FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Cell-Based Object Representation

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Computergraphik/Digitale Bildverarbeitung

eingereicht von

Marcel Nürnberg
Matrikelnummer 0425202

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Associate Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Betreuender Assistent: Dipl.-Ing. Dr.techn. Stefan Jeschke

Wien, 22.6.2011 _____ _____

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Marcel Nürnberg
Heiligenstädter Straße 84/57/9
1190 Wien

„Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.“

Ort, Datum:_____ Unterschrift:_____

Abstract

Today's real-time applications, such as computer games or virtual environments, need to display more and more geometrically complex surfaces. The appearance of such surfaces are achieved by local parallax, correct occlusions, convincing silhouettes and even by sophisticated effects such as self-reflection, refraction, translucency, self-shadowing and caustics to name a few. Hence simple texturing mapping is insufficient to produce such high geometric complexity.
This thesis proposes a *cell-based* approach to model and render repetitive fine scaled details with a high visual quality. The main idea of the precomputation is to decompose the object into a low frequent geometry (the general shape of the object) and high frequent surface details. The high frequent surface details are represented by so-called *cells* tiled all over the object space. The precomputed *cell-based object representation* is displayed by a ray tracer providing correct parallax, occlusions and silhouettes. This thesis proves that sophisticated effects such as specular self-reflection and refraction can easily be rendered with the cell-based approach.

Kurzfassung

Die heutigen Echzeitanwendungen, wie zB Computerspiele oder virtuelle Umgebungen, müssen immer komplexere geometrische Oberflächen darstellen. Dies Beschaffenheit solcher Oberflächen umfassen lokale Parallaxe, korrekten Überdeckungen, überzeugende Silhouetten und sogar kompliziertere Effekte wie zB Selbstreflektion, Refraktion, Transluzenz, Selbst-Schattierung und Kaustik. Mit Texture Mapping kann diese geometrische Komplexität nicht dargestellt werden.
Diese Diplomarbeit schlägt einen zellenbasierten Ansatz vor, um wiederholende feine Oberflächenstrukturen mit hoher visueller Qualität zu modellieren und darzustellen. Die Grundidee der Vorberechnung ist das Objekt in eine tieffrequente Geometrie (grobe Repräsentation des Objekts) und hochfrequente Oberflächendetails zu unterteilen. Die hochfrequenten Oberflächendetails werden durch sogenannte Zellen, die über den Objektspace verteilt werden, repräsentiert. Diese vorberechnete, zellbasierte Objektrepräsentation wird mittels Ray Tracing dargestellt. Dies ermöglicht korrekte Parallaxe, Verdeckungen und Silhouetten. In dieser Diplomarbeit wird auch bewiesen, dass kompliziertere Effekte wie spekulare Selbstreflektion und Refraktion mittels zellenbasierten Ansatz genauso leicht implementiert werden können.

Contents

„If it looks like computer graphics,
it is not good computer graphics. „

Jeremy Birn

Chapter 1

Introduction

The human visual system is able to distinguish between various material properties such as texture, roughness, temperature, thickness and shininess to name a few. If we take a closer look at some real world objects, such as in Figure 1.1, we can recognize that many of these material properties are given by the fine scaled surface details. Hence to reproduce these various types of material properties in computer graphics, surface details play a key role in digital image synthesis. Therefore numerous data structures and rendering techniques got developed especially to define fine scaled details on the object's surface. The basic idea is to decompose the object into a rough representation described by a polygonal mesh and the surface defined by the fine scaled details.

Figure 1.1: *A close up view of real world objects. Upper row from left to right: swan, butterfly, tree. Lower row from left to right: strawberry, lemon, crab.*

Typically, surface appears with some repetitive fine scaled details. If we take a look at Figure 1.1, especially on the close up view of the strawberry, we could clearly see that the strawberry contains little seeds repetitively distributed on its surface. The same for the crab's surface appearing with larger black and smaller white bumps in a repetitive surface structure. Not only on the surface but also in the inside of an object we could find the repetition property. In Figure 1.2 you can see an orange pulp, which appears with a repetitive cellular detail structure.

Figure 1.2: *A close up view of an orange pulp [25] having a repetitive cellular structure.*

Hence there is a need for modeling and rendering repetitive surface details in an efficient way. The term "*efficient*" refers to high image quality by simultaneously performing fast rendering providing a low memory consumption. For example if we take a look again at the strawberry close up view in Figure 1.1 we would possibly only model a single seed once and copy it all over the strawberry's surface to avoid redundant information. This is the basic idea of "*cell-based object representation*" by defining a so-called "*cell*", which represents repetitive surface details, which could be tiled over the object space and finally rendered by a ray tracer.

1.1 Aim of the Thesis

This thesis proposes an approach to model and render repetitive surfaces efficiently by defining a "*cell-based*" data structure. The term cell-based refers to a data structure, which holds cells as spatial elements describing the fine scaled details of an object. This proposed method is derived from solid texturing [18] using a given mesh and a 3D Texture. Instead intersecting the viewing ray with the mesh, our approach intersects the viewing ray with the cells that make up the cell-based object.
The cell-based data structure makes ray tracing very efficiently and avoids to store redundant information, which leads into low memory requirements and therefore getting high frame rates. Due to the ray tracing it is also possible to render effects, such as reflection and refraction easily computed by shooting additional rays. Furthermore the cell-based approach enhances the realism and visual quality of complex surfaces by supporting self-occlusion, parallax and correct object silhouettes.

1.2 Thesis Structure

This thesis is structured into five chapters as follows:

- Chapter 2 gives an explanation of some terms used for classification. Furthermore this chapter gives a survey of common mesostructured rendering techniques and cell-based texturing.

- Chapter 3 proposes the main idea of cell-based object representation. This chapter is split into two parts. First the precomputation stage is discussed. Second a ray tracer is proposed for displaying the precomputed cell-based objects. Beside the basic ray tracing two advanced effects such as specular self-reflection and refraction are explained.

- Chapter 4 deals with the implementation details and explains the preprocessing stage from a technical point of view.

- Chapter 5 discusses the memory consumption, computational time and the visual quality of some results created with the proposed cell-based approach.

„I like to collect things. When I was young I collected stamps; now I collect empty margarine tubs and algorithms for drawing circles.„

Jim Blinn

Chapter 2

State of the Art

Many different algorithms and data structures have been developed to enhance the realism and visual quality in real time applications. In addition, memory consumption has to be kept low. Due to the increasing power of processors and increasing memory storage more complex scenes are rendered. Therefore new algorithms and data structures are needed. Nowadays there are many such mesostructure rendering techniques.
In Chapter 2.1 the most important features are explained to classify the mesostructure rendering techniques mentioned in Chapter 2.3. Chapter 2.2 explains the terms macro-, micro- and mesostructure. Finally Chapter 2.4 gives an overview of some methods for cell-based texturing.

2.1 Features

How do we know which mesostructure rendering technique is the best? Every technique has its pros and cons and it highly depends on the object surface we want to synthesize. However each technique could be classified by its supported features such as parallax, silhouettes, self shadowing, self occlusions, refraction, caustics, self reflections and translucency to name a few. All these mentioned features are necessary for realistic appearance. In this chapter the most important features will be explained.

2.1.1 Parallax

Motion parallax is a depth cue that results from our motion. As we move, objects that are closer to us move farther across our field of view than objects that are in the distance.

2.1.2 Silhouette

An object with a silhouette feature also contains the complex surface on the outermost object's boundary. This feature could be best described by an orange. The rough shape of an orange is a sphere, which means that the outermost object's boundary is approximately a concentric circle. But in a close up view you will realize that the orange outermost boundary isn't exactly circular, but rather circular with bumps on it. If we synthesize this object in our real-time application and the orange appears with circular bumpy shape it contains the silhouette feature.

2.1.3 Self Shadowing

A shadow on an object appears whenever it is not visible from the light source's position. This happens because an occluder lies between the object and the light source. In case the occluder itself is the object's geometry we are speaking of self shadowing.

2.1.4 Self Occlusion

Objects that are behind other objects and not seen from the point of view are occluded. If the object occludes areas of itself we are speaking of self occlusion.

2.1.5 Self Reflection

The law of reflection says that for specular reflection the angle at which the wave is incident on the surface equals the angle at which it is reflected. Self reflection means that a light is reflected on it's surface more than once, before it reaches the observer.

2.1.6 Refraction

Refraction occurs when light waves penetrate into a refractive medium, such as glass or water. At the medium's boundary a change in direction occurs due to increase or decrease of light ray's speed. For example if light penetrates from air into water a decrease of light occurs, which leads into a change of direction. The amount of bending the incident light depends on the media's property.

2.1.7 Caustics

Caustics occur by the possible paths of the light beam through the medium, accounting for the refraction and reflection. This effect is best seen when light shines on a drinking glass. The glass casts a shadow and simultaneously also produces a curved region of bright light.

2.1.8 Translucency

Translucency is a process of sunlight transmitted through the media and exiting on the opposite side. Very thin materials such as paper or leaves show the translucency effect very well by holding those directly into the sunlight. The back-lit side of the leave or the piece of paper occurs because of the translucency effect.

2.2 Macro-, Micro- and Mesostructure

A geometrical object is usually defined on the three scales, the "*mesostructure*" level, "*macrostructure*" level and "*microstructure*" level [2]. A macrostructure level describes a geometric model as a set of polygons, which represents the rough object shape. On the other hand microstructure enriches the geometric model with a more detailed look by surface microfacets, which are indistinguishable by human eyes. This fine scaled detail is mostly defined by a texture map. Mesostructure is in scale between macro- and microstructure, which represents the complex surface of an object. That is the mesostructure level is defined by the geometric details, which are relatively small but still visible for human eyes such as bumps. Note the bark structure on the tree trunk in Figure 1.1. In this case the bark structure represents the mesostructure level due to high-frequency visible geometry.

