Fast visualization of complex 3D models using displacement mapping

Lu, The Kiet

2010

Lu, T. K. (2010). Fast visualization of complex 3D models using displacement mapping. Master's thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/47498>

<https://doi.org/10.32657/10356/47498>

Nanyang Technological University

Downloaded on 26 Apr 2023 10:09:40 SGT

27
97
D/17

FAST VISUALIZATION OF COMPLEX 3D MODELS USING DISPLACEMENT MAPPING

LU THE KIET

School of Computer Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfillment of the requirement for the degree of
Master of Computer Engineering

2010

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

Fast Visualization of Complex 3D Models Using Displacement Mapping

A thesis submitted to the Nanyang Technological University

In partial fulfillment of the requirement for the degree of

Master of Engineering

LU THE KIET

SCHOOL OF COMPUTER ENGINEERING

2010

ABSTRACT

Displaying 3D models with high detail surface at an interactive speed has long been a topic of interest in Computer Graphics research. However, most high detail 3D models acquired from 3D Laser scanning devices often contain large number of points (or 3D vertices). Currently, the number of points can even reach into the billion, an example being the non-simplified 3D scanning model of Michaelangelo's David from Stanford scanning repository [1]. Unfortunately, current hardware is still not capable of visualizing such vast amount of data directly. Thus, much effort has been focused on different methods of pre-processing such data in order to enable it to be rendered more efficiently in modern Computer Graphics hardware, e.g. GPUs. However, most current techniques, for example 3D mesh simplification are not efficient enough, requiring a significant amount of pre-processing time.

Having observed these problems, in this thesis, a simple and efficient method is presented for rendering high-detail 3D models at interactive rates using a real-time displacement mapping technique. The method first uses an octree to decompose a 3D model into a set of height fields which are contained in octree cells, and then displays the model by height fields ray-casting method. Furthermore, simply rasterizing the faces of the octree voxels to produce fragments for ray-casting on the GPU, together with straightforward transformation of view rays to the displacement map's local space, the method is able to accurately render the object's silhouettes with very little special handling.

The advantage of the algorithm is that it is especially suitable for fast visualization of high-detail point-based models, and models that are made up of unprocessed triangle meshes coming straight from range scanning. This is because the method employs an adaptive octree partition scheme to simplify models' geometry complexity from 3D surfaces into 2D height fields, which greatly reduces preprocessing time compared to the traditional triangle-based simplification approach. Unlike the point-based rendering approach, the rendering efficiency of the method is not limited by the number of input points. The method can achieve interactive rendering of models with more than 300 millions points on standard graphics hardware.

ACKNOWLEDGEMENTS

First and foremost, I would like to express my special gratitude to my supervisor, Doctor Jianmin Zheng for his invaluable advice and enthusiastic help. In the past few years, I have not only learnt to conduct academic research but also many good characteristics from him.

Second, I wish to express my thanks to people who gave great support for this project. They are Doctor Low Kok Lim (NUS), Doctor Nelson Chu (Microsoft Corporation), Doctor Chunyan Miao (NTU).

TABLE OF CONTENTS

ABSTRACT..... i

ACKNOWLEDGEMENTSiii

TABLE OF CONTENTSiv

LIST OF FIGURESvii

LIST OF TABLES x

CHAPTER 1 INTRODUCTION 1

 1.1 Background and Motivation..... 1

 1.2 Scope 3

 1.3 Contribution of the thesis 4

 1.4 Outline of the Thesis 5

CHAPTER 2 LITERATURE REVIEW 6

 2.1 3D structure representations..... 6

 2.1.1 Polygonal representations of 3D structures 7

 2.1.2 Point-based representations of 3D structures 8

 2.1.3 Octree representations of 3D volume 9

 2.1.4 Height Field representation of 3D surface 10

 2.2 Image-space Ray-casting..... 12

 2.3 Displacement mapping 13

 2.3.1 Height field Displacement mapping principles 15

 2.3.1.1 2D Height field representation 15

2.3.1.2	<i>Coordinate transformations</i>	16
2.3.2	<i>Per-vertex displacement mapping on the GPU</i>	18
2.3.3	<i>Per-pixel displacement mapping on the GPU</i>	19
2.3.3.1	Technical challenges & Classification	19
2.3.3.2	Single step displacement mapping	21
2.3.3.2	Muti-step displacement mapping	25
2.4	Summary	41
CHAPTER 3 NEW VISUALIZATION ALGORITHM		42
3.1	Overview	42
3.1.1	<i>Surface Decomposition</i>	43
3.1.2	<i>Rendering</i>	45
3.2	Height Field Decomposition	46
3.2.1	<i>Octree Decomposition</i>	46
3.2.2	<i>Height Field Test</i>	47
3.3	Setting up Displacement Mapping	49
3.3.1	<i>Height Map Generation</i>	49
3.3.2	<i>Coordinates Transformation</i>	51
3.4	Rendering	53
3.4.1	<i>Boundary Culling</i>	60
3.5	Advantage of the new Algorithm	62
CHAPTER 4 EXPERIMENTAL RESULTS AND COMPARISON		64
4.1	Algorithm tests	64
4.1.1	<i>Quality test</i>	65

4.1.2 Performance test 67

4.2 Experiment on several Stanford scanning 3D models 68

4.3 Performance and quality comparison test with other methods 71

4.3.1 Comparison with Qsplat 71

4.3.2 Comparison with Efficient Point-Based Rendering 74

CHAPTER 5 CONCLUSION AND FUTURE WORK 76

REFERENCES 79

AUTHOR PUBLICATION 86

TABLE OF FIGURES

Figure 2.1: Illustration of octree decomposition.....	10
Figure 2.2: Illustration of 2D height field representation.	11
Figure 2.3: Illustration of space-skipping in octree traversal.	13
Figure 2.4: 3D decomposition of meso-structure into 2D Height Fields.....	16
Figure 2.5: Illustration of the world and local coordinate system	17
Figure 2.6: Overview of per-pixel displacement mapping technique.....	20
Figure 2.7: An example of inaccurate calculation of Ray - Height Field intersection	21
Figure 2.8: Comparison between Bump mapping (LEFT) and Non-iterative parallax mapping (RIGHT).....	24
Figure 2.9: Comparison between offset - limit parallax mapping (LEFT) and Original parallax mapping (RIGHT).....	25
Figure 2.10: Comparison between Non-iterative parallax mapping (LEFT) and Iterative Parallax mapping (10 iterations) (RIGHT).	27
Figure 2.11: Illustration of Linear search for displacement mapping.....	28
Figure 2.12: Comparison between Linear search (LEFT) – 12 iterations and parallax mapping (RIGHT) – 12 iterations.....	30
Figure 2.13 TOP: illustration of binary search scheme with 3 iteration steps. BOTTOM: Case where binary search gives incorrect intersection.	32
Figure 2.14: Comparison between Binary search and parallax mapping.	34
Figure 2.15 Illustration of secant culling method of Interval displacement mapping	35

Figure 2.16 TOP: Binary search displacement mapping. BOTTOM: Secant Interval displacement mapping.....	36
Figure 2.17: Comparison between Linear search – 12 iterations (LEFT) and Sphere tracing – 12 iterations (RIGHT).....	38
Figure 2.18: Illustration of sphere tracing algorithm. Image taken from [54].....	38
Figure 2.19: Illustration of Cone step tracing algorithm. Image taken from [4]	39
Figure 2.20: Comparison between Relief mapping (Linear (12 iterations) and binary search (5 iterations) - LEFT) and Cone step mapping – 12 iterations (RIGHT).....	40
Figure 3.1: Decomposition of 3D models into 2D height field surfaces.	44
Figure 3.2: Illustration of several steps of the algorithm	45
Figure 3.3: PCA transformation applied on 2D surface for height field construction.....	48
Figure 3.4: Illustration of texture atlas of the height field set.....	50
Figure 3.5: Illustration of local space for each height field.	51
Figure 3.6: Transformation of coordinates..	52
Figure 3.7: Illustration of two cases of Ray casting of height fields	54
Figure 3.8: Illustration of Iterative Parallax ray Casting result over different steps.....	59
Figure 3.9: Difference between 8 and 12 iterations.....	60
Figure 3.10: Illustration of the difference between this new algorithm and previous displacement mapping without silhouette handling.....	61
Figure 3.11: Illustration of the advantage of this new algorithm at handling with silhouette effect	62
Figure 4.1: Iterative parallax mapping with different number of iterations	66
Figure 4.2: The St. Mathew model (368M) with its height field decomposition cells.....	69

Figure 4.3: Sample images produced by the rendering algorithm. From left to right: St. Mathew (368M), David (28M), Lucy (14M), and Thai Statue (5M)..... 70

Figure 4.4: Closed-up of Lucy 3D model. LEFT: Result of Qsplat’s rendering method. RIGHT: result of new rendering method. 73

Figure 4.5: A Close-up view of the Thai statue 3D model. 75

TABLE OF TABLES

Table 1: Rendering results 71

Table 2: Performance comparison between new method and with Qsplat. 71

Table 3: Performance comparison between new method and with Marroquim's method. 74

CHAPTER 1

INTRODUCTION

This chapter presents the background, motivation, and scope of the research, and also the contributions and the outline of the thesis.

1.1 Background and Motivation

Interactive visualization of high-detail 3D models has always been one of the main research interests in computer graphics. Range scanning has become a common means to model real-world objects. Today's advancement in range scanning technology has enabled us to record even fine brush strokes on paintings and chisel marks on sculptures [1]. However, the problem is that the output data often contains very detailed and complex 3D models making display at interactive framerates extremely difficult. Earlier research has mostly focused on mesh simplification algorithms to reduce the number of vertices and triangles to be processed at the rendering stage [2]. However, as these methods require too much preprocessing time [3], more recently researchers have become interested in using the input data directly for rendering (for example, using points as rendering

primitives). While these point-based rendering techniques can reach interactive frame rates for certain 3D models, the main bottleneck still lies at the vertex processing stage such that this approach may not cope well with the rapid growth of data size in the near future.

On the other hand, an alternative rendering approach called “Per-pixel Displacement mapping” (PDM) has recently gained much research focus. PDM performs ray-casting of 2D height field on the local surface instead of processing individual points from the original 3D models like other conventional methods. It is capable of providing a fast image-based visualization, especially for surfaces with high detail at local level or “meso” structure - see Chapter 2.3. However, most PDM techniques perform displacement mapping on disconnected 2D planar height field domains bounded by either a triangle or quadrilateral mesh. They may not maintain the continuity between neighboring height fields if we apply them to connected 2D planar height field domains [4]. This usually results in inaccurate and unpleasing silhouette artifacts at the borders of the triangles or quadrilaterals [5]. Many researchers have proposed different algorithms to compensate the drawbacks [6, 7]. However, most of the current algorithms require many efforts to handle the silhouette effect of the displacement mapping, which is not suitable for interactive visualization.

