

วิธีการการให้แสงและเงาแบบทันทีของฉากที่มีคอสติงส์โดยใช้แผนภาพระดับชั้น



นายณัฐชัย ทิพย์ประเสริฐ

สถาบันวิทยบริการ
วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต

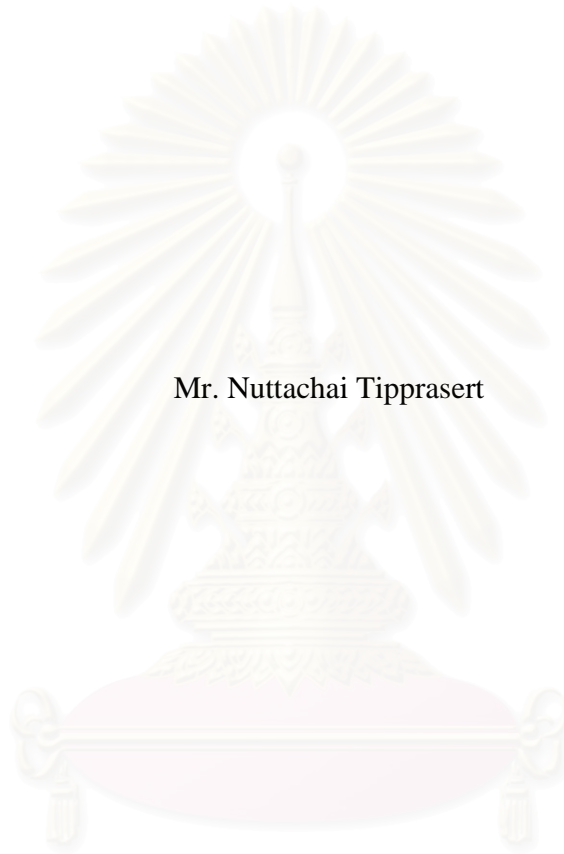
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2549

ISBN 974-17-5158-3

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

AN INTERACTIVE RENDERING METHOD FOR WATER-SIDE SCENE WITH
CAUSTICS USING LEVEL MAP METHOD



Mr. Nuttachai Tipprasert

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Engineering Program in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University


Academic Year 2006

ISBN 974-17-5158-3

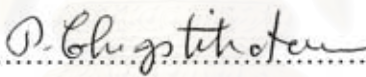
Copyright of Chulalongkorn University

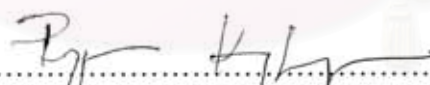
Thesis Title An Interactive Rendering Method for Water-Side Scene
With Caustics Using Level Map Method
By Mr. Nuttachai Tipprasert
Field of Study Computer Engineering
Thesis Advisor Pizzanu Kanongchaiyos, Ph.D.


Accepted by the Faculty of Engineering, Chulalongkorn University in
Partial Fulfillment of the Requirements for the Master's Degree



..... Dean of the Faculty of Engineering
(Professor Direk Lavansiri Ph.D.)

THESIS COMMITTEE


..... Chairman
(Associate Professor Prabhas Chongstitvattana)


..... Thesis Advisor
(Pizzanu Kanongchaiyos, Ph.D.)


..... Member
(Vishnu Kotrajaras, Ph.D.)


..... Member
(Supatana Auethavekiat, Ph.D.)

นายณัฐชัย ทิพย์ประเสริฐ : วิธีการการให้แสงและเงาแบบทันทีของฉากใต้น้ำที่มีคอสติกส์โดย
ใช้แผนภาพระดับชั้น. (AN INTERACTIVE RENDERING METHOD FOR WATER-SIDE
SCENE WITH CAUSTICS USING LEVEL MAP METHOD) อ. ที่ปรึกษา : ดร.พิชณู คนอง
ชัยยศ. จำนวน 106 หน้า. ISBN 974-17-5158-3

การหาวิธีการแสดงภาพของฉากใต้น้ำให้ได้อย่างสมจริงเป็นหนึ่งในหัวข้องานวิจัยในสาขาของ
คอมพิวเตอร์กราฟิกส์ที่ได้รับความสนใจเป็นอย่างมาก ในการจะเพิ่มความสมจริงให้กับฉากประเภทนี้
คอสติก (caustics) ถือเป็นส่วนประกอบสำคัญที่จะขาดไปไม่ได้ แต่ทว่ากระบวนการในการจำลอง
ภาพของคอสติกเป็นกระบวนการที่กินเวลาในการประมวลผลนาน ทำให้อัลกอริทึมที่มีอยู่ในปัจจุบันไม่
สามารถแสดงผลได้ในเวลาตอบสนอง เมื่อไม่กี่ปีมานี้ได้มีวิธีการแสดงผลที่ใช้เทกซ์เจอร์ปริมาตรซึ่ง
สามารถแสดงภาพของฉากใต้น้ำที่มีคอสติกได้ในเวลารวดเร็ว แต่ทว่าวิธีนี้มีความต้องการใช้
หน่วยความจำของการ์ดแสดงผลเป็นจำนวนมากทำให้ไม่สามารถนำไปใช้กับฉากที่ซับซ้อนได้ ใน
งานวิจัยนี้ผู้วิจัยได้นำเสนอกระบวนการแสดงภาพคอสติกในเวลาตอบสนองวิธีใหม่ที่ใช้หน่วยความจำ
น้อยกว่าเดิม วิธีการที่เราเสนอนั้นจะใช้การแทนวัตถุในฉากด้วยคู่ของเทกซ์เจอร์สี และเทกซ์เจอร์ความ
ลึก เทกซ์เจอร์สีนั้นจะถูกใช้ในการเก็บภาพของวัตถุที่เกิดการหักเหที่ผิวน้ำ ผู้วิจัยจะทำการคำนวณการ
กระจายตัวของความเข้มแสงที่ทำให้เกิดคอสติกบนเทกซ์เจอร์อันนี้ จากนั้นจึงนำผลการคำนวณที่ได้
เก็บกลับไปทีเทกซ์เจอร์แผ่นเดิม ในกระบวนการคำนวณความเข้มแสงจะมีการนำเทกซ์เจอร์ความลึก
มาช่วยเพิ่มความถูกต้องของผลการคำนวณ เพื่อให้ได้ภาพของคอสติกที่ออกมามีความสมจริงยิ่งขึ้น
จากการทดลองของแสดงให้เห็นว่าเราสามารถแสดงภาพของฉากใต้น้ำที่มีความซับซ้อนด้วยวิธีการ
ดังกล่าวนี้ได้ในเวลาตอบสนอง และจากการใช้เทกซ์เจอร์คู่สีและความลึกแทนเทกซ์เจอร์ปริมาตร ทำ
ให้สามารถลดปริมาณการใช้หน่วยความจำของการ์ดแสดงผลได้อย่างมีนัยสำคัญ

จุฬาลงกรณ์มหาวิทยาลัย

ภาควิชา.... วิศวกรรมคอมพิวเตอร์.....
สาขาวิชา....วิศวกรรมคอมพิวเตอร์.....
ปีการศึกษา2549.....

ลายมือชื่อนิสิต
ลายมือชื่ออาจารย์ที่ปรึกษา

4770284121: MAJOR Computer Engineering

KEY WORD: GLOBAL ILLUMINATION, CAUSTICS, NATURAL PHENOMENA, COLOR TEXTURE, DEPTH TEXTURE, INTERACTIVE RENDERING

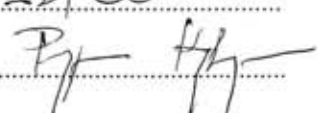
NUTTACHAI TIPPRASERT : AN INTERACTIVE RENDERING METHOD FOR WATER-SIDE SCENE WITH CAUSTICS USING LEVEL MAP METHOD. THESIS ADVISOR : PIZZANU KANONGCHAIYOS, Ph.D. 106 pp. ISBN 974-17-5158-3.

Realistic rendering of underwater scenes is one of the most anticipated research topics in computer graphics. Caustics are the important component enhancing the realism of this kind of scenes. Unfortunately, rendering caustics is a time consuming task. As a result, most existing algorithms cannot handle this at interactive rate. In recent years, volumetric texture based rendering algorithms have been proposed. They can render an underwater scene with caustics in real-time. However, these algorithms require large amount of memory and are restricted to non-complex scenes. In this thesis we present a new interactive caustics rendering algorithm which requires less memory usage. In our proposed method, we represent each object as a pair of color and depth texture. Color texture is used to store the object image viewed from viewing rays which refracted at water surface. We calculate the light intensity distribution on this image and store the result back to the color texture. The depth texture is used in the intensity calculation process to improve accuracy of the caustics patterns. Our experiment shows that proposed algorithm can handle complex underwater scene with caustics at interactive time rate. While using a pair of color and depth instead of volumetric texture, we can reduce memory usage significantly.

Department:Computer Engineering....

Student's Signature.....

Field of Study: Computer Engineering...

Advisor's Signature.....

Academic Year: ...2006.....

ACKNOWLEDGEMENTS

My first thanks goes to Pizzanu Kanongchaiyos, PhD., my thesis advisor. Without his help my life as a Master Degree student would have been much more complicated, if not almost impossible. The second thanks goes to my thesis committee , Associate Professor Prabhas Chongstitvat, Vishnu Kotrajaras, PhD., Supatana Auethavekiat, PhD., for their beneficial guidance and suggestion.

Moreover, I also want to extend this thank to every 20th floor members, my friends and all computer engineering staffs who give me invaluable advices. I particularly am please to thank every member of computer graphics research lab (CG Lab) for their generous helps, and great relationship which fulfill happiness to this life.

Finally, I deeply wish to thank my parents for their love, understanding and invaluable support throughout my graduate study



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

TABLE OF CONTENTS

	Page
Abstracts (Thai).....	iv
Abstracts (English).....	v
Acknowledgements.....	vi
Table of Contents.....	vii
List of Figures.....	xi
List of Tables.....	xvii
Chapter 1 Introduction.....	1
1.1 Background.....	1
1.2 Objectives of Study.....	3
1.3 Scopes of Study.....	3
1.4 Research Procedures.....	3
1.5 Expected Benefits.....	4
1.6 Thesis Outline.....	4
1.7 Publications.....	4
Chapter 2 Theories.....	5
2.1 Graphics Rendering Pipeline.....	5
2.1.1 Vertex Transformation.....	6
2.1.2 Primitive Assembly and Rasterization.....	6
2.1.3 Fragment Texturing and Coloring.....	7
2.1.4 Raster Operations.....	8
2.2 Programmable Graphics Hardware.....	8
2.2.1 The Evolution of Computer Graphics Hardware.....	8
2.3 Local Illumination Model.....	10
2.3.1 Ambient light.....	10
2.3.2 Diffuse reflection.....	10
2.3.3 Specular reflection.....	11
2.3.4 Derived local illumination model.....	12

	Page
2.4 Ray-tracing.....	13
2.5 Projective Texture Mapping	14
2.6 Snell's Law	16
2.7 Image Comparison	17
Chapter 3 Related works.....	19
3.1 Ray-Tracing Based Algorithm.....	19
3.2 Beam-Tracing Based Algorithm.....	20
3.3 Texture Mapping Based Algorithm	21
Chapter 4 Refractive Water Caustics Rendering Method Using Level Map.....	22
4.1 Algorithm Overview	22
4.2 Level Map Creation	23
4.2.1 Reference Plane Creation.....	23
Bounding Box Creation.....	24
Plane Slicing.....	29
4.2.2 Position Map and Diffuse Map Creation	31
4.3 Illumination Volume Creation	33
4.4 Intersection Testing.....	35
4.5 Caustics Map Creation.....	36
4.6 Caustics Casting.....	37
4.7 Algorithm Improvement	38
4.7.1 Handle Multiple Light Sources.....	38
4.7.2 Smart Slicing.....	40
Chapter 5 Algorithm Analysis	44
5.1 Memory Usage Comparison	44
5.2 Rendering Speed Comparison.....	45
5.2.1 Volumetric Based Rendering Procedure.....	45
5.2.2 Level Map Rendering Procedure with Normal Slicing.....	48
5.2.3 Level Map Rendering Procedure with Smart Slicing	50
5.2.4 Algorithm Comparison	51

	Page
Chapter 6 Discussion, Conclusions and Further Improvements	69
6.1 Discussion	69
6.2 Conclusion	71
6.3 Further Improvement	71
Appendix	73
1. Teapot	74
2. Dolphin	75
3. Sphere	76
4. Angle Fish	77
5. Box	78
6. Horse	79
7. Manta	80
8. Red Betta	81
9. Shark	82
10. Siamese Tiger	83
11. Sink	84
12. Car	85
13. Sofa	86
14. Helicopter	87
15. UFO	88
16. Chair	89
17. Barramundi	90
18. Brown Trout	91
19. Leopard Shark	92
20. Lion Head	93
21. Sand Bar Shark	94
22. Steel Head	95
23. Sun Fish	96
24. Whale	97

	Page
25. Camera	98
26. Cross.....	99
27. Bass	100
28. Plane	101
29. Plane	102
30. Hammer	103
References.....	104
Biography.....	106



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

LIST OF FIGURES

	Page
Figure 1-1: Image of swimming pool (a) computer generated (Courtesy of Cynthia Kelsey) and (b) real world.....	1
Figure 1-2: Reflective caustics (image by Don Mitchell) and refractive caustics (image by Chris Wyman).	2
Figure 2-1: The 3D objects construction process.	5
Figure 2-2: The graphics rendering pipeline.....	6
Figure 2-3: Stage of vertex transformation.....	6
Figure 2-4: Primitive assembly steps.....	7
Figure 2-5: Diffuse reflection.	11
Figure 2-6: Specular reflection model.	12
Figure 2-7: Ray-tracing rendering method.	13
Figure 2-8: Rendering method for reflective and refractive image at water surface..	14
Figure 2-9: Transformations for Conventional Camera vs. Projective Texture Mapping	15
Figure 2-10: Sample program using projective texture mapping.	16
Figure 2-11: Snell's law.....	17
Figure 3-1: Results from Wand et al. proposed algorithm.....	20
Figure 3-2: Example result from texture mapping based technique (image by Jose Stam).	21
Figure 4-1: Reference plane.....	23
Figure 4-2: Bounding box.....	24
Figure 4-3: Problem when planes are not aligned.	25
Figure 4-4: Refracted light ray at wavy surface.....	25
Figure 4-5: Reference plane alignment process.....	26
Figure 4-6: Plane alignment with various \vec{U} and \vec{W} specifications.	27
Figure 4-7: Reference plane alignment process (point light source version).	28
Figure 4-8: The initial step for plane slicing.....	30
Figure 4-9: Process of reference plane creation.....	30
Figure 4-10: View volume specification for position map rendering.....	32

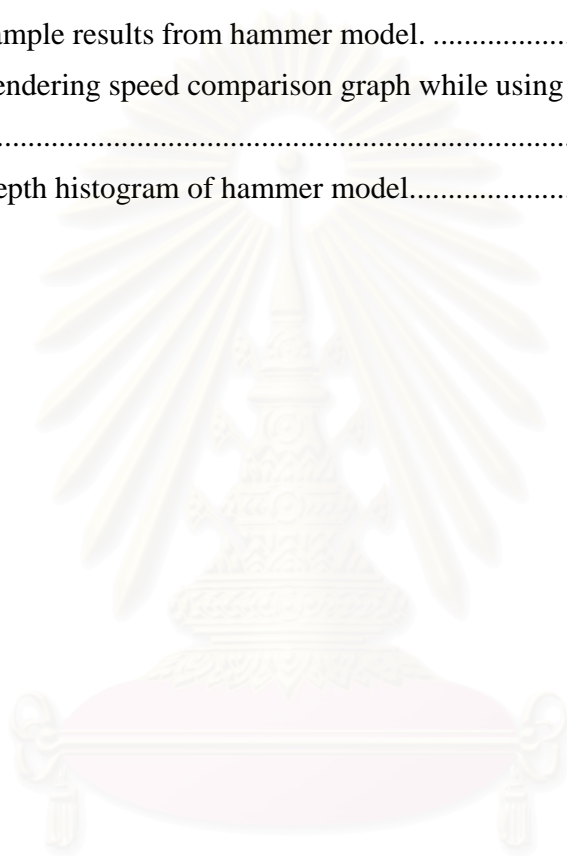
	Page
Figure 4-11: Example of position map.	33
Figure 4-12: Illumination volume.	33
Figure 4-13: Illumination volume representation.	34
Figure 4-14: Diagram of the intersection test algorithm.	35
Figure 4-15: Caustics rendering procedure.	38
Figure 4-16: Problem when the plane is not correctly aligned.	38
Figure 4-17: Caustics rendering process for multiple light sources.	39
Figure 4-18: Problem when plane is equally sliced.	41
Figure 4-19: Bar graph represents pixel's density for each depth level.	42
Figure 5-1: Flowchart shows rendering passes of volumetric based technique.	45
Figure 5-2: Illumination map creation.	46
Figure 5-3: Flow chart shows illumination map creation process.	47
Figure 5-4: Caustics map creation process.	47
Figure 5-5: Rendering passes of normal slicing algorithm.	48
Figure 5-6: Rendering passes of normal slicing when using MRT.	48
Figure 5-7: Flowchart shows normal slicing caustics map creation procedure.	49
Figure 5-8: Rendering procedure of smart slicing.	50
Figure 5-9: Depth profiling.	51
Figure 5-10: Line graph shows comparison of rendering time between level map and volumetric algorithm.	57
Figure 5-11: Line graph shows comparison of rendering time when smart slicing is turn on and off.	61
Figure 5-12: Depth profiles from sample models.	62
Figure 5-13: Comparing result when smart slicing is turned on and off.	67
Figure 6-1: Bounding volume collision testing.	70
Figure 6-2: Multi-resolution air plane. Image by Hugues Hoppe.	70
Figure 6-3: LOD in action.	70
Figure A - 1: Sample results from teapot model.	74
Figure A - 2: Rendering speed comparison graph while using teapot as an input.	74
Figure A - 3: Depth histogram of teapot model.	74
Figure A - 4: Sample results from dolphin model.	75

	Page
Figure A - 5: Rendering speed comparison graph while using dolphin as an input.	75
Figure A - 6: Depth histogram of dolphin model.	75
Figure A - 7: Sample results from sphere model.	76
Figure A - 8: Rendering speed comparison graph while using sphere as an input.	76
Figure A - 9: Depth histogram of sphere model.	76
Figure A - 10: Sample results from angle fish model.	77
Figure A - 11: Rendering speed comparison graph while using angle fish as an input.	77
Figure A - 12: Depth histogram of angle fish model.	77
Figure A - 13: Sample results from box model.	78
Figure A - 14: Rendering speed comparison graph while using box as an input.	78
Figure A - 15: Depth histogram of box model.	78
Figure A - 16: Sample results from horse model.	79
Figure A - 17: Rendering speed comparison graph while using horse as an input.	79
Figure A - 18: Depth histogram of horse model.	79
Figure A - 19: Sample results from manta model.	80
Figure A - 20: Rendering speed comparison graph while using manta as an input.	80
Figure A - 21: Depth histogram of manta model.	80
Figure A - 22: Sample results from red betta model.	81
Figure A - 23: Rendering speed comparison graph while using red betta as an input.	81
Figure A - 24: Depth histogram of red betta model.	81
Figure A - 25: Sample results from shark model.	82
Figure A - 26: Rendering speed comparison graph while using shark as an input.	82
Figure A - 27: Depth histogram of shark model.	82
Figure A - 28: Sample results from Siamese tiger model.	83
Figure A - 29: Rendering speed comparison graph while using Siamese tiger as an input.	83
Figure A - 30: Depth histogram of Siamese tiger model.	83
Figure A - 31: Sample results from sink model.	84
Figure A - 32: Rendering speed comparison graph while using sink as an input.	84

	Page
Figure A - 33: Depth histogram of sink model.....	84
Figure A - 34: Sample results from car model.....	85
Figure A - 35: Rendering speed comparison graph while using car as an input.	85
Figure A - 36: Depth histogram of car model.....	85
Figure A - 37: Sample results from sofa model.....	86
Figure A - 38: Rendering speed comparison graph while using sofa as an input.....	86
Figure A - 39: Depth histogram of sofa model.....	86
Figure A - 40: Sample results from helicopter model.....	87
Figure A - 41: Rendering speed comparison graph while using helicopter as an input.	87
Figure A - 42: Depth histogram of sofa model.....	87
Figure A - 43: Sample results from UFO model.....	88
Figure A - 44: Rendering speed comparison graph while using UFO as an input.	88
Figure A - 45: Depth histogram of UFO model.....	88
Figure A - 46: Sample results from chair model.....	89
Figure A - 47: Rendering speed comparison graph while using chair as an input.	89
Figure A - 48: Depth histogram of chair model.....	89
Figure A - 49: Sample results from barramundi model.....	90
Figure A - 50: Rendering speed comparison graph while using barramundi as an input.	90
Figure A - 51: Depth histogram of barramundi model.....	90
Figure A - 52: Sample results from brown trout model.....	91
Figure A - 53: Rendering speed comparison graph while using brown trout as an input.	91
Figure A - 54: Depth histogram of brown trout model.....	91
Figure A - 55: Sample results from leopard shark model.....	92
Figure A - 56: Rendering speed comparison graph while using leopard shark as an input.....	92
Figure A - 57: Depth histogram of leopard shark model.....	92
Figure A - 58: Sample results from lion head model.....	93

Figure A - 59: Rendering speed comparison graph while using lion head as an input.	93
Figure A - 60: Depth histogram of lion head model.....	93
Figure A - 61: Sample results from sand bar shark model.	94
Figure A - 62: Rendering speed comparison graph while using sand bar shark as an input.....	94
Figure A - 63: Depth histogram of sand bar shark model.....	94
Figure A - 64: Sample results from steel head model.....	95
Figure A - 65: Rendering speed comparison graph while using steel head as an input.	95
Figure A - 66: Depth histogram of steel head model.....	95
Figure A - 67: Sample results from sun fish model.	96
Figure A - 68: Rendering speed comparison graph while using sun fish as an input.	96
Figure A - 69: Depth histogram of sun fish model.	96
Figure A - 70: Sample results from whale model.	97
Figure A - 71: Rendering speed comparison graph while using whale as an input.....	97
Figure A - 72: Depth histogram of whale model.	97
Figure A - 73: Sample results from camera model.	98
Figure A - 74: Rendering speed comparison graph while using camera as an input.	98
Figure A - 75: Depth histogram of camera model.	98
Figure A - 76: Sample results from cross model.	99
Figure A - 77: Rendering speed comparison graph while using cross as an input.	99
Figure A - 78: Depth histogram of cross model.	99
Figure A - 79: Sample results from bass model.....	100
Figure A - 80: Rendering speed comparison graph while using bass as an input.	100
Figure A - 81: Depth histogram of bass model.....	100
Figure A - 82: Sample results from plane model.....	101
Figure A - 83: Rendering speed comparison graph while using plane as an input.	101

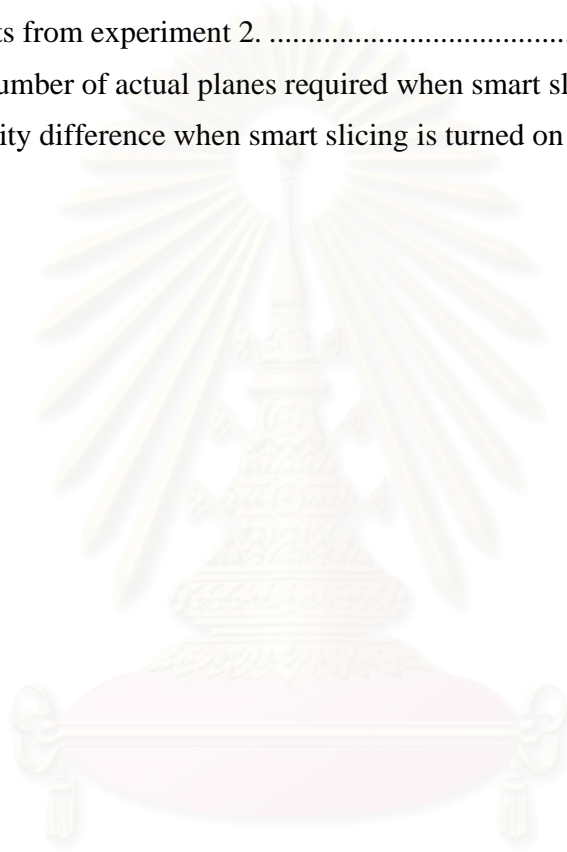
	Page
Figure A - 84: Depth histogram of plane model.....	101
Figure A - 85: Sample results from tank model.....	102
Figure A - 86: Rendering speed comparison graph while using tank as an input.	102
Figure A - 87: Depth histogram of tank model.....	102
Figure A - 88: Sample results from hammer model.	103
Figure A - 89: Rendering speed comparison graph while using hammer as an input.	103
Figure A - 90: Depth histogram of hammer model.....	103



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

LIST OF TABLES

	Page
Table 5-1: Results from experiment 1.	54
Table 5-2: Number of polygons of sample models.....	56
Table 5-3: Results from experiment 2.	58
Table 5-4: The number of actual planes required when smart slicing is used.....	63
Table 5-5: Intensity difference when smart slicing is turned on and off.	65



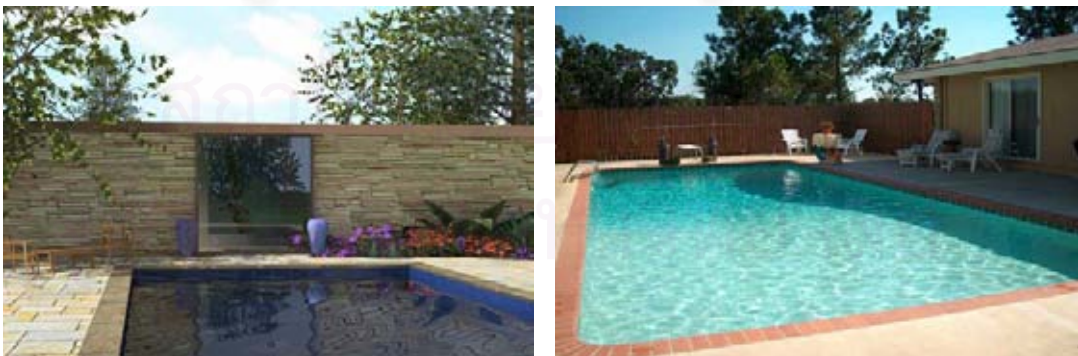
สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER 1

INTRODUCTION

1.1 Background

Realistic natural phenomena rendering is one of the most important subjects in computer graphics. Among several research topics, the realistic rendering of scenes with water is a challenge. To enhance the realism of this kind of scenes, caustics rendering is one of the most important aspects that must be taken into account. The absence of caustics in water-side scene, such as a swimming pool, makes the computer generated images look unrealistic (see Figure 1-1). But the rendering process of this phenomenon involves many path tracings and intersection tests. Therefore, it usually cannot be done in real-time. Moreover, to generate complete underwater scene as viewed from above water surface, the rendering of refracted and reflected objects images at the water surface is another subject that must be taken into consideration. Similarly, these effects also require a lot of intersection tests. As a result, the rendering of a realistic water scene seems to be more suitable for off-line rendering rather than real-time rendering. However, there are many applications, such as video games and virtual realities which require realistic real-time rendering of such scene. Therefore, the traditional rendering algorithm cannot be employed at these applications.



(a)

Figure 1-1: Image of swimming pool (a) computer generated (Courtesy of Cynthia Kelsey) and (b) real world.

To reduce the computation cost, several methods have been published. Many researchers use special hardware setting [1, 2, 3, 4, 5], such as parallel architecture, to maximize rendering speed of conventional ray-tracing algorithm. This approach can give high quality result but they are too far from reaching interactive time frame. Furthermore, the special setting for hardware means extra expense, thus, these optimization techniques are not suited with ordinary PCs. Rather than finding optimization method for ray-tracing based techniques, some researchers develop other approaches to render interactive caustics. Many research topics [6, 7] contribute themselves to texture based or image based rendering method. These methods can achieve real-time rendering capability; however, they are far from realism. Some papers [8, 9] have shown novel rendering technique which can handle both reflective and refractive caustics interactively but they are not suited for water-side scene.

In recent years, volumetric texture based water caustics rendering algorithm has been proposed [10, 11]. This technique uses volumetric textures to represent the objects in the scene and perform the intersection test on these textures instead. Even though the algorithm can achieve interactive rendering capability, it requires a lot of memory. As a consequence, this technique is limited to a simple scene that does not have many objects.

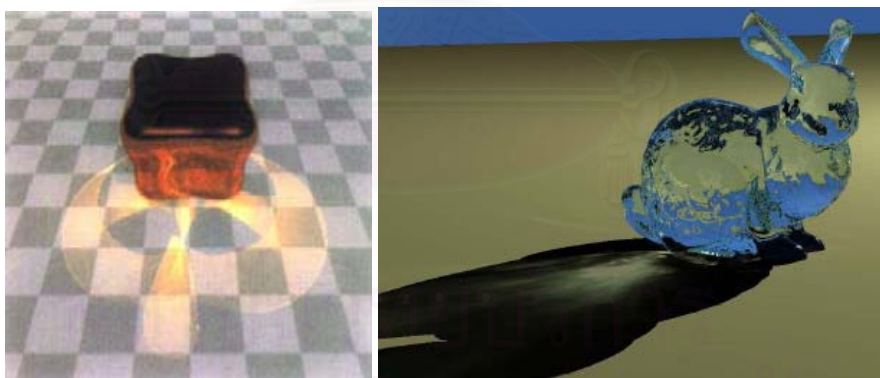


Figure 1-2: Reflective caustics (image by Don Mitchell) and refractive caustics (image by Chris Wyman).

Due to this limitation, we introduce a new interactive method for rendering underwater scene with caustics as viewed from above water. Our technique requires less memory usage. In our proposed method, an the object is represented by pairs of color and depth texture .These textures are used in both caustics casting and refracted objects rendering processes to enhance performance. Color texture is used to store the object

image viewed from viewing rays which refracted at water surface. The depth texture is used to represent 3D position of each pixel in color texture. The algorithm is accelerated by performing intersection and computing intensity distribution on texture-space instead of object-space. We are able to show that this technique can generate complex underwater scene with caustics at interactive time-rate.

1.2 Objectives of Study

The objective of this study is to present water caustics rendering algorithm that

1. Achieve interactive rendering time-rate (1 – 24 frames per second).
2. Able to cast caustics on arbitrary objects.
3. Require low memory usage.

1.3 Scopes of Study

1. This study only covers the area of water caustics rendering algorithm. Other kind of caustics or realistic water rendering method will not be the subject here.
2. This proposed algorithm requires GPU which supports shader model 3.0

1.4 Research Procedures

1. Study theories
 - a. The Fundamental Graphics theories
 - b. Global illumination theories
 - c. Ray-tracing theories
2. Research and study the previous works and analyse the advantages and disadvantages.
3. Design the algorithm.
4. Implement.
5. Develop the program as planned.
6. Test, improve and correct the program.
7. Analyse and evaluate the proposed algorithm.
8. Do the conclusion, suggestions and plan the future works.

1.5 Expected Benefits

1. This method can be used to generate water-side scene with caustics in interactive time-rate.
2. This method can reduce the memory usage that is required for rendering caustics significantly.
3. This method is suitable for interactive applications that require realistic water-side scene, such as virtual realities and games.

1.6 Thesis Outline

This thesis is organized as follows. Next chapter, theories used in this thesis are discussed. Chapter 3 gives a brief discussion about related works. Chapter 4 presents my new rendering strategy. In chapter 5, analysis of experimental results will be shown. Conclusion and further improvement are then discussed in the final chapter.

1.7 Publications

Nuttachai Tipprasert and Pizzanu Knongchaiyos 2006. An Interactive Method for Refractive Water Caustics Rendering using Color and Depth Textures. The 1st International Conference on Computer Graphics Theory and Applications (GRAPP 2006), February, Setubal, Portugal.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER 2

THEORIES

2.1 Graphics Rendering Pipeline

In the field of 3D computer graphics, every object is represented by a collection of primitives: typically polygon, line and point. Similarly, each primitive is also represented by a set of vertices. By assigning a position to each vertex and connecting them together, we can construct 3D objects. Figure 3 shows a rough sketch of this process. In this Figure, the vertices for each primitive are defined as shown in Figure (a). Next, the object primitives are then defined by connecting nearby vertices together (shown in Figure (b)). Finally, the cube object is represented by rendering a collection of primitives.

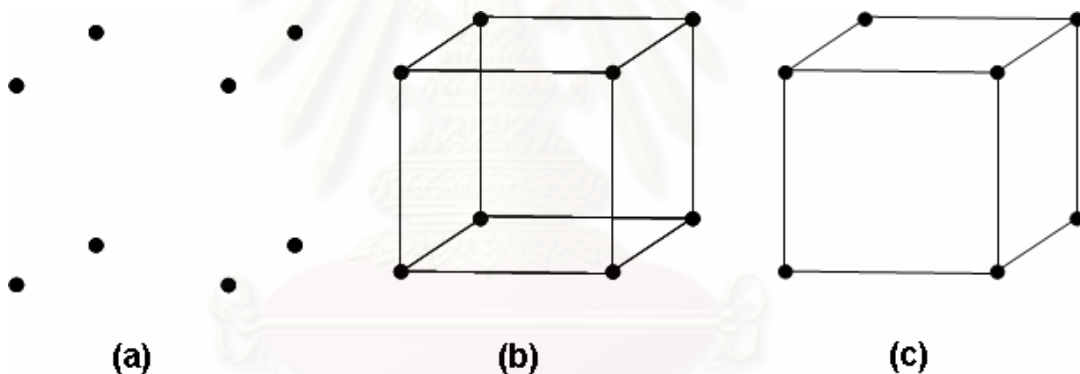


Figure 2-1: The 3D objects construction process.

In order to render 3D object, 3D graphics software must send vertices and primitives data to graphics hardware (GPU). When the GPU receives these data, it processes these data by sending them through a rendering pipeline. The rendering pipeline is a sequence of stages which is used to process incoming raw primitive data. After these data get through every stage of rendering pipeline, they can be used to render to the screen. Figure 4 shows graphics rendering pipeline used by today's graphics hardware. In the following subsection, these stages will be briefly discussed.

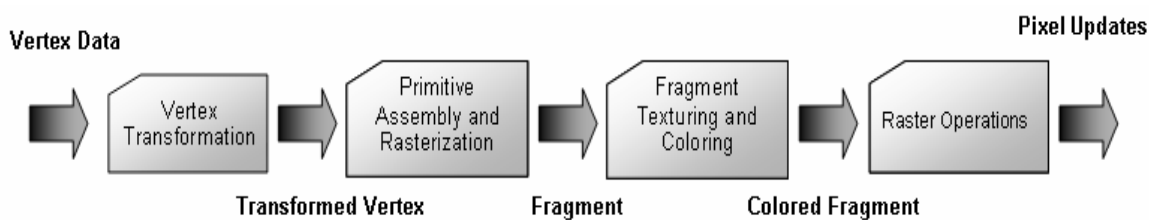


Figure 2-2: The graphics rendering pipeline.

2.1.1 Vertex Transformation

Vertex transformation is the first processing stage of the graphics rendering pipeline. It performs math operations on each vertex. The vertices which pass through this stage will be transformed from their position in local coordinate system to position in world coordinate, camera coordinate and screen coordinate system respectively. These processes of transformation are performed by various transformation matrices which shown in Figure 2-3. After the screen coordinate position of input vertex is determined, it is then used by the rasterizer and primitive assembler for computing the final color of each pixel. These operations are performed in *primitive assembly and rasterization* stage which will be discussed shortly.