2.3 Mesostructure Rendering Techniques

Back in the seventies Catmull, who is the current president of Walt Disney Animation and Pixar Animation Studios, invented the commonly used technique texture mapping [1]. With texture mapping a revolutionary step was taken into a new direction. For the first time it was possible to add details on a surface by mapping textures on an object, which results to an more realistic look.
Mesostructure rendering techniques simulate geometric details by texture mapping techniques. This chapter gives a review of the most prevalent rendering techniques to generate mesostructure surfaces with focus on supported features mentioned in Chapter 2.1.

2.3.1 Bump Mapping

Bump mapping is probably the most known texture mapping technique in computer graphics and it is still used in some real-time applications. Blinn [3] introduced a method that adds detail to a surface without modifying the geometric representation itself. This wrinkled look is achieved by perturbing the surface normal per pixel, which results after the illumination computation in surface irregularities.

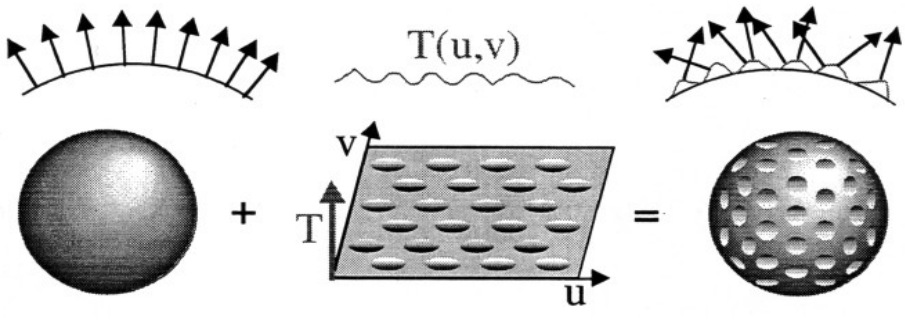

Figure 2.1: *A bump map T(u,v) is applied on a macrostructure to generate bumps on a surface. Surface normals (left) and resulting perturbed surface normals (right). [3]*

The perturbation function T(u,v) itself is defined as a bump map texture as we can see in Figure 2.1. This bump map stores gray values, which represent the offset by which the normals has to be wrinkled. A comparison between conventional texture mapping and bump mapping is shown in Figure 2.2.

2.3.2 Normal Mapping

Another approach of bump mapping is normal mapping, sometimes called as "dot3 bump mapping" [5]. Instead of perturbing the surface normals, normal mapping replaces the existing surface normal entirely. These pre-computed normals are stored in a multichannel texture also referred to as normal map. The values of each channel represent the xyz coordinate of the replacing normal (see Figure 2.3.).

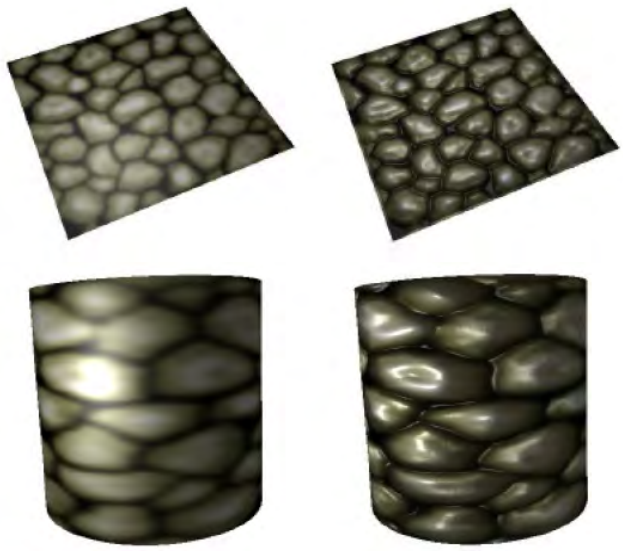

Figure 2.2: *A comparison between conventional texture mapping (left) and bump mapping (right). Note the difference in the specular highlights [2].*

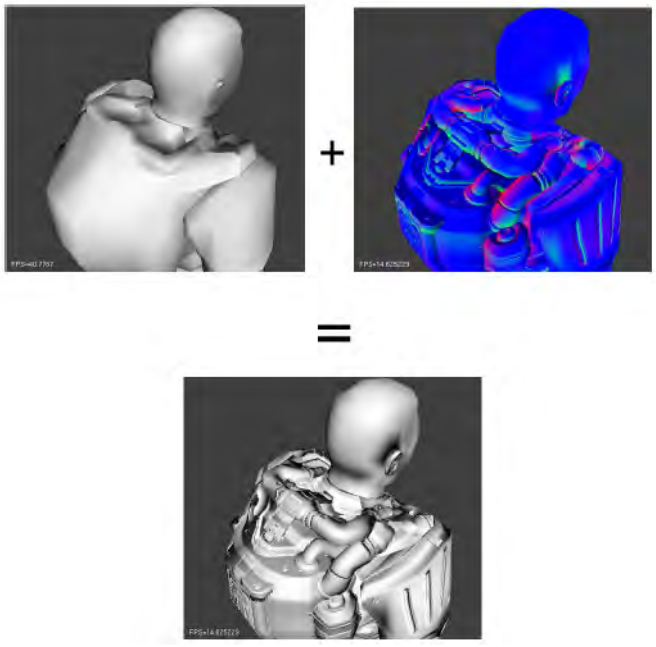

Figure 2.3: *Original model (top left). Normal map projected on normal mapped model (top right). RGB values of the normal map represent the xyz-coordinates of a normal. Resulting image after normal mapping (bottom). [6]*

Normal mapping is usually found in two varieties, object-space and tangent-space normal mapping. They differ in coordinate systems in which the normals are measured and stored (see Figure 2.4).

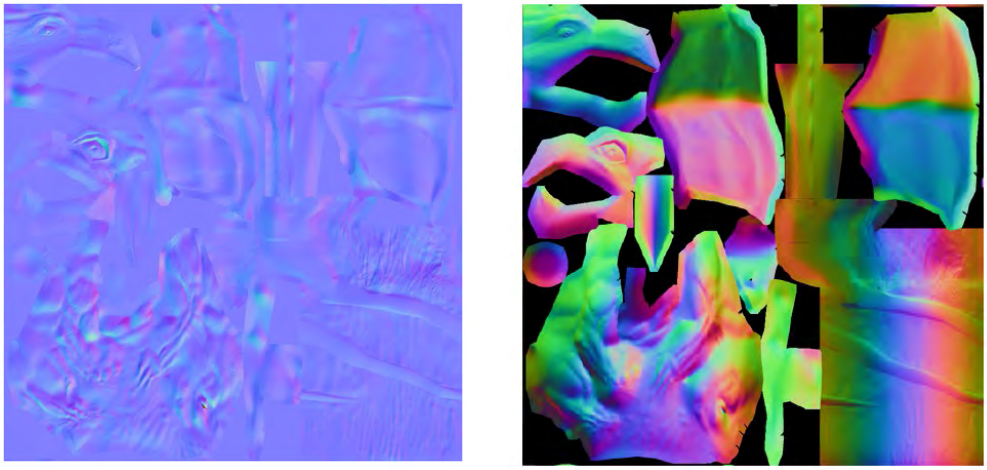

Figure 2.4: *Tangent-space normal map (left). Object-space normal map (right). [7]*

Due to normal map, normal mapping is theoretically faster than bump mapping, because only a simple texture lookup has to be done to obtain the pre-computed normal for light computation. In consideration of the fact that the whole normal is stored in a texture, this approach needs more memory than bump mapping. Nowadays there is no noticeable computational difference due to the modern graphics hardware.
With bump mapping and normal mapping it is possible to model details to a surface without creating new surfaces. Unfortunately this fact leads to artifacts in the silhouette of the object. Hence, the bump mapped sphere in Figure 2.1 still has a circular silhouette. A further problem of this approach is that neither self occlusion nor self shadowing is considered at all. In the following chapters all these problems are going to be solved by advanced rendering techniques.

2.3.3 Horizon Mapping

Shadows provide important perceptual cues for understanding surface shape. However, horizon mapping [8] extends the standard bump mapping by calculating self shadowing. This effect is provided by horizon maps. Each texel contains eight pre-computed angles to the horizon (see Figure 2.5), which holds information at what height the sky becomes visible a given direction. In other words a surface gets illuminated if and only if the light direction is higher than the interpolated horizon value. Figure 2.5 (right) shows an object rendered with horizon mapping.
Unfortunately the self shadowing effect is not for nothing due to additional texture memory requirements. Furthermore no self occlusion and silhouettes are considered.

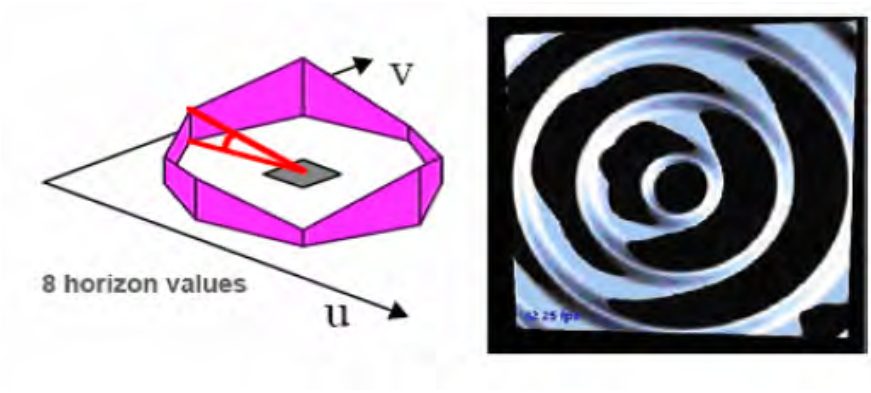

Figure 2.5: *Eight horizon values per pixel for calculating self shadowing (left). Model rendered with horizon mapping (right). Note the self shadows in the valleys. [8]*

2.3.4 Parallax Mapping

Parallax mapping considers the parallax features explained in Chapter 2.1.1 and was introduced by Tomomichi Kaneko et al [9] in 2001.
The fundamental idea of this approach is to shift the texture coordinates dynamically using the view vector and the current height map value shown in Figure 2.6. Consequently the greater angle the more texture coordinate gets displaced, which gives us the illusion of parallax motion. This effect is best shown at the house tops' side walls on the left side of the brick wall in Figure 2.7, which demonstrates a rendered wall with and without parallax mapping.

