 MBit Ethernet.

The algorithm was tested over four surfaces: orthocircle, steiner, bath and weird. The figure 7.5 illustrates the graphical result of the parallel algorithm over the tested surfaces.

The complexity of the surfaces is considered according to the time that the sequential version of the algorithm takes to visualize them. The less complex surface is the orthocircle, and the more complex is the weird surface.

The parallel algorithm was tested using the proposed surfaces. The table 7.1 shows the results using one, two, four and eight processors respectively.

Table 7.1: Times for one, two, four and eight processors in the system.

Surface	1 proc.	2 proc.	4 proc.	8 proc.
Orthocircle	119	60	31.8	15.2
Steiner	318	161.2	80.4	40
Bath's sextic	2210	1107	553	276.8
Weird	2634	1321	660.6	332.2

To evaluate the algorithm, the speed-up and the efficiency were calculated (see table

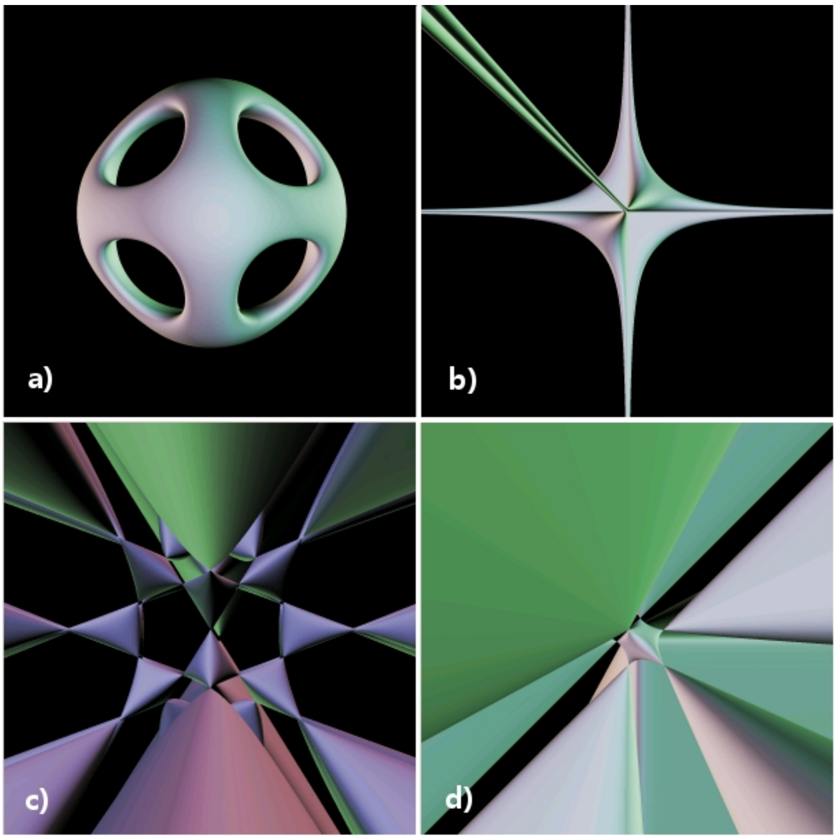

Figure 7.5: Tested surfaces: a) orthochircle, b) steiner, c) bath, d) weird.

7.2). The speed-up is calculated as the ratio between the time to run the algorithm in one processor and the time taken to solve the same problem over more processors (The table shows those values for 2, 4 and 8 processors). In this experiment, the algorithm has a speed-up almost linear because idle states barely occurs due to dynamic load balancing algorithm. The efficiency obtained for eight processors is from 97% to 99% as is shown in the last column of the table. That means that 97% to 99% of the time the processors were performing useful computation on the boxes, and 1% to 3% of the time is spent in other tasks such as communication, list management and also idle states.

Table 7.2: Speed-up and efficiency for 2,4 and 8 processors, measured for the four surfaces.

	2 proc.		4 proc.		8 proc.	
Surface	Speed-up	Efficiency	Speed-up	Efficiency	Speed-up	Efficiency
Orthocircle	1.983	99.17	3.742	93.55	7.829	97.86
Steiner	1.973	98.64	3.955	98.88	7.950	99.38
Bath's sextic	1.996	99.82	3.996	99.91	7.984	99.80
Weird	1.994	99.70	3.987	99.68	7.929	99.11

7.3 Reliable Ray tracing on the GPU

Ray tracing polygonal meshes on the GPU is not a new issue [23, 19]. There are also works over implicit surfaces on the GPU [102], but when this thesis was written, there was only one work which implements a ray tracing of implicit surfaces using Interval Arithmetic on the GPU [59]. However, the interval library used in [59] does not take care of the rounding. As was explained in chapter 3, rounding is essential to keep the reliability during the evaluation of the implicit function. This is even important in the modern GPUs because they use only single precision in the floating point operations. This section includes the description of an algorithm that performs a completely reliable ray tracing in the GPU.

7.3.1 GPU overview

The GPU (Graphics Processing Unit) is a special hardware devoted to accelerate graphic processing. The GPU hardware works in a different way that CPU. GPUs were conceived as hardware accelerators for graphics programming interfaces like Direct X and OpenGL. GPUs are based in a dataflow model in which there are sequential series of steps. This model is known as graphic pipeline. A scene described by vertices (in OpenGL or Direct X) is taken as input of the pipeline. The output from one step is taken as input of another, and the final output of the process is an image made by pixels. Also, the computation is performed in parallel in an array of fragments, in which every fragment generally corresponds to a final pixel in the screen. The general schema of the GPU pipeline is presented in figure 7.6.

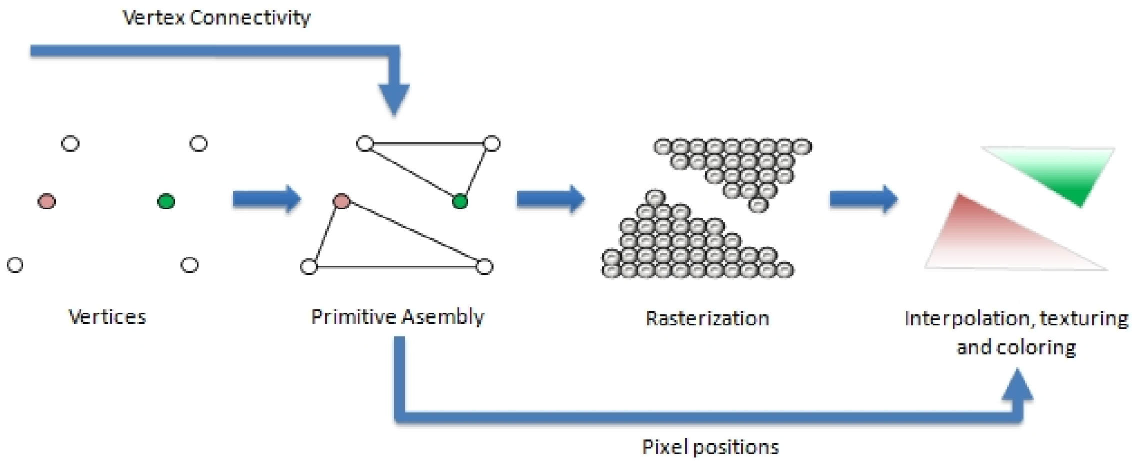

Figure 7.6: Graphic pipeline of the GPU.