This thesis proposes an alternative solution to the previously mentioned real-time visualization problem that aims to provide better approach on visualization without being constrained by the number of input points. By decomposing the input 3D model into a set of simple 2D height fields and displaying the model by rendering these height fields using enhanced PDM, the proposed method has provided better results in term of quality and speed. The technique performs image-space ray-casting on the 2D height fields and allows us to avoid the vertex-processing bottleneck that plagues many point-based rendering techniques. The enhanced displacement mapping can render efficiently at

interactive rates, with accurate rendering of the object's silhouettes without special handling. The preservation of the surface details is only limited by the amount of texture memory on the graphics hardware. The method is general enough. It can also take in unprocessed triangle meshes directly from range scanning devices as input and render them efficiently.

1.2 Scope

The thesis focuses on solving the bottleneck problem of visualizing high-detail 3D models which is often seen in point-based rendering methods where the number of points is in general very high (up to millions of points).

In order to improve the visualization performance, a PDM technique is employed for fast rendering of the 3D models' detail without being constraint by the number of points, as of many point-based methods. However, there are still many issues related to previous PDM techniques, such as correct silhouette rendering, boundary continuity, and ray marching performance as well as memory consumption. Each issue is an interesting research topic and provides many challenges to the research community. The contribution of this work is to solve the visualization problem by introducing a new octree-based rendering approach in order to accurately render the height field's silhouette which results in high performance rendering of high detail 3D models that the conventional PDM techniques or point-based methods are not capable of.

Additionally, for the simplicity of performing comparisons between the new rendering method and the previous rendering pipeline such as triangle-based and point-based's methods, we make the assumption that the GPUs have enough memory to store all 2D height field data into the alpha channel of the texture atlas. Moreover, some other existing

hardware-related techniques for performance optimizing (for example, view-dependent culling, front to back depth-sorting, texture compression, and GPU data fetch caching) are excluded from this discussion.

1.3 Contribution of the thesis

In this thesis, we propose a new algorithm for fast visualization of detailed 3D models. The algorithm first uses an octree decomposition scheme and Principle Component Analysis (PCA) technique to convert the original detail 3D model into simpler 2D height fields. During rendering, each face of the octree voxel will be rasterized to produce fragments for ray-casting on the GPU. With this approach, the object's silhouettes can be well rendered even when using a simple 2D height field ray-casting algorithm, such as the iterative parallax mapping algorithm [8]. This does not like existing stand-alone displacement mapping techniques, which usually do not pay much attention to the continuity between neighboring 2D height fields, thus often resulting in inaccurate silhouette effects [4].

Note that some new PDM techniques require complex decomposition of the input model into a set of base polygons or volumes and use more complex algorithms to compute intersections between curved viewing rays and the displacement maps [5, 9-11] which make them unsuitable for real-time visualization purpose.

The main contribution of this thesis is the first algorithm that allows PDM techniques to be performed efficiently and accurately with very little extra effort needed to handle the displacement mapping silhouette and PDM boundary continuity issues. The method is also a better alternative solution for large point-based model visualization as it does not have the vertex processing bottleneck.

1.4 Outline of the Thesis

The thesis is organized as follows:

- Chapter 1: Introduction. This chapter provides the basic background, motivation, and scope of the research, and also the contributions and the outline of the thesis.
- Chapter 2: Literature Review. This chapter provides literature review on common 3D representations using different rendering primitives (i.e. triangle mesh, point cloud), as well as supporting data structures (i.e. hierarchical octree and 2D height field), and finally related rendering methods (i.e. Ray-casting and Displacement mapping).
- Chapter 3: New Visualization Algorithm. This chapter presents a new visualization algorithm dedicated to fast visualization application of high detail 3D models whose number of points may reach up to millions.
- Chapter 4: Experimental Results and Comparison. This chapter presents some experiments with the new algorithm tested on several 3D models taken from Stanford's scanning repository. It also provides the result comparison with most recent 3D point-based rendering methods (in both speed and quality).
- Chapter 5: Conclusion and Future Work. This chapter summarizes the proposed method, its advantages as well as some limitations. It also discusses about possible future improvement.

CHAPTER 2

LITERATURE REVIEW

This chapter provides literature review on 3D structure representations, ray-casting, and several Displacement mapping techniques.

--- 3D structure representations

One of the fundamental problems in Computer Graphics research is to build up a basic system to represent arbitrary 3D structures for visualization applications. There exist many different representations for 3D structures, which facilitate various visualization purposes. For example, to visualize an architectural building consisting of many flat and precise quadrilaterals or even a piece of Archaeological item with a lot of small scale details on its surface might require two different types of 3D representation for practical visualization purposes. In this section, a brief review of a number of common 3D structure representations in literature including their advantages and disadvantages is presented.

2.1.1 Polygonal representations of 3D structures

3D structures with smooth curved or flat surfaces can be approximated efficiently by a set of polygonal faces – for example, triangles. In fact, polygonal representation is popular for its robustness of representation for almost any type of 3D surface as long as there are sufficient triangles or polygons for approximation. Furthermore, there are many techniques to simplify the representation by approximation [2] or interpolation [12]. This may be another reason for the popularity of such a representation.

However, the major limitation of polygonal representation is the visible linearity detected around silhouette edges [13]. The way to improve the visual quality is done by increasing the polygonal resolution. This means that we have to add more faces to the area with high curvature. As a result, for 3D structures with high details on the surface, it is not uncommon for a large number of polygons (e.g. more than 10^5) to be used. This raises a problem for interactive visualization in terms of memory limitation in the computer hardware as well as its accessing and rendering time.

[14] introduced an alternative approach of warping 3D polygonal information into 2D texture for data compression purpose. However, their method might face potential problem with data compression efficiency on high detail 3D models for several reasons:

- 1) In [14], the whole 3D geometry is warped into a single squared texture. The geometry data will generally be compressed unequally among the model. That is, in some areas, the geometry will be persevered in less detail (with less pixels) than the others. This could potentially lead to unwanted artifacts for high detail visualization of complex 3D models.

- 2) The second point is about the final data size after the “flattening” step. The authors emphasized merely on the 2D resolution of the texture but did not focus much on

the depth range of the texture (e.g. floating point value texture). In fact, their method needs to use 32-bit floating point textures to store the 3D displacement of the geometry instead of traditional 8-bit texture format. They are actually 4 times larger which could cause unnecessary data redundancy on high detail 3D models.

3) The time of “flattening” operation could also limit their application to high detail 3D models. For example, heuristically converting such a huge amount of geometry into a square texture is very time consuming. This could take many days for those scanned 3D models with very high detail geometry like St. Mathew with millions of points.

2.1.2 Point-based representations of 3D structures

In recent years, the point-based representation has become more and more popular and gained many attentions in the Computer Graphics community as an alternative high detail visualization approach. The method is interesting for its flexibility in displaying and processing of detailed 3D structures. For example, compared to polygonal method, point-based method treats individual points as rendering primitives and this concept in some certain cases has simplified the data structure and algorithms. For example, culling off the topological link between primitives in point-based representations becomes trivial while it is not easy for triangle mesh data structure. Therefore, the method is considered to be more flexible compared to polygonal representation when it comes to handle high detail visualization data.

The second reason for using point-based representation is due to quick demand for visualization of range data format. Traditionally, rendering of the acquired 3D structure can only take place after the overlapping range image has been aligned (registered),

merged. Moreover, it requires a lot of further processing such as reconstruction, which would normally take many hours of computation and consume a huge amount of memory for very detailed scans. This is not desirable as users need instant visualization of the scanning result for editing.

Rusinkiewicz and Levoy [3] have proposed to use points as rendering primitives, which can display point-sampled data directly without expensive preprocessing. However, since each point is rasterized or “splatted” directly onto the frame buffer without neighborhood interpolation between point primitives, point-based rendering often results in aliasing artifacts – can be seen in Figure 4.4. For high-quality rendering, low-pass filtering is employed and has been implemented on hardware [15] [16] [15] [17]. However, low-pass elliptical weighted average (EWA) filtering is an expensive operation as it requires multiple passes for rendering smooth surfaces on GPU. The performance, as a result, is thus sacrificed for visual quality. Most recently, Marroquim et al. [17] introduced a new point-based rendering algorithm that has good rendering performance while still maintaining high-quality results. It uses pull-push interpolation in image space to reconstruct the surfaces from points. The method claims to have better performance than previous multi-pass methods because it is more GPU-friendly. It requires only single rendering pass for point projection, and is less limited by the vertex processing bottleneck.

2.1.3 Octree representations of 3D volume

Different from polygonal or point-based method for representation of 3D structure which approximates the structure’s surface through directly rendering primitives i.e. triangles or quadrilaterals or a set of individual 3D points, octree is another attractive data structure

that is also often employed in Computer Graphics research for efficient 3D structure representation and rendering [18]. In general, Octree representation is a hierarchical partition of the volume that bounds the current 3D structures. It decomposes the 3D volume by subdividing it into eight equal nodes called octants along three main axes, which offers a well-regulated node spacing (each child node has exactly half space reduction from its parent in each axis direction). See Figure 2.1 for an illustration. Thus, the resulting volume is in fact a union of every leaf node of the octree whose positions and sizes are regulated variables [18-20]. Due to its well-regulated decomposition, Octree data structure is often integrated into many visualization techniques, such as occlusion, space skipping, hierarchical resolution, and data compression [21].

Figure 2.1: Illustration of octree decomposition

For 3D structure representation efficiency, if any octant is completely inside or outside the structure's surface, the node will be discarded (void cells). Hence, only nodes whose octant intersects with the structure's surface will be kept and they will be recursively decomposed into eight lower level nodes (i.e., child nodes), each of which is tested again until it falls inside or outside the structure's volume or the decomposition has reach a desired criterion, (for example, octree's maximum subdivision level).

2.1.4 Height Field representation of 3D surface

A height field or height map, on the other hand, is a set of vertical height scalar sampled over a 2D planar domain. Terrain depth map is a common example of height field which is used in various applications (e.g., in video games or flight simulation or geographic visualization). An efficient data structure for representing this special surface is important. It is not efficient if it is represented as a set of 3D positions (x, y, z) lying on the surface as 3D geometry representations. In computer graphics, a height map is often a rasterized 2D image that can be stored as grayscale image format, with or without additional data for visualization purpose. A height field map usually occupies one color channel of the image interpreted as a displacement of the surface, extruding from the “base floor” upward, with black or Zero value usually representing minimum level and white or One value representing maximum level. Figure 2.2 (TOP) gives the illustration of height field representation generated by 2D Sine wave function. The maximum level of height field is assigned with white color while the minimum level is assigned with black color. The cross section (BOTTOM) of the surface is used to illustrate how the height field’s value should be set. See Figure 2.2 (BOTTOM).