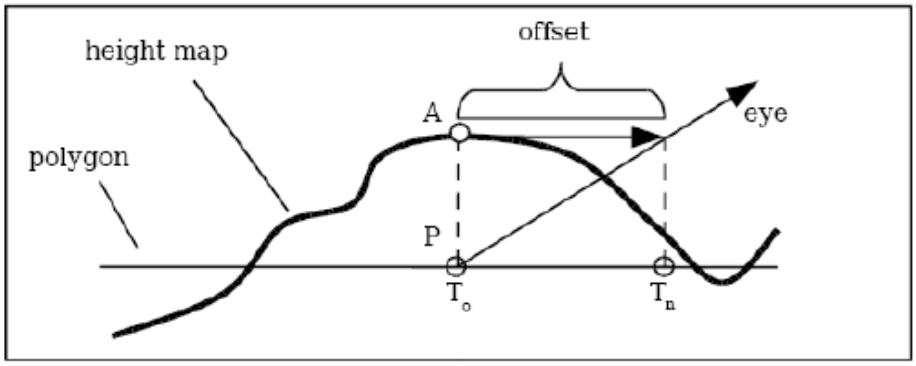

Figure 2.6: *The texture coordinate offset for parallax mapping is obtained by modulating the eye vector by the surface height. [2]*

Figure 2.7: *Wall rendered with bump mapping (left). Same wall rendered with parallax mapping (right). Note the parallax effect at the house tops' side walls on the left side of the brick wall. [9]*

A drawback of this approach is that a noticeable distortion occurs, which is caused by depth approximation. This results when the viewing angle becomes more grazing shown in Figure 2.8.

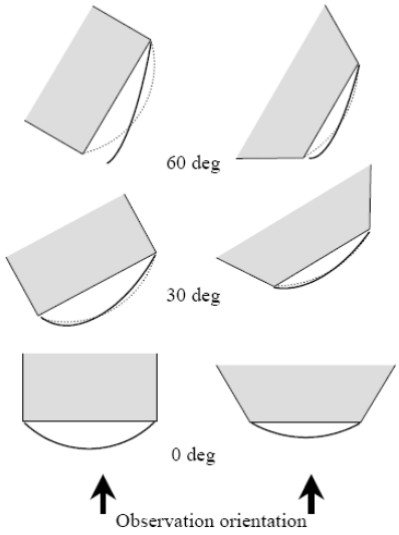

Figure 2.8: *The more the grazing viewing angle grows, the more distortion occurs. Note that the distortion does not grow linearly along with the viewing angle. [9]*

2.3.5 Offset Parallax Mapping

Welsh [10] solves the problem of parallax mapping at grazing angles by limiting the texture coordinate offsets, so that they never get larger than the height at T_o shown in Figure 2.9. Figure 2.10 compares parallax mapping with and without offset limiting.

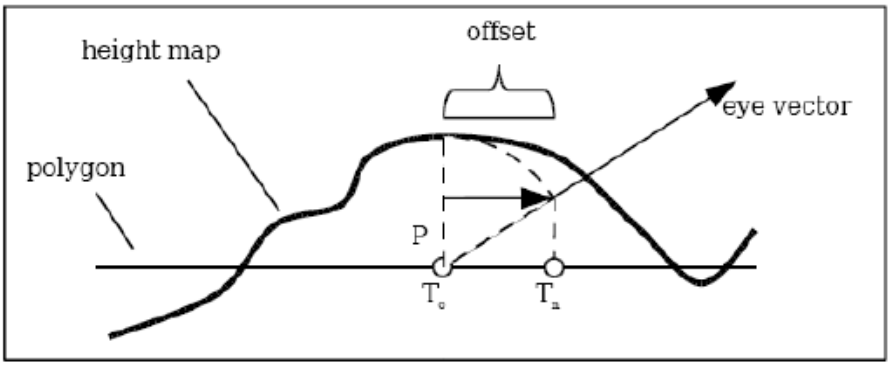

Figure 2.9: *Basic idea of offset parallax mapping. The texture coordinate offset will be no longer than the height at T_o.[2]*

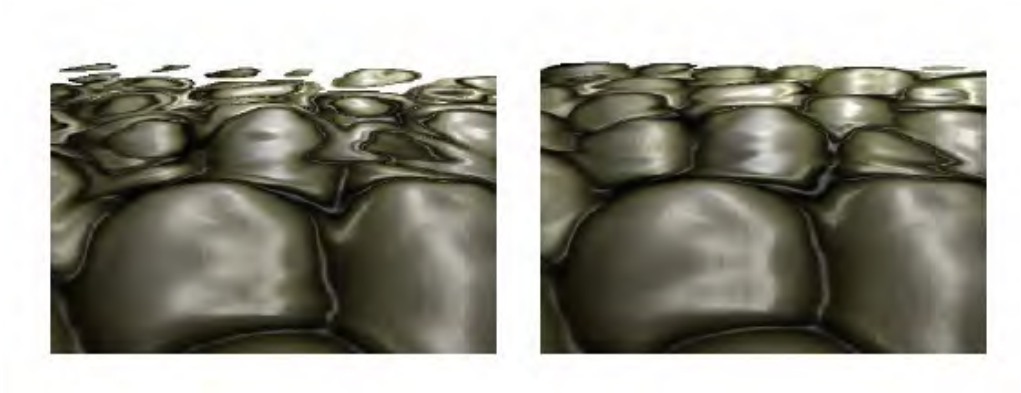

Figure 2.10: *Comparison of parallax mapping (left) and offset parallax mapping (right). Note the swimming artifacts on the stone ground at grazing angles (left), which gets solved by offset parallax mapping (right). [2]*

2.3.6 Displacement Mapping

The fundamental approach of displacement mapping [11] originates from REYES algorithm [12], whereby a geometric model is tessellated into micropolygons, whose size are close to or even less than the area of a pixel. Displacement mapping also tessellates the macrostructure into a more detailed geometric representation, in order to displace the vertices in direction of the surface normal given by the height map stored as a texture map(see Figure 2.11).

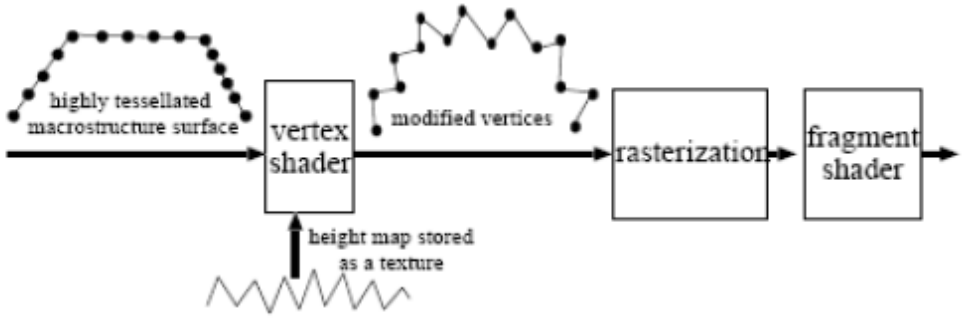

Figure 2.11: *Rendering pipeline for displacement mapping. Displacement mapping is a per-vertex method and therefore done in the vertex shader. [2]*

On the basis of this approach displacement mapping is able to solve the self occlusion and silhouette problem mentioned in the previous sections and seen in Figure 2.12. Although displacement mapping can be implemented in hardware, its performance is limited by the large number of vertices that result from fine mesh subdivision.

Figure 2.12: *Comparison of different mesostructure rendering techniques. Bump mapping (top left), horizon mapping (top right), displacement mapping (down left) and view-dependent displacement mapping with self-shadowing (down right). [2]*

2.3.7 View-dependent Displacement Mapping (VDM)

VDM [13] is an enhancement of displacement mapping achieved by two substantial modifications. Firstly, VDM represents displacements along the viewing direction instead of the mesh normal direction shown in Figure 2.13. Secondly, due to increasing performance of graphics hardware, VDM is implemented using per-pixel operations to obtain faster rendering, since no fine mesh subdivision has to be done. Nevertheless the VDM data consumes most of the graphics hardware memory and therefore a data composition and compression is indispensable. VDM Figure 2.12 shows a comparison of bump mapping, horizon mapping, displacement mapping and VDM. Note that VDM considers self occlusion, self shadowing and silhouettes.

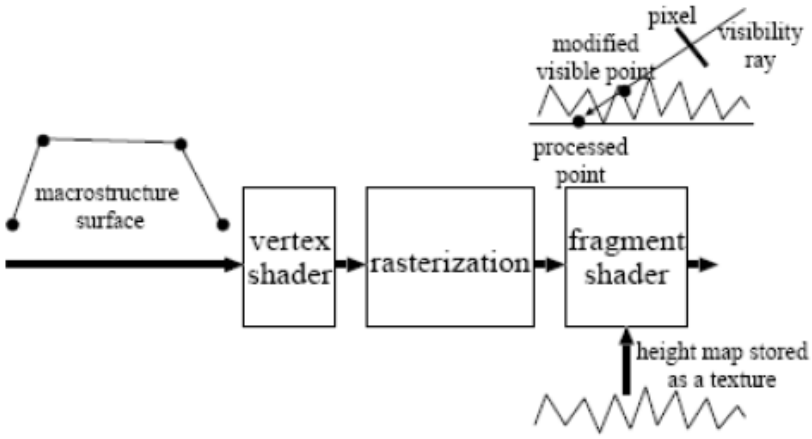

Figure 2.13: *Rendering pipeline for view-dependent displacement mapping. VDM is a per-pixel method and therefore done in the fragment shader. Hence, no fine mesh subdivision has to be done. [2]*

2.3.8 Relief Mapping

Relief mapping [14] is based on an efficient ray-height-field intersection algorithm. That is, to locate the intersection between the height field and the ray, two search algorithms are needed. At first, a linear search is performed to find a pair of points on the ray that enclose the possibly first intersection. Second, a binary search refines these approximations. Both search algorithms are demonstrated in Figure 2.14.

