IMPROVING RAY-TRACING USING INTERVAL ANALYSIS AND ARITHMETIC

by

Mohammed Affan Abdelrahim Zidan

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
June 2010

IMPROVING RAY-TRACING USING INTERVAL ANALYSIS AND ARITHMETIC

by

Mohammed Affan Abdelrahim Zidan

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS

Under the Supervision of

M. Fathy Abu El-Yazeed — **Professor**, **Elec. and Com. Dept.**

Hossam A. H. Fahmy — **Associate Professor**, **Elec. and Com. Dept.**

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
June 2010

IMPROVING RAY-TRACING USING INTERVAL ANALYSIS AND ARITHMETIC

by

Mohammed Affan Abdelrahim Zidan

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
ELECTRONICS AND COMMUNICATIONS

Approved by the
Examining Committee

Prof. Dr. El-Sayed Mostafa Saad

Prof. Dr. Mohsen Abd El-Razek Rashwan

Prof. Dr. Mohamed Fathy Abu El-Yazeed

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
June 2010

Acknowledgment

All praise is to ALLAH Lord of the worlds, for everything.

It is an honor for me to thank my supervisors Prof. Mohamed Fathy Abu El-Yazeed and Ass. Prof. Hossam Aly Fahmy for their encouragement and valuable guidance. This thesis would not have been possible without the valuable help of my supervisors.

Nothing could be achieved without the support and the help of my family, my father, mother, brother and sister.

I am indebted to many of my colleagues who supported me.

Abstract

Ray-tracing is a promising emerging technology for real-time rendering of computer graphics, computer aided graphical design (CAGD), and physical simulations. Although ray-tracing is a common off-line technique, its very time consuming arithmetic operations block the way of its real-time implementation.

Interval arithmetic and analysis provide a very powerful extension for the regular number systems which could pave the way for a real-time ray-tracing system.

Based on the interval analysis and arithmetic techniques we introduced improvements for the ray-tracing system on the algorithmic, architecture and implementation layers.

The core of ray-tracing, the ray intersection process, is believed to consume 95% of the ray-tracing time. Therefore, speeding up such process is the shortest way for the real-time implementation of a ray-tracing system. We introduce an interval rejection test used for speeding up the ray-tracing process, by fast rejection of objects (triangles) not containing a valid intersection.

According to the experiments performed on our C test platform, without any particular hardware support, the rejection test correctly rejects more than 99.9% of the total number of triangles and speeds up the ray intersection process up to 2.21 times. While the rejection test expresses a speed-up on the software layer, its main advantage is its very small area hardware realization.

Based on the collected statistical data, heterogeneous ray-tracing multi-cores architecture has been proposed. This architecture is a computing unit made from an array of heterogeneous multiprocessors. Each of these multiprocessors consists of a high density of interval rejector units. These cores are written in Verilog and simulated for the Altera FPGAs.

To the best of our knowledge, the rejector core unit is the first combination between ray-tracing triangle meshes and interval analysis and arithmetic on the hardware layer. The implementation of the interval rejector shows a reduction more than 22 times in area compared to the conventional intersection process hardware realization.

Contents

List of Tables

List of Figures

List of Abbreviations

$\mathbf{R}$ The set of real numbers

$\mathbb{R}$ The set of computer represented numbers

$\triangle$ Rounding towards infinity

$\triangledown$ Rounding towards negative infinity

out Outside rounding

in Inside rounding

range Range of the function output

$\exists$ There exist

$\forall$ For all

s Source of light ray

d Direction of light ray

Chapter 1

Introduction

Ray-tracing is a common technique for producing very high quality images and video frames, since it simulates the natural by simulating the real light reflection. Also it has very promising properties in physical simulations and computer aided graphical design (CAGD) [1]. Ray-tracing is heavily used for off-line rendering since it does not have any real-time implementation yet [2].

The ray-tracing rendering is based on the simple idea of light reflections. A ray (or more) is traced from the camera (the eye) to each pixel of the rendered image, which is visually placed in front of the camera. Each ray will reflects and refracts several times on the scene's objects, for calculating the final value of the pixels [3].

Compared to the common on-line rendering mode, rasterization (Appendix B), ray tracing produces much more realistic images, due to its realistic reflections, refractions and shadows calculations. Moreover, ray-tracing is a global lighting rendering technique compared to local lighting rasterization, in which light sources affect all members of the scene globally.

On one hand, for the scenes of common complexity, ray-tracing is much slower than rasterization. The main drawback in ray-tracing is its very time consuming intersection process between traced ray and the scene's objects. Calculation of such intersection requires many floating point operations. The intersection process can consume up to 95% of the whole ray-tracing time [4]. The reduction of the intersection process time should pave the way for real-time ray-tracing systems. A real-time system implementation will improve the CAGD response, the physical simulations and the real-time computer graphics.

On the other hand, for very complex scenes, ray-tracing is faster than rasterization. In [1] a complete CAD model of the Boeing 777 airplane of 12 GByte of size is ray-traced in a much smaller time compared to rasterization. Hence, as computer graphics grow more complex the need for ray-tracing increases. In fact, researchers at Intel think that running ray tracing on multi-cores processors will

kill rasterization [5].

This thesis combines ray-tracing with another field of science. Interval analysis and arithmetic are believed to be much more powerful and reliable compared to the point-wise numbers and techniques. Interval arithmetic provides more reliable results than that of the regular floating-point arithmetic [6, 7]. Also interval analysis techniques are more powerful than the point-wise similar algorithms, since they provide more accurate solutions, with a guarantee of convergence [7, 8]. Interval analysis can be also used for speeding-up calculation by using it in an intermediate calculation stage.

In general intervals are continuous sets of numbers represented by their bounding values. The point-numbers can be considered as a special case in which the two bounds have the same value.

There is a wide range of applications for which interval analysis and arithmetic can be used. These applications are split into four main categories, bounding roundoff error, representing physical quantities with a range of uncertainty, interval based analysis algorithms, and molding logically interval applications.

Although that interval arithmetic is not supported in hardware by the majority of the current commercial processors, it is expected that it may be completely supported after its standardization. According to Hayes [9] floating point arithmetic become a reality only after IEEE publishes its floating point standard. Currently, IEEE is working on the interval arithmetic standard, IEEE P1788 [10]. Even the programing languages will support the interval operations on the software layer in the future [11].

According to [6] interval arithmetic can be as fast as floating-point arithmetic with a little extra hardware. Also it shows that the interval arithmetic hardware is immune against the regular floating-point problems such as underflow, overflow, division by zero exceptions. The interval units hardware may support floating-point operations in case of no interval operations are scheduled to the processor.

In general interval analysis and arithmetic are considered more reliable than regular floating-points, provide more powerful analysis techniques, and can be used for speeding-up of data processing. But since intervals representations on computers are based on the floating-point numbers, so it is considered as an extension to floating-points not a complete replacement.

1.1 Previous Work

Many works were introduced for improving ray-tracing and towards real-time systems. These works are split into two directions, speeding-up the tracing process, and improving the accuracy of the output.

Many algorithms are introduced for ray-tracing. While the non-interval based algorithms are common as [12–20], some interval techniques have been introduced for improving ray-tracing. The majority of these techniques provide more accurate solution for the intersection problem as those found in [12, 21–25]. Other algorithms are introduced for speeding up the tracing process as [21, 26].

Some implementations in the hardware layer are introduced trying to achieve a real-time ray-tracing, but none of them achieve this goal. In [27] a rendering architecture named SaarCOR is proposed for ray-tracing triangle-mesh surfaces. In [28] a programmable hardware called RPU (Ray-tracing Programmable Unit) is built over FPGA.

Other implementations use the capabilities of the modern CPU architecture, like SIMD (Single Instruction Multiple Data) vector processing [29, 30]. Also GPUs (Graphical Processing Units), which were originally designed for rasterization, are used for ray-tracing, like in [31].

Many programs are built on the software layer for supporting ray-tracing. OpenRT, in context with OpenGL, is introduced in [2] as an API supporting real-time ray-tracing. In another direction many off-line rendering softwares were produced, like [32–34].

1.2 Achievements

By combining interval analysis and arithmetic techniques with the ray intersection process we introduce optimizations for the ray-tracing on the algorithmic, the architecture and the implementation layers.

We introduce an interval method for speeding-up the ray-triangle intersection process, in ray-tracing triangle-mesh surfaces, by reducing the running data set tested for intersection. This method is a rejection test used for the fast rejection of triangles not containing a valid intersection. The few triangles that are not rejected by our method are then tested using any of the conventional tests. A software test platform is built to validate the algorithm and collect statistical data.

According to the experimental results on our software test platform, more than 99.9% of the triangles are rejected using the interval test. Also on the software layer implementation, without any level of parallelism applied, we gain a speedup between 1.25 and 2.21 after applying the interval rejection test, depending on the 3D model tested. The experimental results also show that the rejection test removes a jitter like phenomena in the execution time of the intersection process, which makes it very suitable for real-time video rendering.

Although the rejection test expresses a speed-up on the software layer, its main

advantage is its very small area when implemented in hardware compared to the hardware realization of any of the conventional intersection tests. Based on the collected statistical data, we introduce new ray-tracing multi-cores heterogeneous architecture, towards a real-time ray-tracing system. The proposed architecture is a computing unit made of an array of heterogeneous multiprocessors. Each unit consists of a high density of our newly introduced interval rejector units. Also each multiprocessor contains a smaller number of intersectors and one unit for final pixel calculations. An area optimized interval rejector is also provided. This rejector requires much smaller on-chip area compared to the conventional intersection algorithms realizations.

To the best of our knowledge, our rejector core unit is the first combination between ray-tracing triangle meshes and interval analysis and arithmetic on the hardware layer. The implementation of the interval rejector shows a reduction more than 22 times in area compared to the conventional intersection process hardware realization. Such reduction in area allows a higher density of rejector units, which proportionally leads to higher speeds.

1.3 Thesis Organization

Chapter 2: This chapter introduces the basics of the interval theory. The main arithmetic operations, properties and functions are given. The strong and weak points of the theory are discussed. The applications and the applications categories in which intervals can be used are also introduced. Finally, the widely used interval theory extension, the modal intervals, is discussed in the last section of this chapter.

Chapter 3: This chapter is a survey on the ray-tracing techniques. The chapter starts with mentioning the previous work in this field followed by explaining the ray-tracing rendering mechanism. After that, the widely used tracing algorithms are introduced for each type of the three object representation used in computer graphics and physical simulations. Each of the algorithms is discussed explicitly showing its strong and weak points. The usage of the intervals within ray-tracing is also highlighted. Finally, the dedicated hardware implementations are mentioned.

Chapter 4: This chapter introduces our interval rejection algorithm, used for improving the intersection process, and shows its detailed steps. Also the proofs that there is no special treatment required for some special cases are also given.

Chapter 5: Our new multi-cores heterogeneous ray-tracing architecture is introduced in this chapter. The architecture's organization and its building blocks are introduced. The observations, which lead to this architecture, are discussed. Also the block diagrams of core unit, the interval rejector unit, are explained showing some of the optimizations implemented within its design.

Chapter 6: The experimental results, the hardware realization, and the test methodology are given and discussed within this chapter. The chapter starts by stating the test methodology used within the whole process. After that, the results of the software test platform are discussed. Within this section detailed tables and graphs are given, based on a test set containing millions of triangles. These results lead to the introduction of the ray-tracing architecture. In the last section the hardware realization results are given, including part of the test-bench test set. An area and delay comparison between the interval rejector realization and our implementation of one of the famous intersection algorithms is also given.

Chapter 7: The conclusions and the future works are discussed within this chapter.

Appendix A: In this appendix the interval extension for the Newton method is introduced.

Appendix B: The rendering steps of the currently dominating real-time technique, rasterization, are given within this appendix.

Appendix C: The CUDA general propose graphical processing units (GPGPU) architecture is introduced is this appendix. The building blocks and the memory organization of the CUDA architecture are also discussed.

Appendix D: Part of the code of the software test platform is written in this appendix.

Chapter 2

Interval Arithmetic and Analysis

2.1 Introduction

Interval analysis and arithmetic are believed to be much more powerful and reliable compared to point-wise numbers and techniques. Interval arithmetic provides more reliable results than that of the regular floating point arithmetic [6, 7]. Furthermore, interval analysis is more powerful than the point-wise similar algorithms. It provides more accurate solutions [6–9, 12, 21], with a guarantee of convergence [7, 8]. Also interval analysis can be used for speeding-up calculation by using it in an intermediate calculation stage.

While the first mention of intervals was in a Cambridge University Ph.D. [9], intervals in its current shape was first introduced by T. Sunaga in 1958 [35] and R. E. Moore in 1966 [36]. Since then, many useful interval based applications and techniques have been introduced. Intervals already find applications in error analysis, error bounding [37], solving systems of equations [22, 38, 39], global optimization [40], control [39, 41], computer graphics[1] [12, 21–25], signal processing [41], mathematical proofs [42], and many other applications [43]. Even an interval extension for the current C++ standard has been proposed [11].

Although that interval arithmetic is not supported in hardware by the majority of the current commercial processors, it is expected that it maybe completely supported after its standardization. According to [9] floating point arithmetic become a reality after IEEE published its floating point standard. Currently, IEEE is working on the interval arithmetic standard, IEEE P1788 [10]. So, most of the interval algorithms which improve the accuracy and/or introduce a speedup using the software layer should witness a considerable breakthrough in performance with the hardware support of interval arithmetic.

Many works in the hardware implementation for the interval arithmetic are

[1] The usage of intervals in computer graphics will be highlighted in Chapter 3.

already introduced [6, 8, 44]. Interval arithmetic can be as fast as floating-point arithmetic with a little extra hardware [6]. Moreover, interval arithmetic is immune against the regular floating-point problems like underflow, overflow, and division by zero. The interval units also can be built to support floating-point operations in case no interval operations are scheduled to the processor.

So, in general interval analysis and arithmetic are considered more reliable than regular floating-points, provide more powerful analysis techniques, and can be used for speeding-up of data processing. But Intervals representations on computers are based on the floating-point numbers, so it can be considered as a very powerful extension to floating-points not a complete replacement. Intervals have some drawbacks like overestimation problem, which can limit their usage is some application.

Many extensions for the main interval theory are introduced to improve some of the weak points of the classical theory. Two of the most famous extensions are the Kaucher [45] and the Modal [46, 47] intervals (Section 2.5). Although these extensions improve the quality of the output, they add much more complexity to the interval theory, and hence its hardware realization. The current draft of the IEEE interval arithmetic standard supports only the classical interval theory [10].

Definition

The interval set of numbers can be defined as,

$$I(\mathbf{R}) = \{[a,b] \,|a \in \mathbf{R}, b \in \mathbf{R}, a \leq b\} \tag{2.1}$$

where $\mathbf{R}$ is the set of real numbers. But due to the fact that computers represent only a small set of the real numbers, the computer representable set of intervals may be defined as,

$$I(\mathbb{R}) = \{[a,b] \,|a \in \mathbb{R}, b \in \mathbb{R}, a \leq b\} \tag{2.2}$$

where $\mathbb{R} \subset \mathbf{R}$ is the representable subset of real numbers on the computer system. According to [10], $I(\mathbb{R})$ can be defined also using $\mathbb{R}^*$, where $\mathbb{R}^* = \mathbb{R} \cup \{-\infty, +\infty\}$.

The following sections of this chapter are organized such that the main applications of the intervals are discussed in section 2.2. The interval arithmetic operations and main properties are provided in section 2.3. Section 2.4 is concerned about the interval functions. The main interval theory extension, the modal intervals, are highlighted in the last section 2.5. In the rest of the thesis, the upper-case letters are reserved for interval numbers.

2.2 Applications Categories

There is a wide range of applications using interval analysis and arithmetic. These applications are split into four main categories, namely, bounding roundoff error, representing physical quantities with a range of uncertainty, interval based analysis algorithms, and modeling logically interval applications. Moreover, the expected, near future, hardware support of the interval arithmetic will increase the probability of selecting interval as a choice. So the range of applications will increase, and intervals may be used in categories beyond the four mentioned below.

2.2.1 Bounding Roundoff Error

One of the main sources of errors in computer calculated quantities is due to the fact that numbers are not accurately represented on the computer systems. The inaccurate representation is due to the finitude in numbers within digital systems; the floating-point numbers. Rounding errors in floating-point arithmetic have been charged for many deadly accidents. The failure of American intelligent Patriot missiles in facing the Iraqi Scud missiles in 1991 is believed to be due to the floating point rounding errors [9]. Also the fall of the Ariane 5 satellite carrying rocket in 1996 is proved to be due to the same reason.

One of the fields of intervals is as a reliable replacement for numbers representation on computer systems. Hence any number will be represented by its two neighbor floating-point numbers, as shown in Fig. (2.1), instead of rounding. So, the final result is guaranteed to be in a given interval. And due to the very tiny interval width used, the overestimation effect is minimal.

Some quantities like $\sqrt{2}$ or π needs infinite precision for correct representation. So, interval representation in these cases is a very valuable selection, and leads to more accurate bounded outputs.

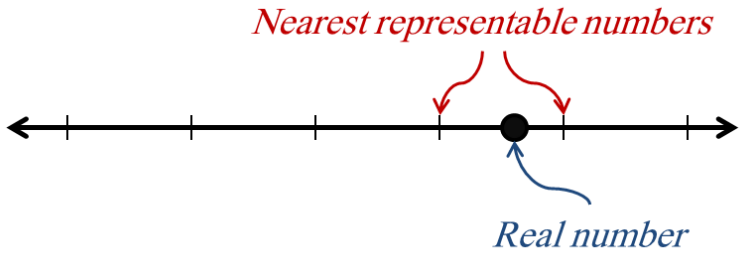


Figure 2.1: A real number can fall between two computer representable numbers.

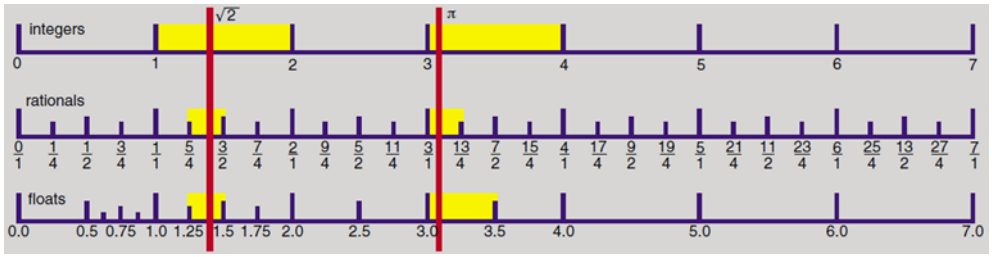


Figure 2.2: π and $\sqrt{2}$ can't be represented using any finite precision number system[9].

2.2.2 Representing Physical Quantities with a Range of Uncertainty

Almost every measured quantity has a range of uncertainty, and the usage of intervals leads to much better representation of these quantities, and so the results depending on them. More accurate bounded outputs leads to better and error free designs. As shown in the example below representing resistance and voltage source as an interval enclosing the range of uncertainty leads to a better knowledge about the current.

Example 1. For a simple circuit with a single loop of resistance $R = 100k\Omega \pm 5\%$ and voltage source $V_s = 5V \pm 2\%$. R and V_s can be rewritten as,

$$R = [95, 105]\, k\Omega$$

$$V_s = [4.9, 5.1]\, V$$

and using the interval arithmetic, the current is enclosed within a range such that,

$$I = \frac{V_s}{R} = \frac{[4.9, 5.1]}{[95, 105]} = [46.67, 53.68]\, \mu A$$

2.2.3 Interval Based Analysis Algorithms

Many interval based analysis algorithms are proved to be more powerful that their point-wise equivalents. Normally, interval analysis produces point numbers and only used as an intermediate calculation stages. Interval analysis methods are usually used for producing accurate results. However, some interval methods have been created for speeding up the calculation process. Currently, interval theory is mainly used in analysis more than being an extension for the floating-point arithmetic.

Many of the famous analysis algorithms have a more powerful interval extension, like interval Gauss–Seidel [7], interval bisection, interval Newton method [6, 7]

(for more about interval Newton method please refer to Appendix A). However, there are a lot of independently created interval analysis methods, like Mitchell Algorithm [12] (Sub-section 3.3.1) or Alefeld-Hansen method [22, 24] (Sub-section 3.3.3).

2.2.4 Modeling Logically Interval Applications

There are many real world problems which are easier to be modeled in computer using intervals. Software and hardware support for intervals will improve the processing of these problems significantly.

2.3 Interval Operations and Properties

Although the intervals are simply defined, their underlying arithmetic rules are not the same. In the following subsections the main arithmetic operations and properties of the interval arithmetic are defined.

2.3.1 Arithmetic Operations

In general the interval arithmetic operations can be defined as,

$$A \circ B = [a \circ b | a \in A,\, b \in B]\,,\ for \circ \in \{+, -, \cdot, /\}\,. \tag{2.3}$$

where $A, B \in I(\mathbb{R})$. A special treatment is required in case of the division operation if $0 \in B$.

Each arithmetic operation can be defined explicitly, more tightly. Let ,

$$\begin{aligned} A &= [a_1, a_2]\,,\ and \\ B &= [b_1,\, b_2] \end{aligned}$$

The four operations can be defined as,

Addition

$$A + B = [a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2] \tag{2.4}$$

Subtraction

$$A - B = [a_1, a_2] - [b_1, b_2] = [a_1 - b_2, a_2 - b_1] \tag{2.5}$$

Multiplication

The interval multiplication can be simply defined as,

$$A \cdot B = [a_1, a_2] \cdot [b_1, b_2] = [\min(a_1b_1, a_1b_2, a_2b_1, a_2b_2), \max(a_1b_1, a_1b_2, a_2b_1, a_2b_2)] \tag{2.6}$$

but this technique requires four floating point multiplications. The separation of this formula into different cases depending on the signs of the interval boundaries [7, 8] leads to fewer multiplications as follows,

Table 2.1: The nine cases of interval multiplication $A \cdot B$.

Case	A	B	$A \cdot B$
1	$0 \leq a_1 \leq a_2$	$0 \leq b_1 \leq b_2$	$[a_1b_1, a_2b_2]$
2	$0 \leq a_1 \leq a_2$	$b_1 \leq b_2 \leq 0$	$[a_2b_1, a_1b_2]$
3	$0 \leq a_1 \leq a_2$	$b_1 < 0 < b_2$	$[a_2b_1, a_2b_2]$
4	$a_1 \leq a_2 \leq 0$	$0 \leq b_1 \leq b_2$	$[a_1b_2, a_2b_1]$
5	$a_1 \leq a_2 \leq 0$	$b_1 \leq b_2 \leq 0$	$[a_2b_2, a_1b_1]$
6	$a_1 \leq a_2 \leq 0$	$b_1 < 0 < b_2$	$[a_1b_2, a_1b_1]$
7	$a_1 < 0 < a_2$	$0 \leq b_1 \leq b_2$	$[a_1b_2, a_2b_2]$
8	$a_1 < 0 < a_2$	$b_1 \leq b_2 \leq 0$	$[a_2b_1, a_1b_1]$
9	$a_1 < 0 < a_2$	$b_1 < 0 < b_2$	$[\min(a_1b_2, a_2b_1), \max(a_1b_1, a_2b_2)]$

Division

The interval division A/B is defined using six cases given that $0 \notin B$ [7, 8], as shown in the table below,

Table 2.2: The six cases of interval division A/B, where $0 \notin B$.

Case	A	B	A/B
1	$0 \leq a_1 \leq a_2$	$0 < b_1 \leq b_2$	$[a_1/b_2, a_2/b_1]$
2	$0 \leq a_1 \leq a_2$	$b_1 \leq b_2 < 0$	$[a_2/b_2, a_1/b_1]$
3	$a_1 \leq a_2 \leq 0$	$0 < b_1 \leq b_2$	$[a_1/b_1, a_2/b_2]$
4	$a_1 \leq a_2 \leq 0$	$b_1 \leq b_2 < 0$	$[a_2/b_1, a_1/b_2]$
5	$a_1 < 0 < a_2$	$0 < b_1 \leq b_2$	$[a_1/b_1, a_2/b_1]$
6	$a_1 < 0 < a_2$	$b_1 \leq b_2 < 0$	$[a_2/b_2, a_1/b_2]$

Eight cases are required for defining the interval division in case of $0 \in B$ [8], as shown in the table below,

Table 2.3: The eight cases of interval division A/B, where $0 \in B$.

$Case$	A	B	A/B
1	$0 \in A$	$0 \in B$	$(-\infty, \infty)$
2	$0 \notin A$	$B = [0, 0]$	ϕ
3	$a_2 < 0$	$b_1 < b_2 = 0$	$[a_2/b_1, +\infty)$
4	$a_2 < 0$	$b_1 < 0 < b_2$	$(-\infty, a_2/b_2] \cup [a_2/b_1, +\infty)$
5	$a_2 < 0$	$0 = b_1 < b_2$	$(-\infty, a_2/b_2]$
6	$a_1 > 0$	$b_1 < b_2 = 0$	$(-\infty, a_1/b_1]$
7	$a_1 > 0$	$b_1 < 0 < b_2$	$(-\infty, a_1/b_1] \cup [a_1/b_2, +\infty)$
8	$a_1 > 0$	$0 = b_1 < b_2$	$[a_1/b_2, +\infty)$

Since the output should bound bounding for all the possible values, one of the two bounds of the output interval must be $\pm\infty$ in case of division by interval containing zero, excluding the undefined case (2).