Figure 2.2: Illustration of 2D height field representation. TOP: Height field of sine wave in 3D. BOTTOM: A cross-section view of the surface

2.2 Image-space Ray-casting

Image-space ray-casting was introduced by Roth in 1982 [22] for visualizing and modeling 3D solid objects. In general, image-space ray-casting offers more flexible visualization than the conventional triangle-based rasterization approach. For example, Levoy introduced a 3D volume rendering algorithm for displaying medical images by compositing colored opacity along viewing rays [23]. Ray-casting is also used for accurate rendering of parametric surfaces [24] [25]. Parametric surfaces such as Bezier or NURBs patches are defined only by sets of control points and cannot be rendered directly using triangle-based rasterization.

Ray-casting is also a very powerful 3D visualization approach for its capability of reducing the geometric processing workload (world-view transformation) to a problem that solves for the intersection between viewing rays against the 3D surface. The advantage of this approach is that the performance is easily improved by organizing the graphics primitives (e.g., triangles or 3D) points into a data structure such as k-d trees, grids, octrees, etc, which will reduce the number of ray-intersection test per pixel/ray and thus speed up the performance. Figure 2.3 illustrates the advantage of Octree structure for ray-intersection calculation, in 2D form i.e. Quad tree. Instead of performing the intersection calculation for all primitives, only primitives inside a few quad cells (grey

shaded) along the ray are needed for testing, thus the overall number of calculations is reduced.

Figure 2.3: Illustration of space-skipping in octree traversal. Only gray color cells are tested for ray intersection.

Before the introduction of GPUs, ray casting was performed on a single CPU or in parallel on a cluster of CPUs [2, 26]. A single CPU performs ray-surface intersection individually and sequentially, which is not efficient. However, with the introduction of multi-threading concepts on multi CPU cores architectures, ray-casting has become more and more efficient [27]. With the increasing flexibility in GPU, ray-casting in GPU has emerged and is gaining popularity [28]. Unfortunately, most of the approaches for ray-casting on GPU still rely on pre-computed space partition data structure, and the cost of storing and traversing data structures is relatively expensive [29-31].

2.3 Displacement mapping

One efficient way of ray-casting is the ray-casting on 2D height field, which is often regarded as Displacement mapping techniques. It works especially well for high-detail

surface. In fact, 3D structures can be subdivided into three types of categories: macro-structure, meso-structure and micro-structure. Macro-structure refers to the high level of geometries that are often made up by large scale polygonal primitives, such as triangles and quadrilaterals. Micro-structure, on the other hand, is often related to tiny geometrical facets that are visually impossible to differentiate by human eyes. In literature, there are many shading algorithms employed to handle micro-structure efficiently such as bi-directional reflectional functions [10] [32-34], or just simply 2D texture mapping [35-38].

However, meso-structure is left as a special type of geometry that cannot be fit into any of the previous two categories. On one hand, its size is too small compared to its bigger counterpart to be modelled efficiently by polygon primitives. But yet on the other hand, the meso-structure is rather large enough so that the geometry detail becomes as obvious to human vision as bumpy surface so that it cannot be modelled easily as micro facet surface [39].

During the last few years, we have seen an interest in displacement mapping techniques for interactive rendering purpose due to its efficiency for handling meso-structure geometry. The surface detail is reserved by adding or subtracting the 2D height field onto macro-structure polygons which act as a base. The main reason for this interest is that most displacement mapping techniques rely on 2D height field to simplify the 3D meso-structure, which provides advantages for optimization.

Generally, displacement mapping techniques retrieve height field data from 2D textures and perform geometry displacement perpendicularly along the normal direction of the polygonal base. Displacement mapping techniques are usually subdivided into two main types. The first type is called vertex-based displacement mapping whereby geometry displacement is performed right on the vertices of the base mesh. The second type is

pixel-based displacement mapping (PDM) technique. Instead of directly modifying the vertices of a mesh, PDM subtly alters the texture coordinate and/or the depth of the pixels rasterized from the polygonal base to provide local surface refinement. It can also be regarded as a visual trick, giving the illusion of depth on the plain surface without expensive actual geometrical deformation.

In the following, a short survey of both per-vertex and per-pixel displacement mapping techniques is provided. However, the thesis will focus on the latter part which is more related to the thesis's visualization method.

2.3.1 Height field Displacement mapping principles

In order to fully understand displacement mapping, it is helpful to get hold of the important mathematics foundation behind the techniques, which builds the relation between displaced geometry and the original geometry. That is, it is important to know how mathematical transformations are employed to maintain the correctness of the displacement mapping result, given the large difference between world coordinate space and local texture coordinate space.

2.3.1.1 2D Height field representation

For convenience, we denote macro, meso-structure, normal and 2D height field by $\vec{p}(u, v)$, $\vec{r}(u, v)$, $\vec{n}(u, v)$ and $h(u, v)$ respectively. Note that while the first three elements (macro, meso-structure, normal) are 3D geometry with three components X, Y, Z; the height field, on the other hand, is a 2D scalar field with a single component, often stored as the alpha channel in a texture map.

2D height field is, in fact, the compressed form of the original 3D geometry from 3D into 2D form along the orthogonal normal vector – see Figure 2.4:

$$\vec{r}(u, v) = \vec{p}(u, v) + \vec{n}(u, v) \times h(u, v) \quad (1)$$

Figure 2.4: 3D decomposition of meso-structure into 2D Height Fields

2.3.1.2 Coordinate transformations

In equation (1), the first three elements, $\vec{p}(u, v)$, $\vec{r}(u, v)$ and $\vec{n}(u, v)$ are expressed in world coordinate space, but the height field exists in its local height field space. Therefore, in per-pixel displacement mapping, all coordinates should be all transformed from the world space to the local space in order to maintain the consistency of displacement.

Figure 2.5: Illustration of the world and local coordinate system

For local coordinate calculation, let $U = (u_1, u_2, u_3)$, $V = (v_1, v_2, v_3)$, $W = (w_1, w_2, w_3)$ are three vectors representing the axes of the local space for some height field and O is the origin of the local coordinate system. All U , V , W , and O are defined in the global coordinate system. As in Figure 2.5, we can easily derive the transformation matrix M as:

$$M = \begin{pmatrix} u_1 & u_2 & u_3 & -U \cdot O \\ v_1 & v_2 & v_3 & -V \cdot O \\ w_1 & w_2 & w_3 & -W \cdot O \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

To convert from global into local coordinates, the inversed matrix form M^{-1} is used.

2.3.2 Per-vertex displacement mapping on the GPU

There are two main types of displacement mapping techniques, which are either vertex shader based methods that modify the vertices of the macro-structure mesh, or the fragment shader based ray-tracing that refines the original texture with or without silhouette handling on geometry accuracy. This section focuses on the vertex based displacement mapping technique.

In Per-vertex displacement mapping on the GPU, the triangle mesh vertices are extruded (or intruded) along the direction of the normal vector of the base surface by sampling the displacement distance from the 2D height field map. The new normal vector might need to be re-derived for accurate lighting due to this new displacement.

The following CG code is written in the vertex shader. Receiving input of the original vertex's position, the displacement is done by fetching the displacement value from computing the height field map (HFMap) in 2D texture map at the given coordinate (uvcoord), computing the new displaced vertex position by adding scaled normal vector (Normal) by the factor of displaced value (HF_SCALE), and finally transforming the new displaced point into screen coordinate space (OutPos) by world view projection matrix (WVProj):

```
float h = tex2D(HFMap, uvcoord).w * HF_SCALE;

float4 nPos = float4(Position + Normal * h, 1);

OutPos = mul(nPos, WVProj);
```

In general, the vertex displacement mapping can produce very highly detailed and accurate deformed silhouette, only if there are enough vertices tessellated from the surface of the model [40]. However, real-time tessellation requires extra cost and is still a very expensive operation. Two reasons are:

- 1) The number of the vertices needed for deformation should be very high to resample the detail from 2D height field map. This is often the bottle neck during real-time visualization.
- 2) Vertex displacement mapping is not optimized for rendering 3D detailed objects that occupy only a small area with a few pixels. (or no Level-of-Detail optimization)

2.3.3 Per-pixel displacement mapping on the GPU

The second type of displacement mapping is Per-pixel Displacement mapping that achieves the displacement/parallax effect in the fragment processing stage instead of vertex processing stage. In this type of displacement mapping, the vertex shader will handle only coordinate transformation, and then the fragment shader will actually calculate ray – height field intersection in a local tangent space.

2.3.3.1 Technical challenges & Classification

In general, the techniques of per-pixel displacement mapping can be categorized into ray tracing or ray-casting algorithm that refines the original base geometry with a higher detail surface through modifying the texture coordinates from the conventional texture mapping in correlation to the viewing point, so as to create the illusion of depth with parallax effect – as illustrated in Figure 2.6

Figure 2.6: Overview of per-pixel displacement mapping technique

After rasterization, each fragment is provided with the local space transformed viewing vector from the vertex shader and the initial interpolated coordinate $P(u_0, v_0)$ that is denoted as the starting point of the ray-casting process. Note that view vector and initial starting coordinate (u_0, v_0) must be all existed in tangent space because the ray-casting can be performed in local space. Then, fragment shader's task is to find the actual intersection coordinate B (instead of point P) between this marching ray with the 2D height field texture – Figure 2.6.

However, there are quite a few challenges in this task:

- 1) The determination of intersection location B often requires a lot of iteration steps (i.e., $X_1, X_2, \dots, X_n$ in Figure 2.6), which will dramatically reduce the overall performance. As a result, lots of research effort has been spent to solve this problem more effectively, such as relief mapping, linear search, sphere search, cone search, interval search, pyramid mapping, parallax mapping, etc. which are described in the next few subsections.

2) Inaccurate ray intersection calculation can easily occur with high frequency sharp slopes surfaces. Under shallow viewing angle where viewing ray is nearly parallel to the height field's PCA plane, it's not easy to detect the "first hit" between the ray and the height field surface with a limited number of sampling points. The sharper the slope of the surface, the denser sampling should be performed to prevent missing cases as illustrated in Figure 2.7

Figure 2.7: An example of inaccurate calculation of Ray - Height Field intersection

3) During ray-casting, there are cases that the ray might not hit the 2D height fields, especially at low level regions where the ray can easily pass towards other neighbouring surfaces. These are the cases that will most likely contribute to the silhouette effects of displacement mapping. Hence, special handling for these cases is very important; otherwise, without the appropriate silhouette from displacement, the surface will look incorrectly flatten.