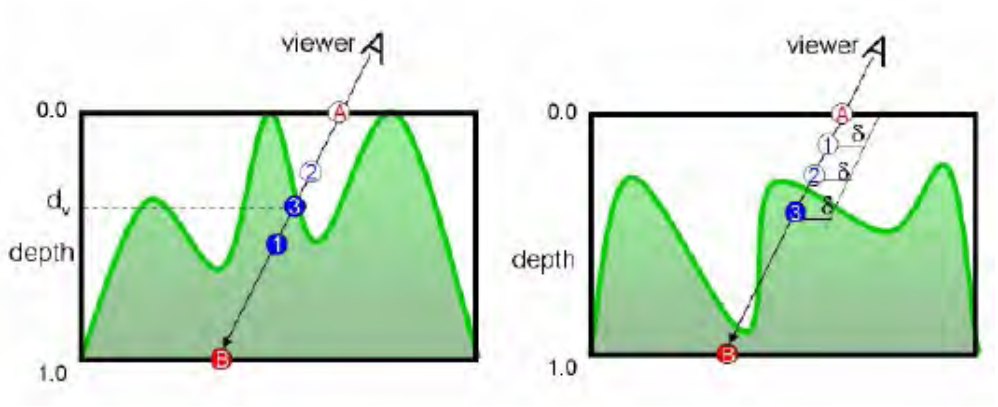

Figure 2.14: *Binary search (left) followed by a linear search (right) to calculate ray intersection with a height field. [14]*

Self shadowing is accomplished in an analogous manner by checking an intersection between a ray from the light source and the height field shown in Figure 2.15. Relief mapping needs two texture maps, a depth map for intersection calculation and a normal map for illumination computation shown in Figure 2.16.

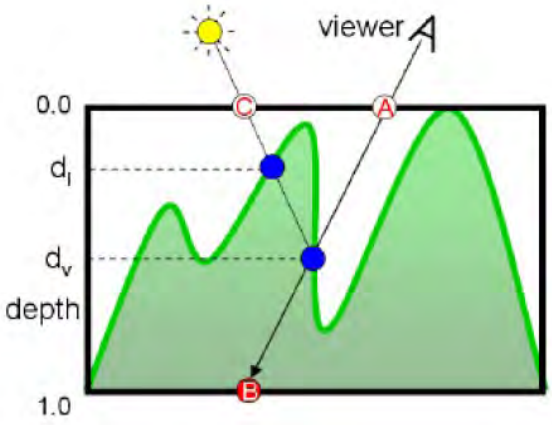

Figure 2.15: *Computing self shadows by shooting another ray to the light source, checking if there is an intersection with the height field.[14]*

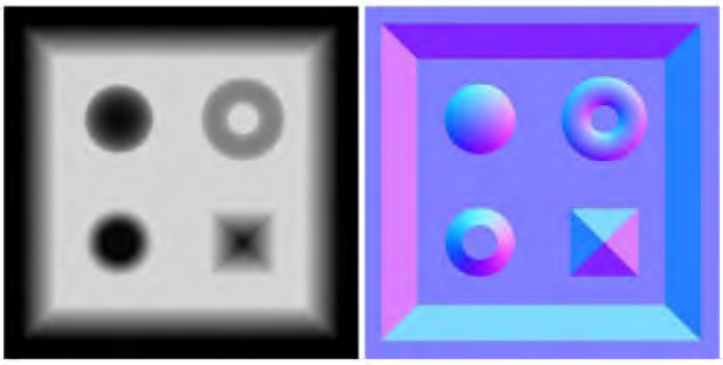

Figure 2.16: *Gray values in the depth map (left) represent the height values and RGB values in the normal map (right) represent the xyz-values of a normal. [14]*

2.3.9 Enhanced Relief Mapping

The Relief mapping algorithm can be extended [15] to handle non-height-field representations. This idea is realized by storing not only one but even more depth values for intersection calculation. Figure 2.17 shows four intersections d_i, d_j, d_k and d_n for ray (α, β).

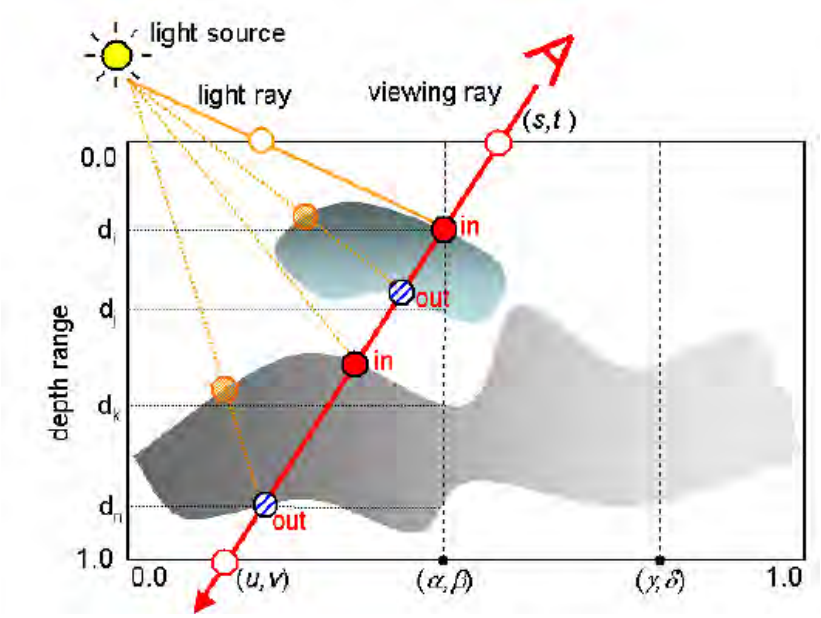

Figure 2.17: *Ray intersection with four layers, which results in intersection with four depth value d_i, d_j, d_k and d_n for ray (α, β). [15]*

In modern graphic hardware four layers could be checked for ray intersection in parallel. Hence, three four-channel (RGBα) texture maps are needed, which is shown in Figure 2.18. One texture contains depth values for the four layers in the RGBα channels, another one stores the x-values of the unit-length normal vectors and the last one holds the y-values also for the unit-length normal vectors. The z components of the normals are computed as:

$$z = \sqrt{max(0, 1 - (x^2 + y^2))}.$$

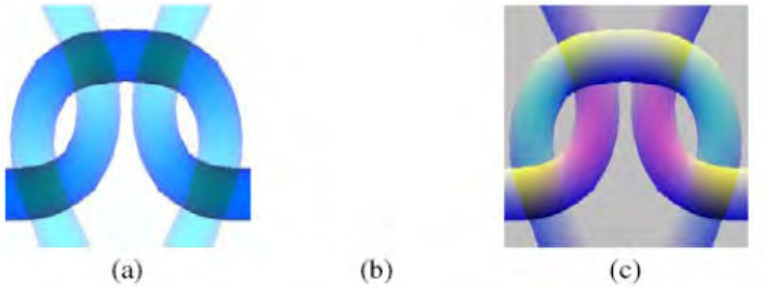

Figure 2.18: *(a) Depth values for the four layers stored in the RGBα channels of the texture. (b) x components of the unit normal vectors for the four layers. (c) y components of the unit normal vectors for the four layers. [15]*

Another enhancement of relief mapping [16] considers the silhouette by adding two coefficients for every vertex representing a quadric surface that locally approximates the object's geometry. These coefficients are pre-computed once for a polygonal model by using a least squares fitting algorithm. Results of the enhanced relief mapping technique are shown in Figure 2.19. Note the silhouette at the columns and the stone object. Furthermore a comparison of bump mapping, parallax mapping and relief mapping is shown in Figure 2.20.
On the whole relief mapping offers all advantages as displacement mapping does without modifying respectively subdividing the underlying geometry. That is, no additional memory consumption is needed, which makes real-time rendering possible.

Figure 2.19: *A room scene rendered with enhanced relief mapping technique. Note the silhouette at the columns and the stone object. The self shadowing effect is best shown at the stone object. [16]*

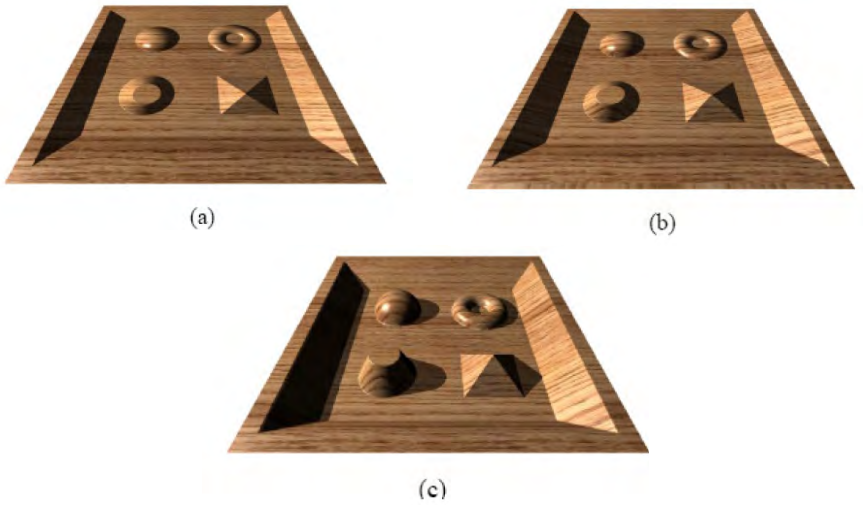

Figure 2.20: *: A comparison of (a) bump mapping, (b) parallax mapping and (c) relief mapping. The upper and the middle image remain flat, whereas the bottom image provides a strong depth cue. Also note the self-occlusion and the self-shadowing effect. [16]*

2.3.10 Shell Mapping

With shell mapping [17] it is possible to model surface details by applying a three-dimensional volume onto a surface. Therefore a new space, namely shell space, has to be specified, which lies between the original surface and an offset surface to the base shown in Figure 2.21. Furthermore a one-to-one function between texture space and shell space so-called shell map is responsible for mapping the surface details onto the polygonal model.