2.3.2 Comparison Relations

Comparison relations of the interval arithmetic are not trivial as those of the floating-point arithmetic. The four main comparison relations can be defined as,

1. $A = B \iff a_1 = b_1 \wedge a_2 = b_2$
2. $A \leq B \iff a_1 \leq b_1 \wedge a_2 \leq b_2$
3. $A \subseteq B \iff a_1 \geq b_1 \wedge a_2 \leq b_2$
4. $A < B \iff a_2 < b_1$

2.3.3 Rounding Modes

There are two rounding modes defined in interval arithmetic, rounding towards outside(out) and rounding towards inside (in) [10], such that,

$$out\,(A \circ B) = \left[a \bigtriangledown b, a \bigtriangleup b\right] \tag{2.7}$$

where $\bigtriangledown$ is rounding towards $-\infty$ and $\bigtriangleup$ is rounding towords $+\infty$. And,

$$in\,(A \circ B) = \left[a \bigtriangleup b, a \bigtriangledown b\right] \tag{2.8}$$

2.3.4 Some of the Interval Arithmetic Properties

Due to [7, 8, 48] interval arithmetic have the following properties,

1. $A + B = B + A$

2. $A \cdot B = B \cdot A$

3. $0 + A = A + 0 = A$

4. $A \cdot 1 = 1 \cdot A = A$

5. $A \cdot 0 = 0 \cdot A = 0$

6. $A \subseteq B \wedge C \subseteq D \Rightarrow A \circ C \subseteq B \circ D$

7. $a \in A \wedge b \in B \Rightarrow a \circ b \in A \circ B$

8. $A \cdot (B + C) \subseteq A \cdot B + A \cdot C$ (sub-distributive law)

2.3.5 Overestimation Problem

To use any tool efficiently one has to know its weakness. The main problem in interval arithmetic is the overestimation of the result interval, in which the result interval guarantee the bounding of the correct result but with an overestimation. That overestimation has two main sources,

Variables dependences

The variables dependences problem happens when the same variable appears in the equation more than one time, in which each instance of the variable is treated independently. As shown in the example below, the native interval treatment of subtraction of two instances of the same variable leads to an over estimation.

Example 2. Let $X = [1, 3]$, due to the defined interval operation,

$$X - X = [1, 3] - [1, 3] = [-2, 2] \supset [0, 0]$$

while X can not be 1 and 3 at the same time. For an obvious dependency as this example, the problem can be simply solved in the software layer. However other more complex examples are not easy to solve.

The main solution for this problem is by reordering the equations, in the design phase, for minimum multi-instances of the same variable. Also, for intervals with small width the variable-dependency is less significant and disappears iteratively, as demonstrated in some of the ray-tracing interval algorithms in Chapter 3.

Sub-distributive law

As mentioned in the interval arithmetic properties Subsection (2.3.4), interval multiplication is not ideally distributed over interval addition. This problem is solved with the same techniques used with variable dependency problem.

Example 3. Let $A = [1, 3]$, $B = [2, 2]$ and $C = [-2, -2]$,

$$\begin{aligned} A \cdot (B + C) &= [1, 3] \cdot ([2, 2] + [-2, -2]) \\ &= [0, 0] \end{aligned}$$

while,

$$\begin{aligned} A \cdot B + A \cdot C &= [1, 3] \cdot [2, 2] + [1, 3] \cdot [-2, -2] \\ &= [2, 6] + [-6, -2] \\ &= [-4, 4] \supset [0, 0] \end{aligned}$$

In general if these overestimations are not treated adequately, they will accumulate. This produces trivial and unusable outputs.

2.4 Interval Functions

In general, the range of a function output values over the interval can be defined as [8, 10],

$$range(f, X) = [f(x) \,|x \in X] \tag{2.9}$$

where $X \in I(\mathbb{R})$. And for multi-variable expression,

$$range(f, X_1, ..., X_n) = [f(x_1, ..., x_n) \,|x_1 \in X_1, ..., x_n \in X_n] \tag{2.10}$$

where $X_1, ..., X_n \in I(\mathbb{R})$.

Even that the range of output is very tight but also it is so complex and expensive to calculate. A more direct calculation is the interval evaluation of an arithmetic expression denoted by,

$$F(X_1, ..., X_n)$$

Due to the interval operations overestimation, an inclusion property is defined such that [8, 10],

$$F(X_1, ..., X_n) \supseteq range(f, X_1, ..., X_n) \tag{2.11}$$

The calculation of the range of function values is equivalent to calculating the global maximum and minimum of an expression over a bounding interval for each variable, which is a very difficult task [7]. On the other hand the interval functions are an easy and fast way for result bounding, with a range of overestimation. The overestimation decreases as we iteratively decrease the interval size, if possible. Also off-line symbolic variable arrangements can produce more tight outputs.

2.4.1 Examples of Interval Elementary Functions

Elementary functions can be defined explicitly as an interval functions more tightly. Following, some examples for interval elementary functions are given,

For $X = [x_1, x_2]$

Square function

$f(X) = X^2$,

$$X^2 = \begin{cases} [x_1^2, x_2^2], & 0 \le x_1 \le x_2 \\ [x_2^2, x_1^2], & x_1 \le x_2 \le 0 \\ [0, \max(x_1^2, x_2^2)], & x_1 \le 0 \le x_2 \end{cases}$$

Note that the third case (when $0 \in X$) is different from defining $f(X) = XX$.

Power function

$f(X) = a^X$,

$$a^X = [a^{x_1}, a^{x_2}]$$

where $a \in \mathbb{R}$.

In the same way many elementary functions can be defined more explicitly.

2.5 Interval Theory Extensions

Many extensions for the main interval theory are introduced to improve some of the weak points of the classical theory. Two of the most famous extensions are the Kaucher [45] and the Modal [46–48] intervals. Although these extensions improve the quality of the output, but also they add much more complexity to the interval theory, and hence its hardware realization. Also there are some applications that

can be implemented using the extended theory and can not be realized using the classical intervals only [9].

Kaucher and Modal intervals are isomorphic, but Modal intervals have a logical meaning not present for Kaucher[48]. Since the modal intervals are believed to be the logical extension for missing parts in the classical theory [48], it will be briefly introduced in the following subsections.

2.5.1 Modal Intervals

The naming of Modal interval is due to that its intervals are treated in two modes of quantifiers, there exist ($\exists$) and for all ($\forall$). The set of Modal intervals are defined as,

$$I^*(\mathbb{R}) = \{(X, Q) \,|X \in I(\mathbb{R}), Q \in \{\exists, \forall\}\} \tag{2.12}$$

where X is a classical interval number and Q is a quantifier.

The classical intervals are treated only in the mode of there exist ($\exists$), and so adding the other pair of quantifier, for all ($\forall$) is considered as the logical extension for the classical intervals.

The equation,

$$a + x = b$$

has only one solution using the classical interval theory but four possible solutions in modal intervals [46].

Example 4. Let,

$$\begin{aligned} a &\in [1, 2]\,, \textit{ and} \\ b &\in [3, 7] \end{aligned}$$

The equation $a + x = b$ can have four solutions using Modal intervals depending on the used quantifiers, such that,

1. $\forall\,(a \in [1, 2])\ \forall\,(x \in [2, 5])\ \exists\,(b \in [3, 7])$,
2. $\forall\,(a \in [1, 2])\ \forall\,(b \in [3, 7])\ \exists\,(x \in [1, 6])$,
3. $\forall\,(x \in [1, 6])\ \exists\,(a \in [1, 2])\ \exists\,(b \in [3, 7])$,
4. $\forall\,(b \in [3, 7])\ \exists\,(a \in [1, 2])\ \exists\,(x \in [2, 5])$

According to [46] only the first solution is be achieved using the classical intervals. The other three, yet logical, solutions need the Modal extension. The four solution have a different logical meaning depending on the quantifier ($\forall$ or $\exists$), and can be used for modeling problems with different logical meaning.

Canonical representation

Modal intervals are defined in a canonical form containing the quantifiers data, such that,

$$[a_1, a_2]' = \begin{cases} ([a_1, a_2], \exists), & if\ a_1 \leq a_2 \\ ([a_2, a_1], \forall), & if\ a_1 > a_2 \end{cases} \tag{2.13}$$

While the first part is the normal representation in the classical intervals, the second one is considered improper representation there.

Fig. 2.3 shows that the proper intervals, improper intervals and point-wise numbers are in the same context of zero and, positive and negative numbers. So, that's another way for seeing the Modal interval as the completion of the classical intervals.

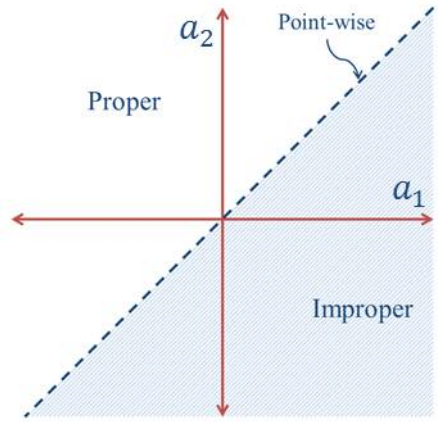


Figure 2.3: Proper, improper and point-wise intervals.

Example 5. Let A' and B' are two modal intervals such that $A' = [1, 3]'$ and $B' = [4, 2]'$,

Table 2.4: Two examples on the canonical representation of the Modal intervals.

	Classical Interval	Mode
A'	$[1, 3]$	$\exists$
B'	$[2, 4]$	$\forall$

Even that Modal intervals have a lot of useful priorities, its arithmetic operation is more complex than the classical intervals. Compared to the classical

intervals multiplication which consists of nine cases, the modal interval multiplication requires sixteen cases (For more about Modal intervals operations please refer to [46]). Beside the increased complexity of arithmetic operations, comparison relations are not straight forward. The increase in complexity reflects on the hardware implementation as increase in area and delay of the circuits.

2.6 Conclusion

Interval analysis and arithmetic are believed to be much more powerful and reliable compared to the point-wise numbers and techniques. Interval arithmetic provides more reliable results than that of the regular floating-point arithmetic. Also interval analysis techniques are more powerful than the point-wise similar algorithms, since they provide more accurate solutions, with a guarantee of convergence. Interval analysis can be also used for speeding-up calculation by using it in an intermediate calculation stage.

In general intervals are continuous sets of numbers represented by their bounding values. The point-numbers can be considered as a special case in which the two bounds have the same value.

There is a wide range of applications for interval analysis and arithmetic can be used in. These applications are split into four main categories, bounding roundoff error, representing physical quantities with a range of uncertainty, interval based analysis algorithms, and molding logically interval applications.

Although that interval arithmetic is not supported in hardware by the majority of the current commercial processors, it is expected that it may be completely supported after its standardization.

Interval arithmetic can be as fast as floating-point arithmetic with a little extra hardware. Also it shows that the interval arithmetic hardware is immune against the regular floating-point problems such as underflow, overflow, division by zero exceptions. The interval units hardware may support floating-point operations in case of no interval operations are scheduled to the processor.

Chapter 3

Ray-Tracing

3.1 Introduction

Ray-tracing simulates the nature, since the images are rendered in ray-tracing by simulating light reflection. Ray-traced images (or video frames) are very realistic, since it contains realistic reflections, refractions and shadows. Compared to the common on-line rendering mode, rasterization (Appendix B), ray-tracing produces much more realistic images, due to its realistic reflections, refractions and shadows. Moreover, compared to rasterization with its local lighting, ray-tracing is a global lighting rendering technique in which light sources affect all members of the sense globally.

Ray-tracing is yet considered to be much slower than rasterization, for the scenes with common complexity. Even so, for high quality off-line rendering, ray-tracing is a very common technique. Most of the animated high quality 3D cinema movies are ray-traced. Fig. (3.1) shows a clear difference in quality between the ray

(a) (b)

Figure 3.1: A clear difference in quality appears between the ray-tracing and rastrization. (a) Ray-traced Pixar's Cars movie [49]. (b) Rasterized Pixar's Cars game.

traced Pixar's Cars movie [49] and the rasterized Pixar's Cars game. While Fig. (3.2) shows a ray-traced image containing real refractions and reflections. Ray-tracing is also common in physical simulations, since it can be used in simulation of optics and electromagnetic waves.

Figure 3.2: Realistic ray-traced images contains real refractions, reflections and shadows [32, 34].

Many works are introduced for improving ray-tracing and towards real-time systems. These improvements are made in the algorithm, the software and the hardware layers concurrently. These works are split into two directions, speeding-up the tracing process, and improving the accuracy of the output.

Many algorithms are introduced for speeding-up the ray-tracing process, as [13–17, 21, 23]. Other algorithms are used for more reliable calculations, as [4, 12, 18–22, 24, 50]. Many of these algorithms are specialized for specific types of objects and others are general for any ray traced object. Some of these algorithms are discussed in Sections 3.3 and 3.4.

Some implementations in the hardware layer are introduced trying to achieve a real-time ray-tracing. In [27] a rendering architecture named SaarCOR is proposed

for ray-tracing triangle-mesh surfaces. The real hardware was not implemented but only simulations were introduced. In [28] a programmable hardware called RPU (Ray-tracing Programmable Unit) is built over FPGA running on 66 MHz and capable of (512×304) pixels resolution at at low frames rates (8 frames/second on average). PRU combines the features of CPUs, GPUs, and custom hardware in order to implement a fully programmable, parallel processor [28]. Also a ray-tracing for computer game on PCs was introduce with cooperation with Intel [51].

Other implementations use the capabilities of the modern CPU architecture, like SIMD (Single Instruction Multiple Data) vector processing [29, 30]. Also GPUs (Graphical Processing Units), which were originally designed for rasterization, are used for ray-tracing, like in [31].

Many programs are built on the software layer for supporting ray-tracing. OpenRT, in context with OpenGL, is introduced in [2] as an API supporting real-time ray-tracing. In another direction many off-line rendering softwares were produced, like [32–34].

Since that the main drawback of ray-tracing is its long processing time, a completely implemented interactive ray-tracing can be easily dominates all the other rendering techniques. According to [4] 95% of the ray-tracing time is spent in finding the intersections between the ray and the scene objects. So, reducing the ray-object intersection processing time is the main challenge in ray-tracing. Many of the improvements in ray-tracing algorithms are concerned with the intersection process. In general the calculation of the intersection calculation technique depends on the way in which the scene objects are represented. There are three major representations for objects used in ray-tracing, triangles-meshes, implicit surfaces and parametric surfaces.

Due to [1] the rending of very complex scenes is faster in ray-tracing than rasterization. In [1] a complete CAD model of Boeing 777 airplane made of 350 million triangles and 12 GByte in size is ray traced at multiple frame per second rate (5 frames/second), while its rasterization needs up to minutes. This is due to the fact that in rasterization all scene objects needed to be accessed including the millions of triangles that are eventually obscured by other triangles that are closer to the viewing point, while in ray-tracing the needed data only is loaded from memory. And so, as computer graphics grows more complex as the need for ray tracing increase.

Ray tracing by its nature have a high level of parallelism, since each ray can be traced independently of the others. In fact, some researchers at Intel think that running ray-tracing on multi-cores processors will kills rastrization [5].

Interval analysis and arithmetic are used efficiently in improving ray-tracing accuracy and speed. And as mentioned before, hardware support will improve

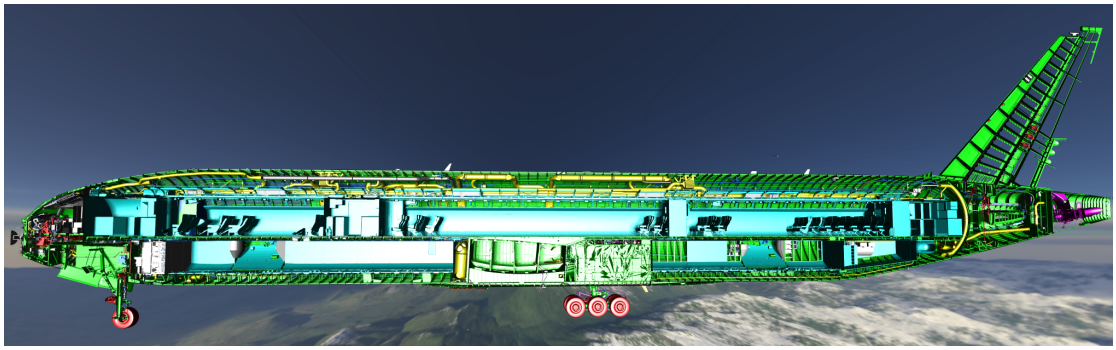


Figure 3.3: Section in the complete model of the Boeing 777 airplane, of 12 GByte in size, ray-traced at multiple frames per seconds [1].

the interval based algorithms significantly and so improves the performance of ray-tracing.

Fig. 3.4 shows a proposed comparison technique for graphics rendering hardware circuits. The figure shows the rendered resolution in mega pixels versus the number of frames per second. The standard resolution and real-time frame rates are marked using dotted lines. The curved solid lines show the expected response for a regular rendering system, by rendering higher resolution at lower frame rates and vise versa. Hence, the rendering systems should be compared using their curves. Also hardware can be benchmarked by satisfying a required resolution and frame rates, as the current TV HD 1080i high definition standard at 100 frames per second.

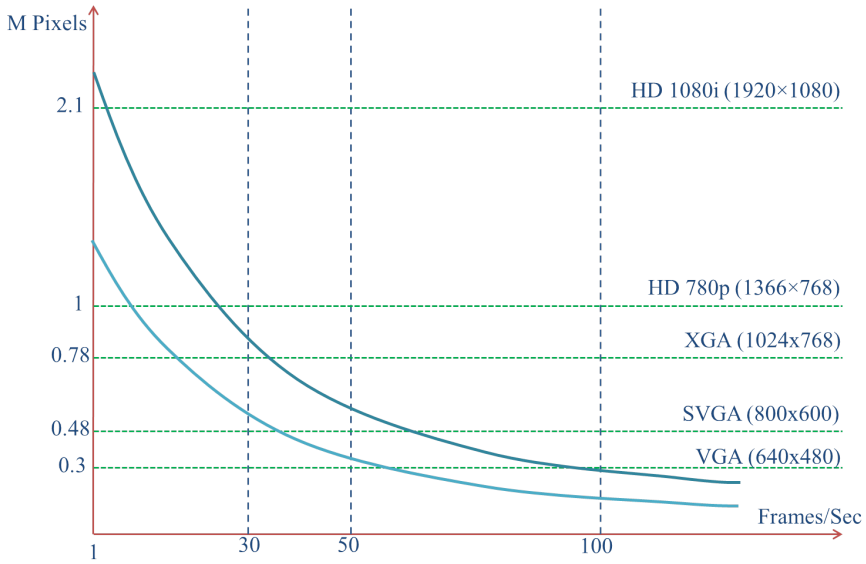


Figure 3.4: A proposed graph for the comparison of graphics rendering hardware showing the standard resolution and real-time frame rates.

3D Object Representations Methods

Computer created 3D models have in general three main type of representations,

- **Implicit surfaces**: in which objects are defined by the implicit equation of the surfaces constructing them.
- **Triangle meshes**: objects are modeled using meshes of triangles. Triangles meshes are one type of surface tessellations.
- **Parametric surfaces**: the objects' surfaces are described using parametric equations. The parametric surfaces can be Bézier, B-Splines or NURBS (Non-Uniform Rational Basis Spline).

Each type of the representations has its own advantages over the other surface. Implicit surfaces are common in physics simulations, while triangle-meshes is the dominate representation used in computer graphics in either rasterization or ray-tracing. Parametric surfaces have many promising properties as a final rendered format, but mainly parametric surfaces are transformed to triangle-meshes before rendering.

Each of the three representations has its own ray-tracing algorithms concerning the calculation of the intersection point, but the main ray-tracing frame is common among them all. The main algorithms used within each representation are introduced in the following sections.

The rest of this chapter is organized such that, in Section 3.2 the ray-tracing rendering mechanism is demonstrated. Ray-tracing implicit surfaces, triangle-mesh surfaces and parametric surfaces are introduced in Sections 3.3, 3.4 and 3.5 respectively. The last section (3.6) is concerned with the speeding-up techniques.

3.2 Rendering Mechanism

Rendering images using ray-tracing is based on the simple idea of light reflections. A ray (or more) is traced from the camera (also called eye) to each pixel (picture element) of the rendered image, which is visually placed in front of the camera. Each ray reflects several times on the scene's objects, for calculating the final value of the pixels. Rays vanishes after certain number of reflections, or certain traveled distance, or by escaping out of the scene. Rays are traced in a revers manner starting from the eye towards the objects and the light sources, since the relevant rays are only the light rays which compose the rendered image by reaching the camera.

Rays are divided into three categories,

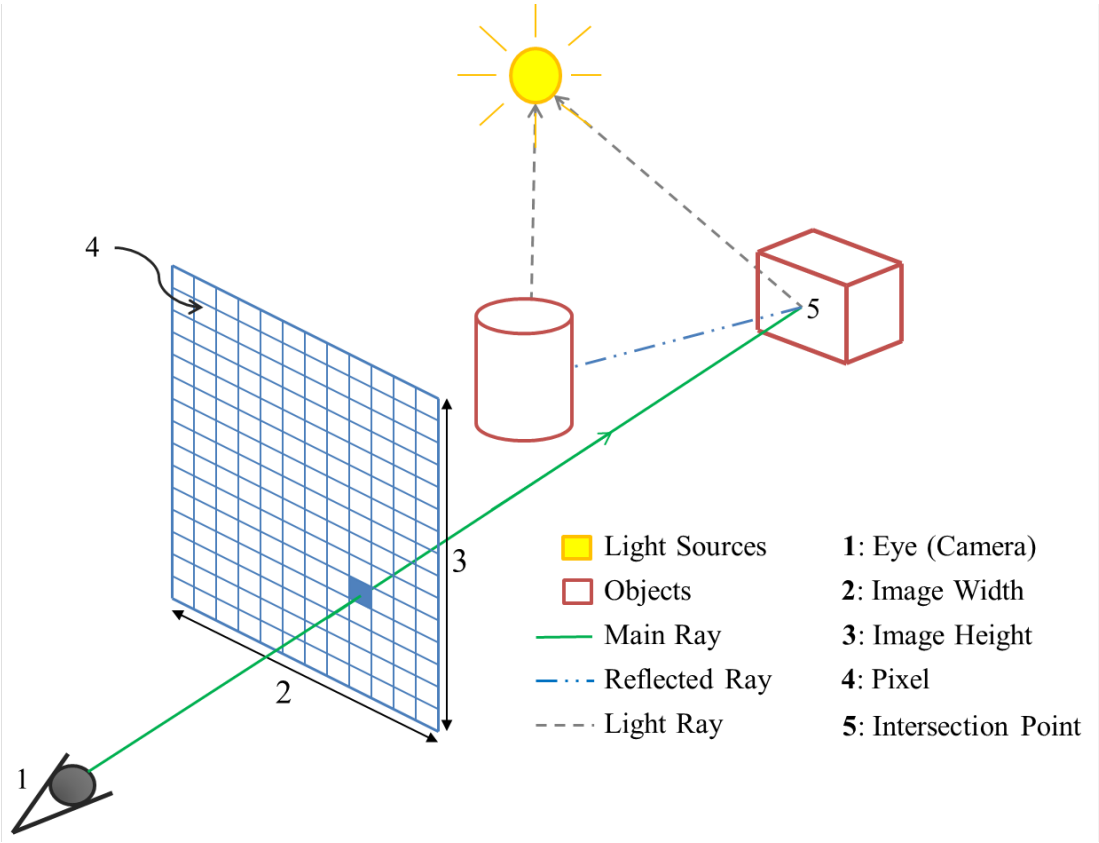


Figure 3.5: Simple ray-tracing is demonstrated on a sense with two objects and a single light source. In general a ray or more is traced thought each pixel of the image plane.

- Main rays: The initial rays coming out of the eye, passing through pixels, and intersecting the nearest to eye objects.
- Secondary rays: The reflected and refracted rays from scene's objects.
- Light rays: rays coming out from each intersection point, between main or secondary ray and objects, towards each light source in the scene.

While main and secondary rays are used to calculate the geometry of the scene, light rays are used to calculate the illumination, and so the color, of the intersection points. If there is an obstacle in the way of the light ray, it is called then a shadow ray, and the intersection point falls in the shadow of this obstacle. So, realistic shadows are calculated. Light rays do not reflect, they reach the light source in a straight way. Main and secondary rays can split to more than one ray in case of hitting a refractive object. Each single ray can be traced independently in a completely parallel manner, if resources are available.

Fig. (3.5) shows a simple scene with two objects and one light source. A main ray comes out of the eye; passes through a pixel in the virtual image plane, and they hit the cuboid object. At the intersection point two rays are created, a light

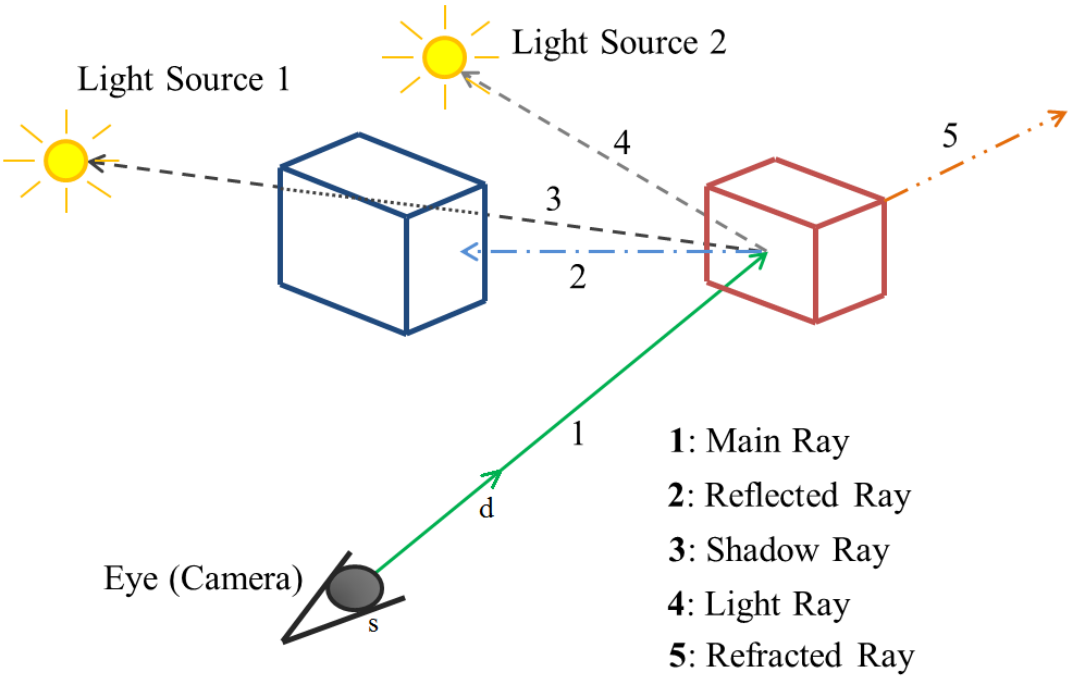


Figure 3.6: A sense with two light sources shows main, reflected, refracted, light and shadow rays.

ray for calculating illumination at the intersection point, and a reflected ray. The new reflected ray will act as a main ray with the intersection point as its origin. This ray may reflect from another object, as shown in the figure, or reach the light source, or vanishes through the background. Vanished rays are assigned the color of the background.