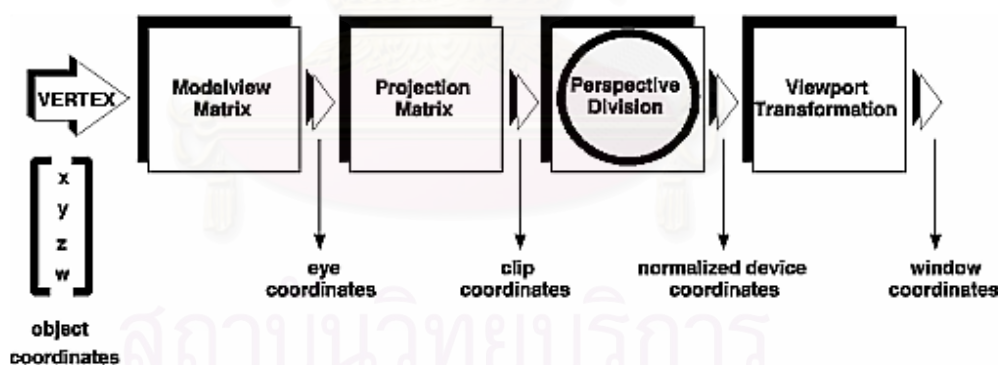


Figure 2-3: Stage of vertex transformation.

2.1.2 Primitive Assembly and Rasterization

The *primitive assembly and rasterization* is the next stage following *vertex transformation* in graphics rendering pipeline. This stage bears responsibility for primitive assembling and rasterizing task. The primitive assembly step assembles vertices into geometric primitives based on primitive batching information which accompanies

vertex collection, as shown in Figure 2-4. After that, the assembled primitive will be clipped by clipping algorithm to determine which part of primitive will be actually seen. The clipping regions are determined by a viewing frustum or by user specified clipping planes. Moreover, each primitive might be tested for its visibility by determining its facing direction. The primitive assembler may discard some polygons facing toward or away from the observer (monitor screen), depending on user specified direction. This process is known as *culling*. By default, the polygon facing away from the observer will be discarded (back face culling). The primitives which survive from clipping and culling will be sent to the rasterizer. The rasterizer bear the task for rasterizing polygon. The rasterization is a process of determining the set of pixels covered by a primitive. After passing through this stage, geometric primitives will be transformed into pixels and ready to be used by next stage.



Figure 2-4: Primitive assembly steps. From this Figure, each vertex is connected together in a specific order to form a geometric primitive.

2.1.3 Fragment Texturing and Coloring

After a set of pixels has been obtained via the rasterization, their color and the relative depth value will be assigned in the following stage. The *fragment texturing and coloring* stage is responsible for assigning a color and relative depth value to newly created pixels. The color of each pixel is obtained by performing lighting calculation and texturing on each primitive's vertex. The result color value is then interpolated and assigned to every pixel. In this process, not only vertex color is interpolated, but also relative depth value is calculated. This depth value will be used to perform depth testing in the *raster operations*, final stage of the rendering pipeline. If these pixels can survive

after some testing in the final stage, their result color value will be used to paint to frame buffer.

2.1.4 Raster Operations

The *raster operations* stage performs a final sequence of per-pixel operations immediately before updating frame buffer. The previously obtained depth value of each pixel is used in this stage to perform hidden surface removal technique known as *depth testing*. Beside depth testing, there are many per-pixel testing steps, such as scissor, alpha and stencil, performed in this stage as shown in Figure 2-4. If the incoming pixel can pass every test, its color will be blended with the previously stored pixel's color value, if blending operation is enabled, and finally painted to the frame buffer. But if the blending operation is disabled by the application, the frame buffer will be painted by the newly incoming pixel value instead.

2.2 Programmable Graphics Hardware

In this section, the concept of programmable graphics hardware is explained. A brief description on how graphics hardware evolves is given and programmable graphics pipeline is discussed in detail.

2.2.1 The Evolution of Computer Graphics Hardware

In the past decade, graphics hardware only took responsibilities for transferring pixels data to the display device and decoding video input. At that time, they were referred to as “VGA controller” and the term “GPU” was not yet introduced. But in the past few years, graphics hardware has drastically evolved, both in term of complexity and functionality. Thus, the term “VGA controller” was no longer an accurate description of graphics hardware. In the late 1990s, NVIDIA[®] Corporation introduced the term “GPU” which is an abbreviation for “Graphics Processing Unit” and the term has been used to refer to graphics rendering hardware since then.

Industry observers have identified four generations of GPU. Each generation delivers better performance and evolving programmability of the GPU feature set. The evolution of each generation is as follows.

First-Generation GPU

The GPUs in the first generation, for example, NVIDIA TNT2 ATI Rage and 3dfx's Voodoo3, are capable of rasterizing pre-transform triangles and applying one or two textures. Although these GPU can reduce the work load on CPU by moving some tasks to them, they still suffer from two limitations. First, they do not perform vertex transformation, which is the most time consuming task in graphics pipeline. Second, they have a limited set of mathematical operation for combining textures or compute final color of each pixel. As a result, CPU must take care of these works by itself.

Second-Generation GPU

The second generation of GPUs offloads 3D vertex transformation and lighting (T&L) from CPU which used to be the trademark of workstation GPU. Although, the coming of these features make PC's GPU able to handle 3D applications faster, the set of math operation for combining textures and coloring pixels are still limited. This generation is more configurable than the first generation but still not yet programmable. The examples of GPUs in this generation are NVIDIA's Geforce 256, Geforce 2, ATI's Radeon 7500 and S3's Savage3D.

Third-Generation GPU

The GPUs in this generation includes NVIDIA Geforce3 and Geforce4 Ti, Microsoft's Xbox and ATI's Radeon 8500. Rather than offering more configurations, the GPUs in this generation provide truly programmable vertex program. These GPUs let applications specify their own operation sequence for manipulating vertex data. Considerably more pixel-level configurability is available, but this mode is not powerful enough to be considered truly programmable.

Fourth-Generation GPU

This is the current generation of GPUs. The examples of GPU in this generation are NVIDIA's Geforce FX, Geforce6, Geforce7 series and ATI's Radeon 9700 and so on. These GPUs offer both vertex-level and pixel-level programmability. This level of programmability opens up the possibility of offloading complex vertex transformation

and pixel shading from CPU to GPU. This thesis also makes an advantage from these programmability features.

2.3 Local Illumination Model

The illumination model for real world lighting involved energy transfer computation between incident light ray and object surface material. Therefore, the calculation for accurate illumination is somewhat complicated. In the field of computer graphics, we use simplified illumination model to represent lighting effects on computer generated image at interactive time-rate. This illumination model simulates real-world lighting mechanism by classifying lighting component into three reflection models, ambient light, diffuse reflection and specular reflection. The following subsections will discuss about these models in details.

2.3.1 Ambient light

In the real-world, the light ray which emitted from the light source may directly hit the object surface or indirectly hit by bouncing off more than one surfaces. This bouncing light is assumed to be so scatter that there is no way to tell its original source. We call this kind of light ambient light. The ambient light has so many impacts on background lighting. It is used to illuminate some part of object that cannot be directly lit by light source. In a basic illumination model, we can incorporate background lighting by setting a general brightness level for a scene. This produces a uniform ambient lighting that is the same for all objects. The ambient light illumination equation is represented as follow

$$I_{ambient} = k_a I_a \quad (2-1)$$

where $I_{ambient}$ is a resulting intensity from ambient light, k_a is the ambient light reflection-coefficient of surface material and I_a is the intensity of ambient light source.

2.3.2 Diffuse reflection

Diffuse reflection (also known as Lambertian reflection) is the light reflection model which is used to represent material property of dull or matte surfaces such as chalk. This model assumes that the incident light is scattered with equal intensity in all

directions, independent of the viewing point. The intensity of light reflected from diffuse reflector can be computed from this equation

$$I_{diffuse} = k_d I_d (\vec{N} \cdot \vec{L}) \quad (2-2)$$

where $I_{diffuse}$ is a resulting intensity from diffuse reflection, k_d is the diffuse light reflection-coefficient of surface material, I_d is the intensity of a diffuse light coming from light source, N is the normal vector of the diffused surface and L is a normalized direction vector which points from current surface position toward light source as shown in Figure 2-5.

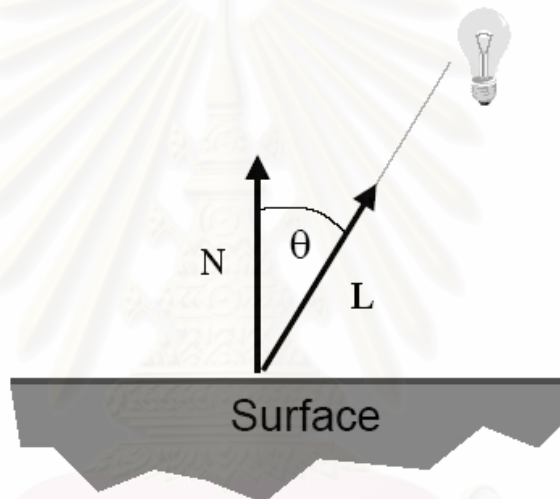


Figure 2-5: Diffuse reflection.

2.3.3 Specular reflection

Specular reflection can be observed as bright spot on a shiny surface (e.g. white dot on an apple). The specular reflection model is visualized in Figure 2-6. From this Figure, angle ϕ is a viewing angle relative to specular-reflection direction, \mathbf{R} . \mathbf{L} is a unit vector that point from incident point toward light source, and the normalized vector that directed to the viewer is represented by \mathbf{V} . For ideal reflectors (a perfect mirror), incident light is reflected only in the specular-reflection direction, and we can see reflected light only when the viewing direction and reflected direction is coincide (angle $\phi = 0$). For non ideal reflectors, specular reflection exhibits over a finite field of view around vector \mathbf{R} (shade area in Figure 2-6 (b)). Shiny surfaces have a narrow reflection angle while dull

surfaces have a wider one. The intensity of specular reflection from a light source can be obtained by this equation

$$I_{spec} = \begin{cases} k_d I_d (\vec{V} \cdot \vec{R})^{ns} & \text{if } \vec{V} \cdot \vec{R} > 0 \text{ and } \vec{N} \cdot \vec{L} > 0 \\ 0, & \text{if } \vec{V} \cdot \vec{R} < 0 \text{ and } \vec{N} \cdot \vec{L} > 0 \end{cases} \quad (2-3)$$

where ns is a specular-reflection coefficient which determined by the type of surface. A shiny surface is represented by using large value of ns while smaller values are used for the duller. For a perfect mirror, ns is infinite. A rough surface has ns value close to 1.

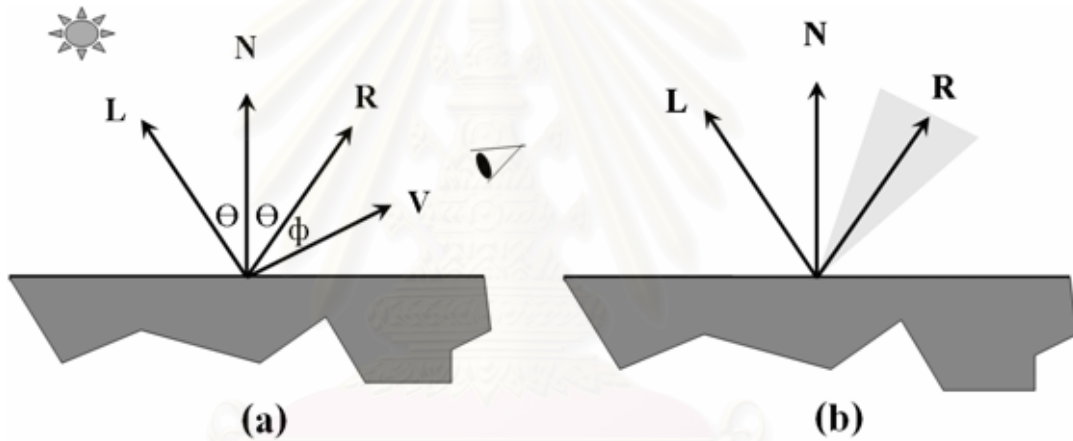


Figure 2-6: Specular reflection model.

2.3.4 Derived local illumination model

By combining previously discuss light reflection models together, basic local-illumination model can be derived as

$$I = I_{ambient} + I_{diffuse} + I_{spec} \quad (2-4)$$

where I is a resulting light intensity.

For multiple light sources, the resulting intensity at each object surface can be computed by summing up $I_{diffuse}$ and I_{spec} of each light. Equation (2-4) can be rewritten as follows

$$I = I_{ambient} + \sum_{i=0}^{numlight} (I_{diffuse} + I_{spec}) \quad (2-5)$$

2.4 Ray-tracing

Ray-tracing is a rendering technique which simulates perception mechanism of human eyes. In this technique, 3D scene is rendered by casting viewing rays from observer's eye through screen pixel, and then tracing their transmission path. Contributions to the pixel intensity are then accumulated at the intersected point (see Figure 2-7).

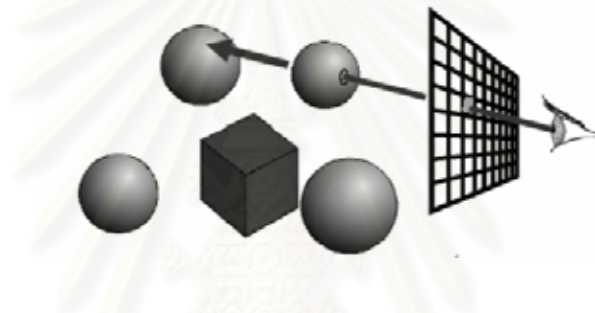


Figure 2-7: Ray-tracing rendering method.

In the case of water-side scene rendering, ray-tracing is used to generate refractive and reflective image of object on water surface. Figure 2-7 illustrates the approach. When viewing rays hit water surface, they are separated into two rays; reflected and refracted (transmitted) viewing ray. After that, we continue tracing these rays and check for their intersection. The final color of water surface at the incident point is then accumulates by combining the color of intersection point of each ray. By using this method, the reflective and refractive image at the water surface can be displayed.



Figure 2-8: Rendering method for reflective and refractive image at water surface.

2.5 Projective Texture Mapping

Projective texture mapping is a technique for generating texture coordinates dynamically via a projection of 3D geometry into a (usually 2D) texture map. This technique shares the same concept with world-to-window transformation. The object geometry is transformed from world coordinate to texture space coordinate via the transformation matrices just like conventional transformation steps discussed in section 2.1.1. However, because the valid range of texture coordinate and window coordinate are different (range of $[0,1]$ and $[-1,1]$ respectively), some modification of transformation pipeline must be taken into account. Figure 2-9 shows the comparison between two rendering pipelines.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

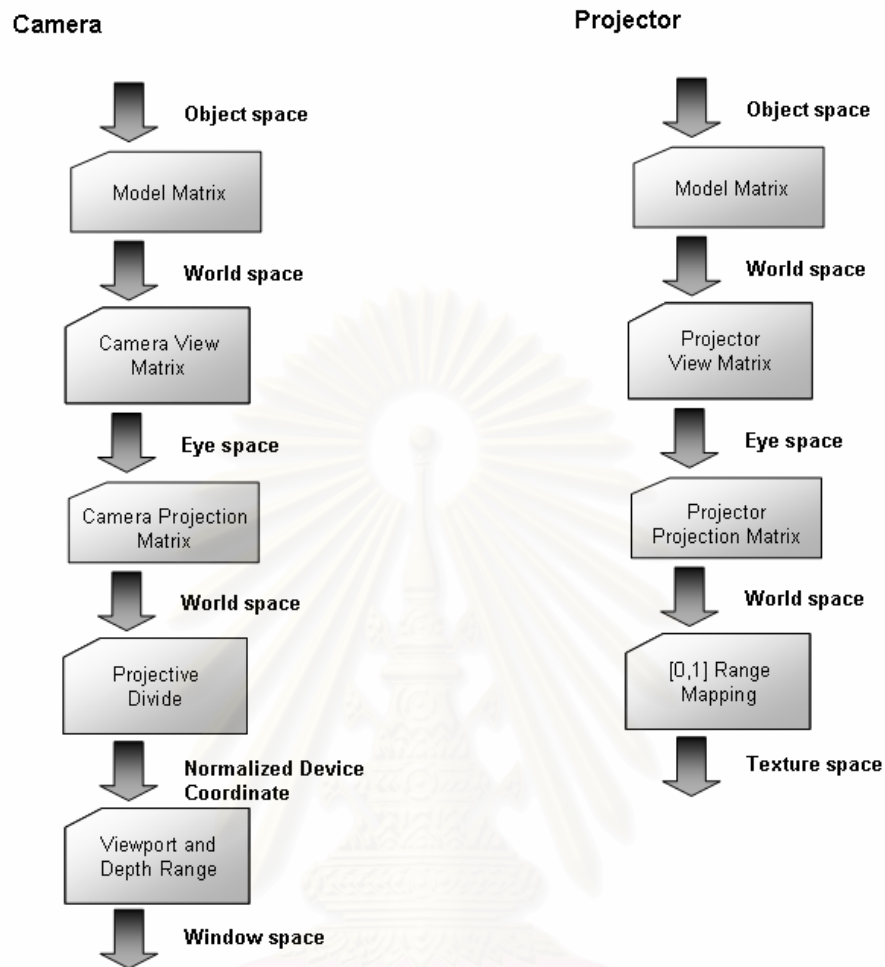


Figure 2-9: Transformations for Conventional Camera vs. Projective Texture Mapping

From Figure 2-9, when a vertex position passes through “Projection Transformation” stage, it enters the stage of range modification. After this stage, the projection space coordinate of that vertex then becomes a texture space coordinate which is valid for using index texture. Equation (2-6) shows the sequence of transformation discussed earlier.

$$\begin{bmatrix} \textit{Texture} \\ \textit{Coordinate} \end{bmatrix} = \begin{bmatrix} 0.5 & & & \\ & 0.5 & & \\ & & 0.5 & \\ & & & 1.0 \end{bmatrix} \begin{bmatrix} \textit{Light} \\ \textit{Frustum} \\ \textit{(projection)} \\ \textit{Matrix} \end{bmatrix} \begin{bmatrix} \textit{Modeling} \\ \textit{Matrix} \end{bmatrix} \begin{bmatrix} \textit{World} \\ \textit{Coordinate} \end{bmatrix}$$

(2-6)

There are several effects which can be created by projective texture mapping, for example, rendering projector like effect and shadow casting. In the field of caustics rendering, some researchers use this technique to simulate effects of water caustics [18]. Figure 2-10 shows some sample of effects created by applying this technique.

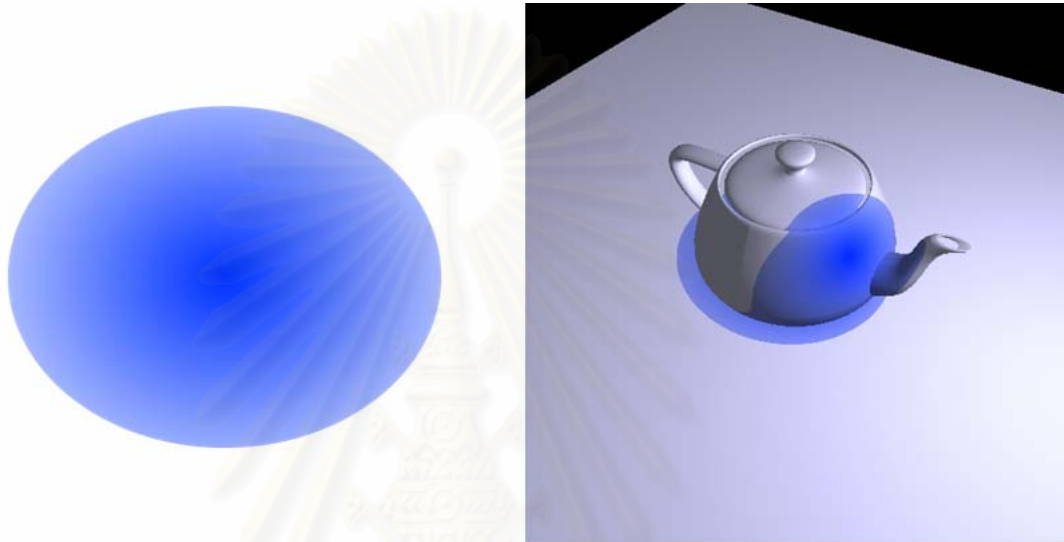


Figure 2-10: Sample program using projective texture mapping. The left image shows an image of texture used to represent projector beam. The right image shows resulting image when this texture is projected onto teapot.

2.6 Snell's Law

Snell's Law describes what happens to the wave at a boundary between two media, as shown in Figure 2-11. In the case of light wave, the example of this boundary may be the interface area between water surface and the air. When the wave changing its medium, its velocity is changed. This change in velocity also changes the moving direction of the wave. In the case of light wave, we called this phenomenon as fraction. Snell's Law is expressed mathematically by Equation 2-7.

$$\eta_1 \sin \theta_i = \eta_2 \sin \theta_T \quad (2-7)$$

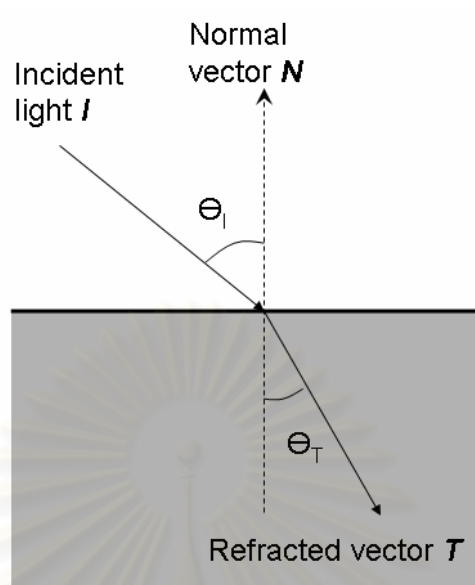


Figure 2-11: Snell's law.

where η_1 and η_2 are index of refraction of the first and the second medium respectively. θ_i is an angle of incident. θ_r is an angle of refraction. Generally, the usage of Snell's Law in computer graphics is to compute refracted vector T from a given incident vector I . In order to do this, Equation 2-7 must be revised. Equation 2-8 shows the revised equation.

$$\vec{T} = \left(\frac{\eta_1}{\eta_2} \cos \theta_i - \cos \theta_r \right) \vec{N} + \frac{\eta_1}{\eta_2} \vec{I} \quad (2-8)$$

2.7 Image Comparison

Image comparison is an error metric used to measure how two images are different from each other. There are many image comparison techniques. Some methods are based on comparing color of pixels one by one. The others also use distance between pixels in the computation. Usually, all the pixels are put into the computation. The output is the average intensity difference of the two images.

This research uses pixel-by-pixel comparison basis. It is straightforward to tell the difference between two images by looking at the number of different pixels on the images and the amount of intensity difference between corresponding pixels. Each pixel in the two images is tested to find the color difference between each other. The comparison is

performed in CIE Luv color model because this model can compute the intensity difference. If the color is stored in RGB color format, it has to be converted to CIE Luv format. The formula for finding the average intensity difference between two images is as follow.

$$\text{Average intensity difference} = \frac{\sum_i^{\text{pixels}} \sqrt{((L_{1i} - L_{2i})^2 + (u_{1i} - u_{2i})^2 + (v_{1i} - v_{2i})^2)}}{\text{pixels}} \quad (2-9)$$

where

- pixels is the number of the object pixels in the image.
- L_{1i} is the L component of the pixel i in the first image.
- L_{2i} is the L component of the pixel i in the second image.
- u_{1i} is the u component of the pixel i in the first image.
- u_{2i} is the u component of the pixel i in the second image.
- v_{1i} is the v component of the pixel i in the first image.
- v_{2i} is the v component of the pixel i in the second image.

The average intensity difference has the range between 0 to 580. The value 580 is the difference between the red color and the blue color. These two colors are the most distinct colors.

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

CHAPTER 3

RELATED WORKS

Caustics rendering has been one of the major global illumination rendering problem for more than a decade. Many algorithms have been developed to simulate this phenomenon. In this chapter, the previously proposed algorithm which related to this thesis will be briefly discussed. These works can be categorized into three classes, that is, ray-tracing based, beam-tracing based and texture mapping based algorithm. The advantage and disadvantage of each algorithm classes will be described.

3.1 Ray-Tracing Based Algorithm

Arvo [12] rendered caustics using backward ray-tracing algorithm. In this algorithm, unlike traditional ray-tracing, light rays are traced backward from light sources to the objects in the scene. This technique can be applied to various global illumination effects. There are many proposed methods extend from it. Heckbert [13] introduced the adaptive radiosity textures (rexes) for storing light distribution pattern of diffuse surfaces in the scene. Mitchell and Hanrahan [14] proposed the method for rendering caustics from curve reflectors. Jensen [15] developed a flexible global illumination rendering framework called “Photon Mapping” and demonstrated how to use it to handle caustics on arbitrary surfaces [16]. Even though these techniques can render realistic caustics, they require high computational time. There were several proposed methods that contributed toward the rendering speed optimization of these techniques [1, 2, 3, 4, 5] but these methods require special hardware setting; therefore, they limit themselves from being used on ordinary PC. Recently, Shah et al. [8] presented real-time caustics rendering algorithm based on backward ray-tracing. In order to speed up the algorithm, they created position texture and used it to store 3D world coordinate of each object in the scene, then, performed the intersection tests in the image-space. The caustics pattern is rendered by using point primitive. Though the main concept of their algorithm is similar to our work, the whole idea has so many differences in details. Besides, their algorithm suffers from alias problem, just like any other image-space algorithms.

Wand et al. [9] presented real-time caustics rendering method by discretizing the specular surfaces into sample points and projecting incoming light through these points to the diffuse receiver. In order to receive more accurate result, the large number of sample points is required, thus, this prone to scalability problem.



Figure 3-1: Results from Wand et al. proposed algorithm. The quality of resulting images is dramatically improved when more sample points used. Figure (a) Show the case when 100 sample points are used, Figure (b) 1,000 sample points and Figure (c) 10,000 sample points.

3.2 Beam-Tracing Based Algorithm

Watt [17] introduced underwater caustics rendering algorithm using backward beam-tracing, which was extended from the algorithm originally proposed by Heckbert [18]. Rather than tracing individual light rays, the backward beam-tracing traces light beam that emerge from light source and then refracts them at each polygons of water mesh. The caustics patterns are generated by accumulated light intensity that each receiver polygons receives from each participated light beam. Though the beautiful images of underwater scene can be generated from this algorithm, the computation time is also extremely long. The main problem about beam-tracing based caustics rendering algorithm is the intersection test between light beam and diffuse receiver. Nishita and Nakamae [19] solved this problem by subdividing light beam and using scan-line algorithm to determine intersection point. Their algorithm was then improved by Iwasaki et al. [20]. Iwasaki optimized the previously proposed algorithm by using hardware stencil buffer and alpha blending function to calculate caustics pattern. In the following works, Iwasaki et al. applied volume rendering technique to handle the case where the observers are above the water [10]. Their proposed method creates slice image of each receiver object in which the caustics pattern that cast on these objects can be depicted by

performing the intersection test of light beam on these images. They continue working on this method by presenting the extended algorithm for casting caustics from arbitrary refractive medium [11]. By performing intersection test on the collection of slice images instead of object mesh, the computation time is greatly reduced. However, these algorithms require large amount of texture memories; as a consequence, they are not suitable for using with complex scene.

3.3 Texture Mapping Based Algorithm

There are several methods developed for underwater caustics rendering. Stam [7] simulated underwater caustics by generating caustics textures and mapping them onto objects in the scene, Crespo [6] has proposed a method that was extended from this concept and implemented it on programmable graphics hardware, although these methods can simulate underwater caustics in real-time, the results are not visually correct due to the fact that they perform light intensity distribution calculation on flat surface.



Figure 3-2: Example result from texture mapping based technique (image by Jose Stam).

CHAPTER 4

REFRACTIVE WATER CAUSTICS RENDERING METHOD USING LEVEL MAP

4.1 Algorithm Overview

This chapter explains refractive water caustics rendering using level map method. In order to optimize the rendering speed of caustics rendering algorithm, the method for testing intersection between light beam and objects must be improved. Iwasaki et al. [9] solved this problem by using volumetric textures. These textures are used to represent an object and the intersection testing is performed on these textures instead of object polygon. Because the number of textures used in this step is much less than the number of polygons, the number of iteration steps required for finding intersection point are extremely reduced. However, this algorithm still has a memory usage problem because large amount of texture memory is required to store these images. From our observation, the heart of this strategy is a usage of these textures as a reference plane for intersection testing, not textures itself. Therefore, the volumetric texture is absolutely not necessary for testing an intersection. However, if the plane does not contain any information about an object, how do we know when light beam will hit the objects? That is where level map come into play. In this thesis, the problem mentioned above is solved via the use of “*Level Map*”.

Level map referred in this thesis is a color texture which has a depth information associates with it. A level map is consisted of three components, *diffuse map*, *position map* and *reference planes*. The level map plays an important role in the intersection test process. By applying level map to this process, the water-side scene with caustics can be rendered in interactive time. A Brief overview of level map rendering algorithm will be given shortly. In the first step of algorithm, the reference planes are defined for each object. Reference planes are a set of planes which virtually slices along some given major axis of object, as shown in Figure 4-1. These planes are used as “reference point” in the intersection test. After reference planes are defined, position map is then created. In this thesis, position map is a 2D texture storing position of every pixels of caustics receiver.

When the position map is created, the algorithm then proceeds to the next step. In this step, the refracted light ray at every water vertex is determined. These rays are used to form a collection of light beam called *illumination volume* which will be used in the intersection test step. In the intersection test step, intersection points are estimated by using position map and reference planes. These intersection points are used for forming intersection area between light beam and object surface. In the next process, we calculate light intensity distribution for each intersection area and store the result in color texture called *caustics map*. The caustics map is used to casting caustics to the scene in the caustics casting step. Finally, the result is then rendered to user screen. In the following subsection, we will describe the caustics casting process in detail.

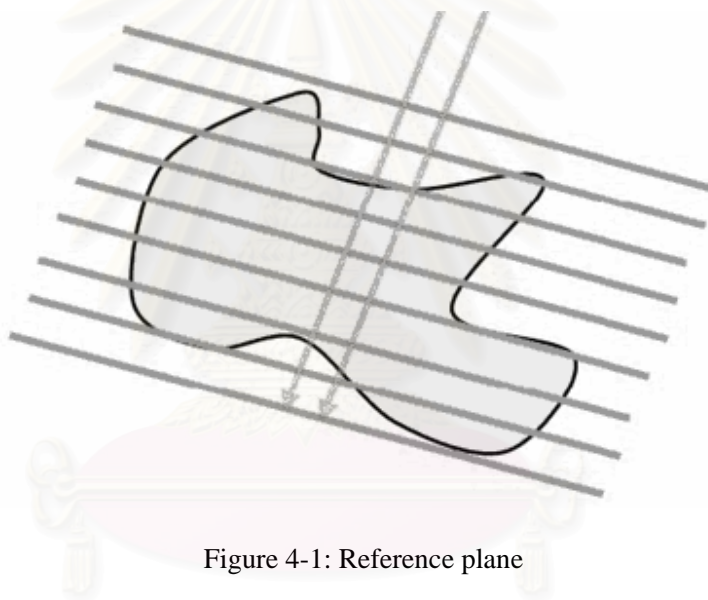


Figure 4-1: Reference plane

4.2 Level Map Creation

The level map creation process begins with the creation of reference planes and then position map. The process for creating these components will be described in the following subsection.

4.2.1 Reference Plane Creation

The reference planes are created by determining a bounding box for each object and slicing this box along a major axis. The creation process of reference planes can be divided into two sub-processes, that is, bounding box creation and plane slicing. The

bounding box creation is a step for specifying reference plane dimension and orientation. After bounding box is created, it is then sliced along its local y axis. We will use these sliced planes as our reference plane. The discussion about each process is given below.

Bounding Box Creation

The first step in the reference planes creation process is bounding box creation. In the field of computer graphics, the bounding box refers to a volume specifying box shape bounding region of an object. Figure 4-2 shows a bounding box of a sphere. In the reference planes creation process, this bounding box is used to specify reference plane dimension and orientation, which will be discussed in the upcoming subsection. This bounding box will also be used to define orthogonal view-volume which used to create position map discussed in section 4.2.2.

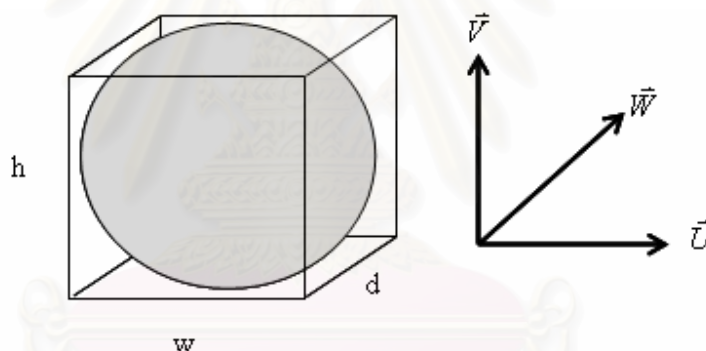


Figure 4-2: Bounding box.

A bounding box is defined by its dimension and orientation. As shown in Figure 4-2, this bounding box has width, height and depth equal to w , h and d respectively. Its orientation is defined by vectors \vec{U} , \vec{V} and \vec{W} . These vectors are referred to local axes of this bounding box. The bounding box creation algorithm is divided into two steps. First we must determine the orientation of a box. Next, we specify its dimension.

In order to get the most accurate result, the reference planes must be as perpendicular to refracted light ray as possible. Otherwise, some light beam will miss the plane and undesirable result may be noticed, as shown in Figure 4-3.

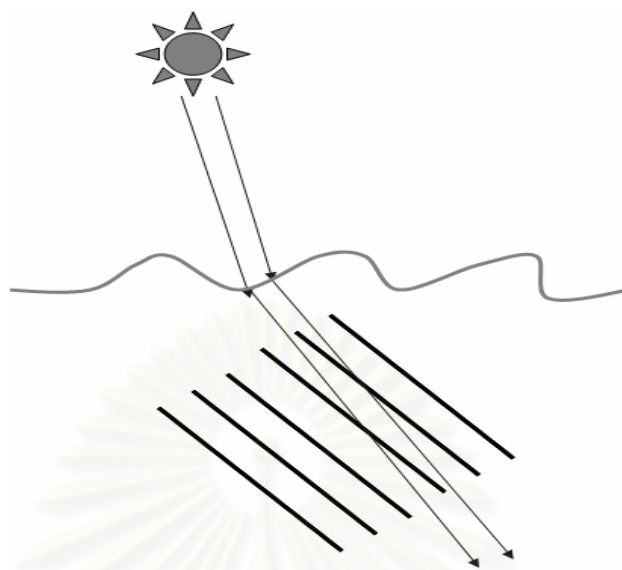


Figure 4-3: Problem when planes are not aligned.

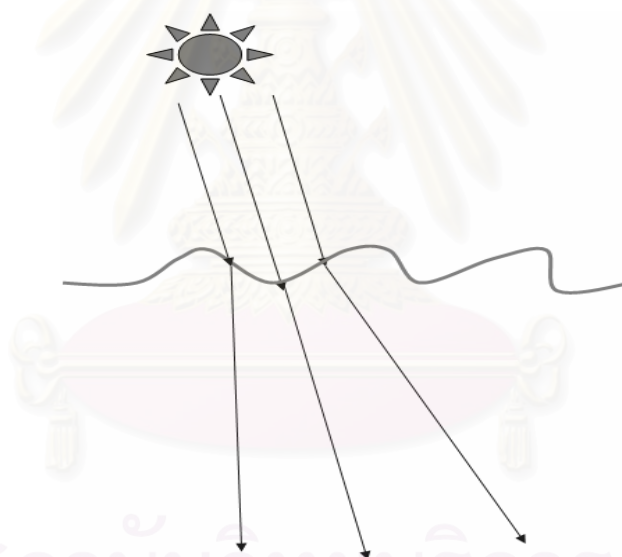


Figure 4-4: Refracted light ray at wavy surface.