2.3.3.2 Single step displacement mapping

Pixel based displacement mapping techniques can also be categorized into two smaller categories: single-step or multi step iterative refinement methods. The single-step method

has been proposed initially for per-pixel displacement mapping technique. The displacement is conducted by only a single texture lookup for the height field value for limited computation resource. Hence, it does not add any significant computational cost to the GPU workload, but instead, provides stronger sense of bumpiness onto the surface even though the displacement effect, in general, is not an accurate approximation of the height field deformation. Although not quite related to the proposed method, for wide range of survey's review, several single step displacement mappings are also included: Bump mapping, Non-iterative Parallax mapping and Offset-limiting Parallax mapping.

Bump mapping

Bump mapping or Normal mapping [41] has been a well-known method ever since the early stage of computer graphics research. It can be regarded as the initial concept of all current displacement mapping techniques. The technique has been long known and implemented even before the introduction of programmable GPUs, for example [39, 42-44]. Barry et al [45] have proposed some extension of the technique to account for self-occlusions. They also discussed some optimization of switching between three rendering options (BRDFs material, bump-mapping and displacement mapping rendering) for performance improvement based on either the viewing angle or level of detail of 3D object models.

Non-iterative Parallax mapping

The parallax mapping [46] moves further from traditional normal shading by introducing view-dependent texture refinement of the base surface texture that re-adjusts the texture

coordinates an amount sampled from height field map, thus enhancing the texture's outlook mapped on the polygon - as in Figure 2.6. The final (u, v) texture coordinate is set by displacing the initial texture coordinates (u_0, v_0) interpolated during rasterization stage by a function of the viewing vector $\vec{V}$ in tangent space and the value of the height map sampled at that point, $h(u_0, v_0)$ fetched from the texture map. Below is the CG code:

```
// sample height field value from current coordinate (uvcoord).
float height = tex2D(HFMap, uvcoord).w * HF_SCALE;
// parallax shifting along the viewing direction (ViewDir)
uvcoord = uvcoord + height * ViewDir.xy * (1.0f / ViewDir.z);
```

Figure 2.8: Comparison between Bump mapping (LEFT) and Non-iterative parallax mapping (RIGHT). It is easy to see the Non-iterative parallax mapping method can give better depth deformation than the bump mapping's method. However, the overall result is still not good enough, mostly due to insufficient iterations needed for intersection calculation.

Figure 2.8 is the comparison between bump mapping and parallax mapping techniques. Note that at certain viewing angles, parallax mapping provides more depth of the surface

than bump mapping. However, it is not very difficult to find that non-iterative parallax mapping is still a crude approximation of its original height field. As the viewing direction turns to more shallow angles, inaccurate texture coordinate offset leads to undesirable displacement effect. This shimmering artefact issue gives an even worse look compared to the original texture mapping.

Offset-limiting Parallax mapping

Another version of parallax mapping is offset limiting Parallax mapping that is more practical for general use, especially in Video Games. Nowadays, one simple way to improve the original parallax mapping at shallow angles is to limit the texture coordinate offset (shift) by a fraction so that the amount of texture coordinate being shifted would never be too far away from the original [47]. Offset limiting can be done simply by removing $(1 / Vz)$ term in parallax mapping, which makes the code even “cleaner” than the original parallax mapping and the result is more visibly improved.

```
// sample height field value from current coordinate (uvcoord).  
float height = tex2D(HFMap, uvcoord).w * HF_SCALE;  
  
// parallax shifting along the viewing direction (ViewDir)  
uvcoord = uvcoord + height * ViewDir.xy;
```

However, a few improvement tweaks still cannot fully solve the unwanted effect often seen on the non-iterative displacement mapping at shallow angles area in which the displaced texture appears to float on the surface – as illustrated in Figure 2.9.

Figure 2.9: Comparison between offset-limit parallax mapping (LEFT) and Original parallax mapping (RIGHT). Here, due to reducing “overshoot” problem at low angle fragments, offset-limit parallax mapping shows a visible improvement compared to previous naive approach.

2.3.3.3 Muti-step displacement mapping

Rather than simple texture coordinate adjustment of single step method, on the other hand, multi-step displacement mapping is the iterative method that performs actual geometry displacement by exhaustively searching within the 2D domain of the height field surface for the correct intersection. In other words, this is an iterative approach assuming that for enough number of iterations, the ray will reach the intersection location correctly. This also means that a large number of texture fetching operation and ray-surface calculations are needed at each iteration step, thus performance will be reduced. Nevertheless, from the experiment results in chapter 4, this technique is still considered to be generally much “cheaper” for high detail visualization purposes. The thesis will discuss the performance evaluation in detail in Chapter 4. Below are several techniques that have been proposed in literature:

Iterative parallax mapping

A single step of sampling height field will never be good enough to achieve good quality result [26]. In order to archive better quality of Parallax mapping, a multi-step iterative attempt must be made to exhaustively search for correct intersection between the viewing ray and the related 2D height field.

The texture coordinate shifting method of iterative parallax mapping is, in fact, an iterative form of the previous step limit-offset parallax mapping, except that here, it is necessary to introduce a third dimension of depth, instead of working on 2D texture coordinate space as in the previous parallax mapping. Iterative parallax mapping treats the base polygon as the upper layer of the height field surface with maximum displacement value of 1. The initial ray-casting point's coordinate is set as $(u_0, v_0, 1)$. Next, each step of iteration is, in fact, considered as an offset limit “jump” of the ray-casting point along the viewing ray:

$$P_{i+1} = P_i + Diff(P_i) \times V$$

where $Diff(P_i)$ is the function giving the difference between current Ray-casting point (P_i) 's Z value and actual height field's value. Below is the source code of the algorithm:

```
// setting initial position (P) at upper layer (height = 1)
float3 P = float3 (uvcoord, 1.0f);
// raycasting loop with ITERNUM iterations
for (int iter = 0; iter < ITERNUM; iter++)
{
    // sample height field value from current coordinate (P.xy).
    float height = tex2D(HFMap, P.xy).w * HF_SCALE;
```



```
// move P along the view direction (ViewDir)  
  
P = P + (height - P.z) * ViewDir;  
  
}
```

Note that iterative parallax mapping is not a robust algorithm due to the fact that there is no guarantee of first-hit intersection, see an example shown in Figure 2.7. It is more likely a heuristic search trying its best to find the closest point to the height field. There is no guaranty that the closest point would be the first intersection point. Figure 2.10 shows the result comparison of Non-iterative parallax mapping and Iterative Parallax mapping.

Figure 2.10: Comparison between Non-iterative parallax mapping (LEFT) and Iterative Parallax mapping (10 iterations) (RIGHT). The displacement quality has been improved strongly with multi-step iterative methods, especially at the “shallow” angle area.

Linear search displacement mapping

Linear search is another type of iterative displacement mapping search. By performing incremental and regular interval samplings along the marching ray, the algorithm finds two closest points along the marching ray that contain the intersection point – one just above and one just below the height field surface. This is in fact the technique that has been commonly employed in volume rendering area [23].

Given the number of iterations is $ITER_NUM$, for every single step, the $CurHeight$ of point P being ray-casted is decreased by $\frac{1}{ITER_NUM}$. That means the ray-casting point will shift towards the height field with $\frac{1}{ITER_NUM} \times \vec{V}$ where $\vec{V}$ is the viewing vector in local height field space – as in Figure 2.11.

Figure 2.11: Illustration of Linear search for displacement mapping

At every step along the ray, the height field value is sampled and compared to the current ray location P_i . If the height field surface is just below P_i at some step, then an intersection must lie somewhere between the previous and the current sampled

positions P_{i-1} and P_i , and thus, the search can be terminated. The CG code of the linear search is as below:

```
float CurHeight = 1.0f; // setting initial height (at upper layer)
float ITER_STEP = 1.0f / ITERATION_NUM; // linear search interval
float HMin; // stores minimum height above the height field
// raycasting loop with ITERNUM iterations
for (int iter = 0; iter < ITERATION_NUM; i++)
{
    // move CurHeight one level down
    CurHeight = CurHeight - ITER_STEP;

    // update new UV coordinate (uvcoord) with new height (CurHeight)
    uvcoord = UV1 * (1 - CurHeight) + UV0 * CurHeight;

    // sample height field value from current coordinate.
    sampledHeight = tex2D(HFMap, uvcoord).w;

    // update ray-casting result: minimal height
    If (CurHeight >= sampledHeight)
    {
        HMin = CurHeight;
    }
}

//update UV coordinate with final ray-casting result
uvcoord = UV1 * (1 - HMin) + UV0 * HMin;
```

This method is different from the previous algorithm for it incrementally updates the most recent location P which is most likely to be the closest point to the intersection. Hence, if

the sampling interval is small enough compared to the height field surface detail, the intersection is always correct.

However, linear search alone often results in “stair” artefacts at shallow angles (Figure 2.12) unless the iteration interval is small enough. Brawley [48] has suggested that more iterations should be added when viewing the surface at “shallow” angles while fewer iterations should be enough for orthogonal view for optimization purpose.

Figure 2.12: Comparison between Linear search (LEFT) – 12 iterations and parallax mapping (RIGHT) – 12 iterations. Note the “staircase” aliasing artefact caused with insufficient number of iteration shown on linear search scheme.

Binary search displacement mapping

Instead of incrementally searching from the entering point to ending point, binary search scheme recursively refines the whole searching range between the entering point and the

ending point of the height field until the gap narrows down into some threshold. Figure 2.13 illustrates the binary search method for ray and height field intersection calculation. The figure on top is a good case of the method whose intersection can be found only after a few steps, while the bottom figure is the bad case when binary search give false intersection due to the high slope surface. In each iteration, the gap will be narrowed down into half [9, 49] by discarding the other half which is either above or below the height field surface. This check is done simply by sampling the depth value and comparing with its current searching point. It is then to keep the half range left that contains the intersection point and continue the iteration until the range becomes sufficiently small.

Figure 2.13 TOP: illustration of binary search scheme with 3 iteration steps. BOTTOM: A case where binary search gives incorrect intersection.