Fig. (3.6) shows a sense with a two light sources. The figure shows that there are light rays (or shadow rays) for each light source. This figure shows also that the main ray can splits into two secondary rays, reflected and refracted, in case of hitting a refractive object. In case of light rays the position of intersection is not required to be calculated. The necessary information is only the presence or absence of intersection. In case the light ray intersects with any object before reaching the light source, it means that the intersection points falls in shade.

Finding the intersection point, as mentioned before, is the most important and time consuming stage of ray tracing. Intersection point calculation will be discussed for each of the main object representation in the following sections. However, the ray equation is generally given as [21],

$$r = s + dt \tag{3.1}$$

where $s \in \mathbf{R^3}$ is the initial point (the source for the main ray and the intersection point for the secondary rays), $d \in \mathbf{R^3}$ is the ray direction, and $t \in \mathbf{R}$ is the

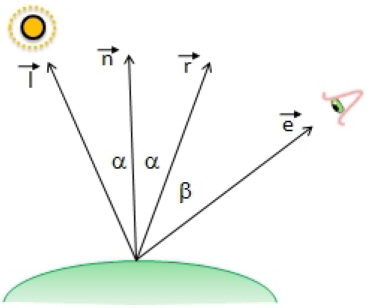


Figure 3.7: Reflected ray has the same angle with the normal as the light ray [21].

scanning parameter ($t = 0$ at the initial point). The parametric shape of the equations simplifies the identification of the nearest to camera object, since the intersection with the smallest positive t is the nearest to eye intersection. The direction is simply calculated by subtracting the pixel position from the source position for the main ray. While for secondary rays the normal to the surface at the intersection point is required to be calculated. The direction of the secondary ray (reflected or refracted) depends on the incident and normal directions. In some techniques normal to the surfaces are calculated off-line and stored with the surface data. Fig. 3.7 shows that the reflected ray has the same angle with the normal as light ray. In case of that the angle between the incident ray is the same of the reflected ray, no reflected ray is created. Fig. 3.8 shows that the refraction angles are calculated according to Snell's law.

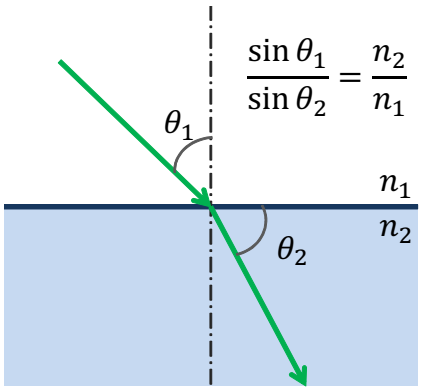


Figure 3.8: Snell's Law.

3.3 Ray-Tracing Implicit Surfaces

Implicit surfaces in general are defined by its implicit equations, such that,

$$f(x, y, z) = 0 \tag{3.2}$$

in which the implicit equation equals to zero on the surface only, and has a non-zero value elsewhere. Implicit surfaces are common in physics simulations more than computer graphics. Complex objects are assembled of multiple implicit surfaces. Fig. 3.9 shows a collection of different implicit surfaces and their equations.

The intersection between the ray and the surface is equivalent to root finding. The intersection point is found simply by substituting the ray equation (3.1) into the surface equation (3.2), such that [21],

$$f\left(s_x + td_x, s_y + td_y, s_z + ty_z\right) = 0 \tag{3.3}$$

where $s = \{s_x, s_y, s_z\}$ is the source point, and $d = \{d_x, d_y, d_z\}$ is the ray direction. Solving such an equation is simple for a surface as the sphere, while it is more complex for other surfaces. Any point-wise root solving algorithm, like bisection, regula-falsi or Newton method (for more about Newton method please refer to Appendix A), can be used for calculating the intersection point. But according to [18] point-sampling methods are not robust and generate defected rendered shapes, especially for thin surfaces. One of the main sources of defects in the point-sampling methods is floating point numbers itself [21].

Interval analysis methods are very common in ray-tracing implicit surfaces, and generate very accurate shapes. According to [21] the drawback of the interval based algorithms is the slow speed of calculations. But as mentioned before, the hardware support for the interval arithmetic can simply overcome this problem.

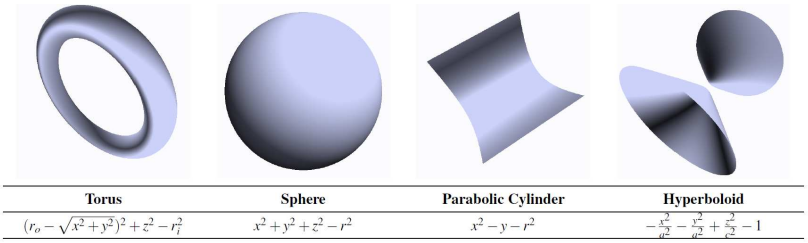
The famous ray-surface intersection methods used in ray tracing are discussed in the following sub-sections. Due to the defects generated by the point-sampling methods, the interval based methods are the commonly used in ray-tracing implicit surfaces.

3.3.1 Mitchell Algorithm

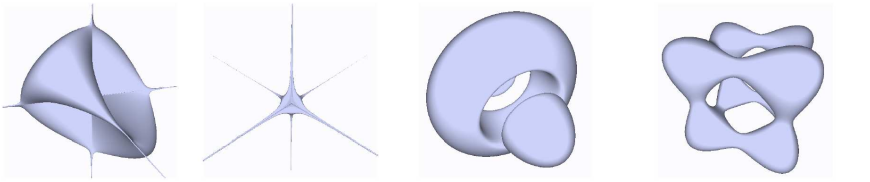
The first interval intersection method for implicit surfaces was introduced by Mitchell in [12]. Although it is a relatively old algorithm (1990), it is still considered as a fast algorithm. The use of interval analysis and arithmetic in this algorithm, like most of implicit surfaces interval algorithm, is for calculating the intersection points in a reliable way.

The algorithm consists of two main phases, root isolation and root refinement. In the root isolation phase, an interval method is used to find all the intervals that enclose only one root. In phase two, any of the point sampling technique can be used for root refinement and finding its location within the interval.

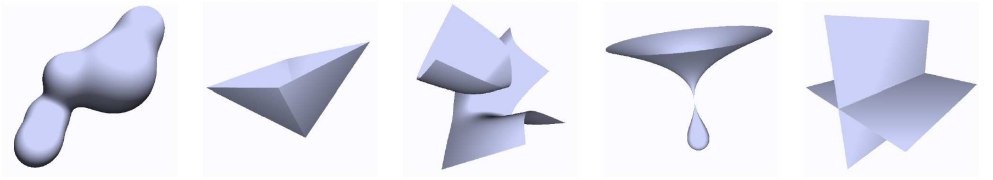
The famous interval over estimation (see sub-section 2.3.5) did not expresses any problem in this algorithm, since the intervals widths are decreased iteratively



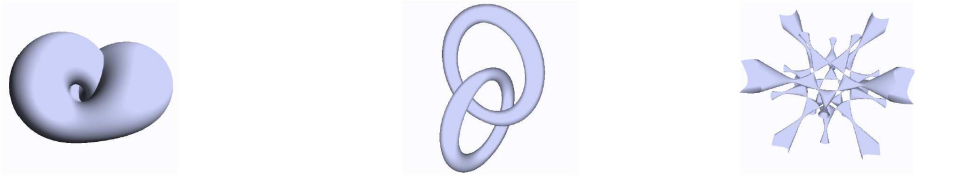
Torus	Sphere	Parabolic Cylinder	Hyperboloid
$(r_o - \sqrt{x^2+y^2})^2 + z^2 - r_i^2$	$x^2+y^2+z^2-r^2$	$x^2 - y - r^2$	$-\frac{x^2}{a^2} - \frac{y^2}{a^2} + \frac{z^2}{c^2} - 1$



Steiner 1	Steiner 2	Mitchell	Tangle
$x^2y^2+y^2z^2+x^2z^2+xyz$	$(x^2y^2+y^2z^2+x^2z^2)^2+xyz$	$4(x^4+(y^2+z^2)^2)+17x^2(y^2+z^2)$ $-20(x^2+y^2+z^2)+17$	$x^4-5x^2+y^4-5y^2+z^4-5z^2+11.8$



Blobby	Absolute value	Inverse function	Teardrop	Intersecting planes
$\sum_{i=1}^{N} \frac{r_i^2}{\|x-p_i\|^2} - 1$	$\|x\| + \|y\| - z$	$\frac{1}{x-y^2} - z$	$0.5x^5 + 0.5x^4 - y^2 - z^2$	xy



Klein Bottle	Linked tori	Barth-sextic
$(x^2+y^2+z^2+2y-1)((x^2+y^2+z^2-2y-1)^2-8z^2)$ $+16xz(x^2+y^2+z^2-2y-1)$	$g(10x,10y-2,10z,13)g(10z,10y+2,10x,13)+1000$ $g(x,y,z,c) = (x^2+y^2+z^2+c)^2 - 53(x^2+y^2)$	$4(\tau^2x^2-y^2)(\tau^2y^2-z^2)(\tau^2z^2-x^2)$ $-(1+2\tau)(x^2+y^2+z^2-1)^2$

Figure 3.9: Different shapes of implicit surfaces with their names and equations [30].

and so the over estimation. The remaining overestimations in the smallest intervals are only considered as an extra void processing for the point sampling technique.

The algorithm can be split into four steps,

Step 1:

Evaluate $F([a,b])$, where F is the implicit surface equation, and a and b are the initial points. If the result interval does not contain zero, so there is no roots in this interval, and no valid intersection exists.

Step 2:

Evaluate the derivative of the surface equation $F'([a,b])$. If the result interval does not contain zero, then the function must be monotonic in this interval. And so, if the function is monotonic and $f(a) \cdot f(b) \leq 0$ (only signs are checked), then there is one root in this interval. The root can be refined after that by any of the standard methods (point sampling).

The derivative of the function measures the change of the output with respect to the input. Given that the monotonic function is a function that preserve order (ex: for increasing monotonic function if $x_2 \geq x_1$, then $f(x_2) \geq f(x_1)$), then the absence of zero in the derivative is a sufficient condition for a function to be monotonic. So, if the function is monotonic and the interval edges of different signs, then there is only one root enclosed by the interval.

Step 3:

If $F([a,b])$ and $F'([a,b])$ both contain zeros. Then subdivide $[a,b]$ at the midpoint into two intervals and repeat the steps starting from step one in a recursive manner.

Step 4:

The process of subdivision stops if the width of the interval reaches machine accuracy (or any desired least accuracy).

Although calculation of the derivative slowdown the speed of one iteration, Mitchell algorithm converges faster than an algorithm not using the derivative [21]. Although the derivative speeds up the algorithm conversion, it can be also considered as the main drawback of the algorithm, because if the derivative can not be derived, Mitchell algorithm is unusable.

According to [18] the point sampling refinement used in Mitchell can produce errors. But in [21] interval approach is also used for final refinement. Also in [30] it was introduced that changing bisection selection improves the speed compared to the original Mitchell. Knoll et al. in [30] also take the advantages of SIMD (Single Instruction Multiple Data) support of the new processors families, and they build a somehow interactive rendering.

3.3.2 Interval Newton Method

Interval Newton is the interval extension for the famous root calculation Newton method. The interval Newton method was originally introduced by R. Moore in [36], as an interval extension for the classical method. Interval Newton is believed to be much more powerful than the classical one. In contrast with the

classical method, the interval Newton method never diverges [6]. For more about interval Newton please refer to appendix A.

3.3.3 Alefeld-Hansen Method

Alefled-Hansen method is a modified version of the interval Newton for implicit surface root isolation. According to [21] this method was introduced separately by Alefeld in [24], and Hansen in [22]. Also [21] shows that Alfeled-Hansen method is faster than the normal common Newton. In this method a new division operator is defined such that,

$$\frac{1}{F'([x_1, x_2])} = \begin{cases} [1/x_2, \infty), & if\ x_1 = 0 \\ (-\infty, 1/x_1], & if\ x_2 = 0 \\ (-\infty, 1/x_1] \cup [x_2, \infty), & otherwise \end{cases} \tag{3.4}$$

3.3.4 Interval Bisection Method

Interval bisection method is a very simple method. It's the interval extension of the famous bisection method. The interval bisection works as follow,

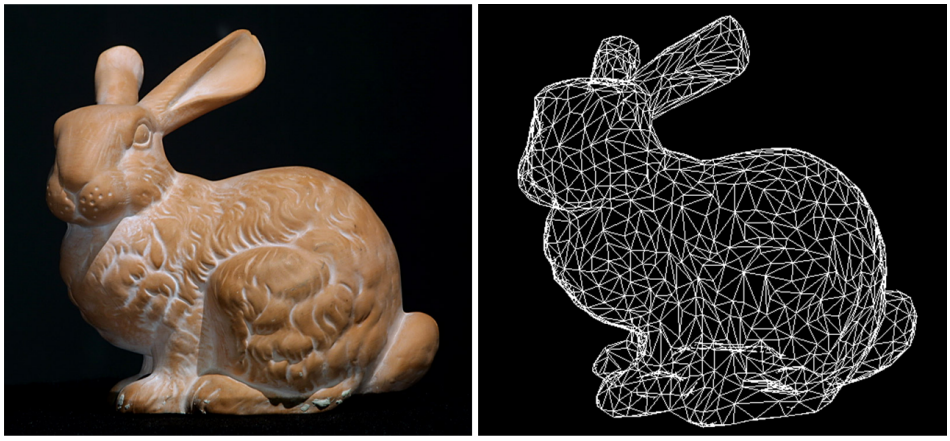
Evaluate $F([a, b])$, where F is the implicit surface equation, and a and b are the initial points. If the resulting interval does not contain zero, so there is no roots in this interval. Else bisect the interval and repeat the test. The test stops if the machine accuracy is reached, or a desired sufficient accuracy. Interval bisection is considered as the non-derivative version of the Mitchell algorithm.

3.3.5 Dedicated Hardware

Implicit surfaces ray tracing needs an interval support from the processor, which is expected in the near future, than building dedicated hardware. Implicit surfaces is an infinite collection which needs interval support for nearly every real function. But that is not the case in the other object representation.

3.4 Ray-Tracing Triangle-Mesh Surfaces

Triangle mesh is the most common representation for 3D-objects in computer graphics. Triangle mesh is a special polygon mesh surface. In this representation all 3D-object surfaces are made of triangles in a tessellation manner (for more about tessellation please refer to [52]). So, the core of ray-tracing is the determination whether the ray intersects the triangles or not. The object details depend



(a) (b)

Figure 3.10: (a) The original model of the Stanford's bunny [53]. (b) Low number of triangle version of the triangle mesh surface of the Stanford's bunny.

on the density of triangles. As the number of triangles increases the quality increases, and the need for more processing also. Fig. 3.10a shows the original Stanford's bunny, and Fig. 3.10b shows the 3D created model using triangle-mesh tessellation.

Triangle is defined by its three vertices, such that,

$$Triangle = \{v_0, v_1, v_2\} \tag{3.5}$$

where v_0, v_1 and v_2 are the triangle vertices, which are stored in memory. but in many ray-triangle intersection algorithms the normal to the triangle surface is stored also with the vertices in memory.

The first ray-triangle intersection for ray tracing was introduced by Snyder and Barr [54]. They used the barycentric coordinates to check the validity of the intersection. Badouel introduced an improvement for Snyder's algorithm [55]. A slightly faster with a lower memory usage algorithm was introduced by Möller and Trumbore [14]. Segura and Feito then proposed an algorithm based on the sign of the volume of the trihedral [15, 56]. In 2004, Wald introduced an improvement in Möller's algorithm by storing pre-computed values for barycentric calculation in memory, instead of the triangle vertices [16].

In another direction, Plöcker coordinates were used as an approach by Teller [57]. An optimization for the Plöcker test was recently proposed [18]. But still the barycentric coordinates is the common way of representing the triangle's contained area in the intersection process. Barycentric coordinates also have an important usage in color and texture mapping. In the following sub-sections the barycentric coordinates and the famous ray-triangle intersection methods are introduced.

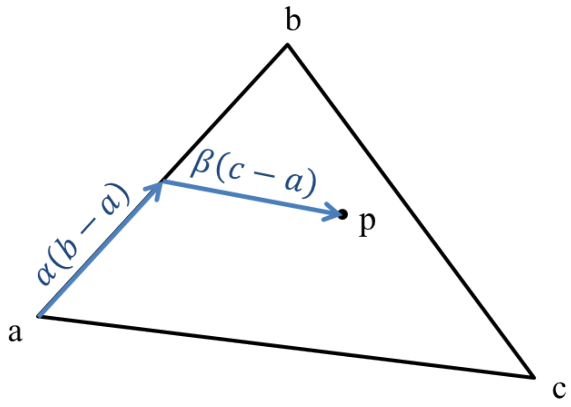


Figure 3.11: Using the barycentric coordinates for describing a point enclosed inside the triangle. The point is described as a sum of the initial point and two weighted vectors.

3.4.1 Barycentric Coordinates

The use of barycentric coordinates is the common way for describing the points belonging to a triangle. In general any point in the space can be represented as a weighted sum of other points, called the barycentric combination [52],

$$p = \sum_{i=0}^{n-1} \alpha_i \cdot p_i, \ \sum_{i=0}^{n-1} \alpha_i = 1 \tag{3.6}$$

where $p_i \in \mathbf{R^3}$ are points in the $3D$ space and α_i is the weighted contribution of the point p_i.

According to [52] the barycentric term is derived from barycenter which means the center of gravity, since the barycentric combination is equivalent to calculating the center of gravity of point masses, which is given by,

$$C_g = \frac{\sum m_i \cdot p_i}{\sum m_i} \tag{3.7}$$

where m_i is the mass at the point p_i.

The barycentric combination can be rewritten as a point and a sum of vectors, such that,

$$p = p_o + \sum_{i=1}^{n-1} \alpha_i \left(p_i - p_o\right) \tag{3.8}$$

since $1 - \sum_{i=1}^{n-1} \alpha_i = \alpha_o$.

The barycentric bounding conditions for a triangle:

Using the barycentric coordinates a bounding condition for the points enclosed by a triangle can be given as following. Let,

$$T = \{a, b, c\} \tag{3.9}$$

is a triangle with a vertices a, b and c.

Any point on the triangle plane can be represented using the barycentric coordinates,

$$p = a + \alpha\,(b - a) + \beta\,(c - a)\,,\ \alpha + \beta \leq 1 \tag{3.10}$$

A point is said to be enclosed by the triangle iff,

$$\begin{aligned} \alpha &\geq 0,\ and \\ \beta &\geq 0 \end{aligned} \tag{3.11}$$

Fig. 3.11 shows how the enclosed by the triangle points are described using the barycentric coordinates. The figure shows a point described as a sum of the initial point and two weighted vectors.

3.4.2 Badouel Algorithm

Badouel algorithm [55] is a slightly modified and faster version of Synder algorithm [54]. In this algorithm the normal to the plane containing the triangle and the plane equation constant are pre-calculated and stored in memory with the triangle vertices.

The normal equation to the triangle plane is given by,

$$n = (b - a) \times (c - a) \tag{3.12}$$

where n is the normal vector and a, b, and c are the triangle vertices.

The implicit equation of the plane can be written as,

$$n \cdot (p - p_o) = 0 \tag{3.13}$$

$$n \cdot p + \lambda = 0 \tag{3.14}$$

where,

$$\lambda = -p_o n \tag{3.15}$$

is constant for any point on the plane.

Step 1: Finding the intersection with the triangle plane

The intersection point is calculated by substituting the ray equation (3.1) in the surface equation (3.14)

$$n \cdot (s + td) = -\lambda \tag{3.16}$$

$$t = -\frac{n \cdot s + \lambda}{n \cdot d} \tag{3.17}$$

There is a valid intersection if,

1. $n \cdot d \neq 0$, the ray is not parallel to the plane, and
2. $t > 0$, intersection in front of the origin, and
3. there is no closer intersection. $t < t_r$, where t_r is the smallest valid previous intersection.

According to [55] step one requires 12 floating point operations, since all the operations are dealing with three-dimension vectors.

Step 2: Intersecting the triangle

In this step barycentric test is used to check if the intersection point lies inside the triangle or not. The intersection point is given by,

$$p = s + td \tag{3.18}$$

The test can be done by calculating α and β in three dimensions coordinates,

$$p_x - a_x = \alpha\,(b_x - a_x) + \beta\,(c_x - a_x) \tag{3.19}$$

$$p_y - a_y = \alpha\,(b_y - a_y) + \beta\,(c_y - a_y) \tag{3.20}$$

$$p_z - a_z = \alpha\,(b_z - a_z) + \beta\,(c_z - a_z) \tag{3.21}$$

For system reduction the triangle can be projected onto one of the major planes xy, xz, or yz. If the triangle is perpendicular to one of the planes, it can not be

projected on it, since it will be projected as a line. Badouel use the same algorithm of Synder to avoid this problem, by rejecting one of the coordinates which is not parallel to the triangle plane.

Let,

$$i_0 = \begin{cases} 0 & ,if\ \ |n_x|\ \ is\, max \\ 1 & ,if\ \ |n_y|\ \ is\, max \\ 2 & ,if\ \ |n_z|\ \ is\, max \end{cases} \tag{3.22}$$

where n_x, n_y and n_z are the normal vector coordinates.

Let $i_1, i_2 \in \{0, 1, 2\}$, such that, $i_0 \neq i_1 \neq i_2$

The barycentric equations can be rewritten now after omitting the i_0 coordinate.

$$p_{i_1} - a_{i_1} = \alpha\,(b_{i_1} - a_{i_1}) + \beta\,(c_{i_1} - a_{i_1}) \tag{3.23}$$

$$p_{i_2} - a_{i_2} = \alpha\,(b_{i_2} - a_{i_2}) + \beta\,(c_{i_2} - a_{i_2}) \tag{3.24}$$

For reusing the calculated data let,

$$\begin{aligned} u_0 &= p_{i_1} - a_{i_1} \\ u_1 &= b_{i_1} - a_{i_1} \\ u_2 &= c_{i_1} - a_{i_1} \\ v_0 &= p_{i_2} - a_{i_2} \\ v_1 &= b_{i_2} - a_{i_2} \\ v_2 &= c_{i_2} - a_{i_2} \end{aligned}$$

α and β can be then calculated,

$$m = \begin{vmatrix} u_1 & u_2 \\ v_1 & v_2 \end{vmatrix} \tag{3.25}$$

$$\alpha = \frac{\begin{vmatrix} u_0 & u_2 \\ v_0 & v_2 \end{vmatrix}}{m} \tag{3.26}$$

$$\beta = \frac{\begin{vmatrix} u_1 & u_0 \\ v_1 & v_0 \end{vmatrix}}{m} \tag{3.27}$$

There is a valid intersection if,

1. $m \neq 0$, and

2. barycentric conditions are valid. Note that the comparisons can be made without the division by m,

   (a) $\alpha \cdot m \geq 0$, and

   (b) $\beta \cdot m \geq 0$, and

   (c) $\alpha \cdot m + \beta \cdot m \leq 1$

Number of required floating-point operations:

Table 3.1: The number of the floating-point operations required for the ray-triangle intersection test in the Badouel algorithm.

	Add	Mul	Div
Step 1	6	3	3
Step 2	22	6	0
Total	28	9	3

The number of the floating point operations is a measure for the required hardware, and not a sufficient index of the algorithm speed. The main index of the algorithm speed is the statistics of the ratio of the triangles rejected by each step.

3.4.3 Möller Algorithm

Möller algorithm [14] is reported to be slightly faster than the Badouel algorithm [55], but it significantly reduces the memory usage. Only the triangles vertices are needed to be stored in the memory, and no need for storing the normals to the triangles planes. Möller is still the common way of calculating the ray-triangle intersection in ray-tracing.