Considering this issue, a problem arises, that is; how do reference planes aligned? When light ray hits the wavy water surface, it can be refracted in any direction, thus, it is difficult to determine which direction will be used to align the plane Figure 4-4 illustrates this problem. In this research, this problem is addressed assuming that the water surface is flat. In the case of flat surface, the refracted ray can be easily determined. The algorithm for reference plane alignment is as follows. Let S be a parallel light source and

B be a bounding box of an object which have vectors \vec{U} , \vec{V} and \vec{W} refer to its x, y and z local axis respectively. We define the direction of light rays emitted from S as vector \vec{L} . In order to compute refracted vector \vec{T} , the vector \vec{L} will be used as incident vector in Equation 2-8. After \vec{T} is determined, we then align the bounding box B by setting \vec{V} to be equal to $-\vec{T}$. Figure 4-5 visualizes the reference plane alignment process we are mentioning.

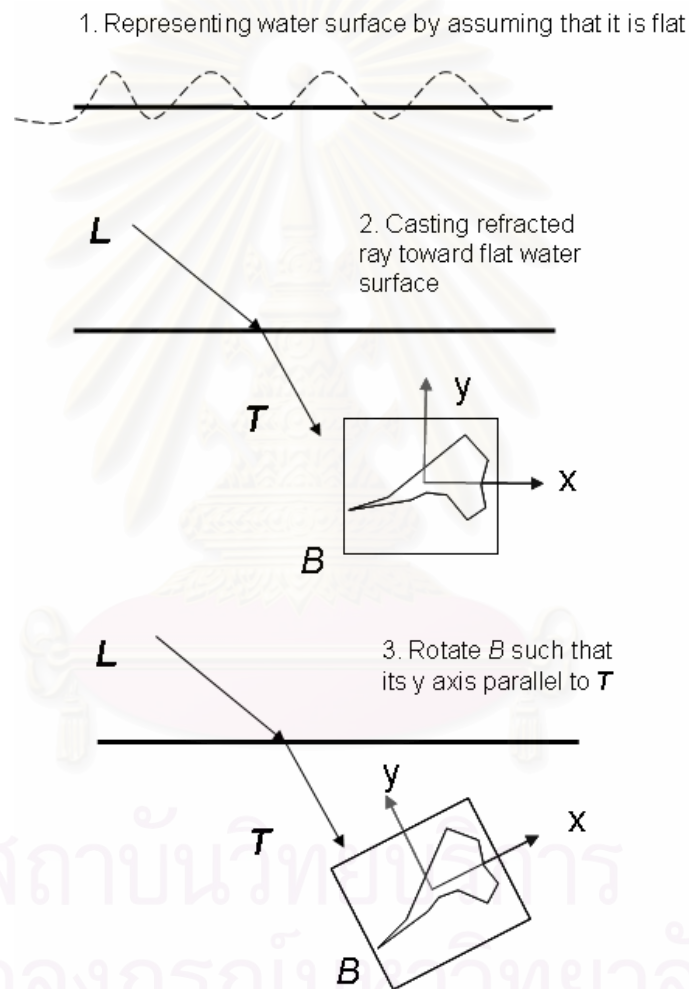


Figure 4-5: Reference plane alignment process.

In our algorithm, we do not care about how vector \vec{U} and \vec{W} of B is specified, therefore, we just assign them randomly. The vector \vec{U} and \vec{W} of B are determined by selecting a random vector \vec{N} such that \vec{N} must not be parallel to \vec{V} . Then we compute \vec{U} and \vec{W} from this equation

$$\begin{aligned}\vec{U} &= \vec{V} \times \vec{N} \\ \vec{W} &= \vec{U} \times \vec{V}\end{aligned}\quad (4-1)$$

It is safe to randomly specify these two vectors, because no matter how they are aligned, the resulting planes can perform its task perfectly, as shown in Figure 4-6. In this figure, the various plane's alignments are shown. As you can see, no matter how \vec{U} and \vec{W} are specified, they cause no problem with our reference plane creation process.

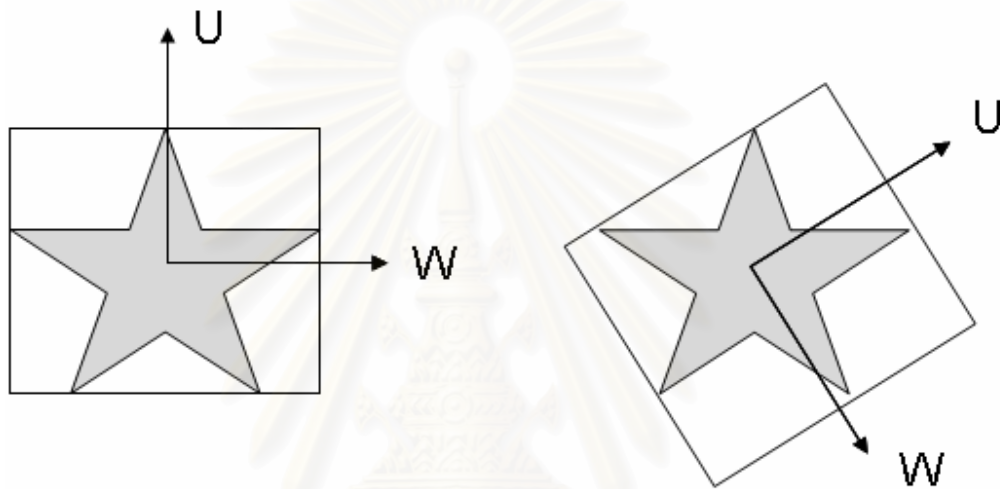


Figure 4-6: Plane alignment with various \vec{U} and \vec{W} specifications.

The algorithm explained earlier, only handles the case when parallel light source is used. When light source is a point light, some modification must be taken. Because, in the case of point light source, the light source itself does not contain any information about light direction. Thus, the direction of incident light must be determined separately. For the point light source, the direction of incident light is determined by shooting a light ray from light source to the origin point of caustics receiver objects. Let us refer to this light as \vec{I} . After \vec{I} hit water surface, the refracted vector \vec{T} is then computed from Equation 2-8 as before. The modified version of the algorithm is shown in Figure 4-7.

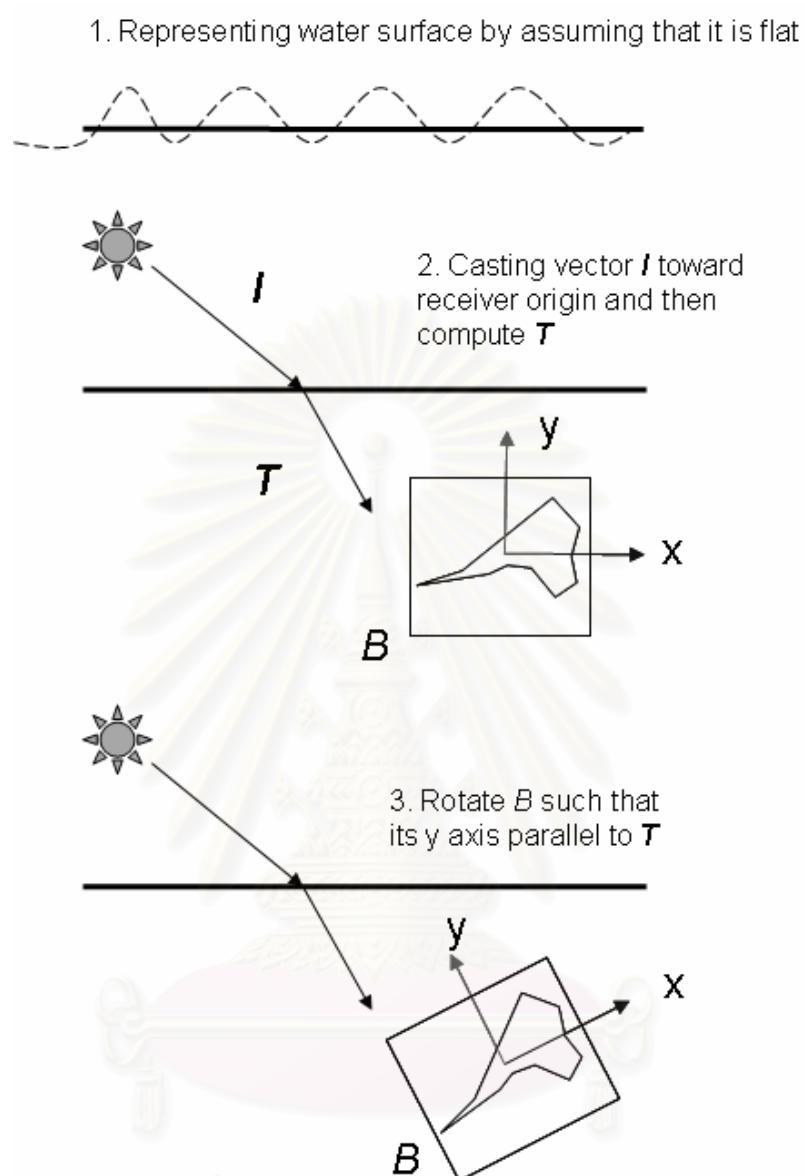


Figure 4-7: Reference plane alignment process (point light source version).

After bounding box is aligned, its dimension is determined. Let v_i be the i^{th} vertex of object O and d_i be a distance from v_i to the point at origin \vec{P}_0 , the algorithm begin by first transforming every vertex to local coordinate space of the object which is expressed by vectors \vec{U}, \vec{V} and \vec{W} computed earlier. Next, we search through every vertex to find three vertices v_x, v_y and v_z which are the vertices located in the most distance along local axis x, y and z respectively. The dimension of our bounding box can be determined from this equation:

$$\begin{aligned}
w &= 2 \|\vec{P}_x - \vec{P}_0\| \\
h &= 2 \|\vec{P}_y - \vec{P}_0\| \\
d &= 2 \|\vec{P}_z - \vec{P}_0\|
\end{aligned} \tag{4-2}$$

where P_x , P_y and P_z are coordinates of v_x , v_y and v_z respectively. But because we calculate these values at local coordinate space, the coordinate of \vec{P}_0 will be equal to (0,0,0). Thus, Equation 4-2 can be revised as follow:

$$\begin{aligned}
w &= 2 \|\vec{P}_x\| \\
h &= 2 \|\vec{P}_y\| \\
d &= 2 \|\vec{P}_z\|
\end{aligned} \tag{4-3}$$

Plane Slicing

After a bounding box is created, the reference planes are then defined by slicing the bounding box along its transformed y axis. Every reference plane used in this thesis can be easily defined by plane equation shown below:

$$RP = N_x \vec{i} + N_y \vec{j} + N_z \vec{k} + D \tag{4-4}$$

where N_x , N_y and N_z are coefficients of unit vectors i , j and k of plane's normal respectively and D is the distance between a reference point on a plan and the origin. Although this equation is sufficient for representing our reference planes, it is easier to represent them in bounding box's space. Because reference planes are aligned by making them perpendicular to the bounding box's local y axis, each normal vector now becomes $\vec{N}(0,1,0)$. Equation (4-4) is revised as follows:

$$RP = \vec{j} + D \tag{4-5}$$

From equation above, we now can create reference planes. Assuming that we have a bounding box B and want to slice it to n pieces which will create $n + 1$ planes, the definition of the i^{th} reference plane can be computed from the following formula:

$$RP = \vec{j} + D_i \tag{4-6}$$

where D_i is the distance from a reference point on a plane i^{th} to the origin. Parameter D_i from Equation (4-6) can be computed from Equation (4-7) to Equation (4-9) presented below:

$$D_i = (D_{max} - s \times (i - 1)) \tag{4-7}$$

$$s = \frac{h}{n} \tag{4-8}$$

$$D_{max} = \frac{h}{2} \tag{4-9}$$

where D_{max} is a distance from the most distance plane to the origin and s the displacement between each plane. The process of plane creation is presented in Figure 4-8 and Figure 4-9.

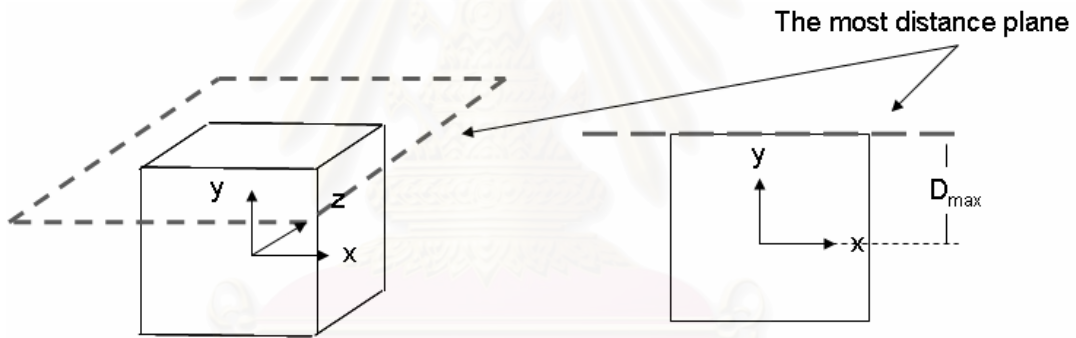


Figure 4-8: The initial step for plane slicing.

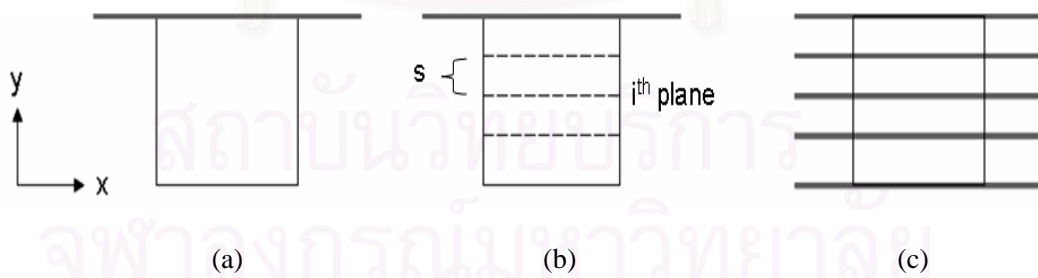


Figure 4-9: Process of reference plane creation. Figure (a) shows the most distance plane which will be use as a reference. The bounding box is then sliced in Figure (b). Figure (c) shows a final result.

4.2.2 Position Map and Diffuse Map Creation

After the reference plane is specified, the algorithm proceeds to the next step, that is, position map and diffuse map creation. These two textures store depth and diffuse color value of an object respectively. These textures play an important role in our caustics casting process which will be described shortly. The position map is created by rendering object by using orthographic projection and stores its depth to the texture memory. Diffuse map also uses the same strategy, however, instead of storing depth, it store diffuse color. In order to create these two maps, we must first determine the dimension and orientation of orthographic view volume. The view volume OV of an object is defined by:

$$OV = (lp, rp, tp, bp, np, fp, \vec{U}, \vec{V}, \vec{N}) \quad (4-10)$$

where:

- lp is the coordinate for the left clipping plane.
- rp is the coordinate for the right clipping plane.
- tp is the coordinate for the top clipping plane.
- bp is the coordinate for the bottom clipping plane.
- np is the coordinate for the near clipping plane.
- fp is the coordinate for the far clipping plane.
- \vec{U} is the vector representing local x axis for the view volume.
- \vec{V} is the vector representing local y axis for the view volume.
- \vec{N} is the vector representing local z axis for the view volume.

The dimension of OV can be easily specified by applying the dimension of the bounding box B created in the previous step. That is, we define the dimension of OV by applying this equation:

$$\begin{aligned}
 lp &= \frac{-w}{2} \\
 rp &= \frac{w}{2} \\
 bp &= \frac{-h}{2} \\
 tp &= \frac{h}{2} \\
 np &= \frac{-d}{2} \\
 fp &= \frac{d}{2}
 \end{aligned}
 \tag{4-11}$$

where w , h , and d is a width, height, and depth of B respectively.

In the case of orientation, it is defined the same way as dimension. But, we cannot directly map \vec{U} , \vec{V} , and \vec{N} to x , y and z of B . In order to create position map, we want to render the object as if we are looking along its negative y axis. Figure 4-10 illustrates this issue. In this figure, the desired result is as figure (a). But if we define the orientation of OV by directly mapping with local axis of B , the camera will look in the wrong direction as shown in figure (b).

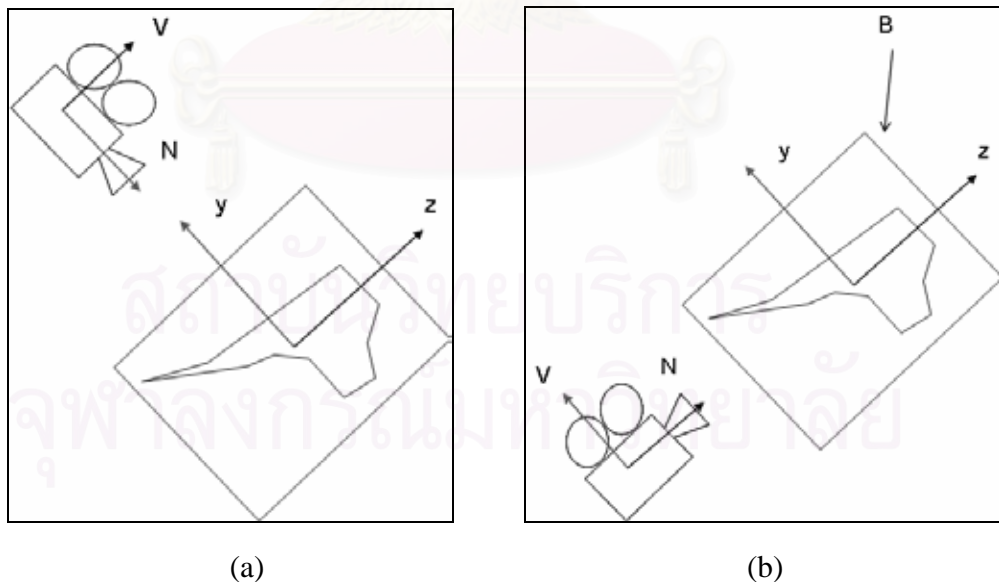


Figure 4-10: View volume specification for position map rendering.

Equation 4-13 shows the specification of OV 's orientation.

$$\begin{aligned}
 \vec{U} &= \vec{x} \\
 \vec{V} &= \vec{z} \\
 \vec{N} &= -\vec{y}
 \end{aligned}
 \tag{4-13}$$

After OV is specified, the object is then rendered to the depth and color texture memory and the creation of position and diffuse map are finished. The example of position map is shown in Figure 4-11.

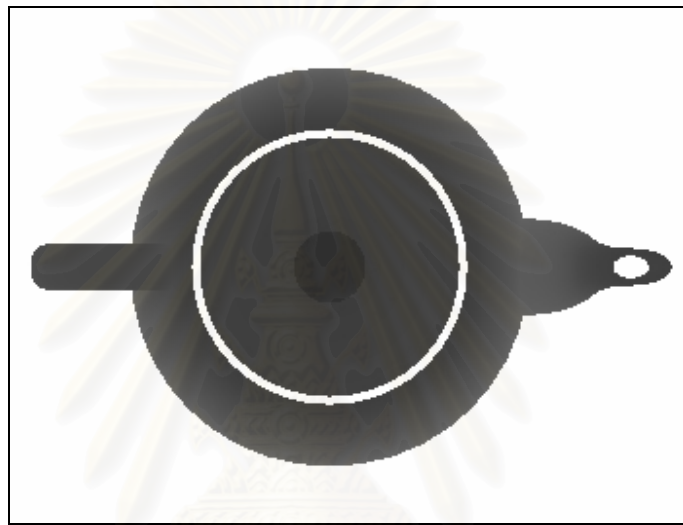


Figure 4-11: Example of position map.

4.3 Illumination Volume Creation

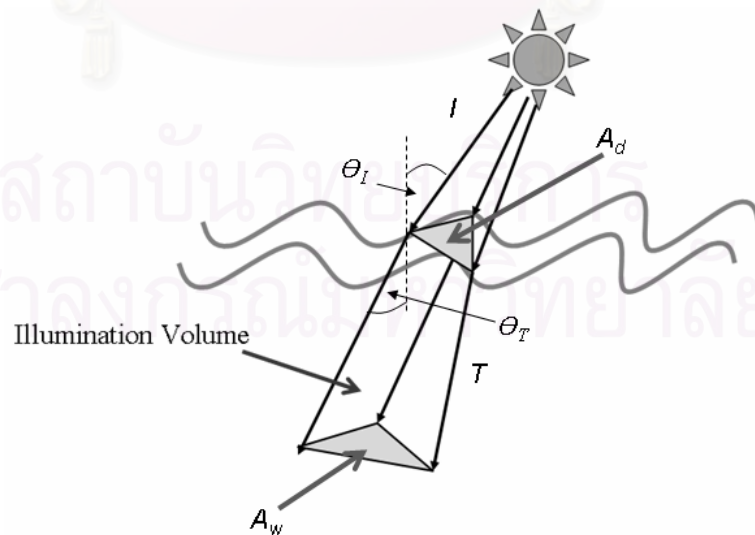


Figure 4-12: Illumination volume.

When light passes through the boundary between air and water, the incident light is refracted. The caustics pattern are formed by multiple refracted light rays converge to a single point on diffuse object geometry. This behavior can be emulated by representing water surface as triangular mesh. When the incident light rays intersect with each water triangle, they create refracted light beam. We called this beam illumination volume, as shown in Figure 4-12. In this process, we let v_i represent the i^{th} vertex and F_j is the j^{th} polygon face of water mesh. Water polygon F_j is a set of three integer value l , m and n where these values are indexing number of vertex forming this face. \vec{T}_i and \vec{T}_i is an incident light ray and a refraction vector at vertex v_i . And IL_j is an illumination volume at the j^{th} polygon face. The illumination volume creation process begins with the calculation of \vec{T}_i . We calculate \vec{T}_i by using Equation 2-8. After \vec{T}_i is determined, we extrude them by setting its size to infinity. The illumination volume IL_j is then defined by following equation:

$$IL_j = \{\vec{T}_l, \vec{T}_m, \vec{T}_n\} \quad (4-14)$$

where l, m and $n \in F_j$. Figure 4-13 shows the representation of illumination volume.

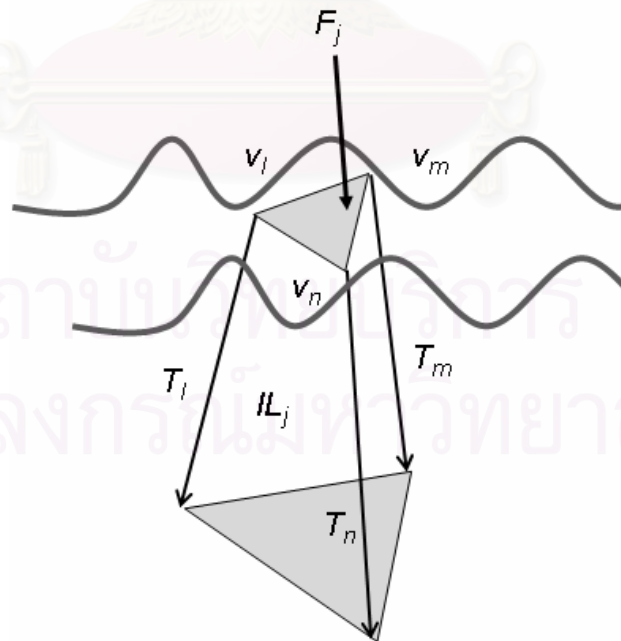


Figure 4-13: Illumination volume representation.

4.4 Intersection Testing

In the intersection test step, the level map created in previous steps is used for estimating intersection point. The intersection testing algorithm proceeds as follows. Assuming there is an object O which is associated to reference planes RP and position texture PT . The algorithm begins by casting every refracted light ray T_i through each plane P_j in a set RP . And for each intersection point $P(x, y, z)_{ij}$, we transform them into texture space coordinate. This point then becomes a point $P(x', y', z')_{ij}$ in the texture space coordinate. The x' and y' coordinate are used to index appropriate entry of PT . We refer to this entry as p' . After p' is retrieved, it is then compared against value of z' . If the differences between this two values are less than some specific threshold ε , this intersection point is then accepted, as shown in Figure 4-14. Normally, we set this acceptance threshold to be equal to the half-distance between two reference planes. If the gap between plane P_1 and plane P_2 have a length d then the value of ε will be equal to:

$$\varepsilon = \frac{d}{2} \quad (4-15)$$

Figure 4-14 visualizes the concept of intersection test algorithm explained here.

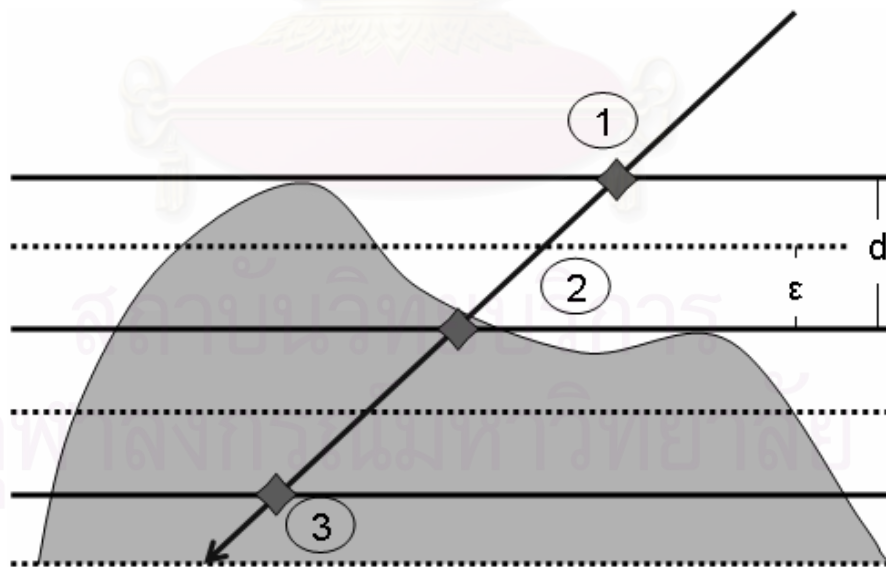


Figure 4-14: Diagram of the intersection test algorithm. The opaque lines represent reference planes, dash lines represent acceptance range of each reference plane and gray curve represents object surface. From this image, the intersection point that will be accepted is the second point.

4.5 Caustics Map Creation

Caustics map is a color texture which is used to store light intensity distribution on diffuse receiver. In order to cast caustics pattern onto receiver surface, each receiver must have its own caustics map. The caustics map creation process begins with finding intersection area of each illumination volume and object geometry. After we find the intersection areas, we compute light intensity for each of them. The intensity of intersection area can be computed from radiant equation:

$$I_c = \frac{\phi_t}{A_d \cos \theta_t} \quad (4-16)$$

where Φ_t is the total flux at intersection area, A_d and θ_t are the angle between the refracted light ray and the normal of intersection area (see Figure 4-12). The value of Φ_t in Equation (4-1) can be obtained by finding total flux Φ_i that passes through water triangle. When light travels through water, some of its energy are absorbed. Thus, the relationship between Φ_t and Φ_i can be written as:

$$\phi_t = \phi_i e^{(-Kd)} \quad (4-17)$$

where K is the absorption coefficient and d is the distant light travels through the water. Let I_i be incident light intensity. By substituting Equation 4-1 with Equation 4-2 and representing Φ_i in term of I_i , we get:

$$I_c = I_i \left(\frac{A_w \cos \theta_i}{A_d \cos \theta_t} \right) e^{(-Kd)} \quad (4-18)$$

where A_w is an area of water triangle and θ_i is an incident light angle. By accumulating the intensity of each participated intersection area, caustics pattern can be depicted.

In order to render an image of caustics pattern on underwater receivers, the intersection area between each illumination volume and each reference plane must be found. After that, the intersection triangle of each area is then drawn to a caustics map by using additive blending function. The color of each triangle vertex is determined by

calculating intensity at the intersection point. After the intersection triangle is rasterized, each pixels of the triangle is transformed into the position map coordinate. The process of transformation from world coordinate to texture coordinate has already been described in the topic of “Projective Texture Mapping” in section 2.6. The transformed coordinate of this pixel will be used to index the value of position map. This indexed value is then compared with pixel’s transformed z coordinate. If the differences of these two values are less than specific threshold, this pixel is accepted; otherwise, it will be discarded. When this process finishes, the caustics map is completed. This texture will be used in the final step of underwater caustics rendering process, that is, caustics casting. This process will be discussed in detail in the upcoming sections.

4.6 Caustics Casting

To create complete image of underwater scene, the caustics due to refractive light beam must be cast onto object surface. This can be done by applying the concept of projective texture mapping. After caustics intensity is acquired from caustics map, the final color of each pixel on diffuse receiver is computed from following equation:

$$I_o = (I_c \times I_d) + I_a \quad (4-19)$$

where I_o is the final color of the object I_d is diffuse light intensity and I_a is ambient light intensity. The following steps describe how caustics can be casted.

1. Render caustics receiver objects by applying texture space transformation matrices to them.
2. Use transformed position to index caustics map previously created in “Caustics Creation” step.
3. Compute color by using Equation 4-4.

Figure 4-15 shows the complete list of process for casting caustics

Caustics Rendering

1. Create illumination volume for each water triangle
2. Loop through these steps for each reference plane
 - 2.1. Find intersection area between illumination volume and reference plane
 - 2.2. Draw intersection triangle by using additive blending function. Specify the color of triangle with light intensity at this intersection area.
 - 2.3. Repeat following process for each fragment of intersection triangle
 - 2.3.1. Transform each fragment to depth texture space and index depth value.
 - 2.3.2. Compare indexed value and the transformed z coordinate. If the differences of these two values are less than specific threshold, we accepted it; otherwise, we discard it.
3. Render the complete scene with caustics by projecting caustics map to all caustics receivers.

Figure 4-15: Caustics rendering procedure

4.7 Algorithm Improvement

The algorithm explained so far can cast underwater caustics quite well but not perfect. To enhance the algorithm, we must make it able to handle multiple light sources and automatically slice plane. The modification process for making algorithm handle these two cases will be made in upcoming subsections.

4.7.1 Handle Multiple Light Sources.



Figure 4-16: Problem when the plane is not correctly aligned.

The overall process discussed is based on the assumption that there is only one light source in the scene. The issue this thesis not yet covered is the rendering method for caustics casted from multiple light sources. This issue leads to two new problems. That is,

how many set of reference planes should each object have and are there any solutions to align reference planes. In order to answer the first question, the second question must be answered first. If we can find any solution to align reference planes such that they are perpendicular for every light source, there is no need to create multiple set of reference planes. It is obvious that there is no way to align the planes to make them perpendicular to every light source. And if we choose to fix reference planes to some specific direction, the crack will be noticed (shown in Figure 4-16). This means that, the reference plane must be aligned to every light source. From this reason, this thesis uses multiple reference planes. For each object the number of a set of reference planes its owns must be equal to the number of light source. This issue leads to some modifications of algorithm. The Equation 4-19 then becomes:

$$I_o = \sum_{i=0}^{numlight} (I_c \times I_d) + I_a \quad (4-20)$$

while I_c and I_d is an intensity from all caustics map and diffuse light sources. And the modified version of caustics rendering process is shown in Figure 4-17.

Caustics Rendering

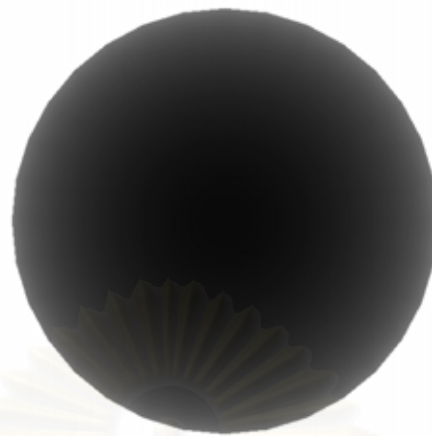
1. Create reference planes for each light source.
2. Loop through these steps for each light source:
 - 2.1. Create illumination volume for each water triangle.
 - 2.2. Loop through these steps for each reference plane:
 - 2.2.1. Find intersection area between illumination volume and reference plane.
 - 2.2.2. Draw intersection triangle by using additive blending function. Specify the color of triangle with light intensity at this intersection area.
 - 2.2.3. Repeat following process for each fragment of intersection triangle.
 - 2.2.3.1. Transform each fragment to depth texture space and index depth value.
 - 2.2.3.2. Compare indexed value and the transformed z coordinate. If the difference of these two values are less than specific threshold, we accepted it; otherwise, we discard it.
3. Render the complete scene with caustics by projecting caustics map to all caustics receivers.

Figure 4-17: Caustics rendering process for multiple light sources.

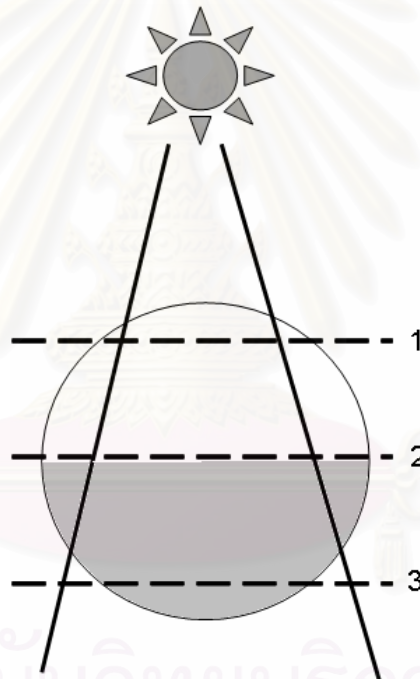
4.7.2 Smart Slicing

The process of plane slicing described in section 4.2 is performed by letting users specify number of reference planes they prefer and slicing object's bounding box by making every plane have a uniform distance between their neighbor. In some situation, this slicing strategy works fine. But in most situations, it is not necessary to equally slice the box. The reason is, reference planes used in this thesis are acted as reference points for polygon near them. If every plane represents to the same amount of polygon, using this slicing strategy for creating reference plane is reasonable. However, this situation rarely exists in practice; it does not make any senses to slice the plane the way shown in Figure 4-18. Figure 4-18 (a) shows a position map of a sphere. The dark gray zone represents relative position of each pixel of the sphere as viewed from light source. The darker the pixel, the closer it is. Notice that, there is no light gray pixel present in this figure. This means that, there are some parts of object which cannot be seen form light source. This is shown in Figure 4-18 (b), where the dashed lines represent reference planes, the gray zone of sphere is absolutely occluded form the light source, thus, the existence of plane 3 is absolutely useless. From this reason, the algorithm must be improved to make them more suitable for generic cases.





(a)



(b)

Figure 4-18: Problem when plane is equally sliced.

Because we do not want to position planes in the regions where light beam cannot intersect, the most important thing we must do is to find out where those regions are located. We, therefore, developed a new algorithm for plane slicing called “*Smart Slice*”. The smart slice algorithm proceeds as follows. First we uniformly divide level map into some amount of intervals. We refer to each interval as a depth level. The number of

divided depth level is specified by user. Then we collect data from the position map to profile the pixel density for each level. Therefore, the level which contains no data will be known. In this research, we refer to such a level as “*Empty Level*”. We then classify each empty level by setting every nearby empty level to the same group. By doing this, we are able to know which depth level is not necessary to position planes. Figure 4-19 shows this idea. In this figure, the 4th and 5th level of a level map contain no pixel, thus, we do not need to assign any reference plane in these two levels.