In the below segment of code, the texture coordinates of the starting point of the searching range are denoted by *UV0*; *Heightmin* and *Heightmax* are the height values at two ends of the searching range; *HalfHeight* and *sampledHeight* are the height values at middle point of the searching range and the actual height field's level. The code is provided below:

```

float Heightmin = 0, Heightmax = 1;

for (int iter = 0; iter < ITER_NUM; iter++)
{
    float HalfHeight = (Heightmin + Heightmax) / 2;

    uvcoord = UV1 * (1 - HalfHeight) + UV0 * HalfHeight;

    sampledHeight = tex2D(HFMap, uvcoord).w;

    if (HalfHeight <= sampledHeight) Heightmin = HalfHeight ;

    else Heightmax = HalfHeight;
}

```

Generally, binary search can produce good convergence rate compared to other searching methods. (In the worst case, when the intersection point is at one end of the search range, the binary search must continue the maximum number of iteration steps; and this will have at most $\lfloor \log_2(N) + 1 \rfloor$ iterations, where N is the number of quantization steps of height field – usually $N = 256$. Hence usually 9 to 10 iteration steps would be sufficient for binary search). However, the algorithm will not be able to give warranty that the intersection is the first intersection point as the case being illustrated in Figure 2.13. Figure 2.14 gives result comparison of binary search and parallax mapping.

Figure 2.14: Comparison between binary search (LEFT) and parallax mapping (RIGHT). Binary search provides faster convergence rate of ray-surface intersection especially at the steep slope region (white arrow). However, it is more subjected to false-hit when the tracing ray may have to pass through many different “Hill” steep slopes of the height field surface (white arrow).

Interval displacement mapping

The binary search algorithm mentioned in the previous section provides a simple but efficient search by recursively subdividing the searching range into two halves and then keeping only the one that potentially contains the intersection point while discarding the other half. However, this searching scheme has one limitation that it does not really utilize all the underlying geometrical information – i.e., the slope of height field. The secant displacement methods [50, 51] make assumption that the height field at current tracing point is planar and the algorithm will project this local planar against the searching ray to perform a more closely self-pruning search. That also means more

calculations that are needed for plane – ray intersection at each iteration – see the code provided below.

Figure 2.15 Illustration of secant culling method of Interval displacement mapping

Similar to the binary search, texture coordinates of the start and ending points of the searching range are denoted by UV0 & UV1 (as in the following code); Height_2_a and Height_2_b are the heights at points A and B; Delta_a, Delta_b are the projection distances of current ending points A and B to the height field – Figure 2.15. The code is provided below:

```
// ray - casting loop with ITERNUM iterations
for (int iter = 0; iter < ITER_NUM; iter++)
{
    //perform secant intersection with Viewing ray

    float NewHeight = Height_2_b + (Height_2_a - Height_2_b) / (Delta_a -
Delta_b) * Delta_a;

    //update new P with NewHeight

    P = UV0 * NewHeight + UV1 * (1 - NewHeight);

    //compare new position (P) with height field

    sign = P.z - tex2D(HFMap, P.xy).w;
```



```
//refine searching range
```

```
if ( sign < 0 ) Delta_b = sign; Height_2_b = NewHeight;
```

```
else Delta_a = sign; Height_2_a = NewHeight;}
```

Figure 2.16 TOP: Binary search displacement mapping (2 iteration + 15 linear steps). BOTTOM: Secant Interval displacement mapping (1 iteration + 15 linear steps). The Secant Interval displacement mapping in general gives smoother interpolation than previous Binary search. However, it requires more calculation for ray-surface calculation so it is not easy to verify which method gives better result.

3D Sphere Tracing

3D Sphere tracing was first suggested by [52] for ray-casting on algebraic surfaces and later was applied to 2D height field by [53]. The algorithm employs a 3D distance map denoted as the guide map to help search for the first intersection. Note that the distance map must be precomputed because it often takes very long time to construct such 3D distance map over the height field. However, the advantage of this algorithm is that, Sphere tracing is a conservative search such that the first intersection is always guaranteed with enough iteration steps. Figure 2.17 gives result comparison of Linear search and Sphere tracing search.

Basically, every voxel inside a 3D distance map stores a radius that encodes an empty spherical space around it. In that way, if moving along the viewing ray point using only that radius information as the interval, consequently the tracing ray will hit the 2D height field after a few iterations – Figure 2.18. Because of the simplicity, the rendering algorithm is very simple and also the fastest method for high performance ray tracing currently. Below is the implementation of the algorithm:

```
P = float3(uvcoord, 1.0f); // setting initial position

// ray - casting loop with ITERNUM iterations
for (int iter = 0; iter < ITER_NUM; iter++)
{
    //ray-tracing within 3D distant map
    P = P + tex3D(SphereMap, P) * ViewDir;
}
```

However, one obvious limitation of this algorithm is the huge amount of memory required to store a three dimensional texture for the 3D distance map. In general, this

method is not usually preferred in practice due to the limitation of video memory in most graphic cards.

Figure 2.17: Comparison between Linear search – 12 iterations (LEFT) and Sphere tracing – 12 iterations (RIGHT). Notice the higher accuracy of the algorithm with less artefact visualization of the algorithm.

Figure 2.18: Illustration of sphere tracing algorithm. Image taken from [53]

Cone step displacement mapping

Cone step displacement mapping and 3D sphere tracing both fall into the same category of pre-computed displacement mapping techniques. However, in some cases, Cone step mapping has more advantages than Sphere tracing's. While spherical method employs 3D texture to encapsulate all the possible spherical empty spaces that exists within the gaps of the height fields, cone step [4] encodes empty these spaces using conical geometry which turns out only to need a single 2D (instead of 3D volume texture) layer to store all the cone information – Figure 2.19. This opens a great deal of simplification in terms of texture memory consumption (from 3D texture down to 2D texture). Another potential advantage of cone step mapping is the flexibility of using cone primitive for describing the empty space, especially with high frequency surfaces.

Figure 2.19: Illustration of Cone step tracing algorithm. Image taken from [4]

However, compared to sphere tracing, cone step mapping requires more calculation at each iteration as ray-cone intersection calculation is not as simple as ray-sphere

intersection. As a result, in some cases, cone stepping displacement mapping might have lower convergence rate than sphere distance mapping technique. However, the texture optimization is still a great advantage of this technique. The below fragment shader code implements the cone step algorithm:

```
P = float3(uvcoord, 1.0f); // setting initial position
Raylength = length(ViewDir.xy);
// ray - casting loop with ITERNUM iterations
for (int iter = 0; iter < ITER_NUM; iter++)
{
    float height = tex2D(ConeMap, P.xy).y;
    // perform ray-cone intersection
    dts = height / (Raylength / ViewDir.xy + height * ViewDir.z);
    RayShift = dts * tex2D(ConeMap, P.xy).x - P.z;
    // update new position
    P = P + ViewDir * RayShift;
}
```

Figure 2.20: Comparison between Relief mapping (Linear (12 iterations) and binary search (5 iterations) - LEFT) and Cone step mapping – 12 iterations (RIGHT)

2.4 Summary

This chapter has described many related works about 3D structure representations using polygons, point-clouds, octree and 2D height fields, ray-casting algorithm for 3D surface visualization and several common displacement techniques that have been developed recently. Each topic discussed is more or less related to the thesis proposal for fast visualization of high detail 3D models.

CHAPTER 3

NEW VISUALIZATION ALGORITHM*

This chapter presents a new algorithm for fast visualization of high detail 3D models. It can achieve interactive rendering speed of 3D models with hundred millions of vertices while still maintaining high quality results comparable to most recent point-based high quality rendering methods.

3.1 Overview

Efficient visualizing high-detail 3D models in real-time is an important and also challenging problem. Here we present a new visualization algorithm designed to achieve real-time performance especially for very high resolution 3D model acquired from range images. Our basic idea is to decompose the complex 3D model's surface into several 2D local height fields and then ray-cast them efficiently and accurately in real-time.

* The algorithm presented in this chapter has been published in *Proceedings of Graphics Interface 2009*, British Columbia, Canada.

Thus our algorithm consists of two main parts - surface decomposition and real-time rendering. The following is a brief description of these two parts whose details are provided in the subsequent sections.

3.1.1 Surface Decomposition

The Octree partition scheme is used to decompose the input 3D model's surface into a finite set of 2D height fields (see Figure 3.2.b). The number of height fields decomposed depends on the model's geometric complexity rather than its vertex density.

Next, each height field is converted into a 2D height map (or displacement map) and packed into hardware memory in the form of a big 2D texture atlas (see Figure 3.2.c)

Finally, the faces of the octree voxel that bounds the height field are appropriately assigned texture coordinates. These faces are the polygons that are being displacement-mapped, and will be rasterized into several fragments during rendering stage (see Figure 3.2.d). Each of these fragments will later be ray-casted to refine the original facets into a high detail surface. Figure 3.1 shows a height-field decomposition of the 3D sphere and Stanford Bunny model, together with the bounding boxes.

(a) HF decomposition of 3D Sphere. LEFT: Input 3D sphere. MIDDLE: height field decomposition of the model. RIGHT: height field surface of one octree cell.

(b) HF decomposition of Bunny. The different colors of the cells indicate the different of the domain planes of the height fields.

Figure 3.1: Decomposition of 3D models into 2D height field surfaces.

3.1.2 Rendering

Now the 3D model is represented by a set of 2D height fields. To render them, these bounding boxes of the height fields are projected and rasterized to produce necessary fragments, where each fragment provides the initial coordinates from which the viewing ray starts marching into the height field domain for displacement refinement (see Figure 3.2.d).

For each fragment, the viewing ray is transformed from the world coordinate frame to the local coordinate frame of the corresponding 2D height map. Finally, the ray-casting algorithm is performed to find the intersection between the ray and the height field surface with correct height field boundary culling (see Figure 3.2.e and Figure 3.2.f). All the steps are illustrated in Figure 3.2.

Figure 3.2: Illustration of several steps of the algorithm. From LEFT to RIGHT: (a) Input model, (b) Octree decomposition, (c) 2D height maps, (d) Voxel rasterization, (e) Ray-casting without height field boundary culling and (f) Final rendering.

3.2 Height Field Decomposition

Given an input 3D model, the main purpose of this step is to subdivide the original complex 3D surface into many simpler 2D height field surfaces. Height field surfaces allow simpler and more efficient ray-surface intersection calculation. Finding suitable height field domain planes such that the number of height fields is minimal is not a trivial problem. Here, we propose a method, in which the Octree is employed to partition the model surface and the PCA is used to test and compute a suitable height field domain plane for each surface patch.

3.2.1 Octree Decomposition

The method subdivides the initial bounding cube of the 3D model into smaller cells such that each cell must contain either a 2D function surface (a height field surface) or just empty space (void cells). For those cells that intersect the 3D model's surface, if the surface being contained inside that cell is not a height field surface, it is further subdivided into smaller cells.