In Möller's algorithm the intersection point is simply calculated by equating the ray equation (3.1) with the barycentric equation (3.10). This is equivalent to a triangle projection.

$$s + td = a + \alpha(b - a) + \beta(c - a) \tag{3.28}$$

This equation can be rewritten as,

$$[-d, b - a, c - a] \begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix} = s - a \tag{3.29}$$

(s, d, a, b, and c are points in 3D and have three components while t, α, and β are scalar numbers). Using Cramer's rule,

$$\begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix} = \frac{1}{|-d, e_1, e_2|} \begin{bmatrix} |e_3, e_1, e_2| \\ |-d, e_3, e_2| \\ |-d, e_1, e_3| \end{bmatrix} \tag{3.30}$$

where $e_1 = b - a$, $e_2 = c - a$, and $e_3 = s - a$

The definitions of vectorial and scalar products in linear algebra lead to $|x_1, x_2, x_3| = (x_1 \times x_2) \cdot x_3 = -(x_1 \times x_3) \cdot x_2 = -(x_3 \times x_2) \cdot x_1$. So, equation (3.30) is reduced to

$$\begin{bmatrix} t \\ \alpha \\ \beta \end{bmatrix} = \frac{1}{k \cdot e_1} \begin{bmatrix} l \cdot e_2 \\ k \cdot e_3 \\ l \cdot d \end{bmatrix} = \frac{1}{m} \begin{bmatrix} l \cdot e_2 \\ k \cdot e_3 \\ l \cdot d \end{bmatrix} \tag{3.31}$$

where $k = d \times e_2$, $l = e_3 \times e_1$, and $m = k \cdot e_1$.

There is a valid intersection if,

1. $m \neq 0$, and
2. Barycentric conditions are valid. The comparisons can be made without the division by m,
   (a) $\alpha \cdot m \geq 0$, and
   (b) $\beta \cdot m \geq 0$, and
   (c) $\alpha \cdot m + \beta \cdot m \leq 1$
3. $t > 0$, intersection in front of the origin, and
4. There is no closer intersection, i.e. $t < t_r$, where t_r is the smallest valid previous intersection.

Number of required floating-point operations:

Table 3.2: The number of the floating-point operations required for the ray-triangle intersection test in the Möller algorithm.

Add	Mul	Div
23	24	9

3.4.4 Segura Algorithm

Segura algorithm was introduced by Segura and Feito in 1998 [15]. This algorithm is reported to be faster than Badouel and Möller algorithms [56]. Segura algorithm

built on the signs of the tetrahedrons enclosed by the main ray and two of the three vertices of the triangle, as shown in Fig. 3.12.

The volume of a tetrahedron *dabc* is defined by,

$$\begin{aligned} volume\,(dabc) &= \frac{1}{6}\begin{vmatrix} x_a - x_d & y_a - y_d & y_a - y_d \\ x_b - x_d & y_b - y_d & y_b - y_d \\ x_c - x_d & y_c - y_d & y_c - y_d \end{vmatrix} \\ &= \frac{1}{6}\begin{vmatrix} x_a & y_a & z_a & 1 \\ x_b & y_b & z_b & 1 \\ x_c & y_c & z_s & 1 \\ x_d & y_d & z_c & 1 \end{vmatrix} \end{aligned} \tag{3.32}$$

where $d = (x_d, y_d, z_d)$, $a = (x_a, y_a, z_a)$, $b = (x_b, y_b, z_b)$ and $c = (x_c, y_c, z_c)$.

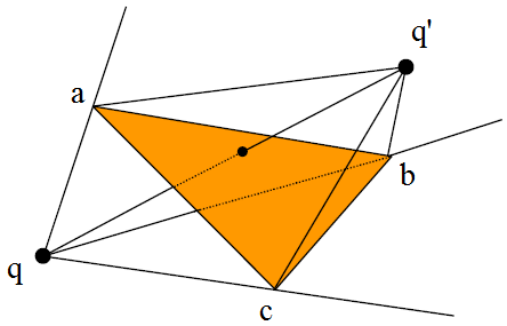


Figure 3.12: Tetrahedrons enclosing the ray and triangle vertices.

Due to [56] the segment *qq*' (the main ray) cuts triangle *abc* iff,

$$\begin{aligned} &sign\,(volume\,(q'abq)) \geq 0 \wedge \\ &sign\,(volume\,(q'cbq)) \geq 0 \wedge \\ &sign\,(volume\,(q'acq)) \geq 0 \end{aligned} \tag{3.33}$$

where,

$$sign\,(x) = \begin{cases} 1, & if\, x > 0 \\ 0, & if\, x = 0 \\ -1, & if\, x < 0 \end{cases} \tag{3.34}$$

Fig. 3.12 shows the segment *qq*' (the main ray) and the tetrahedrons enclosing the triangle vertices.

3.4.5 Dedicated Hardware

Dedicated hardware designs for ray-tracing are achievable compared to implicit surfaces, since triangles made surfaces can be rendered using single algorithm with the same repeated steps. Many dedicated rendering hardware exist for ray-tracing triangle-mesh surfaces. Although all the hardware designs were introduced towards real-time ray-tracing, until now no high quality real-time renderer is available. The main reason for that is the complex ray-triangle intersection algorithms which required many floating-point arithmetic operations. Although the common speeding-up techniques reduce the number of required calculation, they reduce the level of parallelism.

In [27] a rendering Architecture named SaarCOR is proposed for ray-tracing triangle-mesh surfaces. In [28] a programmable hardware called RPU (Ray-tracing Programmable Unit) is built over FPGA running on 66 MHz and capable of (512 × 304) pixels resolution at low frame rates (8 frames/second on average). PRU combines the features of CPUs, GPUs, and custom hardware in order to implement a fully programmable, parallel processor that can do a real-time ray-tracing rendering [28]. Fig. 3.13 shows the main block diagram for the RPU chip. The figure also shows that multiple RPU can be used within the same system. RPU expresses lower performance compared to SaarCOR.

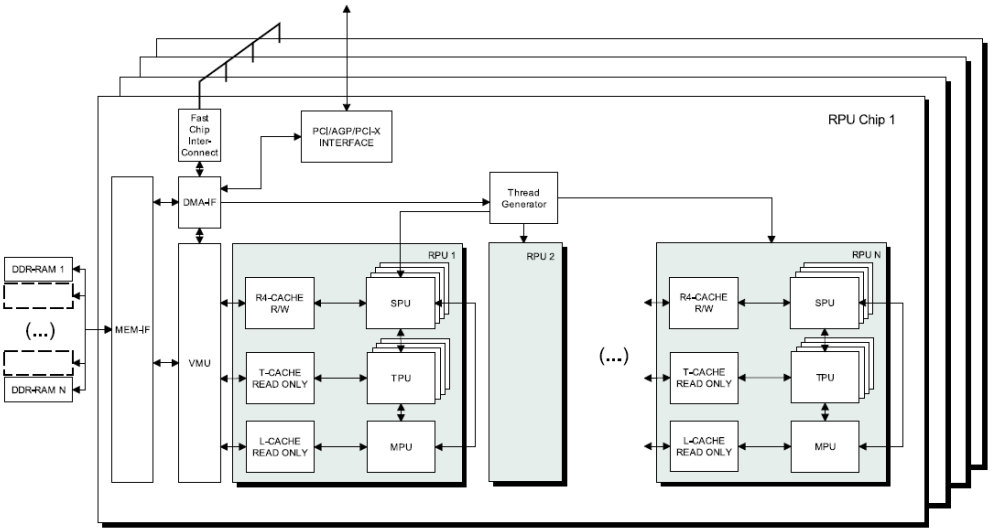


Figure 3.13: The block diagram of the RPU architecture [28].

3.5 Ray-Tracing Parametric Surfaces

According to [52] parametric surfaces are the common technique for representing 3D models in CAGD (Computer Aided Graphical Design) and CAM (computer

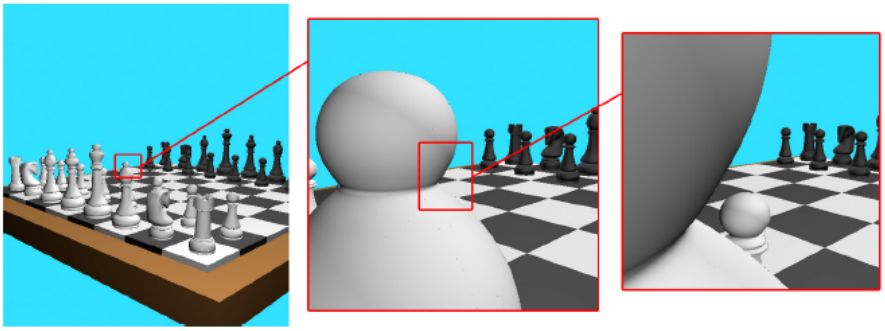


Figure 3.14: Parametric surfaces express a perfect immunity to zooming [19].

Aided Manufacture) tools. Parametric surfaces were first used by the designers of cars industry [52].

Any object can be perfectly assembled using number of parametric surfaces. The parametric surface is described by its control points only, since the same type of parametric equation is used to describe the whole sense.

The two main advantages of these surfaces are its perfect description for the object at any zooming level, and its low storage space required. Fig. 3.14 shows the immunity of the parametric surfaces to zooming. Despite these perfect properties, parametric surfaces can not be rasterized at all, and have to be transformed to triangle-meshes in the pre-rendering phase. Also the ray-tracing of parametric surfaces in its native format is a very slow process [4].

There are three main types of parametric surfaces,

1. Bézier surfaces, and
2. B-Splines (Basis Splines), and
3. NURBS (Non-Uniform Rational Basis Spline).

where NURBS and B-Splines are more general forms of Bézier surfaces. The Bézier surface of order $(n, m)$and parameters (u, v) is defined as,

$$s(u,v) = \sum_{i=0}^{n} \sum_{j=0}^{n} b_i^n(u)\, b_j^n(v)\, p_{i,j} \tag{3.35}$$

where $p_{i,j}$ is the control point (i, j), and

$$b_k^n(t) = \frac{n!}{k!\,(n-k)!} t^k (1-t)^{n-k} \tag{3.36}$$

is the Bernstein polynomials which define the basis functions, where $t = \{u, v\}$ is the function parameter.

While In general tessellations are made for parametric surfaces transforming them into triangle-meshes before rendering, many direct rendering algorithms are

introduced. Interval analysis is a common technique in ray-tracing parametric surfaces. In [20] an iterative algorithm for ray-tracing parametric surfaces using Newton method and interval analysis was introduced, by D. Toth from Ford motors. In [58] an algorithm for ray-tracing Bézier and B-Spline surfaces using hierarchical bounding volumes and interval analysis was introduced. In 2009 an off-line application called Meridian for ray-tracing parametric surfaces directly without tessellations using Modal intervals is introduced [33]. Also there are many non-interval methods for ray-tracing parametric surfaces. An algorithm for ray-tracing NURBS without transformation was introduced in [13]. The algorithm implementation uses CPUs SIMD capabilities. In [19] pre-rendering transformation is made transforming NURBS into Bézier surfaces.

3.6 Speeding-Up Techniques

Improving the intersection process is a very efficient way for speeding-up ray-tracing. In another direction some speeding up techniques are introduced based on splitting space into hierarchies or tracing a bundle of rays coherently. In the following sub-sections three of the famous techniques are introduced.

3.6.1 Space Sub-Division

In this technique space is subdivided using treeing hierarchy as Octrees or binary trees [21]. Such trees reduce the test set very much, but searching the tree itself is not highly parallel and not suitable for multi cores architectures. Among the main drawbacks of tree hierarchy is the fact that the same object assembly may be split on two or more of the tree leafs, which complicates the process. On the other hand it can be built as an efficient software layer, and can be combined with any parallel technique at some level. Fig. 3.15 shows space sub division using Octree.

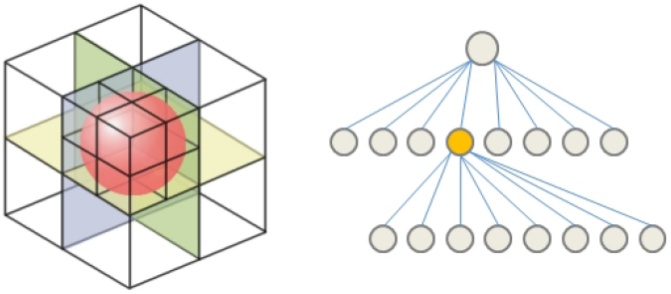


Figure 3.15: Space sub-division using Octree [21].

3.6.2 Coherent Tracing

In such technique a bundle of rays are traced at the same time, and each ray is individually traced only in case the bundle hits some objects [21]. Also there is some varieties form this techniques in which some pixels can be calculated using the bundle tracing only [16]. Such technique is not very efficient with complex scenes.

3.6.3 Bounding Volumes

Bounding volumes is one of the famous ray-tracing techniques [21]. The native idea was introduced in [26], but many further evaluations are proposed until it reaches its current shape [3]. The bounding volumes are used in one level or in a tree like hierarchy [3]. Bounding volumes can be interpreted as irregular space subdivision compared by the uniform grid trees.

In the same analogy of the regular space subdivision, only triangles belong to the volumes passed by the ray are tested for intersection. The number of hierarchy levels and the number of triangles bounded by the same volume are design parameters.

This technique is discussed further in the following chapter and a hardware unit for its implementation is detailed.

3.7 Conclusion

Ray-tracing simulates the nature, since the images are rendered in ray-tracing by simulating light reflection. Ray-traced images (or video frames) are very realistic, since it contains realistic reflections, refractions and shadows. Compared to the common on-line rendering mode, rasterization, ray-tracing produces much more realistic images, due to its realistic reflections, refractions and shadows. Moreover, compared to rasterization with its local lighting, ray-tracing is a global lighting rendering technique in which light sources affect all members of the sense globally.

Ray-tracing produces very realistic images (or video frames) due to its realistic reflections, refractions and shadows , but it yet considered to be much slower than rasterization, for the scenes with common complexity. Even so, for high quality off-line rendering, ray-tracing is a very common technique. Most of the animated high quality 3D cinema movies are ray-traced. Ray-tracing is also common in physical simulations, since it can be used in simulation of optics and electromagnetic waves.

Many works are introduced for improving ray-tracing systems and towards real-time systems. These improvements are made in the algorithm, the software and

the hardware layers concurrently. Also these works are split into two directions, speeding-up the tracing process, and improving the accuracy of the output.

The rending of very complex scenes is faster in tracing than rasterization. This is due to the fact that in rasterization all scene objects needed to be accessed including the millions of triangles that are eventually obscured by other triangles that are closer to the viewing point, while in ray-tracing the needed data only is loaded from memory. And so, as computer graphics grows more complex as the need for ray tracing increase.

The main drawback of ray-tracing is its long processing time, a completely implemented interactive ray-tracing can be easily dominates all the other rendering techniques. Since that, 95% of the ray-tracing time is spent in finding the intersections between the ray and the scene objects. So, reducing the ray-object intersection processing time is the main challenge in ray-tracing.

Ray tracing by its nature have a high level of parallelism, since each ray can be traced independently of the others.

Interval analysis and arithmetic are used efficiently in improving ray-tracing accuracy and speed. And as mentioned before hardware support will improve the interval based algorithms significantly and so improves the performance of ray-tracing.

Chapter 4

Improved Ray-Triangle Intersection Using Interval Analysis

4.1 Introduction

In this chapter we introduce an interval method for speeding-up the ray-triangle intersection process in ray-tracing triangle-mesh surfaces. This method is an interval rejection test used for the fast rejection of triangles not containing a valid intersection. The few triangles that are not rejected by our method are then tested using any of the conventional tests mentioned in Chapter 3. The rejection test may be combined with any of the common ray-triangle intersection tests. This rejection test is the core of our multi-cores ray-tracing architecture introduced in Chapter 5.

This interval test scales efficiently to use highly parallel hardware like the Graphical Processing Units (GPUs), which gives it an advantage over the conventional tree space sub-division, since our algorithm can be implemented in a highly parallel scheme, in contrast with the tree space sub-division. Also by applying a parallel rejection test on the natively parallel ray-tracing enables the design of a highly parallel tracing architecture, in context with the GPUs, as that introduced in the next chapter.

In our experiments (Chapter 6), without any particular hardware support, the rejection test correctly rejects more than 99.9% of the total number of triangles and speeds the ray-triangle intersection process by 1.25 to 2.21 depending on the 3D model. With the proper hardware support in the future, by implementing the IEEE 1778 interval standard [10] in the next generation of the processing units, the algorithm will achieve a much higher speed-up, using only the CPUs. Since the

interval arithmetic support from the hardware will improve the test performance significantly, the anticipated higher speedup may pave the way to real time ray-tracing of video frames.

As mentioned before, most of the interval analysis methods were used in ray-tracing implicit surfaces [12, 21, 22, 24] to provide a higher reliability by calculating the intersection points more correctly, as it is mentioned in Chapter 3. On the other hand, we use interval analysis methods to yield a fast rejection test for the ray-triangle intersection. The main idea behind the test is to make a quick decision on most of the triangles to reject the majority that do not intersect the ray. And those few triangles that pass from the rejection test are tested using any conventional ray-triangle intersection algorithms.

The ray-triangle intersection algorithms require a long execution time on the current general purpose architecture or a large area in case of dedicated hardware. On the other hand the rejection test takes fast decisions and requires much less chip area.

In the algorithm, initially the bounding rectangle containing the triangle's projection is checked. If a final decision to reject the triangle is not reached then the bounding cuboid containing the triangle is checked. If the rectangle and the cuboid are not rejected, any of the common intersection-tests is used for checking the triangle for intersection.

Another algorithm which bounds a group of triangles at the same time, which is very near in its implementation to our introduced algorithm, is mentioned in [3], based on a native bounding idea introduced in [26]. While bounding a group of triangles at the same time can be efficient as a software layer, it's not highly scalable to be implemented on a dedicated architecture design, compared to our algorithm which introduces very high parallel properties. These parallel properties and the small number of synchronizations needed in our algorithm, enables the design of a highly parallel multi-cores architecture for ray-tracing.

4.2 The Interval Rejection Algorithm

Our algorithm is a rejection test using interval analysis and arithmetic for fast rejecting triangles that do not contain a valid intersection in ray-tracing triangle-mesh surfaces. Triangles which are not rejected using the rejection test are tested using any conventional ray-triangle intersection algorithm. The algorithm consists of three on-line phases and one off-line phase.

In the off-line phase the maximums and the minimums of the coordinates of the three triangle vertices are pre-calculated once to be used in the rejection test later. The off-line phase can be done during the creation of the frames, or during

its loading to the memory. While the off-line phase and the on-line steps one and three are mandatory, the rejection test can go without step two.

4.2.1 Step One: Rejecting the rectangles containing the triangles projections

The first step of the algorithm is to check the bounding rectangles of the triangles' 2D projection for rejection, as shown in Fig. 4.1. In this case any two of the three coordinates of the triangle's vertices are selected, which is equivalent to projecting the triangle on the plane of the selected coordinates. The same selections have to be made with all of the triangles, for correct comparisons. According to our experimental results the majority of the triangles are rejected during this phase.

The rectangle containing the triangle's projection is defined by two intervals,

$$X_t = [x_{min}, x_{max}] \tag{4.1}$$

$$Y_t = [y_{min}, y_{max}] \tag{4.2}$$

where x_{min} and x_{max} are the minimum and maximum x-coordinates of the triangle vertices, and y_{min} and y_{max} are the minimum and maximum y-coordinates of the triangle vertices. Maximums and minimums are already calculated during the offline phase, or only one time before starting the intersection test, as mentioned before.

The two intervals X_t and Y_t are representing intervals of straight lines. Using the ray equation (3.1), the intersections between the ray and all the lines bounded by X_t and Y_t are,

$$T_x = \frac{X_t - s_x}{d_x} = \frac{[x_{min}, x_{max}] - s_x}{d_x} \tag{4.3}$$

$$T_y = \frac{Y_t - s_y}{d_y} = \frac{[y_{min}, y_{max}] - s_y}{d_y} \tag{4.4}$$

where T_x is the range of the parameter t in which the ray passes through the interval X_t, and T_y is the range of passing through Y_t.

The triangle is rejected if there is no valid intersection between the two intervals,

$$T_x \cap T_y = \phi \tag{4.5}$$

such as ray 2 in Fig. 4.1, which means that the ray does not pass through the rectangle containing the triangle projection. On the other hand, when $T_x \cap T_y \neq \phi$ there is a possibility that the ray intersects the triangle such as ray 1 in the figure. So this triangle will continue to the next step.

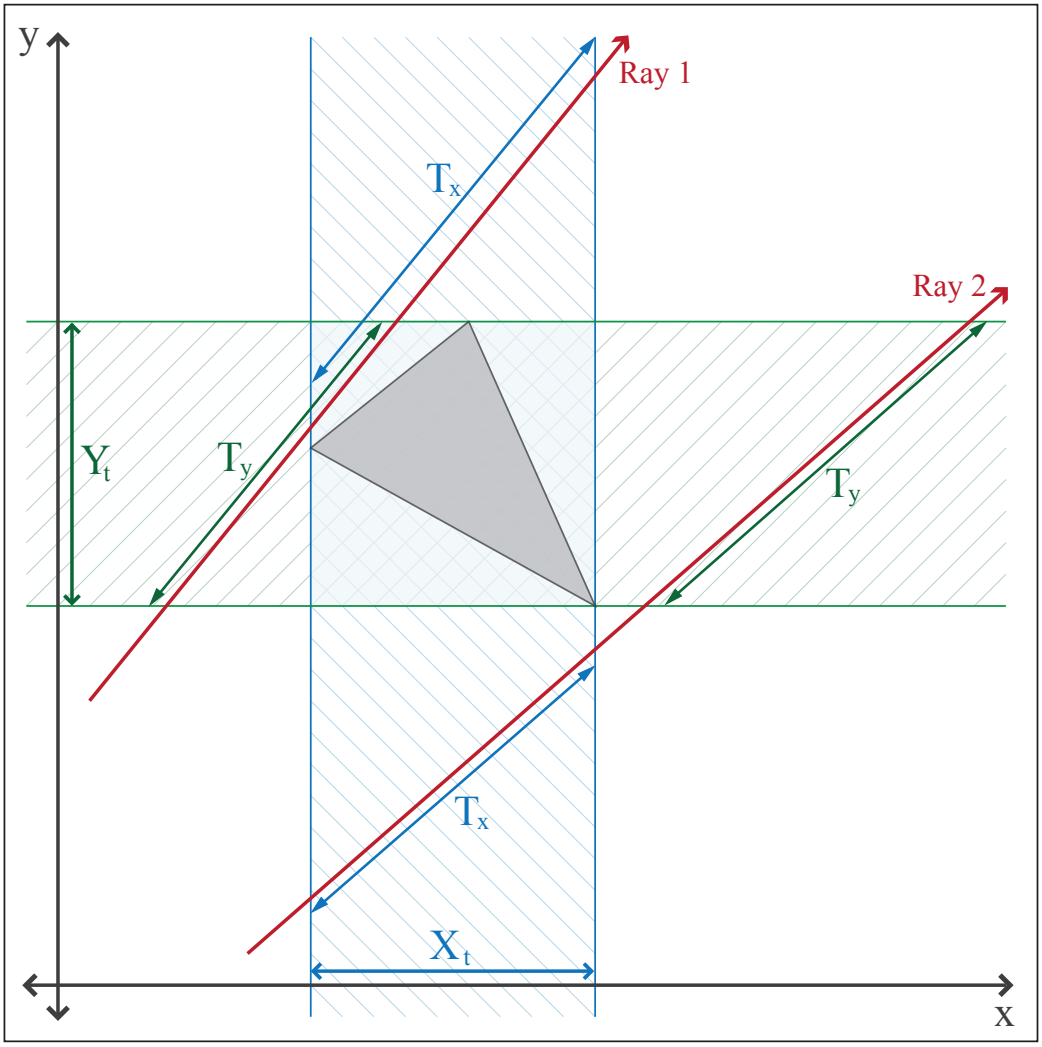


Figure 4.1: Ray 1 intersects the bounding square while Ray 2 passes outside it. The bounding square is defined be the intersection by the lines enclosed by the two intervals X_t and Y_t. It appears from the figure that T_x and T_y have a common region (intersection) Ray 1, while they are completely apart from each other for Ray 2.

Parallel to axis cases (the proof of no special treatment is required):

As shown in Fig. (4.2) the ray may pass perpendicular to one of the selected coordinates (x- or y-coordinate). Even in this case no special treatment is required as it is explained below.

Let the ray be parallel to y-coordinate (the explanations are equivalent also if the ray is parallel to x-coordinate). In the two cases which are mentioned in Fig. (4.2), the range of the parameter t for y-coordinate is,

$$T_y = \frac{Y_t - s_y}{d_y} = [y_1, y_2]$$

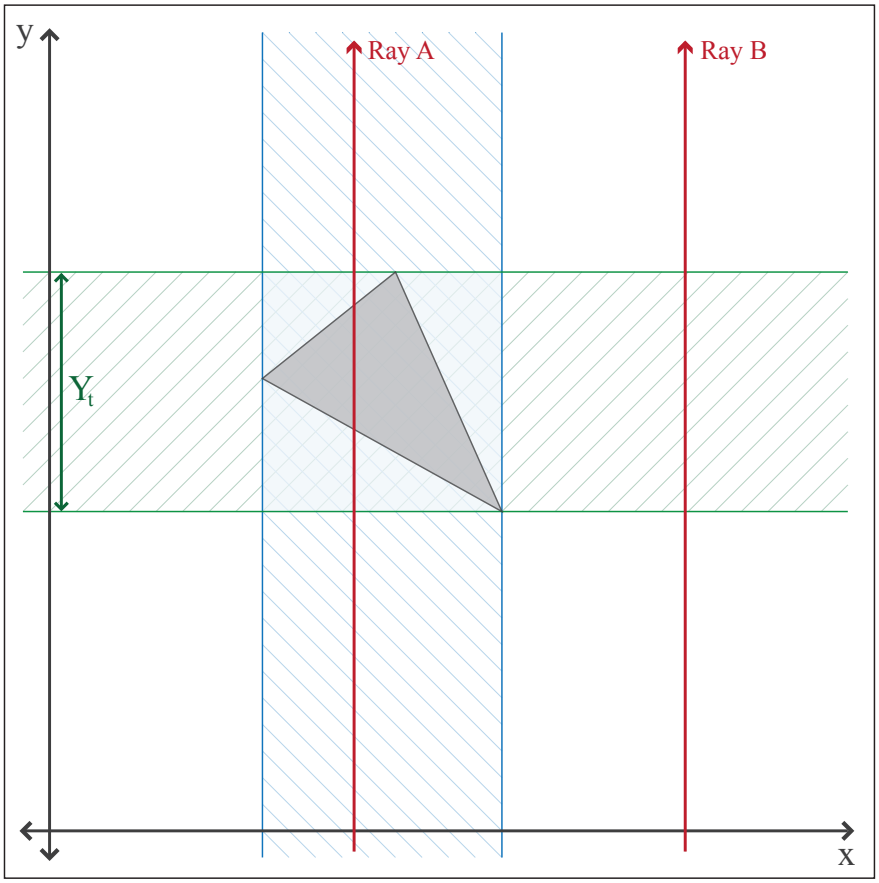


Figure 4.2: No special treatment is required even in the case of parallel to axis rays.

