

**HIGH PERFORMANCE STEREOSCOPIC
RAY TRACING ON THE GPU**

BY

MAZEN ABDULAZIZ SALEH AL-HAGRI

A Thesis Presented to the
DEANSHIP OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

In

COMPUTER SCIENCE

DATE: MAY 2012

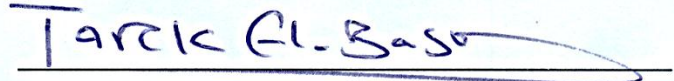
KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS

DHAHRAN 31261, SAUDI ARABIA

DEANSHIP OF GRADUATE STUDIES

This thesis, written by **MAZEN ABDULAZIZ SALEH AL-HAGRI** under the direction of his thesis advisor and approval by his thesis committee, has been presented and accepted by the Dean of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE IN COMPUTER SCIENCE**.

Thesis Committee



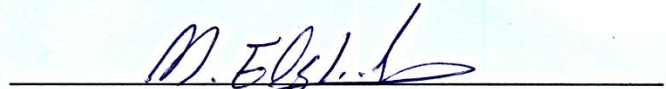
Dr. Tarek Helmy El-Basuny (Chairman)



Dr. Adel F. Ahmed (Co-chairman)



Dr. Adel F. Ahmed
(Department Chairman)



Dr. Moustafa El-Shafei (Member)

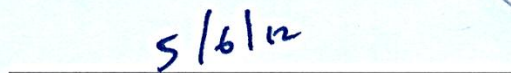


Dr. Salam Zummo
(Dean of Graduate Studies)

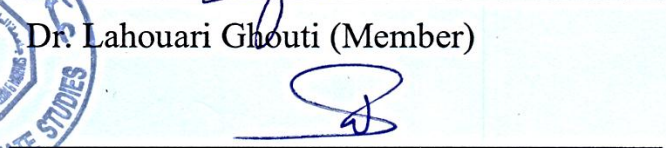




Dr. Lahouari Ghouti (Member)



Date



Dr. Sami Zhioua (Member)

For my father, Dr. Abdulaziz Saleh,

my mother, Awatef Salem,

my wife, Noora Mohammed,

and my son, Laith.

ACKNOWLEDGEMENT

Acknowledgement is due to King Fahd University of Petroleum and Minerals for supporting this research. I would also like to acknowledge my sponsors, Hadhramout Establishment for Human Development, for granting me this outstanding opportunity to obtain my Master's degree, and for their generous financial support.

Moreover, I wish to express my deepest appreciations to the chairman and co-chairman of the thesis committee, Dr. Tarek Helmy and Dr. Adel Ahmed, for their effective advice and support. Also, it gives me great pleasure to appreciate the other committee members, Dr. Moustafa El-shafei, Dr. Lahouari Ghouti and Dr. Sami Zhioua, for their fruitful remarks and comments.

Last but not least, thanks to my dear wife who supported me with love, inspiration and patience. Finally, to everybody who contributed to this achievement either directly or indirectly; thank you.

TABLE OF CONTENTS

Final Approval.....	ii
Dedication.....	iii
Acknowledgement.....	iv
Table of Contents	v
Table of Figures.....	viii
List of Tables.....	xi
List of Algorithms	xii
Thesis Abstract (English)	xiii
Thesis Abstract (Arabic).....	xiv
Chapter One: Introduction.....	1
1.1 Problem Statement.....	3
1.2 Contribution	3
1.3 Motivation.....	4
Chapter Two: Literature Survey	5
2.1 Ray Tracing.....	5
2.1.1 The 3D Scene Model.....	5
2.1.2 The Ray Tracing Algorithm	7

2.1.3	Acceleration Structures	9
2.1.4	Parallel Ray-tracing	10
2.2	Temporal Coherence.....	10
2.2.1	Image-Space Temporal Coherence.....	11
2.2.2	Image-Space Temporal Coherence in Stereoscopic Ray Tracing	13
2.3	Stereoscopy.....	20
2.3.1	Stereoscopic Displays.....	21
2.3.2	Rendering Stereoscopic Images	23
2.4	Massively Parallel Graphics Processing Units	27
2.4.1	Taxonomy.....	27
2.4.2	Trending Architectures	28
Chapter Three: Methodology		30
3.1	Parallel Reprojection.....	30
3.1.1	Missed Pixel Resolution	31
3.1.2	Overlapped Pixel Resolution.....	31
3.1.3	Bad Pixel Resolution	36
3.2	Complexity Analysis.....	37
3.3	Kernels Pseudocode	40
3.3.1	Buffer-Based Kernels	40

3.3.2	Atomic-Based Kernels.....	42
Chapter Four: Experimental Results and Analysis.....		44
4.1	Ray Tracer Implementations.....	44
4.2	Testbeds	46
4.3	Stereo Scene Setup.....	46
4.4	Benchmarks	50
4.5	Results and Discussion	52
4.5.1	Performance Benchmarks Results.....	52
4.5.2	Pixel Error Benchmarks Results.....	54
4.5.3	Time Views	55
4.5.4	Outputs	55
Chapter Five: Conclusion and Future Work.....		75
5.1	Summary.....	75
5.2	Contribution to Knowledge	76
5.3	Limitations	76
5.4	Future Work.....	77
References		79
Vita		86

TABLE OF FIGURES

Figure 1: An image generated using ray-tracing [1].	6
Figure 2: 3D scene model.....	7
Figure 3: Illustration of the core ray tracing algorithm [20].	8
Figure 4: Monoscopic Perspective Projection.	14
Figure 5: Stereoscopic perspective projection.....	15
Figure 6: Reprojection errors. (a) Overlapped pixels. (b) and (c) Bad pixels. [29]	18
Figure 7: Monocular depth cues. (a) Relative size. (b) Lights and shadows. (c) Perspective. (d) Occlusion. (e) Haze.	21
Figure 8: (a) Monoscopic scene. (b) Stereoscopic scene.	24
Figure 9: (a) Positive parallax. (b) Zero parallax. (c) Negative parallax.	25
Figure 10: Safe visible area for inserting 3D surfaces into a stereo scene.	26
Figure 11: GPU vs. CPU performance trends in GFLOPS (10^9 FLOPS) [41].....	28
Figure 12: A possible case when the reprojections to a single position reach a maximum.....	33
Figure 13: Stereo scene parameters.....	35
Figure 14: K is marked as a bad pixel and is fully ray traced.	37
Figure 15: Fixed Spheres scene. (a) Mono output. (b) Stereo output.....	48
Figure 16: Animated Spheres scene. (a) Mono output. (b) Stereo output.	48
Figure 17: Sponza scene. (a) Mono output. (b) Stereo output.....	49
Figure 18: Buddha scene. (a) Mono output. (b) Stereo output.....	49
Figure 19: Dragon scene (a) Mono output. (b) Stereo output.	50

Figure 20: Performance of ray tracing the Fixed Spheres scene when increasing image dimensions in Testbed-1.....	56
Figure 21: Performance of ray tracing the Animated Spheres scene when increasing image dimensions in Testbed-1.....	57
Figure 22: Performance of ray tracing the Sponza scene when increasing image dimensions in Testbed-1.....	58
Figure 23: Performance of ray tracing the Buddha scene when increasing image dimensions in Testbed-1.....	59
Figure 24: Performance of ray tracing the Dragon scene when increasing image dimensions in Testbed-1.....	60
Figure 25: Performance of ray tracing the Dragon scene when increasing the interaxial distance e in Testbed-1.....	61
Figure 26: Performance of ray tracing the Dragon scene when increasing number of lights in Testbed-1.....	61
Figure 27: Speedup summary relative to the naïve ray tracer, Testbed-1.....	62
Figure 28: Performance of ray tracing the Fixed Spheres scene when increasing image dimensions in Testbed-2.....	63
Figure 29: Performance of ray tracing the Animated Spheres scene when increasing image dimensions in Testbed-2.....	64
Figure 30: Performance of ray tracing the Sponza scene when increasing image dimensions in Testbed-2.....	65
Figure 31: Performance of ray tracing the Buddha scene when increasing image dimensions in Testbed-2.....	66

Figure 32: Performance of ray tracing the Dragon scene when increasing image dimensions in Testbed-2.....	67
Figure 33: Performance of ray tracing the Dragon scene when increasing the interaxial distance e in Testbed-2.....	68
Figure 34: Performance of ray tracing the Dragon scene when increasing number of lights in Testbed-2.	68
Figure 35: Speedup summary relative to the naïve ray tracer, Testbed-2.	69
Figure 36: MSE when increasing image dimensions.	70
Figure 37: MSE when increasing the interaxial distance e	70
Figure 38: Time views of the threads generating the right image using: (a) Naïve ray tracer. (b) Atomic-based ray tracer.....	71
Figure 39: Output of naïve ray tracer. (a) Left image. (b) Right image. (c) Anaglyph stereo image.....	72
Figure 40: Output of buffer-based ray tracer. (a) Left image. (b) Right image. (c) Anaglyph stereo image.....	73
Figure 41: Output of atomic-based ray tracer. (a) Left image. (b) Right image. (c) Anaglyph stereo image.....	74

LIST OF TABLES

Table 1: Comparison between different mechanisms for resolving reprojection problems.	31
Table 2: Benchmarked ray tracers.	45
Table 3: Specifications of the testbeds.	46
Table 4: 3D objects of the scenes.	47
Table 5: Performance benchmarks.	51
Table 6: Pixel-error benchmarks.	51

LIST OF ALGORITHMS

Algorithm 1: Stereoscopic Reprojection Algorithm.....	19
Algorithm 2: Kernel for generating the left image using the buffer-based resolution. .	41
Algorithm 3: Kernel for generating the right image.....	42
Algorithm 4: Kernel for generating the left image.....	42
Algorithm 5: Kernel for generating the right image.....	43

THESIS ABSTRACT (ENGLISH)

NAME: Mazen Abdulaziz Saleh Al-Hagri
TITLE: High Performance Stereoscopic Ray Tracing on the GPU
MAJOR FIELD: Computer Science
DATE OF DEGREE: May, 2012

Nowadays, large 3D stereoscopic displays are trending, requiring rendering at higher resolution and at high frame rates. This development aims at delivering more realistic details, but it also comes at a significant cost: bowing to the computational constraints, since synthesizing stereo image pairs separately doubles the rendering cost. This poses a problem for interactive applications viewed on those displays, especially if computationally expensive rendering techniques, such as ray tracing, are employed.

In order to achieve high-quality rendering of stereo image pairs at a lower cost, one can exploit *temporal coherence* techniques: taking advantage of the inherent similarity of contents between both stereo pairs to reduce the rendering cost. This work attempts to modify one of the most effective techniques for utilizing temporal coherence between a ray traced stereo pair, called the *reprojection algorithm*, in order to make it run efficiently in massively parallel processors; such as the graphics processing units.

Keywords: Ray Tracing, Stereoscopy, Image-Space Temporal Coherence, Graphics Processing Unit (GPU), Massively Parallel Processors, Reprojection.

THESIS ABSTRACT (ARABIC)

ملخص الرسالة

الاسم:	مازن عبدالعزيز صالح الهجري
العنوان:	خوارزمية تتبع الشعاع عالية الأداء في وحدة معالجة الرسومات
التخصص:	علوم الحاسب الآلي
تاريخ التخرج:	جمادى الآخرة، 1433 هـ

شاشات العرض ثلاثية البعد أو الاستيريوسكوبية الضخمة بدأت تشيع في الوقت الراهن، متطلبة تركيب الصور المعروضة فيها بمقاسات كبيرة وبسرعات عالية لمعدلات عرض الصور في الثانية. يهدف هذا التطور إلى عرض تفاصيل أكثر قرباً للواقع، لكنه يأتي بكلفة عالية: الخضوع لقيود قدرات الحوسبة الآلية؛ لأن تركيب زوج الصور الاستيريوسكوبية يُترجم إلى مضاعفة الجهد في حال تصوير كل صورة على حدة. هذا الخضوع يخلق مشكلة للتطبيقات التفاعلية المعروضة في تلك الشاشات، خصوصاً إذا تم توظيف تقنيات تصوير مكلفة حاسوبياً – مثل تقنية تتبع الشعاع – في هذه التطبيقات.

من أجل الحصول على تصوير عالي الجودة لزوج الصور الاستيريوسكوبية بكلفة أقل، من الممكن تسخير تقنيات التشيع المؤقت؛ حيث يُستغل التشابه المتأصل ما بين زوج الصور الاستيريوسكوبية لتحقيق ذلك. أحد أكثر هذه التقنيات فعالية لتوليد الصور الاستيريوسكوبية باستخدام طريقة تتبع الشعاع تسمى "خوارزمية إعادة الإسقاط". تقوم في هذه الأطروحة بتعديل هذه التقنية لجعلها تنفذ بشكل كفؤ في المعالجات المتوازية الهائلة – مثل وحدات معالجات الرسومات.

كلمات مفتاحية: تتبع الشعاع، الاستيريوسكوبية، التشيع المؤقت في فضاء الصورة، وحدات معالجة الرسومات، المعالجات المتوازية الهائلة، إعادة الإسقاط.

CHAPTER ONE

INTRODUCTION

To humans, the visual system is the most important sensory device, since the perception and recognition of the surrounding world heavily rely on it. For thousands of years, prehistoric humans spanning all cultures kept visual memories of their surroundings through simple paintings that exhibited little visual cues, lacking perspective and depth information. With the development of arts, paintings exhibited more sensory cues, including precise perspective drawings, shadows and even depth-of-field.

Artists of the renaissance era had realized that each human eye perceives a slightly different image, resulting in a depth cue that was impossible for a painter to portray in a single canvas. Stereopsis, the process of perceiving different depths from the two slightly different projections of the world onto the two eyes, was only well established in the 18th century. Understanding this concept led to the invention of the stereoscopy technique, where an added depth cue of an image is enhanced by presenting two offset images, called stereo images pair or stereo pair, separately to the left and right eye of the viewer. The invention of photography made it easy to produce stereo images later on.

The history of computer graphics started similar to that of human arts; where the first image-synthesis techniques produced simple 2D drawings. With the evolvement of

computer hardware, Graphics Processing Units (GPUs) and computer graphics algorithms, 3D and more realistic images could be produced. Modern techniques for realistic image synthesis include Ray-tracing [2, 3] and Radiosity [4]. However, it takes an ample amount of time to compute precise realistic images with these techniques.

Nowadays, large displays are becoming mainstream, requiring rendering at higher-resolution and at high frame rates. This development aims at delivering more realistic details and better accuracy, but it also comes at a significant cost: bowing to the computational constraints. Hence, interactive applications (such as video games) use rendering techniques that are less computationally-intensive, such as Rasterization [5], at the cost of producing less realistic images.

Throughout the years, researchers have been competing to develop algorithms that are able to perform ray-tracing in real time. One of the earliest attempts to implement an interactive ray-tracer dates back to 1994, when Bishop et al. [6] introduced Frameless Rendering. The state of the art interactive ray-tracers implement various optimization techniques, including Acceleration Structures [7] and Temporal Coherence [8], the topic of interest in this thesis. Moreover, the recent advent of massively parallel processing units, CPUs and General Purpose GPUs (GPGPUs), also contributed to the realization of interactive ray-tracing [9, 10].

1.1 Problem Statement

Another challenge for interactive realistic rendering is being posed by large stereoscopic 3D displays that are currently trending. Even for the 3D displays at the lowest end, the 3D Stereo Displays, each frame of an animation sequence must be rendered twice as a stereo pair, doubling the rendering cost if naïvely implemented.

Generally, in order to realize high-quality rendering at a lower cost, Temporal Coherence can be exploited. Temporal Coherence is the correlation of content in object space and image space between adjacent rendered frames. By taking advantage of temporal coherence, redundant computation can be avoided, and the rendering cost can be significantly reduced with a minimal decrease in quality.

Temporal coherence was also exploited to render stereo images, where the second image of a stereo pair is computed by exploiting information computed in the first image, thus speeding up rendering. However, some of the existing techniques are sequential in nature, and are not optimized to run on modern massively parallel processing units or GPUs.

1.2 Contribution

This work will focus on devising an efficient ray-tracing algorithm, based on an existing one, which produces high quality stereo images using temporal coherence in image space. The resulting algorithm is expected to produce outputs of comparably high frame rates and at high resolution, and it will be completely executed on a state-of-the-art GPU.

1.3 Motivation

Our motivations for this work can be stated as follows:

- The quest for higher performance in stereoscopic ray tracing.
- The recent increased popularity of stereoscopic displays.
- The need to re-invent resolutions for one of the most powerful algorithms that produce high-quality rendering at a lower cost; the reprojection algorithm [11], so that it can work in massively parallel environments.

The rest of this work is organized as follows. Chapter 2 addresses a detailed literature review on ray tracing, temporal coherence, stereoscopy and massively parallel graphical processing units, alongside their related work. Then, the conceptual design of the proposed algorithms including the suggested resolutions to optimize existing algorithms to run them in a massively parallel manner, are presented in Chapter 3. Chapter 4 provides a detailed account of the experimental results and analysis of the approaches developed in this thesis when implemented in an existing GPU development platform. Finally, Chapter 5 presents the conclusion, detailing the main contributions of this work in addition to the limitations of the proposed solutions. Also, possible future improvements and refinements of the current work are drawn therein.

CHAPTER TWO

LITERATURE SURVEY

This thesis spans four different areas of computer graphics, display technologies and parallel architecture: ray tracing, temporal coherence, stereoscopy and massively parallel graphics processing units. This chapter provides the necessary background in each area, alongside the related state of the art work in each one of them.

2.1 Ray Tracing

Ray tracing is an image synthesis technique for generating images from a 3D model of a scene (Section 2.1.1). It is famous for producing images that exhibit effects with high degree of realism (Figure 1). However, it is also known for the associated high computational cost, due to the way it operates (Section 2.1.2). Due to this, its use in interactive applications is mostly limited to research.

2.1.1 The 3D Scene Model

A 3D scene model is a set of data structures, describing the attributes of a virtual scene elements, including the camera, image-plane, geometric primitives, lights, materials, etc. For our purposes, we will keep track of the following attributes as illustrated in Figure 2:



Figure 1: An image generated using ray-tracing [1].

- Camera position in 3D space.
- Image plane I_p , and pixels in the image plane I_p that correspond to a point p in 3D space.
- Image plane dimensions w and h .
- Camera frustum, a region of the 3D space that specifies the field of view of the camera.
- Distance from camera to image plane d .

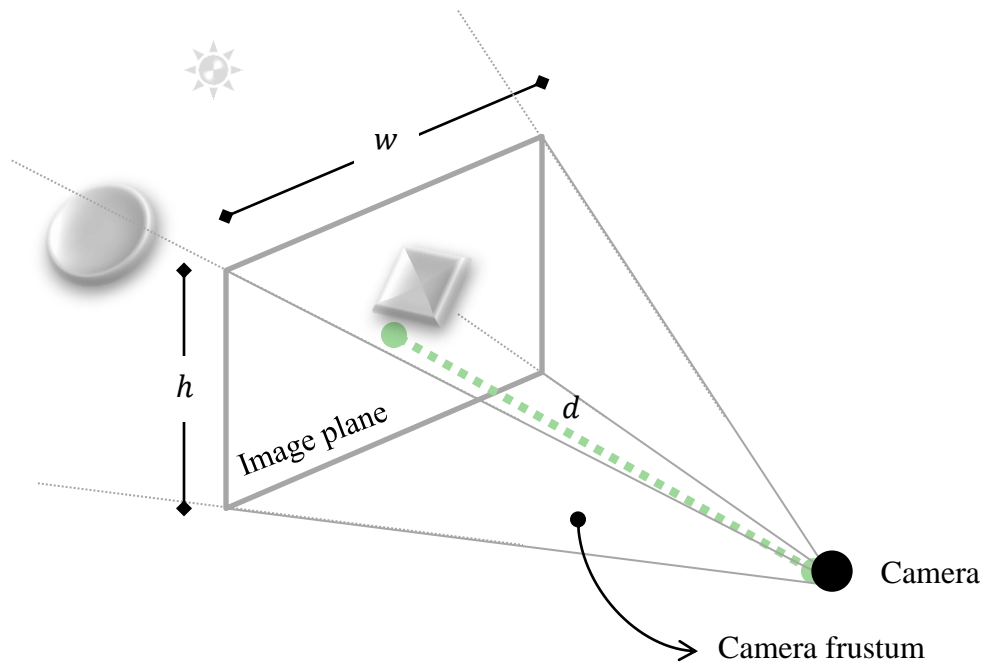


Figure 2: 3D scene model.

2.1.2 The Ray Tracing Algorithm

The core algorithm of ray-tracing was presented by Whitted [2], and is illustrated in Figure 3. It represents the fundamental basis for many ray-tracing-based rendering algorithms. Whitted-style ray-tracing produces pleasing effects such as reflections, refractions, transparent surfaces and shadows. Later on, Cook extended this recursive ray-tracing approach to support additional effects such as glossy reflection, illumination by area light sources, motion blur, and depth of field. This extended approach is called distribution ray-tracing [3]. More advanced algorithms were illustrated later on and were capable of computing the complete global illumination within a scene, including indirect illumination and caustic effects [12, 13, 14, 15, 16]. Even though the purpose and supported accuracy of each algorithm is different, the key

point is that they all heavily rely on the core ray-tracing algorithm as their fundamental base.

A ray is defined in the parametric equation $R(t) = O + t \cdot D$, where O is the ray origin and D the ray direction. According to the core ray tracing algorithm, a primary ray proceeds from the camera position to the scene through each pixel of the image plane. The first intersection, with the smallest distance $t_{min} \in [0, \infty)$ between the ray and any 3D surface, is determined and tested for illumination by the light sources, and potential secondary reflection or refraction rays are generated. For each of these secondary rays, the contribution is recursively evaluated in the same way as for primary rays. Then, the corresponding pixel is shaded, depending on the material of the intersected surface. For detailed information on ray tracing, refer to these famous textbooks [17, 18, 19].

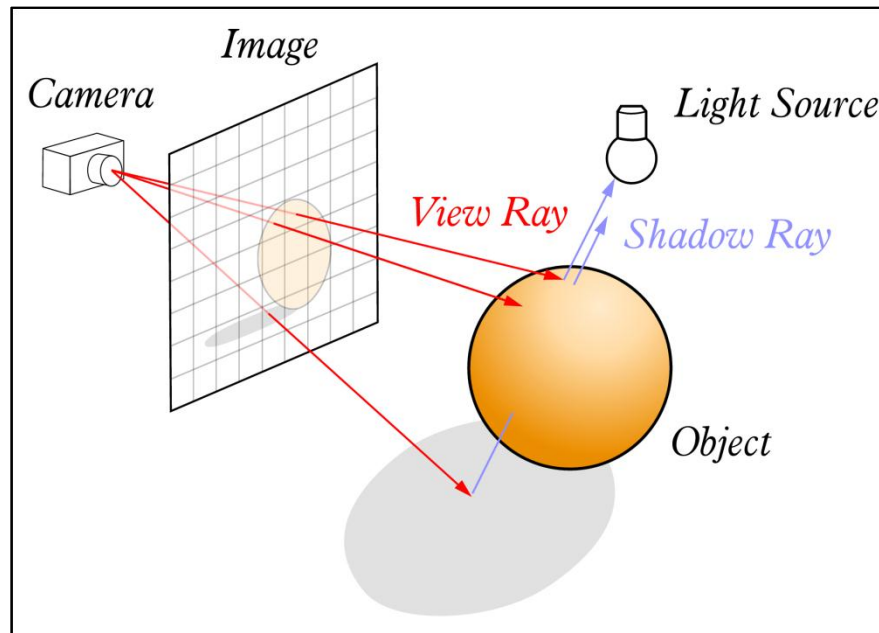


Figure 3: Illustration of the core ray tracing algorithm [20].