Besides its simplicity, octree gives us the following advantage. The octree partitioning scheme subdivides volume into eight equal orthogonal and non-overlapping bounding cubes. This regular subdivision allows simple but robust ray-casting of the 2D height field, which enables the algorithm to perform efficient and correct visualization of high detail 3D geometry – see Section 3.5 for more details.

Note that this step does not like other iterative displacement mapping techniques that use triangular prisms as bounding volumes [5]. The triangular prisms are formed by extruding the base triangle mesh in the direction of the vertex normals. As of now, there is still no accurate and efficient algorithm that can correctly render displacement maps using a base

triangle mesh. Porumbescu et al. [5] have proposed to subdivide the extrusion prisms into tetrahedral to provide a piece-wise linear mapping from "shell space" to height field "texture space". However, these mappings lead to objectionable artefacts at the subdivision boundary, similar to the artefacts induced by non-perspective-correct texture mapping [10]. The reason is that the straight ray path in the prism space becomes a non-linear curve in the height field texture space [10]. Jeschke et al. [10] proposed curved shell mapping that is accurate and tangent-continuous at prism borders. However, curved shell mapping is slow due to its complexity and is not suitable for real-time visualization. Chen and Chang [11] attempted to address the tangent discontinuities along the viewing ray by employing an extra tangent-space map to bend the viewing ray in the texture space. However, the result is not accurate with noticeable inconsistent displacement from different viewing angles (these artefacts are visible in their demo video, and have been acknowledged by one of the authors). The generalized displacement mapping method [6] is able to give correct intersection between ray and height field surface in general cases because ray marching distances have been precomputed and are retrieved during rendering.

3.2.2 Height Field Test

For each octree cell, it is important to determine whether the enclosed surface is a height field surface. If it is not, then the octree cell is subdivided. Otherwise a height field domain plane can be computed. Here, we have employed the PCA transformation [54] for the height field condition test. First, we apply the PCA to all the points or vertices in the octree cell to find the best-fit plane. Next, using the plane's normal vector as the projection direction, it can be determined whether any of the points or vertices in the cell has its normal vector backfacing the projection plane (assuming all the points are on one

side of the projection plane). If there is, then this is not yet a height field surface and the cell needs to be subdivided. Otherwise, the best-fit plane is used as the height field domain plane for the cell. Figure 3.3 gives a 2D illustration of the PCA technique applied to find the height field domain plane. Given an arbitrary surface, the PCA technique identifies the domain plane of height field by rotating the axes such that the maximum variability is maximized.

Figure 3.3: PCA transformation applied on 2D surface for height field construction.

The PCA height field condition test may fail even when the surface is already a height field surface. This can happen when many points are distributed on a few "tall mountains" on the surface. For this case, it may actually be better to let the test fail and subdivide the surface further. This is because "tall-mountain" surfaces result in abrupt changes in the height maps, which are more expensive to intersect accurately during image-space ray-casting – as in Figure 2.7.

3.3 Setting up Displacement Mapping

After the height field decomposition, it is ready to generate the height maps (displacement maps) and to find the transformation from the world space to the local height field texture space, so that it is possible to set up the texture coordinates of the bounding boxes, and transform the viewing ray from the world space to the texture space during rendering step.

3.3.1 Height Map Generation

Since a height map is a discrete 2D image that records the scalar distance between each point on the height field surface and the domain plane, for each octree cell, if the input model is made up of triangular meshes, the algorithm simply orthographically projects the meshes onto the frame buffer along the domain plane normal vector direction, with the camera placed at the cell centre. The content in the depth buffer is used as the height map for the cell. If the input model is made up of points, then the efficient pull-push interpolation method [17] is employed to reconstruct the height map.

The height maps are tightly packed into texture atlas (see Figure 3.4). The image resolution allocated to each height map is half (in both dimensions) of that in its parent's level. Therefore, by counting the number of octree leaf nodes at different levels and with the given total available texture image size; it is possible compute the image resolution that can be allocated to each height map. In the experiment, the resolution is 4096 x 4096 for each atlas textures. The resolution for individual height field is determined heuristically by the factor of 2 so that it is convenient for texture tiling step. For example, at the lowest level, the height field resolution is 16x16 which is sufficient for detail preserving. Figure 3.4 is the texture atlas for Bunny 3D model. The atlas consists of several height field maps being decomposed at three different levels of the octree. For

atlas packing, each height field maps of the same decomposition level sharing the same dimension are first packed into groups, and then they finally are regrouped again into a big texture atlas.

3.3.2 Coordinates Transformation

Ray-casting stage is performed in the texture coordinate space. Hence, it is necessary to transform each viewing ray from the world space into the texture coordinates space. The transformation in fact consists of two successive transformations. The first is a transformation from the world coordinate space to the height field coordinate space of each Octree cell that is determined during height field decomposition. Figure 3.5 illustrates the height field space of Bunny after decomposition.

Figure 3.5: Illustration of local space for each height field. Hard lines stand for Normal Vector

The second transformation is one from the height field space to the texture coordinate space. Figure 3.6 illustrates these transformations. Let M be the combined transformation in matrix form:

$$M = M_2 \cdot M_1$$

where

$$M_1 = \begin{pmatrix} u_1 & u_2 & u_3 & -U \cdot O \\ v_1 & v_2 & v_3 & -V \cdot O \\ w_1 & w_2 & w_3 & -W \cdot O \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad M_2 = \begin{pmatrix} s_1 & 0 & 0 & t_1 \\ 0 & s_2 & 0 & t_2 \\ 0 & 0 & s_3 & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

In M_1 , the vectors $U = (u_1, u_2, u_3)^t$, $V = (v_1, v_2, v_3)^t$, and $W = (w_1, w_2, w_3)^t$ are the orthogonal axes of the height field coordinate frame, and $O = (o_1, o_2, o_3)^t$ is the coordinates of its origin. The matrix M_2 maps the height maps into the atlas' texture size and location by scaling vector $S = (s_1, s_2, s_3)^t$ and translating vector $T = (t_1, t_2, t_3)^t$. s_3 & t_3 are responsible for mapping height value from the height field space into the range $(0, 1)$ of texture space. s_1, t_1, s_2 & t_2 are used to scale and translate the height maps into texture atlas.

Figure 3.6: Transformation of coordinates. During ray-casting, each viewing ray is transformed from the world coordinate space into the texture coordinate space using the transformation matrix M .

In the preprocessing stage, each vertex of a bounding box is assigned texture coordinates. The texture coordinates are computed by multiplying the matrix M_1 to the world

coordinates of the vertex. During the rendering stage, these texture coordinates will be interpolated at each fragment produced by the rasterization of the bounding box. The interpolated texture coordinates provide the initial position of the viewing ray in the height field coordinate frame. The viewing ray direction vector is transformed by the top-left 3×3 sub-matrix of M_1 . Each query of the height map is transformed into the texture space by M_2 .

3.4 Rendering

The rendering algorithm has two steps. The first step starts with the rasterization of the bounding boxes. Note that not all six faces of octree will be rasterized. Most current hardware support back face culling which can cull-off those polygons facing backward to the viewing point. Fortunately, there is no conflict of backface culling option with the current algorithm because those faces of the bounding box that facing backward will not contribute any additional fragment during rasterization stage, hence, no effect on ray-casting result as well. As a result, with backface culling applied, at most three faces of each bounding box need to be rasterized with interpolated coordinate. The interpolated texture coordinates provide the initial position of the viewing ray in the height field coordinate frame (X_0 in Figure 3.7).

(a) Case 1: Ray hits surface

(b) Case 2: Ray passes over

Figure 3.7: Illustration of two cases of ray-casting of height fields

In the next step, for each fragment, the viewing ray direction is transformed from the world space into the height field space using the precomputed matrix M in Section 3.2.2. From the initial position of the viewing ray, iterative parallax search was employed to find the first intersection between the ray and the height map. This step is illustrated in Figure 3.7.

Basically, the height field ray-casting method computes the intersection by marching along the ray, from the initial location, to find the position that has the closest distance above the height map. In fact, many algorithms have been proposed for 2D height field ray-casting [7, 53, 55-57]. Szirmay-Kalos and Umenhoffer [58] have presented a detailed

analysis of the different existing techniques. However, the algorithm has adapted the iterative parallax mapping algorithm [8] for two reasons. Firstly, unlike many other methods [4, 6, 53], the iterative parallax mapping algorithm does not need to pre-compute extra information. This makes it memory efficient. Secondly, iterative parallax mapping produces less aliasing artefacts compared to other methods such as the relief mapping [58]. Usually iterative parallax search will reach converging point after a few iterations. Eight to twelve iteration steps would be enough for ray-casting search in the experiment. Figure 3.8 and Figure 3.9 show the visualization of the ray-casting progress of the algorithm over several iteration steps.

Figure 3.8

(a)

Figure 3.9

Figure 3.10

(b)

a) 0 Iteration

a) 1 Iteration

a) 2 Iterations

a) 4 Iterations

a) 6 Iterations

a) 8 Iterations

a) 10 Iterations

a) 12 Iterations

Figure 3.8: Illustration of Iterative Parallax ray Casting result over different steps. From TOP to BOTTOM:
0 – 1 – 2 – 4 – 6 – 8 – 10 – 12 Iterations result

Figure 3.9: Difference between 8 and 12 iterations. The coordinate difference happens mostly around “shallow” angle areas which show a good result of ray-casting algorithm.

3.4.1 Boundary Culling

After an intersection on the height map has been found, it is next to perform a test, called the boundary test, to ensure that the ray indeed intersects the current height map and not a neighbouring height map in the texture atlas. This may happen when the viewing ray is at a shallow angle and passes over the current height map (see Figure 3.7.b). If the intersection is in the wrong height map, the corresponding fragment is discarded. The boundary test tells whether the ray intersects the height field surface within the bounding boxes. If it does not, the fragment is discarded so that it will not occlude the correct fragment value when the ray intersects height fields inside bounding boxes that are further away from the viewpoint. The boundary test can be easily implemented since we have already known the limits of each height map in the texture atlas.

Note that each pixel of the image represents a viewing ray, and it may intersect multiple bounding boxes, and thus multiple height fields. Z-buffering is used to keep only the front most height field fragment intersected by the ray.

Figure 3.10: Illustration of the difference between this new algorithm and previous displacement mapping without silhouette handling. TOP: Iterative parallax mapping without Height field boundary culling. BOTTOM: Iterative parallax mapping with Height field boundary culling

3.5 Advantage of the new Algorithm

The main advantage of the new algorithm is its efficient image-space ray-casting on 2D height fields, and thus, its overall performance has no relation to the original number of vertices or points. The time complexity of the rendering algorithm is $O(F + B)$ where F is the total number of fragments produced by the rasterization and B is the total number of bounding boxes processed by Vertex Shader. In the experiments, even for a very huge number of input vertices, the value of B is typically less than 100K. In modern graphics processors, this number of boxes can be processed quickly by the vertex processing units at high frame rates. The performance of the rendering algorithm is thus very unlikely to be geometry-bound.