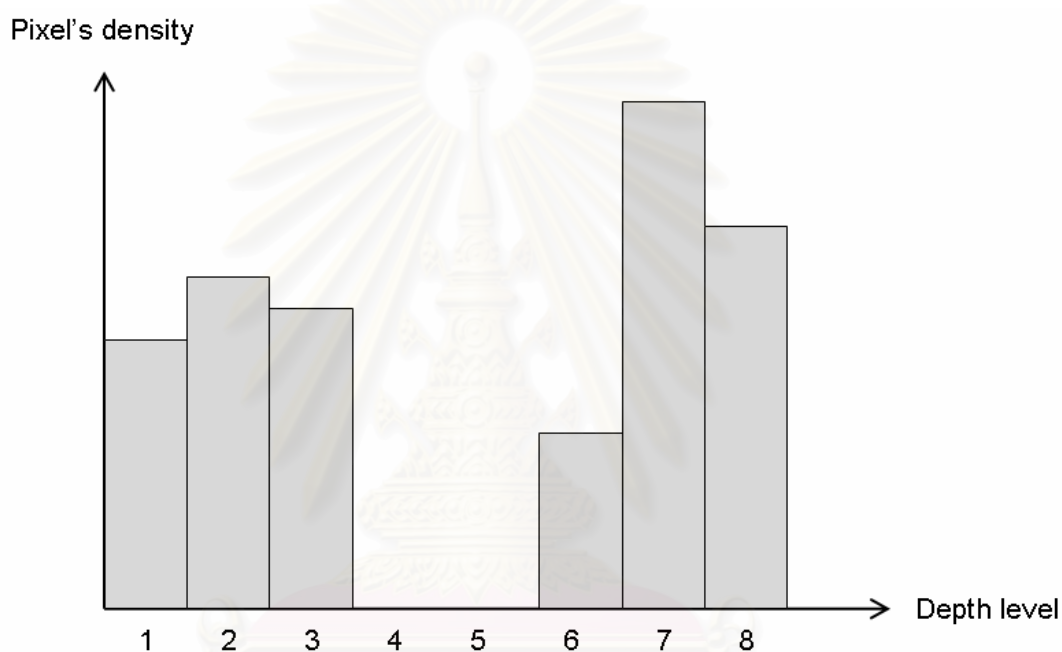


Figure 4-19: Bar graph represents pixel's density for each depth level.

After we can determine the regions where reference planes are unnecessary, the next important task is to determine the position of each reference plane. As in a case of empty level, we refer to a level which there are some pixels contained as “*Valid Level*”. By grouping nearby valid level together, we can now determine the region where the reference plane should exist.

By using this new approach, the new method for positioning reference plane must be invented. The previous method assumed that object's bounding box is a single piece box. However, because we now know that there might be some depth level absolutely occluded from light source, it is not necessary to represent this bounding box into single piece. The new bounding box is represented by dividing the original box into sub-boxes.

These sub-boxes position are determined from sets of valid level created from previous step. Assuming that we want to create level map for object in Figure 4-19, after position map is created and analyzed, the height of bounding sub-boxes be computed from this equation:

$$h_i = h \times hr_i \quad (4-21)$$

where h is the height of bounding box computed from Equation 4-3 and hr_i is a height ratio of i^{th} sub-box. The value of hr_i is computed by counting number of valid level in each set and dividing it with a number of total depth levels. After the dimension of each sub-box is computed, we then specified reference planes for every box. In this new plane slicing process, we let user specify the acceptance threshold they need. Let ε is an acceptance threshold users specify, the number of reference plane n_i for each sub-box is computed as follow:

$$n_i = \frac{hr_i}{2 \times \varepsilon} \quad (4-22)$$

After the number of reference plane is determined, the plane slicing process discussed in section 4.2.1 is then used to create reference plane for this object.

CHAPTER 5

ALGORITHM ANALYSIS

We have already described our caustics rendering algorithm in the previous chapter. In this chapter, some comparison analysis between our algorithm and previously proposed volumetric texture based rendering technique will be provided. These comparisons will mainly focus on efficiency of the algorithm; in term of rendering speed and memory consumption. The next section will provide a full description of how memory usage is greatly reduced by using our algorithm. And the final section will analyze the rendering process of each rendering algorithm in details and make a conclusion why our algorithm is the faster.

5.1 Memory Usage Comparison

Volumetric texture based rendering technique [10], [11] overcomes the problem of interactive caustics rendering by using a series of sample plane for estimating intersection point. As described in section 4.1, because the number of these sample planes is much less than the number of objects polygon, testing intersection this way is much faster than per vertex basis. At the same time, the usage of color texture for representing sample plane causes a large memory consumption problem. Consider the case where there are 100 objects in the scene, each object owns 100 sample planes and each sample plane is represented by a 16 bits color texture with 512 bytes width and 512 bytes height; it is obvious that enormous amount of memory is required to handle this scene ($100 \times 100 \times 512 \times 512 \times 2$ bytes). Compare to our proposed technique, it requires only 2 textures per objects no matter how many reference planes used. To handle a scene mentioned above, assuming that both color and depth textures use 16 bits format, our algorithm uses only 100 MB rather than those gigantic 5000 MB required in the first algorithm. There is only one situation where our algorithm uses more memory than volumetric based algorithm, that is, when each object owns only one plane. In this situation, however, both volumetric based and our proposed algorithms are not perfect choices. The best candidates are texture mapping based techniques which are described in section 3.3. Because if only one plane is supplied for intersection testing, both algorithms give the result which has no

differences with the texture mapping based methods, besides, they require extra computational step. From this reason, it can be safely concluded that, when comparing with volumetric texture based algorithm, the algorithm proposed in this thesis has a better memory management. Although there is a rare situation where volumetric based outperform our algorithms, it is not a wise decision to choose both algorithm for generating such caustics in the first place.

5.2 Rendering Speed Comparison

In this section, we will compare the rendering time from each three rendering algorithms. The comparison performed by a benchmark program developed for this purpose. It is written in C++ and OpenGL API. The input of this program is a set of 3D object model created from external authoring tools. The number of slice plane used for each algorithm is used as an independent variable. Controlled variable for this program is a water mesh and its wave function. The outputs are resulting image of each model and the rendering time for each image. This program run on Intel® Pentium® IV 2.8 GHz CPU PC with 1.25 GB system RAM and NVIDIA® Geforce® 6800 display card with 128 MB of memory. In the following sub sections, we will give a description of render procedure of each rendering algorithm. And then some of the experimental results are shown in the followed subsection. The final subsection analyzes result and gives a detailed discussion of how our rendering technique can cause rendering speed gained.

5.2.1 Volumetric Based Rendering Procedure

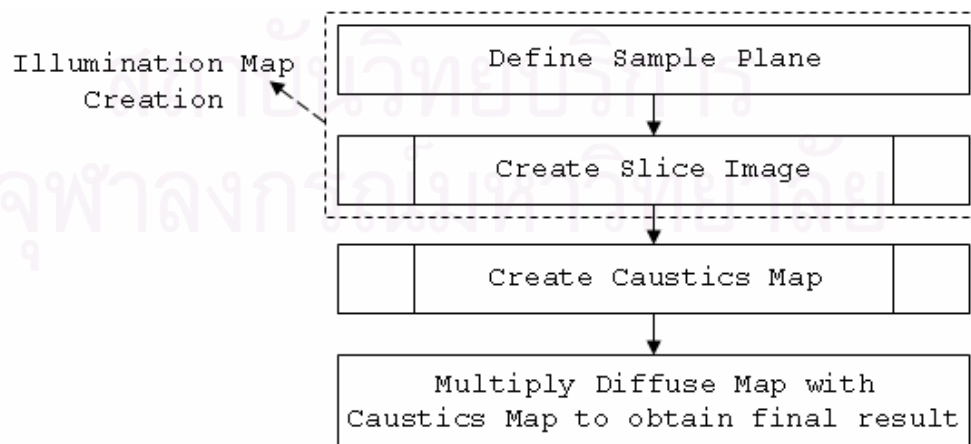


Figure 5-1: Flowchart shows rendering passes of volumetric based technique.

Figure 5-1 shows broad scope of volumetric based rendering procedure. It begins with sample plane locating. Sample planes are located by using the same strategy of locating reference plane as described in section 4.2.1. After sample planes are defined, the next step is an illumination map creation process. Illumination map in this rendering technique is a volumetric texture which stores a set of slice images of diffuse receiver. The creation process of these slice images is performed by drawing the portion of the receiver surface onto nearest sample planes. In this drawing process, the near and far clipping plane of the orthogonal view volume is defined by the distance half way through the previous and next sample plane respectively. Figure 5-2 shows this drawing operation. In this figure, all sub surfaces that exist between the dashed box are rendered to the second sample plane. A flowchart of illumination map creation process is shown in Figure 5-3.

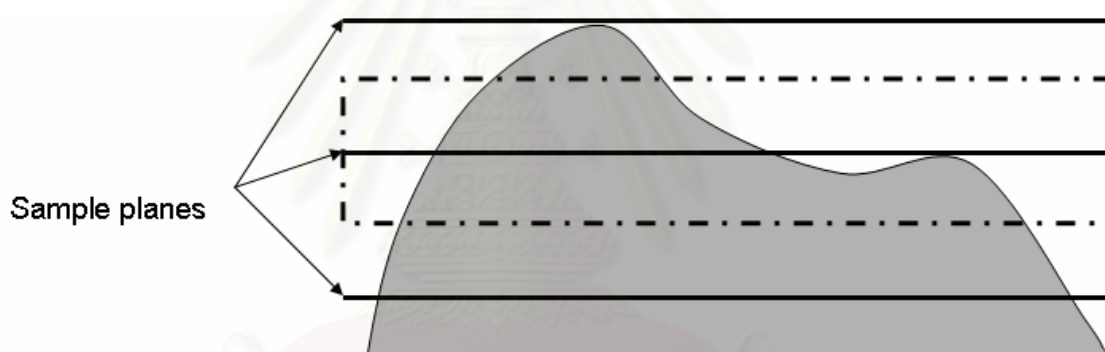


Figure 5-2: Illumination map creation.

After the illumination map is created, the algorithm proceeds to caustics map creation step. Caustics map is created by finding intersection point between each illumination volume and sample plane and rendering intersection triangle onto a color texture. After caustics map is created, it is then multiplied with illumination map and the result is stored back to the illumination map again. This process loops until every illumination map has caustics pattern casted on them. When it finishes, all illumination maps are combined together to create final result. Flowchart in Figure 5-4 shows the inner process of how caustics map is created.

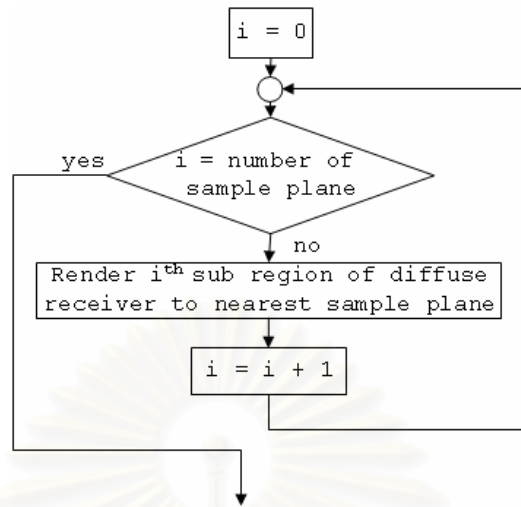


Figure 5-3: Flow chart shows illumination map creation process.

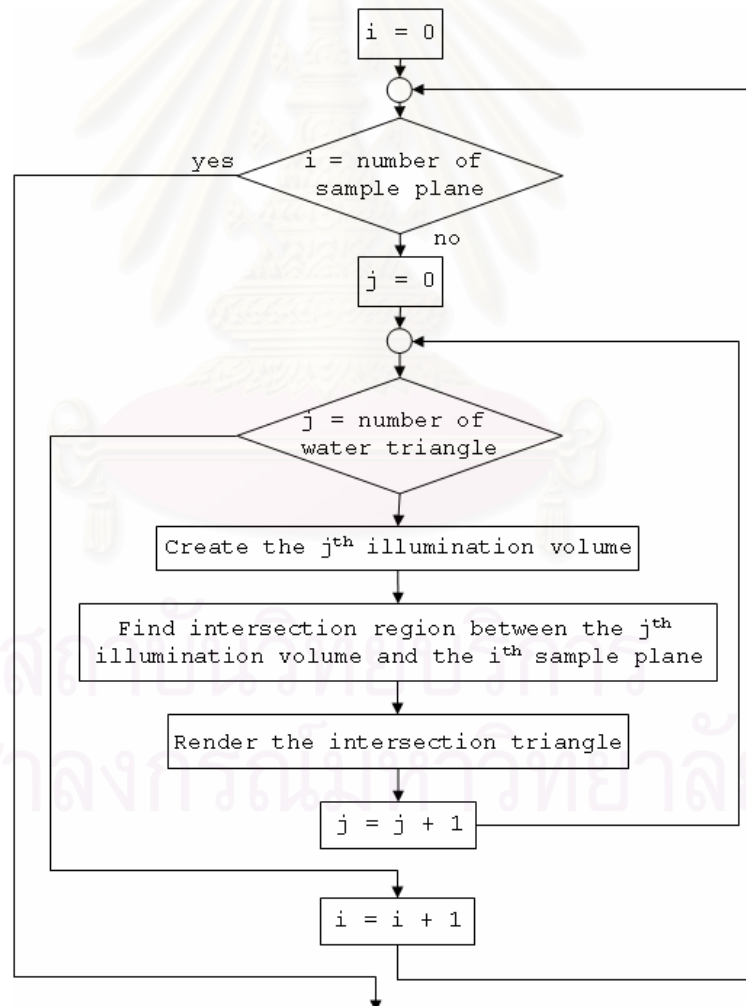


Figure 5-4: Caustics map creation process.

5.2.2 Level Map Rendering Procedure with Normal Slicing

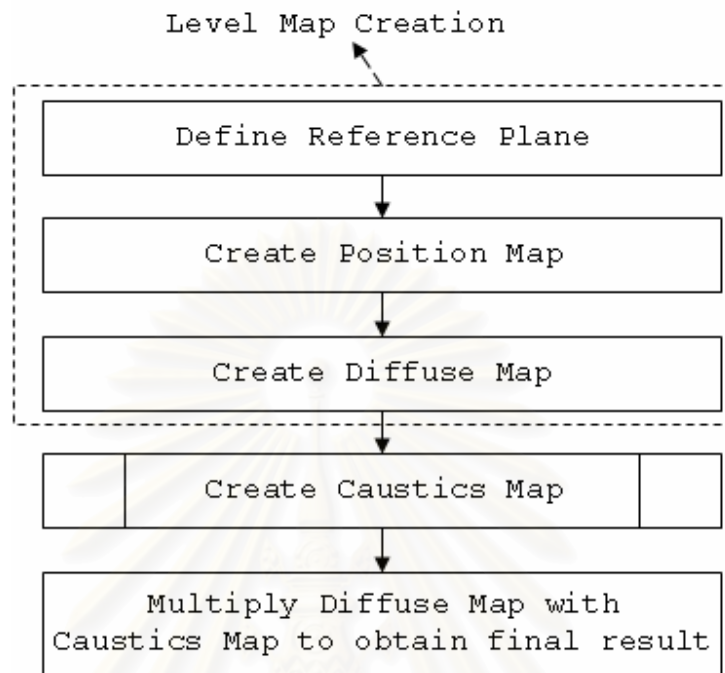


Figure 5-5: Rendering passes of normal slicing algorithm.

The process for rendering caustics using our level map algorithm has been already described in detailed in Chapter 4. It begins with reference plane defining and follows by position and diffuse map creation. The creation process of these two textures is just two rendering passes which use color and depth texture as a render target respectively. In some modern graphics hardware, these processes can be done in a single rendering pass by using multiple render target (MRT) feature. When using MRT, the rendering process presented in Figure 5-5 can be rewritten as shown in the following figure.

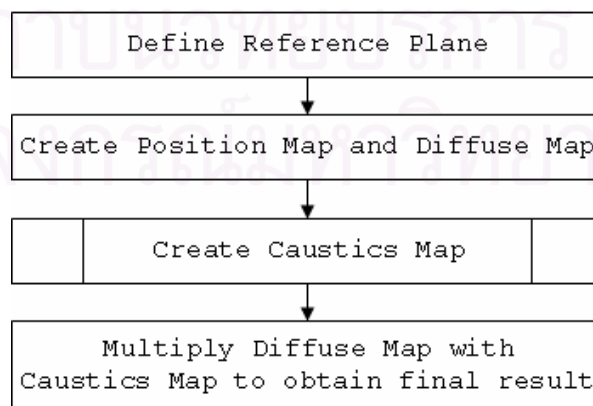


Figure 5-6: Rendering passes of normal slicing when using MRT.

After diffuse and caustics map are created, the algorithm proceeds to caustics map creation process. This process uses the same procedure as in volumetric based technique. Its flow chart is shown in Figure 5-7.

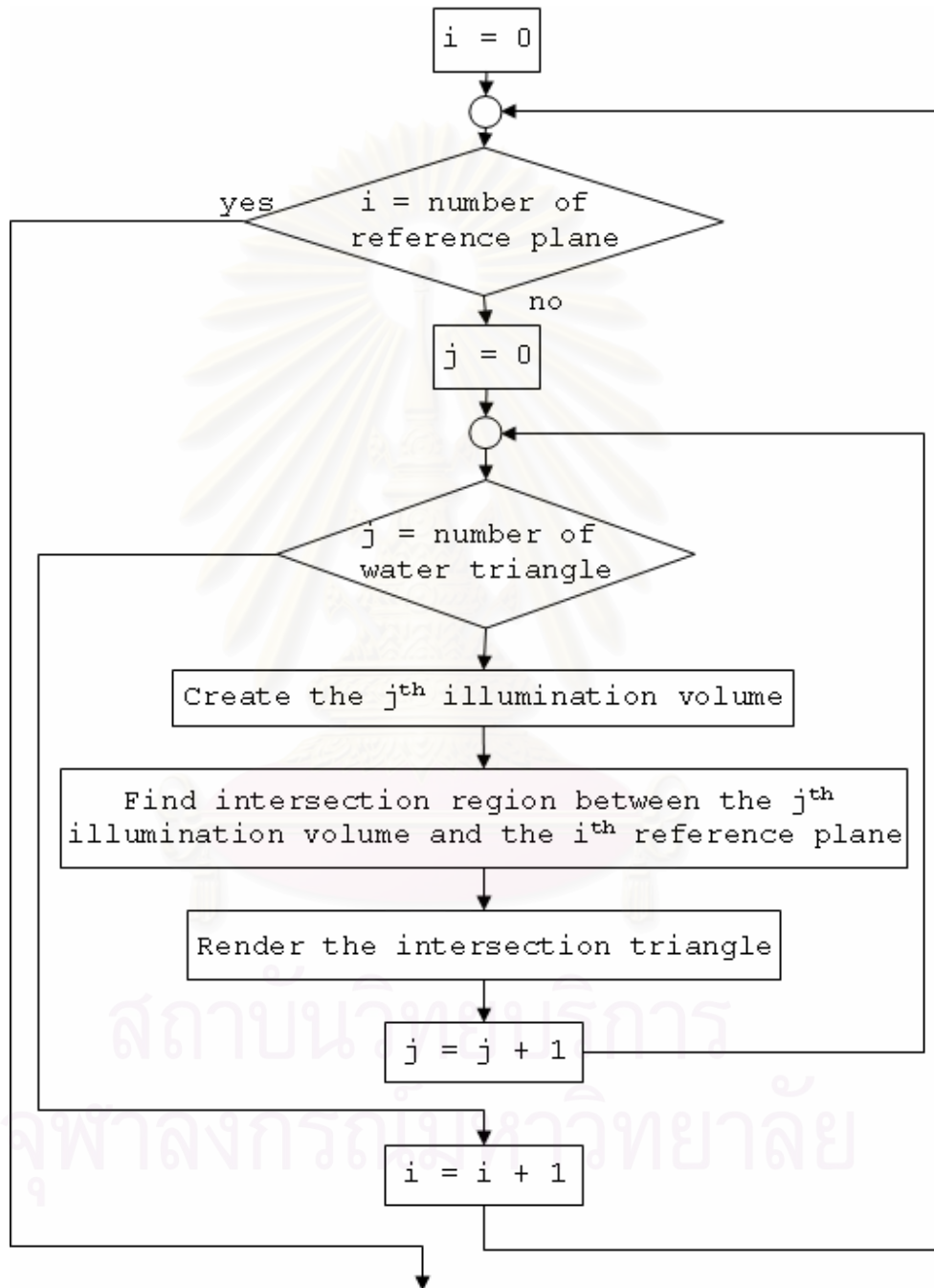


Figure 5-7: Flowchart shows normal slicing caustics map creation procedure

When finish with caustics map, same process is applied to combine caustics map with diffuse map.

5.2.3 Level Map Rendering Procedure with Smart Slicing

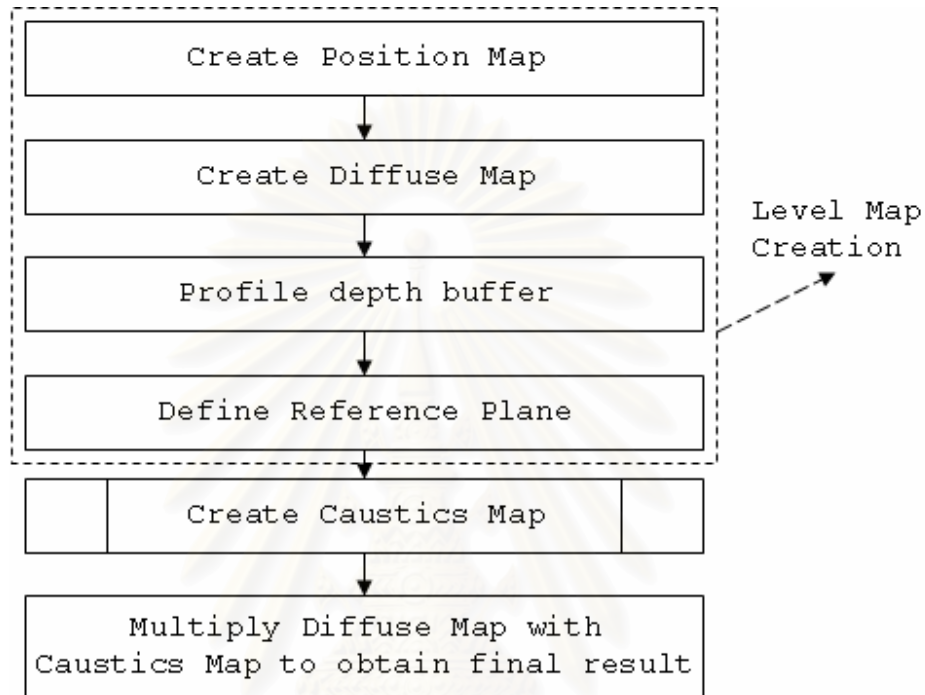


Figure 5-8: Rendering procedure of smart slicing.

As shown in Figure 5-8, smart slicing rendering procedure is very similar to normal slicing except that there is an additional depth profiling process. Depth profiling process is performed by reading depth data from graphics hardware depth buffer and then looping through every pixel for profiling its distribution density. A flowchart in Figure 5-9 shows operation step of this process.

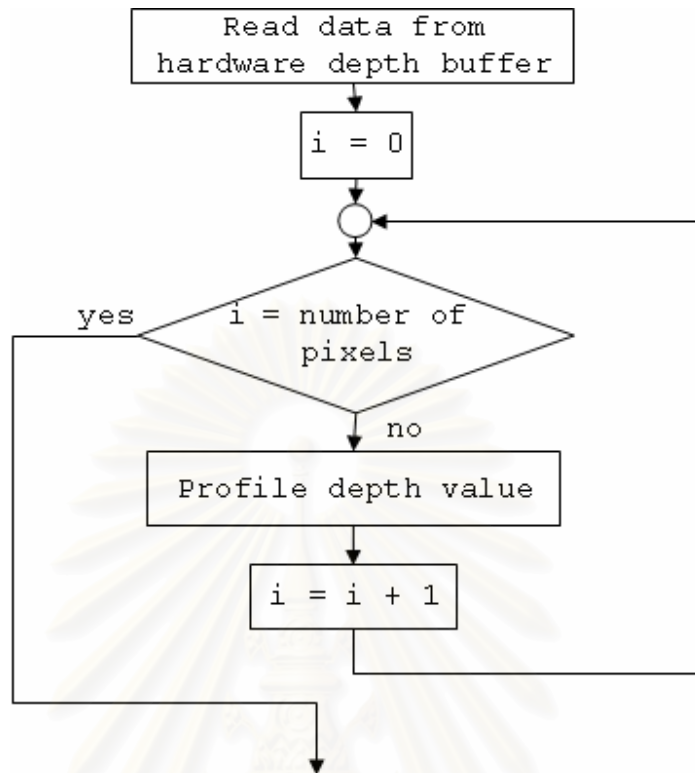


Figure 5-9: Depth profiling.

After depth profiling is finish, the caustics map creation is performed as in the other rendering algorithm. Flowchart in Figure 5-7 is, again, used to represent rendering step for caustics map creation.

5.2.4 Algorithm Comparison

Two experiments are performed in this thesis. The objective of the first experiment is to compare the rendering time between level map and volumetric based algorithm. The second experiment compares the rendering time when smart slicing is turn on and off. The test cases for both two experiments use 30 different input models as inputs. For clarity of comparison, results from three input models, teapot, sphere and dolphin, are chosen to be presented and discussed here. The experimental program is written in C++ using OpenGL API and run on Intel® Pentium® IV 2.8 GHz CPU with 1.25 GB system RAM and NVIDIA® Geforce® 6800 display card with 128 MB of memory. To achieve interactive rendering time rate, the proposed rendering technique is implemented by

taking an advantage of programmability of the GPU. We use NVIDIA[®] Cg[®] shader to create vertex and fragment program.

Experiment 1

In this experiment, we have an assumption that, when compare to volumetric base algorithm, our level map algorithm can handle an underwater scene with caustics faster. This assumption comes from the algorithm analysis discussed in previous sub sections. When take a close look at flowcharts in Figure 5-1 and Figure 5-3, we can evaluate equations for determining rendering time for both algorithms as follows:

$$T(V) = T(IL) + T(C) + T(TC) \quad (5-1)$$

$$T(LM) = T(L) + T(C) + T(TC) \quad (5-2)$$

where:

- $T(V)$ is the rendering time for overall caustics rendering process used in volumetric algorithm.
- $T(IL)$ is the processing time used for illumination map creation step.
- $T(C)$ is the processing time used for caustics map creation step.
- $T(TC)$ is the processing time used for texture combining step.
- $T(L)$ is the processing time used for level map creation step.

As we discussed earlier, the process for caustics map creation and texture combining used in both algorithms are the same. Thus, the rendering time for these processes can use the same variable in both equations. From this point, if we want to compare the rendering time of both algorithms, we must compare the processing time used for illumination map creation with the rendering time used for level map creation. As shown in Figure 5-1, the illumination map creation process can be further divided into two sub processes; plane defining and slice image creation. Thus, we can re assign $T(IL)$ in Equation 5-1 as:

$$T(IL) = T(P) + T(I) \quad (5-3)$$

where $T(P)$ is the processing time used for plane defining, and $T(I)$ is the processing time used for slice image creation. The variable $T(L)$ in Equation 5-2 can be replaced by:

$$T(L) = T(P) + T(DM) + T(PM) \quad (5-4)$$

where $T(DM)$ and $T(PM)$ are the processing time used for diffuse map and position map creation respectively. But as we have already discussed, the diffuse map and position map can be created in a single rendering pass. Thus, the Equation 5-4 can be rewritten as:

$$T(L) = T(P) + T(DP) \quad (5-5)$$

As shown in Equation 5-3 and Equation 5-5, the major difference of both algorithms are the diffuse map and caustics map and slice image creation step. Diffuse map and position map in level map technique are color texture and depth texture respectively. And slice images for illumination map are color textures. The number of slice images used in volumetric based algorithm depends on the number of sample planes used for intersection test. This means that, if 50 sample planes are required, the 50 slice images must be created. It is obvious that the processing time for illumination map creation has a direct variation with number of sample planes. On the other hand, the level map algorithm needs only two textures no matter how many planes are needed. From this reason, we can safely conclude that:

$$T(DP) \leq T(I) \quad (5-6)$$

and:

$$T(LM) \leq T(V) \quad (5-7)$$

Table 5-2 shows a number of polygon counts of each model. Figure 5-10 shows line graphs comparing results from this experiment.

Table 5-1: Results from experiment 1

Num Plane	Rendering Time					
	Teapot		Dolphin		Sphere	
	Level Map	Volumetric	Level Map	Volumetric	Level Map	Volumetric
3	0.068	0.13	0.063	0.12	0.063	0.13
4	0.083	0.184	0.08	0.16	0.079	0.184
5	0.099	0.22400001	0.095	0.199	0.093	0.22400001
6	0.115	0.27900001	0.111	0.24	0.109	0.27900001
7	0.12899999	0.31	0.127	0.279	0.123	0.31
8	0.14399999	0.361	0.142	0.33	0.14	0.361
9	0.16599999	0.39700001	0.15899999	0.353	0.156	0.39700001
10	0.17900001	0.42699999	0.175	0.396	0.17	0.42699999
11	0.191	0.46900001	0.19	0.433	0.18700001	0.46900001
12	0.205	0.53600001	0.229	0.476	0.20299999	0.53600001
13	0.22499999	0.57599998	0.222	0.513	0.21699999	0.57599998
14	0.245	0.61799997	0.244	0.553	0.236	0.61799997
15	0.25400001	0.66000003	0.25	0.592	0.25	0.66000003
16	0.26499999	0.72500002	0.26699999	0.631	0.264	0.72500002
17	0.285	0.74900001	0.28	0.67	0.28099999	0.74900001
18	0.29499999	0.75300002	0.29699999	0.711	0.29699999	0.75300002
19	0.315	0.83999997	0.31299999	0.75	0.31200001	0.83999997
20	0.32699999	0.86699998	0.33500001	0.789	0.324	0.86699998
21	0.34799999	0.87599999	0.36300001	0.829	0.34200001	0.87599999
22	0.359	0.95999998	0.366	0.867	0.35499999	0.95999998
23	0.382	0.99699998	0.38499999	0.907	0.37200001	0.99699998
24	0.39199999	1.09300005	0.39199999	0.945	0.389	1.09300005
25	0.40900001	1.08099997	0.40599999	0.989	0.40099999	1.08099997
26	0.42199999	1.14999998	0.42500001	1.028	0.41800001	1.14999998
27	0.43799999	1.14400005	0.442	1.064	0.43700001	1.14400005
28	0.44999999	1.222	0.45500001	1.106	0.447	1.222
29	0.47	1.26400006	0.47	1.149	0.47499999	1.26400006
30	0.48300001	1.36199999	0.484	1.184	0.479	1.36199999
31	0.51200002	1.37100005	0.51300001	1.222	0.495	1.37100005
32	0.51300001	1.39100003	0.51999998	1.263	0.50999999	1.39100003
33	0.52999997	1.42900002	0.52899998	1.302	0.528	1.42900002
34	0.54500002	1.47500002	0.54900002	1.347	0.53899997	1.47500002
35	0.56800002	1.55900002	0.56300002	1.409	0.55900002	1.55900002
36	0.59799999	1.53699994	0.57800001	1.421	0.57200003	1.53699994
37	0.59200001	1.63600004	0.59399998	1.477	0.588	1.63600004
38	0.60799998	1.66900003	0.61000001	1.497	0.60100001	1.66900003
39	0.62199998	1.75300002	0.62599999	1.544	0.63499999	1.75300002
40	0.667	1.73599994	0.66500002	1.591	0.63700002	1.73599994
41	0.653	1.81200004	0.671	1.682	0.64999998	1.81200004
42	0.67699999	1.82500005	0.67400002	1.664	0.667	1.82500005

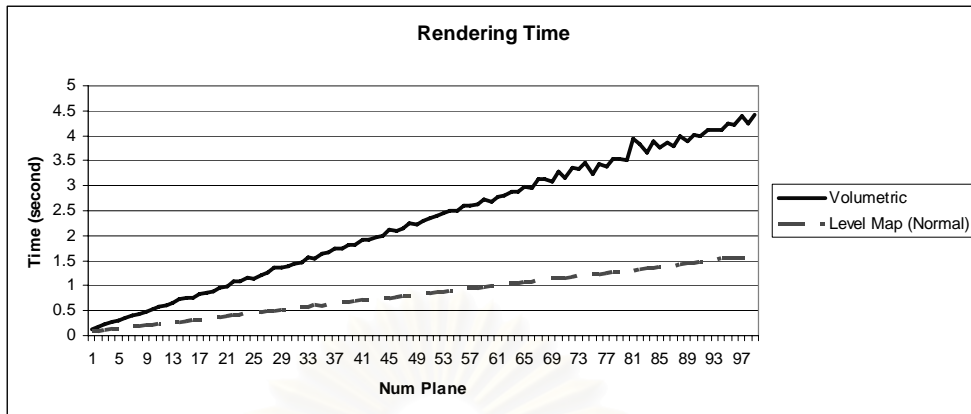
Num Plane	Rendering Time					
	Teapot		Dolphin		Sphere	
	Level Map	Volumetric	Level Map	Volumetric	Level Map	Volumetric
43	0.69599998	1.91100001	0.68800002	1.701	0.704	1.91100001
44	0.69700003	1.91299999	0.704	1.739	0.69800001	1.91299999
45	0.71499997	1.96899998	0.71899998	1.776	0.713	1.96899998
46	0.72899997	1.99899995	0.73500001	1.82	0.72600001	1.99899995
47	0.74400002	2.1329999	0.75300002	1.853	0.741	2.1329999
48	0.76200002	2.102	0.76499999	1.891	0.75800002	2.102
49	0.77499998	2.1559999	0.77999997	1.939	0.77100003	2.1559999
50	0.79299998	2.25	0.79900002	1.982	0.787	2.25
51	0.80699998	2.22300005	0.81099999	2.019	0.80299997	2.22300005
52	0.82099998	2.29800001	0.82800001	2.049	0.81900001	2.29800001
53	0.838	2.34800005	0.84299999	2.096	0.833	2.34800005
54	0.852	2.40700006	0.86400002	2.172	0.85000002	2.40700006
55	0.86900002	2.45000005	0.87400001	2.177	0.86400002	2.45000005
56	0.88499999	2.49799991	0.88800001	2.218	0.88200003	2.49799991
57	0.89600003	2.50099993	0.90600002	2.25	0.91000003	2.50099993
58	0.91399997	2.60400009	0.92199999	2.29	0.91900003	2.60400009
59	0.93300003	2.59500003	0.93900001	2.342	0.92699999	2.59500003
60	0.94400001	2.63400006	0.95200002	2.371	0.94400001	2.63400006
61	0.96799999	2.73900008	0.96899998	2.407	0.958	2.73900008
62	0.97299999	2.67400002	0.98000002	2.446	0.97299999	2.67400002
63	0.991	2.78900003	1.00699997	2.506	0.99199998	2.78900003
64	1.00699997	2.80500007	1.01699996	2.542	0.99900001	2.80500007
65	1.02499998	2.88599992	1.02999997	2.637	1.02400005	2.88599992
66	1.04299998	2.88599992	1.04499996	2.847	1.03600001	2.88599992
67	1.05299997	2.97000003	1.06299996	2.695	1.05599999	2.97000003
68	1.06700003	2.95799994	1.08200002	2.77	1.06299996	2.95799994
69	1.08000004	3.13000011	1.09899998	2.831	1.08000004	3.13000011
70	1.09599996	3.13100004	1.10599995	2.8	1.10000002	3.13100004
71	1.12699997	3.079	1.125	2.851	1.11099994	3.079
72	1.13100004	3.2809999	1.14199996	2.86	1.12699997	3.2809999
73	1.148	3.14599991	1.16499996	2.893	1.15499997	3.14599991
74	1.15999997	3.35899997	1.16999996	2.956	1.15699995	3.35899997
75	1.17799997	3.3269999	1.18700004	2.972	1.17200005	3.3269999
76	1.20200002	3.46099997	1.20099998	2.997	1.18799996	3.46099997
77	1.21399999	3.24399996	1.21899998	3.073	1.20299995	3.24399996
78	1.22300005	3.43600011	1.24300003	3.265	1.21800005	3.43600011
79	1.24000001	3.39499998	1.28900003	3.162	1.23500001	3.39499998
80	1.25100005	3.54500008	1.30599999	3.201	1.25300002	3.54500008
81	1.27199996	3.54500008	1.31700003	3.27	1.26400006	3.54500008
82	1.28699994	3.50300002	1.29700005	3.238	1.27900004	3.50300002
83	1.29900002	3.93400002	1.30999994	3.27	1.29700005	3.93400002
84	1.31500006	3.83999991	1.32500005	3.333	1.31099999	3.83999991

Num Plane	Rendering Time					
	Teapot		Dolphin		Sphere	
	Level Map	Volumetric	Level Map	Volumetric	Level Map	Volumetric
85	1.33099997	3.66400003	1.34500003	3.353	1.32599998	3.66400003
86	1.34800005	3.9000001	1.36199999	3.407	1.34300005	3.9000001
87	1.35899997	3.76600003	1.37300003	3.472	1.35399997	3.76600003
88	1.37300003	3.852	1.403	3.486	1.37100005	3.852
89	1.38999999	3.77800012	1.41799998	3.511	1.38800001	3.77800012
90	1.41900003	3.98799992	1.421	3.57	1.40199995	3.98799992
91	1.44400001	3.88100004	1.49800003	3.629	1.41900003	3.88100004
92	1.44799995	4.01100016	1.47399998	3.67	1.43599999	4.01100016
93	1.45599997	3.99600005	1.51100004	3.701	1.449	3.99600005
94	1.47099996	4.12699986	1.48800004	3.728	1.46300006	4.12699986
95	1.48199999	4.10599995	1.50199997	3.783	1.48000002	4.10599995
96	1.53499997	4.12599993	1.52999997	3.799	1.49699998	4.12599993
97	1.54200006	4.23799992	1.528	3.835	1.51499999	4.23799992
98	1.52999997	4.21199989	1.55400002	3.927	1.52699995	4.21199989
99	1.54400003	4.38399982	1.56099999	4.168	1.54100001	4.38399982
100	1.56200004	4.24300003	1.57500005	4.016	1.55299997	4.24300003
101	1.574	4.41300011	1.60099995	4.089	1.57200003	4.41300011

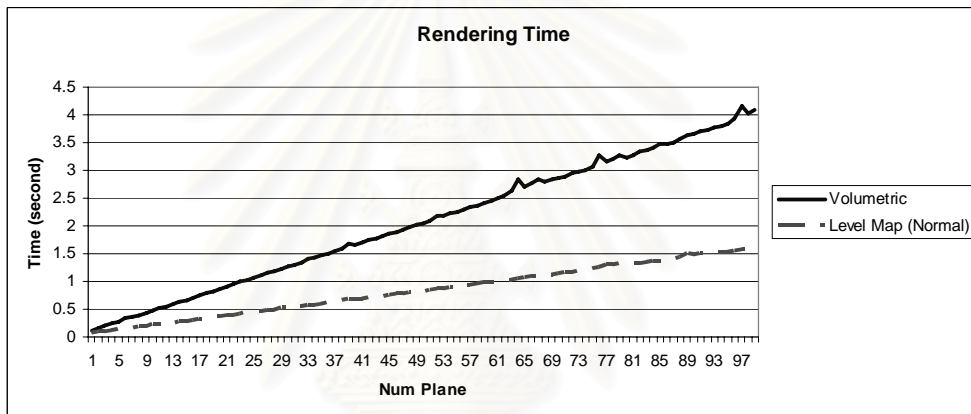
Table 5-2: Number of polygons of sample models.