Distribution ray-tracing achieves better visual outputs by emitting multiple diverse primary rays per pixel, and multiple diverse secondary rays per intersection point, and then averaging the computed values per each set of rays to, ultimately, illuminate the pixel. Therefore, distribution ray-tracing requires more computation than Whitted-style ray-tracing. To reduce computations, techniques for optimizing ray-tracing, such as acceleration structures and temporal coherence (Section 2.2), have been heavily exploited in the literature.

2.1.3 Acceleration Structures

As shown earlier, at the heart of most ray-tracing based algorithms is the idea of following a ray into a model 3D scene and finding the intersection point between this ray and the nearest object in the scene. Hence, for large scenes, it is important to efficiently exclude surfaces which the ray will not intersect. Otherwise, the ray would test against millions of surfaces before finding the nearest intersection point.

This exclusion is accomplished through a data structure called *acceleration structure*. Broadly, there are two main type of acceleration structures used in ray-tracing: *spatial acceleration structures*, which subdivide the scene into several smaller regions which can be tested efficiently against each ray, and the geometry residing inside these regions that the ray does not interact with can be safely ignored; and *bounding volumes* that surround groups of complex objects in a simple shape, which are tested against each ray, and only if the ray intersects the bounding shape does the ray test against the enclosed geometry. Incorporating acceleration structures in a ray tracer can result in significant performance improvements.

In the literature, several different types of ray-tracing acceleration structures are explored. An excellent survey of several acceleration structures is provided by Walt et al. [6].

2.1.4 Parallel Ray-tracing

Since the color of each pixel is computed independently, ray-tracing algorithms can be easily implemented in a fine-grain level of parallelism. Exploiting this is one way to bring ray tracing closer to interactive execution times. Although this observation was first established by Whitted [1], and many attempts were made towards implementing parallel ray-tracing [15-17], it was only recently that it proved efficient due to the emergence of multi-core CPUs and massively parallel General Purpose GPUs (GPGPUs) (Section 2.4).

2.2 Temporal Coherence

Computer animation can be achieved through displaying synthesized images/frames in rapid succession to create the illusion of motion. The naïve way of producing animations is by synthesizing each image separately. However, the inherent similarity of contents between adjacent synthesized frames can be exploited in order to reduce the associated cost. This similarity is called *temporal coherence*, and can either be between all elements of the scene model states at these frames, or between pixels representing the synthesized image of each of these frames. To distinguish between these two types of temporal coherence, the former is called *object-space temporal*

coherence, while the latter is referred to as *image-space temporal coherence*, which is of interest in this work.

Temporal coherence has been exploited since the early days of computer graphics. For example, the term frame-to-frame coherence was first introduced by Sutherland et al. [21]. It has been used in all techniques of image synthesis, including ray tracing. The following section briefly reviews some of the existing image-space temporal coherence techniques. For object-space temporal coherence, refer to this [8] thorough survey presented by Scherzer et al.

2.2.1 Image-Space Temporal Coherence

Image-space temporal coherence can be adapted in all image synthesis techniques, including rasterization, ray tracing and radiosity (see [8]). One of the earliest adaptations of temporal coherence in ray traced animations was presented by Badt [11], where he introduced the *reprojection algorithm*. Reprojection is a key concept incorporated in almost all later developments of image-space temporal coherence techniques in ray tracing. It involves moving the pixels in one image of an animation to their correct position in the second, and cleaning up the image by recalculating only those pixels whose value is unknown or in question after the transformation, as we will discuss in the next section. Badt reported a speedup of 2.4 in rendering the second image. However, his technique was capable of computing diffuse shading (none view-dependent) only. Thereafter, Adelson and Hodges [16] extended this approach to ray tracing of arbitrary scenes, incorporating other view-dependent

sources of illumination. Although the results of their work exhibited little to no noise, their technique was slow and sequential in nature.

Later, Bishop et al. [6] introduced *frameless rendering*. Here, the concept of frame-based rendering is abandoned and, instead, a set of randomly sampled pixels are progressively rendered based on the most recent input, and gets immediately updated. Due to the delay introduced when the selective pixels are rendered, this method suffered from significant noise artifacts. This method was later improved by Dayal et al. [22] by adaptively biasing the sampled pixels towards the regions of change in scene objects, and this resulted in a relatively substantial reduction of noise artifacts. Still, this adaptive frameless rendering technique, although fast, suffers from noticeable noise.

Walter et al. [23] introduced another technique for achieving interactive framerates in ray-traced animations. The technique decouples the rendering and the display processes to enable high interactivity, and utilizes a point based structure, called *the render cache*, that stores intersection positions and shading values for previous frames in order to reproject them in the current frames. Sampling heuristics and spatio-temporal image coherence are used to refine the reprojections. Later, the authors extended the refinement with predictive sampling and interpolation filters [24]. Lastly, both Edgar Velázquez-Armendáriz et al. [25] and Zhu et al. [26] proposed an accelerated implementation of the render cache on modern GPUs. Yet, the render cache technique suffers from conspicuous artifacts in the produced animations.

2.2.2 Image-Space Temporal Coherence in Stereoscopic Ray Tracing

To create a ray-traced stereo images pair, slightly different views of the same scene must be rendered, potentially doubling the required work. However, the stereo pair is temporally coherent to a high extent.

Adelson and Hodges [27] were the first to exploit temporal coherence to produce the second view image of a stereo pair rendered using ray-tracing. Their work was based on Badt's reprojection algorithm [11], where the pixels generated in the left image are reprojected to the right image, and pixels of reprojection errors are ray-traced. Also, their technique was only limited to render diffuse shading (effects that are not affected by changing the position of the camera). At a later development [28], they extended their technique to render precise specular highlights, resulting in the first mature temporally coherent stereoscopic ray-tracing. Adelson and Hodges reported a speedup of 92% when rendering the right view using their technique. Following is a brief description of their technique.

Assume, in a standard monoscopic scene, a perspective projection is used to project the 3D scene model objects onto the image plane with a camera at position $C = (0, 0, -d)$ for an image plane located at $z = 0$, as depicted in Figure 4. Given a point $P = (x, y, z)$ in the scene, its corresponding image plane position $I_p = (x_p, y_p)$ is:

$$x_p = \frac{x \cdot d}{z + d} \tag{1}$$

And,

$$y_p = \frac{y \cdot d}{z + d}$$

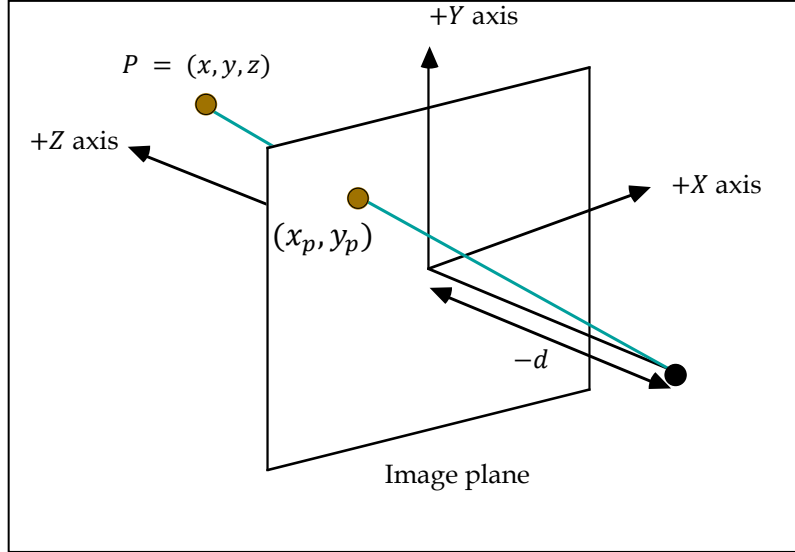


Figure 4: Monoscopic Perspective Projection.

For stereoscopic scenes (Section 2.3.2), as portrayed in Figure 5, two different projections, one per each camera, of the scene are required. Each of these cameras will have a different position; both horizontally displaced by the interaxial distance e . Therefore, the left camera is located at $C_l = (-e/2, 0, -d)$, and for the right camera $C_r = (e/2, 0, -d)$. A point $P = (x, y, z)$ in the scene is projected twice, one per each image plane of each camera, where the corresponding coordinates in the left image plane is $I_{pl} = (x_{pl}, y_{pl})$, such that

$$x_{pl} = \frac{x - z \cdot e/2}{d + z} \quad (2)$$

And,

$$y_{pl} = \frac{y \cdot d}{d + z} \quad (3)$$

And the corresponding image plane coordinates of P for the right camera is $I_{pr} = (x_{pr}, y_{pr})$, where

$$x_{pr} = \frac{x + z \cdot e/2}{d + z} \quad (4)$$

And,

$$y_{pr} = \frac{y \cdot d}{d + z} \quad (5)$$

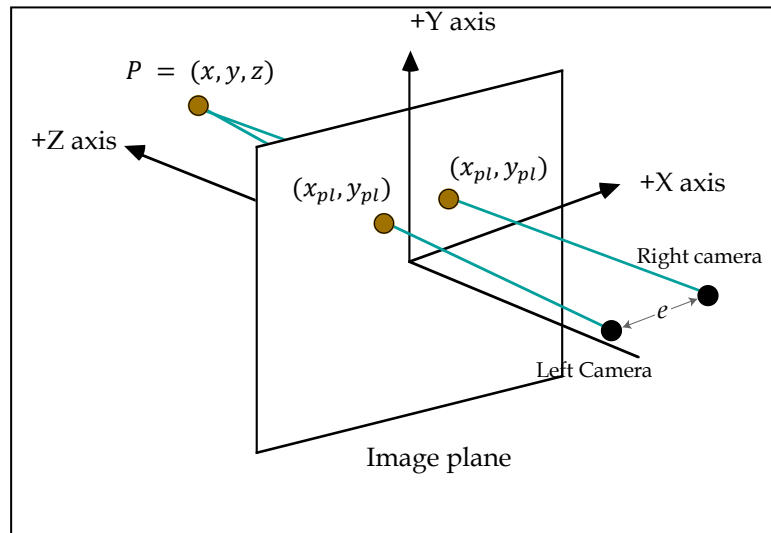


Figure 5: Stereoscopic perspective projection.

To put it in a matrix form:

$$\begin{bmatrix} x_{pl} \\ y_{pl} \\ 0 \end{bmatrix} = M_l \cdot P \quad (6)$$

And,

$$\begin{bmatrix} x_{pr} \\ y_{pr} \\ 0 \end{bmatrix} = M_r \cdot P \quad (7)$$

Where,

$$M_l = \frac{1}{d+z} \begin{bmatrix} 1 & 0 & -\frac{e}{2} \\ 0 & d & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (8)$$

And,

$$M_r = \frac{1}{d+z} \begin{bmatrix} 1 & 0 & \frac{e}{2} \\ 0 & d & 0 \\ 0 & 0 & 0 \end{bmatrix} \quad (9)$$

Also, notice that if the left image plane position $I_{pl} = (x_{pl}, y_{pl})$ was computed, we can compute the right image positions $I_{pr} = (x_{pr}, y_{pr})$ as follows:

$$x_{pr} = x_{pl} + \frac{e \cdot z}{d+z} \quad (10)$$

With $y_{pl} = y_{pr}$. In other words, a point P will move horizontally between the views by a distance dependent on its depth z , the distance d from the cameras position to the projection plane, and the interaxial distance e between the two cameras positions. This transformation from I_{pl} to I_{pr} is called the *reprojection function*, and can be formalized as:

$$\begin{bmatrix} x_{pr} \\ y_{pr} \\ 0 \end{bmatrix} = \text{rep}(I_{pl}, z) = \begin{bmatrix} 1 & 0 & \frac{e \cdot z}{d + z} \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_{pl} \\ y_{pl} \\ 0 \end{bmatrix} \quad (11)$$

Note that the reprojection does not yield a one-to-one correspondence between pixels in the two image planes. Moreover, the reprojection function produces a real valued x_{pr} position for the reprojected pixel, which should be rounded to an integer value to be positioned correctly in the right image. This causes small errors in the color of the reprojected pixel, as opposed to the fully ray traced pixel (see Section 4.5.2).

M_l and M_r represent world-to-image-space transformations, used to transform the model scene points for the left and right views of a stereoscopic scene, respectively; where the left image is rendered using M_l and the right image is rendered using M_r . To incorporate the reprojection technique, the left image I_l is fully ray-traced to be generated, and the intersection position for each primary ray is recorded per pixel in the set W_l . Then, to generate the second image, all the recorded positions of the left image W_l are transformed by the reprojection function to calculate pixel locations on the right image. As described in [28], and as shown in Figure 6, there are three possible reprojection errors:

- *Overlapped pixel problem*: occurs when multiple pixels from one image reproject onto the same pixel in the other. In this case, the reprojection with the maximum z value is chosen to be the correct reprojection.
- *Missing pixel problem*: takes place when no reprojections occur at one pixel in the right image. This can be solved by fully ray-tracing the missed pixels.

- *Bad pixel problem*: occurs when two horizontally adjacent pixels in one image reproject to nonadjacent positions, producing a gap of more than one pixel. Pixels on this gap are questionable and constitute bad pixel problem.

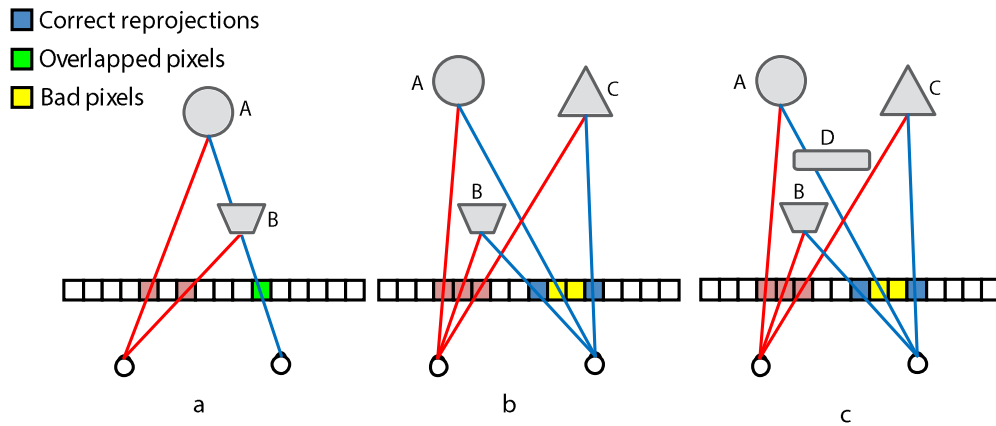


Figure 6: Reprojection errors. (a) Overlapped pixels. (b) and (c) Bad pixels. [29]

In order to rule-out reprojection errors, the “left image is ray traced scan-line by scan-line from left to right. The status of all right image pixels I_r is set to *NoHit* initially. As the left image pixels I_l are ray traced, they are reprojected to the right image. Reprojected pixels in the right image are marked as *Hit*. If gaps are detected between any adjacent reprojected locations, the gap is marked as *NoHit*. After a scan-line is done, pixels marked as *NoHit* are ray traced for the right image”[29]. This algorithm is shown in Algorithm 1.

```

For each scan line  $j$  in the both images,  $0 \leq j < w$  do
  For each pixel  $I_{ri}$  in scan-line  $j$  of the right image,  $0 \leq i < w$  do
    hitStatus[ $I_{ri}$ ] := NoHit
  End for

```



```

oldR := -1
For each pixel  $I_{li}$  in in scan-line  $j$  of the left image,  $0 \leq i < w$  do
  Trace a ray through  $I_{li}$ :
    intersec[ $I_{li}$ ] := Intersection Point or Miss
    norm[ $I_{li}$ ] := Intersection Normal or Miss
    color[ $I_{li}$ ] := Compute_color(intersec[ $I_{li}$ ], norm[ $I_{li}$ ], LeftCamPos)

  rep[ $I_{ri}$ ] :=  $M_r \cdot$  intersec[ $I_{li}$ ]
  If rep[ $I_{ri}$ ] <  $w$  then
    If oldR - rep[ $I_{ri}$ ] > 1 then
      For each pixel  $I_{rk}$  in the right image, oldR <  $k$  < rep[ $I_{rk}$ ] do
        hitStatus[ $I_{rk}$ ] := NoHit
      End for
      hitStatus[ $I_{ri}$ ] := Hit
      color[ $I_{ri}$ ] := Compute_color(intersec[ $I_{li}$ ], norm[ $I_{li}$ ], RightCamPos)
      oldR := rep[ $I_{ri}$ ]
    Else
       $I_{ri} := I_{li}$ 
      oldR :=  $w - 1$ 
    End if
  End if
End for
End for

```

Algorithm 1: Stereoscopic Reprojection Algorithm.

The technique introduced by Adelson and Hodges is sequential by nature, and there is only one attempt in the literature to parallelize it, authored by Es and Isler [29]. Although Es and Isler's parallel technique was implemented on the GPU, the level of parallelism in their implementation is not fine enough to harness the full potential of the GPU; since they chose the obvious way of parallelizing the reprojections: parallel

scan-lines processing. Our implementation goes at a finer level of GPU- and ray-tracing-friendly parallelism; pixel level, therefore it is expected to outperform their technique.

2.3 Stereoscopy

Several depth cues can enable depth perception in 2D scenes, images and paintings, including:

- Relative size: objects of known sizes look smaller the farther away they are.
- Lightening and shadows: closer objects look brighter, distant ones dimmer.
- Perspective: the farther away the object, the smaller it looks, and parallel lines recede to a vanishing point.
- Occlusion: closer objects occlude farther ones.
- Haze: distant objects tend to diminish and look blurry.
- Motion parallax: objects of same speed seem to move faster when closer to the viewer.

As portrayed in Figure 7, these cues are called the *monocular depth cues*, and are the basis for the perception of depth in all 2D displays. Artists of renaissance era had realized that, due to *retinal disparity*, each human eye perceives a slightly different image than the other, resulting in a depth cue that was impossible for a painter to portray in a single canvas; the *stereoscopic depth cue*. *Stereopsis*, the process of perceiving depth produced by retinal disparity, was only well established in the 18th century. Understanding this concept led to the invention of the *stereoscopy* technique,

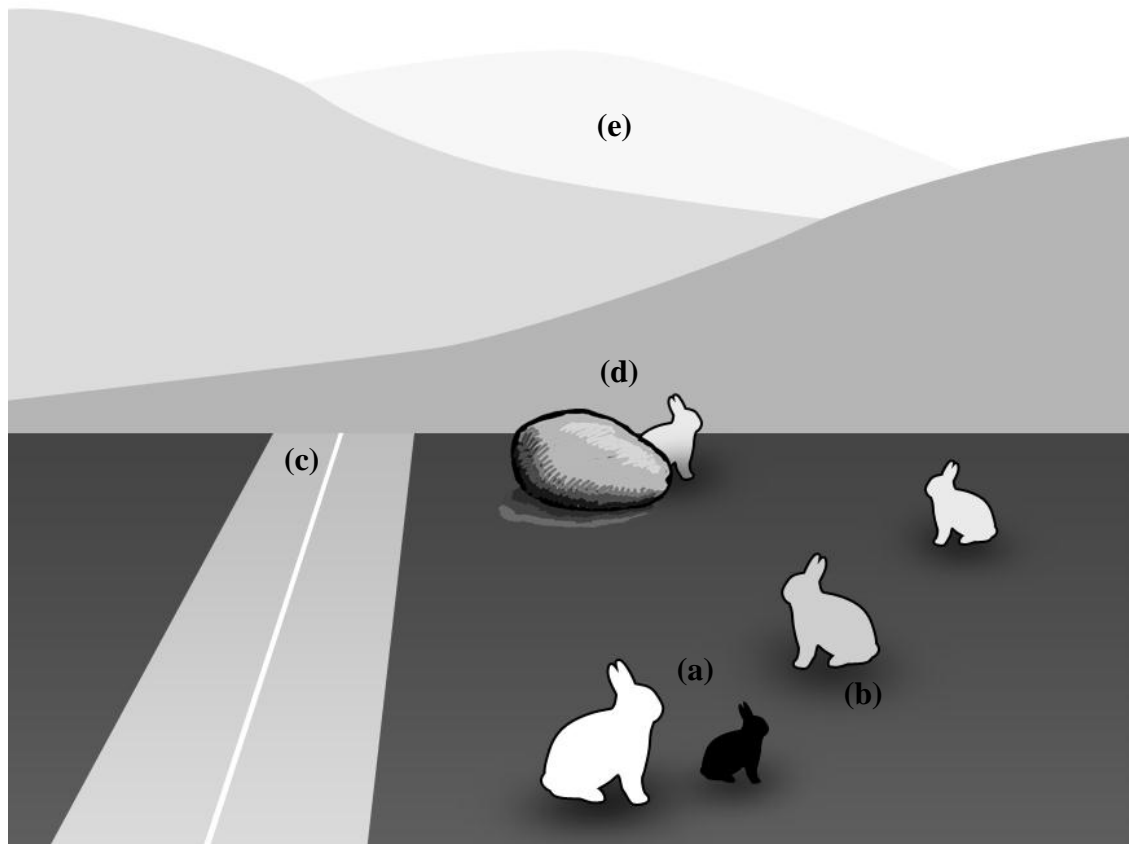


Figure 7: Monocular depth cues. (a) Relative size. (b) Lights and shadows. (c) Perspective. (d) Occlusion. (e) Haze.

where an added depth cue of a scene is enhanced by presenting two offset images, called stereo images pair or *stereo pair*, separately to the left and right eyes of the viewer.

2.3.1 Stereoscopic Displays

Stereoscopic displays (commercially known as 3D displays) utilize stereoscopy to introduce the stereoscopic depth cue to the viewer. The main aim of all stereoscopic displays is to present each eye of the viewer with the corresponding image of a stereo

pair. Multiple techniques are employed by various stereoscopic displays in order to achieve that, and each has its own advantages and disadvantages. Following is a list of the popular techniques used in modern stereoscopic displays. For a thorough description of most of existing techniques, refer to [30, 31].

- Anaglyph: where a stereo pair in which the right image of a scene, usually red in color, is superposed on the left image of a contrasting color to produce a new image, called the *anaglyph image*, which establishes a stereoscopic depth cue when viewed through correspondingly colored filters in the form of glasses. Typical contrasting colors used in anaglyph images and their corresponding filter glasses are red/blue, red/cyan and red/green. This technique is cost-effective since it requires no special hardware except for the cheap filter glasses, but it comes at the cost of degrading the original colors of the scene, and suffers from crosstalk, whereby each eye perceives a small portion of the image targeting the other eye, limiting the ability to successfully fuse the stereo pair in the brain and hence reducing the overall perceived quality [32].
- Polarization: in which the two images of a stereo pair are superimposed in the display through different polarizing filters. The viewer wears low-cost glasses which contain a pair of different polarizing filters. As each filter passes only that light which is similarly polarized and blocks the light polarized in the opposite direction, each eye sees a different image. Although the polarized glasses are cheap, the equipment required to generate polarized images is expensive. Moreover, crosstalk can occur if the viewer is not positioned

correctly in front of the display. It also suffers from reduced brightness due to the polarized filters [30].

- Shutter glasses: this technique requires a special display that alternately switches between left and right images of a stereo pair. The viewer wears special glasses, in which the lenses alternately darken over one eye, and then the other, in synchronization with the display. This technique produces the best output at the cost of expensive display equipment and viewing glasses.
- Autostereoscopic: the main advantage of this method is that no special viewing equipment is required. Autostereoscopic techniques employ a wide range of technologies, mostly including lenticular lenses or parallax barriers. These techniques redirect each of the displayed stereo pair to the intended eye. However, they also suffer from limited viewing positions [31].