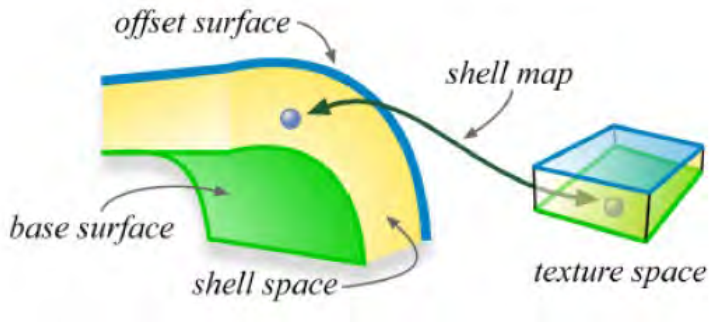

Figure 2.21: *Shell space lies between base surface and offset surface. A shell map itself is a one-to-one function between texture space and shell space. [17]*

Shell Mapping meets two main challenges. That is to say, firstly generating an offset surface paying attention to self intersections and second maintaining a continuous tetrahedral mesh. To construct a continuous tetrahedral mesh, first and foremost every vertex from the base surface has to be connected with the corresponding vertex in the offset surface (see Figure 2.22). In doing so prisms are generated, which have to be split into three tetrahedrals again shown in Figure 2.23. To avoid discontinuity in the tetrahedral mesh, each tetrahedral has to be labeled with a splitting direction. Finally the rippling algorithm comes into play, which propagates the inconsistent labeled edges away, by flipping the label on an adjacent edge, until all prisms are consistent (see Figure 2.24). Hence given a tetrahedron in shell space with its corresponding tetrahedron in texture space, any point can be located by using barycentric coordinates.
Due to usage of ray-tracing, refraction and caustic is supported by shell mapping shown in Figure 2.25. Furthermore a correct parallax, self occlusion, self shadowing and silhouettes are provided by shell mapping.

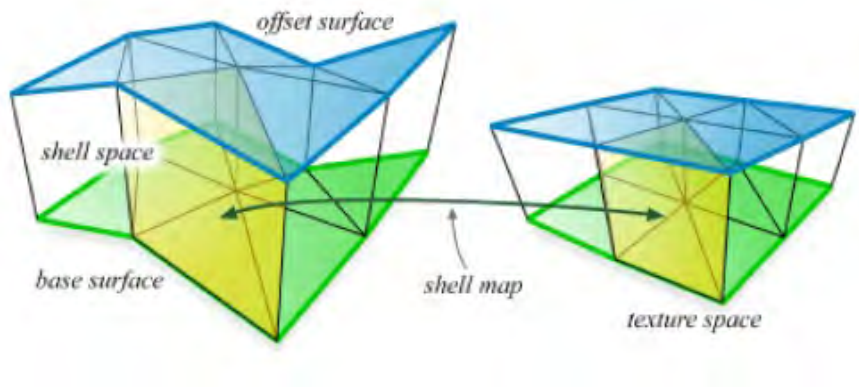

Figure 2.22: *Generated prism by connecting every vertex from the base surface to the corresponding vertex in the offset surface. Prisms in shell space correspond to prisms in texture space defined by a shell map. [17]*

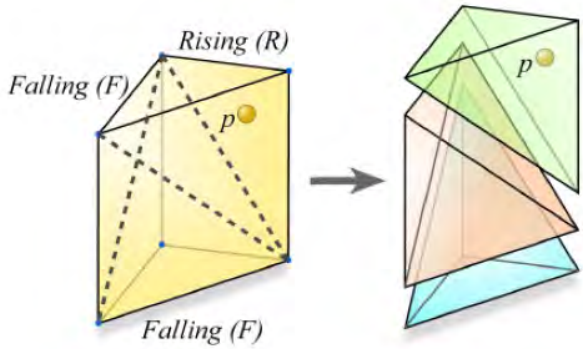

Figure 2.23: *A Prism can be split in six ways, depending on the direction of the triangulation of the quadrilateral faces. The six splits can be characterized with labels R(Raising) and F(Falling). [17]*

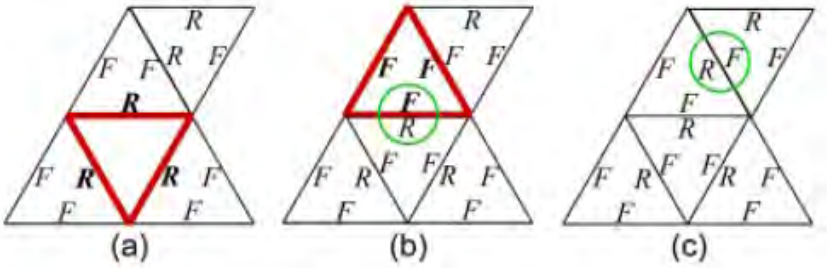

Figure 2.24: *A demonstration of the rippling algorithm. (a) The red triangle represents a prism with inconsistency. (b) Rippling algorithm flips the label on an adjacent edge, until all prisms are consistent shown in (c). [17]*

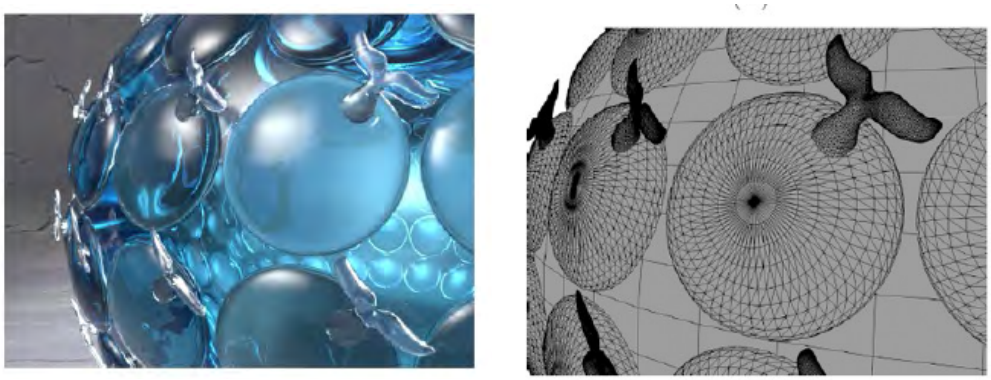

Figure 2.25: *A shell mapped model rendered with ray tracing (left) Generated mesh by shell mapping (right). [17]*

2.4 Cell-Based Texturing

The fundamental concept of cell-based texturing will be discussed in the following related work.

2.4.1 Texture Bombing

Often in large scenes texturing requires a huge amount of memory due to adding visual details to make it more realistic. For example in a nature scene the artist wants to texture thousands of plants on the grassland producing a huge texture, which requires a huge amount of memory. An idea would be to texture only a few tens of plants and irregularly distributing them on the grassland to safe texture memory. This is exactly the basic idea of Texture bombing [26], which is a procedural technique placing patterns in an irregular interval shown in Figure 2.26. Additionally it reduces the problem of regular looking patterns.

Figure 2.26: *Four images stored in a texture map (bottom), which are irregular and randomly tiled over a single texture using texture bombing.*

Texture bombing divides the UV space into a regular grid of cells. Every cell contains an randomly placed image by using a pseudo-random number function. Since the placing image could cross adjacent cells, neighboring cells need to be considered as well. On the whole two coordinates are relevant for image sampling. Firstly, by dividing the UV coordinate by the grid cell size we get the grid cell index, which defines in which image we want to sample. Secondly, the remaining cell offset represents the texture coordinate for sampling in the image. This method isn't only applicable for 2D UV space, but also extendable to the 3D space by dividing the object-space into 3d grid cells.

2.4.2 Voronoi Diagram

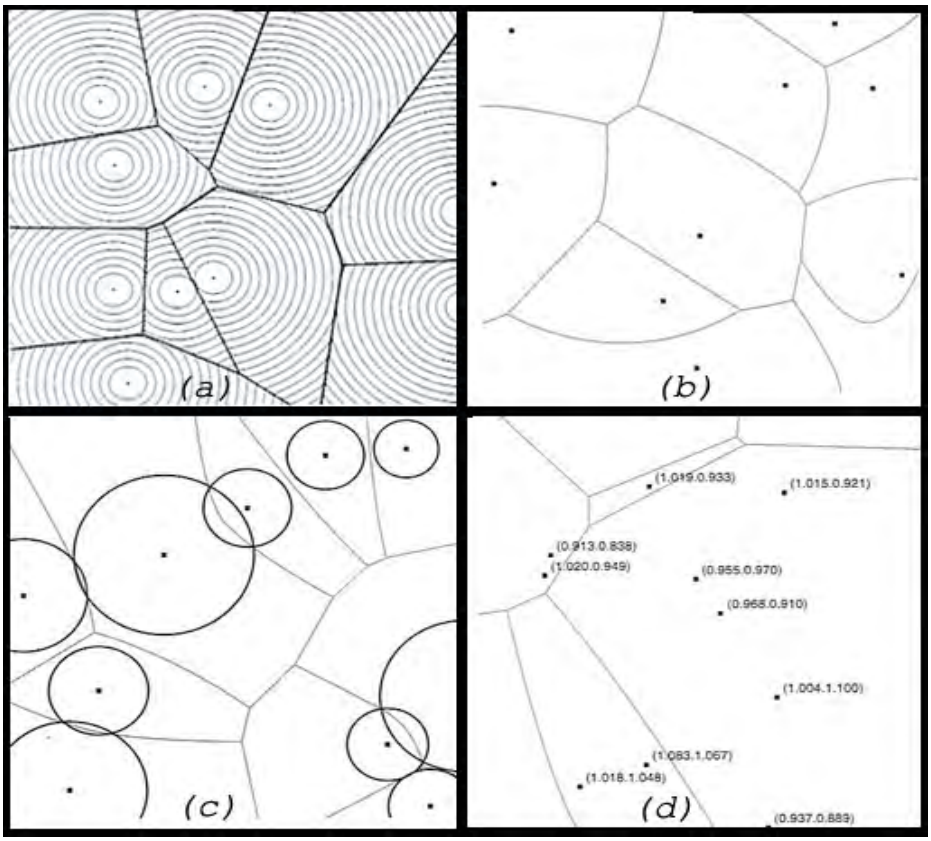

Figure 2.27: *Variations of Voronoi diagrams depending on the distance function. (a) Voronoi diagram with Euclidean distance function. The concentric circles representing the distances to the placed seeds of each Voronoi cell. (b) Anisotrop diagram, (c) Apolonius diagram and (d) Moebius diagram. [21]*