Case 1: The ray passes through the bounding square as Ray A in Fig. (4.2). In this case, $x_{min} < S_x < x_{max}$ and $d_x = 0$. So,

$$T_x = \frac{X_t - s_x}{0} = (-\infty, \infty)$$

and

$$T_x \cap T_y = [y_1, y_2]$$

so, the triangle will not be rejected. This is true.

Case 2: Let the ray is parallel to y-coordinate, and passes outside the bounding square as Ray B in Fig (4.2). In the case of $S_x > x_{max}$ and $d_x = 0$,

$$T_x = \frac{X_t - s_x}{0} = (-\infty, -\infty)$$

while in the case of $S_x < x_{min}$ and $d_x = 0$,

$$T_x = \frac{X_t - s_x}{0} = (\infty, \infty)$$

In the both cases the intersection is,

$$T_x \cap T_y = \phi$$

so, the triangle will be rejected. This is true.

4.2.2 Step Two: Rejecting the cuboid containing the triangles

In this step the triangles which are not rejected in step one are checked for rejection using the third coordinate, which is not used in step one. Using three coordinates for rejection test is equivalent to rejecting the Cuboid containing the triangles. If highly parallel architecture is use, like the Graphical Processing Units (GPUs), step one and two can be executed at the same time. Even that step two is not a mandatory step, and test can be done without it. Implementing step two or not is a design decision, where the added required on-chip area is compared with gained performance from this step as detailed in Chapter 6.

The cuboid containing the triangle is defined by three intervals X_t, Y_t, and Z_t, in which cuboid located at the intersection of the volumes defined by the three intersection, as shown in Fig. 4.3. Z_t is defined as,

$$Z_t = [z_{min}, z_{max}] \tag{4.6}$$

where z_{min} and z_{max} are the minimum and maximum z-coordinates of the triangle vertices. The intersections between the ray and all the lines bounded by Z_t is given by,

$$T_z = \frac{Z_t - s_z}{d_z} = \frac{[z_{min}, z_{max}] - s_z}{d_z} \tag{4.7}$$

The rejection test will be extended such that,

$$T_x \cap T_y \cap T_z = \phi \tag{4.8}$$

is the new rejection condition.

As mentioned before, it is possible to calculate T_x, T_y, and T_z in parallel, since there are no dependences between them. If the calculations are performed sequentially then the intersection with the first two dimensions (say T_x and T_y)

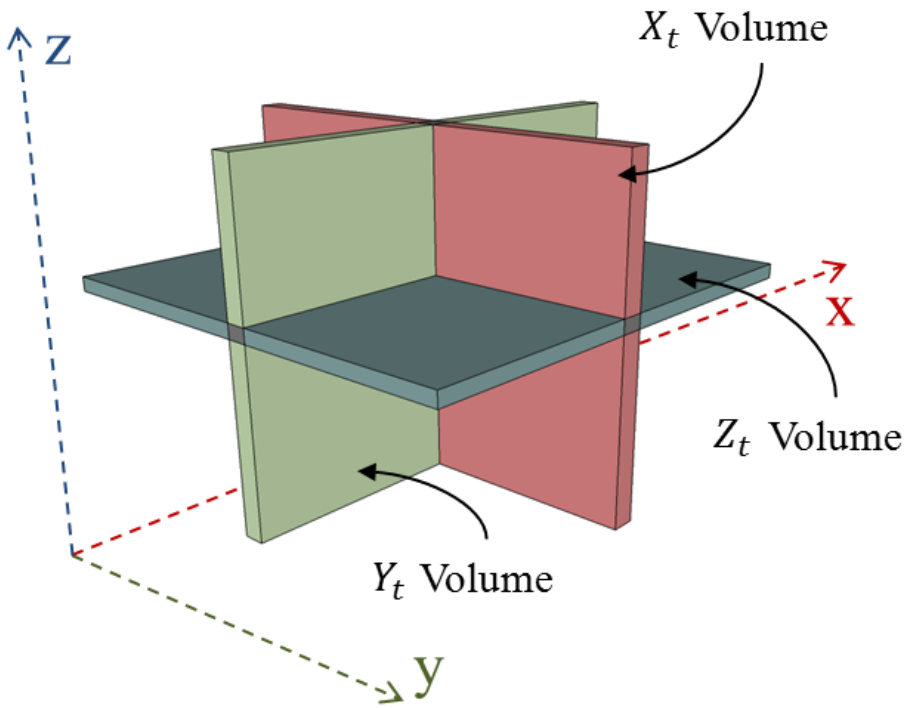


Figure 4.3: The tested triangle bounded by the cuboid which is defined by the intersection of the three volumes X_t, Y_t and Z_t.

should be checked first before calculating the third (T_z). The rejection test based on the two dimensional projection (with only two intervals) yields the majority of the rejection percentage as shown in our experimental results in the next chapter. Hence, the extension of the rejection test to the third dimension is an option for higher performance systems when enough resources are available.

If a potential intersection is detected $T_{intersection} = T_x \cap T_y \cap T_z \neq \phi$, the intersection interval is compared with the previous 'nearest to the eye' intersection (t_r). If the lower bound of the intersection interval is greater than the nearest intersection such that,

$$(T_{intersection}) > t_r \tag{4.9}$$

then the triangle is rejected.

4.2.3 Step Three: Applying any of the conventional tests for the non-rejected triangles

The few triangles that pass through all these tests without rejection get tested by a conventional intersection test, mentioned in section 3.4. Only the nearest to the eye intersection is kept for further pixel calculations.

Any future improvement in the ray-triangle intersection algorithm will improved its combination with the rejection test. Since the rejection test is a general test which can be combined with any ray-triangle intersection algorithm.

Number of required arithmetic operations:

Table 4.1 shows the number of required interval operation by the rejection algorithm steps.

Table 4.1: The number of the interval operations required for rejection algorithm.

	Add	Mul	Div
Step 1	2	0	2
Step 2	1	0	1

In case of absence of interval arithmetic support by the hardware, each interval operation is counted as two regular floating-point operations. In this case the floating-point operations can be reordered to reduce the number of expensive divisions, by calculating the value of $1/d_{x,y,z}$ and then multiply it two times. Table 4.2 shows the number of required floating point operations after reordering.

Table 4.2: The number of the floating-point operations required for rejection algorithm.

	Add	Mul	Div
Step 1	4	4	2
Step 2	2	2	1

The number of the arithmetic operations is a measure for the required hardware, and not a sufficient index of the algorithm speed. The main index of the algorithm speed is the statistics of the ratio of the triangles rejected by each step as shown in Chapter 6.

4.3 Conclusion

In this chapter we introduce an interval method for speeding-up the ray-triangle intersection process in ray-tracing triangle-mesh surfaces. This method is an interval rejection test used for the fast rejection of triangles not containing a valid intersection. The few triangles that are not rejected by our method are then tested using any of the conventional tests.

We use interval analysis methods to yield a fast rejection test for the ray-triangle intersection. The main idea behind the test is to make a quick decision on most of the triangles to reject the majority that do not intersect the ray. And those

few triangles that pass from the rejection test are tested using any conventional ray-triangle intersection algorithms.

In the algorithm, initially the bounding rectangle containing the triangle's projection is checked. If a final decision to reject the triangle is not reached then the bounding cuboid containing the triangle is checked. If the rectangle and the cuboid are not rejected, any of the common intersection-tests is used for checking the triangle for intersection.

Chapter 5

Multi-Cores Ray-Tracing Architecture

5.1 Introduction

In this chapter a new ray-tracing multi-cores heterogeneous architecture is proposed, towards a real-time ray-tracing system. As discussed before, the ray-triangle intersection is a heavy computing process, which requires many floating-point operations. Native hardware implementation of such algorithms is not efficient in area, delay or even power consumption. Also many of the speeding-up techniques are not highly scalable to be implemented in a parallel scheme on hardware.

The proposed architecture is designed in the same context with GPU CUDA architecture, in which the computing unit is made from an array multiprocessors, and also the same synchronization and scheduling techniques as mentioned later (for more about CUDA architecture please refer to Appendix C). But on the other hand each multiprocessor is a heterogeneous design, compared with the homogeneous GPU multiprocessor. The heterogeneous design is based on the interval rejection test discussed in Section 4.2.

According to our experimental results, most of the triangles are rejected using the interval rejection test, and real intersection test is only needed for a small number of triangles. An even smaller number of triangles completely pass the intersection test. Between the very small numbers of triangles which pass the intersection test, only one triangle will be used for calculating the final value of the pixel. And so, the fast and simple rejection test is executed most of the time, the complex intersection test is executed very few times, and the more complex final pixel calculation is executed only one time per pixel.

Based on the mentioned ray-tracing facts observed clearly from experimental

results, and from the previous works, we propose a multiprocessor heterogeneous architecture. The proposed architecture consists of a high density of interval rejectors per multiprocessor, which require the smallest area. Also each multiprocessor contains a few intersectors and one unit for final pixel calculations.

Although the rejection test contains several subtraction and division operations, but the boundary accuracy is very low in significance compared to the interval width in most of the cases. And so, the interval rejector can be built using a narrow width fixed-point addition and division calculations. This reduces the area of the rejector significantly and enables a higher density of rejection units per multiprocessor. Even the few triangles which will pass the test due to the less accurate interval edges will be cached by the intersection test. The out rounding mode for intervals is used to ensure not rejecting triangles with a valid intersection.

Also based to our experimental results the second phase of the rejection test has a low contribution in the speeding-up achieved. So, only the first phase of rejection is implemented in hardware.

In the same context of the GPU architecture design, synchronization only is needed between the members of the same multiprocessor, and no synchronization needed between the different multiprocessors. This limited synchronization simplifies the scheduling process very much.

Any of the space sub-division techniques which are not not highly scalable, as BSP or Octrees can be combined with architecture in the software layer. Also beam and coherent tracing can be used.

Fig. 5.1 shows the proposed multi-cores ray-tracing architecture. The figure shows that the architecture consists of an array of multiprocessors and an on-chip memory on the main device chip. Also an off-chip memory is required storing the bulk data. Each multiprocessor consists of many interval rejectors, few intersection validation units, and one pixel calculation unit.

5.2 Interval Rejector

The interval rejector is the core and the most usable unit in the proposed multi-core architecture. So, the rejector requires the most efficient design compared to other blocks.

A 16-bits fixed point is used for representing numbers used within the rejector. Since, as mentioned before, the accuracy of the interval bounds is very low in significance compared to the interval width. On the other hand using fixed-point number representations increases the speed and reduces the area significantly and enables a higher rejector cores density.

Step two of the algorithm is not implemented in the hardware. Since, the soft-

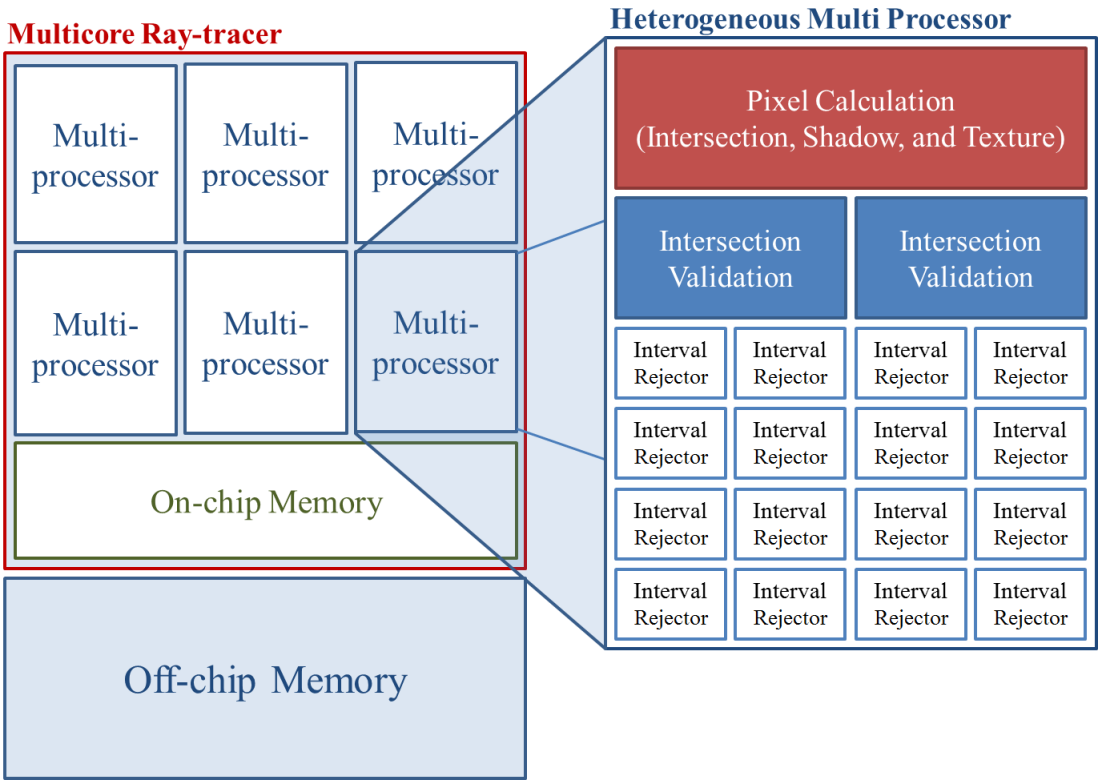


Figure 5.1: Proposed multi-cores ray-tracing architecture.

ware experimental results show that this step contributes with a very low increment in the performance (less than 0.025 of the speedup), while its implementation increases the area significantly, as shown in the next chapter.

Fig. 5.2 shows the main block diagram of the rejector circuit. The circuit contains two identical units for calculating the parameter range T_x and T_y. The circuit also contains an interval intersector unit. Each of the parameter range circuits and the interval intersectors have a valid output signal. The output is valid if the three output signals are true (AND), as discussed below. Triangles proceed to the intersection validation unit if the final output is true.

The parameter circuit is the realization of the equations (4.3) and (4.4), as shown in Fig. 5.3. The used fixed point adder and divider units are not area consuming compared to floating-point numbers.

In case the two limits of the interval are negative, the interval have to be reordered. But this step is not required concerning the logical meaning of the output. If both of the interval limits are negative, then if there is a valid intersection will be negative, and so the final t will be negative also, which will be rejected. And in case of one negative limit, it will be reset to zero, since only positive t interval is the region of interest. Considering the validation of the output with two negative limits, resetting will generate the same final valid output as if negative limit is used, but with much simpler interval intersector circuit. Finally if the two limits

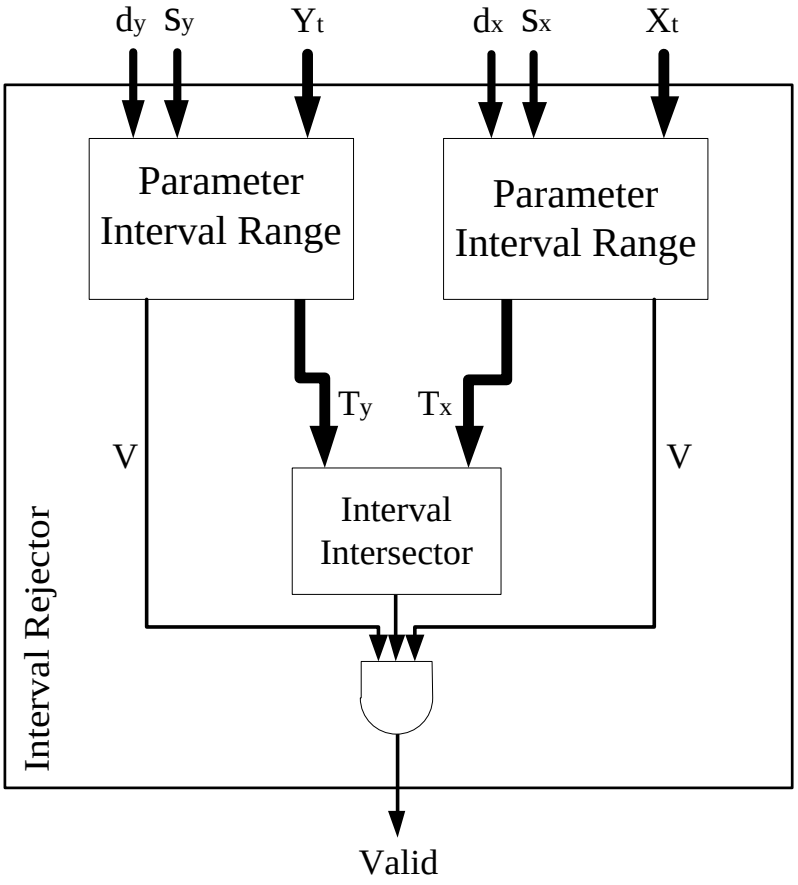


Figure 5.2: The interval rejector block diagram.

are positive reordering may be required in case of negative d and also negative subtraction output.

The interval intersector is shown in Fig. 5.4. There is a valid intersection between the two intervals T_x and T_y if,

1. $\overline{T}_x \geq \underline{T}_Y$, and

2. $\overline{T}_y \geq \underline{T}_x$,

where $\overline{T}$ is upper limit of the interval T, and $\underline{T}$ is the lower one. The intersector circuit realization is an anding between the outputs of two comparators. Look-ahead design is used within the comparator for fast decision output.

The main rejector circuit can be designed in a pipelined scheme in which the first stage is the interval parameter calculation and the second pipeline stage is the interval intersector. The pipelining implementation is a final design decision depending on the whole architecture required performance and on-chip available area.

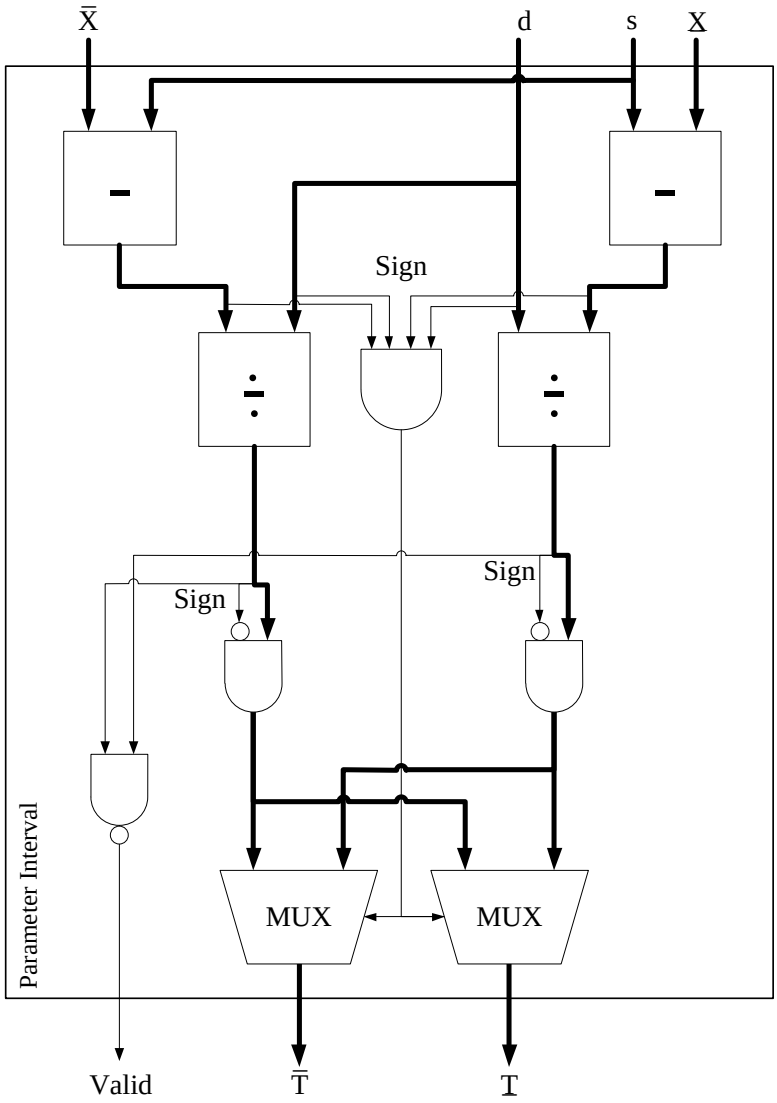


Figure 5.3: Parameter interval calculation circuit block diagram.

5.3 Conclusion

In this chapter a new ray-tracing multi-cores heterogeneous architecture is proposed, towards a real-time ray-tracing system. The proposed architecture is designed in the same context with GPU CUDA architecture.

According to our experimental results, most of the triangles are rejected using the interval rejection test, and real intersection test is only needed for a small number of triangles. An even smaller number of triangles completely pass the intersection test. Between the very small numbers of triangles which pass the intersection test, only one triangle will be used for calculating the final value of the pixel. And so, the fast and simple rejection test is executed most of the time, the complex intersection test is executed very few times, and the more complex

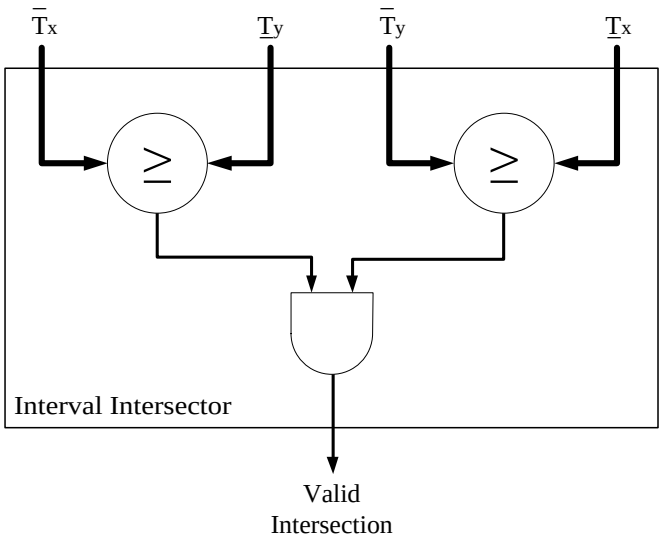


Figure 5.4: Interval intersector circuit.

final pixel calculation is executed only one time per pixel.

Based on the mentioned ray-tracing facts observed clearly from experimental results, and from the previous works, we propose a multiprocessor heterogeneous architecture. The proposed architecture consists of a high density of interval rejectors per multiprocessor, which require the smallest area. Also each multiprocessor contains a few intersectors and one unit for final pixel calculations.

Chapter 6

Experimental Results

6.1 Test Methodology

The system test methodology is constructed of two main parts,

- software test platform, and
- hardware circuit validation.

The software platform is built for the algorithm validation and the statistical results. The statistical results are based on testing millions of triangles as a test set. Initially, a software platform was coded to test the proposed ideas. The intention was to get speed-up from the highly parallel heterogeneous architecture proposed. However, even on th software level,the proposed algorithm achieved a significant speed-up as shown in the following experimental results.

The second half of this chapter is for the testing and validating the real hardware implementation. The hardware test set is designed for circuit validation, by testing wide range of input variation, and not for statistical results. Also area comparison is made between the interval rejector implementation and Möller algorithm implementation, showing a significant gain in area in case of the rejector implementation.

6.2 Software Results

We built two test platforms for testing the interval rejection algorithms using C# and C++. The C# version of the test platform is provided in Appendix D.

Both programs were compiled on the Microsoft Visual Studio complier, 2008 version. For ease of debugging and testing purposes, our code is written as a single-thread program. The code is compiled using the compile for speed compiler

option. An Intel Core 2 Duo 3 GHz processor machine is used to test our program (The Core 2 processor does not have interval arithmetic support).

To test our algorithm, 3D models in PLY (Polygon File Format also known as the Stanford's Triangle Format) format are used. 3D models are from Stanford University (3D Scanning Repository) [53], Georgia Institute of Technology (Large Geometry Models Archive) [59], and University of North Carolina at Chapel Hill (GAMMA Project - Power Plant Model) [60]. The power plant model is a complete model of an actual coal fired power plant. The model consists of 12,748,510 triangles [60]. Any other 3D models in PLY format can be used. Part of the test platform is written to grab the data from the PLY files. For testing using 3D models other than the PLY models, any free 3D format converter can be used. The testing models are selected to cover wide range of number of triangle per model, starting from 3,736 to 1,765,388, and a total 10,259,042 triangles. Also models are selected to be from different categories of shaped.

The performance of the ray-intersection process in the original Möller's algorithm and the Möller's algorithm with the interval rejection test added to it are presented below. Möller algorithm is used since it is the most common ray-triangle intersection algorithm. However, the interval rejection test can be added on any ray-triangle algorithm.

Since the rejection algorithm is concerned with the intersection process, only the ray-triangle intersection performances are compared. But these comparisons reflect the whole system performance, since the intersection process consumes most of tracing time, as mentioned before in Chapter 3. All the given times and the performance matrices are for intersecting the main rays with all the 3D-Model triangles. Fig. 6.1 shows an example of main stage of ray-tracing, using the main rays only. The figure is an output of the test platform and it shows the ghost of the Stanford's bunny 3D model ray-traced using the main rays intersected with its triangles.