In this work, we use the anaglyph technique to produce stereoscopic images. Several methods has been proposed to do so [33, 34, 35], of which we employ the Dubois algorithm [35], due to its relative efficiency.

2.3.2 Rendering Stereoscopic Images

A monoscopic 3D scene model contains one camera in its description, and produces an image exhibiting only monocular depth cues when rendered. To render a stereo pair, the rendering algorithm has to be modified to account for two horizontally offset cameras (Figure 8(b)), each with its own image plane, such that each camera produces the corresponding image of a stereo pair when the scene is rendered. The stereo pair is then presented to the viewer based on the stereo display in use.

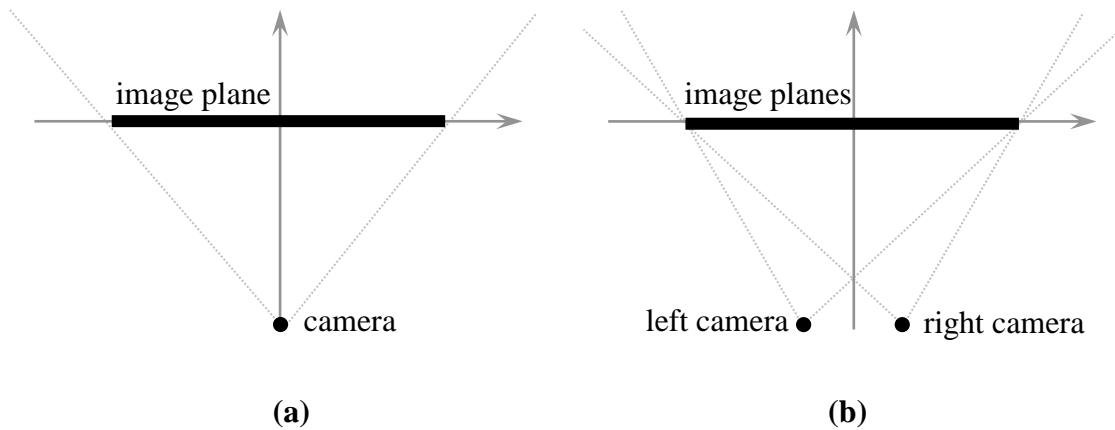


Figure 8: (a) Monoscopic scene. (b) Stereoscopic scene.

Besides the attributes used to describe a monoscopic scene model (including image plane width w , distance from camera to image plane d and camera viewing frustum), a *stereoscopic 3D scene model*, or a *stereo scene*, specifies a value for the horizontal distance between its cameras, called the *interaxial distance*, denoted by e . In principle, both d and e values can be specified arbitrarily. However, some guidelines shall be followed as to produce a correct ‘fusible’ stereo pair. Following is a discussion of the most important guidelines.

- If we denote the interocular distance between the human eyes (6.5 cm on average) by e_{real} , and the width of the target stereo display by $w_{display}$, then we should set a value for e such that:

$$\frac{e}{w} < \frac{e_{real}}{w_{display}} \quad (12)$$

Otherwise, the produced stereo pair will not converge on the eyes of the viewer, and the experience might become painful [36].

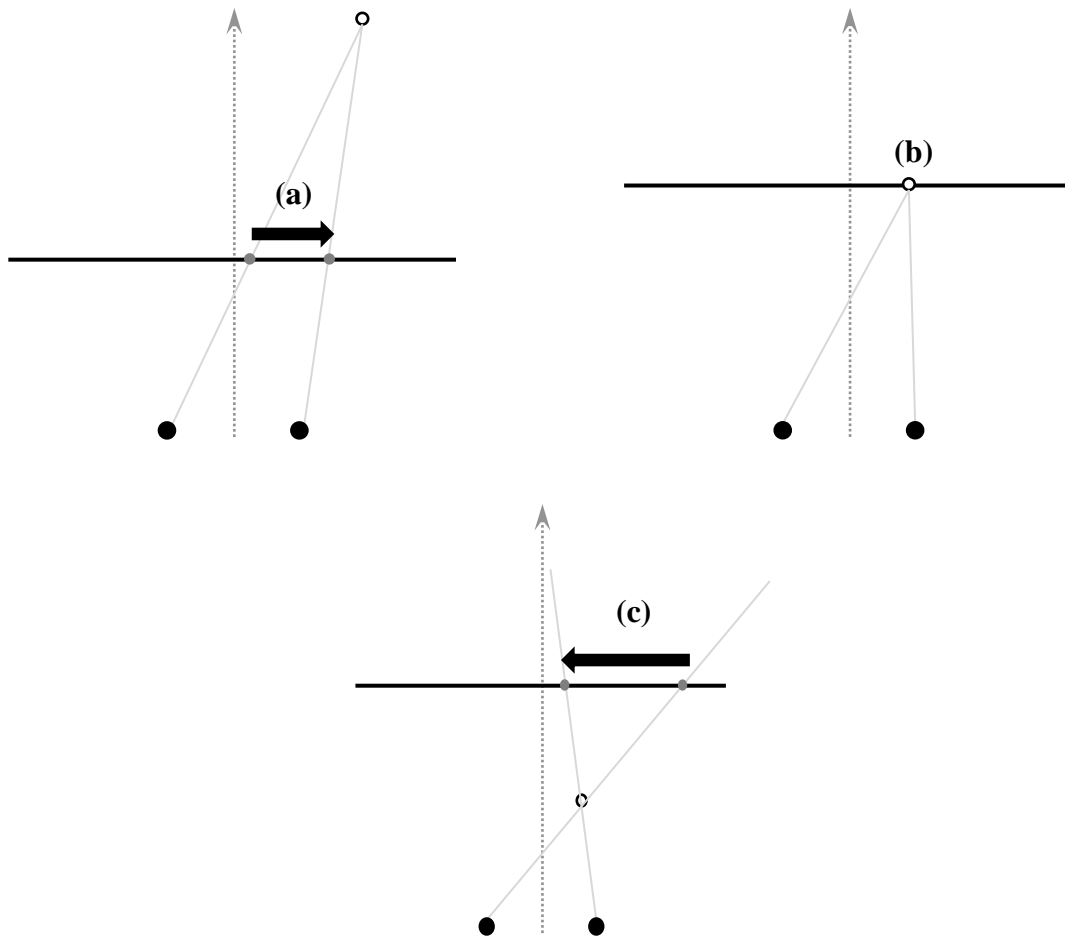


Figure 9: (a) Positive parallax. (b) Zero parallax. (c) Negative parallax.

- The horizontal parallax p is the distance between the projections of a 3D surface in the scene to the left and right image planes. As in Figure 9, p can take a negative, a zero or a positive value. When displaying the stereo pair, surfaces with a positive parallax will appear to be in the display, and surfaces of zero parallax will appear to be at the display, while surfaces of negative parallax will seem to float in front of the display. A negative parallax equal to the interaxial distance e occurs when the projected surface is at a distance of

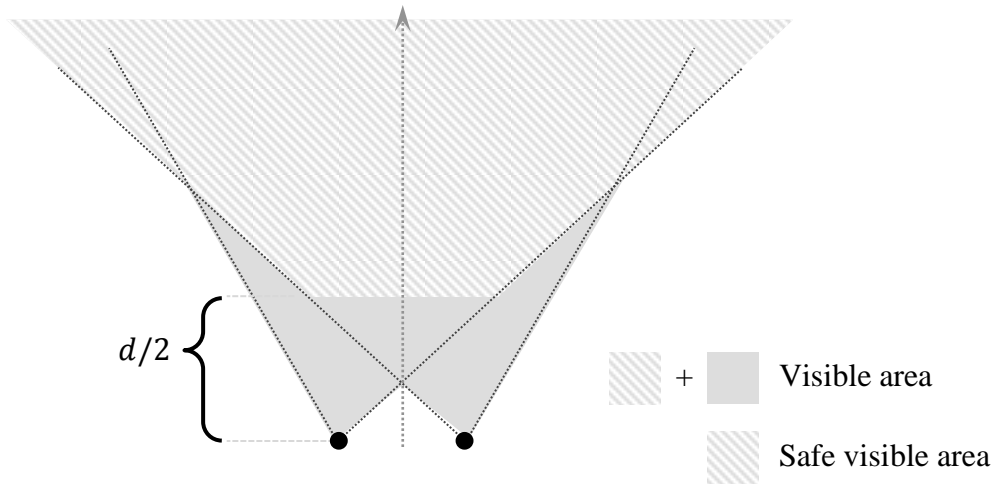


Figure 10: Safe visible area for inserting 3D surfaces into a stereo scene.

$d/2$ from the center of the cameras position. As the surface moves closer to the viewer, the negative parallax diverges to infinity, and this should be avoided, since the projected surface will become impossible to fuse in the viewer’s brain [37]. Based on this restriction, Figure 10 shows the safe visible area in which 3D surfaces can be inserted in a stereo scene.

- Many methods exist for setting up both cameras frustum in a stereo scene. Nevertheless, the vast majority of them introduce discomfoting stereo pair to the viewer [38]. The correct way of building up the cameras frustum is portrayed in Figure 8(b), where each camera’s point of focus is parallel to the z-axis, and both image planes coincide. This methods is called the *off-axis projection* [37].

In summary:

- Keep the interaxial distance e value low.
- Position all 3D surfaces in a scene at a distance $d_{surface} \geq d/2$ from the center of the cameras position.
- Build cameras frustum based on off-axis projection.

2.4 Massively Parallel Graphics Processing Units

Recently, the performance of GPU has been increasing much faster than the CPU. Modern GPUs substantially outperform the CPUs, especially in floating point operations. As Figure 3 illustrates, the number of executed floating-point operations per second (FLOPS) in modern GPUs vastly exceeds that of CPU. Besides the computing capability, GPUs also have their own memory system, which offers substantially higher bandwidth than ordinary CPU systems. As a result, GPUs have been heavily used in research spanning multiple areas in the last decade, and have been shown to deliver orders-of-magnitude gains in performance over optimized CPU applications[39].

2.4.1 Taxonomy

GPUs are massively parallel processors, and may contain hundreds of cores that can execute thousands of threads. They fall into the *Single Instruction, Multiple Data* (SIMD) family in the famous Flynn's taxonomy of parallel processors [40]. Generally, all SIMD processing units execute the same instruction at a given time, where each processing unit can operate on a different data element.

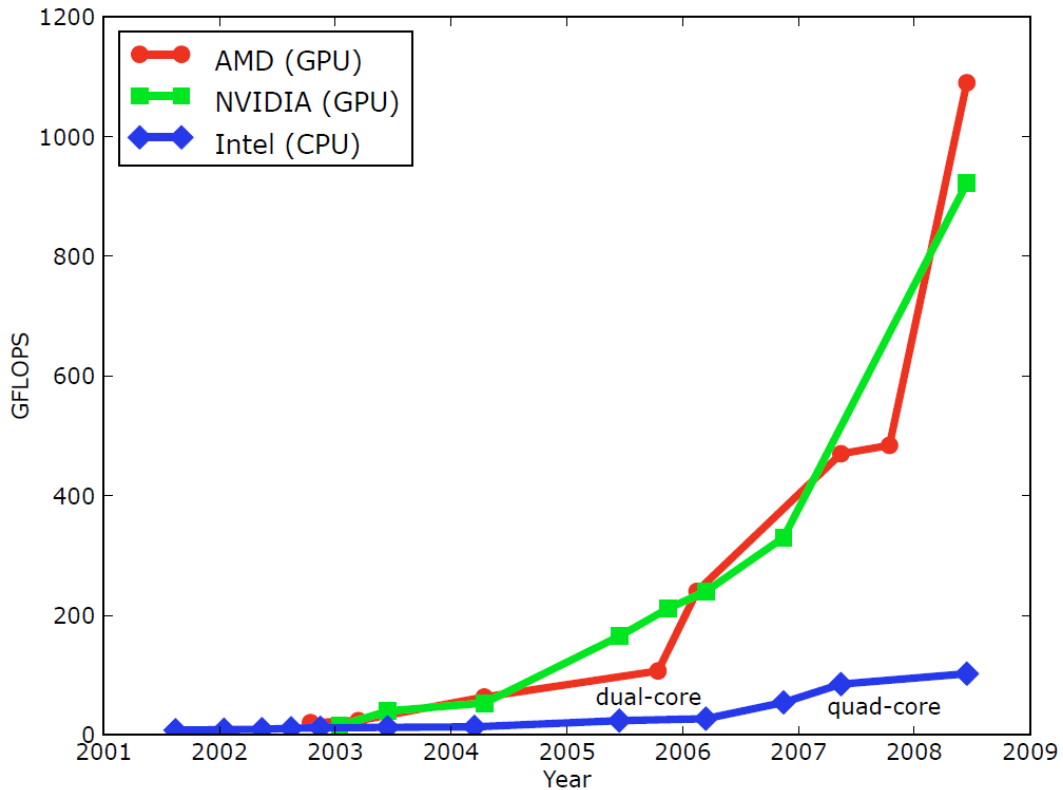


Figure 11: GPU vs. CPU performance trends in GFLOPS (10⁹ FLOPS) [41].

GPUs are designed to exploit problems that can be implemented at a fine-grain level of parallelism, such as graphics and image processing problems. Although GPUs are characterized by high throughput and performance, they add another layer of complexity for code development. For example, simple branching operations in GPU code can considerably slow down the performance. Also, moving data blocks back and forth between CPU and GPU is considered a bottleneck [42].

2.4.2 Trending Architectures

Early GPUs had sophisticated programming languages. However, new simpler GPU programming interfaces has emerged recently, including NVidia *Compute Unified*

Device Architecture (CUDA)[43], *OpenCL*[44] and Microsoft *DirectCompute*. The programming models of these interfaces are conceptually similar. They provide abstract programming interfaces that include functions for managing memory allocations, performing CPU-GPU memory transfers, compiling GPU programs – *kernels* – and launching them. Once a kernel is launched, many threads containing identical code to it are spawned and executed in the GPU cores.

Existing GPU programming interfaces are general purpose, and developing efficient ray tracers on them can be quite a challenge. Nevertheless, NVidia provides a platform for accelerating the development of ray tracing applications, called NVidia OptiX [45]. We use this platform for our experiments.

2.4.2.1 NVidia OptiX

The NVIDIA OptiX platform is a ray tracing engine that is built upon CUDA, and is intended to accelerate the development of ray tracers on modern GPUs. OptiX offers many features, such as out-of-the-box acceleration structures, threads scheduler and various ray tracing helper functions. OptiX also has its own programming model. A thorough description of OptiX can be found in [46].

CHAPTER THREE

METHODOLOGY

3.1 Parallel Reprojection

As mentioned in Section 2.2.2, reprojection introduces three problems: missed pixel problem, overlapped pixel problem and bad pixel problem. Adelson and Hodges proposed a strictly sequential processing of each scan-line in order to resolve them. However, since we are targeting implementing reprojection in massively parallel GPUs, ray tracing and reprojection are better done at a finer level of parallelism: a thread per pixel. This will render the order of execution unguaranteed; therefore we devise new resolution mechanisms to the reprojection problems.

We assume a stereo scene setup as shown in Figure 5, with fixed stereo cameras positions¹ at $(-e/2, 0, -d)$ and $(e/2, 0, -d)$. Our algorithms proceed as follows. First, all pixels I_{li} in the left image are fully ray-traced in parallel. Once the pixel value is determined, its scene depth D_{li} – related to the left camera – is stored. Then, the pixel is reprojected to its corresponding position to the right image. The following sections thoroughly discuss our resolution mechanisms to rule out the reprojection errors in parallel. Table 1 summarizes those mechanisms.

¹ This algorithm can be easily modified to handle arbitrary positioning of the stereo cameras.

Method	Missed Pixel Resolution	Overlapped Pixel Resolution	Bad Pixel Resolution
Adelson and Hodges / Es-Isler (Sequential)	Fully ray trace	Proceed sequentially from left to right	
Buffer-Based (Parallel)		Store all reprojected pixel in a 3D buffer, prevent race condition.	Cast a length-restricted ray
Atomic-Based (Parallel)		Use atomic operations to prevent race condition.	

Table 1: Comparison between different mechanisms for resolving reprojection problems.

3.1.1 Missed Pixel Resolution

We use the same strategy employed by the original authors to resolve this problem; by fully ray tracing the missed pixels in the right image, done in parallel.

3.1.2 Overlapped Pixel Resolution

Since the reprojection function does not yield a one-to-one correspondence between pixels in both image planes, multiple pixels from the left image reproject to the same position. Executing in parallel, this will introduce a race condition. Thus, we propose two different approaches to resolve this case of race condition, one of which assumes that the underlying parallel hardware provides atomic operations, while the other makes no assumptions about the hardware in use, but exploits a property exhibited by stereoscopic scenes when equipped with reprojection. We call the latter method *Buffer-*

Based Overlapped Pixels Resolution, and the former *Atomic-Based Overlapped Pixels Resolution*.

3.1.2.1 Buffer-Based Overlapped Pixels Resolution

In this approach, all reprojected pixels from the left image, alongside their original horizontal position x_{li} in the left image, are stored in an intermediate 3D buffer of size $w \times h \times t$, such that no value is overwritten, and mutual exclusion is guaranteed. Then to find the correct pixel value in the right image, this buffer is traversed at each corresponding pixel position towards the depth, picking the pixel with the maximum related value of x_{li} .

To determine the optimal depth t of the 3D buffer, this approach draws on the following lemma, which states that the maximum writes to a single pixel position in the right image when using reprojection is upper bounded by $r \cdot e$, where the closet surface in the stereo scene is at a distance d/r from the center of cameras position, for some real value r . It can be concluded from the discussion in Section 2.3.2 that, so as to assure a comfortable viewing experience, the maximum value for r in most stereo scenes is set to 2, and therefore, their corresponding optimal depth of the 3D buffer is $t = 2e$. Notice there are rare cases in which this maximum can be reached is illustrated in Figure 12, where a geometrical object, located at a distance $d/2$ in front of the right camera, extends to an infinite depth.

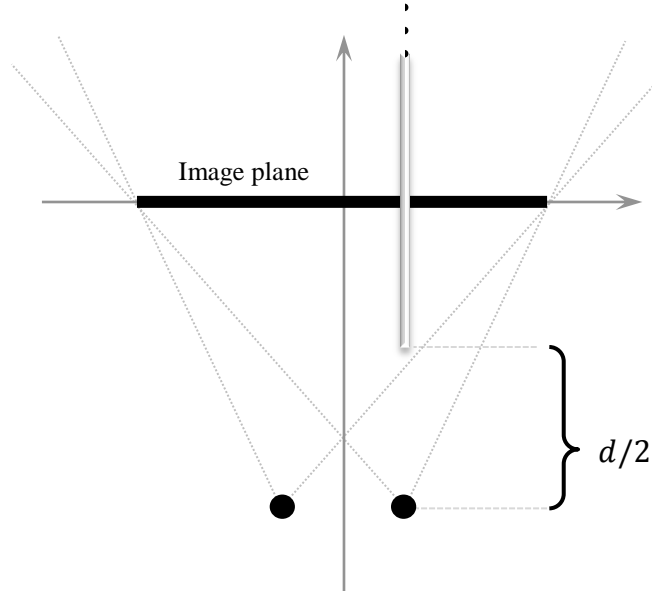


Figure 12: A possible case when the reprojections to a single position reach a maximum.

This approach is expected to underperform the atomic-based approach discussed next, especially for higher values of e ; since each pixel-generating thread in the right camera has to traverse over a vector of size $2e$ of the 3D buffer; even when the vector contains no values.

Lemma 1: Assuming that the nearest surface in a stereo scene is positioned at a distance r away from the center of cameras positions, then the maximum number of pixels in the left image reprojecting to the same pixel position in the right image is equal to $r \cdot e$. Formally,

$$\|\{\forall I_{li} \in I_l \mid \text{rep}(I_{li}, z_i) = J\}\| \leq r \cdot e \quad (13)$$

Where I_{li} and J are pixel positions in the left and right images, respectively, and $\text{rep}(a, b)$ is the reprojection function.

Proof:

Assume a scene, as in Figure 13, with fixed left and right cameras positioned at $(-e/2, 0, -d)$ and $(e/2, 0, -d)$, respectively, such that $d \geq 0$ and $e \geq 0$. Moreover, the nearest surface in the object space is positioned at a distance of d/r from the center of both cameras position for some real number $r > 0$. Assume also two points $P_i = (x_i, y_i, z_i)$ and $P_j = (x_j, y_j, z_j)$ in the object space that correspond to different left-image plane positions $I_{li} = (x_{li}, y_{li})$ and $I_{lj} = (x_{lj}, y_{lj})$, respectively, such that $z_i \geq \frac{d}{r} - d$, $z_j \geq \frac{d}{r} - d$, $x_{lj} > x_{li}$, and

$$\text{rep}(I_{li}) = \text{rep}(I_{lj}) \quad (14)$$

Therefore,

$$x_{li} + \frac{e \cdot z_i}{z_i + d} = x_{lj} + \frac{e \cdot z_j}{z_j + d} \quad (15)$$

$$x_{lj} - x_{li} = \frac{e \cdot z_i}{z_i + d} - \frac{e \cdot z_j}{z_j + d} \quad (16)$$

Set $x_{lj} - x_{li} = n$. Here, n represents the distance between two pixels in the left image such that their reprojection to the right-image plane is the same. At the extreme case, all the pixels in the set $\{I_{lk} = (x_{lk}, y_{lk}) | x_{li} \leq x_{lk} \leq x_{li} + n, y_{lk} = y_{li} = y_{lj}\}$ reproject to the same pixel position in the right image I_r for the maximum value of n . In this case, n represents the maximum number of pixels in the left image that reproject to the

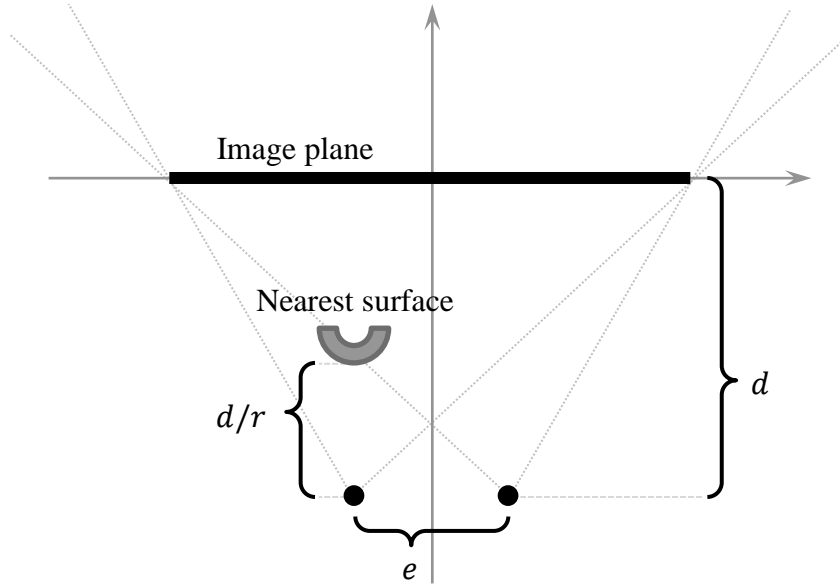


Figure 13: Stereo scene parameters.

same pixel position in the right image. We can compute the maximum value for n as follows:

$$\max n = \max \left(\frac{e \cdot z_i}{z_i + d} - \frac{e \cdot z_j}{z_j + d} \right) \quad (17)$$

Given that $z_i \geq \frac{d}{r} - d$ and $z_j \geq \frac{d}{r} - d$. Solving this equation yields the following conclusion:

$$n \leq r \cdot e \blacksquare \quad (18)$$

3.1.2.2 Atomic-Based Overlapped Pixels Resolution

Another proposed resolution to the race condition introduced by reprojection is through employing atomic operations, ensuring only one thread accessing the corresponding

right image position when overwriting. Overwrites take place only if the reprojecting pixel has a higher corresponding position x_{li} than the residing value in the right image.