Object	No. Polygon
Teapot	6,320
Dolphin	4,442
Sphere	1,984

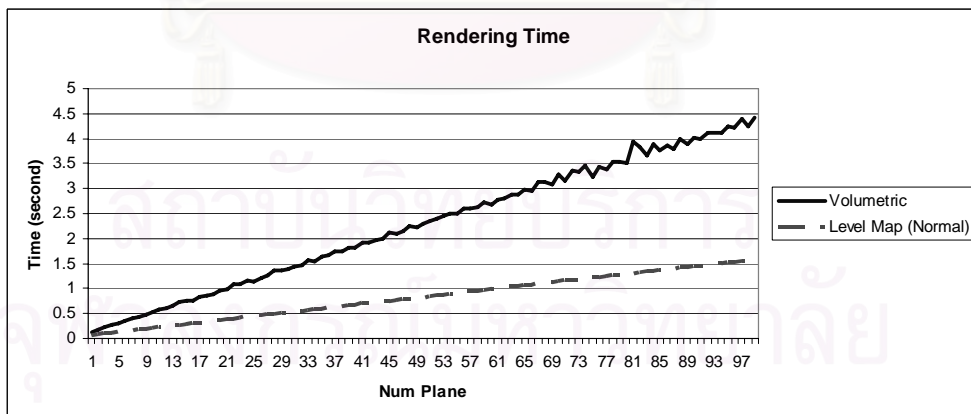
From graphs and table, the result from experimental program supports our assumption; the level map algorithm outperforms volumetric algorithm in all cases. This lead to the conclusion that, our level map algorithm can handle underwater scene with caustics faster than volumetric based algorithm.



(a)



(b)



(c)

Figure 5-10: Line graph shows comparison of rendering time between level map and volumetric algorithm. Figure (a), (b) and (c) show the result when teapot, dolphin and sphere is used as an input respectively.

Experiment 2

In this sub-section, we perform an experiment to compare the rendering time when smart slicing is turned on and off. Table 5-3 shows some example results from this experiment. The Normal columns in this table refer to rendering time when smart slicing is not used. And Smart columns mean the smart slicing is turned on. Figure 5-11 shows comparison graphs.

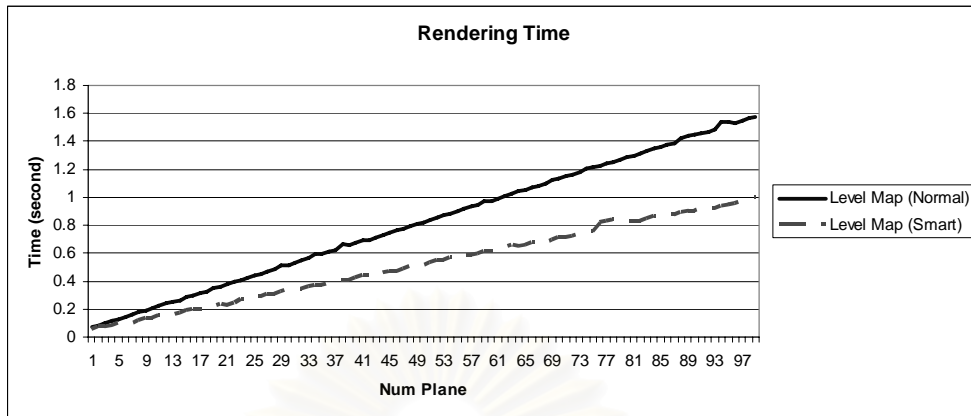
Table 5-3: Results from experiment 2.

Num Plane	Rendering Time					
	Teapot		Dolphin		Sphere	
	Normal	Smart	Normal	Smart	Normal	Smart
3	0.068	0.053	0.063	0.049	0.063	0.032
4	0.083	0.069	0.08	0.066	0.079	0.05
5	0.099	0.069	0.095	0.081	0.093	0.048
6	0.115	0.084	0.111	0.096	0.109	0.064
7	0.12899999	0.101	0.127	0.113	0.123	0.066
8	0.14399999	0.104	0.142	0.12800001	0.14	0.082
9	0.16599999	0.102	0.15899999	0.14300001	0.156	0.08
10	0.17900001	0.117	0.175	0.15700001	0.17	0.098
11	0.191	0.133	0.19	0.176	0.18700001	0.111
12	0.205	0.133	0.229	0.189	0.20299999	0.114
13	0.22499999	0.15099999	0.222	0.205	0.21699999	0.114
14	0.245	0.155	0.244	0.22499999	0.236	0.131
15	0.25400001	0.16500001	0.25	0.24699999	0.25	0.131
16	0.26499999	0.168	0.26699999	0.259	0.264	0.14399999
17	0.285	0.185	0.28	0.26800001	0.28099999	0.14399999
18	0.29499999	0.197	0.29699999	0.287	0.29699999	0.164
19	0.315	0.198	0.31299999	0.30000001	0.31200001	0.18799999
20	0.32699999	0.20299999	0.33500001	0.31400001	0.324	0.177
21	0.34799999	0.213	0.36300001	0.33199999	0.34200001	0.19499999
22	0.359	0.23100001	0.366	0.34599999	0.35499999	0.19499999
23	0.382	0.229	0.38499999	0.361	0.37200001	0.197
24	0.39199999	0.244	0.39199999	0.37900001	0.389	0.213
25	0.40900001	0.26800001	0.40599999	0.39399999	0.40099999	0.212
26	0.42199999	0.27599999	0.42500001	0.412	0.41800001	0.226
27	0.43799999	0.28	0.442	0.426	0.43700001	0.23100001
28	0.44999999	0.29100001	0.45500001	0.44100001	0.447	0.24699999
29	0.47	0.30899999	0.47	0.45699999	0.47499999	0.248
30	0.48300001	0.30700001	0.484	0.47400001	0.479	0.25999999
31	0.51200002	0.32600001	0.51300001	0.49700001	0.495	0.26699999
32	0.51300001	0.34	0.51999998	0.509	0.50999999	0.27900001
33	0.52999997	0.34099999	0.52899998	0.52999997	0.528	0.27399999

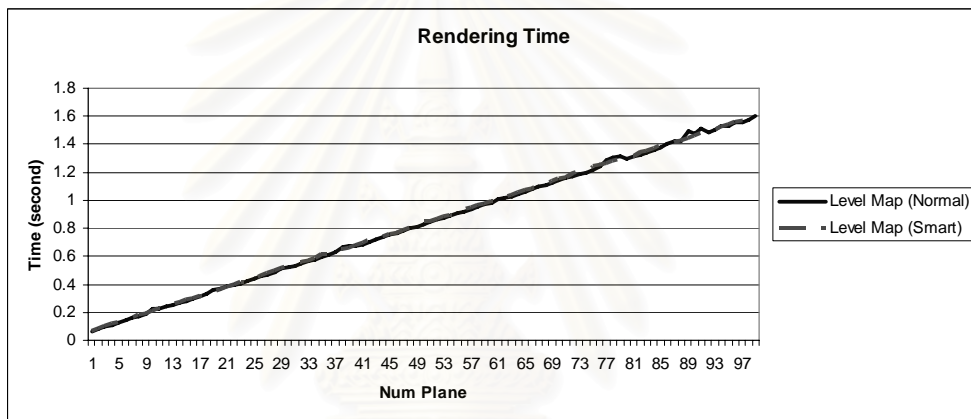
Num Plane	Rendering Time					
	Teapot		Dolphin		Sphere	
	Normal	Smart	Normal	Smart	Normal	Smart
34	0.54500002	0.34	0.54900002	0.53600001	0.53899997	0.30399999
35	0.56800002	0.359	0.56300002	0.55500001	0.55900002	0.29300001
36	0.59799999	0.373	0.57800001	0.57099998	0.57200003	0.30599999
37	0.59200001	0.373	0.59399998	0.583	0.588	0.32100001
38	0.60799998	0.389	0.61000001	0.61500001	0.60100001	0.32699999
39	0.62199998	0.38999999	0.62599999	0.61199999	0.63499999	0.32600001
40	0.667	0.40400001	0.66500002	0.63499999	0.63700002	0.34999999
41	0.653	0.40599999	0.671	0.64499998	0.64999998	0.37
42	0.67699999	0.42199999	0.67400002	0.65899998	0.667	0.35299999
43	0.69599998	0.43700001	0.68800002	0.67699999	0.704	0.35699999
44	0.69700003	0.44	0.704	0.69300002	0.69800001	0.373
45	0.71499997	0.44299999	0.71899998	0.70700002	0.713	0.37200001
46	0.72899997	0.456	0.73500001	0.72299999	0.72600001	0.38800001
47	0.74400002	0.46799999	0.75300002	0.73900002	0.741	0.40599999
48	0.76200002	0.47	0.76499999	0.75400001	0.75800002	0.405
49	0.77499998	0.486	0.77999997	0.76800001	0.77100003	0.41600001
50	0.79299998	0.5	0.79900002	0.78500003	0.787	0.42300001
51	0.80699998	0.51899999	0.81099999	0.80199999	0.80299997	0.417
52	0.82099998	0.51599997	0.82800001	0.81900001	0.81900001	0.43900001
53	0.838	0.53299999	0.84299999	0.84399998	0.833	0.44100001
54	0.852	0.546	0.86400002	0.84899998	0.85000002	0.44999999
55	0.86900002	0.55199999	0.87400001	0.86299998	0.86400002	0.46799999
56	0.88499999	0.56300002	0.88800001	0.87900001	0.88200003	0.46799999
57	0.89600003	0.57999998	0.90600002	0.89399999	0.91000003	0.46799999
58	0.91399997	0.58099997	0.92199999	0.91299999	0.91900003	0.48500001
59	0.93300003	0.58099997	0.93900001	0.92799997	0.92699999	0.48699999
60	0.94400001	0.59600002	0.95200002	0.94499999	0.94400001	0.51099998
61	0.96799999	0.611	0.96899998	0.95999998	0.958	0.50599998
62	0.97299999	0.61299998	0.98000002	0.97500002	0.97299999	0.51599997
63	0.991	0.63099998	1.00699997	0.99400002	0.99199998	0.51599997
64	1.00699997	0.62900001	1.01699996	1.00300002	0.99900001	0.53200001
65	1.02499998	0.653	1.02999997	1.01999998	1.02400005	0.53200001
66	1.04299998	0.64499998	1.04499996	1.03699994	1.03600001	0.55000001
67	1.05299997	0.65899998	1.06299996	1.05200005	1.05599999	0.56400001
68	1.06700003	0.67900002	1.08200002	1.06900001	1.06299996	0.56800002
69	1.08000004	0.68199998	1.09899998	1.08299994	1.08000004	0.56599998
70	1.09599996	0.676	1.10599995	1.10000002	1.10000002	0.583
71	1.12699997	0.69400001	1.125	1.11399996	1.11099994	0.57999998
72	1.13100004	0.70899999	1.14199996	1.13199997	1.12699997	0.59799999
73	1.148	0.70999998	1.16499996	1.14900005	1.15499997	0.611
74	1.15999997	0.72299999	1.16999996	1.16199994	1.15699995	0.61299998
75	1.17799997	0.74199998	1.18700004	1.19000006	1.17200005	0.61199999

Num Plane	Rendering Time					
	Teapot		Dolphin		Sphere	
	Normal	Smart	Normal	Smart	Normal	Smart
76	1.20200002	0.75599998	1.20099998	1.20099998	1.18799996	0.62800002
77	1.21399999	0.75599998	1.21899998	1.21500003	1.20299995	0.63200003
78	1.22300005	0.81900001	1.24300003	1.24600005	1.21800005	0.64600003
79	1.24000001	0.829	1.28900003	1.25300002	1.23500001	0.66299999
80	1.25100005	0.83999997	1.30599999	1.25800002	1.25300002	0.66500002
81	1.27199996	0.815	1.31700003	1.27600002	1.26400006	0.67900002
82	1.28699994	0.824	1.29700005	1.28900003	1.27900004	0.67900002
83	1.29900002	0.829	1.30999994	1.29999995	1.29700005	0.68400002
84	1.31500006	0.824	1.32500005	1.31700003	1.31099999	0.70899999
85	1.33099997	0.84299999	1.34500003	1.33700001	1.32599998	0.699
86	1.34800005	0.86500001	1.36199999	1.35000002	1.34300005	0.71799999
87	1.35899997	0.86000001	1.37300003	1.36399996	1.35399997	0.72299999
88	1.37300003	0.87199998	1.403	1.38199997	1.37100005	0.72500002
89	1.38999999	0.87300003	1.41799998	1.398	1.38800001	0.72600001
90	1.41900003	0.89099997	1.421	1.41100001	1.40199995	0.741
91	1.44400001	0.90399998	1.49800003	1.426	1.41900003	0.74400002
92	1.44799995	0.90399998	1.47399998	1.44299996	1.43599999	0.75999999
93	1.45599997	0.92299998	1.51100004	1.46000004	1.449	0.77399999
94	1.47099996	0.92000002	1.48800004	1.47800004	1.46300006	0.77399999
95	1.48199999	0.917	1.50199997	1.49399996	1.48000002	0.792
96	1.53499997	0.93300003	1.52999997	1.50600004	1.49699998	0.792
97	1.54200006	0.94700003	1.528	1.52199996	1.51499999	0.79299998
98	1.52999997	0.95099998	1.55400002	1.53699994	1.52699995	0.81
99	1.54400003	0.96399999	1.56099999	1.55700004	1.54100001	0.80900002
100	1.56200004	0.98000002	1.57500005	1.57000005	1.55299997	0.82099998
101	1.574	1	1.60099995	1.57099998	1.57200003	0.82499999

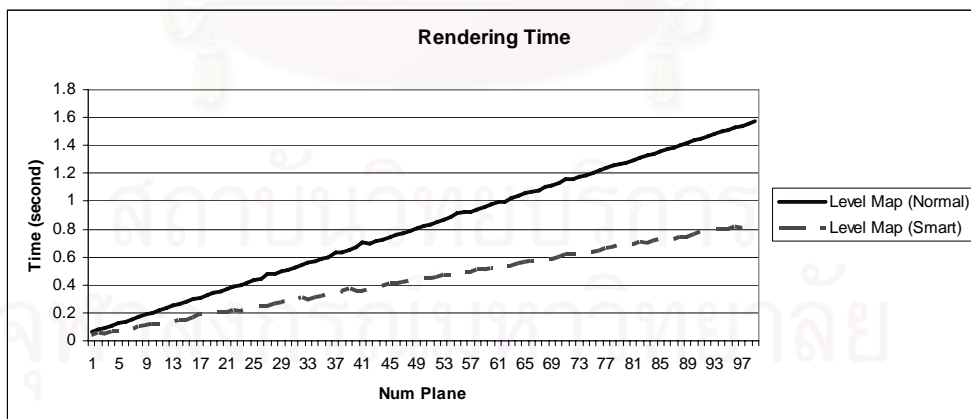
สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย



(a)



(b)



(c)

Figure 5-11: Line graph shows comparison of rendering time when smart slicing is turn on and off. Figure (a), (b) and (c) show the result when teapot, dolphin and sphere is used as an input respectively

When considering these results, it can be seen that the benefit from using smart slicing varies from one kind of model to another kind of model. When using sphere as an input, the algorithm gives the best performance, however, using with dolphin gives no difference. This comes from the fact that the smart slicing algorithm can perform at its best when there are “many” empty levels. When there are many empty levels, the algorithm can eliminate many unnecessary planes. Figure 5-12 show depth profiles from each sample input. There is only one empty level in dolphin model while the others have several. This is the reason why smart slicing cannot perform well with dolphin. Table 5-4 proves the earlier statement by showing the number of actual planes used when performing smart slicing.

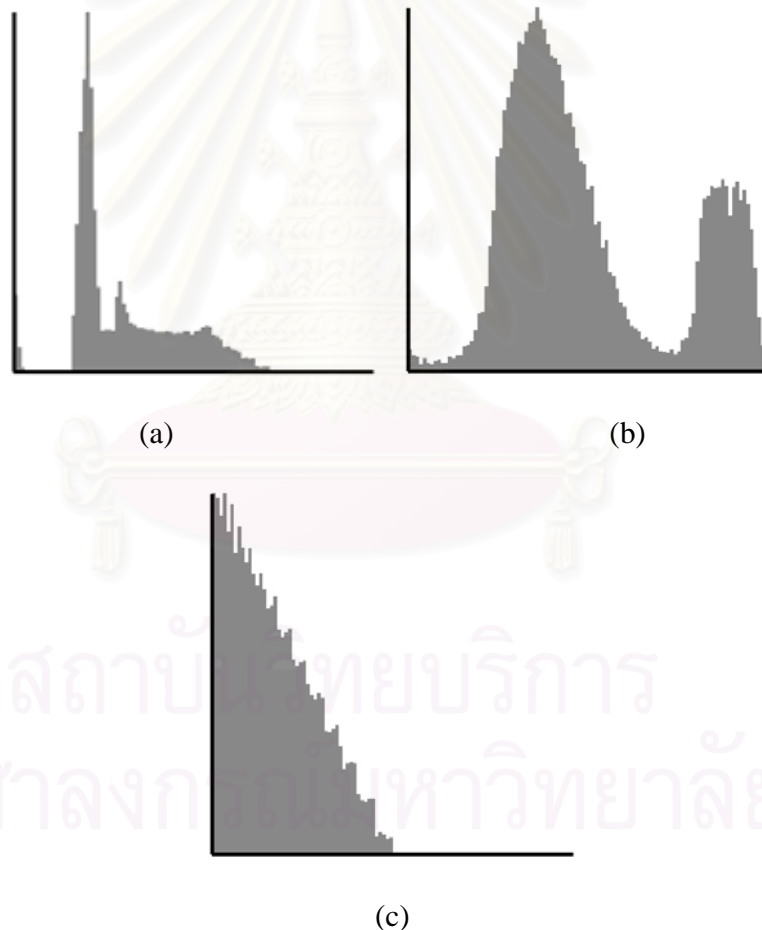


Figure 5-12: Depth profiles from sample models. Figure (a) shows depth profile of teapot. Figure (b) shows dolphin. And figure (c) shows sphere.

Table 5-4: The number of actual planes required when smart slicing is used.

Num Specified Plane	Num Actual Plane		
	Teapot	Dolphin	Sphere
3	2	2	1
4	3	3	2
5	3	4	2
6	4	5	3
7	5	6	3
8	5	7	4
9	5	8	4
10	6	9	5
11	7	10	6
12	7	11	6
13	8	12	6
14	8	13	7
15	9	14	7
16	9	15	8
17	10	16	8
18	11	17	9
19	11	18	10
20	11	19	10
21	12	20	11
22	13	21	11
23	13	22	11
24	14	23	12
25	15	24	12
26	16	25	13
27	16	26	13
28	17	27	14
29	18	28	14
30	18	29	15
31	19	30	15
32	20	31	16
33	20	32	16
34	20	33	17
35	21	34	17
36	22	35	18
37	22	36	19
38	23	37	19
39	23	38	19
40	24	39	20
41	24	40	21
42	25	41	21
43	26	42	21

Num Specified Plane	Num Actual Plane		
	Teapot	Dolphin	Sphere
44	26	43	22
45	26	44	22
46	27	45	23
47	28	46	24
48	28	47	24
49	29	48	24
50	30	49	25
51	31	50	25
52	31	51	26
53	32	52	26
54	33	53	27
55	33	54	28
56	34	55	28
57	35	56	28
58	35	57	29
59	35	58	29
60	36	59	30
61	37	60	30
62	37	61	31
63	38	62	31
64	38	63	32
65	39	64	32
66	39	65	33
67	40	66	34
68	41	67	34
69	41	68	34
70	41	69	35
71	42	70	35
72	43	71	36
73	43	72	37
74	44	73	37
75	45	74	37
76	46	75	38
77	46	76	38
78	47	77	39
79	48	78	40
80	48	79	40
81	49	80	41
82	50	81	41
83	50	82	41
84	50	83	42
85	51	84	42
86	52	85	43

Num Specified Plane	Num Actual Plane		
	Teapot	Dolphin	Sphere
87	52	86	44
88	53	87	44
89	53	88	44
90	54	89	45
91	54	90	45
92	55	91	46
93	56	92	47
94	56	93	47
95	56	94	48
96	57	95	48
97	58	96	48
98	58	97	49
99	59	98	49
100	60	99	50
101	61	99	50

There is a question we have not yet answered, that is, does smart slicing affect the quality of output image? Because the development of smart slicing is intended for eliminating unnecessary plane, theoretical answer for this question is “No”. This answer can be easily proved and presented in Table 5-5. This table shows the intensity difference of the result image when smart slicing is turned on and off. Equation (2-9) presented in Section 2-7 is used for computing intensity difference of resulting images. Table 5-5 shows that enabling smart slicing do not affect the quality in any cases.

Table 5-5: Intensity difference when smart slicing is turned on and off.

Num Specified Plane	Intensity Diff.		
	Teapot	Dolphin	Sphere
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	0	0	0
11	0	0	0
12	0	0	0
13	0	0	0
14	0	0	0

Num Specified Plane	Intensity Diff.		
	Teapot	Dolphin	Sphere
15	0	0	0
16	0	0	0
17	0	0	0
18	0	0	0
19	0	0	0
20	0	0	0
21	0	0	0
22	0	0	0
23	0	0	0
24	0	0	0
25	0	0	0
26	0	0	0
27	0	0	0
28	0	0	0
29	0	0	0
30	0	0	0
31	0	0	0
32	0	0	0
33	0	0	0
34	0	0	0
35	0	0	0
36	0	0	0
37	0	0	0
38	0	0	0
39	0	0	0
40	0	0	0
41	0	0	0
42	0	0	0
43	0	0	0
44	0	0	0
45	0	0	0
46	0	0	0
47	0	0	0
48	0	0	0
49	0	0	0
50	0	0	0

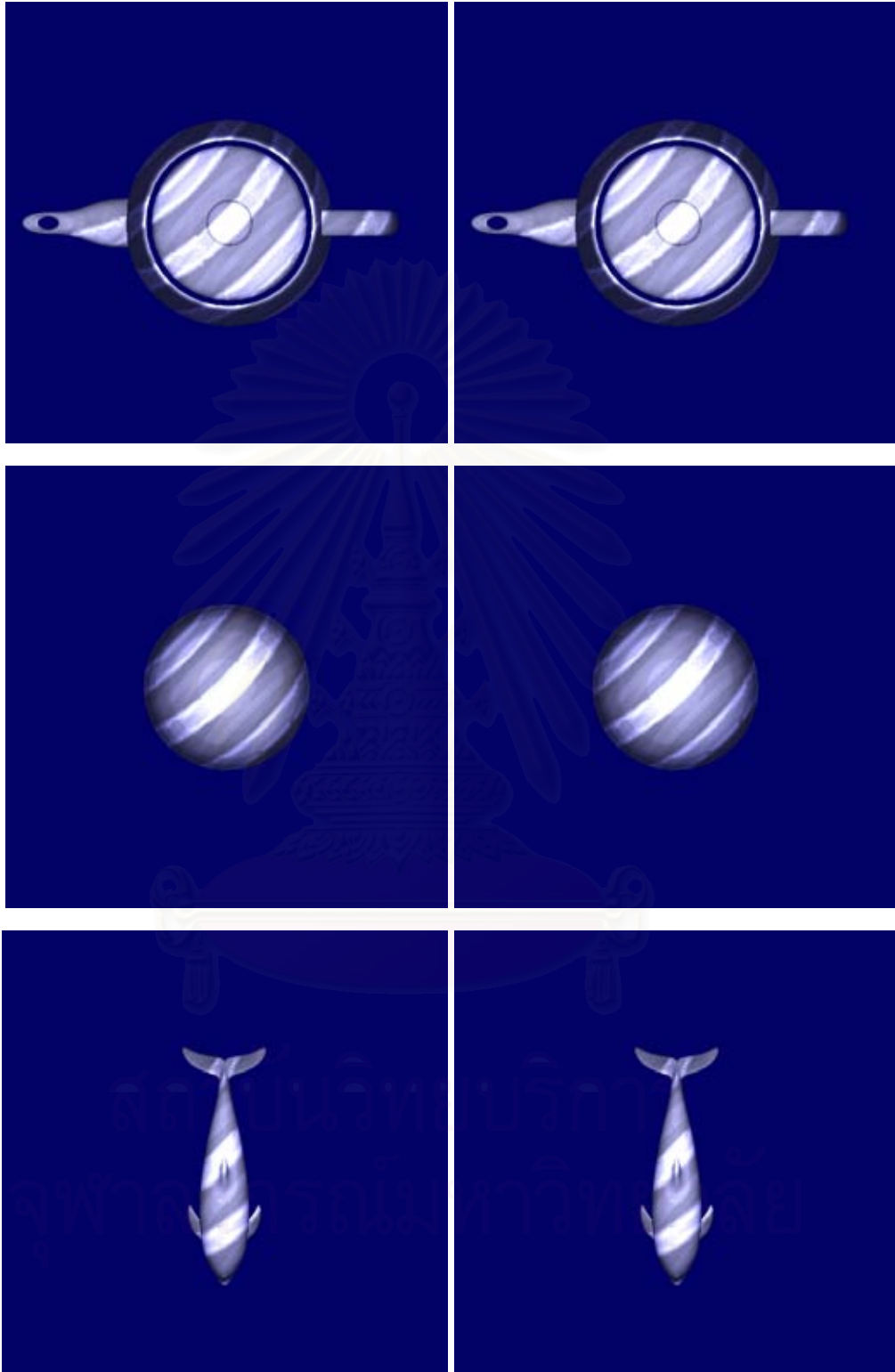


Figure 5-13: Comparing result when smart slicing is turned on and off. The left image is the result when smart slicing is turned off, while in the right image, smart slicing is turned on.

There is another interesting issue that should be discussed. As discussed in section 5.2, when using smart slicing there is an additional cost to pay, that is, the processing time to perform depth profiling. Depth profile is generated by looping through every pixel in the position map. Considering the case when the dimension of position map is 512x512, this means that there are extra 262144 iteration steps to perform each frame. Thus, it is interesting to understand how much this extra operation costs. When thoroughly examining Table 5-3 and Table 5-4, we can see that at the same number of reference planes, the rendering time used when smart slicing turn on is a little bit longer. But this additional processing time has a very little significance when compared to the processing time used for intersection test. Because the intersection test is performed on GPU, which has much less processing power than CPU. When comparing with benefit gained from unnecessary plane elimination, this price is worth to pay. From this reason, we can safely conclude that smart slicing algorithm can greatly reduce rendering time for level map algorithm in most situations. While there are some cases where smart slicing cannot perform well, the processing time is not much different from when smart slicing is disabled.

CHAPTER 6

DISCUSSION, CONCLUSIONS AND FURTHER IMPROVEMENTS

This chapter gives a detailed discussion about limitations of proposed algorithm. And then we give the conclusion about what we've done so far is also given. Finally, possible improvement methods are summarized.

6.1 Discussion

Although the level map algorithm can bring caustics to interactive time-frame, there are so many areas to be improved. The first one is high amount of polygon fill-rate. In the intersection test step, we render intersection triangle on each plane and then rasterize them and check which pixel should be accepted. Consider the case where water mesh consists of 10,000 polygons, which is the common polygon count in real-time application, and the caustics receiver object has 100 reference planes, this mean that there are about 1,000,000 polygons to be filled per frame. This extremely high polygon fill-rate will use long computational time and, in many case, this cannot be done in interactive time frame. Consider the same case with 100 objects on the scene, the situation is worse. In this case, there are 100,000,000 polygons to be filled, even though it is processed on a very high-end PC, it is nearly impossible to handle this scene in real-time.

To address this problem, the number of polygon fill-rate must be reduced. To achieve this goal, the bounding volume collision detection scheme can be used. This algorithm can be used to discard unnecessary rendering of intersection triangles that are fully outside the caustics receiver bounding box. Figure 6-1 presents this idea. As you can see from this figure, it is wasteful rendering the intersection triangle of left beam, because it misses the object. By performing early intersection test, we can reduce numerous numbers of polygons to be filled.

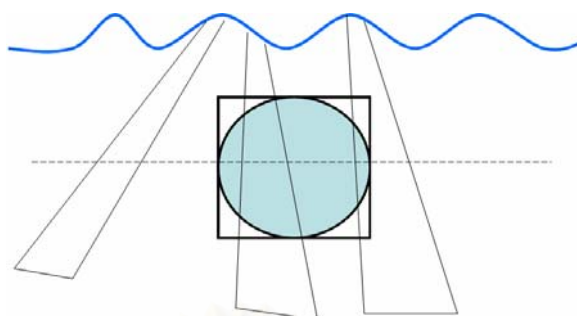


Figure 6-1: Bounding volume collision testing. The dash line represents a reference plane.

The second approach, which can be applied for reducing the number of polygon fill-rate, is based on Level of Details (LOD) method. In computer graphics, LOD is used for reducing the number of polygons in any model when the high detail version of that model is unnecessary. For example, when objects are in the close distant relative to the viewer, we render them with high detail version. However, when they are far away, we can not see much detail from them because the detail is relatively small to the viewer eyes, we use the low detail version to render them. Figure 6-2 and Figure 6-3 demonstrate the usage of LOD. Notice that when object is further away, the need for high detail representation is reduced. We can apply the concept of level of details to our level map algorithm by reducing the number of reference plane for testing intersection when objects are far. This approach can help organizing complex scene easier.



Figure 6-2: Multi-resolution air plane. Image by Hugues Hoppe.



Figure 6-3: LOD in action.

6.2 Conclusion

This thesis has presented the algorithm for presenting caustics in real-time applications. Level map algorithm presents new approach for testing intersection between light beam and object. This new algorithm is the heart of our algorithm which can increase rendering speed drastically. The level map algorithm not only handle underwater caustics in interactive time, but also reduce memory usage which is the main limitation of previously proposed volumetric based algorithm significantly. The memory usage, when a color texture used for representing a sample plane, in the volumetric based algorithm are incomparable to the level map algorithm, of which only two texture are used for each object. Moreover, the smart slicing technique presented here can eliminate unnecessary intersection test, which can further improve the rendering speed of level map algorithm. Smart slicing technique is not limited to level map algorithm, any height field or depth map based intersection testing algorithm can benefit from smart slicing. In term of quality, our experiments have already proven that the resulting images from level map are per-pixel identical to volumetric based algorithm.

For the down side of level map, this algorithm requires high polygon fill-rate and programmability feature, which limits its usage to this generation medium or high-end hardware. Various enhancing techniques are proposed, but they need some time to prove themselves.

6.3 Further Improvement

The main improvement we need to do next is reducing number of polygon filled. Algorithms presented in section 6.1 must be implemented in the next version of program. Exploration of algorithm that places intersection triangle on actually hit is worth to try.

REFERENCES

- [1] Guenther, J., Wald, I., & Slusallek, P. 2004. Realtime Caustics Using Distributed Photon Mapping. Rendering Techniques 2004 : Eurographics Symposium on Rendering: 111-121.
- [2] Purcell, T. J., Donner, C., Cammarano, M., Jensen, H.W., & Hanrahan, P. 2003. Photon Mapping on Programmable Graphics Hardware. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (2003): 41-50.
- [3] Trendall, C., & Stewart, A.J. 2000. General Calculation Using Graphics Hardware, with Application to Interactive Caustics. Proceedings of the Eurographics Workshop on Rendering Techniques 2000 (2000): 287-298.
- [4] Wald, I., Kollig, T., Benthin, C., Keller, A., & Slusallek P. 2002. Interactive Global Illumination Using Fast Ray Tracing. Proceedings of the 13th Eurographics workshop on Rendering (2002): 15-24.
- [5] Wyman, C., Hansen, C. D., & Shirley, P. 2004. Interactive Caustics Using Local Precomputed Irradiance. Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04) 00 (2004): 143-151.
- [6] Crespo, D. S., & Guaydado, J. 2004. Rendering Water Caustics. Fernando, R., GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, 31-44. Addison Wesley.
- [7] Stam, J. 1996. Random Caustics: Natural, Textures and Wave Theory Revisited. ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96 (1996): 150.
- [8] Shah, M. A., & Pattanaik, S. 2005. Caustics Mapping: An Image-Space Technique for Real-Time Caustics. IEEE Transactions on Visualization and Computer Graphics: (To be appeared).
- [9] Wand, M., & Straßer, W. 2003. Real-Time Caustics. Computer Graphics Forum 22 (2003): 610-619.