To check the correctness of our implementation, we made a special test program to mark the triangles that would be rejected by our interval test and to check whether they are rejected or not with the normal Möller intersection test. Our implementation performs correctly for all the models, without a single error in any triangle. These validity tests enable the conversion of the algorithm to be implemented in hardware.

According to our experimental results, more than 99.9% of the triangles are rejected using the interval test, depending on the model used. The rejection percentage of the triangles may also vary slightly depending on the position of the camera. Which shows that 99.9% of the triangles are tested only using the fast and area efficient rejection test, while a very few number of triangles are

Figure 6.1: The ghost of the Stanford's bunny is the result of ray-tracing using the main rays only.

needed to be tested using any of the complex ray-triangle intersection algorithms (Möller algorithm is used in our test platform). This output enables the idea of the heterogeneous architecture introduced in Chapter 5, in which the multiprocessor consists of a high density of the light weight rejector cores, and few of the complex intersection validation cores.

Although the main advantage of the rejection test is its light weight and highly parallel scheme, Table 6.1 shows that we gain a speedup between 1.25 and 2.21 after applying the interval rejection test to Möller's algorithm. The speedup depends on the number of triangles and their distribution in the space. The table shows the execution times per pixel and per pixel and triangle for both of the native Möller algorithm, and the interval rejector test added to Möller algorithm. The gained speed-up in the software layer is much less than the expected speed up of the hardware implementation, since in the proposed architecture each Möller intersector is combined with many of the interval rejectors. In the software implementation each Möller intersector is combined with only one rejector.

The experimental results show that the second phase of the rejection test (the extension to the third-dimension) only contributes less than 0.025 of the speedup. Hence, if in a certain implementation, the designer wishes to reduce the amount of hardware or software used for our proposal, the third-dimension phase may be eliminated without a significant loss is the speedup.

Fig. 6.2 shows the execution time per pixel in milliseconds for both Möller algorithm and Möller with the rejection test added to it, for the test set mentioned before. This figure shows the gained execution time reduction using the rejection test. While Fig. 6.3 Shows the speed up gained by the rejection test. The speed up ranges from 1.25 to 2.21, depending on the tested model and the number of triangles.

Table 6.1: The intersection time in milliseconds per pixel and in nanoseconds per pixel·triangle are given for Möller's algorithm and Möller with the rejection-test added to it for each model in our test set. The last column shows the speedup gained.

Model	Number of Triangles	Möller Only		Möller + Rejection Test		Speedup
		$[ms/pixel]$	$[ns/pixel \cdot triangle]$	$[ms/pixel]$	$[ns/pixel \cdot triangle]$	
Power Plant (S12 - B - G2)	3 736	0.08	21.41	0.065	17.40	1.25
Power Plant (S06 - B - G0)	4 880	0.10	20.49	0.079	16.19	1.27
Bone 0	7 990	0.196	24.53	0.138	17.31	1.42
Power Plant (S08 - B - G5)	18 762	0.39	20.79	0.31	16.52	1.26
Bone 7	29 331	0.77	26.26	0.48	16.26	1.62
Stanford Bunny	69 451	1.58	22.75	1.26	18.14	1.25
Power Plant (S03 - A - G2)	78 030	1.67	21.40	1.30	16.66	1.28
Bone 1	81 656	2.22	27.17	1.45	17.71	1.53
Bone 3	86 210	2.24	25.98	1.41	16.36	1.59
Bone 5	88 497	2.48	28.02	1.52	17.17	1.63
Horse	96 966	2.31	23.82	1.68	17.33	1.38
Bone 8	124 018	3.26	26.29	2.12	17.09	1.54
Power Plant (S15 - D - G0)	156 224	3.38	21.64	2.61	16.71	1.30
Bone 2	355 511	10.33	29.06	5.87	16.51	1.76
Bone 4	522 567	19.23	36.80	8.71	16.67	2.21
Power Plant (S20 - A - G0)	615 156	15.57	25.31	10.55	17.15	1.48
Skeleton Hand	654 666	21.46	32.78	11.03	16.85	1.95
Power Plant (S01 - A - G0)	747 948	19.04	25.46	12.65	16.91	1.51
Power Plant (S19 - B - G0)	814 424	20.60	25.29	13.87	17.03	1.49
Power Plant (S01 - E - G0)	841 756	27.01	32.09	14.44	17.15	1.87
Dragon	871 414	23.19	26.61	15.08	17.31	1.54
Happy Buddha	1 087 716	29.93	27.52	19.15	17.61	1.56
Bone 6	1 136 745	34.08	29.98	19.33	17.00	1.76
Turbine Blade	1 765 388	50.18	28.42	29.56	16.74	1.70

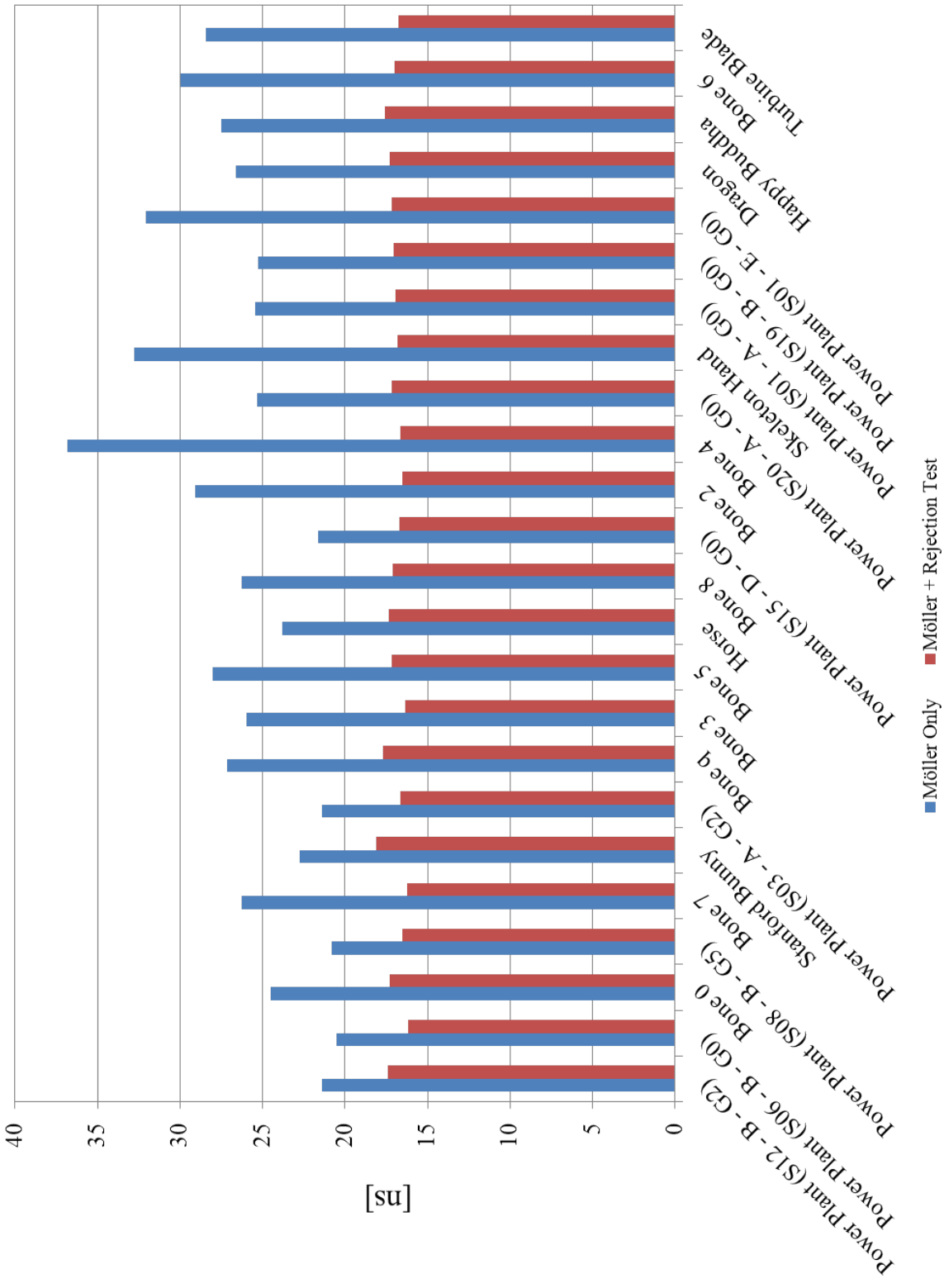


Figure 6.2: Graph of the intersection time in milliseconds per pixel for Möller's algorithm and Möller with the rejection-test added to it for each model in the test set.

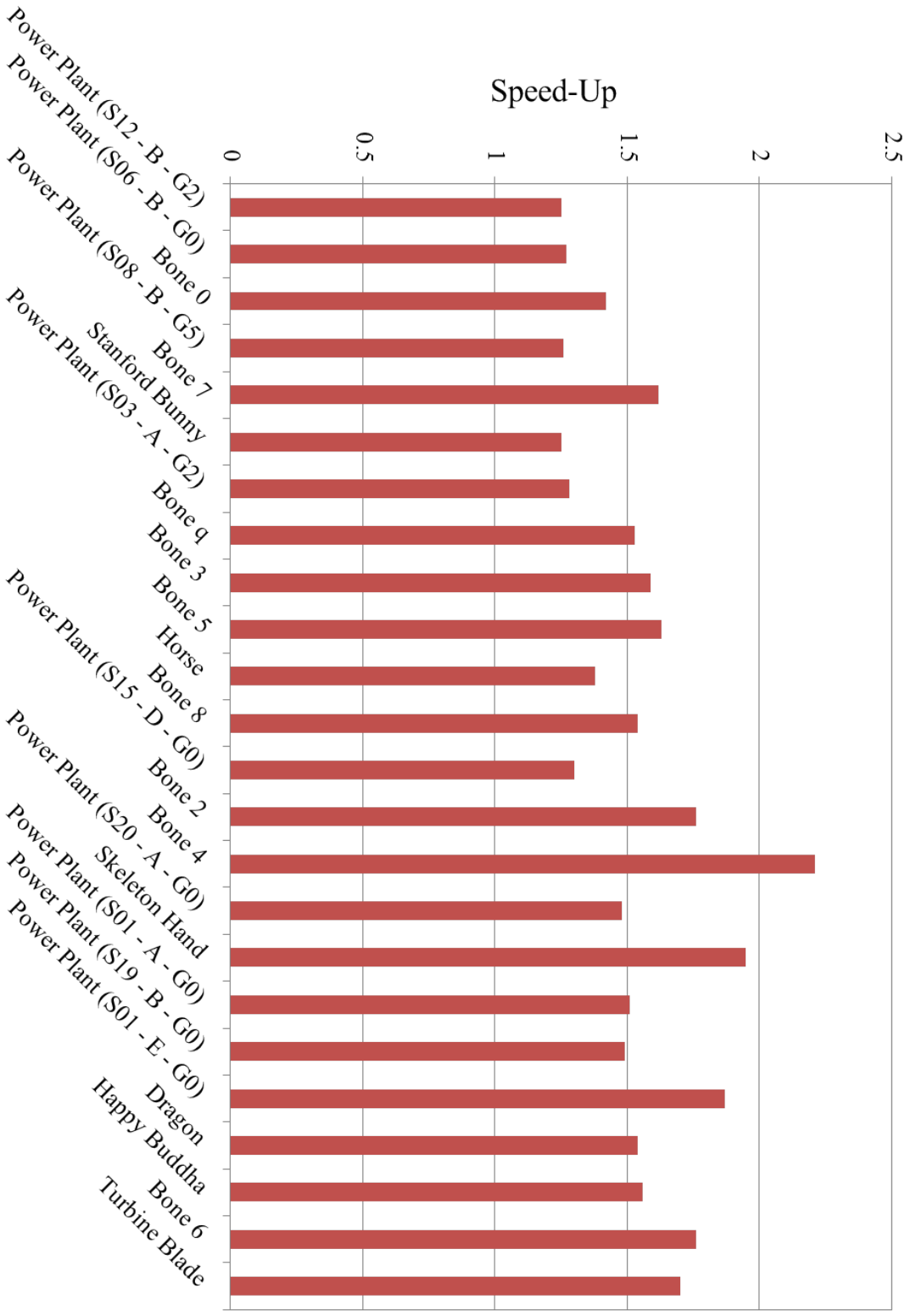


Figure 6.3: Graph of the speed-up gained by applying the interval rejection test added to the Möller's algorithm compared to the native Möller's algorithm.

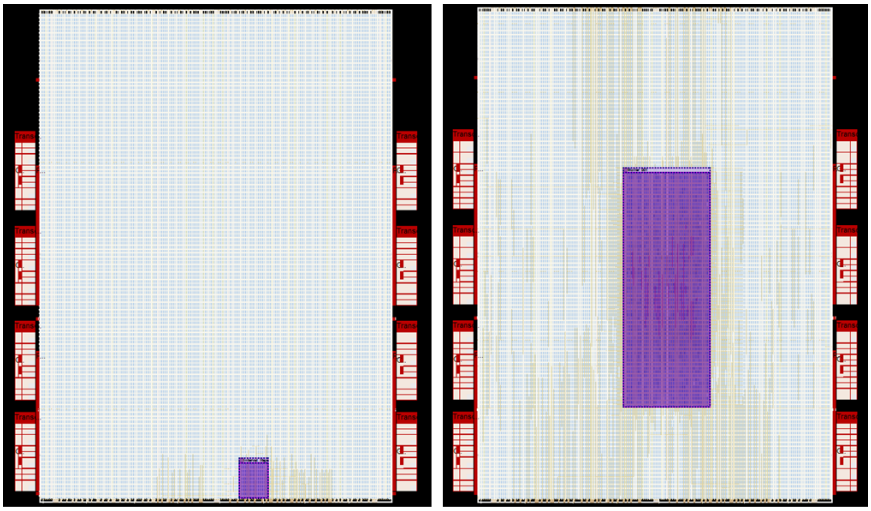
Figure 6.4: The normalized intersection time per pixel·triangle versus the number of triangles.

Fig. 6.4 shows a graph for the normalized intersection time per pixel·triangle versus the number of triangles. The normalized execution time per triangle shows a large variation for the native Möller algorithm (between 20.49 and 36.80 nanoseconds), while a very small variations is expressed for the added on rejection test (between 16.19 and 18.14 nanoseconds). This property is very valuable concerning real-time rendering with multiple frames per seconds. Since a variable execution time causes a jitter like phenomena in time. The removal of such jitter like phenomena is very desirable in real time video rendering for frames of various complexity.

6.3 Hardware Results

The interval rejector hardware and the Möller's intersector unit are described using Verilog hardware description language, in structural techniques. The interval rejector is built as a completely combinational and parallel unit. On the other hand pipelining is used in implementing the Möller intersection algorithm due the large number of floating-point operations it contains.

As mentioned before the interval analysis (rejection test) does not require high precision of calculations, bounds precision compared to the interval width in interval analysis can be neglected, and may reduce the rejection rate but without any errors. On the other hand interval analysis (rejection test) have to be very fast and area efficient. But the final refinement technique (Möller's intersection test) has to be very accurate. So, 16-bit fixed-point are selected for the rejector



(a) (b)

Figure 6.5: Altera Stratix IV - EP4SGX530NF45C4ES floor-plan showing the occupied areas in purple while the empty areas are in light colors. (a) Interval rejector implementation. (b) Möller intersection algorithm implementation.

implementation, while 32-bit floating-point are selected for the final intersector implementation.

Synthesis and simulations are made using Altera Quartus II v9.1 build 222. Synthesis are made using synthesis for optimized area option. Table 6.2 shows the occupied area in number of units and percent for the interval rejector and Möller intersection algorithm implementations due to the Altera Quartus II results. The occupied area ratio between the interval rejector and Möller implementations is also given. The area is calculated for the Altera Stratix IV FPGA EP4SGX530NF45C4ES device.

The interval rejector implementation express a delay of 38.46 [ns] using the static timing analysis for delay estimation. on the other hand the Möller requires a total delay of 153.77 [ns], and a stage delay of 30.75 [ns].

Fig. 6.5 shows the floor-plan of each of the interval rejector and the Möller in-

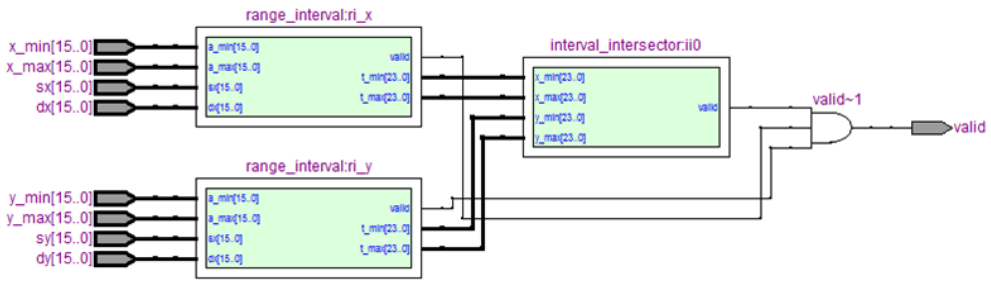


Figure 6.6: Interval rejector block diagram as generated from the Verilog descriptions using Altera Quartus II RTL viewer.

Table 6.2: Altera Stratix IV - EP4SGX530NF45C4ES occupied area in number of units and percent for the interval rejector and Möller intersection algorithm implementations. The occupied area ratio between the interval rejector and Möller implementations is also given at the last column.

	Interval Rejector (Non-Pipelined)	Möller Intersection (Pipelined)	Interval Area/Möller Area (%)
Combinational ALUTs (424,960)	2,663 (0.626%)	27,377 (6.44%)	9.72%
Memory ALUTs (212,480)	0 (0%)	1,047 (0.493%)	0%
Dedicated Registers (424,960)	0 (0%)	31,719 (7.46%)	0%
DSP Bloks (1,024)	0 (0%)	16 (1.56%)	0%
Total (%)	~0.25%	~5.66%	~4.42%

tersection algorithm implementations. The figure shows a clear difference between the percentages of the occupied areas in both cases. Möller intersection algorithm implementation requires over twenty times of the area of the used interval rejector. In case of completely parallel non-pipelined implementation for Möller algorithm, more area will be required.

Fig. 6.6 and 6.7 shows block diagram as generated from the Verilog descriptions using Altera Quartus II RTL viewer.

As mentioned before the hardware test set is designed for circuit validation, by testing wide range of input variation, and not for statistical results. Table 6.3 shows a subset of the test set of the interval rejector circuit, showing the source and the direction and the triangle projection bounding rectangle values as inputs in hexadecimal and the valid bit as an output. The table shows the circuit validation test, while the statistical data are shown in the software layer results as mentioned before.

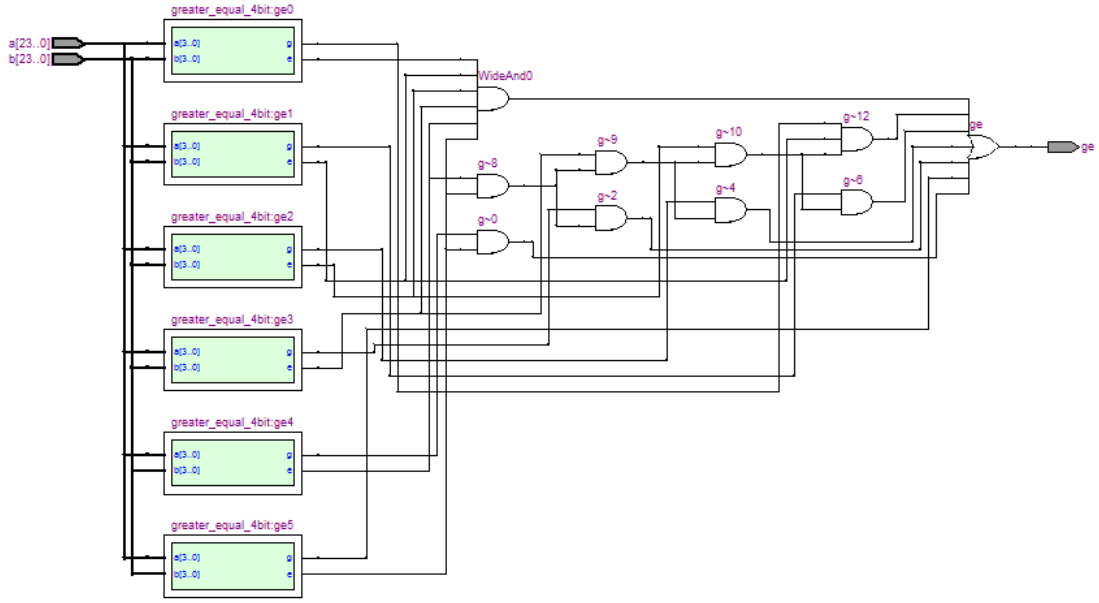


Figure 6.7: Interval intersector block diagram as generated from the Verilog descriptions using Altera Quartus II RTL viewer.

6.4 Conclusion

This chapter shows the experimental results, the hardware realization, and the results validation. It shows that software platform is built for the algorithm validation and the statistical results.

According to the experimental results using a standard test containing millions of triangle, more than 99.9% of the triangles are rejected using the interval

Table 6.3: Subset of the test set of the interval rejector circuit, showing the source and the direction and the triangle projection bounding rectangle values as inputs in hexadecimal and the valid bit as an output.

Source		Direction		Triangle Projection Bounding				T_x		T_y		Output
S_x	S_y	d_x	d_y	x_{max}	x_{min}	y_{max}	y_{min}	max	min	max	min	(Valid bit)
00.00	00.00	01.00	01.00	04.00	02.00	04.00	02.00	0004.00	0002.00	0004.00	0002.00	1
03.0B	00.40	00.20	01.01	15.30	10.FA	33.01	32.AB	0091.40	006F.90	0032.8E	0032.38	0
12.0D	0A.56	FF.F8	00.01	89.8A	70.0B	65.74	64.A0	0000.77	0000.5E	5B1E.00	59B2.00	0
00.01	00.02	02.00	02.01	A8.50	A8.45	A9.12	A7.32	0054.27	0054.22	0054.5D	0053.6E	1
01.2A	02.EC	03.12	00.FA	F0.12	E0.00	50.10	3F.FF	004D.CF	0498.93	004E.FD	003E.8A	1
03.00	12.0F	02.68	05.36	23.23	06.65	77.86	6A.4D	000D.5B	0001.69	0013.78	0010.EF	0
03.00	12.0F	01.12	05.36	23.23	06.65	77.86	6A.4D	001C.5E	0002.FF	0013.78	0010.EF	1
F0.32	40.2A	F1.32	05.56	32.32	12.44	EE.32	43.DA	0001.55	0001.33	0020.9D	0000.BD	1
00.23	42.43	44.50	56.42	34.23	12.44	EE.32	A3.43	0000.C2	0000.43	0001.FE	0001.1F	0
00.11	00.11	00.44	00.34	00.64	00.59	07.65	07.06	0001.38	0001.0F	0024.13	0022.40	0
03.23	01.2F	F0.01	01.00	01.21	00.43	04.00	02.00	0000.2F	0000.21	0002.D1	0000.D1	0
04.02	01.0F	03.24	05.43	06.5D	03.99	0D.DA	0B.FF	0000.C0	0000.00	0002.6E	0002.14	0
34.5	02.37	0A.4F	0A.00	0A.A4	01.F4	01.2F	00.23	0000.B7	0000.00	0000.00	0000.00	0
00.21	00.21	01.00	01.00	43.DA	43.D1	43.33	0042.21	0043.B9	0042.B0	0043.12	0042.00	1
01.00	01.00	00.10	00.10	66.78	66.00	66.49	0661.10	0657.80	0650.00	0654.90	0651.10	1
13.21	B3.44	B0.00	D3.24	04.23	02.21	66.49	66.12	0000.37	0000.30	0001.B9	0001.B8	0
F1.32	B3.44	B0.34	D3.24	02.FF	01.B1	66.49	66.12	0000.00	0000.00	0001.B9	0001.B8	0
EE.32	54.33	0A.A9	04.32	01.12	00.12	0A.EE	00.FD	0001.C5	0001.AD	0000.00	0000.00	0
00.00	00.00	FF.FF	FF.FF	F4.32	F0.00	F4.32	F0.00	1000.00	0BCE.00	1000.00	0BCE.00	1

test, depending on the model used. Also on the software layer implementation, without any level of parallelism applied, we gain a speedup between 1.25 and 2.21 after applying the interval rejection test. The experimental results also show that the rejection test removes a jitter like phenomena in the execution time of the intersection process.

The testing and validating the real hardware implementation is introduced. The hardware test set is designed for circuit validation, by testing wide range of input variation, and not for statistical results. Also area comparison is made between the interval rejector implementation and Möller's algorithm implementation, showing a significant gain in area in case of the rejector implementation, more than 22 times in area compared to the conventional intersection process hardware realization.

Chapter 7

Conclusions and Future Works

7.1 Conclusions

In this thesis we introduce optimizations for the ray-tracing on the algorithmic, the architecture and the implementation layers, by combining interval analysis and arithmetic techniques with the ray-intersection process.

We introduce an interval method for speeding-up the ray-triangle intersection process, in ray-tracing triangle-mesh surfaces, by reducing the data set tested for intersection. This method is a rejection test used for the fast rejection of triangles not containing a valid intersection.

A software test platform is built for validating the algorithm and collecting statistical data. 3D models in PLY (Polygon File Format also known as the Stanford's Triangle Format) format are used. The models are from Stanford University (3D Scanning Repository) [53], Georgia Institute of Technology (Large Geometry Models Archive) [59], and University of North Carolina at Chapel Hill (GAMMA Project - Power Plant Model) [60]. The power plant model is a complete model of an actual coal fired power plant. The model consists of 12,748,510 triangles [60]. The testing models are selected to cover wide range of number of triangle per model, starting from 3,736 to 1,765,388, and a total 10,259,042 triangles. Also models are selected to be from different categories of shaped.