Before primitive assembly and also raster operations there are three programmable units which allow a more flexible use of the graphic pipeline. The vertex processor works on properties of vertices like color, position and texture coordinates. The fragment processor can perform operations to modify fragments by accessing the interpolated values obtained during rasterization. Also, the last models of graphic cards include a geometry shader, which is executed after the vertex shader. The geometric shader can generate a new primitive or even eliminate an existing one. This shader works over the vertices that were sent at the beginning of the graphic pipeline.

7.3.2 Interval Arithmetic Library on GPU

The main problem in ray tracing using interval arithmetic on the GPU is the implementation of rounding operations. Fortunately, there are many studies of how arithmetic operators work in the graphics hardware [42, 31]. Because the behavior of those operations have variations depending of the manufacturer and the model, we had to select one. However, similar analysis can be performed for any other model in order to implement the interval operations using a correct rounding. In this chapter, a GeForce 8800 GTX model from Nvidia was selected for the development of the interval library.

The GeForce 8800 GTX has 16 blocks that can perform SIMD floating point operations such as addition and multiplication in single precision, but rounding toward plus infinity or less infinity is not supported. Every block can execute a pack of 32 floating point additions, multiplications, multiply-and-adds or integer additions, bitwise operations, comparison, or evaluations of the minimum or maximum in 4 clock cycles. As there is no 32-bits integer multiplication in hardware, evaluating such operation requires 16 clock cycles for a pack.

To perform the ray tracing process, the Interval library was developed in the CG language. This language is similar to the C language, and was created to facilitate the programming of graphic cards. Before CG, the programming of the graphic cards was performed in assembler language.

Every interval operation was programmed as an independent function in which the interval values are vectors of two float values. The advantage of CG is the portability to the graphic cards of the two main vendors: Nvidia and AMD ATI. In our implementation, the interval operations do not run in parallel; only ray tracing process was implemented to run in parallel.

7.3.3 Rounding

GPUs do not support rounding modes toward plus or minus infinity which is required for interval arithmetic operations. To perform rounding, a ULP (Unit in the Last Place) must be subtracted or added to the results of the operations. According to Kahan [56], a ULP(x) is "the gap between the two finite floating-point numbers nearest x, even if x is one of them".

According to the studies performed by Collange *et al.* [26] for the GeForce 8800 GTX, the rounding can be obtained by the multiplication of the results by $1+2^{-23}$ for positive infinity, and by $1-2^{-23}$ for rounding to negative infinity. However, the solution can be lost when underflow occurs; in that case, the result returned is 0.

The Addition operation is presented in figure 7.7 as an example. Addition has faithful rounding, that is, the error is strictly less than 1 ULP. First, the addition is performed between two input parameters (vectors of two float values) in step 2. In step 3 and 4, the values for the rounding to positive infinity $1+2^{-23}$ and negative infinity $1-2^{-23}$ are defined. In 5 and 6, the rounding is obtained by the multiplication of the previous result and the rounding values. Here, the minimum and maximum results of the operations are calculated to prevent errors like overflow.

7.3.4 Intersection test

The intersection test is performed by means of a program running in the fragment shader. This program contains the instructions to perform the intersection test in every pixel of the screen. The rejection test is performed as it was described in section 5.4.2. This means that a beam is used instead of individual rays. The root is found by means of an interval bisection, which is implemented using a ping-pong strategy to perform the iterations of the algorithm. This strategy can be performed using Frame Buffer Objects (FBO), rendering the intermediate results to textures [42]. The ping-pong consists in a recursive rendering between two textures, in which the output of the processing of all the fragments for the first texture is put in a second one, and the output of the processing over the second texture is again put in the first one, and so on.

In general applications on GPU, the textures are used as two dimensional arrays that store float variables. Every position of the textures contains four channels (red, green, blue and alpha channels) that represents the color for a potential pixel. A float value can be stored in every channel; those values are used to store the information needed to calculate the intersection point in every pixel.

In the algorithm presented in this section, these textures must cover all the pixels of the screen, in this way, a fragment will be used for every pixel in the screen. The ping-pong process is presented in the figure 7.8.

At each iteration, the interval T must be bisected and each new section must be evaluated. The evaluation instructions are the same for each interval T, and these instructions remain in the instruction cache of the GPU. The position of every pixel in the screen corresponds to every position of the texture. The position information is available during the execution of the fragment program, which is used to determine the pixel that is being processed.

The values for the parameter T are also stored in every position of the texture. The red channel (R) is used to store the number of subdivisions performed over the parameter T in the current fragment, in power of two. This means that a one in the red channel indicates that the parameter is currently subdivided in two, a value of two indicates a subdivision in four sections and so on. The value of the green channel (G) indicates the subdivision currently evaluated in the current level of subdivision.

A value less than one in R or G indicates that the current fragment is inactive. This occurs during the intersection test in a fragment, when a root is not found. Those fragments are no longer processed, while processes in other fragments can be still looking for roots.

The calculation of the current value of T used to evaluate the implicit function is as follows:

$$T = [\frac{maxT}{2^R}(G-1)\ ,\ \frac{maxT}{2^R}(G)] \tag{7.1}$$

where $maxT$ represents the size of parameter T defined at the beginning of the process.

```
float2 sumI(float2 I1, float2 I2) {
    float2 result;
    result = I1 + I2;
    float one_minus_2_23=1-1.0/8388608;
    float one_plus_2_23=1+1.0/8388608;
    float2 to_zero=result*one_minus_2_23;
    float2 to_inf=result*one_plus_2_23;
    float lower=min(to_zero.x,to_inf.x);
    float upper=max(to_zero.y,to_inf.y);
    return float2(lower,upper);
}
```

Figure 7.7: Addition operation with rounding.

This value of T and the current values of R and G for the fragment are used to evaluate the implicit function.

The evaluation is performed for every position of the texture individually. The process starts with a one in R and a one in G, which indicates that at the beginning, the parameter T is bisected and the first section is evaluated. The algorithm for the intersection test in GPU, is presented in figure 7.9.

The ping-pong process is performed in a fixed number of steps. This number of steps can vary from surface to surface, but, in all the experiments that we performed, 100 steps was enough to obtain a correct visualization of the surfaces.

The fragment process finishes when: a) every fragment is processed until the number of subdivisions is reached, or b) when the process for the fragment does not find roots, or c) when the difference between the maximum and the minimum value of the current evaluated section of the parameter T is smaller than 0.00001. In the last case, the values of R and G are set to $R = -R$ and $G = -G$ to indicate that this fragment does not need to be processed anymore. A conditional at the beginning of the process evaluates that flag: if the red or green channel have negative numbers, the process is avoided for the fragment.