Figure 3.11: Illustration of the advantage of this new algorithm at dealing with silhouette effect (when the marching ray crossing over several local height field boundaries).

Another advantage of the method is its simplicity in dealing with silhouettes correctly. In the top diagram of Figure 3.11, two polygons have been mapped with two displacement maps, just like polygons being assigned texture maps. A viewing ray intersects the left

polygon, but it does not intersect the left displacement map. Instead it should intersect the displacement map of the right polygon. However, the right polygon is not intersected by the viewing ray and thus cannot provide a fragment for us to trace the same viewing ray to intersect its displacement map. This will result in incorrect silhouettes. (Note that when a ray enters one local height field, it cannot continue to march into another height field as this second height field texture map might not be adjacent to the first in texture atlas)

In this new approach, the displacement map is not explicitly mapped to any polygon. This means there is no actual one-to-one mapping for each polygon and 2D height field. Instead, each displacement map is enclosed in a bounding box that comes from an Octree voxel. The rasterization of the bounding box will provide the fragment the method needs to trace any viewing ray to the enclosed displacement map. This is illustrated in the bottom diagram of Figure 3.11. Note that a viewing ray may intersect many displacement maps, but this is automatically resolved by the z-buffer.

CHAPTER 4

EXPERIMENTAL RESULTS AND COMPARISON

This chapter presents some experiments we did with the new algorithm. In particular, we look at:

- Ray-casting tests with different numbers of iteration steps and the experiment with performance cost on every rendering stage of the algorithm.
- Performance experiment on several Stanford scanning 3D models (David, Lucy, St. Mathew models).
- Performance and quality comparison with other methods (Qsplat [3] and Efficient Point-Based Rendering Using Image Reconstruction [17])

4.1 Algorithm tests

In this section, there are two tests. One is for quality ray-casting test with different numbers of iteration steps and the second test is for performance measuring of the algorithm.

4.1.1 *Quality test*

The number of iteration steps performed during ray-casting search mainly controls the rendering quality of this algorithm. However, if there were too many steps performed, the overall speed would be reduced rapidly in the other way. Fortunately, in our experiments a modest number of steps are required in order to produce reasonably high quality results of rendering high-detail 3D models. In particular, only 4 iteration steps are needed for Lucy model to provide high visualization quality with almost no visual difference compared with 8 iteration steps – as in Figure 4.1. This also shows a very good convergence rate of the algorithm's ray-casting. However, for precaution, 12 iterations are performed for quality comparison with other methods in later section.

1 iteration

4 iterations

8 iterations

Figure 4.1: Iterative ray-casting results with different number of iterations

4.1.2 Performance test

In this experiment, the algorithm is tested on an Intel Pentium 4 505 / 2.66 GHz processor, 4Gbs RAM, Nvidia 8800 GT graphic card 512 MB – PCI express. 3D Model Lucy is selected for having a moderate number of points. There are 41K of height fields and 91M of texels generated for the input of 14M points. The statistics regarding height field decomposition of other 3D models is provided in Table 1. There are 4 performance tests on different rendering steps of the algorithm. The performance is recorded as frames per second (FPS). The higher FPS, the better the algorithm’s performance is.

Chart 1: Performance test of the algorithm

From the result, we can see that iterative Ray-casting is the most expensive step that caused the sharp fall over 4 steps on the chart. This is expected since Ray-casting step involves several texture fetching on for 2D height field and many ray-surface intersection calculations.

4.2 Experiment on several Stanford scanning 3D models

The performance test is done on several models taken from Stanford repository, which are summarized in Table 1. The rendered image is 1024x1024 in size. Resulting images from the rendering of these models are shown in Figure 4.3, with close-up views for quality justification. It is evident from these images that the objects' silhouettes have been properly rendered. To achieve accurate results, 12 iterations in the height field ray-casting are sufficient to produce images with no noticeable errors.

It is worth pointing out that even though the Thai Statue model has only 5 million vertices; its geometric complexity is actually quite high, which results in a high number of octree subdivisions. As a result, there are 64 million texels needed to represent the height maps. This results in a lot of data redundancy since the original model has only 5 million vertices.

On the other hand, the St. Mathew model is an ideal case for the algorithm where the overall surface is relatively smooth and simple, but covered with fine surface details. This leads to fewer octree subdivisions because many large surface regions satisfy the height field condition. A smaller number of bounding boxes results in fewer pixels overdraws and thus speeds up the rendering.

Figure 4.2: The St. Mathew model (368M) with its height field decomposition cells (57K). The different colors of the cells indicate the different orientations of the domain planes of the height fields.

Figure 4.3: Sample images produced by the rendering algorithm. From left to right: St. Mathew (368M), David (28M), Lucy (14M), and Thai Statue (5M)

Model	Vertices	Cells	Texels	Preprocessing Time (mins)	Frames Per Second		
					(1)	(2)	(3)
St. Mathew	368M	57K	101M	20.7	147	18	12
David	28M	31K	96M	0.7	300	32	23
Lucy	14M	41K	91M	1.4	162	28	17
Thai Statue	5M	74K	64M	1.2	170	21	9

Table 1: Rendering results (12 iterations). The framerates are for (1) NVIDIA GeForce 8800 GT, (2) NVIDIA GeForce 8500 GT, and (3) NVIDIA GeForce 7900.

4.3 Performance and quality comparison test with other methods

For comparison, two recent methods have been chosen based on popularity. They are Qsplat [3] and Efficient Point-Based Rendering Using Image Reconstruction [17].

4.3.1 Comparison with Qsplat

In comparison with Qsplat, Lucy 3D model with 14 millions of points was tested. As expected, the new proposed method out-performs Qsplat in rendering speed (Qsplat rendered Lucy with 0.41 FPS compared to 17 FPS from the new method – Table 2, second row). The improvement has shown even sharper for models with more high-detail, for example the model of St. Mathew statue consisting of 368M points. Table 2 is the summary of the comparison.

Model name	New method (FPS)	Qsplat (FPS)	Improvement Rate
Thai statue (5M)	9	1.16	7.76
Lucy (14M)	17	0.41	41.46
David (28M)	23	0.2	115
St.Mathew (368M)	12	0.0574	209.06

Table 2: Performance comparison between new method and with Qsplat.

From the close-up view of Lucy in Figure 4.4, we can see that the new proposed method has produced a smoother and finer silhouette than Qsplat. The reason is mainly because the new proposed method takes advantage of automated texture linear interpolation (which is well supported on most modern GPUs) to provide higher quality with surface interpolation. Yet, Qsplat used point sprite from OpenGL extension to display directly the point-cloud without proper surface interpolation between neighbouring points [17]. Consequently, Qsplat often results in bad quality around silhouette areas or areas where there exists a lot of noises.

Our method can also be flexible at memory consumption. Depending on the level of detail, the model's detail can be expanded or reduced easily with 2D texture resize operation. However, this is not the case in Qsplat, where each level of detail must be pre-computed in a fixed memory size. In terms of memory size optimization, our method is more compressed (PCA compressed) than Qsplat. For example, for each point, Qsplat may need 3 floating point values for the position, 3 floating point values for the normal, plus another 2 more values for the shape of the point. For our method, in theory it only needs 1 pixel = 4 bytes to store the depth and the normal data of the point.

Note that Qsplat used hierarchical structure for fast speed visualization. It is expected that our algorithm can perform better if we also use hierarchical structure. However, due to time constraint, we chose octree for simple and robust solution.

Figure 4.4: Closed-up of Lucy 3D model. TOP: Result of Qsplat's rendering method. BOTTOM: result of new rendering method. Notice the smoother surface rendered the new proposed method.

4.3.2 Comparison with Efficient Point-Based Rendering

In this experiment, we compare the new method with efficient Point-based rendering method, which was proposed by Marroquim [17]. For the Thai statue model, Marroquim’s method out-performs the proposed method in rendering speed with 12 FPS vs. 9 FPS. The reason is that the Thai statue model is a simplified 3D model consisting of only 5 millions of points. Hence, in this case, point-based methods may have more advantage over image based ray-casting method. However, when the number of vertices increases, the proposed method wins as being illustrated in Table 3.

Model name	New method (FPS)	Marroquim's Method(FPS)	Improvement Ratio
Thai statue (5M)	9	12	N/A
Lucy (14M)	17	2.33	4.25
David (28M)	23	2.6	8.846
St.Mathew (368M)	12	0.16	75

Table 3: Performance comparison between new method and Marroquim's method.

As in closed-up view of Thai statue model in Figure 4.5, the proposed method has shown not much visual difference with Marroquim’s method. Marroquim’s method employs a “Box filtering” method on neighbouring points, which is also a linear interpolation filtering type. Hence, both methods provide almost the same high quality rendering results.

Figure 4.5: A Close-up view Thai statue 3D model for Quality comparison between Efficient Point-Based Rendering using Image Reconstruction (TOP) and new proposed method (BOTTOM). The quality is almost identical.

CHAPTER 5

CONCLUSION AND FUTURE WORK

The thesis has presented a new approach for fast visualization of high detail 3D models. The solution consists of two steps. First, 3D models are decomposed quickly into 2D height fields using octree decomposition and second, the height fields are rendered using an image-space ray-casting algorithm. Compared to the existing point-based rendering methods, the proposed algorithm offers significant performance improvement by avoiding the vertex-processing bottleneck. Moreover, the way it decomposes the 3D model into height fields, together with the improved ray-casting algorithm, has resulted in a simple displacement mapping approach that can render objects' silhouettes correctly and efficiently.

The experimental results from Chapter 4 have shown that the algorithm works very well on high-detail 3D models. In fact, contrast to point-based rendering methods, the proposed algorithm performance is not constrained by the number of input points but rather the number of height fields decomposed. The best result of the algorithm compared

to two selected point-based rendering methods [15] is no doubt the St. Mathew 3D model with more than 300 million of points with almost 75 times faster measured on Intel Pentium 4 505/ 2.66 Ghz processor with 4 GBs Ram on 7900 Nvidia, while the rendering quality is shown to be non-significantly different.