According to the experimental results on the test platform, more than 99.9% of the triangles are rejected using the interval test, depending on the model used. Also on the software layer implementation, without any level of parallelism applied, we gain a speedup between 1.25 and 2.21 after applying the interval rejection test. The experimental results also show that the rejection test removes a jitter like phenomena in the execution time of the intersection process, which it very suitable for real-time video rendering.

Based on the collected statistical data, we introduce new ray-tracing multi-

cores heterogeneous architecture, towards a real-time ray-tracing system. The proposed architecture is a computing unit made from an array of heterogeneous multiprocessors. Each unit consists of a high density of, our newly introduced, interval rejector units. Also each multiprocessor contains a few intersectors and one unit for final pixel calculations. An area optimized interval rejector is also provided. This rejector requires much smaller on-chip area compared to the conventional intersection algorithms realizations.

To the best of our knowledge, our rejector core unit is the first combination between ray-tracing triangle meshes and interval analysis and arithmetic on the hardware layer. The implementation of the interval rejector shows a reduction more than 22 times in area compared to the conventional intersection process hardware realization. Such reduction in area allows a higher density of rejector units, which proportionally leads to higher speeds, paving the way for real-time ray-tracing.

7.2 Future Works

There are a lot of promising ideas that can be proposed based on our current work, statistical results and, hardware and software implementations. Following, some of the future work ideas are introduced,

- Building a complete ray-tracing hardware system using the newly introduced architecture.
- Tuning the proposed architecture for future general propose usages in context with general purpose graphical processing units (GPGPU).
- Extending the rejection algorithm for the other objects representations, implicit and parametric surfaces.
- Using the statistical results for creating a multi-layer rejection test, reducing the required steps in the intersection test.

Bibliography

[1] A. Dietrich, I. Wald, and P. Slusallek, "Large-scale CAD model visualization on a scalable shared-memory architecture," in *Vision, Modeling, and Visualization*, 2005.

[2] I. Wald, "The OpenRT - API," in *International Conference on Computer Graphics and Interactive Techniques*, 2005.

[3] P. Slusallek, P. Shirley, W. Mark, G. Stoll, and I. Wald, *Introduction to real-time ray tracing*, ser. ACM SIGGRAPH 2005 Courses. ACM, 2005.

[4] S.-W. Wang, Z.-C. Shih, and R.-C. Chang, "An efficient and stable ray tracing algorithm for parametric surfaces," *Journal of Information Science and Engineering*, vol. 18, no. 4, pp. 541–561, 2002.

[5] "Intel official blog - real time ray-tracing: The end of rasterization?" [accessed Aug-2009]. [Online]. Available: http://blogs.intel.com/research/2007/10/real_time_raytracing_the_end_o.php

[6] U. W. Kulisch, *Numerical Validation in Current Hardware Architectures*. Springer-Verlag, 2008, ch. Complete Interval Arithmetic and Its Implementation on the Computer, pp. 7–26.

[7] R. E. Moore, R. B. Kearfott, and M. J. Cloud, *Introduction to Interval Analysis*. Society for Industrial and Applied Mathematics, 2009.

[8] U. W. Kulisch, *Advanced Arithmetic for the Digital Computer*. Springer, November 2002.

[9] B. Hayes, "A lucid interval," *American Scientist*, vol. 91, pp. 484–488, 2003.

[10] "P1788: IEEE standard for interval arithmetic," not published yet. [Online]. Available: http://grouper.ieee.org/groups/1788

[11] H. Brönnimann, G. Melquiond, and S. Pion, *A Proposal to add Interval Arithmetic to the C++ Standard Library*, ISO C++ Standardization Std., 2005.

[12] D. P. Mitchell, "Robust ray intersection with interval arithmetic," in *Proceedings on Graphics interface*, 1990, pp. 68 – 74.

[13] O. Abert, M. Geimer, and S. Muller, "Direct and fast ray tracing of NURBS surfaces," in *IEEE Symposium on Interactive Ray Tracing*, 2006, pp. 161–168.

[14] T. Möller and B. Trumbore, "Fast, minimum storage ray-triangle intersection," *Journal of Graphics Tools*, vol. 2, pp. 21 – 28, 1997.

[15] R. J. Segura and F. R. Feito, "An algorithm for detecting intersection segment polygon in 3D," *Computer & Graphics*, vol. 22, pp. 587–592, 1998.

[16] I. Wald, "Realtime ray tracing and interactive global illumination," Ph.D. dissertation, Saarland University, 2004.

[17] I. Wald and P. Slusallek, "State of the art in interactive ray tracing," 2001.

[18] O. Caprani, L. Hvidegaard, M. Mortensen, and T. Schneider, "Robust and efficient ray intersection of implicit surfaces," *Reliable Computing*, vol. 6, pp. 9–21, 2000.

[19] A. Efremov, V. Havran, and H.-P. Seidel, "Robust and numerically stable Bézier clipping method for ray tracing NURBS surfaces," in *Proceedings of the 21st spring conference on Computer graphics*. ACM, 2005, pp. 127 – 135.

[20] D. L. Toth, "On ray tracing parametric surfaces," in *ACM SIGGRAPH Computer Graphics*, 1985, pp. 171 – 179.

[21] J. E. FlŽorez, "Improvements in the ray tracing of implicit surfaces based on interval arithmetic," Ph.D. dissertation, Universitat de Girona, 2008.

[22] E. Hansen, "A globally convergent interval method for computing and bounding real roots," *BIT Numerical Mathematics*, vol. 18, no. 4, pp. 415–424, 1978.

[23] W. Enger, "Interval ray tracing - a divide and conquer strategy for realistic computer graphics," *The Visual Computer*, vol. 9, no. 2, pp. 91–104, 1992.

[24] G. Alefeld, "Eine modification des newtonverfarens zur bestimmung der reellen," *Numeriche Matematik*, vol. 50, pp. 32–33, 1970.

[25] A. Neumaier, "Computer graphics, linear interpolation, and nonstandard intervals," 2009.

[26] T. L. Kay and J. T. Kajiya, "Ray tracing complex scenes," in *Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, 1986, pp. 269 – 278.

[27] J. Schmittler, I. Wald, and P. Slusallek, "SaarCOR: a hardware architecture for ray tracing," in *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, 2002, pp. 27 – 36.

[28] S. Woop, J. Schmittler, and P. Slusallek, "RPU: A programmable ray processing unit for realtime ray tracing," in *ACM Trans. Graph*, 2005, pp. 434–444.

[29] M. Shevtsov, A. Soupikov, and A. Kapustin, "Ray-triangle intersection algorithm for modern CPU architectures," in *Proceedings of GraphiCon*, 2007.

[30] A. Knoll, Y. Hijazi, C. Hansen, I. Wald, and H. Hagen, "Interactive ray tracing of arbitrary implicits with simd interval arithmetic," in *IEEE/Eurographics Symposium on Interactive Ray Tracing*. IEEE Computer Society, 2007, pp. 11–18.

[31] A. Knoll, Y. Hijazi, A. Kensler, M. Schott, C. Hansen, and H. Hagen, "Fast and robust ray tracing of general implicits on the GPU," Scientitfic Computing and Imaging Institute, University of Utah, Tech. Rep., 2007.

[32] "Povray," [accessed Feb-2010]. [Online]. Available: http://www.povray.org/

[33] "Meridian rendering," [accessed Feb-2010]. [Online]. Available: http://www.sunfishstudio.com/

[34] "Yafa ray project," [accessed Oct-2009]. [Online]. Available: http://www.yafaray.org/

[35] T. Sunaga, "Theory of interval algebra and its application to numerical analysis," *Research Association of Applied Geometry (RAAG) Memoirs*, vol. 2, pp. 29–46, 1958.

[36] R. E. Moor, *Interval Analysis*. Prentice-Hall, 1966.

[37] J. Armengol and M. A. Sainz, "Generation of error-bounded envelopes by means of modal interval analysis," 1999.

[38] E. Hansen and S. Sengupta, "Bounding solutions of systems of equations using interval analysis," *BIT Numerical Mathematics*, vol. 21, no. 2, pp. 203–211, 2005.

[39] R. J. Bhiwani and B. M. Patre, "Solving first order fuzzy equations: A modal interval approach," *Emerging Trends in Engineering & Technology, International Conference on*, vol. 0, pp. 953–956, 1999.

[40] E. R. Hansen and G. W. Walster, *Global optimization using interval analysis*, 2nd ed. CRC Press, 2004.

[41] W. Edmonson, R. Gupte, S. Ocloo, J. Gianch, and W. Alex, "Interval arithmetic logic unit for signal processing and control applications," 2006.

[42] G. Melquiond and C. Munoz, "Guaranteed proofs using interval arithmetic," in *Proceedings of the 17th IEEE Symposium on Computer Arithmetic*, 2005, pp. 188 – 195.

[43] V. Kreinovich, F. Modave, S. Starks, and G. Xiang, "Towards real world applications: Interval-related talks at NAFIPS'05," *Reliable Computing*, vol. 12, no. 1, pp. 73–77, 2006.

[44] R. Kirchner and U. W. Kulisch, "Hardware support for interval arithmetic," *Reliable Computing*, vol. 12, pp. 225–237, 2006.

[45] E. Kaucher, "Interval analysis in the extended interval space IR," *Computing Supplement*, vol. 2, pp. 33–49, 1980.

[46] E. Gardeñes, M. A. Sainz, L. Jorba, R. Calm, R. Estela, H. Mielgo, and A. Trepat, "Model intervals," *Reliable Computing*, vol. 7, pp. 77–111, 2001.

[47] E. Gardeñes, H. Mielgo, and A. Trepat, "Modal intervals: reason and ground semantics," in *International Symposium on interval mathematics*, 1986, pp. 27 – 35.

[48] N. T. Hayes, "Introduction to modal intervals," in *The Working Group of the IEEE 1788*, 2009, p. 61.

[49] P. H. Christensen, J. Fong, D. M. Laur, and D. Batali, "Ray tracing for the movie "cars"," in *IEEE Symposium on Interactive Ray Tracing*, 2006.

[50] L. P. Kobbelt, K. Daubert, and H. p. Seidel, "Ray tracing of subdivision surfaces," in *In Rendering Techniques 1998 - Proceedings of the Eurographics Workshop*. Springer-Verlag, 1998, pp. 69–80.

[51] "Ray-tracing Quake game on Intel's machines," [accessed Nov-2009]. [Online]. Available: http://www.idfun.de/temp/q4rt/

[52] G. E. Farin, *Curves and Surfaces for CAGD*, fifth edition ed., ser. The Morgan Kaufmann Series in Computer Graphics. Morgan Kaufmann, 2002.

[53] "The stanford 3D scanning repository," [accessed Dec-2009]. [Online]. Available: http://www-graphics.stanford.edu/data/3Dscanrep/

[54] J. M. Snyder and A. H. Barr, "Ray tracing complex models containing surface tessellations," in *ACM SIGGRAPH Computer Graphics*, vol. 21, 1987, pp. 119 – 128.

[55] D. Badouel, *Graphics gems*. Academic Press Professional, Inc, 1990, ch. An efficient ray-polygon intersection, pp. 390 – 393.

[56] J. Segura and F. R. Feito, "Algorithms to test ray-triangle intersection," in *Journal of WSCG*, 2001.

[57] S. J. Teller, "Computing the antipenumbra of an area light source," in *ACM SIGGRAPH Computer Graphics*, vol. 26, 1992, pp. 139 – 148.

[58] W. Barth and W. Stürzlinger, "Efficient ray tracing for Bezier and B-spline surfaces," *Computers & Graphics*, vol. 17, no. 4, pp. 423 – 430, 1993. [Online]. Available: http://www.sciencedirect.com/science/article/B6TYG-48TMS93-63/2/ca5f097d718f367b8be6f65ec7285542

[59] "Georgia institute of technology - large geometry models archive," [accessed Dec-2009]. [Online]. Available: http://www.cc.gatech.edu/projects/large_models/

[60] "University of north carolina at chapel hill - GAMMA project - power plant model," [accessed Dec-2009]. [Online]. Available: http://gamma.cs.unc.edu/powerplant/

[61] M. T. D. Dinh, "GPUs - graphics processing units," Institute of Computer Science, University of Innsbruck, Tech. Rep., 2008.

[62] P. Shirley, K. Sung, E. Brunvand, A. D. S. Parker, and S. Boulos, "Rethinking graphics and gaming courses because of fast ray tracing," in *International Conference on Computer Graphics and Interactive Techniques*, 2007.

[63] "OpenCL - the open standard for parallel programming of heterogeneous systems," 2009. [Online]. Available: http://www.khronos.org/opencl/

[64] H. Nguyen, *GPU Gems 3*. Addison-Wesley Professional, 2007.

[65] M. Y. Siu, "A high-performance area-efficient multifunction interpolator," in *17th IEEE Symposium on Computer Arithmetic*, 2005, pp. 272 – 279.

[66] nVidia, "OpenCL programming guide for the cuda architecture," p. 23, 2009.

Appendix A

Interval Newton Method

A.1 Classical Newton Method

The classical Newton method is a famous method used for calculating the root (zeros) of nonlinear functions, with a level of approximation.

$$f(x) = 0$$

The idea behind the Newton method is finding the root of the function's tangent at an initial point and use the result as a new initial point iteratively till the required precision is reached. The tangent function can be written as,

$$t(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) \tag{A.1}$$

where $t(x)$ is the tangent to the function $f(x)$. The iteration can be written as,

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)},\ i = 0, 1, 2, ... \tag{A.2}$$

where x_o is the initial search point, x_{i+1} is a better approximation for the root than x_i, in case of convergence, and i is the iteration counter.

According to [8] it is well-known that if $f(x)$ has a single zero x_z, $f(x)$ is twice continuously differentiable and x_o is close enough to x_z, then the method will converge quadratically.

A.2 Interval Extension for the Newton Method

The interval Newton method was originally introduced by R. Moore in [36], as an interval extension for the classical Newton method. Interval Newton is believed to be much more powerful than the classical one. In contrast with the classical method, the interval Newton method never diverges [8].

The new iteration is defined as,

$$X_{i+1} = \left(m\left(X_i\right) - \frac{f\left(m\left(X_i\right)\right)}{F'\left(X_i\right)} \right) \cap X_i, \qquad i = 0, 1, 2, ... \tag{A.3}$$

where X_o is the initial search interval, $m\left(X\right)$ is the midpoint of the interval X, $F'\left(X\right)$ is the interval evaluation of $f'\left(x\right)$, and i is the iteration counter. Any point $x \in X_i$ can be used instead of the midpoint. The interval Newton can be applied only if,

$$0 \notin F'\left(X_o\right)$$

which guarantees the existence of only one zero in the initial selected interval.

The expression,

$$N\left(X\right) = x - \frac{f\left(x\right)}{F'\left(X\right)} \tag{A.4}$$

is defined as the interval Newton operator.

- If $N\left(X\right) \cap X = \phi$, then there is no zeros in the tested interval.
- $N\left(X\right) \subseteq X$, then $f\left(x\right)$ has only one zero $x_z \in X$.

Appendix B

Rasterization Steps

B.1 Geometry Stage

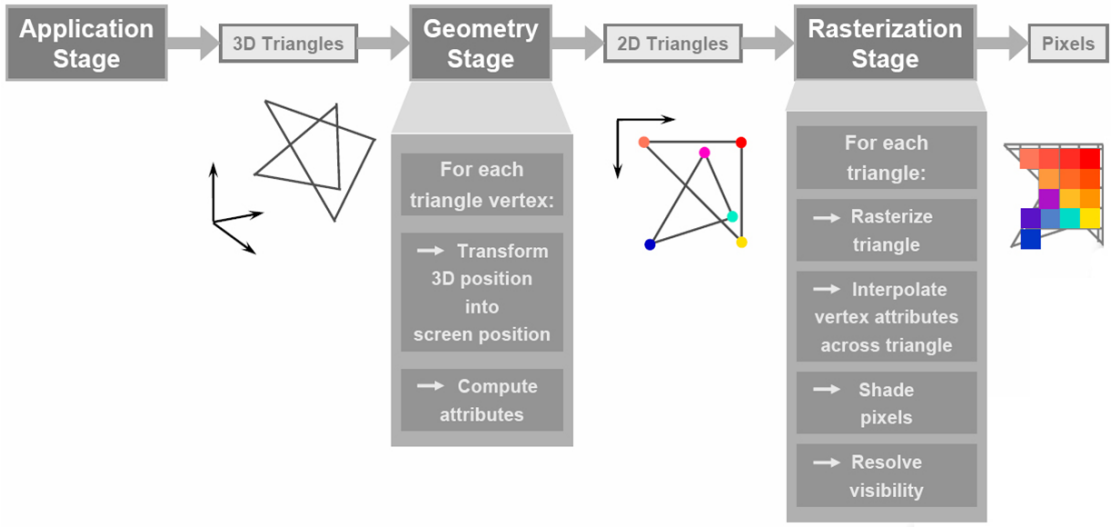


Figure B.1: Rasterization steps [61].

Geometry stage is the first stage of the process. In this stage the 3D triangles provided by the application layer are transformed into 2D triangles ready for rasterization. These transformations are made in a number of sub-steps [61], as shown in Fig. B.1,

1. Transform each object from its coordinates to a unified coordinates, called the world space.

2. Transform the scene such that the camera (eye) becomes in the origin and looking directly towards the z-direction.

3. The view frustum is clipped and transformed into normalized cube. Objects completely outside the frustum are discarded, while those that are partially

outside are clipped by creating new vertices.

4. The third dimension is discarded, and only the nearest to screen vertices are kept for further processing. Buffers are used in this process.

B.2 Rasterization Stage

In this stage triangles are rasterized into pixels. These transformations are made in a number of sub-steps [61], as shown in Fig. B.1,

1. 2D triangles are transformed into fragments. Fragments are created by intersecting each pixel position by 2D triangles. In case the intersection is not with a vertex but with a primitive (triangle or line), the attributes of the fragment is interpolated. The Color of each fragment is calculated by combining the attributes of color, texture and lighting. Also α (lighting coefficient) and/or fog can be combined.

2. Converging fragments into final pixels. In general each pixel is composed of many fragments.

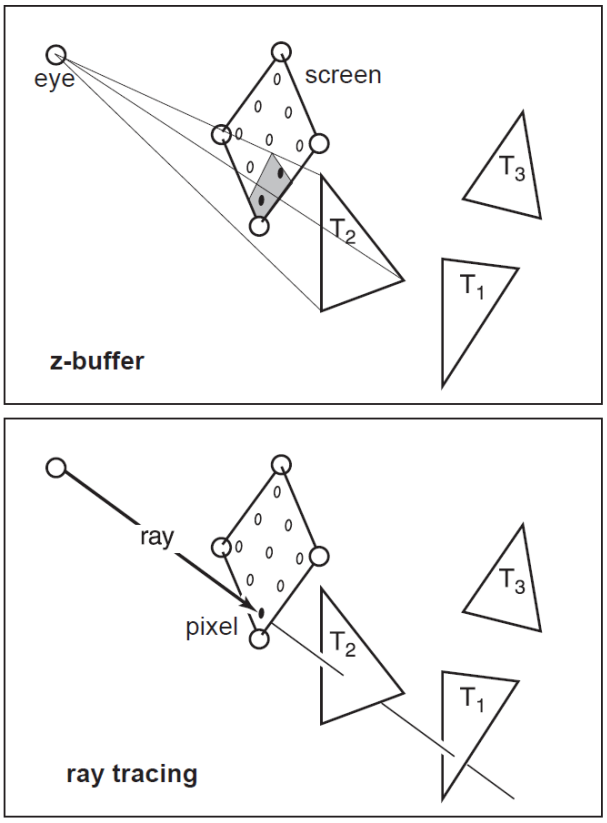


Figure B.2: In case of rasterization (z-buffer) the triangles are projected on the image plane, while in case of ray-tracing rays are traced through each pixel [62].

The third dimension of the 2D triangles are saved for being used in selecting the nearest to camera triangle, using z-buffer technique. Fig. B.2 shows the main difference between rasterization and ray-tracing. In rasterization the triangles are projected on the image plane, while in rasterization the traced rays that pass through pixels are used to calculate pixels attributes.

Appendix C

CUDA GPU Architecture

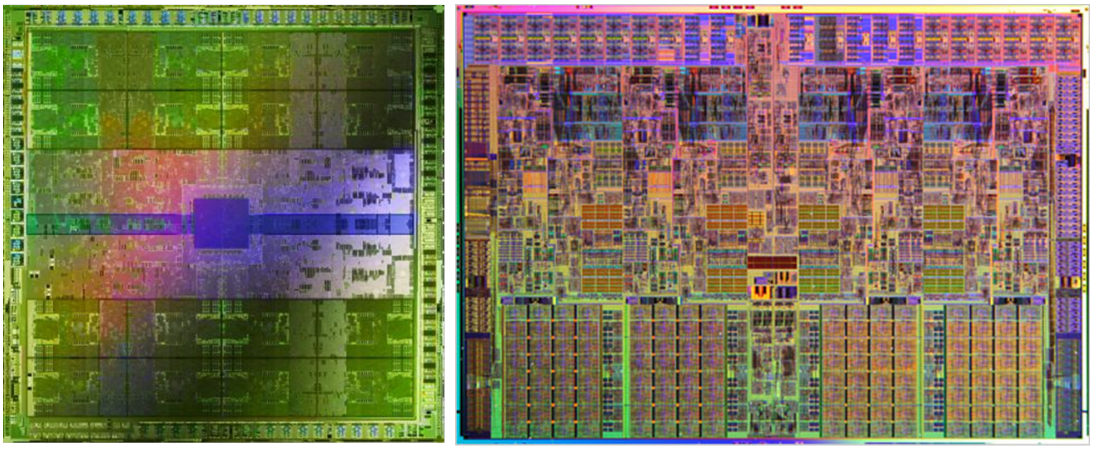
C.1 GPUs versus CPUs

Graphical processing units (GPUs) are originally designed for graphical processing. However the highly parallel design and the huge processing power of the GPU, opened the door for non-graphics (rasterization specially) processing on the GPUs, namely GPGPU (General Propose GPU). Current GPUs are considered a highly parallel may cores designs. Latest GPUs can have up to hundreds of computing cores. According to [63] the market demand for real-time, high-definition 3D graphics have driven the GPU industry to create a highly parallel, multithreaded, many-cores processor with tremendous computational power and very high memory bandwidth GPUs.

Current GPUs founds many general purpose applications and physics simulations [64]. Using the current GPGPU programing language (C++ extensions), like C for CUDA or OpenCL [63], many of the current CPU applications can be modified for running on the GPU. GPUs begin as a simple rasterization pipeline, then multi cores designs are introduced with two types of cores, vertex shader and pixel shader. With the introduction of unified shaders [65] many cores highly parallel deigned are now a reality.

GPUs are built on the idea of many small cores, while in the multi-cores CPUs few complex cores are used. Fig. C.1 shows the clear different between a GPU (nVidia GeForce 310) and a multi-cores CPU (Intel Core i7) layouts. The GPU layout shows highly repeated small units while few complex units appears in the CPU's layout.

Compute Unified Device Architecture or CUDA is a general purpose parallel computing architecture introduced by nVidia in 2006 [66]. CUDA architecture consists of N multiprocessors (MP), each multiprocessor consists of M processors (cores), as shown in Fig. C.2. In which the cores of the same MP executes the



(a) (b)

Figure C.1: (a) nVidia's GeForce 310 chip layout photo. (b) Intel's Core i7 layout photo .

same code (GPU kernel), in a parallel scheme. Such hierarchy of computing cores simplifies scheduling and the synchronization processes between the huge numbers of cores. GPU kernels are of the same instructions but with different sets of data.

Data sharing and execution synchronization are supported only between cores of the same MP. The smallest executable unit in the CUDA GPUs is called the warp. The warp can be considered as a variation on the SIMD (Single Instruction Multiple Data) instructions of the CPU, since GPU thread of the same kernel built up of the same instructions but with different running data. Each warp can carry up to 32 threads simultaneously.

C.2 Memory Organization

The CUDA GPUs' memory is organized as on-chip and off-chip memory. The memory is divide into six categories depending on each type level of sharing, maximum size and caching capabilities, as shown in Fig. C.2. The memory types are defined as,

- **Global memory:** off-chip non-cached memory, shared by all the cores of the GPU and the host device. The global memory is the main memory storage of the GPU.
- **Shared memory:** on-chip memory per MP, shared locally between the cores of each MP, and not accessed directly by the host device.
- **Local memory:** off-chip non-cached extension for the shared memory, with the same level of sharing.