Once the ping-pong process finishes, the values of T for every fragment are calculated using the values in R and G (see equation 7.1). If G or R have negative values (in the case of fragments that found a root), those values are taken as positive. The shading

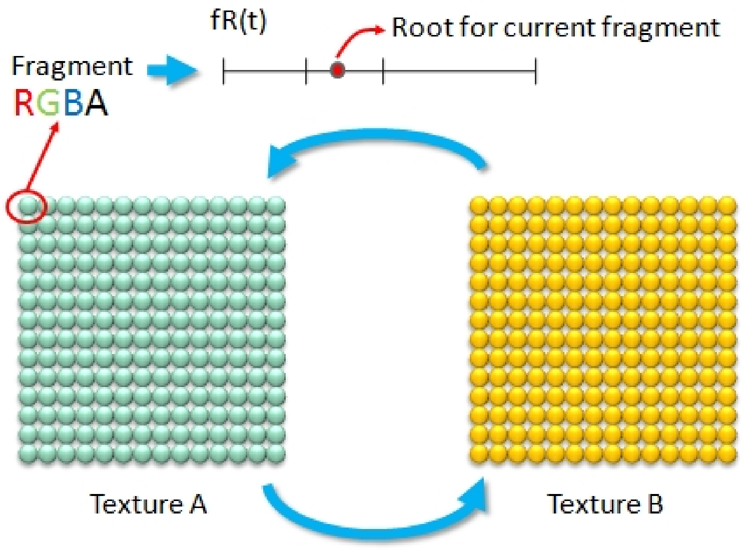

Figure 7.8: A ping-pong algorithm is used to perform the evaluation of the implicit function. Every position of the texture contains the information needed to calculate the intersection test.

value can be determined using the final values of T. This last step is done in another fragment program, executed at the end of the ping-pong process.

An example of the subdivision process is presented in figure 7.10

If the currently evaluated position does not contain any roots (case 1 and 4), then the level of subdivision remains unchanged, and the position value is incremented by one. When an evaluated section contains roots (case 2, 3 and 5), that section must be subdivided and the new sections evaluated. This means that the value of R is incremented by 1, and G becomes equal to $G * 2 - 1$, which puts the current level of subdivision in the next section of T that has not been evaluated.

In case a given section does not include any root, a test is done to determine whether the given section correspond to the first or second part of the interval T. In the first case, the value of R is incremented in 1 so that the next subdivision will be evaluated at the next step. In the second case, the next section of the previous bisection must be

```
  Algorithm for the Fragment Program
$\rightarrow$ ($R$=Red channel; $G$=Green channel)

        If $R < 1$ or $G < 1$
            Fragment inactive: leave process
        Endif
        $T = [(maxT/2^R) * (G - 1), (maxT/2^R) * (G)]$
        If width($T$)<0.00001
            Inactivate the fragment ($R = -R$, $G = -G$)
        Else
            Calculate $X_p$,$Y_p$ using position of the fragment
            If $0 \in fR(X_p, Y_p, T)$
                $R = R + 1$
                $G = G * 2 - 1$
            Else
                If $G\%2 = 0$
                    $R = R - 1$
                    $G = G/2 + 1$
                    If $G > 2^R$
                        Discard the fragment($G = 0$, $R = 0$)
                    Endif
                Else
                    $G = G + 1$
                Endif
            Endif
        Endif
```

Figure 7.9: Fragment program used to perform the intersection test.

evaluated. The value of R is decremented, and G is updated.

7.3.5 Anti-aliasing approach

To improve the quality of the generated images, more rays can be traced at the borders of the surfaces, in which aliasing is more noticeable. The rejection and inclusion test are used to determine these borders. As was explained in previous chapters, rejection test is used to detect beams that miss the surface, and the inclusion test is used to detect beams that completely intersect the surfaces. Beams that do not fulfill both test, are on the borders of the surface, that is, the beam contains rays intersecting and rays that do not intersect the surface.

The anti aliasing is performed by means of a super sampling; in this technique, every pixel is traced with a constant number of rays, and the color calculated for every ray intersecting the surface is averaged to obtain the final color of the pixel. In this case, a beam is used to represent a set of rays.

A regular texture can have 1024x1024 positions that can be used to represent a beam. At the beginning, only one fragment will represent a set of rays, for that reason, some fragments will not be used, saving computation effort. This idea is presented in figure 7.11.

The process start reserving a set of fragments for every pixel. The results presented in the next section were obtained using 9 fragments per pixel. At the beginning of the process, only one fragment is used (figure 7.11a). This fragment will process the intersection test for a beam crossing all the area of the pixel. According to the algorithm used for the intersection test presented in last section (see figure 7.9), doing $G = 0$ and $R = 0$ is enough to inactivate a fragment.

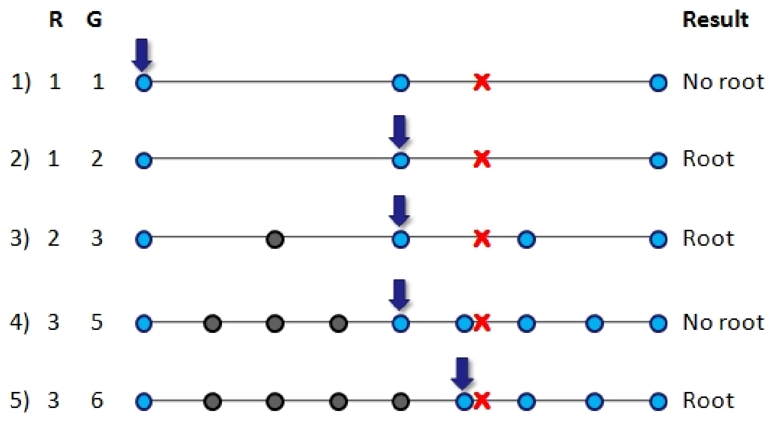

Figure 7.10: Subdivision process performed over every fragment. The values of subdivision and position are stored in a texture. The red cross indicates the position of the root.

If the beam is crossing a border of the surface, the current fragment must be re-initialized to process a smaller area of the pixel. Other fragments must be also activated to process its corresponding areas of the pixel. This means that a new beam must be traced in every fragment to process its corresponding areas (figure 7.11b). Those fragments can be activated doing $G = 1$ and $R = 1$.

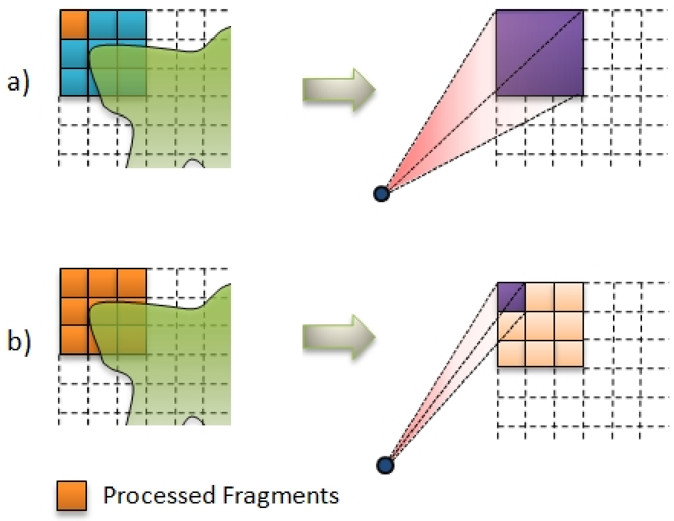

Figure 7.11: a) One fragment is used to evaluate the pixel. b) If the beam is crossing a border, then other fragments are activated to evaluate more sub-areas of the pixel.