However, there are still some limitations from the proposed algorithm that require future research:

- 1) It has been shown that the algorithm works well for surfaces with high-frequency features, but it is not efficient for flat and smooth surfaces. Thus, an interesting approach might be a combination of triangle-based rendering and per-pixel displacement mapping.
- 2) The amount of details that we can store from the input model is limited by the amount of texture memory. A compression method for height field detail would also be interesting. However, for iterative ray-casting purpose, decompression cost must be reasonable due to the fact that ray-casting search may require several sampling iterations over the height field map.
- 3) The new approach has not used mipmapping for the height maps, which may help produce aliasing artifacts when viewing the height fields from a distance. However, we cannot use mipmapping in a straightforward way because the texture atlas is made up of many different individual height maps with non-regulated texture coordinate referencing.
- 4) Another possible extension to the algorithm is to make it suitable for dynamic or deformable surfaces. This requires selective update of the height field decomposition of the affected surface regions. Accordingly, the preprocessing steps, such as height map generation, must be highly optimized to provide the

desired speed. Furthermore, the preprocessing algorithm is currently not out-of-core. An out-of-core algorithm is necessary when the input model is too large to fit into the system memory.

REFERENCES

1. Levoy, M., et al., *The Digital Michelangelo Project: 3D Scanning of Large Statues*. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 2000: p. 131--144.
2. Heckbert, P.S. and M. Garland, *Survey of polygonal surface simplification algorithms*. 1997, CARNEGIE-MELLON UNIV PITTSBURGH PA SCHOOL OF COMPUTER SCIENCE.
3. Rusinkiewicz, S. and M. Levoy, *Qsplat: a multiresolution point rendering system for large meshes*. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 2000: p. 343--352.
4. Dummer, J., *Cone Step Mapping: An Iterative Ray-Heightfield Intersection Algorithm*. 2006: <http://www.lonesock.net/files/ConeStepMapping.pdf>.
5. Porumbescu, S.D., et al., *Shell maps*. Proceedings of ACM SIGGRAPH 2005, 2005. **24**(3): p. 626--633.
6. Wang, X., et al., *Generalized displacement maps*. Eurographics Symposium on Rendering, 2004: p. 227--233.
7. Wang, X.C., et al., *View-dependent displacement mapping*. ACM Transactions on Graphics, 2002. **22**(3): p. 334--339.
8. Tatarchuk, N., *Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering*. SIGGRAPH '06: ACM SIGGRAPH 2006 Courses, 2006: p. 81--112.
9. Oliveira, M.M. and F. Policarpo, *An Efficient Representation for Surface Details*. 2005, UFRGS Technical Report RP-351.
10. Jeschke, S., S. Mantler, and M. Wimmer, *Interactive Smooth and Curved Shell Mapping*. Proceedings of EGSR'07, 2007: p. 351--360.

11. Chen, Y.-C. and C.-F. Chang, *A Prism-Free Method for Silhouette Rendering in Inverse Displacement Mapping*. Computer Graphics Forum, 2008. **27**(7): p. 1929-1936.
12. Bishop, G. and D.M. Weimer, *Fast phong shading*. ACM SIGGRAPH Computer Graphics, 1986. **20**(4): p. 103--106.
13. Watt, A. and M. Watt, *Advanced Animation and Rendering Techniques*. 1992: ACM Press. 472
14. Gu, X., S.J. Gortler, and H. Hoppe, *Geometry images*. ACM Transactions on Graphics, 2002. **21**: p. 355--361.
15. Mario, B. and K. Leif, *High-Quality Point-Based Rendering on Modern GPUs*. Proceedings of the 11th Pacific Conference on Computer Graphics and Applications, 2003: p. 335--343.
16. Sainz, M., R. Pajarola, and R. Lario, *Points reloaded: Point-based rendering revisited*. Proceedings Symposium on Point-Based Graphics, 2004: p. 121--128.
17. Marroquim, R., M. Kraus, and P.R. Cavalcanti, *Interactive Point Based Rendering Using Image Reconstruction*. Symposium on Point Based Graphics, 2007: p. 101--108.
18. Ahuja, N. and C. Nash, *Octree representations of moving objects*. Comput. Vision, Graphics, Image Processing, 1984. **26**: p. 207--216.
19. Jackins, C.L. and S.L. Tanimoto, *Oct-trees and their use in representing three-dimensional objects*. Computer Graphics Image Processing, 1980: p. 249--270.
20. Osse, W. and N. Ahuja, *Efficient octree representation of moving objects*. Proceedings of the IAPR 7th International Conference on Pattern Recognition, 1984: p. 821--823.
21. Knoll, A., *A Short Survey of Octree Volume Rendering Techniques*. GI Lecture Notes in Informatics, 2006.

22. Roth, S.D., *Ray Casting for Modeling Solids*. j-CGIP, 1982. **18**: p. 109--144.
23. Levoy, M., *Efficient ray tracing of volume data*. ACM Transactions on graphics, 1990. **9**(3): p. 245--261.
24. Kajiya, J.T., *Ray tracing parametric patches*. Proceedings of the 9th annual conference on Computer graphics and interactive techniques, 1982: p. 245--254.
25. Pabst, H.F., et al., *Ray Casting of Trimmed NURBS Surfaces on the GPU*. Interactive Ray Tracing 2006, IEEE Symposium on, 2006: p. 151--160.
26. Premecz, M., *Iterative parallax mapping with slope information*. Central European Seminar on Computer Graphics, 2006.
27. Ize, T., I. Wald, and S.G. Parker, *Asynchronous BVH construction for ray tracing dynamic scenes on parallel multi-core architectures*. Eurographics Symposium on Parallel Graphics and Visualization, 2007: p. 101--108.
28. Popov, S., et al., *Stackless kd-tree traversal for high performance GPU ray tracing*. Computer Graphics Forum, 2007. **26**: p. 415--424.
29. Horn, D.R., et al., *Interactive kd- tree GPU ray tracing*. Proceedings of I3D 2007, 2007: p. 167--174
30. Zhou, K., et al., *Real-time KD-tree construction on graphics hardware*. ACM Trans. Graph., 2008. **27**(5): p. 1--11.
31. Wei, L.-Y., et al., *Nonlinear beam tracing on a GPU*. Microsoft Research Technical Report MSR-TR-2007-34, 2008.
32. Cook, R.L. and K.E. Torrance, *A reflectance model for computer graphics*. ACM SIGGRAPH Computer Graphics, 1981. **15**(3): p. 307--316.
33. Michael, A., P. Simon, and S. Peter, *A microfacet-based BRDF generator*. Proceedings of the 27th annual conference on Computer graphics and interactive techniques, 2000.

34. Kelemen, C. and L. Szirmay-Kalos, *A microfacet based coupled specular-matte BRDF model with importance sampling*. Eurographics Conference, 2001: p. 25--34.
35. Blinn, J.F. and M.E. Newell, *Texture and reflection in computer generated images*. Communications of the ACM, 1976. **19**(10): p. 542--547.
36. Blinn, J.F., *Models of light reflection for computer synthesized pictures*. ACM SIGGRAPH Computer Graphics, 1977. **11**(2): p. 192--198.
37. Carey, R.J. and D.P. Greenberg, *Textures for realistic image synthesis*. Computers & graphics, 1985. **9**(2): p. 125--138.
38. Heckbert, P.S., *Survey of texture mapping*. IEEE Computer Graphics and Applications, 1986. **6**(11): p. 56--67.
39. Bennebroek, K., et al., *Design principles of hardware-based phong shading and bump-mapping*. Computers & Graphics, 1997. **21**(2): p. 143--149.
40. Bunnell, M., *Adaptive tessellation of subdivision surfaces with displacement mapping*. GPU Gems, 2005. **2**: p. 109--122.
41. Blinn, J.F., *Simulation of wrinkled surfaces*. ACM SIGGRAPH Computer Graphics, 1978. **12**(3): p. 286--292.
42. Peercy, M., J. Airey, and B. Cabral, *Efficient bump mapping hardware*. Proceedings of the 24th annual conference on Computer graphics and interactive techniques, 1997: p. 303--306.
43. Kilgard, M.J., *A practical and robust bump-mapping technique for today's GPUs*. GDC 2000: Advanced OpenGL Game Development, 2000.
44. Tarini, M., et al., *Real time, accurate, multi-featured rendering of bump mapped surfaces*. EUROGRAPHICS 2000. **19**: p. 119--130.

45. Barry, G.B. and L.M. Nelson, *Smooth transitions between bump rendering algorithms*. Proceedings of the 20th annual conference on Computer graphics and interactive techniques, 1993: p. 183--190.
46. Kaneko, T., et al., *Detailed Shape Representation with Parallax Mapping*. Proceedings of ICAT 2001, 2001: p. 205--208.
47. Welsh, T., *Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces*. Infiscape Corporation, 2004.
48. Brawley, Z. and N. Tatarchuk, *Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing*. ShaderX3: Advanced Rendering with DirectX and OpenGL, 2004: p. 135--154.
49. Policarpo, F., M.M. Oliveira, and J.L.D. Comba, *Real-time relief mapping on arbitrary polygonal surfaces*. Proceedings of the 2005 symposium on Interactive 3D graphics and games, 2005: p. 155--162.
50. Keith, Y. and J. Martin, *Displacement mapping with ray-casting in hardware*. ACM SIGGRAPH 2004 Sketches, 2004.
51. Eric A. Risser and M. Pattanaik, *Interval Mapping*. 2006.
52. Hart, J.C., *Sphere tracing: Simple robust antialiased rendering of distance-based implicit surfaces*. SIGGRAPH'93 Modeling, Visualizing, and Animating Implicit Surfaces course notes: p. 14--1.
53. Donnelly, W., *Per-Pixel Displacement Mapping with Distance Functions*. 2005, Addison-Wesley. p. 334--339.
54. Jolliffe, I.T., *Principal component analysis*. 2nd ed. 2002: Springer New York.
55. Mc Guire, M., *Steep Parallax Mapping*. I3D 2005 Poster, 2005.
56. Policarpo, F. and M.M. Oliveira, *Relaxed Cone Stepping for Relief Mapping*. GPU Gems, 2007. 3.

57. Oh, K., H. Ki, and C.H. Lee, *Pyramidal displacement mapping: a GPU based artifacts-free ray tracing through an image pyramid*. Proceedings of the ACM symposium on Virtual reality software and technology, 2006: p. 75--82.
58. Szirmay-Kalos, L. and T. Umenhoffer, *Displacement Mapping on the GPU State of the Art*. Computer Graphics Forum, 2008. **27**: p. 1567--1592.

AUTHOR PUBLICATION

- [1] Lu T.-K., Low K.-L., and Zheng J., *Fast visualization of complex 3D models using displacement mapping*, in *Proceedings of Graphics Interface 2009*, British Columbia, Canada. pp. 7--14.