This approach is best used when the following holds:

- The underlying parallel architecture provides atomic operations.
- The associated penalty of using atomic operations is not substantial. This can be confirmed if using this approach proves to perform better than the buffer-based approach.

Threads writing atomically into one memory position are processed linearly, and thus this approach introduces a slight linear overhead. Since, as shown in Lemma 1, a maximum of $r \cdot e$ threads can write to the same pixel position at rare cases, the linear overhead introduced by this approach is expected to be $o(r \cdot e)$ per pixel position. Therefore, the runtime of this approach is upper-bounded by the buffer-based approach runtime.

3.1.3 Bad Pixel Resolution

Figure 14 illustrates the case when the bad pixel problem occurs. Sequentially processed, pixel K will be marked as a bad pixel in Adelson and Hodges implementation, because it was reprojected onto a gap between originally adjacent pixels; L and M. Bad pixels are fully ray traced once detected.

In our parallel implementation, we mark all reprojected pixels as potentially bad pixels. Then, per each of them, a ray of restricted length is casted and tested for intersection. The length of this ray is determined by the depth of its corresponding reprojected pixel

in the original left image, D_{li} . If this ray intersects anything on its way, the linked reprojected pixel is discarded and fully ray traced. Otherwise, the reprojected pixel is approved as correct. This approach assumes that tracing a length-restricted ray is efficient; which holds in case of using acceleration structures in the scene.

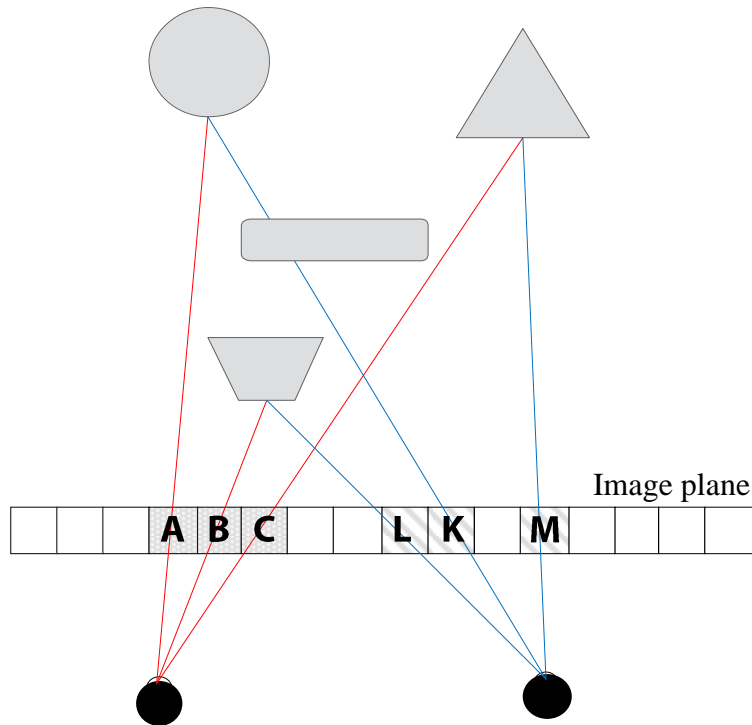


Figure 14: K is marked as a bad pixel and is fully ray traced.

3.2 Complexity Analysis

The following analysis provides an approximation to the runtime of the buffer-based algorithm, which can be generalized as an upper bound to the atomic-based algorithm.

Assume a stereo scene to be ray traced, where the number of pixels is $n = w \times h$, the number of objects is m and the number of lights is l . Assume also an arbitrary

acceleration structure is used in the scene, denoted by AC . Since some traced rays in the right image are length-restricted, their corresponding AC tree will be truncated [47]. As a result, we will use the following notation to differentiate between three possible scenarios of the cost incurred while tracing the rays:

- $AC_f(m, l)$: cost of traversing a full AC tree over m objects for an unrestricted ray, where an intersection is found, and l shadow rays are spawned and traced consequently.
- $AC_t(m, l)$: cost of traversing a truncated AC tree over m objects for a length-restricted ray, where an intersection is found, and l shadow rays are spawned and traced consequently. At the worst case, the length of the restricted ray is set to ∞ , and thus $AC_t(m, l) \leq AC_f(m, l)$ in general.
- $AC_t(m)$: cost of traversing a truncated AC tree over m objects for a length-restricted ray, where no intersection is found and hence no shadow rays are spawned. Even at the worst case, where the ray length is set to ∞ , it is obvious that $AC_t(m) < AC_t(m, l) \leq AC_f(m, l)$, because no shadow rays are traced.

Based on this, it can be easily confirmed that the cost of computing one pixel value in each of the stereo pair, using the naïve approach, is $O(AC_f(m, l))$, and therefore the total time it takes to render one image of the stereo pair is $n \cdot O(AC_f(m, l))$.

Moreover, we can evaluate the total cost of computing the value of one pixel position in the stereo pair, using the buffer-based approach, as follows:

- Left image: the cost for tracing a ray through a pixel is $O(AC_f(m, l))$. Also, each pixel is reprojected to the right image at a constant budget $O(1)$; independent of the input variables. Therefore, the total complexity of generating one pixel is $O(AC_f(m, l))$.
- Right image: to resolve overlapped pixel problem, the 3D buffer is traversed towards the depth at a cost of $O(re)$, which reduces to $O(2e) = O(e)$ for most stereo scenes. Then, one of the following scenarios takes place:
 - No pixel value is found at the 3D buffer (missed pixel), thus the pixel has to be fully ray traced at a cost $O(AC_f(m, l))$.
 - Pixel value is found, but it constitutes a bad pixel. The cost of traversing the length-restricted ray to recompute the pixel value is $O(AC_t(m, l))$.
 - Pixel value is found, marked as a potential bad pixel, but the traced restricted ray confirms that it is not. This costs $O(AC_t(m))$.

Let q_m denote the number of missed pixels, q_b denote the number of bad pixels and $q = q_m + q_b$ denote the number of pixels in error in the right image. The overall cost of finding all pixel values in the right image can be given through the following equation:

$$n \cdot O(e) + q_m \cdot O(AC_f(m, l)) + q_b \cdot O(AC_t(m, l)) + (n - q) \cdot O(AC_t(m)) \quad (19)$$

Since it can be empirically shown that q_m and q_b , and thus q , have smaller values relative to n (refer to [28] for details), most of the pixel-generating threads in the right

image will execute at a cost of $O(e) + O(AC_t(m))$, which can be less than the cost of fully tracing through the pixel at $O(AC_f(m, l))$; especially for scenes with a small value of e and a number of lights $l \geq 2$. Coupled with a dynamic or near-optimal threads scheduler, which executes threads of similar runtime together, the performance gain of the buffer-based approach is expected to outperform that of the naïve approach based on this analysis. Additionally, since the buffer-based approach constitutes an upper bound to the atomic-based approach, which runs at $o(e) + O(AC_t(m))$; the latter is expected to deliver the best performance.

3.3 Kernels Pseudocode

The two kernels, corresponding to each camera, that generate the stereo pair of a scene using the buffer-based resolution technique are presented in Algorithm 2 and Algorithm 3, respectively, while Algorithm 4 and Algorithm 5 describe the kernels that make use of the atomic-based resolution technique. For any technique, their corresponding pair of kernels should be executed one after the other, starting at the first kernel, to generate animations. The logic behind all kernels is dependent on the discussions of Section 3.1.

3.3.1 Buffer-Based Kernels

Kernel LeftCameraRayTrace – Buffer Based

Inputs: I_l , shared left image 2D buffer of size $w \times h$

D_l , shared buffer of size $w \times h$

repBuf, shared buffer of size $w \times h \times 2e$

1. (i, j) = Retrieve thread index in 2D

```

2.  For  $k := 1 \rightarrow 2e$  do
3.     $\text{repBuf}[i, j, k] := (\text{rp} := \text{NiL}, \text{dp} := -1, \text{ip} := \text{NiL})$ 
4.  End for
5.   $I_l[i, j] = \text{Ray trace pixel at position } i, j$ 
6.   $D_l[i, j] = \text{Compute depth of pixel } I_l[i]$  in scene space
7.   $\text{rep}[i, j] = \text{reproject}(I_l[i], D_l[i])$ 
8.  if  $\text{rep}[i, j] \rightarrow x < w$  then
9.     $\text{depth} := i \bmod (2 \times e)$ 
10.    $\text{repBuf}[i, j, \text{depth}] := (\text{rp} := \text{rep}[i, j], \text{dp} := i, \text{ip} := I_l[i, j])$ 
11. End if

```

Algorithm 2: Kernel for generating the left image using the buffer-based resolution.

Kernel RightCameraInfer – Buffer Based

Inputs: I_r , shared right image 2D buffer of size $w \times h$

D_l , shared buffer of size $w \times h$

repBuf , shared buffer of size $w \times h \times 2e$

```

1.   $(i, j) = \text{Retrieve thread index in 2D}$ 
2.   $\text{rb} := \text{Retrieve refBuf}[i, j, k]$  such that  $\text{refBuf}[i, j, k] \rightarrow \text{dp}$  is the maximum
    value in the set  $\{\text{refBuf}[i, j, 1] \rightarrow \text{dp}, \dots, \text{refBuf}[i, j, 2 \times e] \rightarrow \text{dp}\}$ 
3.  If  $\text{rb} \rightarrow \text{dp} < -1$  do
4.     $I_{buf} := \text{Ray trace a ray of restricted length based on } D_l[i, j]$  at pixel
    position  $i, j$ 
5.  Else
6.     $I_{buf} := \text{Ray trace a ray at pixel position } i, j$ 
7.  End if
8.  If  $I_{buf} \neq \text{NiL}$  do
9.     $I_r[i, j] := I_{buf}$ 
10. Else
11.    $I_r[i, j] := \text{rb} \rightarrow \text{ip}$ 

```

12. **End if**

Algorithm 3: Kernel for generating the right image.

3.3.2 Atomic-Based Kernels

Kernel LeftCameraRayTrace – Atomic Based

Inputs: I_l , shared left image 2D buffer of size $w \times h$

D_l , shared buffer of size $w \times h$

iBuf, shared buffer of size $w \times h$, initially all values set to -1

1. (i, j) = Retrieve thread index in 2D
2. $I_l[i, j]$ = Ray trace pixel at position i, j
3. $D_l[i, j]$ = Compute depth of pixel $I_l[i, j]$ in scene space
4. $rep[i, j]$ = $reproject(I_l[i, j], D_l[i, j])$
5. **if** $rep[i, j] \rightarrow x < w$ **then**
6. **Critical section begins**
7. $li := rep[i, j] \rightarrow x$
8. **If** $i > iBuf[li, j]$ **then**
9. $iBuf[li, j] := i$
10. **End if**
11. **End critical section**
12. **End if**

Algorithm 4: Kernel for generating the left image.

Kernel RightCameraInfer – Atomic Based

Inputs: I_l , shared left image 2D buffer of size $w \times h$

I_r , shared right image 2D buffer of size $w \times h$

D_l , shared buffer of size $w \times h$

iBuf, shared buffer of size $w \times h$

1. (i, j) = Retrieve thread index in 2D


```

2.  rb := iBuf[i, j]
3.  If rb <> -1 do
4.     $I_{buffer} :=$  Ray trace a ray of restricted length based on  $D_l[i, j]$  at pixel
      position i, j
5.  Else
6.     $I_{buffer} :=$  Ray trace a ray at pixel position i, j
7.  End if
8.  If  $I_{buffer} <>$  NiL do
9.     $I_r[i, j] := I_{buffer}$ 
10. Else
11.   $I_r[i, j] := I_l[i, j]$ 
12. End if
13. iBuf[i, j] := -1

```

Algorithm 5: Kernel for generating the right image.

CHAPTER FOUR

EXPERIMENTAL RESULTS AND ANALYSIS

Ray tracing on the GPU has proved to be more efficient than on CPU [45, 48]. Therefore, this chapter will discuss the implementation and results of the GPU-based stereoscopic ray tracers and their outcomes with respect to the hypotheses: our algorithms, buffer-based and atomic-based stereoscopic ray tracing, outperform the naïve stereoscopic ray tracing algorithm that generates the stereo pair by fully ray tracing through them. Also, it will be shown that they outperform Es-Isler’s suggested method.

4.1 Ray Tracer Implementations

We implemented five different stereoscopic ray tracing algorithms on the GPU as NVidia OptiX kernels, two of which are based our algorithms (*buffer-based* and *atomic-based* algorithms, sections 3.3.1 and 3.3.2), and the other three are a *naïve* stereoscopic ray tracing implementation that generates the stereo pair by fully ray tracing them, a ray tracer based on the *Es-Isler’s* technique, and an *imaginary ideal* implementation that generates the left image by fully ray tracing it, and generates the right image by merely copying the left image. The naïve implementation serves as the baseline in our benchmarks, while the imaginary ideal implementation sets an imaginary optimal runtime for a stereoscopic ray tracer. All implementations use the

Dubois algorithm, also implemented as an OptiX kernel, to fuse both stereo pair into a single anaglyph image (Section 2.3.2). Table 2 summarizes those ray tracers.

Ray Tracer	Platform	Stereoscopic Ray Tracing Strategy	Output
Naïve	NVIDIA OptiX version 2.5.0	Fully ray trace both stereo pair	Single anaglyph image
Imaginary ideal		Fully ray trace left image, copy left image to right image	
Buffer-based		Using the buffer-based kernels	
Atomic-based		Using the atomic-based kernels	
Es-Isler		Fully ray trace left image, reproject in parallel scan-line by scan-line, fully ray trace missed/bad pixels in the right image	

Table 2: Benchmarked ray tracers.

All implementations make use of the optimization techniques for GPUs. Specific to OptiX, this translates to minimizing the actual branching calls and lowering the transactions between CPU and GPU. Moreover, the implementations utilize the Split Bounding Volume Hierarchies (SBVH) [49] ray tracing acceleration structures (Section 2.1.3) offered by OptiX, as to increase performance based on the analysis in Section 3.2. OptiX provides an out-of-the-box scheduler, so we leave the scheduling of threads to it.

4.2 Testbeds

To evaluate the performance of each ray tracer, we set up two different testbeds. The first, Testbed-1, is a workstation equipped with Intel Xeon processor, 88 GB of RAM and NVidia Quadro Plex 7000 graphics card. Testbed-2 is a laptop equipped with Intel Core i7-2640M processor, running at 2.80 GHz with 8 GB of RAM. It has an embedded NVidia Geforce GT 525M graphics card with dedicated 2 GB of VRAM. These specifications are summarized in Table 3.

Testbed	CPU	RAM	GPU	OS	Graphics Driver Version
Testbed-1	Intel Xeon	88 GB	NVidia Quadro Plex 7000	Windows Server 2008 R2 Workstation	301.32
Testbed-2	Intel Core i7-2640M @ 2.80 GHz	8 GB	NVidia Geforce GT 525M	Windows 7	301.27

Table 3: Specifications of the testbeds.

4.3 Stereo Scene Setup

All benchmarked ray tracers are fed a unified stereoscopic scene model with the following setup:

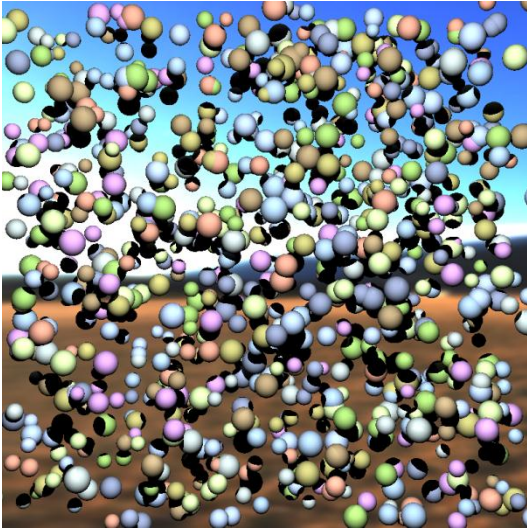
- Stereo cameras position: fixed in 3D space, with a variable interaxial distance e in pixels, specified by the benchmark.

- Image planes: fixed in 3D space, with variable dimensions w and h in pixels, specified by the benchmark.
- Light sources: benchmark-specific number of omni-directional light sources with fixed intensities.
- 3D geometric objects: we use 5 different setups for scene objects as illustrated in Table 4.
- Materials: diffuse only.
- A skymap.

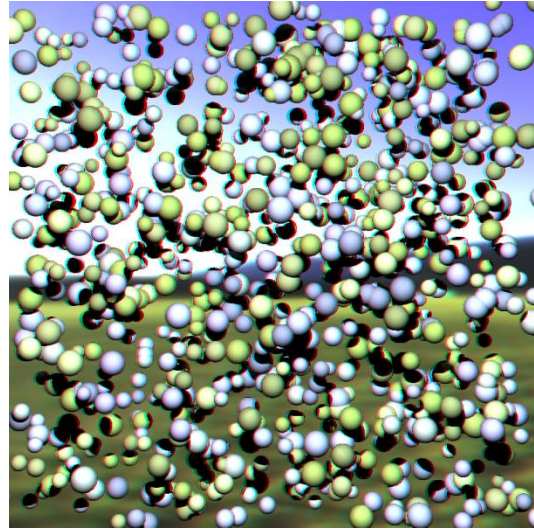
Figures 15-19 present the outputs of fully rendering the scene with different 3D objects.

Scene	3D Objects	Number of Polygons	Output
Fixed Spheres	1000 randomly distributed spheres in fixed positions	-	Figure 15
Animated Spheres	1000 randomly distributed spheres, rotating around the Y-axis	-	Figure 16
Sponza	Sponza 3D model [50]	279,163	Figure 17
Buddha	Happy Buddha 3D model [51]	1,087,716	Figure 18
Dragon	Stanford Dragon 3D model [51]	1,132,830	Figure 19

Table 4: 3D objects of the scenes.

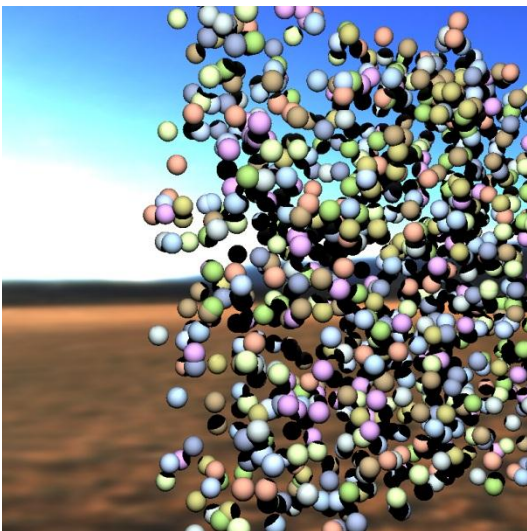


(a)

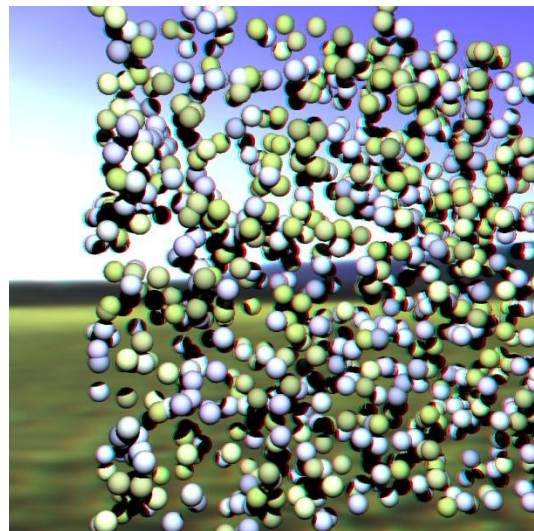


(b)

Figure 15: Fixed Spheres scene. (a) Mono output. (b) Stereo output.



(a)

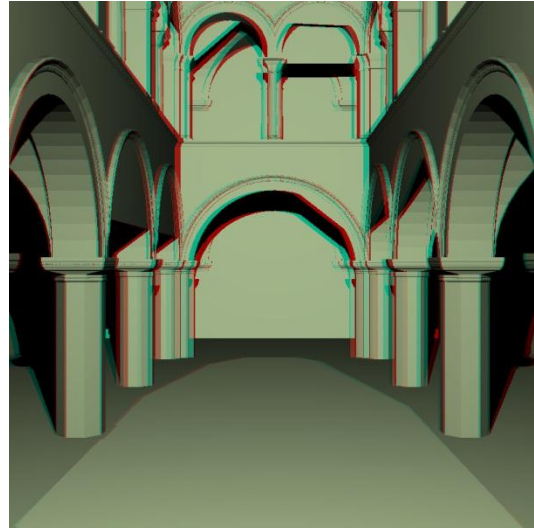


(b)

Figure 16: Animated Spheres scene. (a) Mono output. (b) Stereo output.



(a)



(b)

Figure 17: Sponza scene. (a) Mono output. (b) Stereo output.

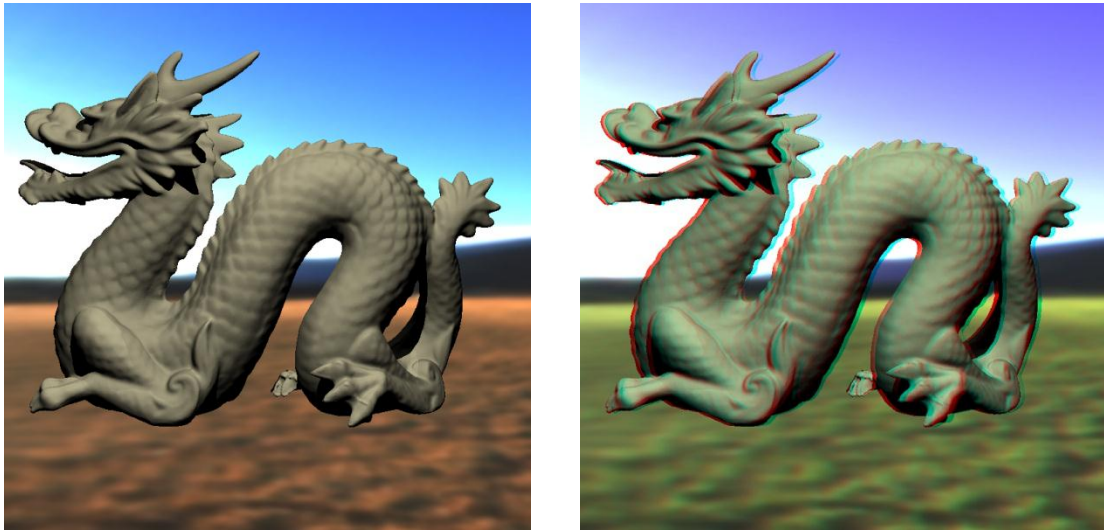


(a)



(b)

Figure 18: Buddha scene. (a) Mono output. (b) Stereo output.



(a)

(b)

Figure 19: Dragon scene (a) Mono output. (b) Stereo output.

4.4 Benchmarks

We run five different benchmarks over all the ray tracers. Three of these benchmarks are executed in both testbeds to test the performance of the ray tracers. These benchmarks alter the scene parameters, fixing two of the parameters and varying one. Each step in these benchmark is executed 5 times, each time runs for 25 seconds over each of the ray tracers; of which 5 seconds are for warming up the ray tracer, and the rest 20 seconds contribute to computing the average frames per seconds (fps) a ray tracer performs; which constitute the performance measure. To avoid unnecessary performance delays, the outputs of the ray tracers are not displayed on the monitor. We call these three benchmarks the *performance benchmarks*, and are summarized in Table 5.

The other couple of benchmarks, the *pixel error benchmarks*, test for the average pixel color error in the right image when tracing the Dragon scene with one of our methods against the naïve method. The details of these benchmarks are summarized in Table 6.