The detection of borders is performed using the *meet* operation, as was explained in section 5.4.3. A border is found when for a beam that intersects the surface, $0 \notin meet$. The value of the *meet* operation can be saved in the blue (B) and alpha (A) channels: the minimum of the meet is saved in B and the maximum in A. Those channels were not used in the intersection test presented in last section, for that reason, they are available to store other values. The pseudocode for the anti aliasing algorithm is presented in figure 7.12.

Figure 7.13 represents a texture to be processed, in which the yellow spheres represent the fragments that are performing the intersection test. Those fragments will represent an area that correspond to nine rays (In this case, those fragments are in the left-up position of the beam). In the other fragments, a zero in the red channel disables further processing.

If a root is found and the inclusion test fails (the beam contains rays intersecting and other missing the surface), the fragments near to the main fragment are activated. Those fragments are used to calculate the intersection test for a proportional area of the pixel according to the number of fragments activated. If there are nine fragments available, the area is subdivided in nine parts. Also, the fragment that started the subdivision (the yellow one) is reactivated to calculate a part of the area of the pixel.

In figure 7.13, the green spheres represent the fragments activated after the first subdivision step. Other fragments are disabled (excepting the yellow fragment that correspond to the evaluated area) during this process, then the entire calculation power of the GPU is used only in them.

When the number of subdivisions set at the beginning of the process is reached, another fragment program is run. This program averages the colors of the pixels. In the case of fragments that covers a big area (the initial yellow fragments), the color is not averaged and used as final color for the corresponding pixel. In the case of fragments corresponding to borders of the surface, the color of all the fragments is averaged and put as final color for the corresponding pixel, as was shown in the right of the figure 7.13.

7.3.6 Results

The algorithm was tested on a DELL 670 Workstation, with a Xeon processor (3 GHz) and 3 Gigabytes of RAM Memory, using a GeForce 8800 GTX GPU. The resolution selected for the images was 1024 x 1024 (1048576 pixels). In this first test, the anti aliasing strategy is not used. OpenGL is used to create an initial rectangle that covers all the screen. This makes the GPU use the fragment shader over all the pixels in the screen.

Two surfaces that have thin parts are rendered: a Drop (figure 7.14a) and a Tri-thrumpet (figure 7.14c). Figure 7.14b shows a line connecting the drop with its body; it is not possible to obtain this result using a ray tracing algorithm based in point sampling. Also, the tri-thrumpet have thin connections which are correctly rendered (figure 7.14d).

The execution time of the GPU algorithm is also compared with a CPU ray tracing algorithm based on an interval bisection. In this case, the anti aliasing version on GPU was used, using nine rays per pixel. The size of the final images is 340x340. Figure 7.15

```
Anti aliasing Algorithm
    Reserve $n$ fragments per pixel
    Activate one Fragment
    Execute the Intersection test
    If the beam is intersecting a border
        Re-activate the current fragment and activate the other fragments
        Execute the Intersection test in every fragment
    endif
    Calculate the color for every active fragment
    Average the color for active fragments
```

Figure 7.12: Fragment program used to perform the intersection test.

shows four surfaces used to test the efficiency and exhibit the quality of this algorithm.

Times are given in table 7.16. Note that the time required to render these surfaces with the GPU is two orders of magnitude faster. The execution time measured corresponds to the time necessary to load the data and instruction, execute the program and get the final results in both CPU and GPU version.

Those results shows that a GPU implementation is much faster than a CPU implementation. However, the algorithm has many limitations compared with a CPU implementation, like a more elaborated anti aliasing strategy. Other characteristics like acceleration structures or illumination models are harder (but not impossible) to implement in GPU than CPU.

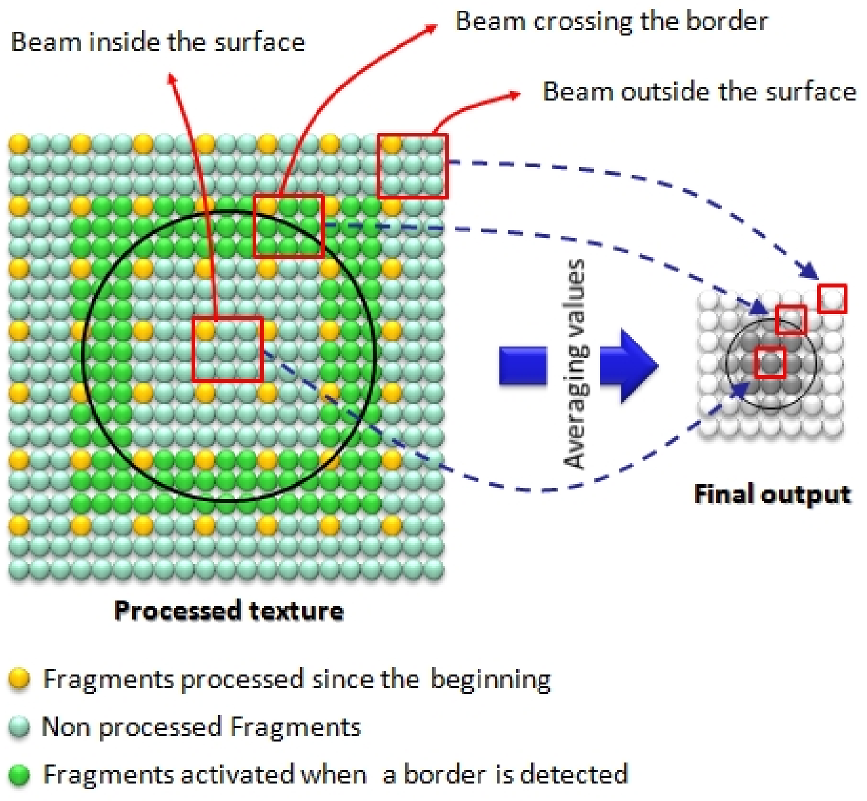

Figure 7.13: An anti aliasing strategy for ray tracing on GPU. Only a fragment representing a set of rays is used (in yellow). If that beam crosses a border of the surface, more fragments are used to calculate a more accurate color (in green). In the right, the final image is obtained averaging the color obtained in the processed fragments. During the process, many fragments are not processed saving computation effort of the GPU.

7.4 Conclusions

In this chapter the reliable algorithm for ray tracing implicit surfaces was improved by means of two new algorithms: a distributed version and an algorithm running in a GPU.

The results of the two implementations show that it is possible to improve the efficiency if there is special hardware available. It is remarkable that the improvement in efficiency using a GPU is between 2 and 3 orders of magnitude.

The coherency between neighboring rays in the ray tracing algorithm fits in the presented hardware implementation to improve the efficiency: in the distributed implementation is used to reduce the number of rays shared between workstations, and in the second case to reduce the number of fragments used in a anti aliasing algorithm.