- [10] Iwasaki, K., Dobashi, Y., & Nishita, T. 2003. A Fast Rendering Method for Refractive and Reflective Caustics Due To Water Surfaces. Computer Graphics Forum 22 (September 2003): 601-609.
- [11] Iwasaki, K., Yoshimoto, F., Dobashi, Y., & Nishita, T. 2005. A Method for Fast Rendering of Caustics from Refraction by Transparent Objects. IEICE Transactions on Information and Systems 2005 E88-D 5 (2005): 904-911.
- [12] Arvo, J. 1986. Backwards Ray Tracing. SIGGRAPH' 86 Course Note 12(August 1986): 259-263.
- [13] Heckbert, P. S. 1990. Adaptive Radiosity Textures for Bidirectional Ray Tracing. Proceedings of the 17th annual conference on Computer graphics and interactive techniques 17 (1990): 145-154.
- [14] Mitchell, D., & Hanrahan, P. 1992. Illumination from Curved Reflectors. Proceedings of the 19th annual conference on Computer graphics and interactive techniques (1992): 283-291.
- [15] Jensen, H. W. 1996. Global Illumination Using Photon Maps. Proceedings of the Eurographics Workshop on Rendering Techniques '96 (1996): 21-30.
- [16] Jensen, H. W. 1996. Rendering Caustics on Non-Lambertian Surfaces. Proceedings of the conference on Graphics interface '96 (1996): 116-121.
- [17] Watt, M. 1990. Light-Water Interaction Using Backward Beam Tracing. Proceedings of the 17th annual conference on Computer graphics and interactive techniques (1990): 377-385.
- [18] Heckbert, P. S., & Hanrahan, P. 1984. Beam Tracing Polygonal objects. Proceedings of the 11th annual conference on Computer graphics and interactive techniques 11 (1984): 119-127.
- [19] Nishita, T., & Nakamae, E. 1994. Method of Displaying Optical Effects within Water Using Accumulation-Buffer. Proceedings of the 21st annual conference on Computer graphics and interactive techniques (1994): 373-379.
- [20] Iwasaki, K., Dobashi, Y., & Nishita, T. 2002. An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware. Computer Graphics Forum 21 (November 2002): 701-711.



APPENDIX

สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

This chapter shows results of level map algorithm from experimental program. The number of test case is 30. Each input model is selected randomly to make them cover various kinds of surface.

Results are shown as three sample images. The first image presents the result from volumetric texture, the second is from level map when smart slicing turn-off and the third is from level map when smart slicing turn-on. Rendering speed comparison graph and depth histogram also provided.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย

1. Teapot

Polygon Count: 6,320



Figure A - 1: Sample results from teapot model.

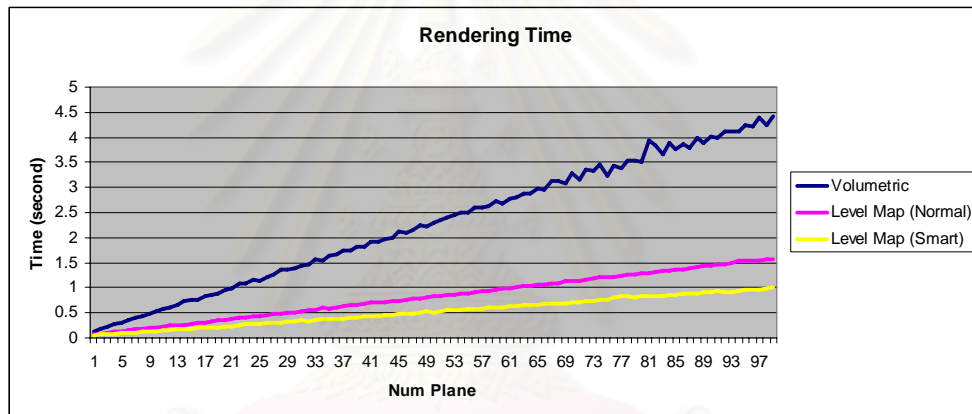


Figure A - 2: Rendering speed comparison graph while using teapot as an input.

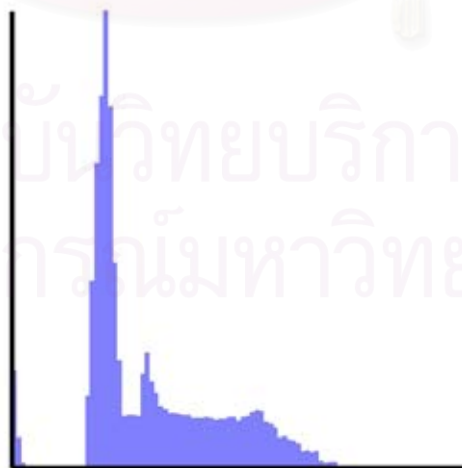


Figure A - 3: Depth histogram of teapot model.

2. Dolphin

Polygon Count: 4,442



Figure A - 4: Sample results from dolphin model.

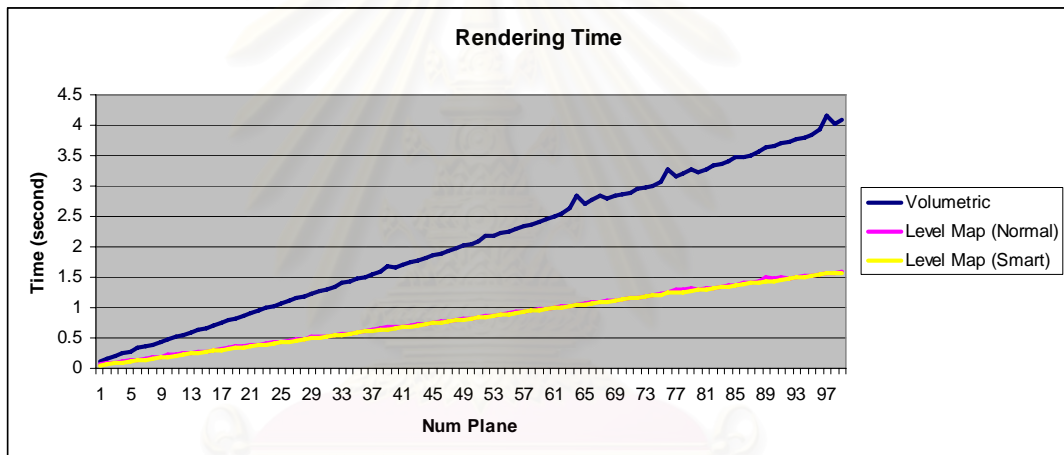


Figure A - 5: Rendering speed comparison graph while using dolphin as an input.

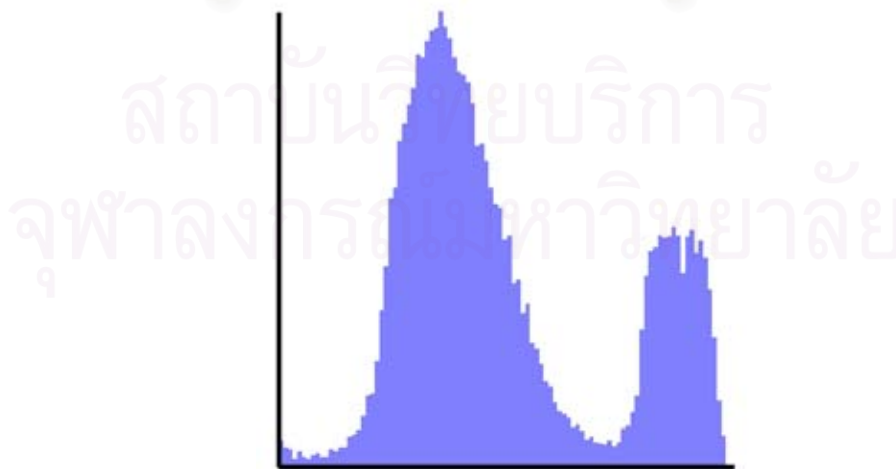


Figure A - 6: Depth histogram of dolphin model.

3. Sphere

Polygon Count: 1,984



Figure A - 7: Sample results from sphere model.

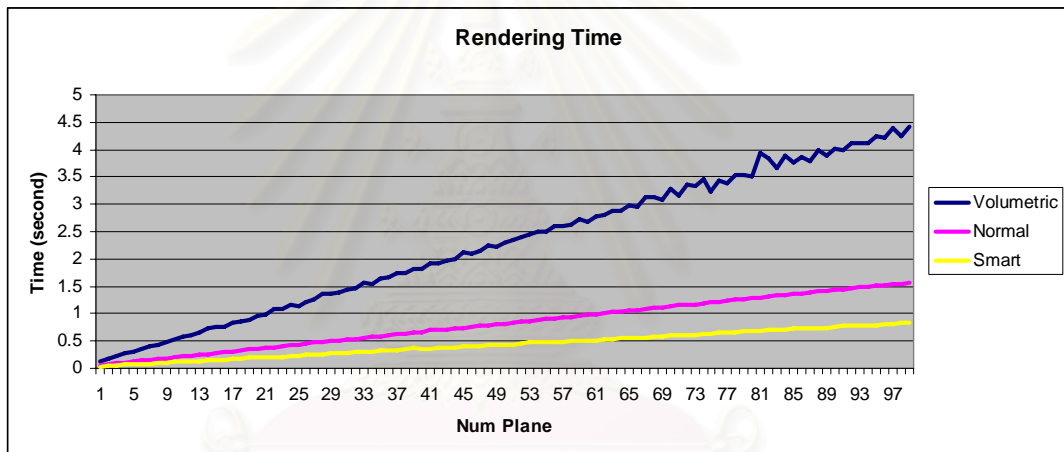


Figure A - 8: Rendering speed comparison graph while using sphere as an input.

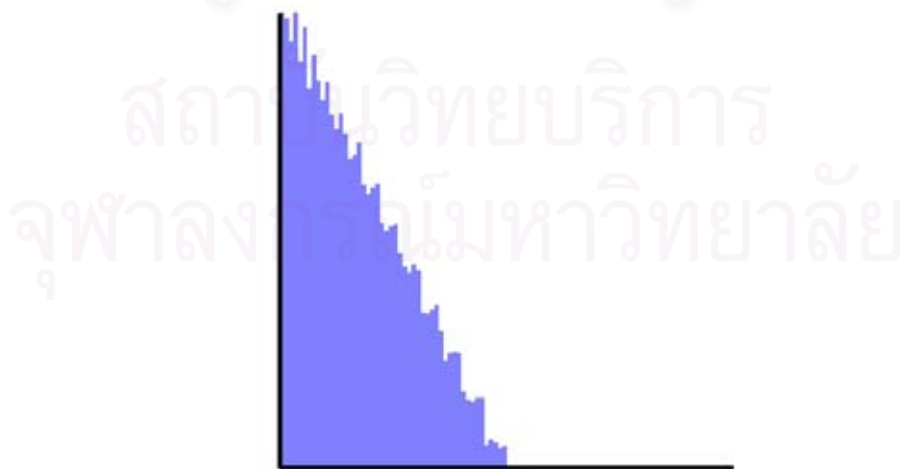


Figure A - 9: Depth histogram of sphere model.

4. Angle Fish

Polygon Count: 6,996

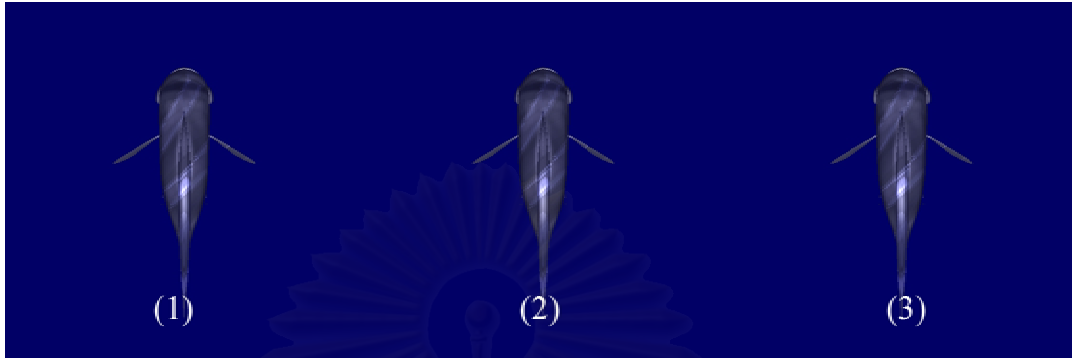


Figure A - 10: Sample results from angle fish model.

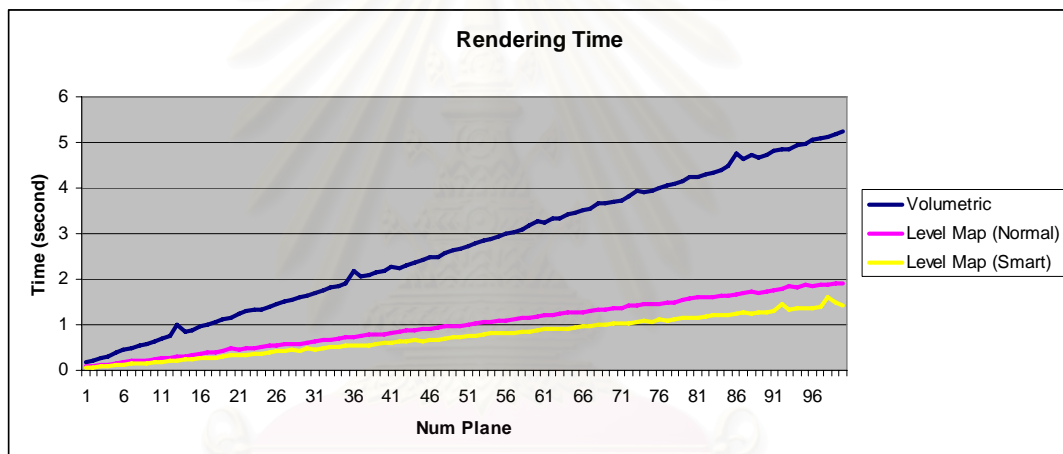


Figure A - 11: Rendering speed comparison graph while using angle fish as an input.

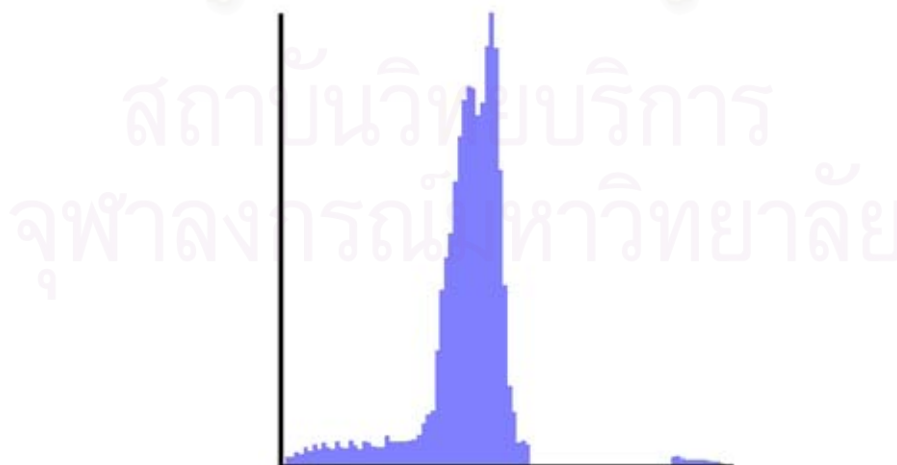


Figure A - 12: Depth histogram of angle fish model.

5. Box

Polygon Count: 12



Figure A - 13: Sample results from box model.

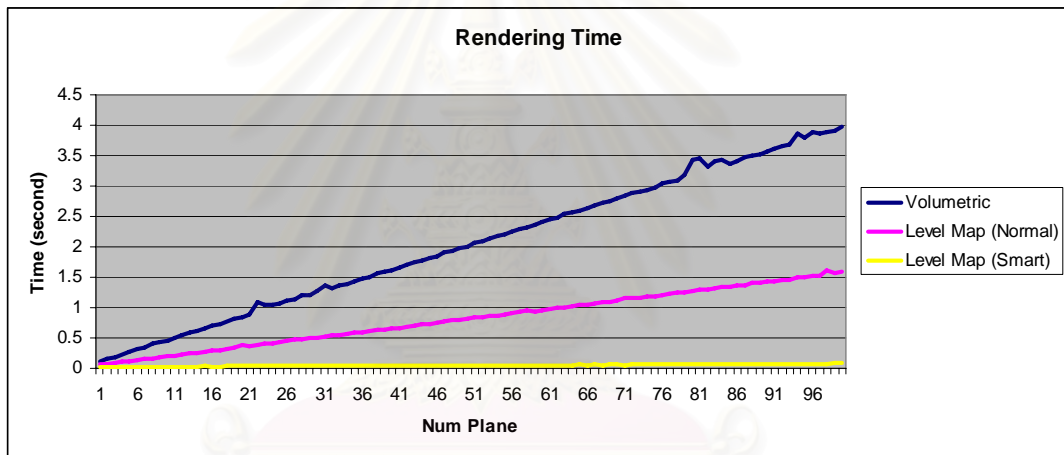


Figure A - 14: Rendering speed comparison graph while using box as an input.



Figure A - 15: Depth histogram of box model.

6. Horse

Polygon Count: 4314

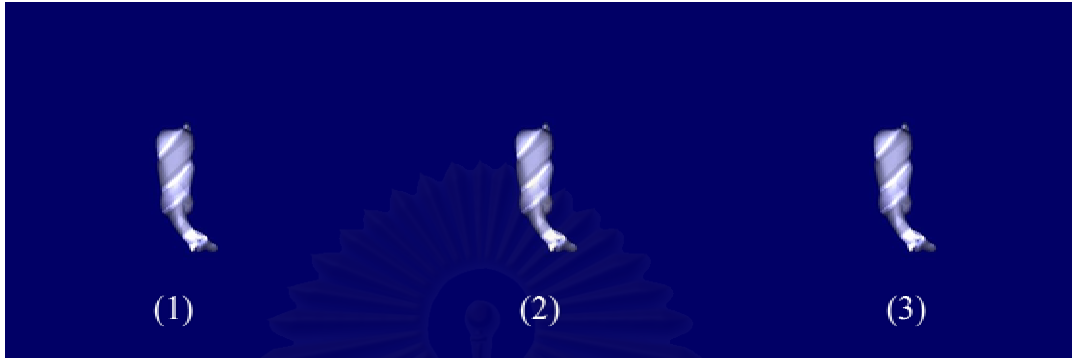


Figure A - 16: Sample results from horse model.

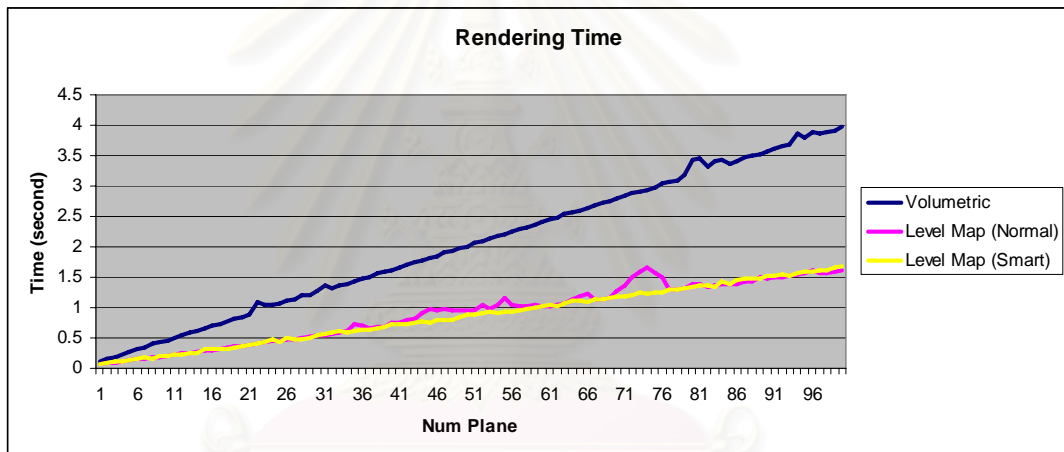


Figure A - 17: Rendering speed comparison graph while using horse as an input.

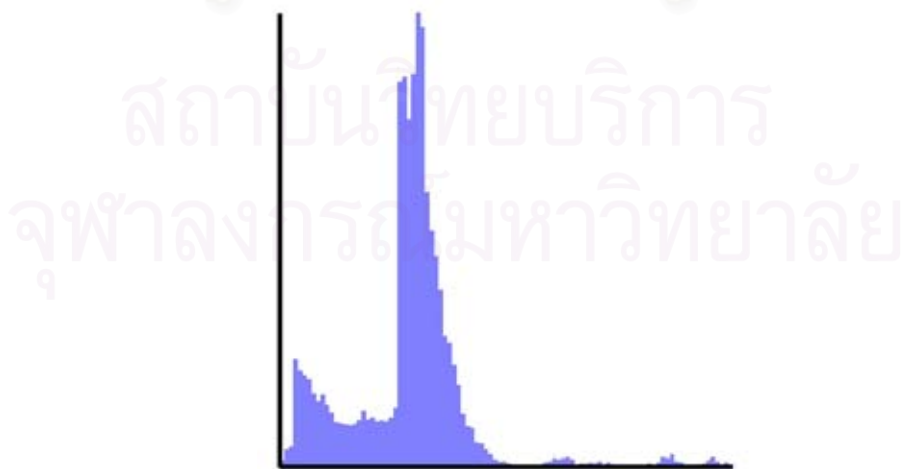


Figure A - 18: Depth histogram of horse model.

7. Manta

Polygon Count: 6820



Figure A - 19: Sample results from manta model.

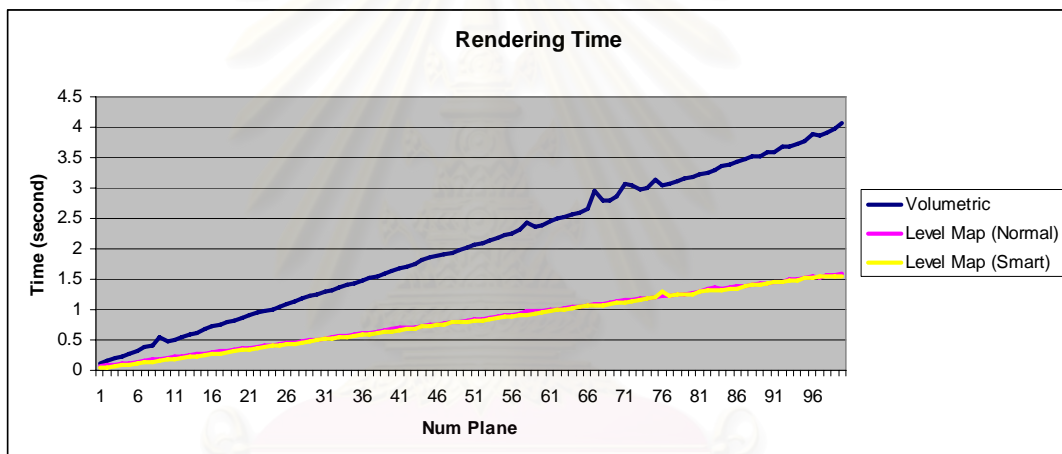


Figure A - 20: Rendering speed comparison graph while using manta as an input.

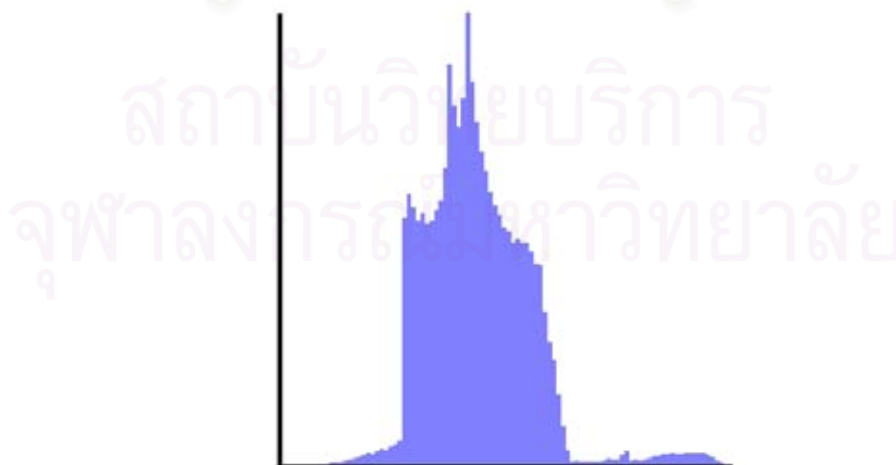


Figure A - 21: Depth histogram of manta model.

8. Red Betta

Polygon Count: 6760

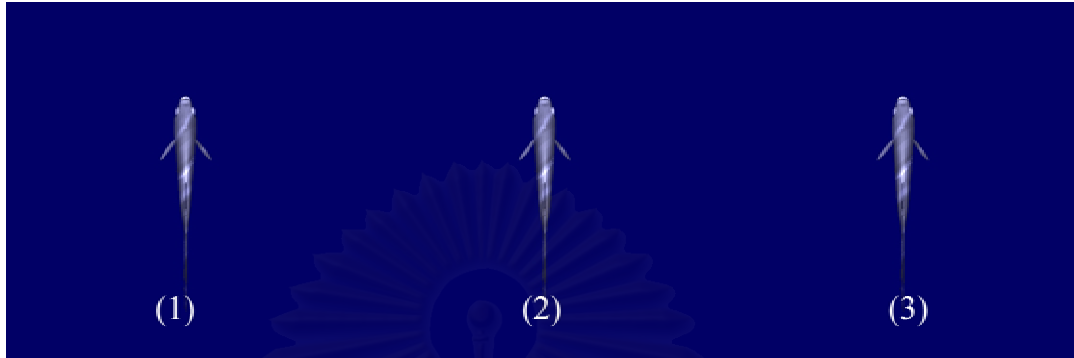


Figure A - 22: Sample results from red betta model.

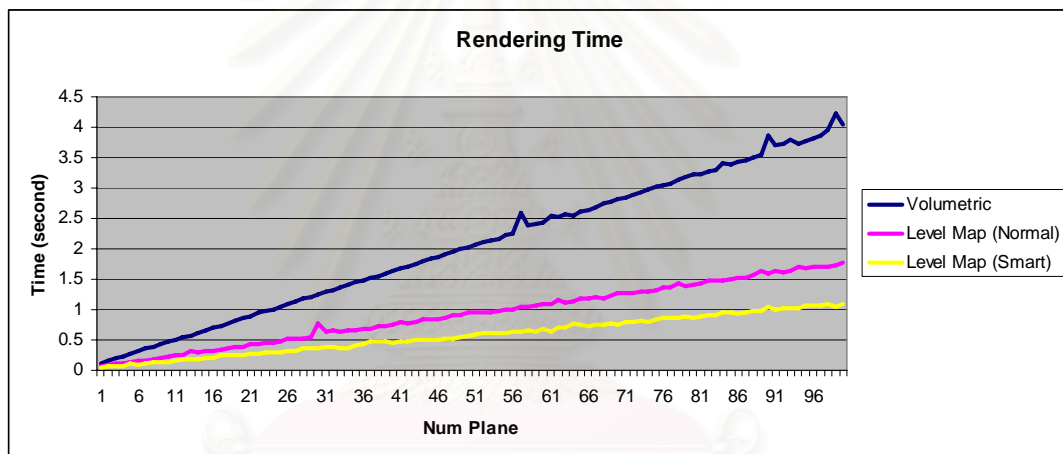


Figure A - 23: Rendering speed comparison graph while using red betta as an input.

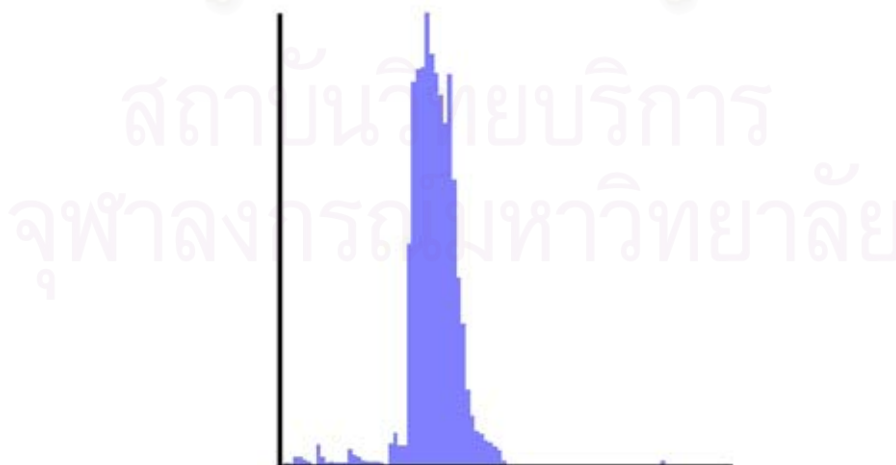


Figure A - 24: Depth histogram of red betta model.

9. Shark

Polygon Count: 942

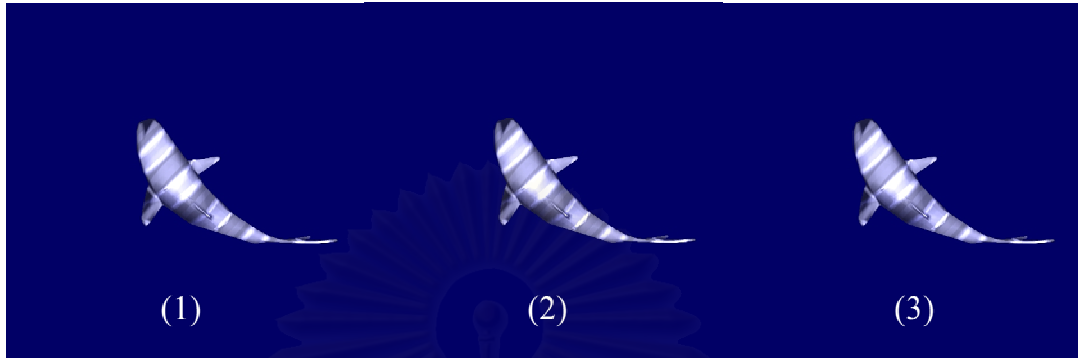


Figure A - 25: Sample results from shark model.

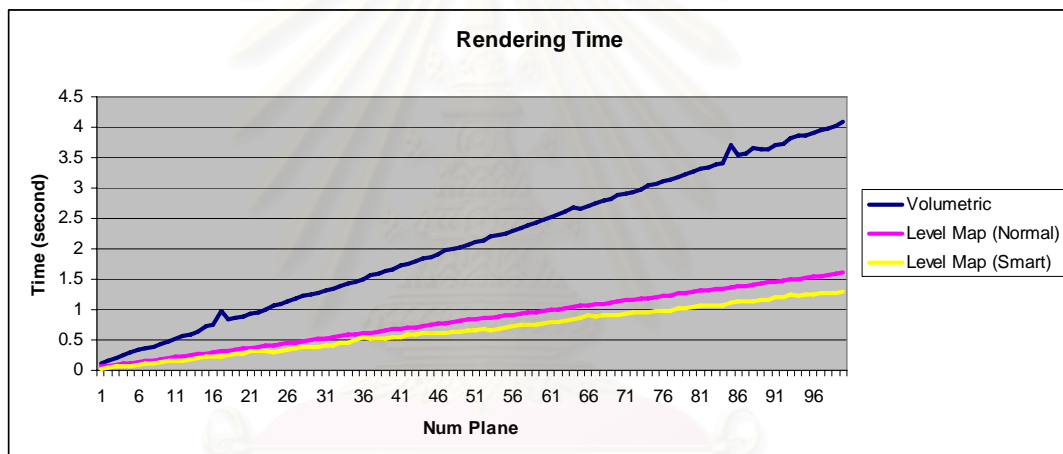


Figure A - 26: Rendering speed comparison graph while using shark as an input.

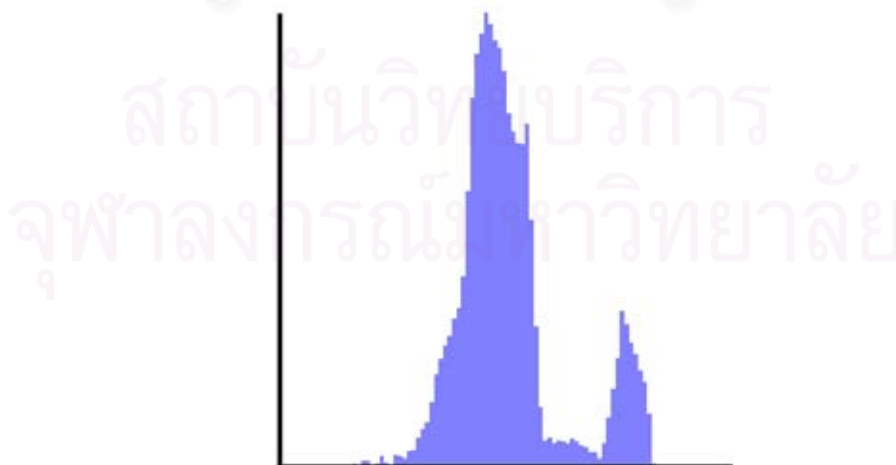


Figure A - 27: Depth histogram of shark model.

10. Siamese Tiger

Polygon Count: 6822

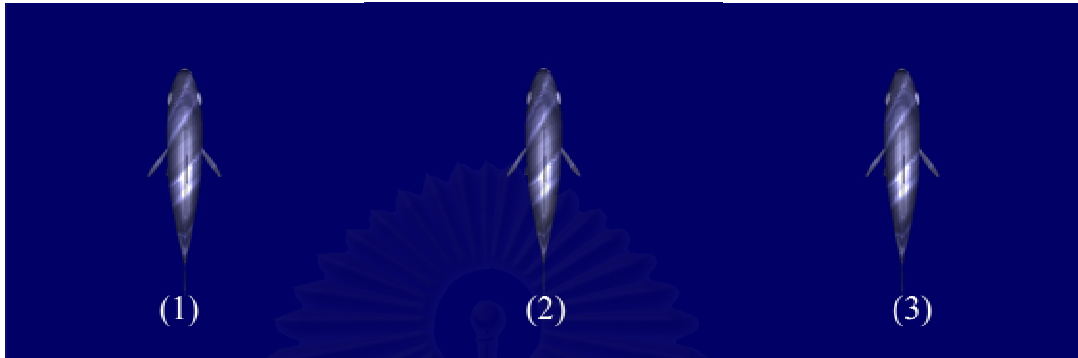


Figure A - 28: Sample results from Siamese tiger model.

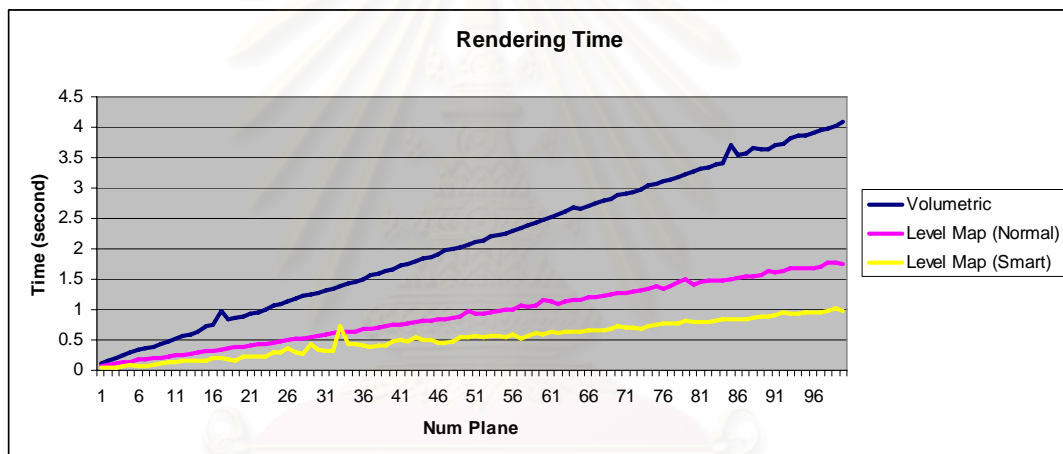


Figure A - 29: Rendering speed comparison graph while using Siamese tiger as an input.