Performance Benchmark	Scene	Dimensions $w \times h$	Interaxial Distance e	Number of Lights
Varying dimensions	All five scenes	200 × 200 pixels up to 1500 × 1500 pixels, step size: 50 × 50 pixels	15 pixels	5
Varying interaxial distance	Dragon	600 × 600 pixels	10 → 100 pixels, step size: 10 pixels	
Varying number of lights			15 pixels	

Table 5: Performance benchmarks.

Pixel Error Benchmark	Images Generated	Dimensions $w \times h$	Interaxial Distance e	Number of Lights
Varying dimensions	Right images only, using the naïve ray tracer and the atomic-based ray tracer	200 × 200 pixels up to 1600 × 1600 pixels, step size: 200 × 200 pixels	15 pixels	5
Varying interaxial distance		600 × 600 pixels	10 → 100 pixels, step size: 10 pixels	

Table 6: Pixel-error benchmarks.

4.5 Results and Discussion

4.5.1 Performance Benchmarks Results

The performance in these benchmarks is measured by the average fps a ray tracer performs in 20 runs, each run execute for 25 seconds, of which 5 seconds are for warming up the ray tracer.

4.5.1.1 Results on Testbed-1

Figures 20-24 show the results of running the first performance benchmark, when varying image dimensions, spanning all ray tracers in all scenes. The average speedup per ray tracer in each scene is summarized in Figure 27, showing that the performance of the imaginary ideal ray tracer has an average speedup range of around 29% to 65% over the baseline. This is due to the penalty associated with copying the left image to the right image as implemented in this ray tracer. Moreover, our ray tracers are performing at speedup ranges of 14% to 47% for the atomic-based ray tracer, and 6% to 40% for the buffer-based ray tracer, relative to the baseline. As expected, the atomic-based ray tracer outperforms the buffer-based one. Surprisingly, Es-Isler ray tracer exhibited poor performance relative to the baseline. This is maybe due to the fact that their technique was not optimized for massively-parallel processors. The plotted performance trend-lines in this testbed show some fluctuations which we could not explain².

² We are in contact with NVidia team in this regard.

Figure 25 plots the performance when applying the second benchmark to the ray tracers; increasing the interaxial distance e in the Dragon scene. Increasing e , as the figure shows and as expected, does not affect the performance of both the imaginary ideal and the naïve ray tracers. Relevant to the complexity analysis in the previous chapter, incrementing e has a slight impact on the performance of the atomic-based ray tracer, due to the fact that increasing e will increase the number of missed pixels in the right image, alongside expanding the linear overhead incurred by using the atomic operations. Similar performance drop can be observed in the reprojection-based Es-Isler ray tracer. Furthermore, the performance of the buffer-based ray tracer highly declines once e is increased. This is because increasing e reflects on the depth of the 3D buffer used in the ray tracer, rendering a slower traversal towards the depth (Section 3.1.2.1). It is worth mentioning that the suggested 3D buffer size, $2e$, serves as a maximum size to handle extreme cases as presented in Figure 12, which rarely happen in a scene. It is therefore possible to set the buffer size in the buffer-based ray tracer to a small fixed value – independent of e – while getting correct outputs. This way, the only impact of increasing e in this modified ray tracer will be caused by the congruently increased number of missed pixels.

Lastly, Figure 26 illustrates the performance of the ray tracers when increasing the number of lights in the scene as per the third benchmark. Generally, increasing scene lights reflects an exponential drop in any ray tracer's performance, due to the need of tracing a shadow rays per each light source in the scene. However, this drop in performance is slower on our ray tracers as opposed to the baseline, because only the

cost of computing pixels in error in the right image is expanded by increasing the number of lights.

4.5.1.2 Results on Testbed-2

Figures 28-34 show the performance of the corresponding benchmarks when applied to Testbed-2. We still get the same trendlines in each graph as of the previous testbed results, but with lower performance in general, and smoother trendlines when varying image dimensions. Figure 35 shows that the average speedup ratios are better in this testbed, with value ranges of 16%-58%, 32%-67%, and 71%-92% for the buffer-based, the atomic-based and the imaginary ideal ray tracers, respectively. It also shows that the Es-Isler ray tracer is again underperforming the baseline.

4.5.2 Pixel Error Benchmarks Results

Reprojection causes no visible structural differences in the produced right image when compared to a fully ray traced image (Figure 39(a) vs. Figure 40(a)). Therefore, it is sufficient to assess the quality of reprojected images using error sensitivity based techniques [52].

Let R denote the right image produced by one of our techniques, and \bar{R} denote a right image that is fully ray traced. To quantize the pixel error value, each pixel in both right images, R and \bar{R} , outputted by these benchmarks is represented as a vector $c_i = (r, g, b)$ in RGB space ($256 \times 256 \times 256$), and the pixel error is computed as the Mean Squared Error (MSE):

$$\text{MSE}(R, \bar{R}) = \frac{\sum_{i=1}^{wxh} \|(c_i - \bar{c}_i)\|^2}{3 \times w \times h}$$

Increasing the image dimensions while fixing the interaxial distance e , as Figure 36 shows, reflects a better quality in the reprojected right image R . However, increasing e seems to increasingly affect the quality of R in comparison to the fully ray traced image \bar{R} , as shown in Figure 37.

4.5.3 Time Views

Time views are grayscale images utilized to illustrate the amount of time each thread in a ray tracer spends on generating one pixel where, relative to other pixels, lighter pixels indicate high ray tracing time, and vice-versa.

Figure 38 presents two time views for two kernels generating the right image, one using the naïve ray tracer and the other using the atomic-based ray tracer. It is evident that the threads of the naïve ray tracer spend much time ray tracing the geometry, while the time-consuming threads of the atomic-based ray tracer are only distributed around the edges of the geometry, where most of bad and missed pixels occur.

4.5.4 Outputs

Figures 39-41 show the outputs of the ray tracers. As established earlier, no visible differences can be spotted between the outputs.

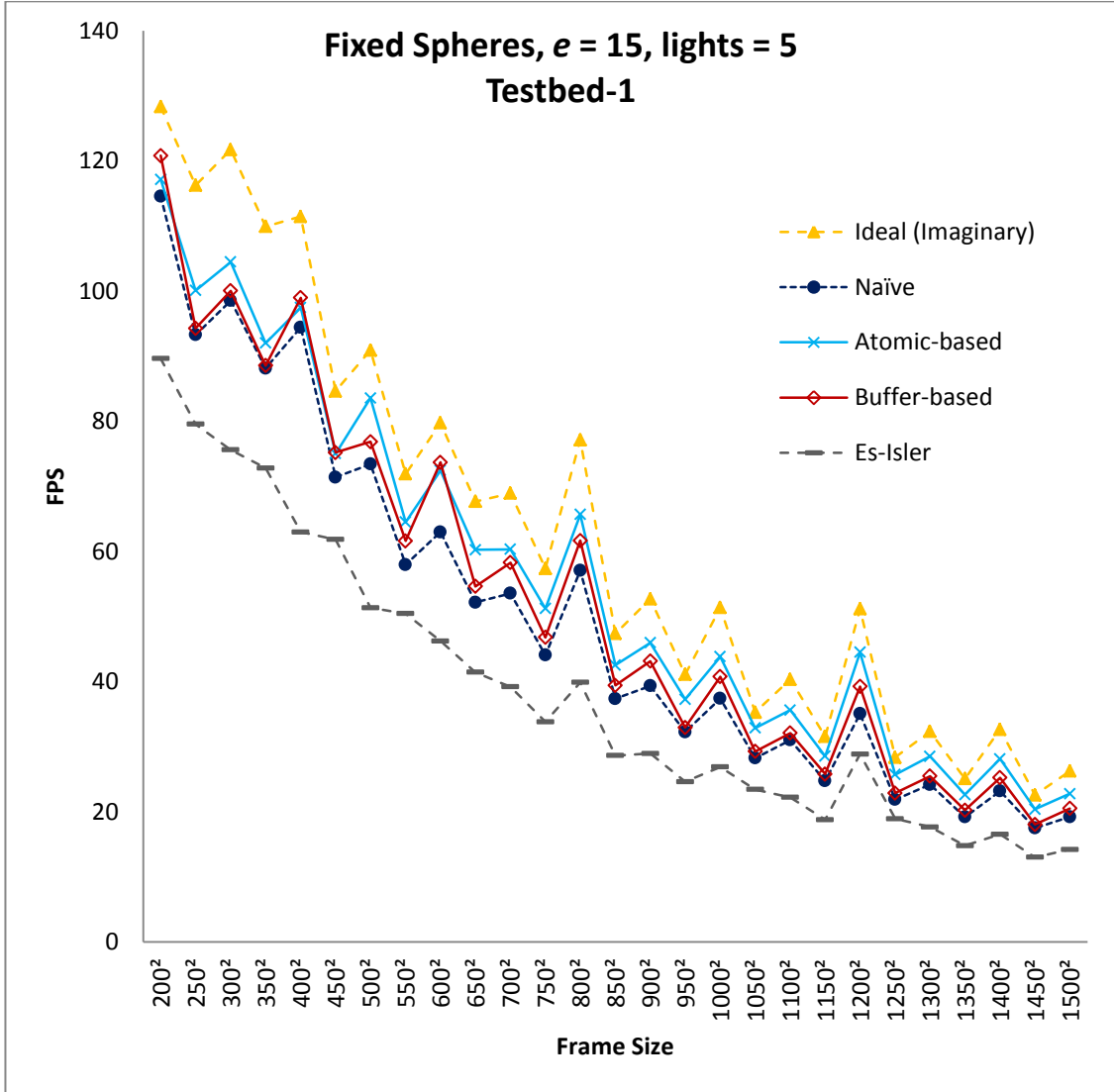


Figure 20: Performance of ray tracing the Fixed Spheres scene when increasing image dimensions in Testbed-1.

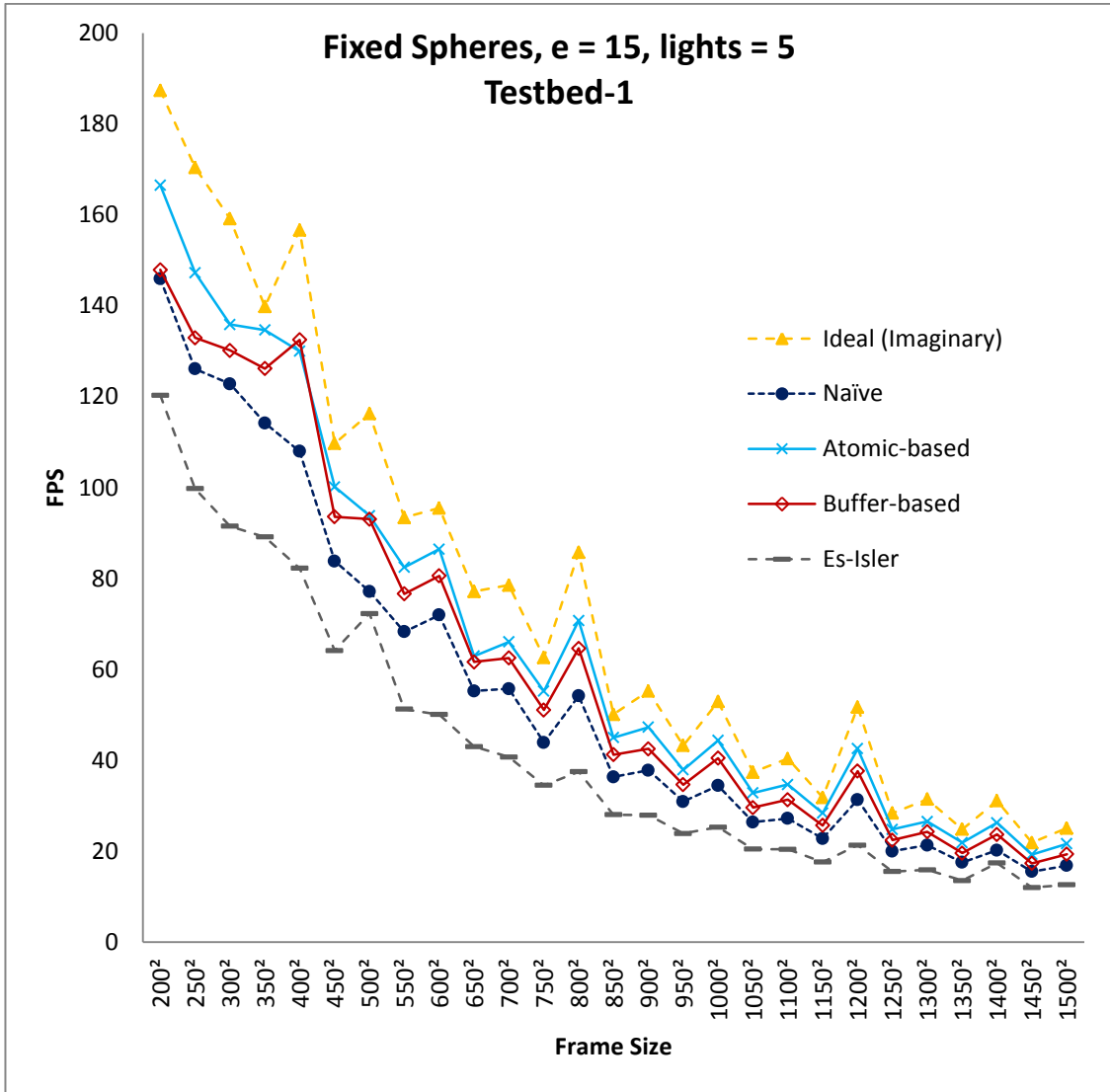


Figure 21: Performance of ray tracing the Animated Spheres scene when increasing image dimensions in Testbed-1.

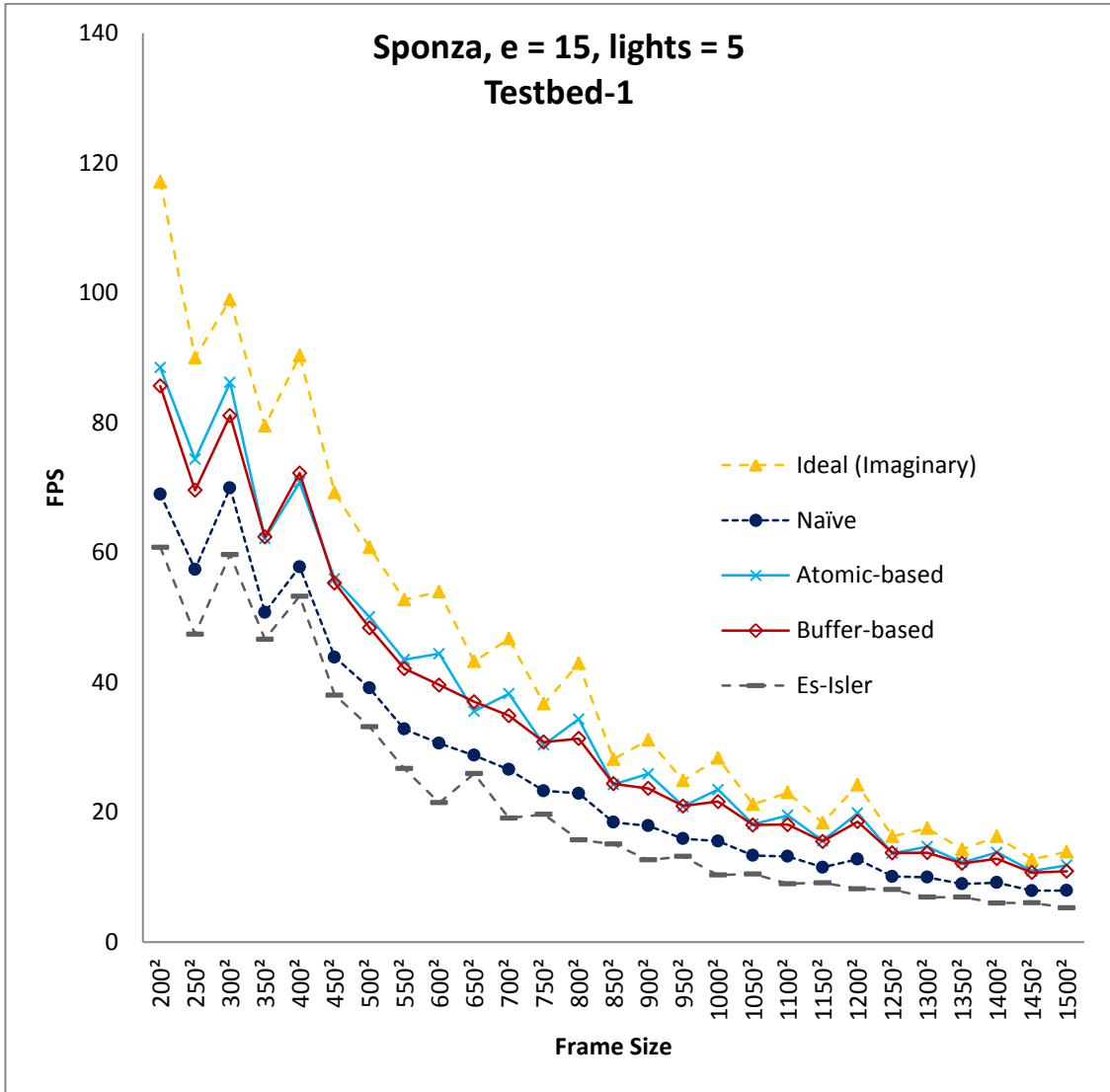


Figure 22: Performance of ray tracing the Sponza scene when increasing image dimensions in Testbed-1.

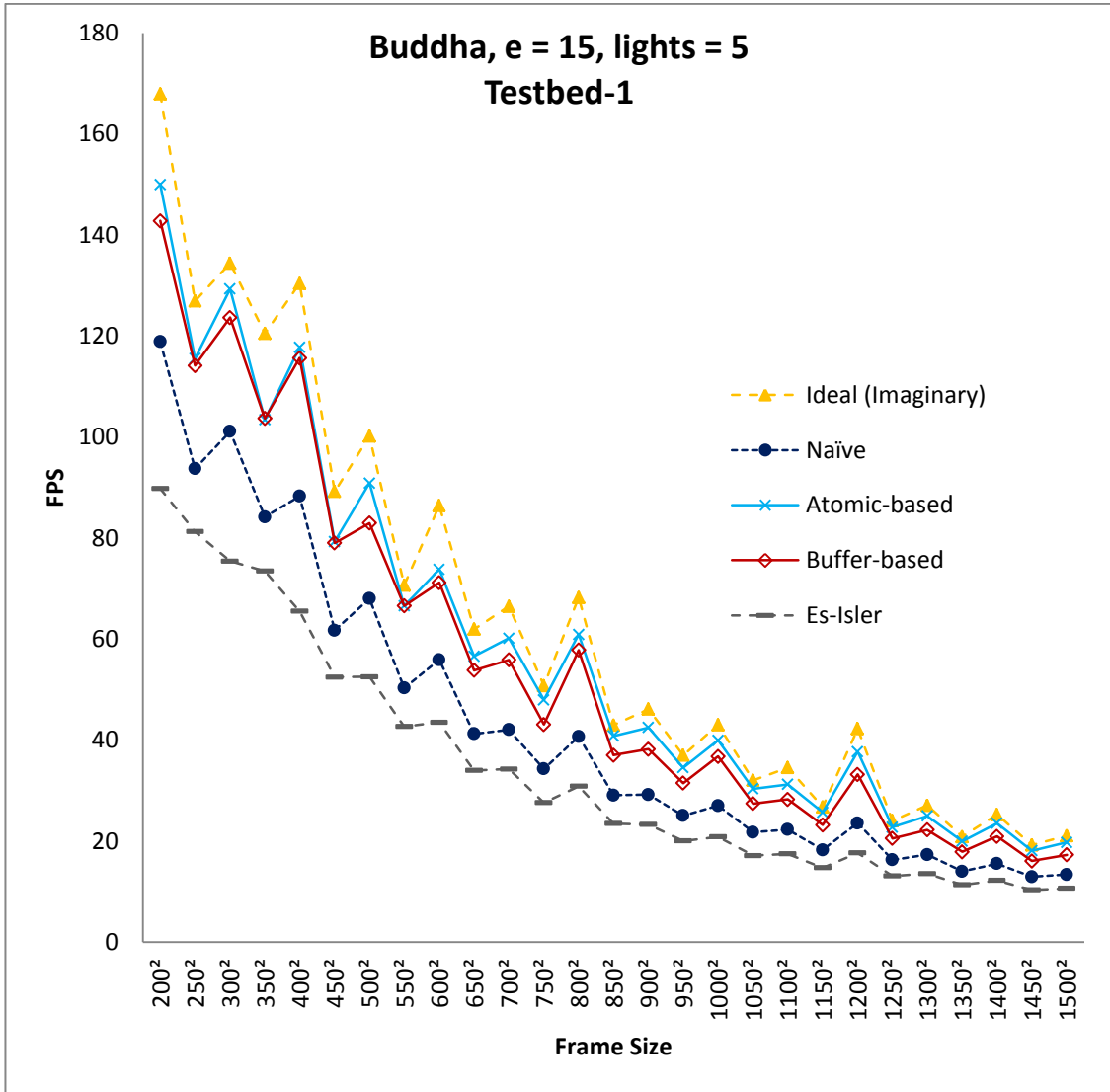


Figure 23: Performance of ray tracing the Buddha scene when increasing image dimensions in Testbed-1.

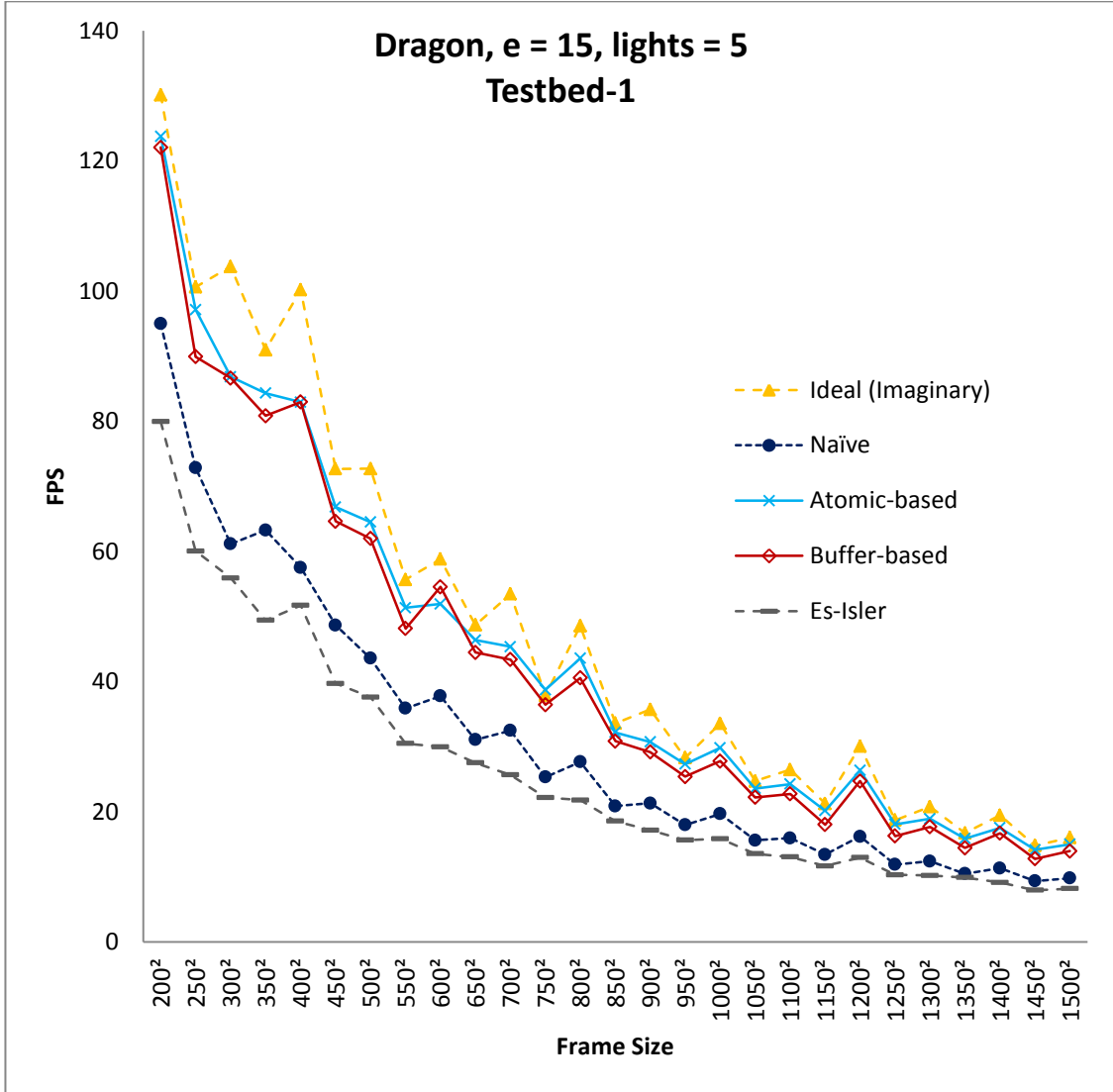


Figure 24: Performance of ray tracing the Dragon scene when increasing image dimensions in Testbed-1.