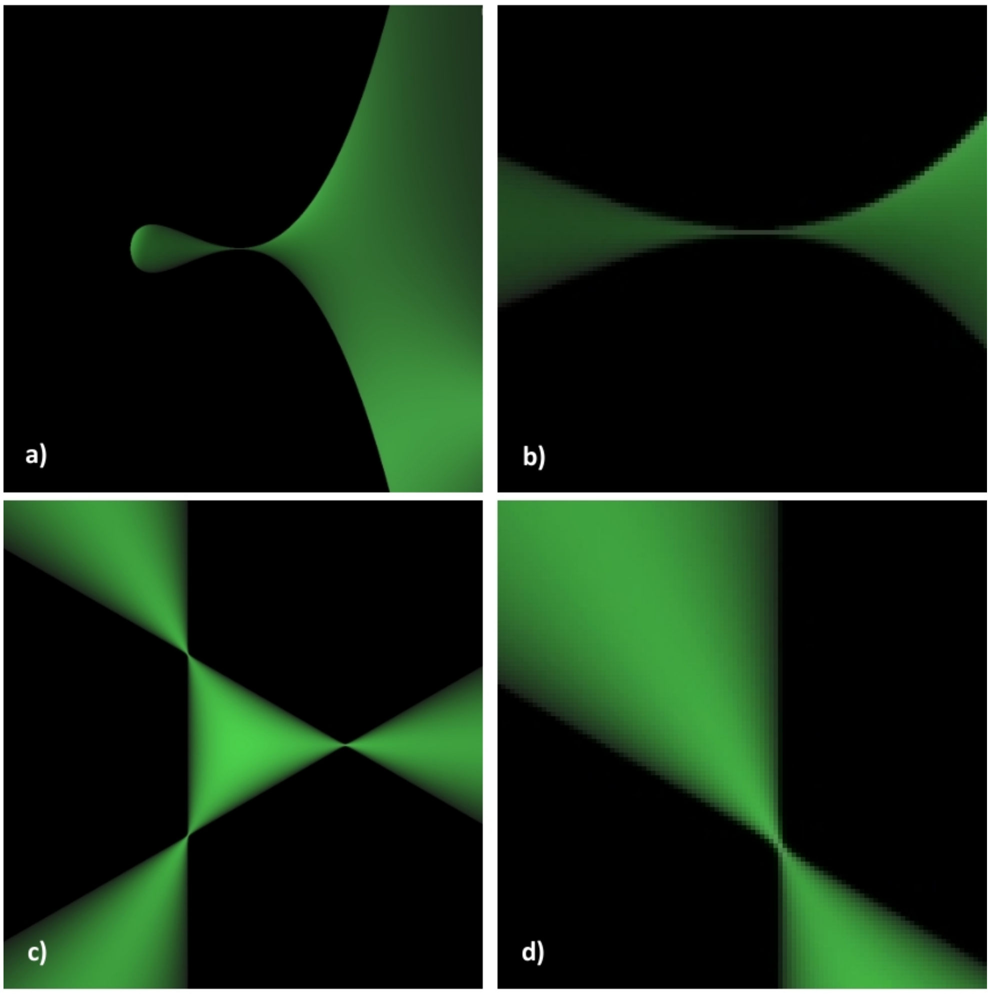

Figure 7.14: A Drop surface (a) showing in detail the thin section (b). Also a Tri-trumpet surface (c), and a detail (d).

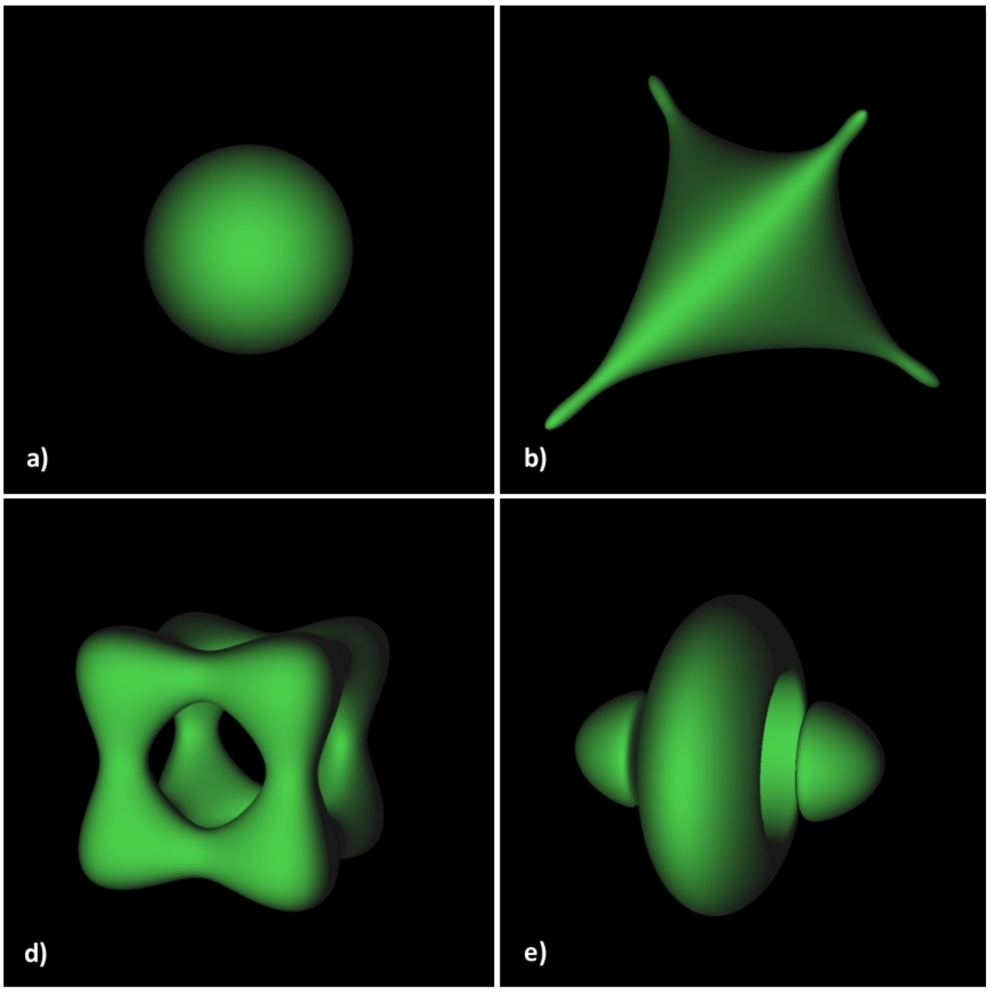

Figure 7.15: Some tested surfaces: a) Sphere, b)Kusner-Schmitt, c) Tangle and d) Gumdrop torus.

Figure 7.16: Comparison of CPU Times vs. GPU times for four surfaces (in seconds).

Surface	CPU	GPU
Sphere	300	2
Kusner-Schmitt	720	2
Tangle	900	3
Gumdrop Torus	1080	3

8 Conclusion

This thesis is concluded by summarizing the contributions presented in the previous chapters. Moreover, some directions for possible future works in the subject of this thesis are presented.