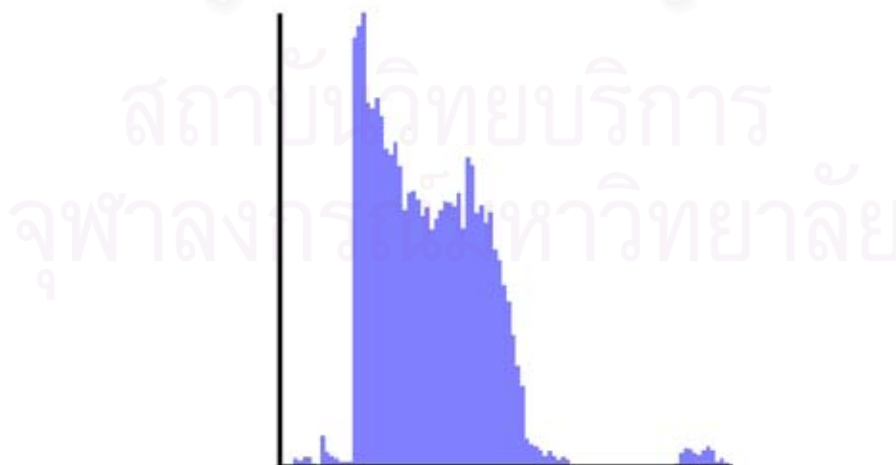


Figure A - 30: Depth histogram of Siamese tiger model.

11. Sink

Polygon Count: 1068



Figure A - 31: Sample results from sink model.

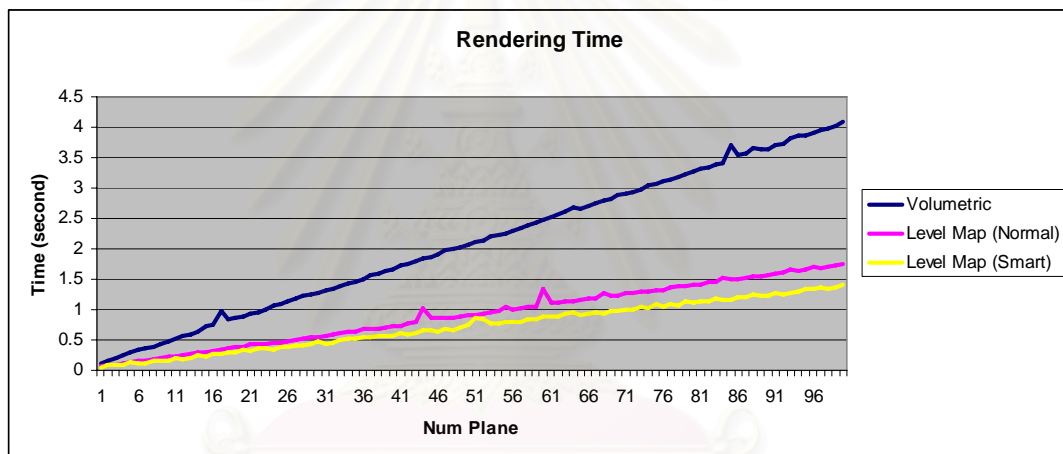


Figure A - 32: Rendering speed comparison graph while using sink as an input.

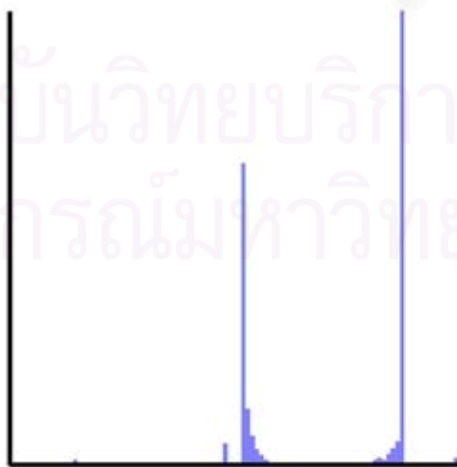


Figure A - 33: Depth histogram of sink model.

12. Car

Polygon Count: 9907



Figure A - 34: Sample results from car model.

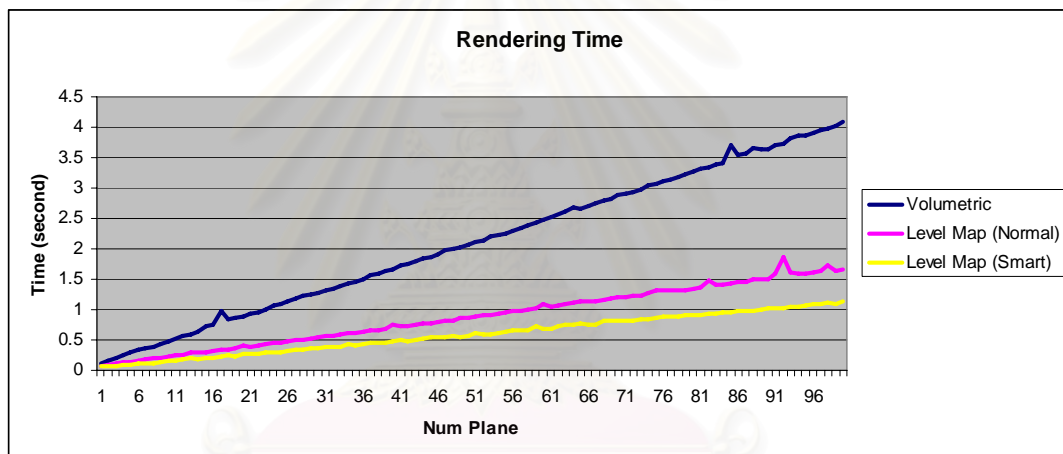


Figure A - 35: Rendering speed comparison graph while using car as an input.



Figure A - 36: Depth histogram of car model.

13. Sofa

Polygon Count: 10360



Figure A - 37: Sample results from sofa model.

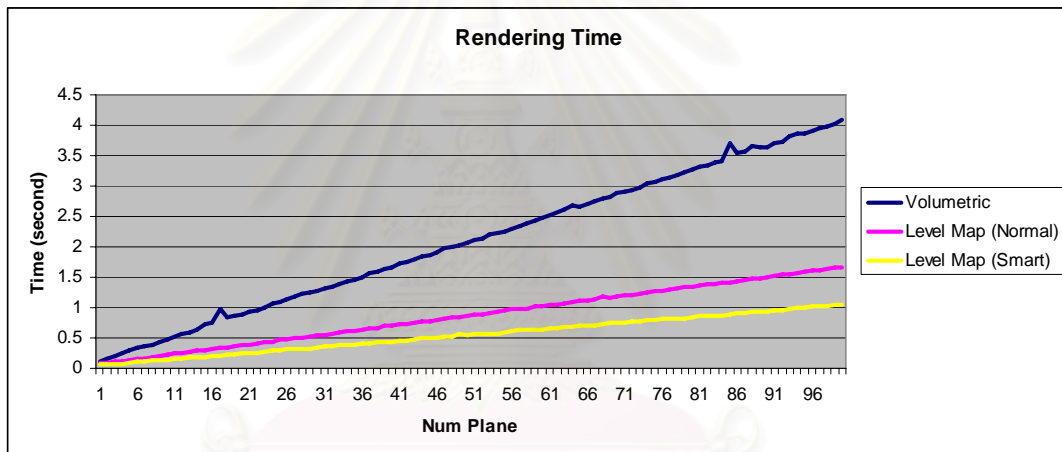


Figure A - 38: Rendering speed comparison graph while using sofa as an input.

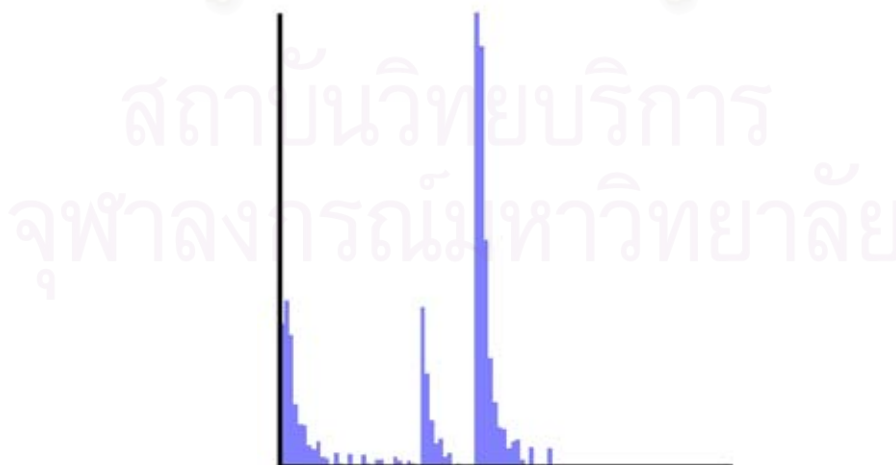


Figure A - 39: Depth histogram of sofa model.

14. Helicopter

Polygon Count: 5138

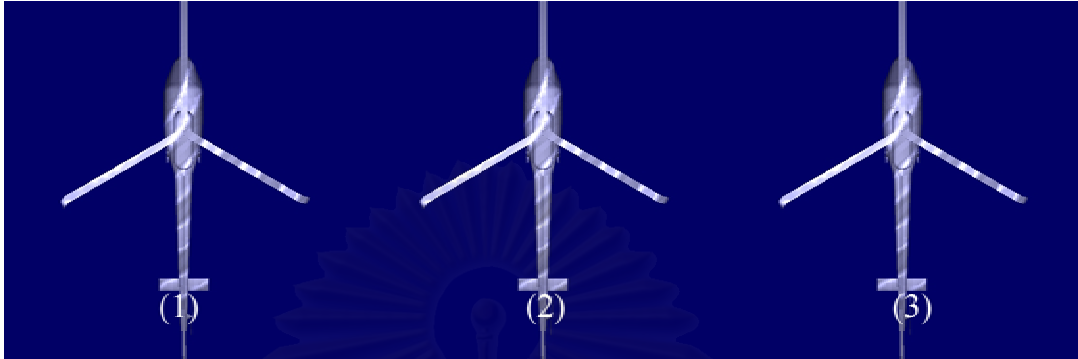


Figure A - 40: Sample results from helicopter model.

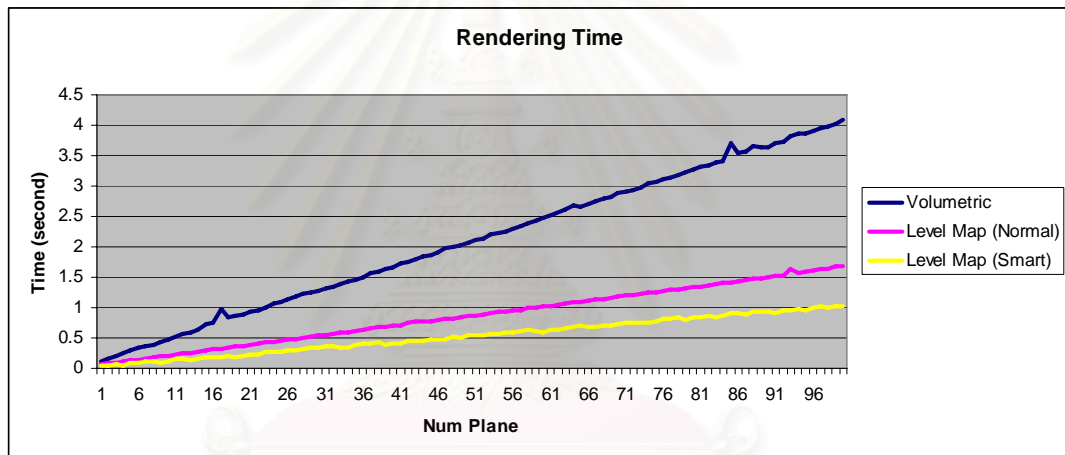


Figure A - 41: Rendering speed comparison graph while using helicopter as an input.

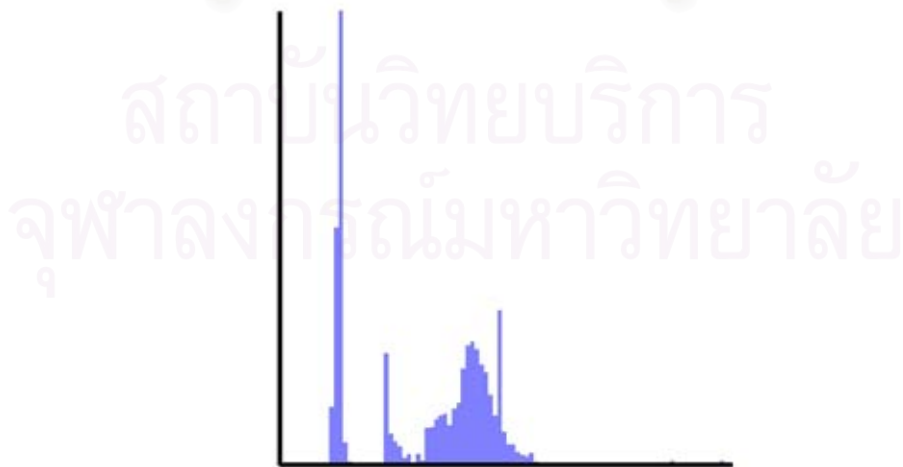


Figure A - 42: Depth histogram of sofa model.

15. UFO

Polygon Count: 20663

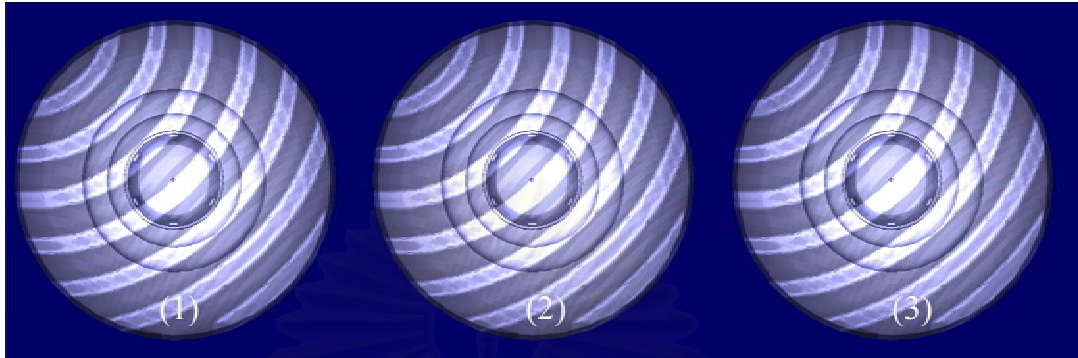


Figure A - 43: Sample results from UFO model.

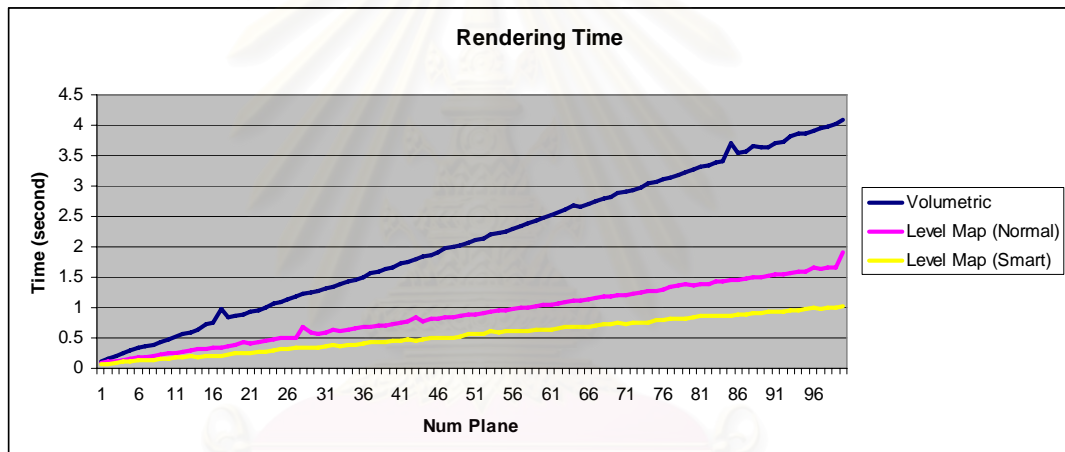


Figure A - 44: Rendering speed comparison graph while using UFO as an input.

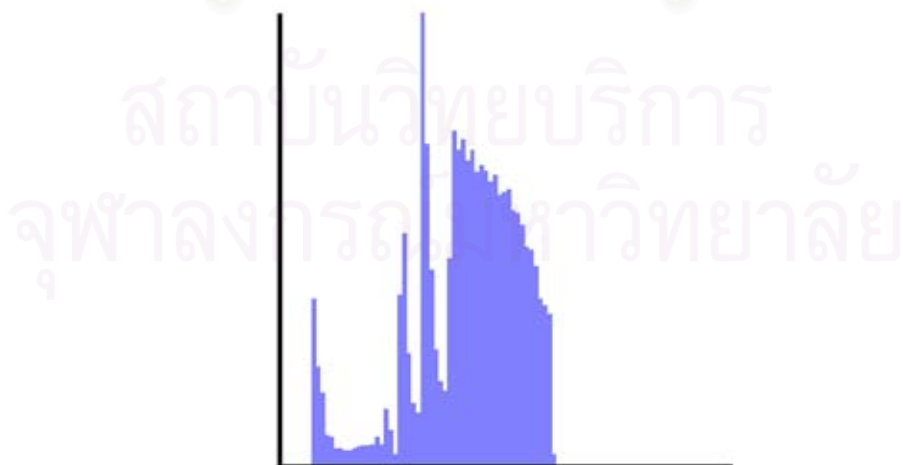


Figure A - 45: Depth histogram of UFO model.

16. Chair

Polygon Count: 190



Figure A - 46: Sample results from chair model.

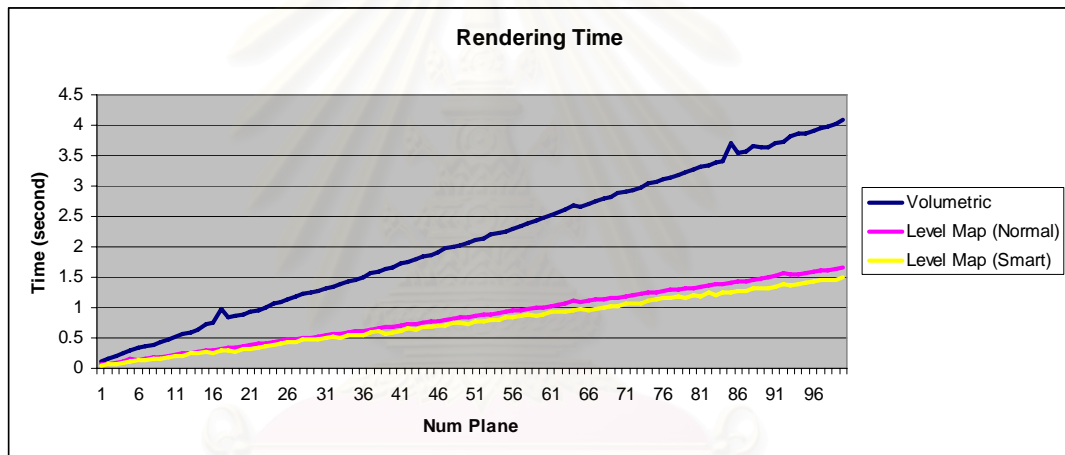


Figure A - 47: Rendering speed comparison graph while using chair as an input.

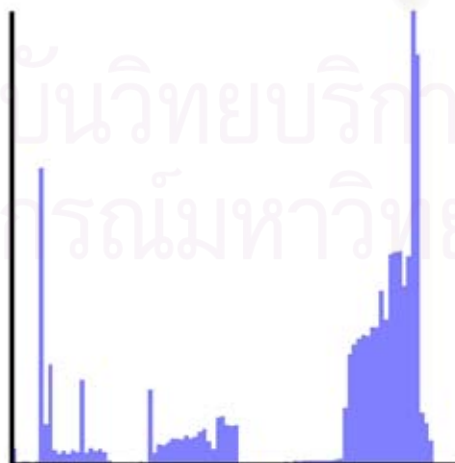


Figure A - 48: Depth histogram of chair model.

17. Barramundi

Polygon Count: 6524



Figure A - 49: Sample results from barramundi model.

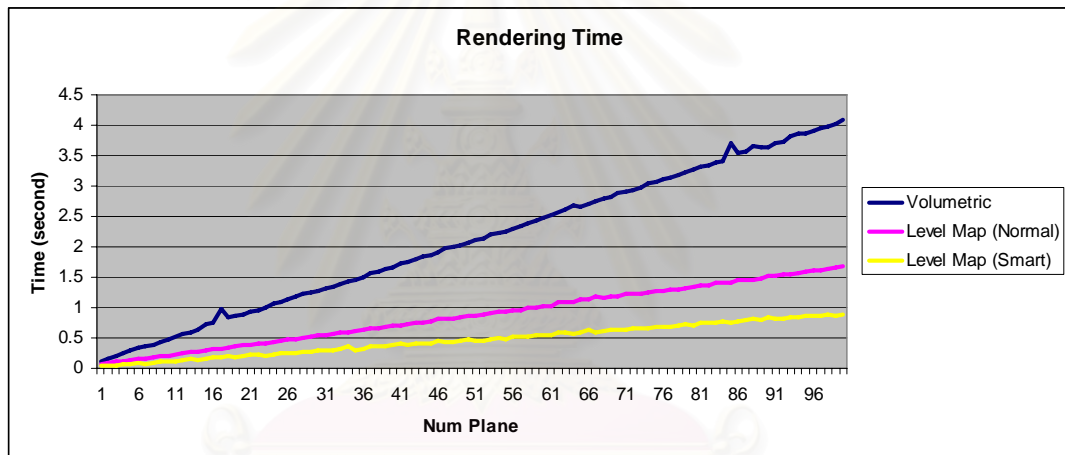


Figure A - 50: Rendering speed comparison graph while using barramundi as an input.

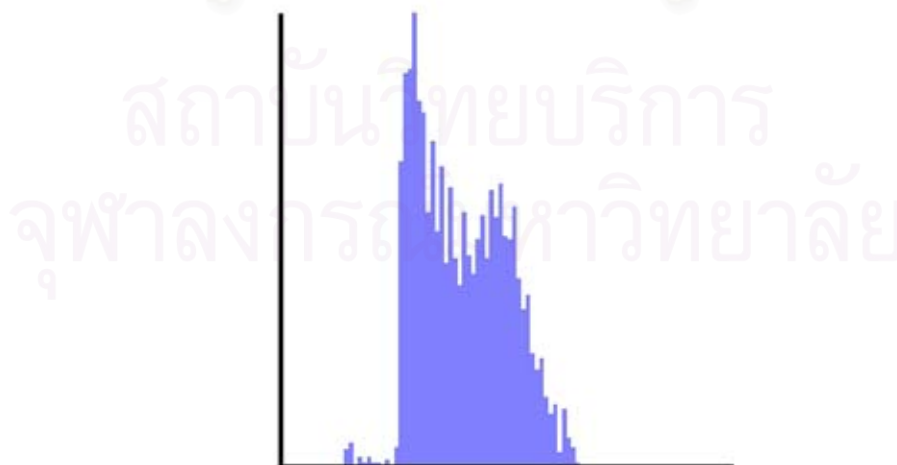


Figure A - 51: Depth histogram of barramundi model.

18. Brown Trout

Polygon Count: 6828



Figure A - 52: Sample results from brown trout model.

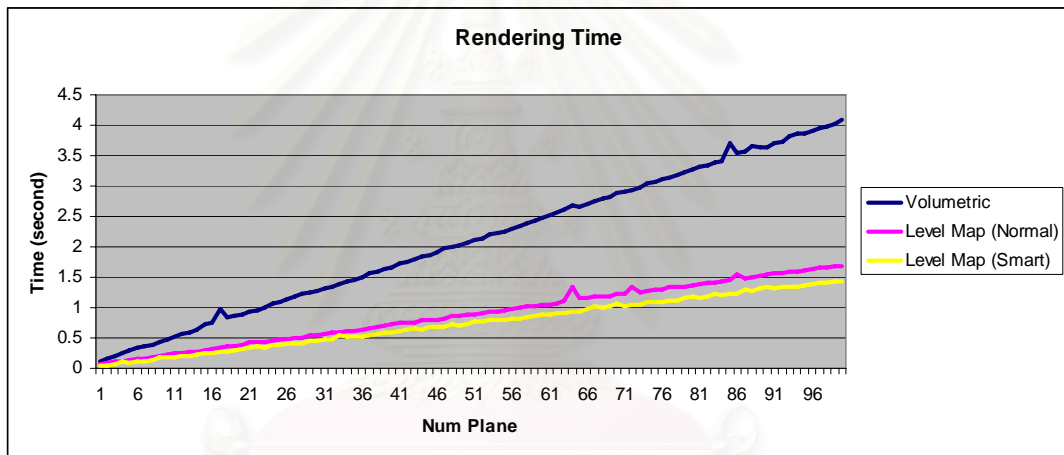


Figure A - 53: Rendering speed comparison graph while using brown trout as an input.

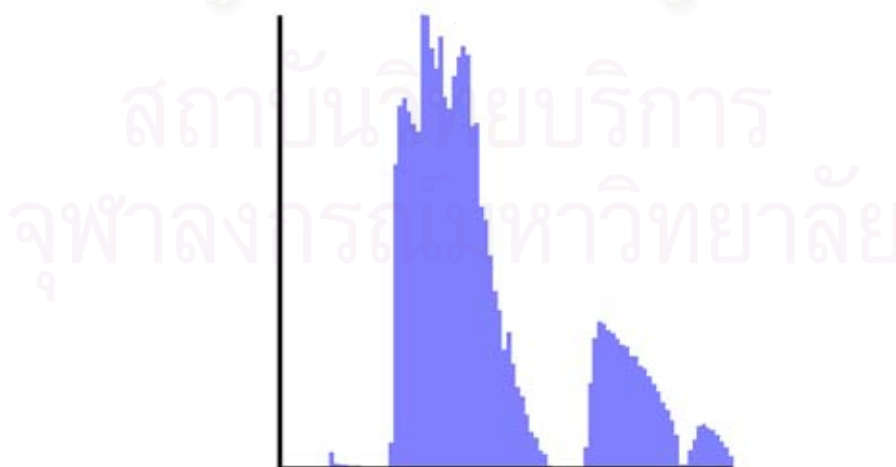


Figure A - 54: Depth histogram of brown trout model.

19. Leopard Shark

Polygon Count: 5200

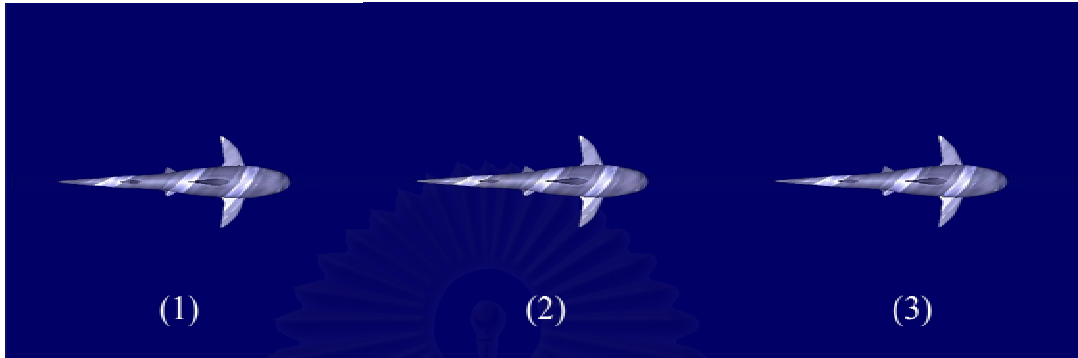


Figure A - 55: Sample results from leopard shark model.

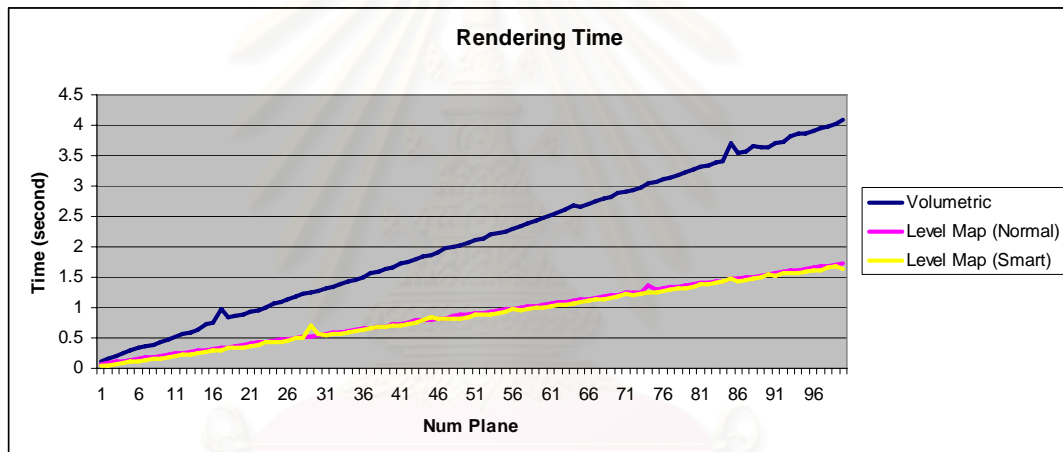


Figure A - 56: Rendering speed comparison graph while using leopard shark as an input.

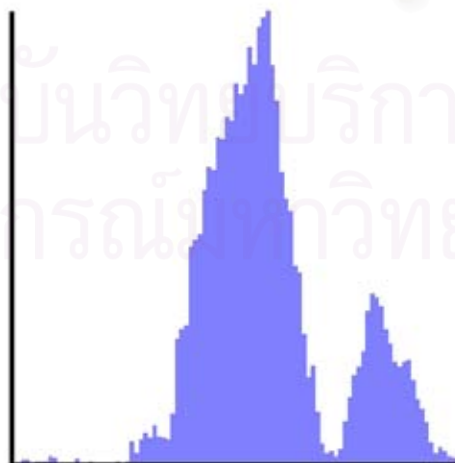


Figure A - 57: Depth histogram of leopard shark model.

20. Lion Head

Polygon Count: 6602



Figure A - 58: Sample results from lion head model.

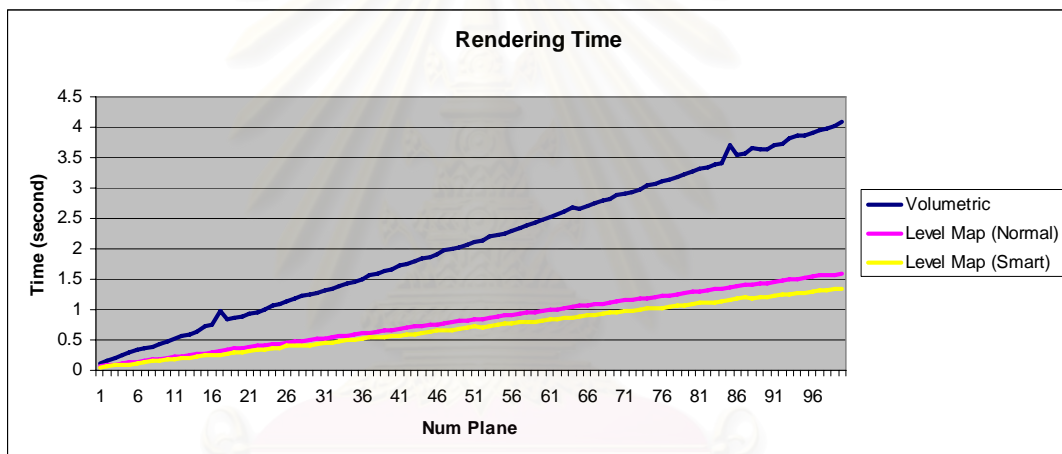


Figure A - 59: Rendering speed comparison graph while using lion head as an input.

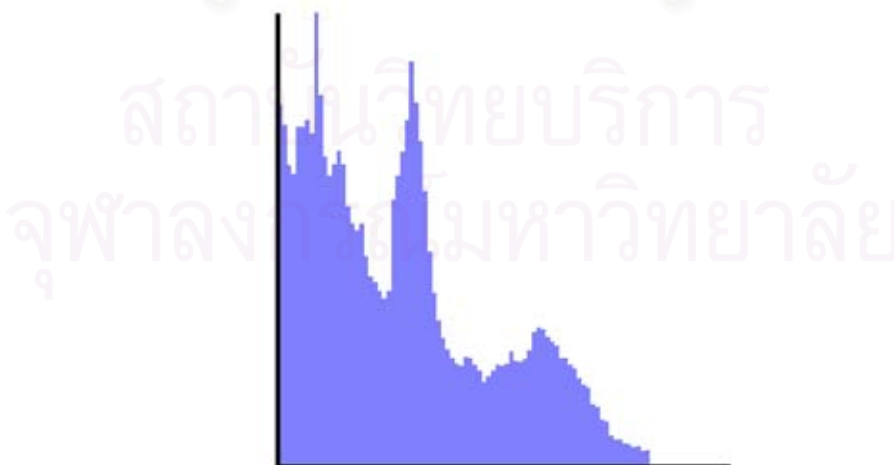


Figure A - 60: Depth histogram of lion head model.

21. Sand Bar Shark

Polygon Count: 5200



Figure A - 61: Sample results from sand bar shark model.

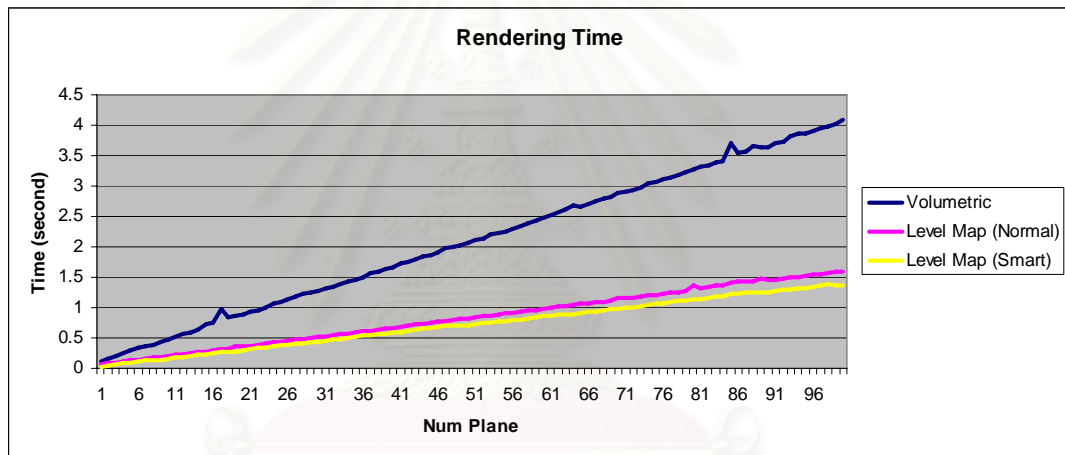


Figure A - 62: Rendering speed comparison graph while using sand bar shark as an input.