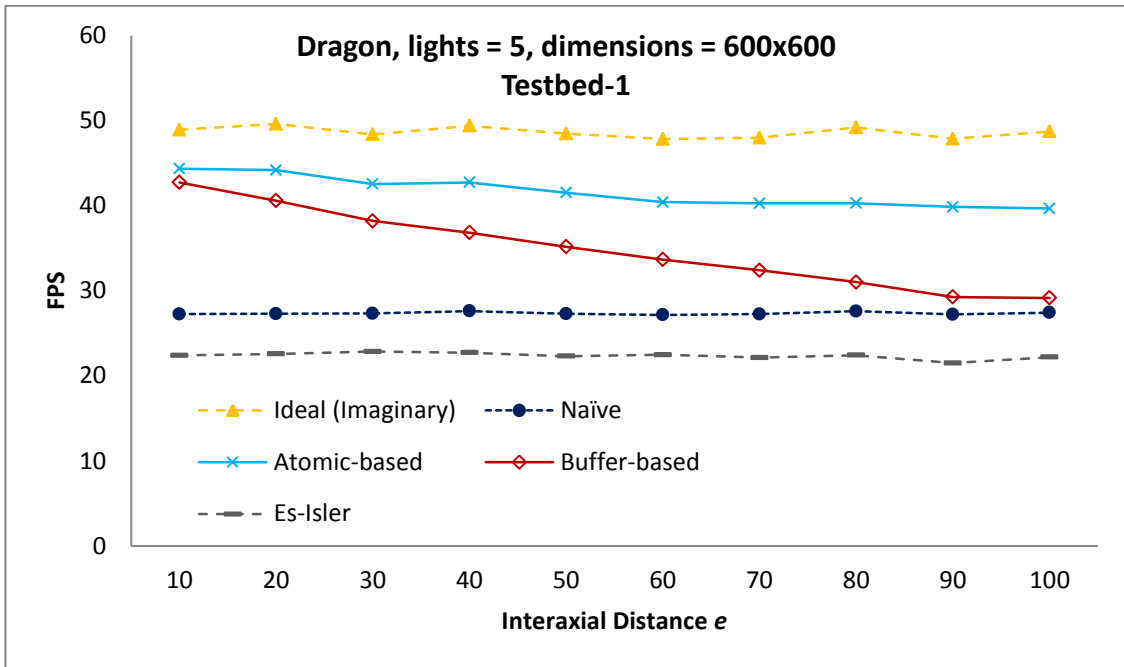


Figure 25: Performance of ray tracing the Dragon scene when increasing the interaxial distance e in Testbed-1.

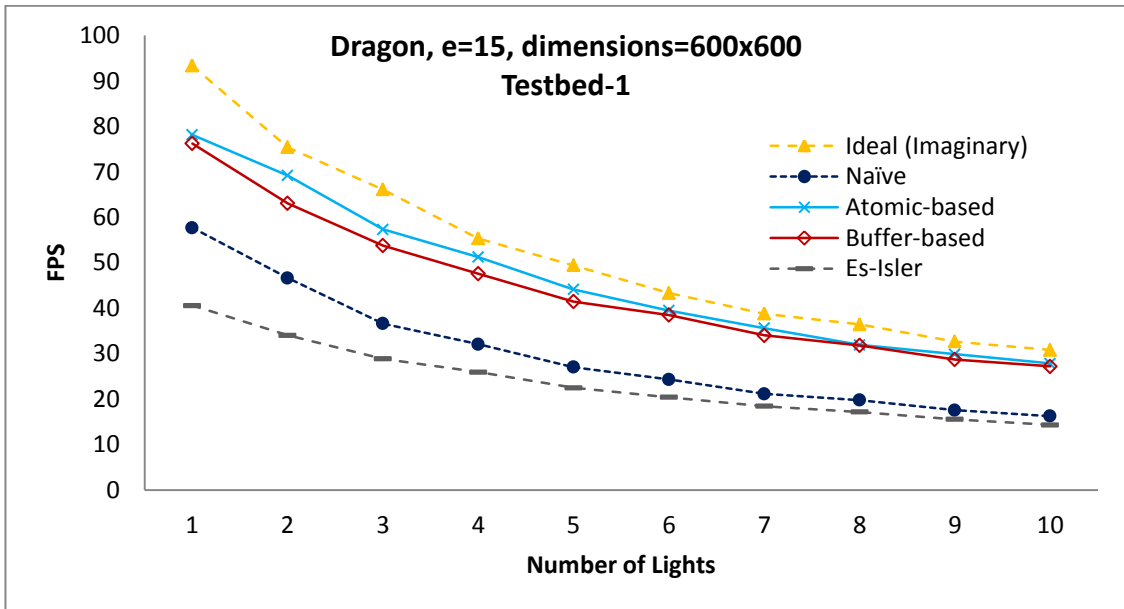


Figure 26: Performance of ray tracing the Dragon scene when increasing number of lights in Testbed-1.

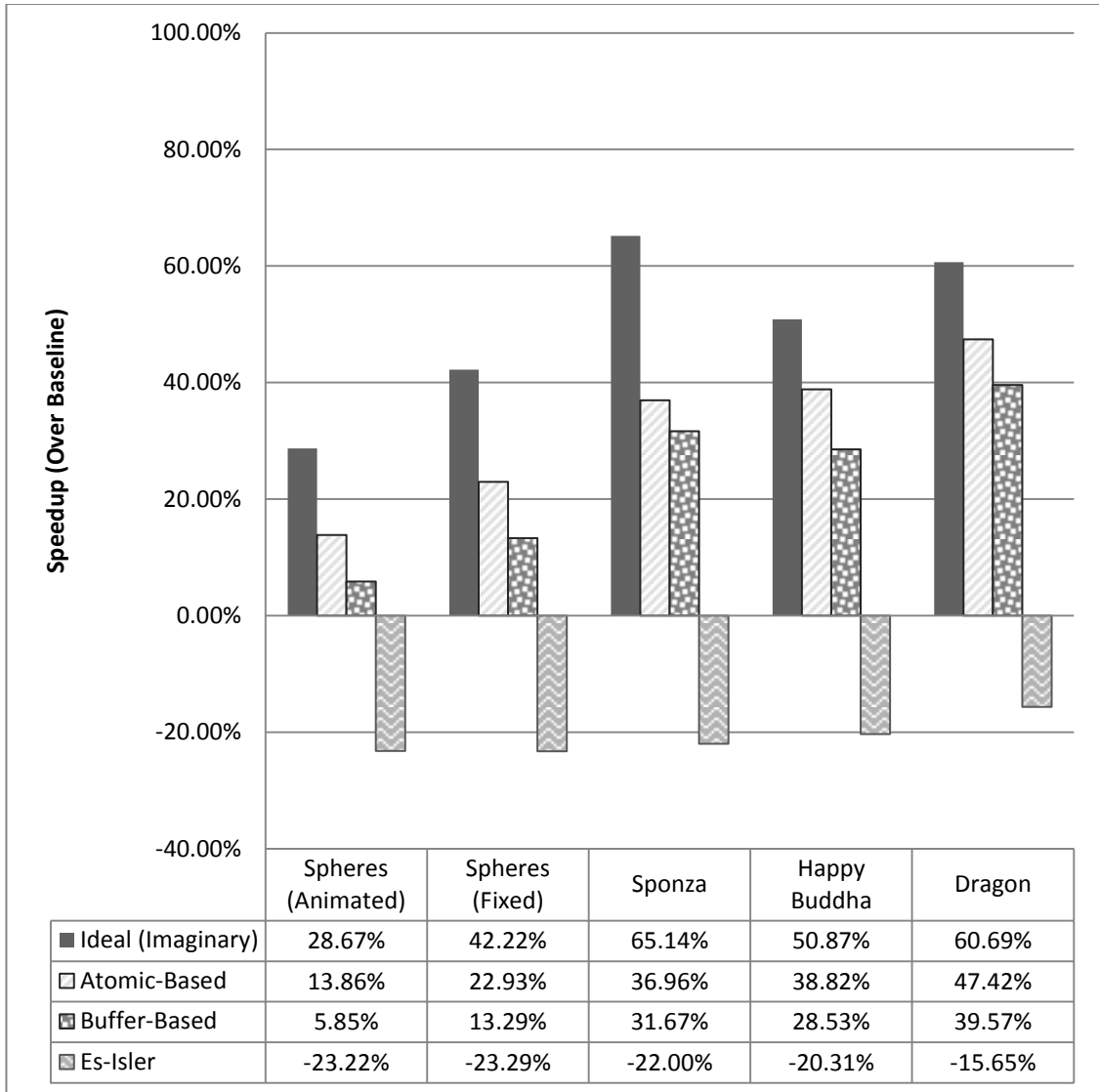


Figure 27: Speedup summary relative to the naïve ray tracer, Testbed-1.

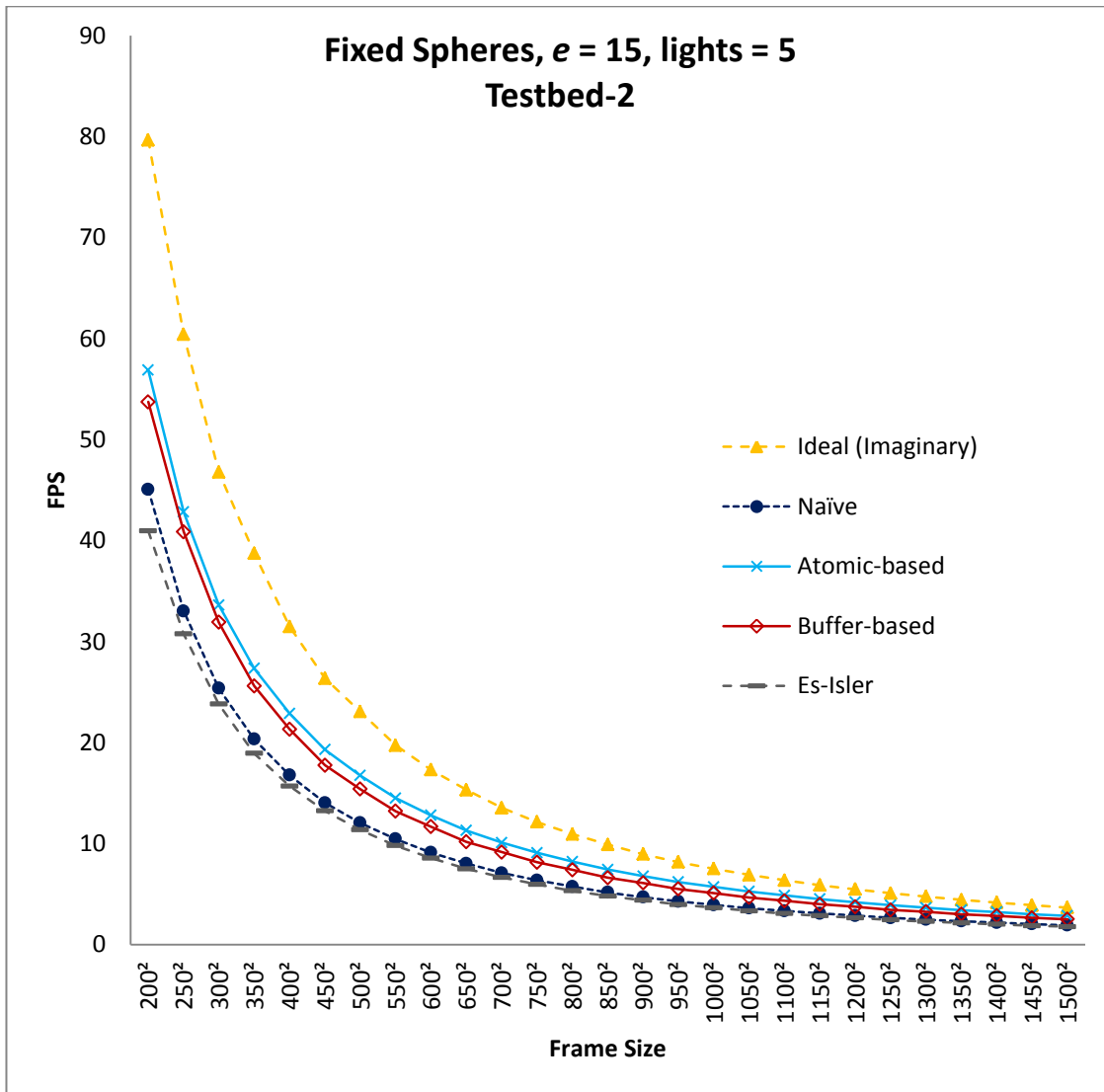


Figure 28: Performance of ray tracing the Fixed Spheres scene when increasing image dimensions in Testbed-2.

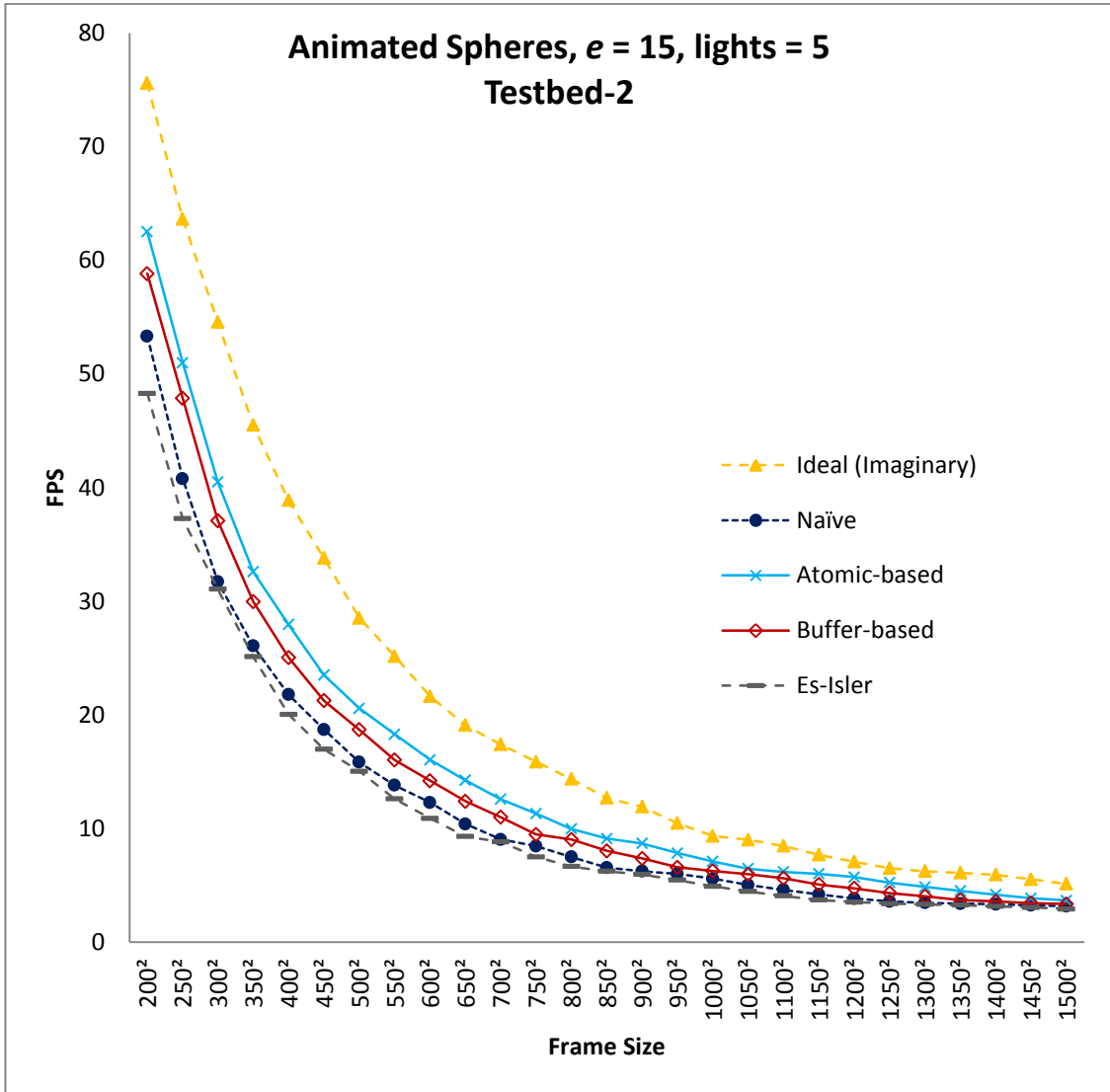


Figure 29: Performance of ray tracing the Animated Spheres scene when increasing image dimensions in Testbed-2.

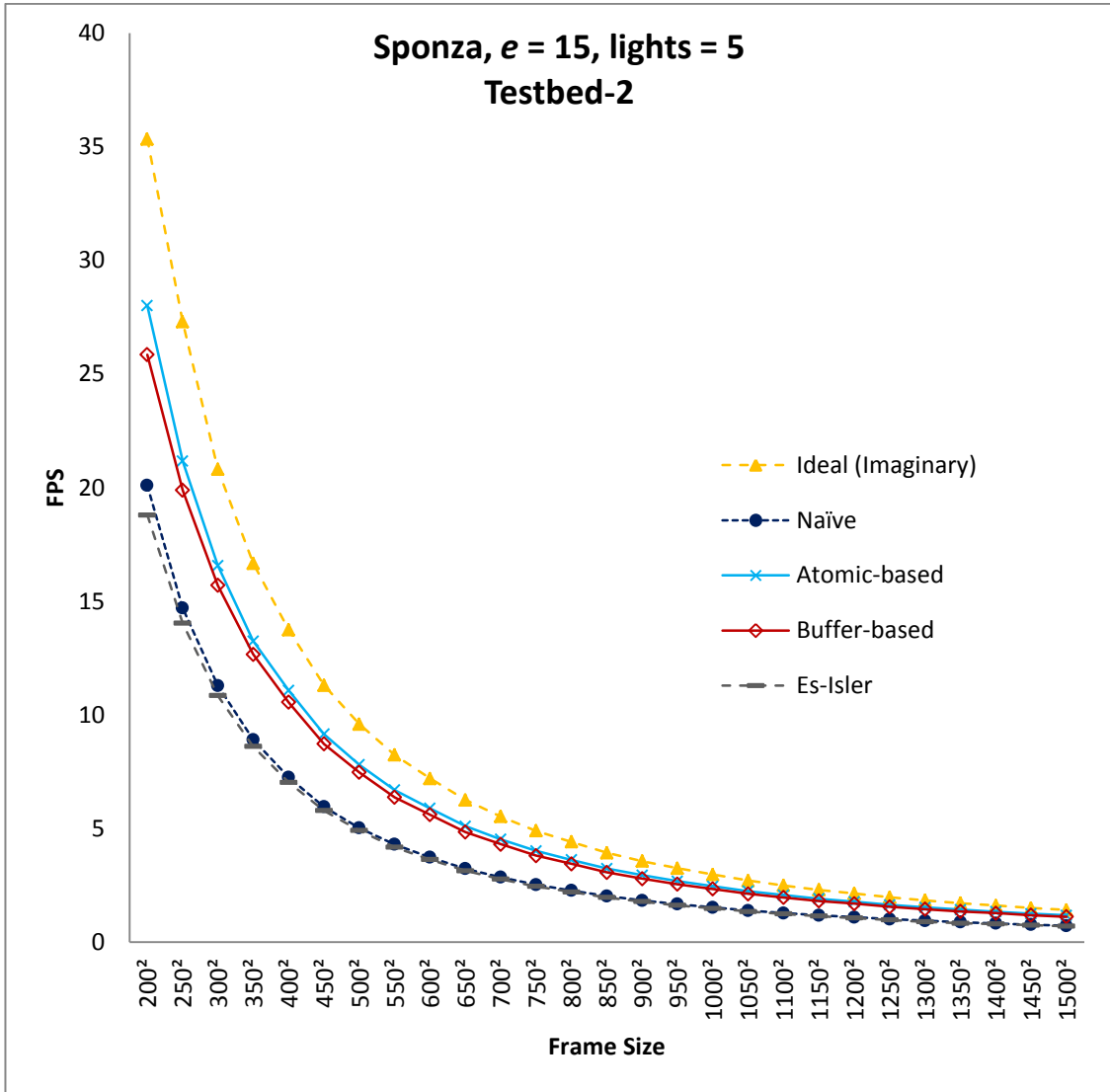


Figure 30: Performance of ray tracing the Sponza scene when increasing image dimensions in Testbed-2.