Voronoi diagrams are named after the famous Russian mathematician Georgy Fedoseevich Voronoi. He firstly defined the Voronoi diagram in the general n-dimensional case in 1908. A huge range of applications for Voronoi diagrams [4] could be found in many scientific areas, such as biology, chemistry, geology and many more. Even in computer graphics Voronoi diagram plays an important factor for spatial descriptions.
In general a Voronoi diagram is a finite set S of points Pn in Euclidean space defined by

S = {P1,....,Pn}

For every point x in Euclidean space, there is one point of S closest to x defined by a distance function. The set of all points closer to one point S than to any other point of S is the interior of convex polytope called the "*Voronoi cell*". Set of points x, which are equally distant to more than one point in S are called "*Voronoi cell boundary*". This is best shown in Figure 2.27 (a). Note that the cell boundaries are placed exactly between two equally distant circles.
The appearance of each cell depends highly on the distance function. In the simplest case a Euclidean distance is taken, which could be found in Figure 2.27. Many variations of Voronoi diagrams such as Möbius, Apollonius, anisotropic, spherical or hyperbolic Voronoi diagrams can be easily integrated on the GPU [21] also shown in Figure 2.27.

2.4.3 A Cellular Texture Basis Function

The cellular texture basis function [19] was developed for practical texture design and complements Perlin noise. The basic idea is to divide space into a grid of uniformly spaced cubes. Each cube contains one or more feature points, which get spread through space based on a Poisson distribution. Now a function F1(x) defines the distance from x to the closest feature point. Boundaries are defined in locations where x is equidistant to more than one feature point. These boundaries describe exactly the Voronoi cell boundaries explained in Chapter 2.4.2. Furthermore functions F2(x), F3(x),...Fn(x), are defined to build even more

interesting textures (n stands for n-closest feature points). Consequently, F2(x) gives the distance between x and the closest and second-closest feature point shown in Figure 2.28. More variations are produced by linear combination of Fn(x) by the following formula:

C1F1 + C2F2 + C3F3 + C4F4

Due to higher n start looking similar, they choose the lower n values (up to 4), which look more interesting and distinct. In Figure 2.29 some combinations with various values of Cn are shown.

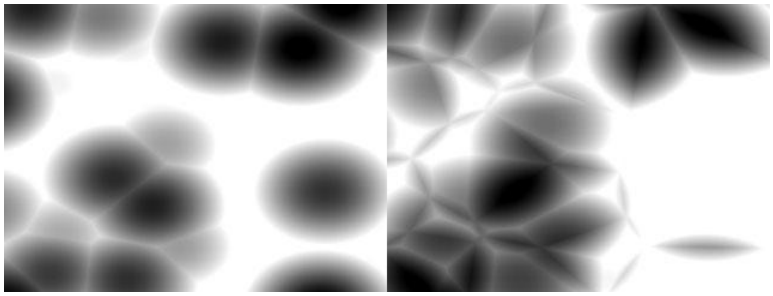

Figure 2.28: *F1(x) producing polka dots (left) and F2(x) has rapid changes and internal structure (right) [21]*

2.4.4 Feature-Based Cellular Texturing

Feature-based cellular texturing [27] is a method especially designed for architectural models. The architectural model is decomposed into two components. Firstly, the basic shape, which provides the rough representation and secondly the cellular texture containing repetitive 3D pattern, such as a brick wall pattern. Three kinds of features faces, edges and corners are identified in the basic mesh (see Figure 2.30). Then a fitting algorithm applies cells onto the three types of features by using a occupancy map. This map is a bit mask that holds information about areas are already occupied and areas responsible for filling. In Figure 2.30 you can see a basic mesh with identified features and applied cellular texture.

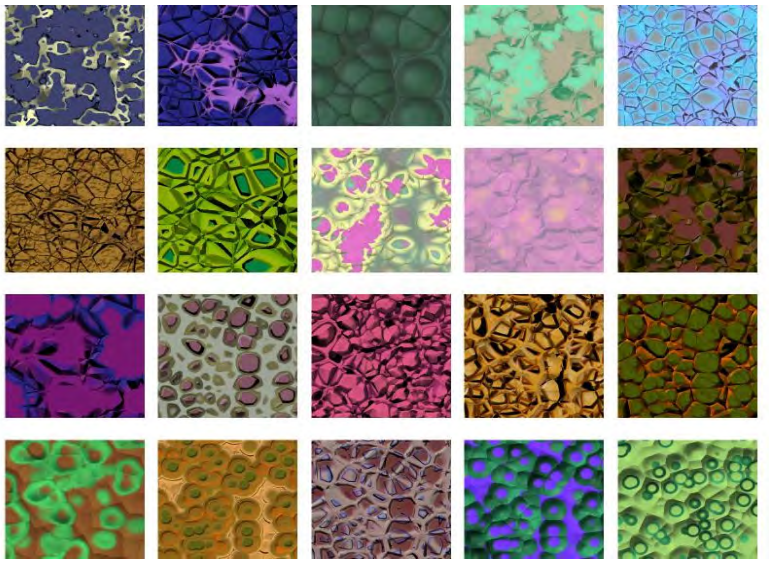

Figure 2.29: *Some interesting linear combinations of C1F1 + C2F2 + C3F3 + C4F4. [19]*

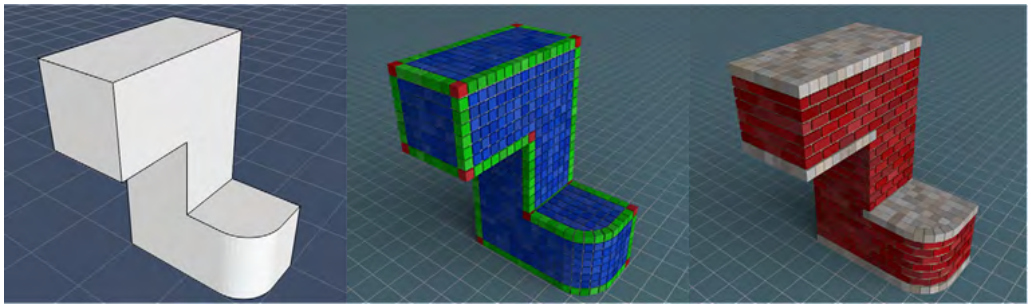

Figure 2.30: *Basic mesh (left), identified features faces (blue), edges (green) and corners (red) for the basic mesh (middle) and basic mesh applied with brick wall pattern (right) [27]*

2.4.5 Solid Texturing

Texture mapping is usually done in two-dimensional space. However, solid texturing [18] uses texture functions defined throughout a region of three-dimensional space. That is this method must not concern about the shape of the surface being textured. Generally, the solid texture function gets evaluated procedurally by the object-space coordinates. Hence we not only get color information on the surface of an object but also in the inside. The most known example for solid texturing is to synthesize wood by the solid texture function shown in Figure 2.31.

Figure 2.31: *Three-dimensional solid wood texture. [28]*

„It's not an idea until you write it down.„

Ivan Sutherland

Chapter 3

Cell-Based Object Representation

This chapter presents a cell data structure to model fine scale details of an object with a repetitive surface structure for a real-time application. Furthermore it shows how to display a cell-based object by using a ray tracer. First, some precomputations, such as tiling and generation of cell membership information, has to be done before rendering explained in Chapter 3.1. In the rendering stage at runtime a ray-tracing algorithm is performed, which uses the precomputed information to render the fine scaled details described in Chapter 3.2.

3.1 Preprocessing Stage

In this chapter the data structure for a cell-based object is presented. The data structure contains important information, which are precomputed in an offline process. This precomputed information makes the cell-based rendering efficient.

3.1.1 Basic Mesh

First of all a rough representation of the cell-based object is defined by a polygonal mesh. In the following the rough representation will be called the "*basic mesh*". A basic mesh has two main functionalities. In the preprocessing stage the basic mesh acts as a bounding area for modeling the fine scale details. In Figure 3.1 (c) the fine scale details, represented by the colored dots are enclosed by a basic mesh of a torus (blue circular lines). Secondly, the basic mesh acts as an entry point for ray tracing in the rendering stage, which gets explained more precisely in Chapter 3.2.

3.1.2 Voxel-Based Cells

After defining the area of modeling by a basic mesh, the inner area has to be described somehow. Therefore a three-dimensional, cellular component is specified, a so-called "*cell*". A cell is a building block, which is needed for modeling the fine scale details of the cell-based object. More precisely a cell is defined by a set of connected voxels shown in Figure 3.2. Consequently the set of all inside cells make up the final cell-based object. This constitutive separation into a basic mesh and corresponding inside cells can be found in the preprocessing stage in Figure 3.1 (c) . Note that all inside cells (colored dots) are completely inside the basic mesh (blue silhouette of the torus) not touching the basic mesh's boundary.