8.1 Contributions

This thesis has contributed to the state-of-the-art in the ray tracing of implicit surfaces in the following items:

- Ray tracing based on interval arithmetic has been extended to work with a set of rays simultaneously. This permits the exploitation of coherence in image space, reducing the number of intersection tests required in a traditional ray tracing algorithm. The result of this is the reduction of computational time required for the ray tracing of implicit surfaces based on interval arithmetic. This makes the technique applicable in algorithms in which both efficiency and reliability are required.
- An efficient method to trace shadow rays has been presented. This method is efficient and reliable, without the complexity of other techniques like beam tracing for polygonized surfaces. Moreover, it is possible to discover areas that are not under shadow. Computational time is saved if shadow rays are not traced for those areas.
- An anti-aliasing algorithm based on interval arithmetic theory has been presented. The thin details of the surfaces are detected because the beams scan the surface of the pixel completely. Problems related with point sampling disappear using this technique. Moreover, the adaptive algorithm introduced can be used to reduce computational effort in regions with low level of aliasing, obtaining better results than traditional anti aliasing algorithms based in point sampling.
- A method to exploit spatial coherence has been also presented. Although it was used for animation purposes, the same algorithm can be used to structure a static scene. The innovation of this technique is that it works with beams and it is completely based on interval arithmetic, which guarantees the reliability during all the process. The application of CGS operations in this method was also introduced.

- The spatial and time coherence was used to improve the rendering of scenes composed of implicit surfaces. A method to introduce those variables directly in the definition of the surfaces was presented. In this way, consecutive frames can be rendered using the same spatial structure and beam, which reduces the rendering time of the whole animation. The results show that spatial and time coherence can be efficiently exploited using interval arithmetic.
- A parallel implementation has been obtained for a cluster of computers. This permits further reduction of the time required for ray tracing based on interval arithmetic. Image space coherence can be used to reduce the communication between nodes. This occurs because areas that do not need to be ray traced are discarded before they are transmitted.
- An algorithm working on the GPU was developed. This algorithm was based on a GPU-based library where rounding was correctly implemented. The improvement obtained in time was two orders of magnitude, which makes the use of GPU a good alternative to improve efficiency for methods based on interval arithmetic.

It is possible to conclude that, although interval arithmetic increases the computational time of the algorithms, the same arithmetic can be used to exploit coherence to reduce these times. Also, quality of the visualization can be improved replacing point sampling by algorithms working for entire sets of values, instead of some samples.

This means that it is possible to keep the reliability obtained thanks to the use of this arithmetic when efficiency of the algorithm is also required, but with the benefit of a better visualization. This statement was demonstrated in the contents of this thesis.

8.2 Future work

Besides the contributions presented, there are some open improvements that should be undertaken. The first one is related to the exploration of alternatives based in GPU. Every year, GPUs are improving their performance at higher rates than CPUs. Also, new features are constantly included, allowing a flexible use of the GPU. As an example, the new models of GPU include a new programable unit to modify the geometry before the rasterization. There is also a new language called CUDA, developed by NVidia, that allows to easily exploit parallelism of the GPU. This language which is similar to C language, permits the use of the GPU as a co-processor, avoiding the need to fit the programs to the traditional graphics pipeline. These new features should be used to implement a reliable beam tracing of implicit surfaces running in GPU. There are other hardware alternatives like the new architecture of Intel called Larrabee, that could also be considered to compare the efficiency with an implementation in GPU.

Although there are several approaches to render implicit surfaces using graphics hardware, there are two issues that were not deeply studied. First, an efficient approach requires that most of the computations should be done in the GPU, because the transfer from main memory to GPU is slow. This includes the construction of acceleration structures, ray traversing and intersection test all in GPU. Also, it is desirable to combine both approaches presented in chapter 6. That is, we are planning to work with a cluster of computers in which all the processing related to ray tracing in every computer is done by the GPU. We expect that such distributed algorithm could improve the efficiency of our implementation up to three orders of magnitude.

Taking advantage of the efficiency obtained with the GPU, there is other improvement that can be undertaken: the trace of secondary beams. This thesis works with primary and shadow beams. Reflection beams were not taken into account. This problem can be complex, due to the different shapes obtained for the intersection between the primary beams and the implicit surface. We are planning to study different ways to apply Interval Arithmetic to exploit coherence for reflection beams, taking advantage of the efficiency obtained by means of the use of the GPU.

Bibliography

[1] Amitabh Agrawal and Aristides A. G. Requicha. A paradigm for the robust design of algorithms for geometric models. *Eurographics*, 1994.

[2] G. Alefeld. Eine modification des newtonverfarens zur bestimmung der reellen nullstellen einer reellen funktion. *Numeriche Matematik*, (50):32–33, 1970.

[3] J. Amanatides. Ray tracing with cones. *Computer Graphics*, 18(3):129–135, 1984.

[4] W. Barth an R. Lieger and M. Schindler. Ray tracing general parametric surfaces using interval arithemtic. *Visual Computer*, 10(7):363–371, 1994.

[5] Joaquim Armengol, Louise Travé-Massuyès, Josep Vehí, and Josep Lluís de la Rosa. A survey on interval model simulators and their properties related to fault detection. *Annual reviews in control*, 2000.

[6] James Arvo and David Kirk. Fast ray tracing by ray classification. *Computer Graphics*, 1(4):55–64, 1987.

[7] Didier Badouel, Kadi Bouatouch, and Thierry Priol. Distributed ray tracing and control for ray tracing in parallel. *IEEE Computer Graphics and Applications*, pages 69–77, 1994.

[8] S. Badt. Two algorithms for taking advantage of temporal coherence in ray tracing. *The Visual Computer*, (3):123–132, 1988.

[9] Ronald Balsys and Kevin Suffern. Visualisation of implicit surfaces. *Computer & Graphics*, 25:89–107, 2001.

[10] L. Barthe, V. Gaildrat, and R. Caubet. Combining implicit surfaces with soft blending in a csg tree. *CSG Conference Series, Ammerdown, U.K.*, pages 17–31, 1998.

[11] Vincent Bènèdet, Loic Lamarque, and Dominique Faudot. Volumetric reconstruction of unorganized set of points with implicit surfaces. *Computational Science and Its Applications*, 3480:838–846, 2005.

[12] Jim Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982.

[13] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 4(5):341–355, 1988.

[14] Jules Bloomenthal and Brian Wyvill. Interactive techniques for implicit modeling. *Symposium on Interactive 3D Graphics*, pages 109–116, 1990.

[15] Jules Bloomenthal and Brian Wyvill, editors. *Introduction to Implicit Surfaces.* Morgan Kaufmann Publishers, 1997.

[16] R. Bracewell, editor. *The Fourier Transform and its Applications.* McGraw-Hill, New York, 1978.

[17] Katja Buhler. Implicit linear interval estimations. *International Conference on Computer Graphics and Interactive Techniques*, 1998.