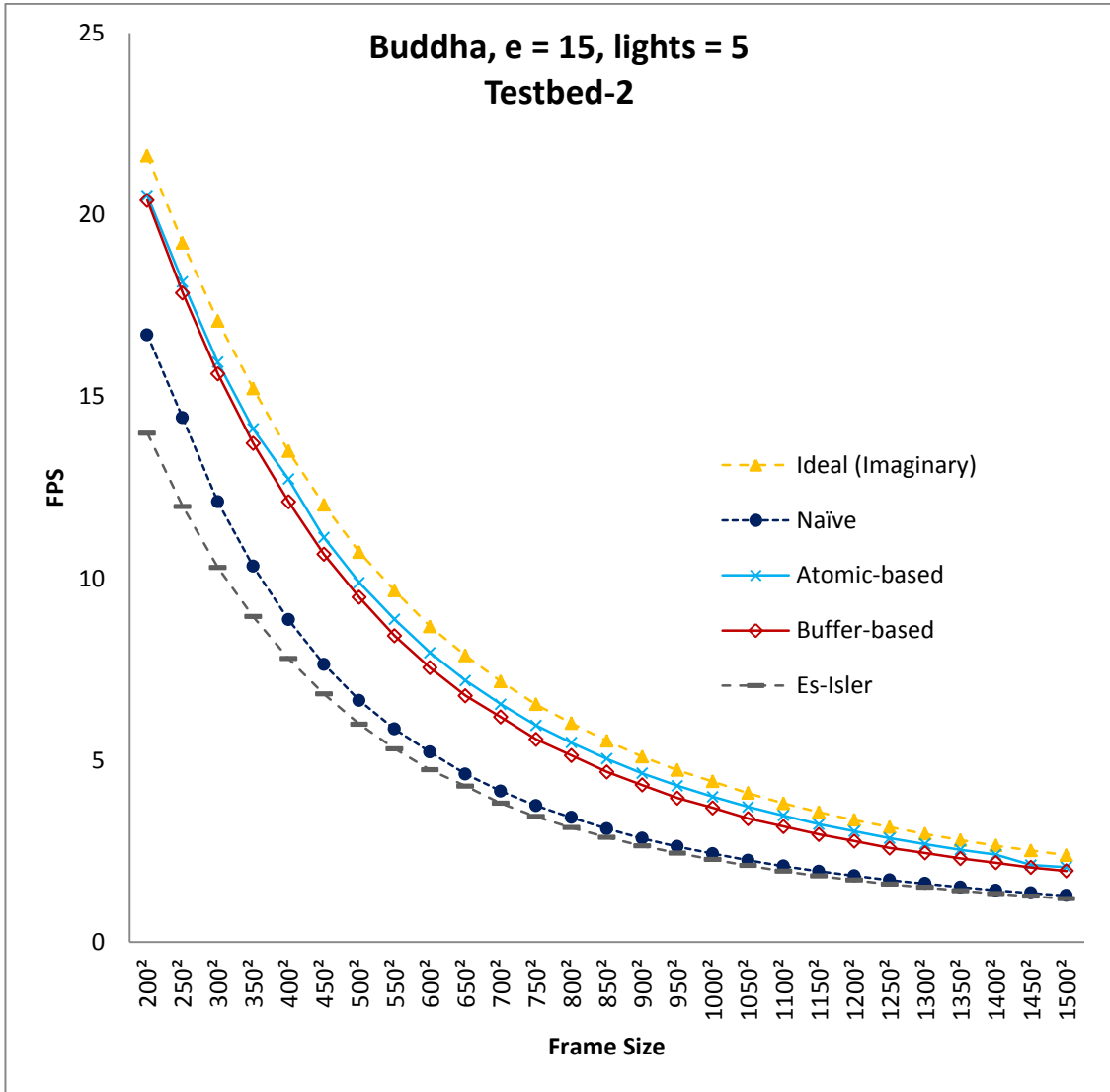


Figure 31: Performance of ray tracing the Buddha scene when increasing image dimensions in Testbed-2.

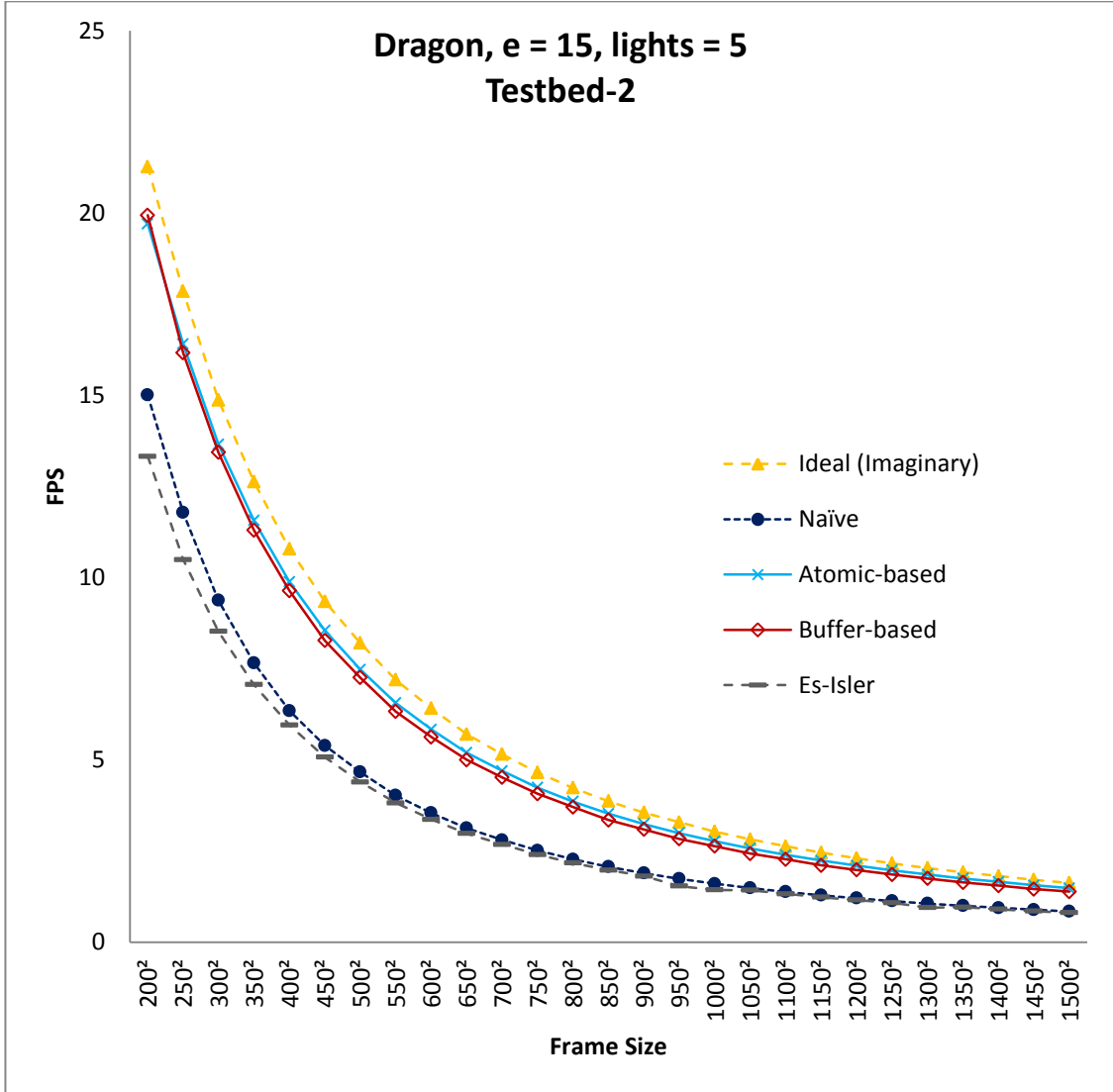


Figure 32: Performance of ray tracing the Dragon scene when increasing image dimensions in Testbed-2.

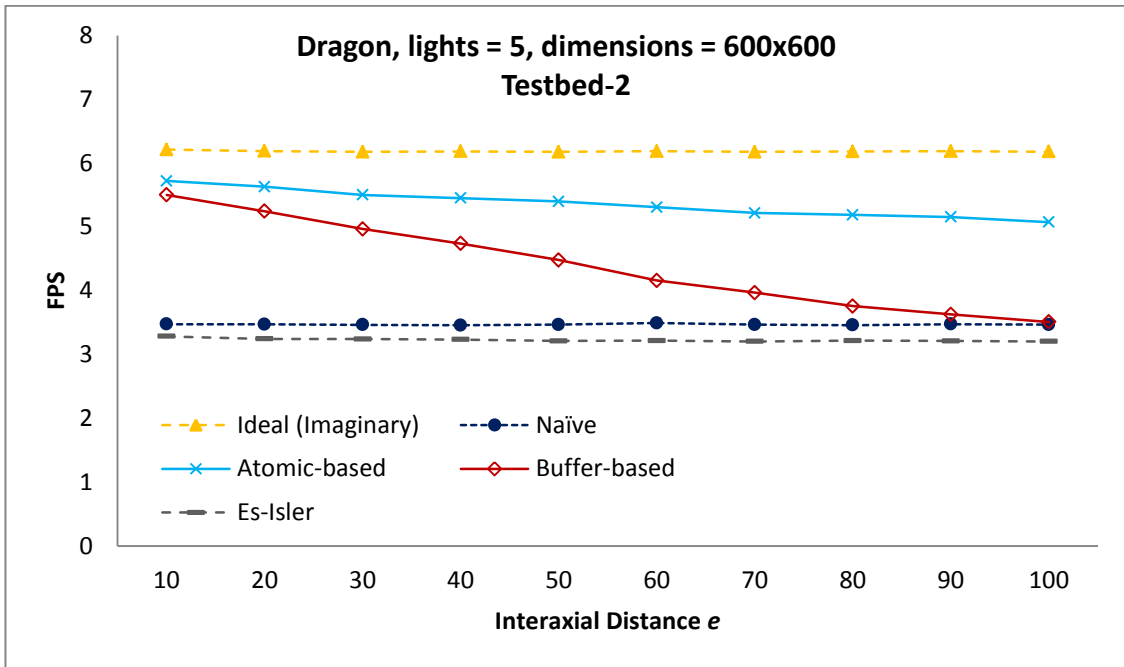


Figure 33: Performance of ray tracing the Dragon scene when increasing the interaxial distance e in Testbed-2.

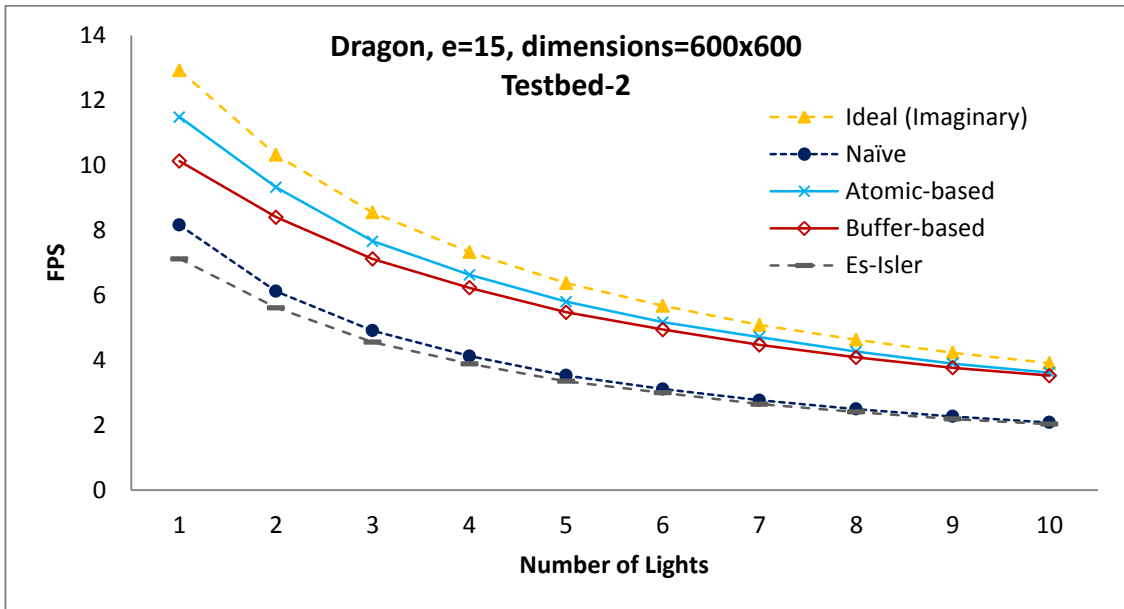


Figure 34: Performance of ray tracing the Dragon scene when increasing number of lights in Testbed-2.

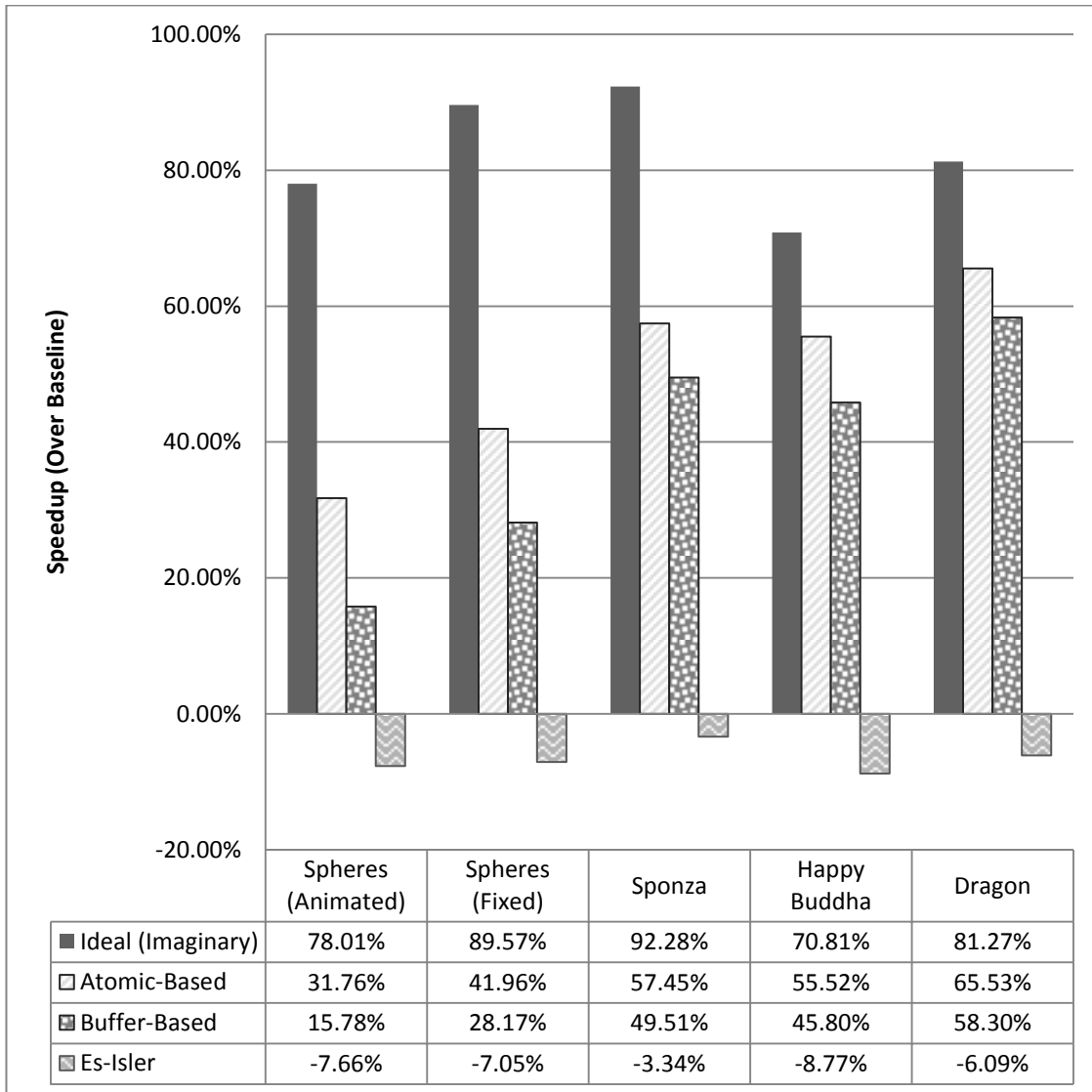


Figure 35: Speedup summary relative to the naïve ray tracer, Testbed-2.

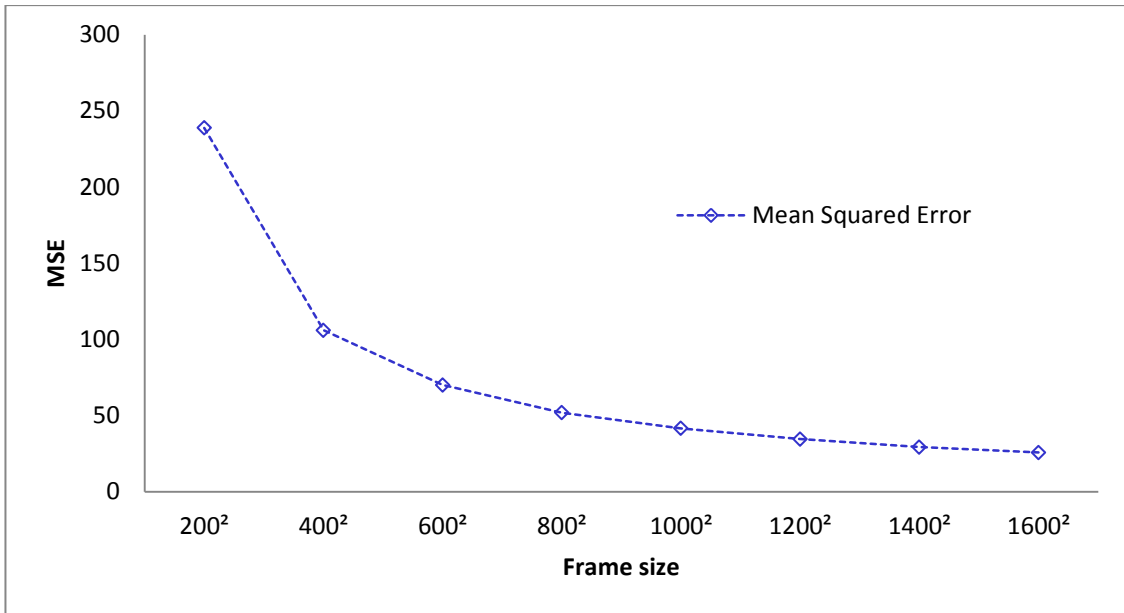


Figure 36: MSE when increasing image dimensions.

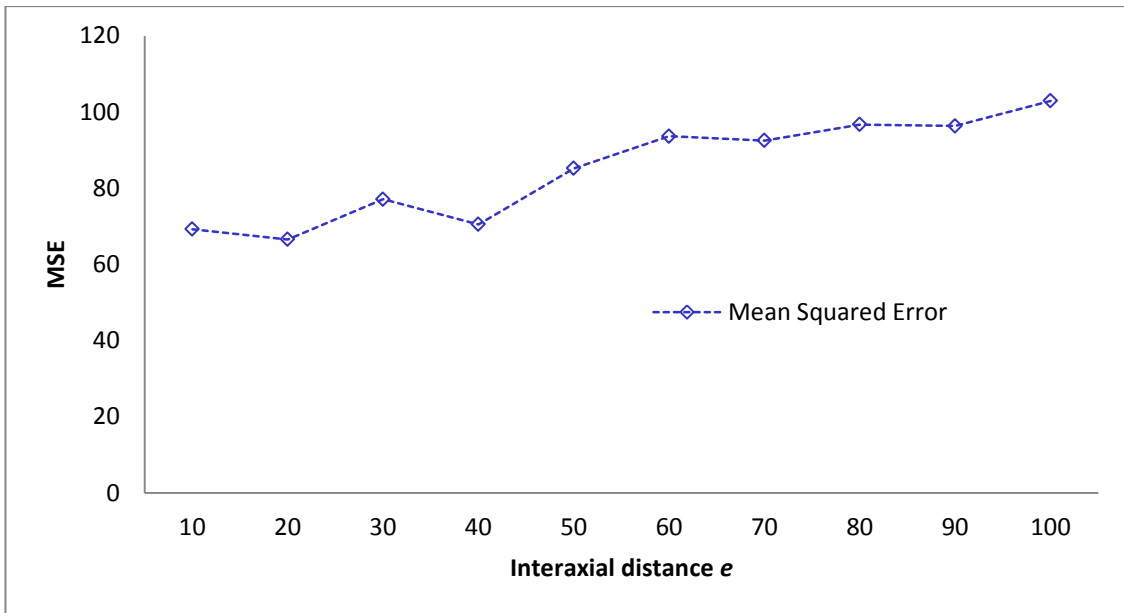
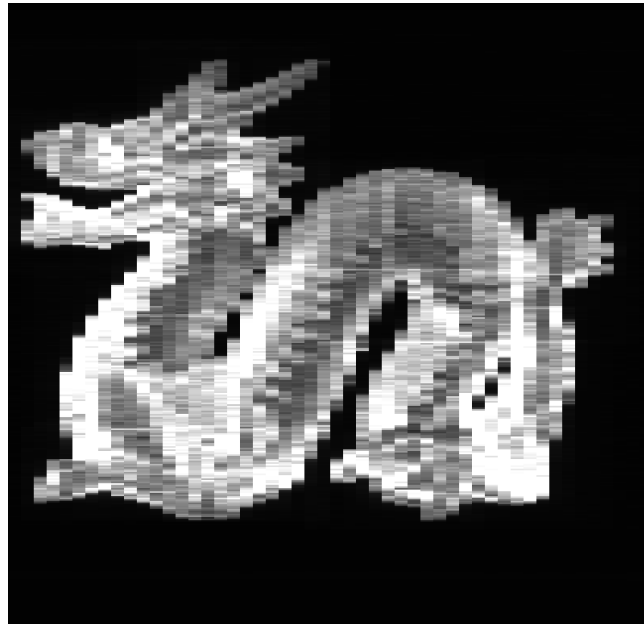


Figure 37: MSE when increasing the interaxial distance e .

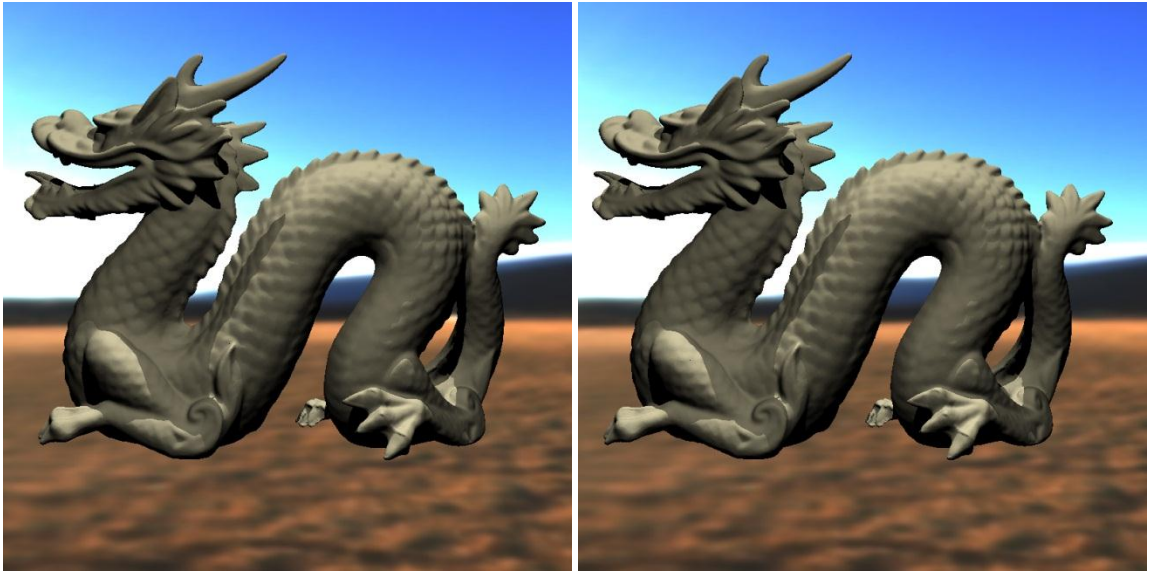


(a)



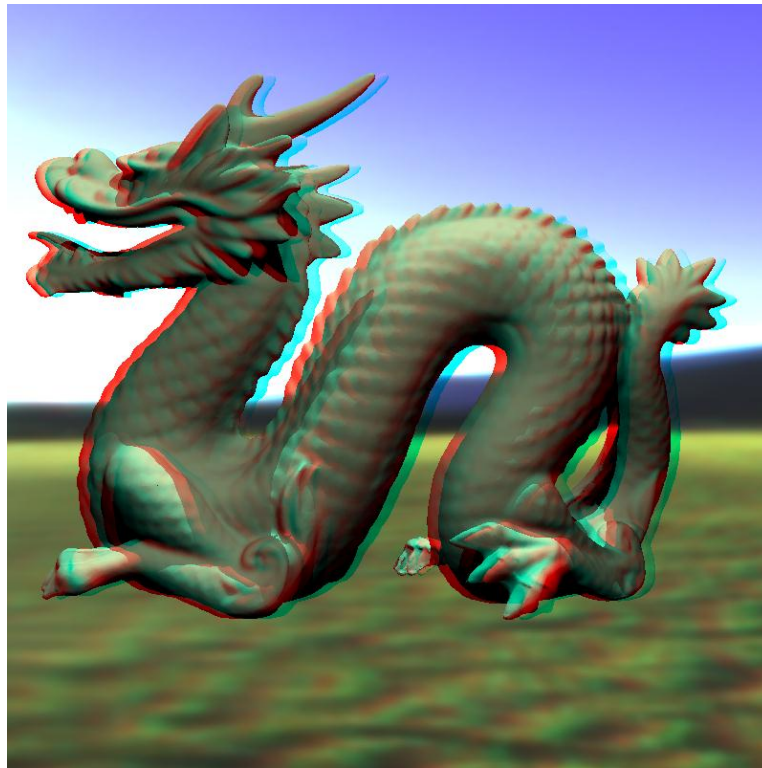
(b)

Figure 38: Time views of the threads generating the right image using: (a) Naïve ray tracer. (b) Atomic-based ray tracer.