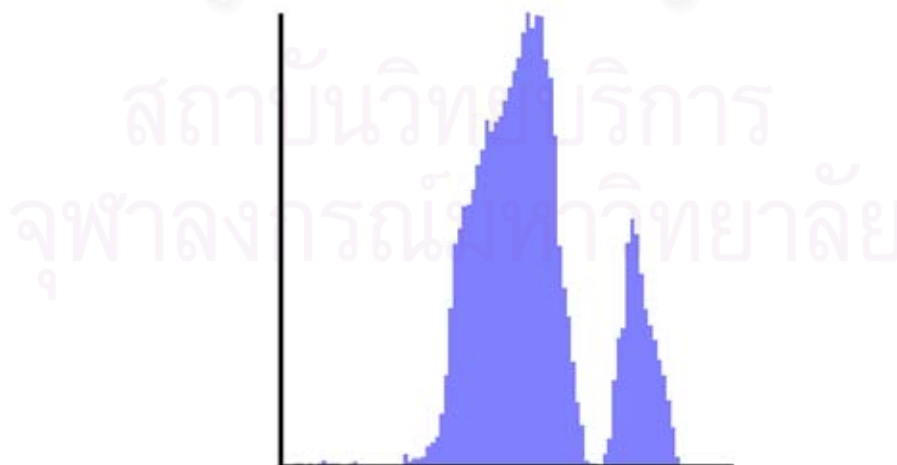


Figure A - 63: Depth histogram of sand bar shark model.

22. Steel Head

Polygon Count: 7226



Figure A - 64: Sample results from steel head model.

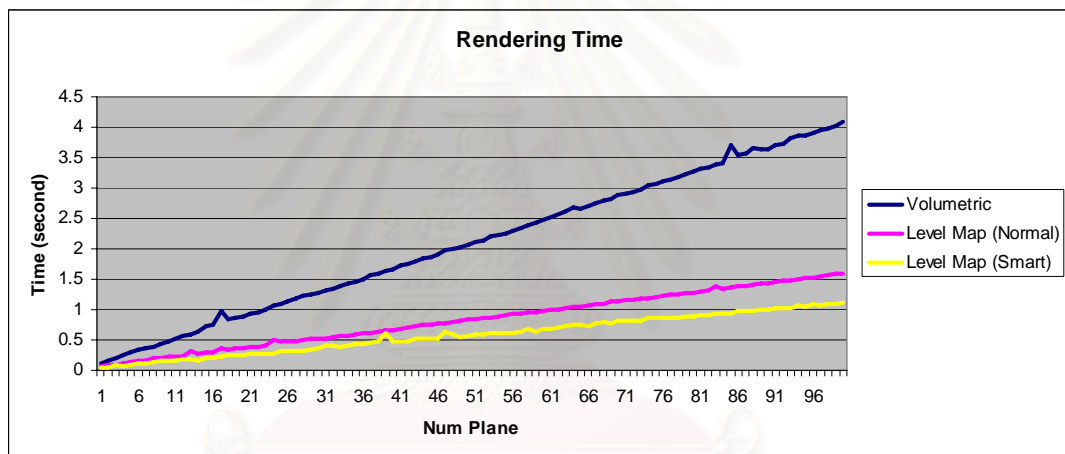


Figure A - 65: Rendering speed comparison graph while using steel head as an input.

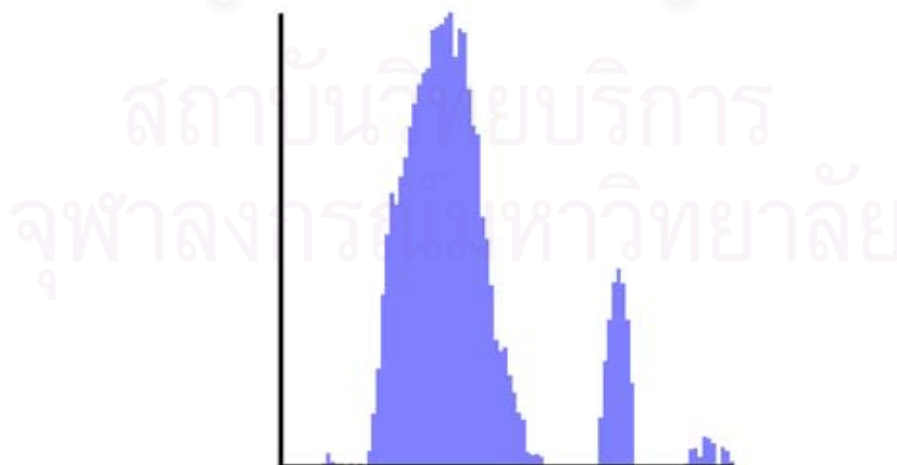


Figure A - 66: Depth histogram of steel head model.

23. Sun Fish

Polygon Count: 4912



Figure A - 67: Sample results from sun fish model.

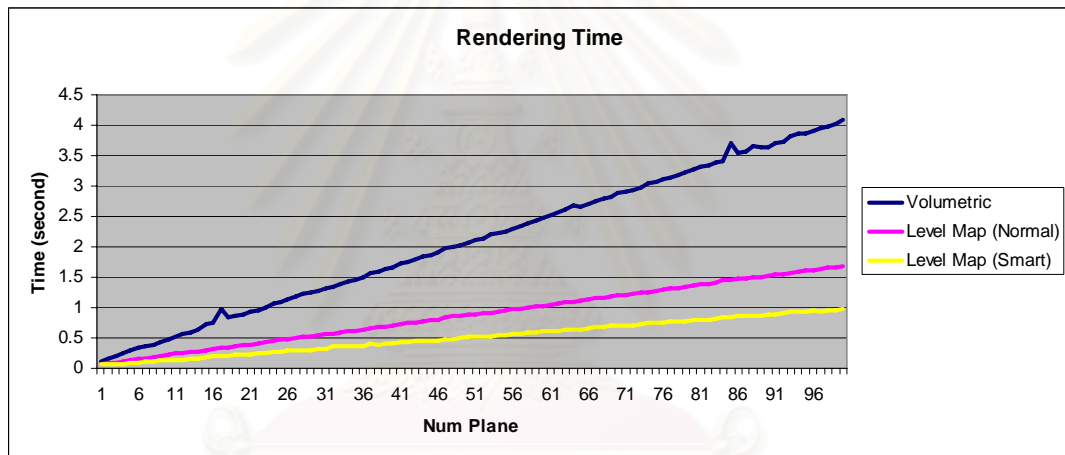


Figure A - 68: Rendering speed comparison graph while using sun fish as an input.

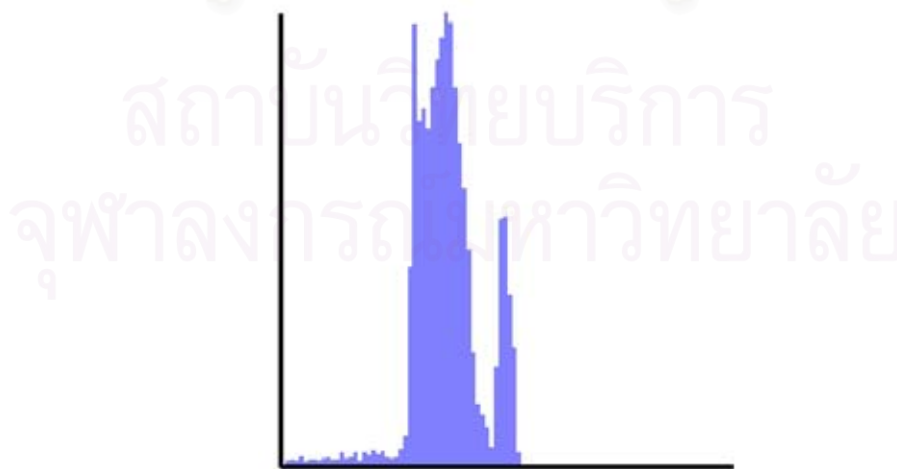


Figure A - 69: Depth histogram of sun fish model.

24. Whale

Polygon Count: 6288



Figure A - 70: Sample results from whale model.

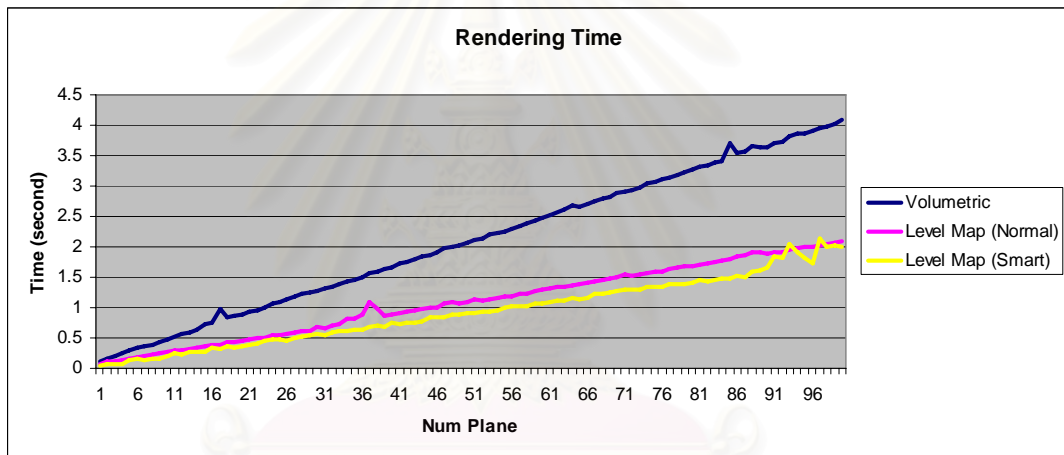


Figure A - 71: Rendering speed comparison graph while using whale as an input.

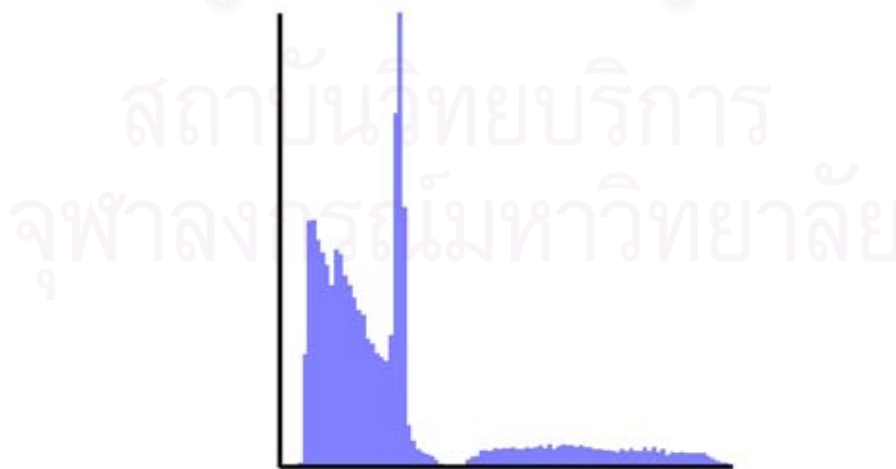


Figure A - 72: Depth histogram of whale model.

25. Camera

Polygon Count: 15496



Figure A - 73: Sample results from camera model.

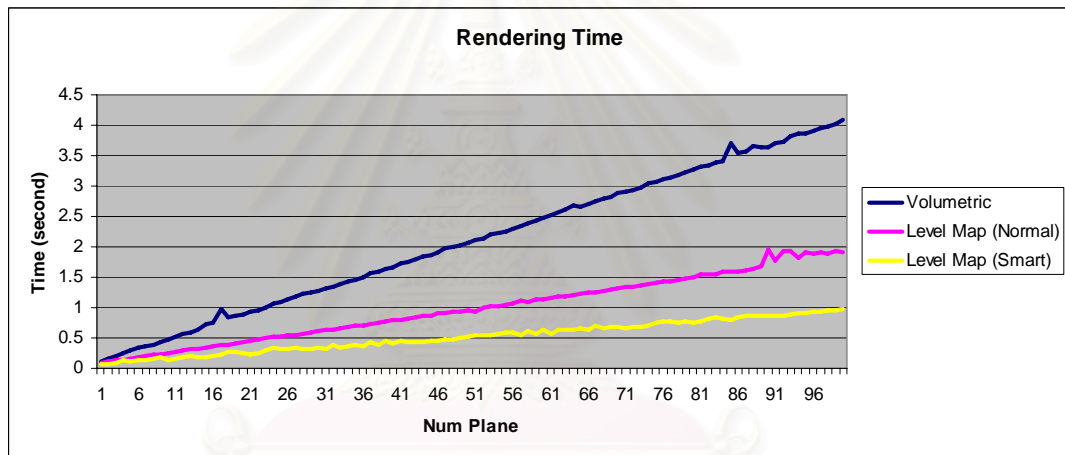


Figure A - 74: Rendering speed comparison graph while using camera as an input.

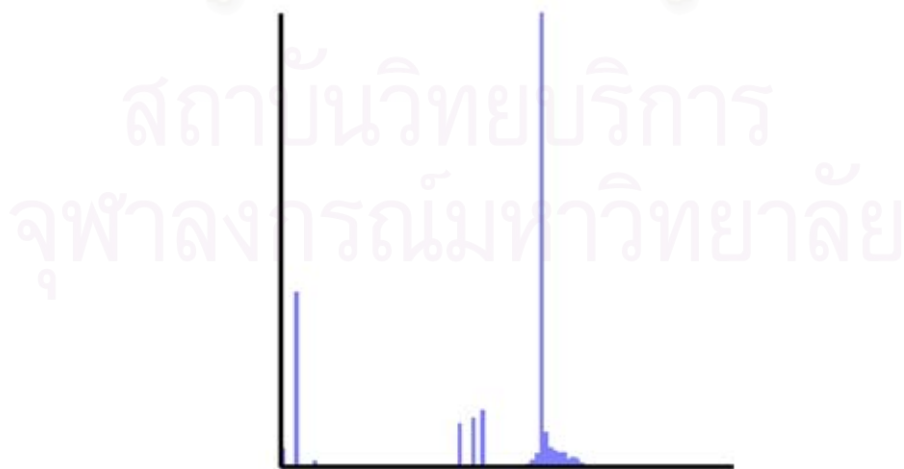


Figure A - 75: Depth histogram of camera model.

26. Cross

Polygon Count: 76



Figure A - 76: Sample results from cross model.

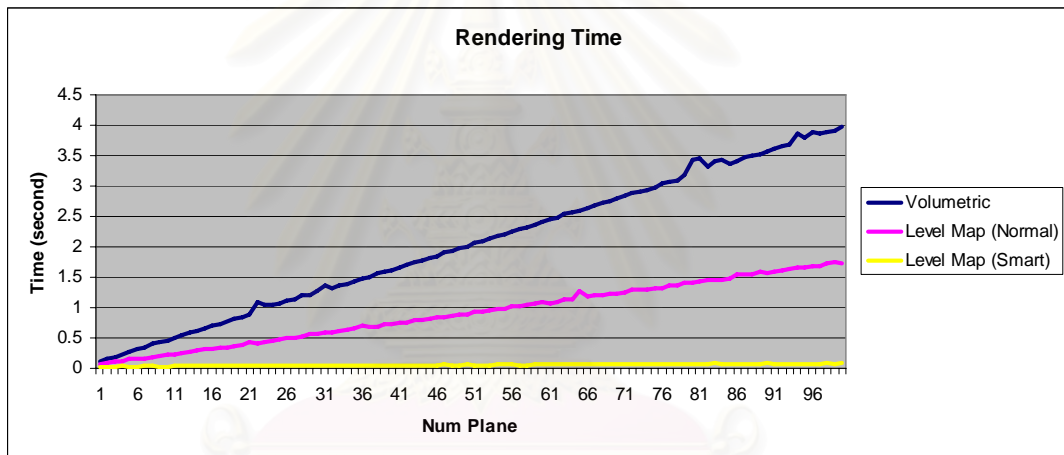


Figure A - 77: Rendering speed comparison graph while using cross as an input.



Figure A - 78: Depth histogram of cross model.

27. Bass

Polygon Count: 2253

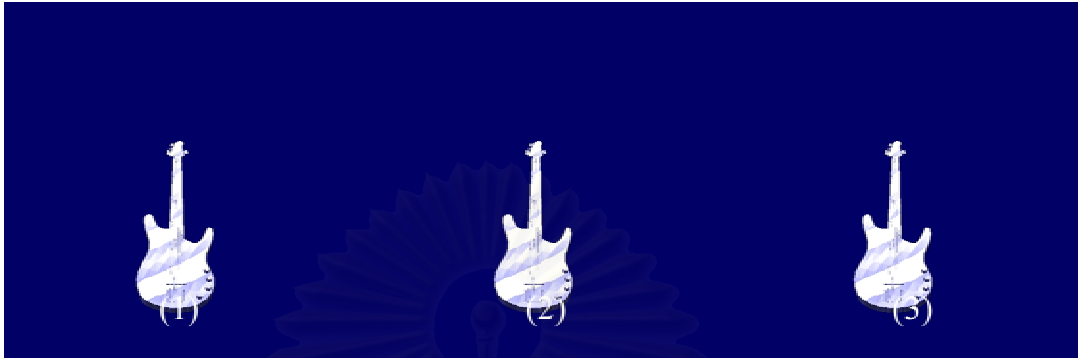


Figure A - 79: Sample results from bass model.

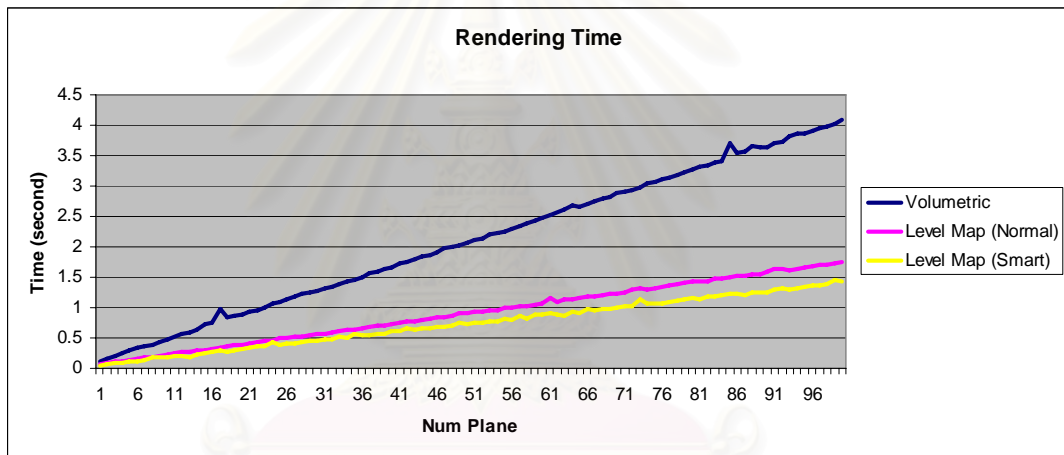


Figure A - 80: Rendering speed comparison graph while using bass as an input.

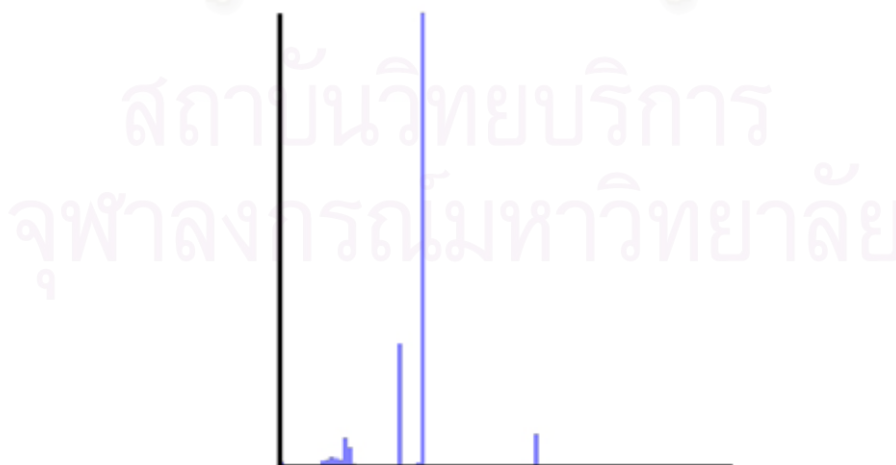


Figure A - 81: Depth histogram of bass model.

28. Plane

Polygon Count: 3030



Figure A - 82: Sample results from plane model.

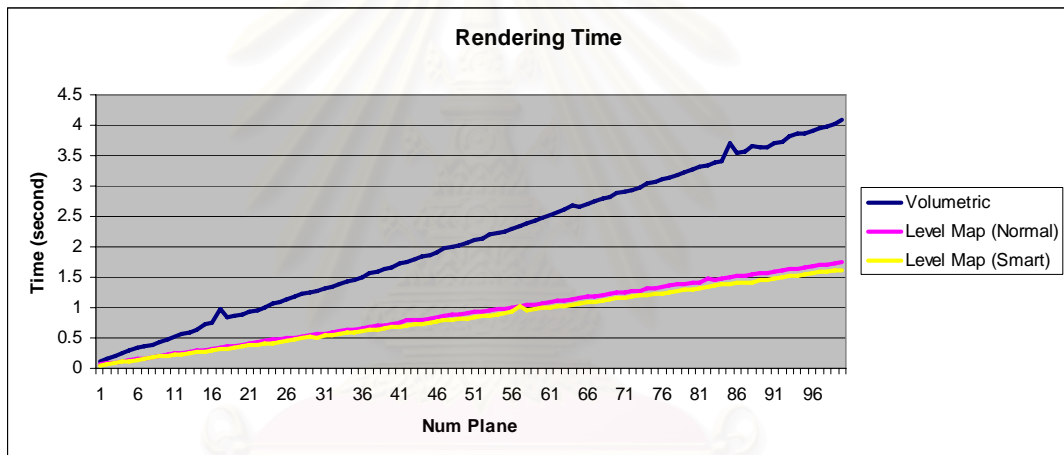


Figure A - 83: Rendering speed comparison graph while using plane as an input.

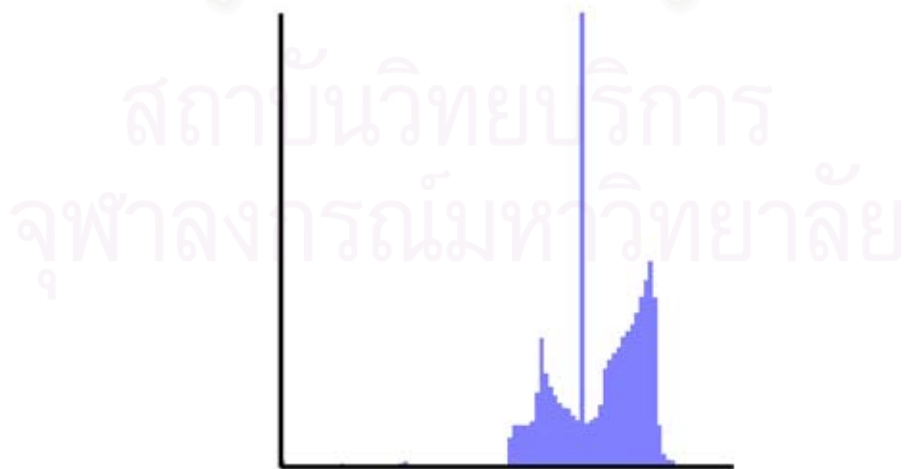


Figure A - 84: Depth histogram of plane model.

29. Plane

Polygon Count: 25812

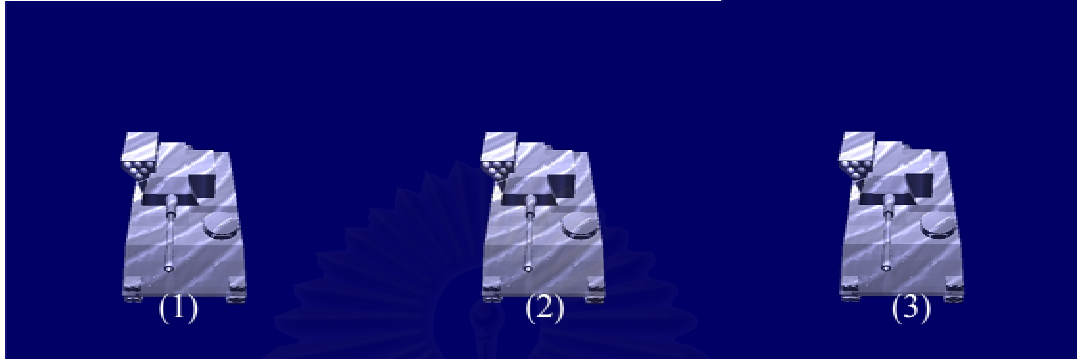


Figure A - 85: Sample results from tank model.

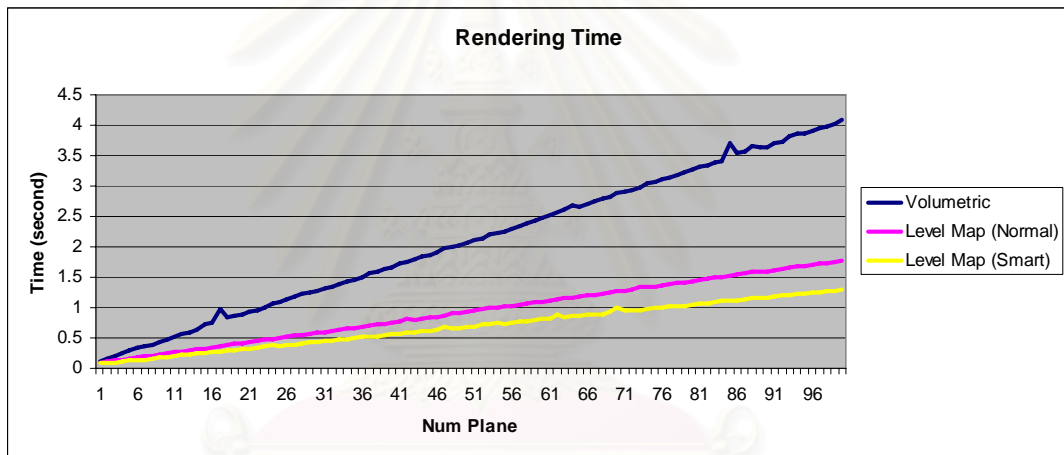


Figure A - 86: Rendering speed comparison graph while using tank as an input.

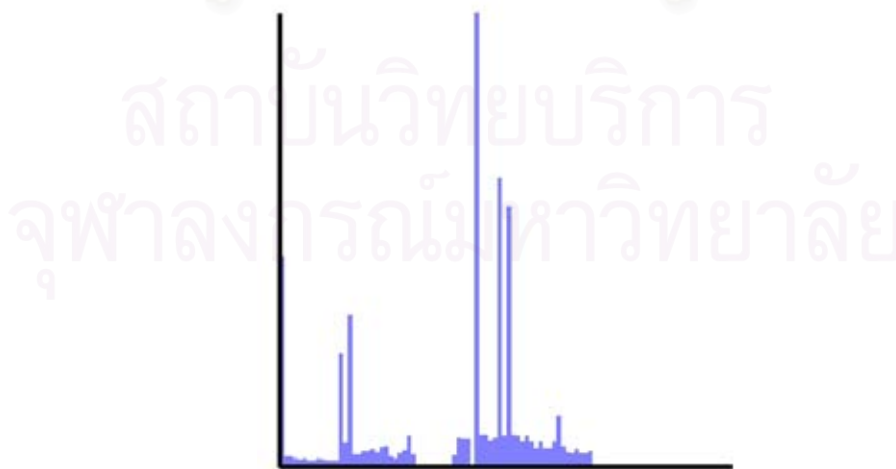


Figure A - 87: Depth histogram of tank model.

30. Hammer

Polygon Count: 13478

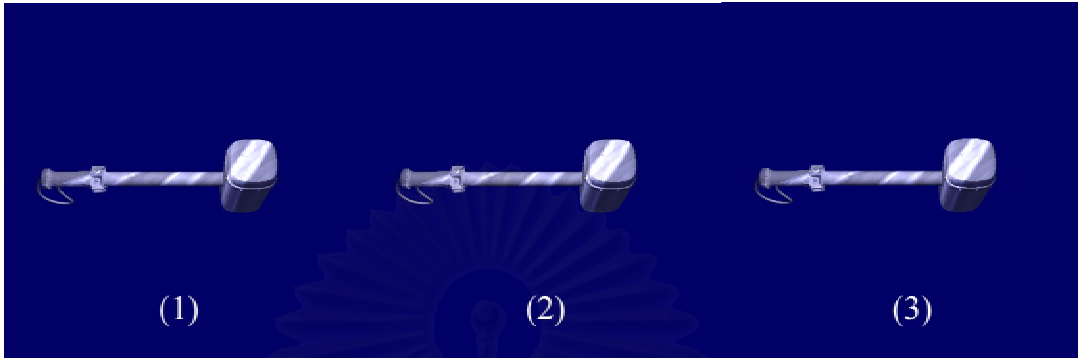


Figure A - 88: Sample results from hammer model.

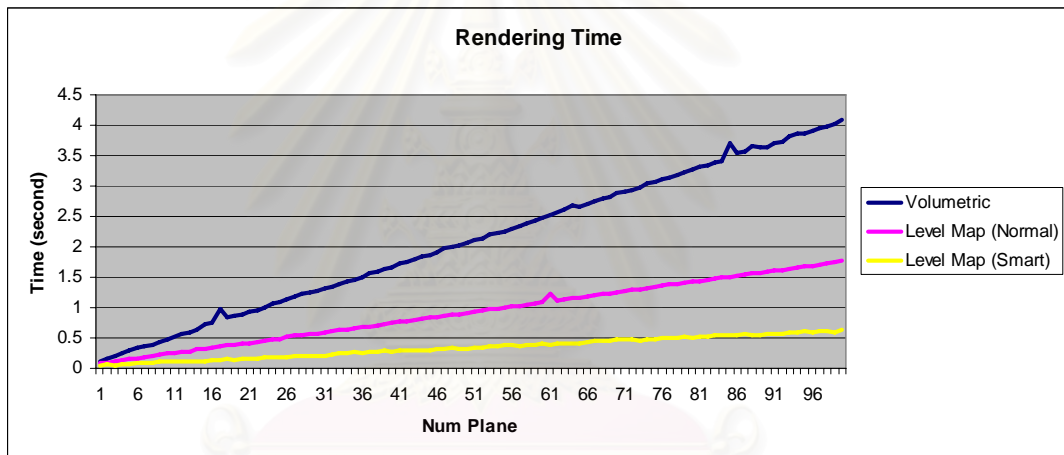


Figure A - 89: Rendering speed comparison graph while using hammer as an input.

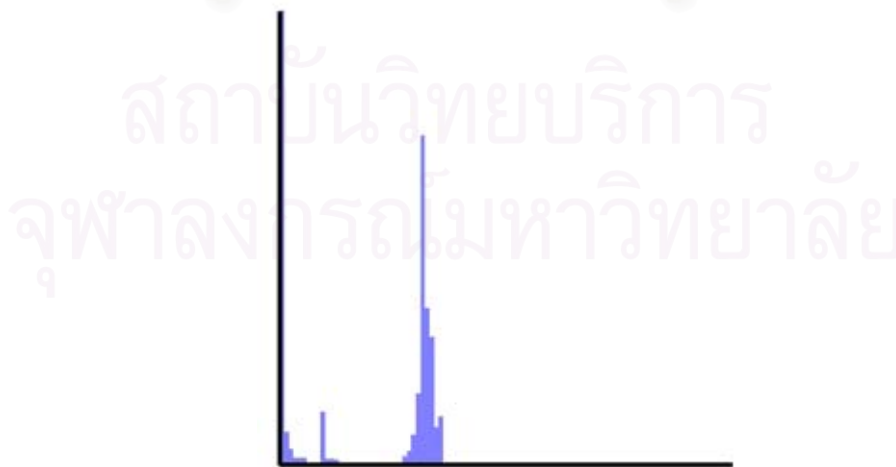


Figure A - 90: Depth histogram of hammer model.

REFERENCES

- [1] Guenther, J., Wald, I., & Slusallek, P. 2004. Realtime Caustics Using Distributed Photon Mapping. Rendering Techniques 2004 : Eurographics Symposium on Rendering: 111-121.
- [2] Purcell, T. J., Donner, C., Cammarano, M., Jensen, H.W., & Hanrahan, P. 2003. Photon Mapping on Programmable Graphics Hardware. Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware (2003): 41-50.
- [3] Trendall, C., & Stewart, A.J. 2000. General Calculation Using Graphics Hardware, with Application to Interactive Caustics. Proceedings of the Eurographics Workshop on Rendering Techniques 2000 (2000): 287-298.
- [4] Wald, I., Kollig, T., Benthin, C., Keller, A., & Slusallek P. 2002. Interactive Global Illumination Using Fast Ray Tracing. Proceedings of the 13th Eurographics workshop on Rendering (2002): 15-24.
- [5] Wyman, C., Hansen, C. D., & Shirley, P. 2004. Interactive Caustics Using Local Precomputed Irradiance. Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04) 00 (2004): 143-151.
- [6] Crespo, D. S., & Guaydado, J. 2004. Rendering Water Caustics. Fernando, R., GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, 31-44. Addison Wesley.
- [7] Stam, J. 1996. Random Caustics: Natural, Textures and Wave Theory Revisited. ACM SIGGRAPH 96 Visual Proceedings: The art and interdisciplinary programs of SIGGRAPH '96 (1996): 150.
- [8] Shah, M. A., & Pattanaik, S. 2005. Caustics Mapping: An Image-Space Technique for Real-Time Caustics. IEEE Transactions on Visualization and Computer Graphics: (To be appeared).
- [9] Wand, M., & Straßer, W. 2003. Real-Time Caustics. Computer Graphics Forum 22 (2003): 610-619.

- [10] Iwasaki, K., Dobashi, Y., & Nishita, T. 2003. A Fast Rendering Method for Refractive and Reflective Caustics Due To Water Surfaces. Computer Graphics Forum 22 (September 2003): 601-609.
- [11] Iwasaki, K., Yoshimoto, F., Dobashi, Y., & Nishita, T. 2005. A Method for Fast Rendering of Caustics from Refraction by Transparent Objects. IEICE Transactions on Information and Systems 2005 E88-D 5 (2005): 904-911.
- [12] Arvo, J. 1986. Backwards Ray Tracing. SIGGRAPH' 86 Course Note 12(August 1986): 259-263.
- [13] Heckbert, P. S. 1990. Adaptive Radiosity Textures for Bidirectional Ray Tracing. Proceedings of the 17th annual conference on Computer graphics and interactive techniques 17 (1990): 145-154.
- [14] Mitchell, D., & Hanrahan, P. 1992. Illumination from Curved Reflectors. Proceedings of the 19th annual conference on Computer graphics and interactive techniques (1992): 283-291.
- [15] Jensen, H. W. 1996. Global Illumination Using Photon Maps. Proceedings of the Eurographics Workshop on Rendering Techniques '96 (1996): 21-30.
- [16] Jensen, H. W. 1996. Rendering Caustics on Non-Lambertian Surfaces. Proceedings of the conference on Graphics interface '96 (1996): 116-121.
- [17] Watt, M. 1990. Light-Water Interaction Using Backward Beam Tracing. Proceedings of the 17th annual conference on Computer graphics and interactive techniques (1990): 377-385.
- [18] Heckbert, P. S., & Hanrahan, P. 1984. Beam Tracing Polygonal objects. Proceedings of the 11th annual conference on Computer graphics and interactive techniques 11 (1984): 119-127.
- [19] Nishita, T., & Nakamae, E. 1994. Method of Displaying Optical Effects within Water Using Accumulation-Buffer. Proceedings of the 21st annual conference on Computer graphics and interactive techniques (1994): 373-379.
- [20] Iwasaki, K., Dobashi, Y., & Nishita, T. 2002. An Efficient Method for Rendering Underwater Optical Effects Using Graphics Hardware. Computer Graphics Forum 21 (November 2002): 701-711.

BIOGRAPHY

Nuttachai Tipprasert was born on November 3, 1980, Bangkok, Thailand. Receive a Bachelor's of Computer engineering Degree from Faculty Engineering, Chulalongkorn University in March 2003. After that he has studied a Master's degree of Computer Engineering at Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University.



สถาบันวิทยบริการ
จุฬาลงกรณ์มหาวิทยาลัย