[18] O. Capriani, L. Hvidegaard, M. Mortensen, and T. Schneider. Robust and efficient ray intersection of implicit surfaces. *Reliable Computing*, 1(6):9–21, 2000.

[19] Nathan Carr, Jared Hoberock, Keenan Crane, and John Hart. Fast gpu ray tracing of dynamic meshes using geometry images.

[20] Paulo Cezar Carvalho, Luiz Henrique de Figueiredo, and Paulo Roma Cavalcanti. Computing arrangements of implicit surfaces. *CGC Workshop on Computational Geometry*, 1998.

[21] J. Chapman, T. Calvert, and J. Dill. Exploiting temporal coherence in ray tracing. *Proceedings of Gaphics Interface*, 1991.

[22] S. Chen and L. Williams. View interpolation for image synthesis. *ACM Computer Graphics*, 24(4):279–288, 1993.

[23] Martin Christen. *Ray Tracing on GPU.* PhD thesis, 2005.

[24] John Cleary, Brian Wyvill, Graham Birtwistle, and Reddy Vatti. A parallel ray tracing computer. *Proc. XI Association of Simula Users Conference. Paris*, pages 77–80, 1983.

[25] John Cleary and Geoff Wyvill. Analysis of an algorithm for fast ray tracing using uniform space subdivision. *The Visual Computer*, 4(2), 1988.

[26] S. Collange, M. Daumas, and D. Defour. Graphic processors to speed-up simulations for the design of high performance solar receptors. *IEEE 18th International Conference on Application-specific Systems, Applications and Processors*, 2007.

[27] A. de Cusatis, L. de Figueredo, and M. Gatas. Interval methods for ray casting implicit surfaces with affine arithmetic. *Proceedings of SIBGRAPH*, 1999.

[28] Erwing de Groot and Brian Wyvill. Rayskip: faster ray tracing of implicit surface animations. *Computer graphics and interactive techniques in Australasia and South East Asia*, pages 31–36, 2005.

[29] Daniel Dekkers, Kees van Overveld, and Rob Goldsteijn. Combining csg modeling with soft blending using lipschitz-based implicit surfaces. *The Visual Computer*, pages 281–391, 2004.

[30] Tom Duff. Interval arithmetic and recursive subdivision for implicit functions and constructive solid geometry. *Computer Graphics*, 26(2):131–138. (SIGGRAPH'92).

[31] Dan Fay, Ali Sazegari, and Dan Connors. A detailed study of the numerical accuracy of gpu-implemented math function. *Supercomputing 2006, General-purpose GPU computing*, 2006.

[32] S. Foufou, J. Brun, and A. Bouras. Surface/surface intersections: a three states classification. *International Conference on Knowledge transfer visualization and graphics 96*, pages 499–506, 1996.

[33] Akira Fujimoto, Takayuki Tanaka, and Kansei Iwata. Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 6(4):16–26, 1986.

[34] Oscar García, Josep Vehí, Jose Campos e Matos, Antonio Henriques, and Joan Casas. Structural assessment under uncertain parameters via interval analysis. *Journal of Computational and Applied Mathematics*, 218(1):43–52, 2007.

[35] E. Gardenes, M. Sainz, L. Jorba, R. Calm, R. Estela, H. Mielgo, and A. Trepat. Modal interval. *Reliable Computing*, 7(2):77–111, 2001.

[36] Marcel Gavriliu. *Towards more efficient interval analysis: corner forms and remainder interval newton methods.* PhD thesis, 2005.

[37] J. Genetti and D. Gordon. Ray tracing with adaptive supersampling in object space. *Graphics Interface*, pages 70–77, 1993.

[38] Christopher Giertsen and Johnny petersen. Parallel volume rendering on a network of workstations. *Computer graphics and applications*, 13(6):16–23, 1993.

[39] A. Glassner. Space time ray tracing for animation. *IEEE Computer Graphics and Applications*, pages 60–70, 1988.

[40] Andrew Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 4(10):15–22, 1984.

[41] Andrew Glassner, editor. *An Introduction to Ray Tracing.* Academic Press Inc., 1993.

[42] Dominik Goddeke. Gpgpu - basic math tutorial. *http://www.mathematik.uni-dortmund.de/ goeddeke/gpgpu/tutorial.html.*

[43] J. Haines and D. Greenberg. The light buffer: a shadow testing accelerator. *IEEE Computer Graphics and Applications*, 6(9):6–16, 1986.

[44] E. Hansen. Globally convergent interval method for computing and bounding real roots. *BIT*, (18):415–424, 1978.

[45] E. Hansen. *Geometric computations with interval and new robust methods.* Horwood publishing limited, 2003.

[46] Bhanu Hariharan and Srinivas Aluri. Efficient parallel algorithms and software for compressed octress with applications to hierarchical methods. *Parallel computing*, 31(3):311–331, 2005.

[47] John Hart. Ray tracing implicit surfaces. *SIGGRAPH 1993 Course Notes*, 1993.

[48] John Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1997.

[49] V. Havran, C. Damez, K. Myszkowski, and H. Seidel. An efficient spatio-temporal architecture for animation rendering. *Proceedings of Eurographics Symposium on Rendering*, 2003.

[50] P. Heckbert and P. Hanrahan. Beam tracing polygonal objects. *Computer Graphics*, 18(3):119–127, 1984.

[51] Luiz de Figueredo Helio Lopes, João Batista. Robust adaptive approximation of implicit surfaces. *Proceedings of SIBGRAPI 2001*, pages 10–17, 2001.

[52] Brian Von Herzen and Alan H. Barr. Accurate triangulations of deformed intersecting surfaces. *Computer Graphics (Proc. SIGGRAPH 87)*, 21(4):103–110, 1987.

[53] M. Hu and J. Foley. Parallel processing approaches to hidden-surface removal in image space. *Computer and graphics*, 9(3):303–317, 1985.

[54] Ernst H. Huber. Intersecting general parametric surfaces using bounding volumes. *10th Canadian Conference on Computational Geometry*, 1998.

[55] David Jevans and Brian Wyvill. Adaptive voxel subdivision for ray tracing. *Proc. Graphics Interface 1989*, pages 164–172, 1989.

[56] W. Kahan. A logarithm too clever by half. *http://http.cs.berkeley.edu/ wkahan/LOG10HAF.TXT*, 2004.

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

[58] M.J. Keates and R.J. Hubbold. Interactive ray tracing on a virtual shared-memory parallel computer. *Computer Graphics Forum*, 14(4):189–202, 1995.

[59] Aaron Knoll, Younis Hijazi, Charles Hansen, Ingo Wald, and Hans Hagen. Interactive ray tracing of arbitrary implicit functions. *Scientific computing and imaging institute, Technical report UUSCI-2007-002*, 2007.

[60] Kwan liu Ma, James Painter, Charles Hansen, and Michael Krogh. Parallel volume rendering using binary-swap compositing. *Computer graphics and applications*, 14(4):59–68, 1994.

[61] Charles Loop and Jim Blinn. Resolution independent curve rendering using programmable graphics hardware. pages 1000–1009, 2005.