(a)

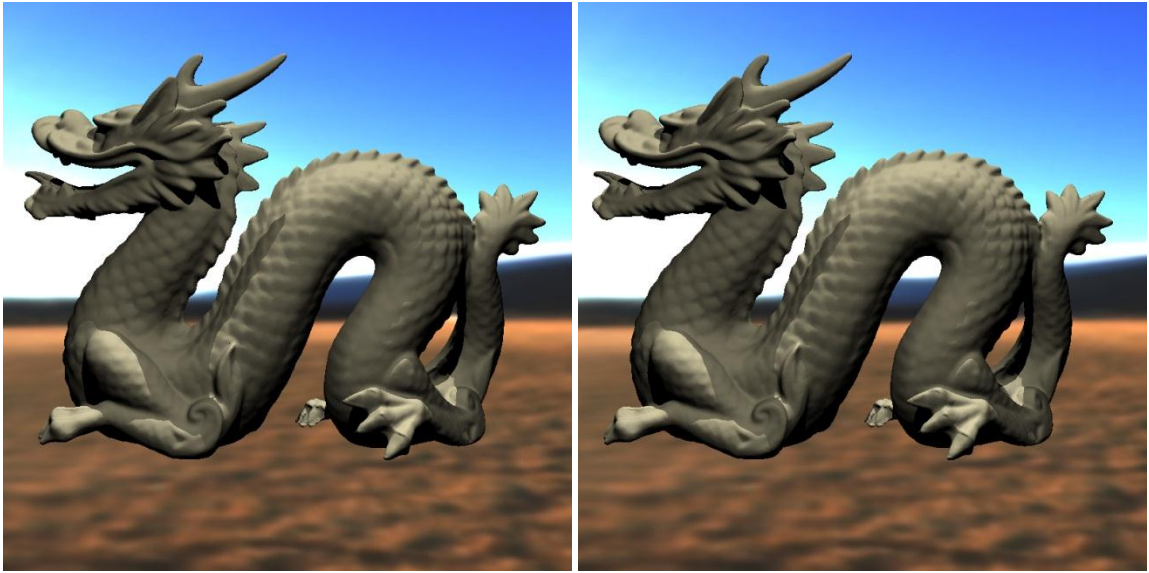
(b)



(c)

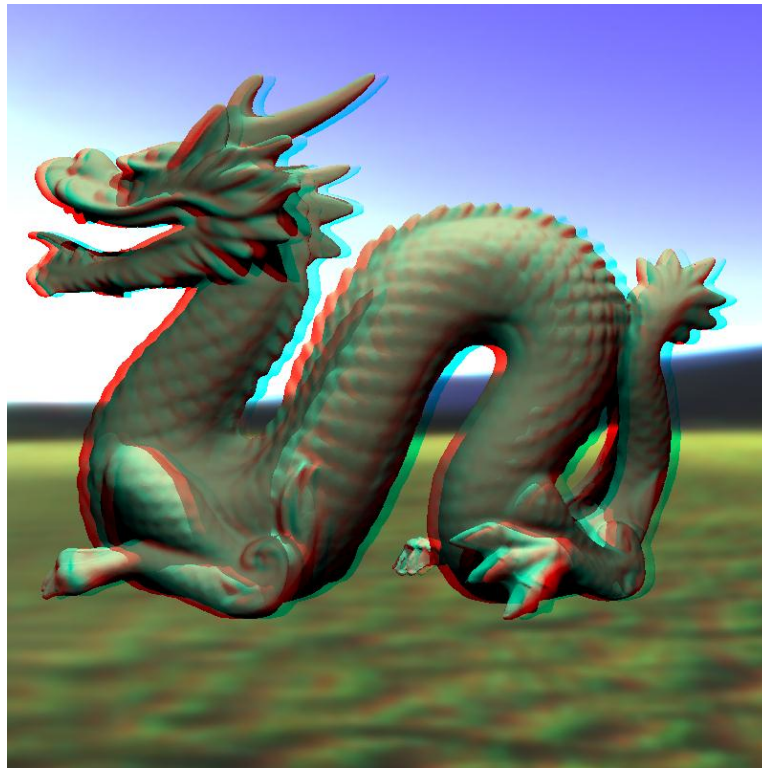
Figure 39: Output of naïve ray tracer. (a) Left image. (b) Right image.

(c) Anaglyph stereo image.



(a)

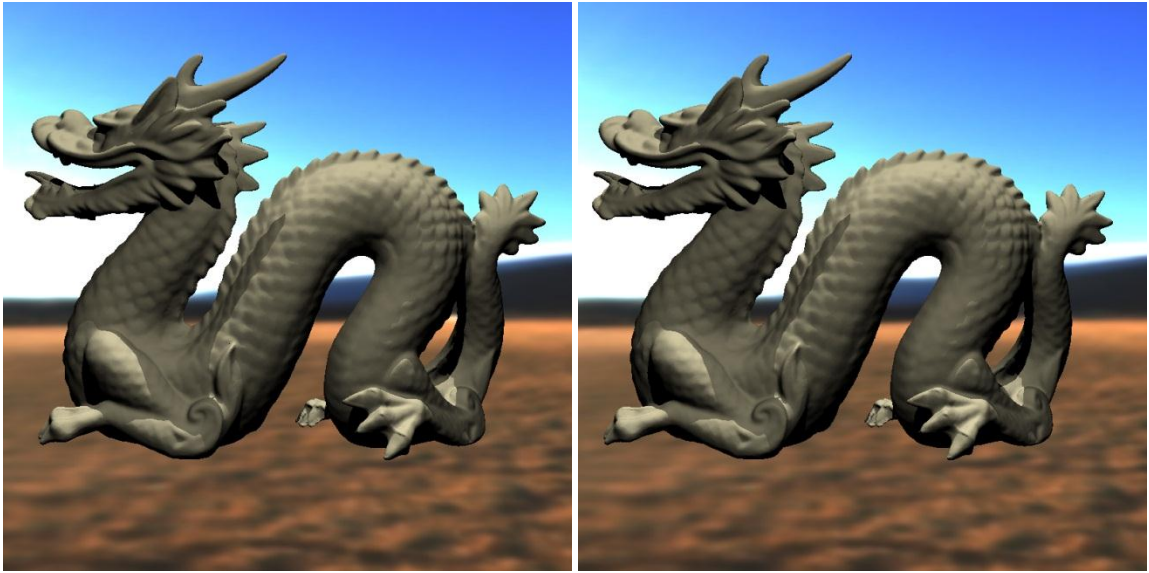
(b)



(c)

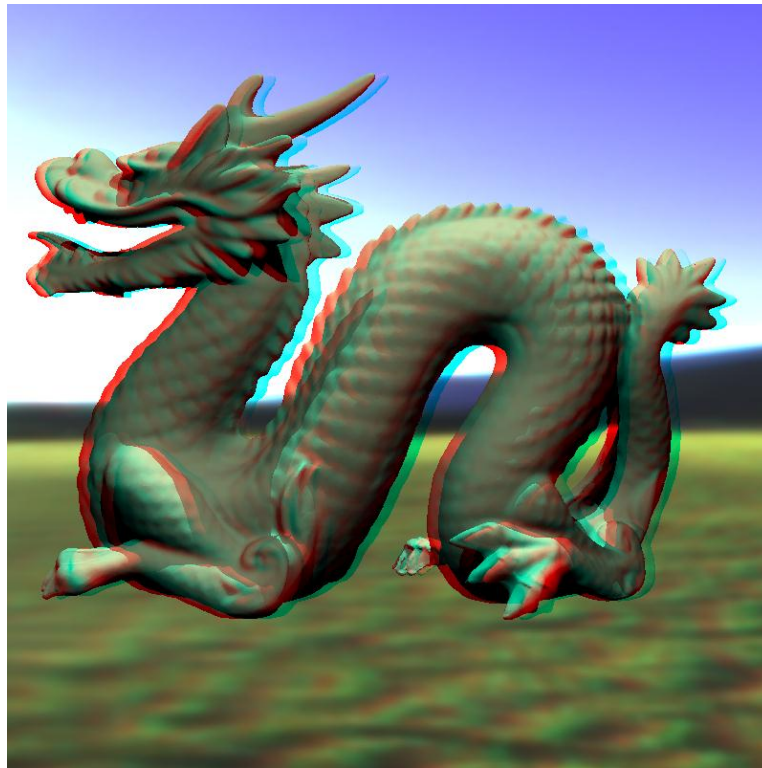
Figure 40: Output of buffer-based ray tracer. (a) Left image. (b) Right image.

(c) Anaglyph stereo image.



(a)

(b)



(c)

Figure 41: Output of atomic-based ray tracer. (a) Left image. (b) Right image.

(c) Anaglyph stereo image.

CHAPTER FIVE

CONCLUSION AND FUTURE WORK

5.1 Summary

This work focuses on developing efficient stereoscopic ray tracing on the GPU, by utilizing image-space temporal coherence between the stereo pair.

The recent explosion of GPU performance naturally grabbed the attention of researchers to develop existing algorithms on GPUs to harness their full potential; especially that an added layer of complexity is associated with GPU development due to its architecture.

One of the most effective techniques for utilizing temporal coherence between a ray traced stereo pair is the reprojection algorithm, introduced by Badt and later developed by Adelson and Hodges. This technique produces high quality results when transferring pixels from the left image to the right image. However, the technique, targeting CPUs, is sequential in nature, and the existing attempts to make it run on parallel are not optimized for massively parallel processors.

Novel resolutions to reprojection problems have been developed and presented throughout this work. These resolutions allowed the originally sequential reprojection to be implemented on massively parallel processors, such as GPUs.

The results show that our developed techniques outperform the naïve technique of fully ray tracing through both images of a stereo pair, and approach the performance of an imaginary ideal implementation.

5.2 Contribution to Knowledge

This work has achieved the following contributions that were never addressed in the literature before:

- Re-invented the way reprojection errors are handled so that reprojection can work on massively parallel processors.
- Lemma 1 set an upper bound to the maximum number of writes to one pixel position in the right image when using reprojection.

5.3 Limitations

Despite the good performance of the developed techniques in this work, there are some shortcomings:

- Reprojection, and therefore our techniques, produces correct outputs only for surfaces of diffuse material, and can handle a narrow subset of camera-dependent materials such as Phong [53]. Pixels produced from surfaces of other camera-dependent materials such as reflective and refractive materials do not reproject correctly. To mitigate this problem, Adelson and Hodges suggested to fully ray trace these pixels in the right view, downgrading the performance.

- The performance of our techniques can be shown to be independent of most 3D stereo scene attributes, except for the interaxial distance e . Large values of e introduce more pixel problems and, therefore, will render our algorithms underperforming the naïve method.
- For large values of r , i.e. if the nearest surface in the stereo scene is positioned at relatively small distance from cameras, the buffer-based approach will be rendered inefficient, because the 3D buffer will grow in depth and this will reflect on slowing the performance of the algorithm as showed earlier.

5.4 Future Work

Our techniques serve as a possible core for utilizing image-space temporal coherence in stereoscopic ray tracing implemented on massively parallel processors. Still, there is plenty of room to further optimize and enhance them. Following is a list of possible enhancements that are worth investigating in the future:

- Implementing our techniques on a distribution ray tracer.
- Use reprojections from previously rendered animation frames to reduce pixel errors, as suggested by Adelson and Hodges in another work [16].
- Allowing the 3D buffer depth in the buffer-based ray tracer to be set adaptively, relative to the scene being rendered.
- Bad pixels heavily occur around the edges of the rendered geometry. This heuristic can be used to directly ray trace through edge-surrounding pixels without having to check if they constitute bad pixels.

- Nehab et al. [54] developed another technique for image-space temporal coherence using reverse reprojections alongside a caching technique. Applied to stereo rendering, this means fully ray tracing the left view, and ‘reversely’ reprojecting pixel positions from the right view to the left view so as to find their corresponding colors. Their technique avoids reprojection errors, but comes at the cost of computing the depth of the pixels in the right image. Moreover, their technique is optimized for use in rasterization-based renderers. It would be interesting to investigate the possibility to adopt their technique with our techniques to achieve yet further optimizations.
- Investigating other thread scheduling mechanisms as to assure load-balancing on the GPU cores, and possibly increasing the performance.

REFERENCES

- [1] G. Tran. (2012, 25/5/2012). Available:
<http://www.oyonale.com/modeles.php?lang=en&page=40>
- [2] T. Whitted, "An improved illumination model for shaded display," *Commun. ACM*, vol. 23, pp. 343-349, 1980.
- [3] R. L. Cook, T. Porter, and L. Carpenter, "Distributed ray tracing," *SIGGRAPH Comput. Graph.*, vol. 18, pp. 137-145, 1984.
- [4] C. M. Goral, K. E. Torrance, D. P. Greenberg, and B. Battaile, "Modeling the interaction of light between diffuse surfaces," *SIGGRAPH Comput. Graph.*, vol. 18, pp. 213-222, 1984.
- [5] J. D. Foley, A. v. Dam, S. K. Feiner, and J. F. Hughes, *Computer graphics: principles and practice (2nd ed.)*: Addison-Wesley Longman Publishing Co., Inc. , 1990.
- [6] G. Bishop, H. Fuchs, L. McMillan, and E. J. S. Zagier, "Frameless rendering: double buffering considered harmful," presented at the Proceedings of the 21st annual conference on Computer graphics and interactive techniques, 1994.
- [7] I. Wald, W. R. Mark, J. Günther, S. Boulos, T. Ize, W. Hunt, S. G. Parker, and P. Shirley, "State of the Art in Ray Tracing Animated Scenes," *Computer Graphics Forum*, vol. 28, pp. 1691-1722, 2009.
- [8] D. Scherzer, L. Yang, O. Mattausch, D. Nehab, P. V. Sander, M. Wimmer, and E. Eisemann, "A Survey on Temporal Coherence Methods in Real-Time

- Rendering," in *In State of the Art Reports Eurographics*, ed. Llandudno UK, 2011.
- [9] M. Shih, Y.-F. Chiu, Y.-C. Chen, and C.-F. Chang, "Real-Time Ray Tracing with CUDA," presented at the Proceedings of the 9th International Conference on Algorithms and Architectures for Parallel Processing, Taipei, Taiwan, 2009.
- [10] J. Gunther, S. Popov, H. P. Seidel, and P. Slusallek, "Realtime Ray Tracing on GPU with BVH-based Packet Traversal," in *Interactive Ray Tracing, 2007. RT '07. IEEE Symposium on*, 2007, pp. 113-118.
- [11] S. Badt, "Two algorithms for taking advantage of temporal coherence in ray tracing," *The Visual Computer*, vol. 4, pp. 123-132, 1988.
- [12] E. P. Lafortune and Y. D. Willems, "Bi-Directional Path Tracing," in *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93*, ed, 1993, pp. 145-153.
- [13] M. Cohen, J. Wallace, J. Radiosity, I. Artificial, L. Iii, C. G. Langton, and E. Addison-wesley. (1993). *Radiosity and Realistic Image Synthesis*.
- [14] E. Veach and L. J. Guibas, "Metropolis light transport," presented at the Proceedings of the 24th annual conference on Computer graphics and interactive techniques, 1997.
- [15] Shirley, P. a. Morley, and R. Keith, *Realistic Ray Tracing*, 2nd ed. Natick, MA, USA: A. K. Peters, Ltd., 2003.
- [16] S. J. Adelson and L. F. Hodges, "Generating exact raytraced animation frames by reprojection," *IEEE Computer Graphics Applications*, vol. 15, pp. 43-52, 1995.

- [17] A. S. Glassner, *An Introduction to Ray Tracing*. London, UK: Academic Press Ltd., 1989.
- [18] K. Suffern, *Ray Tracing from the Ground Up*: A. K. Peters, Ltd. , 2007.
- [19] P. Shirley, M. Ashikhmin, and S. Marschner, *Fundamentals of Computer Graphics*: A K Peters, 2009.
- [20] Wikipedia. (2012, 25/5/2012). *Ray Tracing*. Available: [https://en.wikipedia.org/wiki/Ray_tracing_\(graphics\)](https://en.wikipedia.org/wiki/Ray_tracing_(graphics))
- [21] I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, "A Characterization of Ten Hidden-Surface Algorithms," *ACM Computer Survey*, vol. 6, pp. 1-55, 1974.
- [22] A. Dayal, C. Woolley, B. Watson, and D. Luebke, "Adaptive frameless rendering," presented at the ACM SIGGRAPH 2005 Courses, Los Angeles, California, 2005.
- [23] B. Walter, G. Drettakis, and S. Parker, "Interactive Rendering using the Render Cache," in *Rendering techniques '99 (Proceedings of the 10th Eurographics Workshop on Rendering)*, 1999, pp. 235-246.
- [24] B. Walter, G. Drettakis, and D. Greenberg, "Enhancing and optimizing the render cache," in *EGRW '02: Proceedings of the 13th Eurographics workshop on Rendering*, 2002, pp. 37-42.
- [25] E. Velázquez-Armendáriz, E. Lee, K. Bala, and B. Walter, "Implementing the render cache and the edge-and-point image on graphics hardware," presented at the Proceedings of Graphics Interface 2006, Quebec, Canada, 2006.

- [26] T. Zhu, R. Wang, and D. Luebke, "A GPU accelerated render cache," *Pacific Graphics (short paper)*, 2005.
- [27] S. J. Adelson and L. F. Hodges, "Visible surface ray-tracing of stereoscopic images," presented at the Proceedings of the 30th annual Southeast regional conference, Raleigh, North Carolina, 1992.
- [28] S. J. Adelson and L. F. Hodges, "Stereoscopic Ray-Tracing," *The Visual Computer*, vol. 10, pp. 127-144, 1993.
- [29] A. Es and V. Isler, "GPU based real time stereoscopic ray tracing," in *Computer and information sciences, 2007. iscis 2007. 22nd international symposium on*, 2007, pp. 1-7.
- [30] I. Sexton and P. Surman, "Stereoscopic and autostereoscopic display systems," *Signal Processing Magazine, IEEE*, vol. 16, pp. 85-99, 1999.
- [31] N. S. Holliman, "3D display systems," in *Handbook of optoelectronics.*, J. P. Dakin and R. G. W. Brown, Eds., ed: IOP Press, 2006.
- [32] A. J. Woods and C. R. Harris, "Comparing levels of crosstalk with red/cyan, blue/yellow, and green/magenta anaglyph 3D glasses," *Proceedings of SPIE Stereoscopic Displays and Applications XXI*, vol. 7253, pp. 0Q1-0Q12, 2010.
- [33] E. Dubois, "A projection method to generate anaglyph stereo images," in *Acoustics, Speech, and Signal Processing, 2001. Proceedings. (ICASSP '01). 2001 IEEE International Conference on*, 2001, pp. 1661-1664 vol.3.
- [34] I. Ideses and L. Yaroslavsky, "New Methods to Produce High Quality Color Anaglyphs for 3-D Visualization Image Analysis and Recognition." vol. 3212,

- A. Campilho and M. Kamel, Eds., ed: Springer Berlin / Heidelberg, 2004, pp. 273-280.
- [35] W. R. Sanders and D. F. McAllister, "Producing anaglyphs from synthetic images," Santa Clara, CA, USA, 2003, pp. 348-358.
- [36] S. Gateau and S. Nash, "Implementing Stereoscopic 3D in Your Applications ", ed: NVIDIA, 2010.
- [37] P. Bourke. (1999, 21/4/2012). *Calculating Stereo Pairs*. Available: <http://paulbourke.net/miscellaneous/stereographics/stereorender/>
- [38] L. F. Hodges, "Tutorial: time-multiplexed stereoscopic computer graphics," *Computer Graphics and Applications, IEEE*, vol. 12, pp. 20-30, 1992.
- [39] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU Computing," *Proceedings of the IEEE*, vol. 96, pp. 879-899, 2008.
- [40] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Trans. Comput.*, vol. 21, pp. 948-960, 1972.
- [41] E. H. Phillips, Y. Zhang, R. L. Davis, and J. D. Owens, "Rapid Aerodynamic Performance Prediction on a Cluster of Graphics Processing Units," in *Proceedings of the 47th AIAA Aerospace Sciences Meeting*, ed. Orlando, FL, 2009.
- [42] D. Kirk and W.-m. Hwu, *Programming massively parallel processors : a hands-on approach*: Morgan Kaufmann Publishers, 2010.
- [43] Nvidia, *NVIDIA CUDA Programming Guide 2.0*, 2008.

- [44] J. Stone, D. Gohara, and G. Shi, "OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems," *Computing in Science & Engineering*, vol. 12, pp. 66-73, 2010.
- [45] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "OptiX: a general purpose ray tracing engine," *ACM Trans. Graph.*, vol. 29, pp. 1-13, 2010.
- [46] NVidia, *NVIDIA® OptiX™ Ray Tracing Engine Programming Guide*: NVIDIA Corporation, 2012.
- [47] M. Pharr and G. Humphreys, *Physically Based Rendering, Second Edition: From Theory To Implementation*: Morgan Kaufmann, 2010.
- [48] L. Holger and A. C. Elster, "Real-Time Ray Tracing Using Nvidia OptiX," presented at the EUROGRAPHICS 2010, Norrköping Sweden, 2010.
- [49] M. Stich, H. Friedrich, and A. Dietrich, "Spatial splits in bounding volume hierarchies," in *Proceedings of the Conference on High Performance Graphics 2009*, ed. New Orleans, Louisiana: ACM, 2009, pp. 7-13.
- [50] M. Dabrovic. (2002). Available: <http://hdri.cgtechniques.com/~sponza/files/>
- [51] B. Curless and M. Levoy, "A volumetric method for building complex models from range images," in *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, pp. 303-312.

- [52] Z. Wang, A. C. Bovik, and L. Lu, "Why is image quality assessment so difficult?," in *Acoustics, Speech, and Signal Processing (ICASSP), 2002 IEEE International Conference on*, 2002, pp. IV-3313-IV-3316.
- [53] B. T. Phong, "Illumination for computer generated pictures," *Commun. ACM*, vol. 18, pp. 311-317, 1975.
- [54] D. Nehab, P. V. Sander, J. Lawrence, N. Tatarchuk, and J. R. Isidoro, "Accelerating real-time shading with reverse reprojection caching," presented at the Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware, San Diego, California, 2007.

VITA

Mazen Abdulaziz Al-Hagri was born on October 2nd, 1984 in Mukalla, Yemen. He obtained a B.S. degree in Computer Science with first honors from Al Al-Bayt University in Mafraq, Jordan. Upon graduation, he enrolled the College of Medicine in Hadhramout University of Science and Technology (HUCOM) in Yemen, where he established the HUCOM Information Technology Center. After one year and a half of directing the center, he started pursuing his M.S. degree in Computer Science in King Fahd University of Petroleum and Minerals. His research interests include computer graphics, parallel computing and bioinformatics. Mazen can be reached on mazen@hucom.org or mazen.abdulaziz@gmail.com.