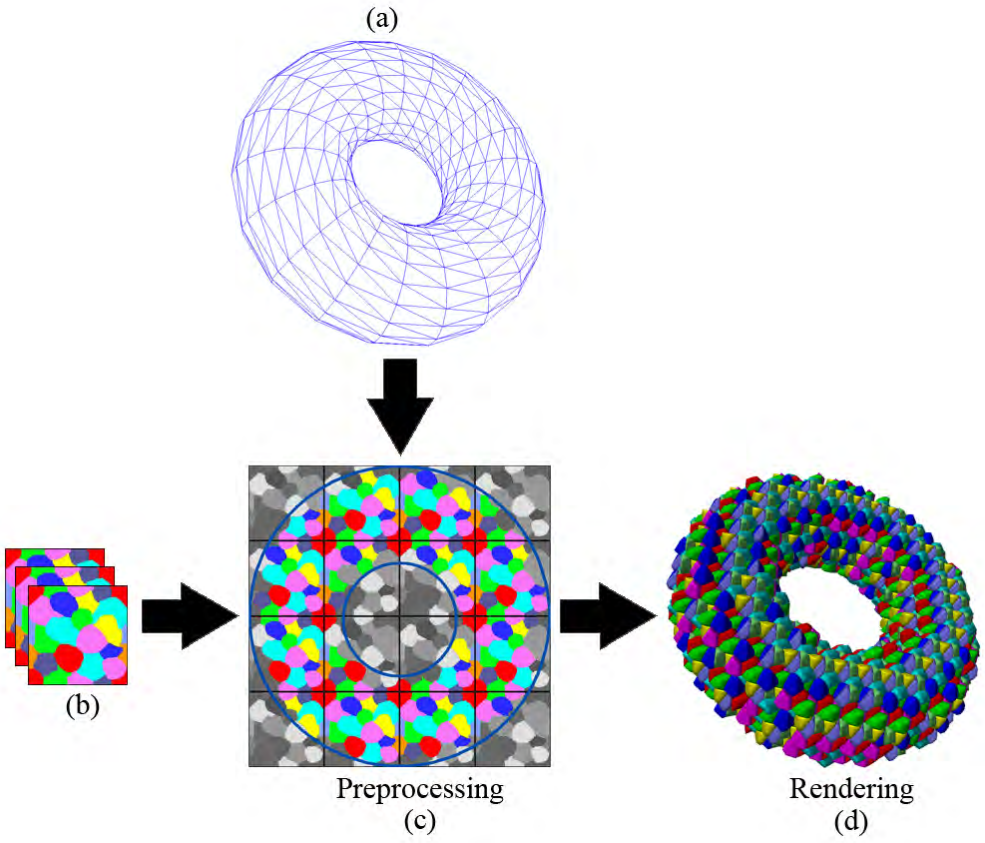

Figure 3.1: *Three-dimensional cell collection (b) gets tiled over the object-space (c). The blue silhouette of the torus represents the basic mesh. The basic mesh is also shown in (a) by a wireframed polygonal mesh. The colored dots represent the inside cells. Rendering results to a torus with fine scale detail surface (d).*

3.1.3 Cell Tiles

To model the fine scale details by placing every single cell in the bounded three-dimensional inner area of the basic mesh is a tedious task. Furthermore this type of modeling requires a huge amount of memory storing each cell separately. A better solution is to define a small amount of cells in such a way, so that a repetitive tiling over the whole object-space is possible. This approach has the advantage, that a tileable cell collection has to be modeled only once. Furthermore it avoids huge memory consumption. The idea of tiling is done by dividing the three-dimensional object-space into a grid of uniformly spaced cubes as we can see in Figure 3.1 (c) . In the following these cubes will be called "*tiles*". Each tile contains the same cell collection (seen in Figure 3.1 (b)). To have a smooth transition between each tile border the cell collection has to be tileable in all six directions. That is a cell, which ends on the one side of the cube, continues on the opposite side. Since cells are described by voxels a cell collection of a tile can be stored in a three-dimensional texture shown in Figure 3.2. This three-dimensional texture will be called the "*tile map*".

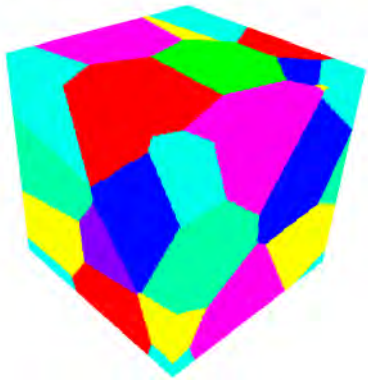

Figure 3.2: *A three-dimensional texture of a tile's cell collection, which contains eight tileable cells. For example the red cell gets tiled over the edges.*

3.1.4 Representing Objects as Voxel Cell Tiles

Since the cells are defined by a set of connected voxels and the fact that cells have to be tested against the basic mesh to determine the cell membership (explained in Chapter 3.1.5) a voxelization of the basic mesh is indispensable. Voxelization is concerned with converting geometric objects from their continuous geometric representation into a set of voxels that best approximates the continuous object [29]. In the preprocessing stage the basic mesh is voxelized to determine which parts of the basic mesh are inside or outside respectively. The idea is to render the basic mesh slice by slice with an orthogonal projection [22]. The clue is to set the near clipping plane to the slice depth and the far clipping plane to infinity as we can see in Figure 3.3. Additionally the stencil operation for front faces is set to decrement and for the back faces to increment. After rendering, in each slice nonzero values represent the area inside the basic mesh. This information is stored voxel by voxel. The whole inside-outside voxelization can be done on the GPU even in real-time. This approach works only with water-tight closed meshes.

3.1.5 Cell Membership Determination

After tiling the tiles(cell collections) all over the object-space it has to be determined which cells are totally inside, totally outside and intersected by the basic mesh. To determine the cell membership for a cell, every cell in the object-space has to be tested against the basic mesh voxel by voxel. If there is an intersection between a cell and the basic mesh's boundary the tested cell is tagged as "*intersected*" (shown in Figure 3.4 by the red colored cells). That is if only one voxel of a cell intersects with the basic mesh, the whole tested cell is tagged as intersected. Accordingly, cells totally inside or outside (meaning all containing voxels of the cell), i.e. the basic mesh, not touching the basic mesh's boundary are tagged as "*inside*" (green colored cells in Figure 3.4) or "*outside*" (gray colored cells in Figure 3.4), respectively. At the end of this testing algorithm each cell in object-space is tagged with a membership stored in the "*cell membership map*".

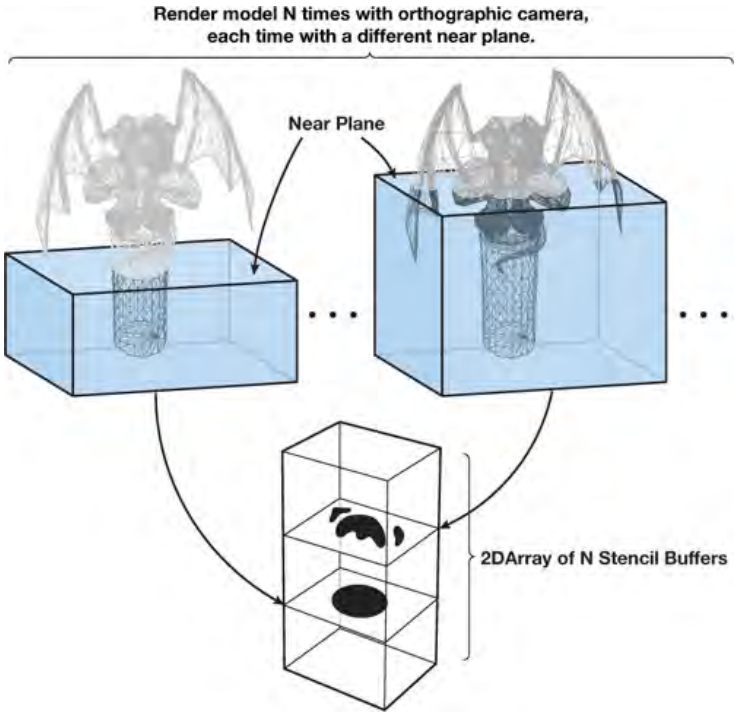

Figure 3.3: *Inside-outside voxelization of a polygonal mesh of a gargoyle. [22]*

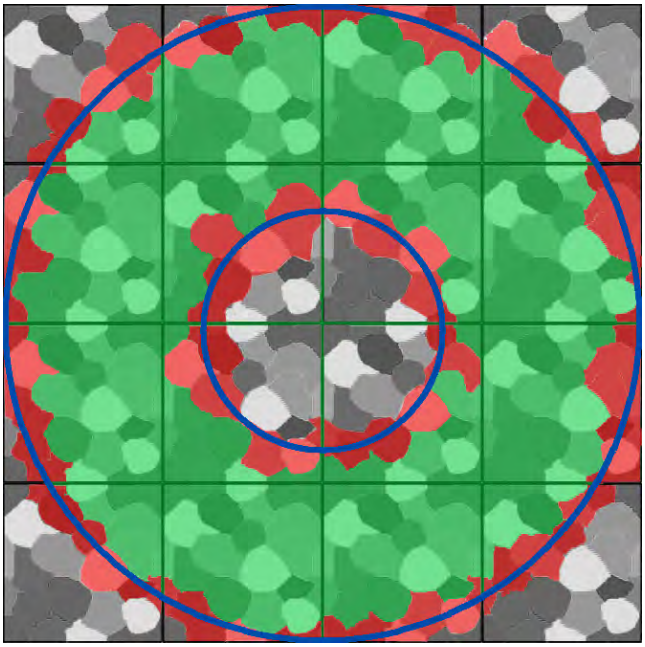

Figure 3.4: *Cell membership definition: inside cells (green cells), outside cells (gray cells) and intersected cells(red cells). The blue line represents the basic mesh's boundary.*

3.1.6 Cell Membership Map Indexing

Since the cell membership map has a spatial structure based on tiles, an efficient indexing is required to localize a cell in a tile. The first step for cell localization is to define a "*tile index*", which describes a specific tile position in the uniformly divided object space. Due to the known tile size it is very easy to determine the tile index. Since a tile is specified, a second index, the so-called "*cell index*", has to be defined to finally localize a cell. If we take a look at Figure 3.5 on the left we can see a tile containing four cells. Note that the blue cell gets tiled over the corners, the green and magenta cells are tiled over the edges and the red cell lies totally inside the tile. However, for localizing each cell in this tile, four cell indices are insufficient. Therefore the cell index range has to be expanded by splitting each tiling cell into "*split cells*". That is in the given tile example in Figure 3.5 the blue cell is split into four split cells. On the other hand the green and magenta cells are split into two split cells. Note the new indexing of each split cell in Figure 3.5 on the right.