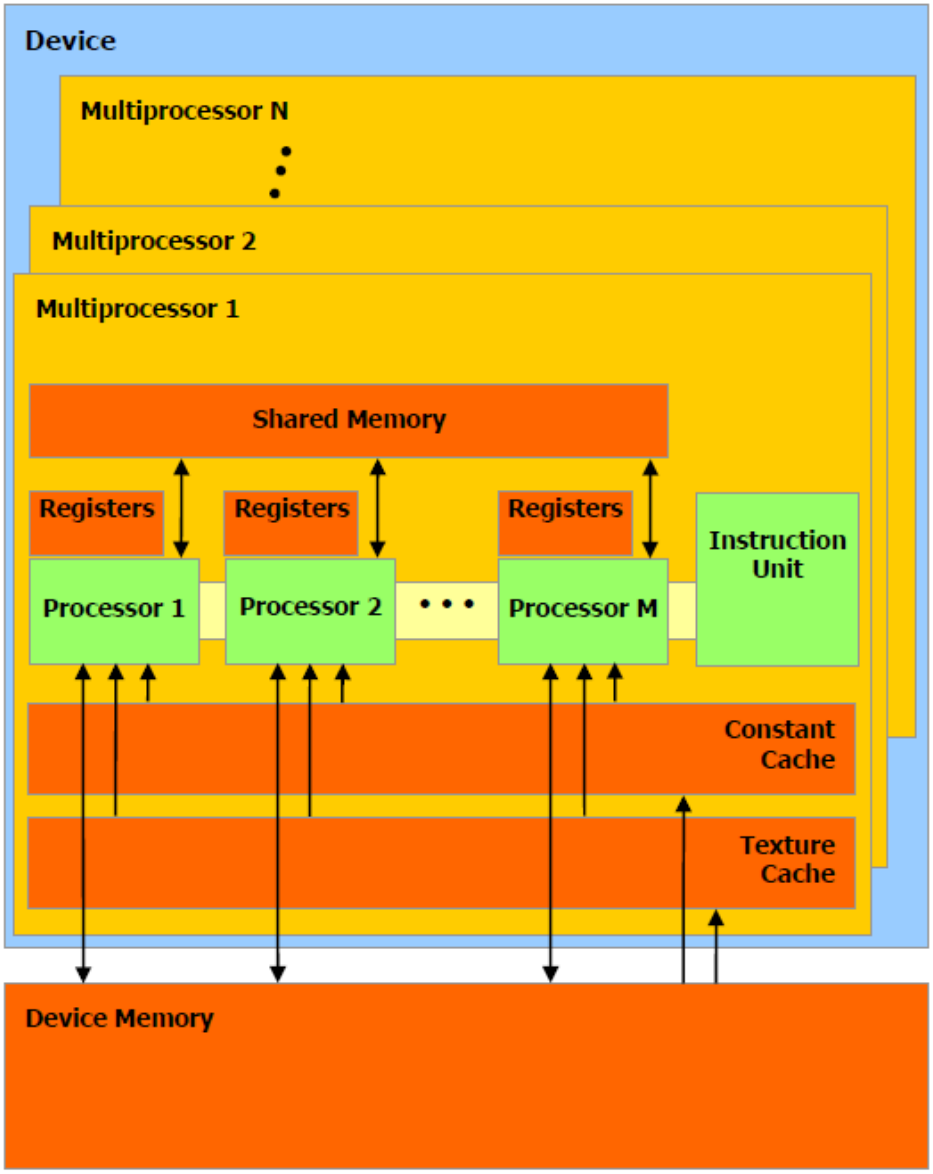


Figure C.2: The CUDA architecture organization showing the off-chip memory and GPU computing unit [66].

- **Texture memory:** off-chip read-only cached memory, with a global level of sharing between the GPU cores and the host. Texture memory gets its name because in graphics application it is used for storing the texture data.
- **Constant memory:** off-chip read-only cached constant size memory (64-KByte), shared by all the cores of the GPU.
- **Registers:** local registers per each processing core.

Appendix D

C# Test Platform

D.1 The Main Program

```
using System;
using System.Drawing;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RayTracing
{
    class Program
    {
        static void Main(string[] args)
        {
            // Read the database file
            ply_read db = new ply_read("bunny.ply");

            // Printing data
            Console.WriteLine("Number of vertices : {0}",
                db.nVertix);
            Console.WriteLine("Number of triangles: {0}\n",
                db.nTriangle);

            // Image pixels
            int width = 100;
            int length = 100;
```

```
// Set the image boundaries
vertix imTopLeft = new vertix(-1,0,0.1);
vertix imDownRigt = new vertix(-1,0.2,-0.1);

// Create the image plane
imageGray image =
    new imageGray(length,width,imTopLeft,
        imDownRigt);

// Camera position
vertix source = new vertix(-25,0.1,0);

// Main ray direction
vertix direction = new vertix();

// Secondary ray
vertix refliction = new vertix();

// Light source
vertix light = new vertix(-10,-10,30);

// Calculate the normals
Intersection.GetNormals(db);

// Calculate the maximums and the
// minimums of the triangle vertices
Intersection.GetMaxMin(db);

// Create the output image
Bitmap bitmapImage = new Bitmap(width, length);


/////----- Calculate all pixels -----/////

// Measure Execution time
DateTime start = DateTime.Now;

// Loop on all pixels
```

```
for (int i = 0; i < length; i++)
{
    for (int j = 0; j < width; j++)
    {
        // Main ray direction
        direction = image.pixelPos[i, j]
            - source;

        // Find the nearest intersection per
           pexil
        Intersection.
           FindNearestIntersectionPerPixel
                (db, image, source,
                direction, i, j);
    }
}

// End time
DateTime end = DateTime.Now;

// Measure Execution time
TimeSpan duration = end - start;
int d = duration.Milliseconds +
        1000 * duration.Seconds +
        60000 * duration.Minutes;

// Write the execution time to the screen
Console.WriteLine
    ("Intersection time : {0} [msec]", d);

// Write the final image
for (int i = 0; i < length; i++)
{
    for (int j = 0; j < width; j++)
    {
        // Check for intersection validation
        if (image.intersection[i, j].t == -1)
        {
            image.pixel[i, j] = 0;
```

```
                        bitmapImage.
                            SetPixel(j, i, Color.Black);
                    }
                    else
                    {
                        image.pixel[i, j] = 255;
                        bitmapImage.SetPixel
                            (j, i, Color.FromArgb
                            (255, 255, 255));
                    }
                }
            }

            // Save the image
            bitmapImage.Save("image.bmp");
        }
    }
}
```

D.2 Defined Types

```
namespace RayTracing
{
    // Type vertix
    class vertix
    {
        // Member Variables
        public float[] c; // coordinates

        // Constructors
        public vertix(float c0, float c1, float c2)
        {
            c = new float[3] { c0, c1, c2};
        }

        public vertix()
        {
            c = new float[3] { 0, 0, 0 };
```

```
    }

    // + operators overloading
    public static vertix operator
        +(vertix t1, vertix t2)
    {
        return new vertix(  t1.c[0] + t2.c[0],
                            t1.c[1] + t2.c[1],
                            t1.c[2] + t2.c[2]);
    }

    // - operators overloading
    public static vertix operator
        -(vertix t1, vertix t2)
    {
        return new vertix(  t1.c[0] - t2.c[0],
                            t1.c[1] - t2.c[1],
                            t1.c[2] - t2.c[2]);
    }

    // -ve operators overloading
    public static vertix operator -(vertix t)
    {
        return new vertix(- t.c[0],
                          - t.c[1],
                          - t.c[2]);
    }

    // * operators overloading 1
    public static vertix operator
        *(vertix t1, vertix t2)
    {
        return new vertix(  t1.c[0] * t2.c[0],
                            t1.c[1] * t2.c[1],
                            t1.c[2] * t2.c[2]);
    }

    // * operators overloading 2
    public static vertix operator
```

```
        *(float t1, vertix t2)
    {
        return new vertix(  t1 * t2.c[0],
                            t1 * t2.c[1],
                            t1 * t2.c[2]);
    }
    // / operators overloading 1
    public static vertix operator
        /(vertix t1,  float t2)
    {
        return new vertix(  t1.c[0] / t2,
                            t1.c[1] / t2,
                            t1.c[2] / t2);
    }

    // Dot product
    static public float
        Dot(vertix a, vertix b)
    {
        return (a.c[0] * b.c[0] +
                a.c[1] * b.c[1] +
                a.c[2] * b.c[2]);
    }

    // Cross product
    static public vertix
        Cross(vertix a, vertix b)
    {
        vertix temp = new vertix();

        temp.c[0] = a.c[1] * b.c[2]
                    - a.c[2] * b.c[1];
        temp.c[1] = a.c[2] * b.c[0]
                    - a.c[0] * b.c[2];
        temp.c[2] = a.c[0] * b.c[1]
                    - a.c[1] * b.c[0];
        return temp;
    }
}
```

```
// Type triangle
class triangle
{
    // Member variables
    public vertix[] ver;     // The three vertices
    public vertix[] normal; // normal
    public float[] max;      // maximum coordinates
    public float[] min;      // minimum coordinates

    // Constructors
    public triangle()
    {
        ver = new vertix[3];

        max = new float[3];
        min = new float[3];

        normal = new vertix[3];
    }
    public triangle(vertix a, vertix b, vertix c)
    {
        ver = new vertix[3];
        ver[0] = a;
        ver[1] = b;
        ver[2] = c;

        max = new float[3];
        min = new float[3];

        normal = new vertix[3];
    }
}

// Type ray
class ray
{
    // Member variables
    public float[,] source;
```

```
        public float[,] direction;

        // Constructor
        public ray(float[,] sourceIn, float[,] directionIn)
        {
            source = sourceIn;
            direction = directionIn;
        }
    }

    // Type light source
    class lightSource
    {
        // Members
        public float[,] position;

        // Constructor
        public lightSource(float[,] posIn)
        {
            position = posIn;
        }
    }

    // Type image
    class imageGray
    {
        /*
         *          j (width)
         *     +------------------------------------>
         *     |
         *   i | (length)
         *     |
         *     |
         *     |
         *     |
         *     |
         *     v
         *
         */
```

```
// Image length and width
public short length, width;

// Pixels array
public short[,] pixel;

// Pixels position in the virtual plane
public vertix[,] pixelPos;

// intersection point
public intersectionPoint[,] intersection;

// Bounding positions
private vertix postionLeftUp, postionRightDown,
        delta, ratio;

// Constructors
public imageGray(short lenghtIn, short widthIn,
                 vertix postionLeftUpIn, vertix
                    postionRightDownIn)
{
    // Set the input data
    length = lenghtIn;
    width = widthIn;
    postionLeftUp = postionLeftUpIn;
    postionRightDown = postionRightDownIn;

    // Calculate delta
    delta = postionRightDown - postionLeftUp;

    // Create arrays
    pixel = new short[length, width];
    pixelPos = new vertix[length, width];
    intersection = new intersectionPoint
                        [length, width];

    // Ratio vertix
    ratio = new vertix();
```

```
            // Intialize pixels by calculating the ratio
               and the
            // pixels postion and create intersection
               points
            for (int i = 0; i < length; i++)
            {
                // ratio [x]
                ratio.c[0] = (float)
                    ((i+ 0.5) / length ); // x

                // ratio [z]
                ratio.c[2] = (float)
                    ((i+ 0.5) / length ); // z

                for (int j = 0; j < width; j++)
                {
                    // ratio [y]
                    ratio.c[1] = (float)
                        ((j+ 0.5) / width ); // y

                    // Intialize with zero
                    pixel[i, j] = 0;

                    // postionLeftUp + (postionRightDown -
                       postionLeftUp) * ratio
                    pixelPos[i, j] = postionLeftUp + delta
                       * ratio;

                    // Create intersection point
                    intersection[i, j] =
                        new intersectionPoint();
                }
            }
        }
    }

    // Type intersection point
```

```
    class intersectionPoint
    {
        // Member variables
        public vertix point;       // intersection point
        public vertix normal;      // normal
        public float alfa, beta;   // reflection angles
        public int triangleIndex; // aaray index of the
                                  // triangle
        public float t;            // pameter t
        public bool shadow;        // light or shadow
        public vertix cosPhi;      // light model cos(phi)

        // constructor
        public intersectionPoint()
        {
            point = new vertix();
            normal = new vertix();
            alfa = beta = 0;
            triangleIndex = -1;
            t = -1;
            shadow = false;
            cosPhi = new vertix();
        }
    }
}
```

D.3 Database Reading

```
using System;
using System.IO;

namespace RayTracing
{
    // Read PLY file
    class ply_read
    {
        // Number of vertices and triangles
        public int nVertix, nTriangle;
```

```
// Database file name
private string fileName;

// Input stream
private StreamReader stream;

// Temp
private string line;

// Data array
public vertix[] ver;
public triangle[] tri;

// Constructor
public ply_read(string fileNameIn)
{
    fileName = fileNameIn;
    stream = new StreamReader(fileName);
    ReadHeader();
    ReadVert();
    ReadTriangle();
}

// Read header
private void ReadHeader()
{
    // Header end flag
    bool headerEnd = false;

    // check for file end and header end
    while(stream.EndOfStream == false &&
          headerEnd == false)
    {

        // Read a new line
        line = stream.ReadLine();

        // check for header end
```

```
            if(line.Contains("end_header"))
            {
                headerEnd = true;
                break;
            }

            // check for elements
            if(line.Contains("element"))
            {
                HeaderElement(line);
            }
        }
    }

    // Read header element
    private void HeaderElement(string line)
    {
        // Split on spaces
        string[] words = line.Split(' ');

        // Check for keyword vertix
        if(line.Contains("vertex"))
        {
            nVertix = Convert.ToInt32(words[2]);
            return;
        }

        // Check for keyword face
        if (line.Contains("face"))
        {
            nTriangle = Convert.ToInt32(words[2]);
            return;
        }
    }

    // Read vertices data
    private void ReadVert()
    {
        // Create vertices array
```

```
        ver = new vertix[nVertix];

        string[] words; // temp

        // Loop on all lines contain vertices' data
        for (int i = 0; i < nVertix; i++ )
        {
            // Read a new line
            line = stream.ReadLine();

            // Split on spaces
            words = line.Split(' ');

            // Creat a new vertix
            ver[i] = new vertix();

            // Convert string into float (single)
            ver[i].c[0] = Convert.ToSingle(words[0]);
            ver[i].c[1] = Convert.ToSingle(words[1]);
            ver[i].c[2] = Convert.ToSingle(words[2]);
        }
    }

    // Read triangles data
    private void ReadTriangle()
    {
        // Create triangles array
        tri = new triangle[nTriangle];

        string[] words; // temp
        int v0, v1, v2; // temp, Triangle's 3 vertices
                        // postions

        // Loop on all lines contain triangles' data
        for (int i = 0; i < nTriangle; i++)
        {
            // Read a new line
            line = stream.ReadLine();
```

```
                // Split on spaces
                words = line.Split(' ');

                // Convert string into int
                v0 = Convert.ToInt32(words[1]);
                v1 = Convert.ToInt32(words[2]);
                v2 = Convert.ToInt32(words[3]);

                // Create a new triangle
                tri[i] = new triangle();

                // Read triangls' vertices
                tri[i].ver[0] = ver[v0];
                tri[i].ver[1] = ver[v1];
                tri[i].ver[2] = ver[v2];
            }

        }
    }
}
```

D.4 Tracing Algorithms

```
using System;
using System.Drawing;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace RayTracing
{
    static class Intersection
    {
        // Find the nearest to eye intersection point
        static public void FindNearestIntersectionPerPixel
            (ply_read db, imageGray image, vertix source,
               vertix direction, int i, int j)
        {
```

```
// temp intersection point
intersectionPoint tempIntersection =
    new intersectionPoint();

// Loop on all triangles to get the nearest
   intersection
// (only as a test for algorithm)
for (int tI = 0; tI < db.nTriangle; tI++)
{
    // Get current intersection
    tempIntersection =
        Intersection.
           GetIntersectionNativeInterval
        (source, direction, db.tri[tI], tI,
           image.intersection[i, j].t);

    // The intersection is valid if t not equal
    // to -1
    if (tempIntersection.t >= 0)
    {
        image.intersection[i, j] =
           tempIntersection;
        image.intersection[i, j].triangleIndex
           = tI;
    }

}

// if there is an intersection get the point
// and the normal
if (image.intersection[i, j].t != -1)
{
    // Point
    image.intersection[i, j].point =
        source + image.intersection[i, j].t *
           direction;

    // Normal
```

```
            int tIndex = image.intersection[i,j].
               triangleIndex;
            image.intersection[i, j].normal =
                ( 1 - image.intersection[i, j].alfa
                - image.intersection[i, j].beta)
                * db.tri[tIndex].normal[0]
                + image.intersection[i, j].alfa
                * db.tri[tIndex].normal[1]
                + image.intersection[i, j].beta
                * db.tri[tIndex].normal[2];
        }
    }

    // Check for intersection using Moller algorithm -
    // implementation 1/2
    static public intersectionPoint
       GetIntersectionMoller
        (vertix source, vertix direction, triangle tri,
        int triIndex, float oldT)
    {
        // Temp intersection point
        intersectionPoint iPoint = new
           intersectionPoint();

        // Calculate algorithm's variables
        vertix E1 = tri.ver[1] - tri.ver[0];
        vertix E2 = tri.ver[2] - tri.ver[0];
        vertix T = source - tri.ver[0];

        vertix P = vertix.Cross(direction, E2);
        vertix Q = vertix.Cross(T, E1);

        float A = vertix.Dot(P, E1);

        // Check intersection validation condition
        if (A <= 0)
        {
            iPoint.t = -1;
            return iPoint;
```

```
        }

        // Calculate parameter t
        float tTemp = vertix.Dot(Q, E2) / A;

        // Check intersection validation condition
        if(tTemp < 0 || (oldT != -1 && tTemp > oldT))
        {
            iPoint.t = -1;
            return iPoint;
        }

        // calculate alfa
        iPoint.alfa = vertix.Dot(P, T);  //

        // Check intersection validation condition
        if (iPoint.alfa > A || iPoint.alfa < 0)
        {
            iPoint.t = -1;
            return iPoint;
        }

        // calculate beta
        iPoint.beta = vertix.Dot(Q, direction) ;

        // Check intersection validation condition
        if (iPoint.beta > A - iPoint.alfa  ||
            iPoint.beta < 0)
        {
            iPoint.t = -1;
            return iPoint;
        }

        // return the intersection point in case of
        // valid intersection
        iPoint.t = tTemp;
        return iPoint;
    }
```

```
// Intersection rejection test + Moller algorithm -
// implementation 1/2
static public intersectionPoint
    GetIntersectionMollerInterval(vertix source,
       vertix direction, triangle tri,
    int triIndex, float oldT)
{
    // Temp intersection point
    intersectionPoint iPoint =
        new intersectionPoint();

    // Tx, Ty, Tz
    float TxMax, TxMin, TyMax, TyMin, TzMax, TzMin,
        TMax, TMin, t;

    /////----- Rejection Phase 1 -----/////

    // Calculating Tx
    TxMax = (tri.max[0] - source.c[0])
            / direction.c[0];
    TxMin = (tri.min[0] - source.c[0])
            / direction.c[0];

    // Calculating Ty
    TyMin = (tri.min[1] - source.c[1])
            / direction.c[1];
    TyMax = (tri.max[1] - source.c[1])
            / direction.c[1];

    // Reorder the interval in case of wrong order
    if (TyMin > TyMax)
    {
        t = TyMin;
        TyMin = TyMax;
        TyMax = t;
    }

    // Reorder the interval in case of wrong order
    if (TxMin > TxMax)
```

```
{
    t = TxMin;
    TxMin = TxMax;
    TxMax = t;
}

// Find the intersection of Tx and Ty
TMax = TxMax > TyMax ? TyMax : TxMax;  // min
   of 2
TMin = TxMin > TyMin ? TxMin : TyMin;  // max
   of 2

// Check for valid intersection
if (TMin > TMax)
{
    iPoint.t = -1;
    return iPoint;
}

/////----- Rejection Phase 2 -----/////

// Calculating Tz
TzMax = (tri.max[2] - source.c[2])
        / direction.c[2];
TzMin = (tri.min[2] - source.c[2])
        / direction.c[2];

// Reorder the interval in case of wrong order
if (TzMin > TzMax)
{
    t = TzMin;
    TzMin = TzMax;
    TzMax = t;
}

// Find the intersection of Txy and Tz
TMax = TzMax > TMax ? TMax : TzMax;
TMin = TzMin > TMin ? TzMin : TMin;
```

```
// Check for valid intersection
if (TMin > TMax) //|| TMin > oldT
{
    iPoint.t = -1;
    return iPoint;
}

// Compare Tmin with the current nearest
// to eye intersection
if (oldT != -1 && TMin > oldT)
{
    iPoint.t = -1;
    return iPoint;
}


/////----- Intersection Phase -----/////

// Calculate algorithm variables
vertix E1 = tri.ver[1] - tri.ver[0];
vertix E2 = tri.ver[2] - tri.ver[0];
vertix T = source - tri.ver[0];

vertix P = vertix.Cross(direction, E2);
vertix Q = vertix.Cross(T, E1);

float A = vertix.Dot(P, E1);

// Check intersection validation condition
if (A <= 0)
{
    iPoint.t = -1;
    return iPoint;
}

// Calculate parameter t
float tTemp = vertix.Dot(Q, E2) / A;

// Check intersection validation condition
```

```
        if (tTemp < 0 || (oldT != -1 && tTemp > oldT))
        {
            iPoint.t = -1;
            return iPoint;
        }

        // calculate alfa
        iPoint.alfa = vertix.Dot(P, T);

        // Check intersection validation condtion
        if (iPoint.alfa > A || iPoint.alfa < 0)
        {
            iPoint.t = -1;
            return iPoint;
        }

        // calculate beta
        iPoint.beta = vertix.Dot(Q, direction);

        // Check intersection validation condition
        if (iPoint.beta > A - iPoint.alfa ||
            iPoint.beta < 0)
        {
            iPoint.t = -1;
            return iPoint;
        }

        // return the intersection point in case of
        // valid intersection
        iPoint.t = tTemp;
        return iPoint;
    }


    // Check for intersection using Badouel algorithm
    static public intersectionPoint
       GetIntersectionBadouel
        (vertix source, vertix direction, triangle tri,
        int triIndex, float oldT)
```

```
{
    // Temp intersection point
    intersectionPoint iPoint =
        new intersectionPoint();

    // Algorithm's variables
    int i0, i1, i2;

    // Temp t
    float tempT =
        -(tri.d + vertix.Dot(tri.normal[0], source)
            /
        vertix.Dot(tri.normal[0], direction));

    if (tempT < 0 || (oldT != -1 && tempT > oldT))
    {
        iPoint.t = -1;
        return iPoint;
    }

    i0 = Utilities.max3Pos(tri.normal[0].c[0],
        tri.normal[0].c[1], tri.normal[0].c[2]);

    // Set i0, i1 and i2
    if (i0 == 0)
    {
        i1 = 1;
        i2 = 2;
    }
    else if (i0 == 1)
    {
        i1 = 0;
        i2 = 2;
    }
    else
    {
        i1 = 0;
        i2 = 1;
    }
```

```
// Algorithm's variables
float u0, u1, u2, v0, v1, v2;

// Plane intersection poit position
iPoint.point = source + iPoint.t * direction;


// Calculate algorithm's variables
u1 = tri.ver[1].c[i1] - tri.ver[0].c[i1];
v1 = tri.ver[1].c[i2] - tri.ver[0].c[i2];

u2 = tri.ver[2].c[i1] - tri.ver[0].c[i1];
v2 = tri.ver[2].c[i2] - tri.ver[0].c[i2];

float A = u1 * v2 - u2 * v1;

// Rejection condition 1
if(A==0)
{
    iPoint.t = -1;
    return iPoint;
}

u0 = iPoint.point.c[i1] - tri.ver[0].c[i1];
v0 = iPoint.point.c[i2] - tri.ver[0].c[i2];

// Alfa
float tempAlfa = (u0 * v2 - u2 * v0)/A;

// Rejection condition 2
if (tempAlfa < 0 || tempAlfa > 1)
{
    iPoint.t = -1;
    return iPoint;
}

// Beta
float tempBeta = (u1 * v0 - u0 * v1)/A;
```

```
        // Rejection condition 1
        if (tempBeta < 0 || tempBeta > 1)
        {
            iPoint.t = -1;
            return iPoint;
        }

        // return the intersection point in case of
        // valid intersection
        iPoint.t = tempT;
        return iPoint;
    }


    // Calculate the normals to the triangle plane
    // at each vertix
    public static void GetNormals(ply_read db)
    {
        // Loop on all triangles
        for (int i = 0; i < db.nTriangle; i++)
        {
            db.tri[i].normal[0] = vertix.Cross
                ( db.tri[i].ver[1] - db.tri[i].ver[0],
                  db.tri[i].ver[2] - db.tri[i].ver[0]);

            db.tri[i].normal[1] = vertix.Cross
                ( db.tri[i].ver[2] - db.tri[i].ver[1],
                  db.tri[i].ver[0] - db.tri[i].ver[1]);

            db.tri[i].normal[2] = vertix.Cross
                ( db.tri[i].ver[0] - db.tri[i].ver[2],
                  db.tri[i].ver[1] - db.tri[i].ver[2]);

            db.tri[i].d = -vertix.Dot
                ( db.tri[i].ver[0], db.tri[i].normal
                   [0]);

        }
```

```
        }

        // Calculate the maximums and the minimums
        // of the vertices coordinates
        public static void GetMaxMin(ply_read db)
        {
            for (int i = 0; i < db.nTriangle; i++)
            {
                for (int j = 0; j < 3; j++)
                {
                    db.tri[i].max[j] = Utilities.max3(
                        db.tri[i].ver[0].c[j],
                        db.tri[i].ver[1].c[j],
                        db.tri[i].ver[2].c[j]);

                    db.tri[i].min[j] = Utilities.min3(
                        db.tri[i].ver[0].c[j],
                        db.tri[i].ver[1].c[j],
                        db.tri[i].ver[2].c[j]);
                }

            }

        }
    }

    public static class Utilities
    {
        // Get max. of 3
        public static float max3(float a, float b, float c)
        {
            float max = a;

            if (b > max)
            {
                max = b;
            }
```

```
        if (c > max)
        {
            max = c;
        }

        return max;
    }

    // Get max position of 3
    public static int max3Pos(float a, float b, float c
        )
    {
        float max = a;
        int maxPos = 0;

        if (b > max)
        {
            max = b;
            maxPos = 1;
        }

        if (c > max)
        {
            max = c;
            maxPos = 2;
        }

        return maxPos;
    }

    // Get min. of 3
    public static float min3(float a, float b, float c)
    {
        float min = a;

        if (b < min)
        {
            min = b;
        }
```

```
            if (c < min)
            {
                min = c;
            }

            return min;
        }

    }
}
```