[62] Pascual González López. *Coherencia de Objetos, coherencia de Rayos y Paralelismo, en la aceleración del trazado de rayos.* PhD thesis, 1999.

[63] D. Marini, A. Canesi, C. Gatti, and M. Rossi. Parallelising accelerated ray tracing for high quality image synthesis. *Transputer Applications and Systems*, 20:40–53, 1994.

[64] Ralph Martin, Irina Voiculescu Huahao Shou, and Guojim Wang Adrian Bowyer. Comparison of interval methods for plotting algebraic curves. *Computer Aided Geometric Design*, 2002.

[65] Don Mitchell. Robust ray intersection with interval arithmetic. *Proceedings on Graphics interface '90*, pages 68–74, 1990.

[66] R. E. Moore. Methods and applications of interval analysis. *Studies inApplied Mathematics (SIAM)*, 1979.

[67] Ramon Moore. *Interval Analysis.* Englewood Cliffs, New Jersey, 1966.

[68] Ulrich Neumann. Communication cost for parallel volume-rendering algorihtms. *Technical report: TR93-038*, 1994.

[69] M. Ohta and M. Maekawa. Ray coherence theorem and constant time ray tracing algorithm. *Computer Graphics (proceedings of CG International 87)*, pages 303–314.

[70] Joao Batista Oliveira and Luiz Henrique de Figueiredo. Robust approximation of offsets, bisectors and medial axes of plane curves. *Reliable Computing*, 2002.

[71] A. Paiva, H. Lopez, T. Lewiner, and L.H. de Figueredo. Robust adaptive meshes for implicit surfaces. *Computer Graphics and Image Processing, SIBGRAPH*, 2006.

[72] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 2(8):429–446, 1995.

[73] J. Peiró, L. Formaggia, M. Gazzola, A. Radaelli, and V. Rigamonti. Shape reconstruction from medical images and quality mesh generation via implicit surfaces. *International Journal for Numerical Methods in Fluids*, 53(8):1339–1360, 2006.

[74] Paul Pitot. The voxar project. *Computer graphics and applications*, 13(1):27–33, 1993.

[75] H. Ratschek and Jon Rokne. *Geometric computations with interval and new robust methods.* Horwood publishing limited, 2003.

[76] A. Ricci. Constructive geometry for computer graphics. *The computer Journal*, 16(2):157–160, 1973.

[77] S. Rubin and T Whitted. A three-dimensional representation for fast rendering of complex scenes. *Computer Graphics*, 14(3):110–116, 1980.

[78] J.F. Sanjuan-Estrada, L.G. Casado, and I. García. Reliable algorithms for ray intersection in computer graphics based on interval arithmetic. *XVI Brazilian Symposium on Computer Graphics and Image Processing*, pages 35–44, 2003.

[79] Vladimir Savchenko, Alexander Pasko, Oleg Okunev, and Tosiyasu Kunni. Function representation of solids reconstructed from scattered surface points and contours. *Computer Graphics Forum*, 14(4):181–188, 1995.

[80] Ryan Schmidt, Brian Wyvill, Mario Costa-Sousa, and Joaquim Jorge. Shapeshop: Sketch-based solid modeling with the blobtree. In *Proc. 2nd Eurographics Workshop on Sketch-based Interfaces and Modeling*, pages 53–62. Eurographics, Eurographics, 2005. Dublin, Ireland, August 2005.

[81] Ryan Schmidt, Brian Wyvill, and Eric Galin. Interactive Implicit Modeling with Hierarchical Spatial Caching. In *Proc. Shape Modelling International, MIT, USA*. IEEE Press, May 2005.

[82] Thomas Sederberg and Rida Farouki. Approximation by interval bezier curves. *IEEE Computer Graphics and Applications*, 12(5):87–95, 2006.

[83] M. Shinya, T. Takahashi, and S. Naito. Principles and applications of pencil tracing. *Computer Graphics*, 21(4):45–54, 1987.

[84] Peter Shirley. *Fundamentals of computer graphics.* A K Peters Ltd., 2002.

[85] Peter Shirley and R. Keith Morley. *An Introduction to Ray Tracing.* A K Peters, Ltd., 2003.

[86] SIGLA/X. Modal intervals. basic tutorial.

[87] John Snyder, Adam Woodbury, Kurt Fleischer, Bena Currin, and Alan Barr. Interval methods for multi-point collisions between time-dependent curved surfaces. *Proceedings of the 20th annual conference on Computer Graphics and interactive techniques*, pages 321–334, 1994.

[88] L. Speer, T. DeRose, and B. Barsky. A theoretical and empirical analysis of coherent ray tracing. *Computer-Generated images*, pages 11–25, 1986.

[89] Barton Stander and John Hart. Guaranteeing the topology of an implicit surface polygonization for interactive modelling. *SIGGRAPH 97*, pages 279–286, 1997.

[90] J. Stolfi and L.H. de Figueiredo. Self-validated numerical methods and applications. *Monograph for 21st Braziliand mathematics colloquium, IMPA, Rio de Janeiro*, 1997.

[91] Nilo Stolte and René Caubet. Fast high definition descrete ray tracing implicit surfaces. *Discrete Geometry for Computer Imaginery*, pages 61–70, 1995.

[92] Nilo Stolte and Arie Kauffman. Robust hierarchical voxel models for representation and interactive visualization of implicit surfaces in spherical coordinates. *Implicit surfaces 98*, pages 81–88, 1998.

[93] Nilo Stolte and Arie Kaufman. Discrete implicit surfaces models using interval arithmetics. *Second CGC Workshop on computational geometry*, 1997.

[94] Kevin Suffern and Ronald Balsys. Rendering the intersection of implicit surfaces. *IEEE Computer Graphics and Aplications*, 23(5):70–77, 2003.

[95] I. Sutherland, R. Sproull, and R. Schumacker. A characterization of ten hidden-surface algorithms. *Computing Surveys*, 6(1):1–55, 1974.

[96] G. Taubin. Rasterizing algebraic curves and surfaces. *IEEE Computer graphics and applications*, 14(2):14–23, 1994.

[97] D. Toth. On ray tracing parametric surfaces. *Conference proceedings SIGGRAPH 85*, 19(3):171–179, 1985.

[98] Greg Turk and James O'Brien. Modelling with implicit surfaces that interpolate. *ACM Transactions on Graphics*, 21:855–873, 2002.

[99] Josep Vehí, Inés Ferrer, and Miguel Sainz. A survey on applications of interval analysis to robust control. *IFAC World Congress*, 2000.

[100] Alan Watt. *3D computer graphics*. Addison-Wesley, second edition, 1993.

[101] Turner Whitted. An improved illumination model for shadow display. *Communications of ACM*, 23(6):343–349, 1980.

[102] Andrew Wood, Brendan McCane, and Scott King. Ray tracing arbitraty objects on the gpu. *Proceedings of Image and Vision Computing*, pages 21–23, 2004.

[103] B. Wyvill, A. Guy, and E. Galin. Extending the csg tree. warping, blending and boolean operations in an implicit surface modeling system. *Computer Graphics Forum*, 18(2):149–158, 1999.

[104] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.