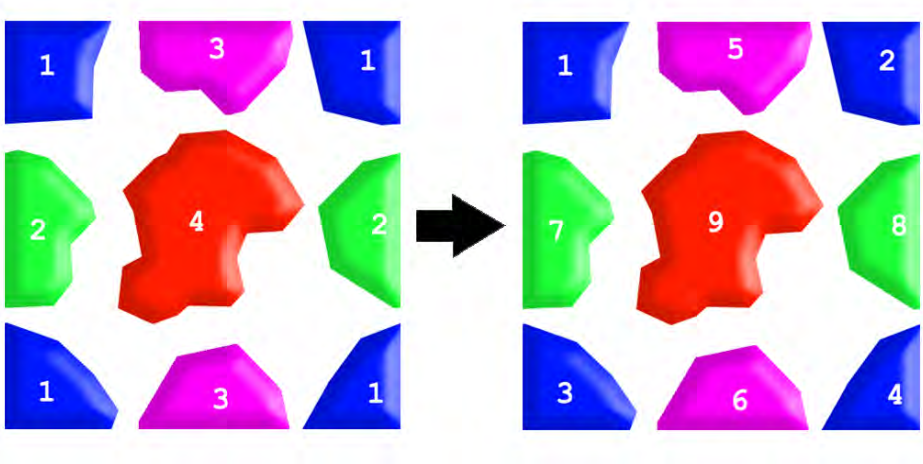

Figure 3.5: *Tile containing four cells on the left. Each tileable cell gets split into split cells. Due to cell splitting a new indexing could be found on the right.*

In Table 3.1 the cell-to-split-cell connectivity for the tile example in Figure 3.5 is shown. Note that after cell splitting the cell index range changes from four to nine. With this important splitting step it is possible to localize every cell in the cell membership map by the tile and cell index.

Cell index	Split cell index
1	1,2,3,4
2	5,6
3	7,8
4	9

Table 3.1: *Cell index before cell splitting (left column). Corresponding split cell indices for each tiled cell (right column).*

3.1.7 Color Map

As we have seen in Chapter 2.3 every mesostructure rendering technique needs an input for the rendering process, mostly a two-dimensional color map. However cell-based rendering requires a three-dimensional color map. That is to say that cell-based rendering also handles color information inside the object such as in solid texturing explained in Chapter 2.4.5. To accomplish this feature a mapping between a three-dimensional point in object space and texture space is done. So even if we cut through the object we get color information also on the cutting plane. Furthermore if the color map contains transparency information and by the fact that cell-based rendering uses a ray tracer for rendering, it is easy to render refraction and translucency by shooting further rays in the pixel shader.
Since a tile has only a description of a cell collection and containing no color information, a three-dimensional color map has to be created additionally. It is up to the artist to inherit the cellular structure of the tile in his three-dimensional color map. In doing so the color map has to be also tileable in all directions to get nice results for rendering. In Figure 3.6 a three-dimensional color map (e) is applied as input for cell-based rendering, which results into a more interesting looking torus as we have seen in Figure 3.1 (d). Figure 3.7 shows further results with other color maps as input. Note that no shading is applied on these results.

3.1.8 Normal Map

Shading makes an object more interesting and realistic looking. Therefore a precomputation of normals is indispensable. Generally in volume rendering gradients serve as normals for shading calculation. Often gradients are precomputed (or even in realtime) by the central difference [24]. Since a cell-based object uses three-dimensional textures a modified gradient precomputation has to be done. Instead calculating the normals in non-homogenous areas such as in volume rendering our normals gets precomputed on the cell's boundary. A detailed implementation description is explained in chapter 4.7. In Figure 3.8 a color (a) and a normal map (b) is applied, which results into a colored and phong shaded look (d). In Figure 3.9 a comparison between an unshaded (left) and a phong shaded (right) cell-based torus is shown.

3.1.9 Conclusion

This proposed cell-based object representation is especially designed for repetitive surface details. With this representation it is possible to avoid huge memory consumption by precomputing a cell membership map. The cell membership map contains all inside cells, from which the cell-based object is build. Furthermore the cell membership map offers an easy indexing for localizing each cell in object space, which can be very helpful for rendering as explained in Chapter 3.2.

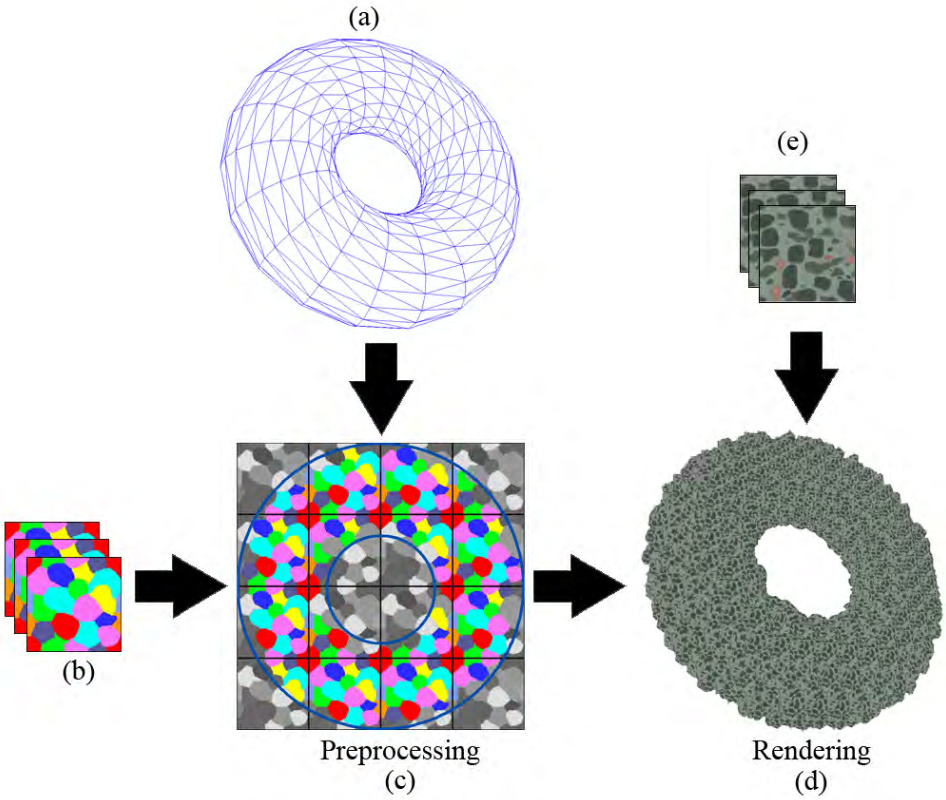

Figure 3.6: *Cell-based rendering with a color map [23] (e) as input. Note that the final result (d) has no shading applied.*

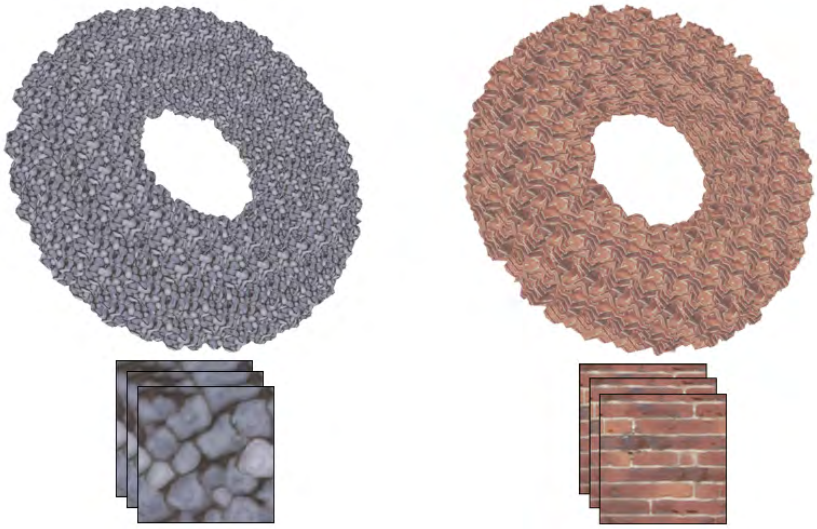

Figure 3.7: *Cell-based rendering with further color maps [23] (bottom) as input. Note that the results (top) have no shading applied.*

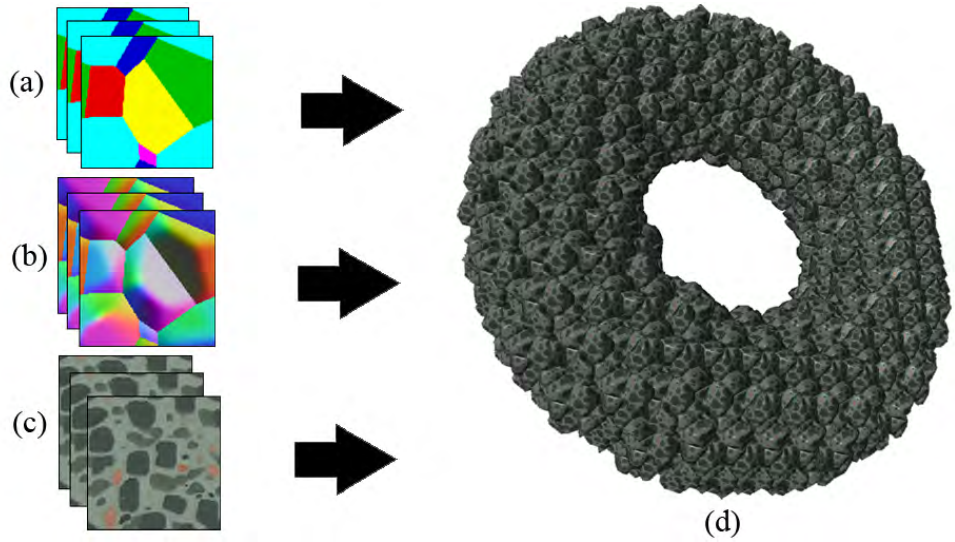

Figure 3.8: *Cell-based Rendering with a specific cell collection (a) with corresponding normal map (b) and color map [23] (c) as input. A phong shading is applied to the cell-based torus object (d).*

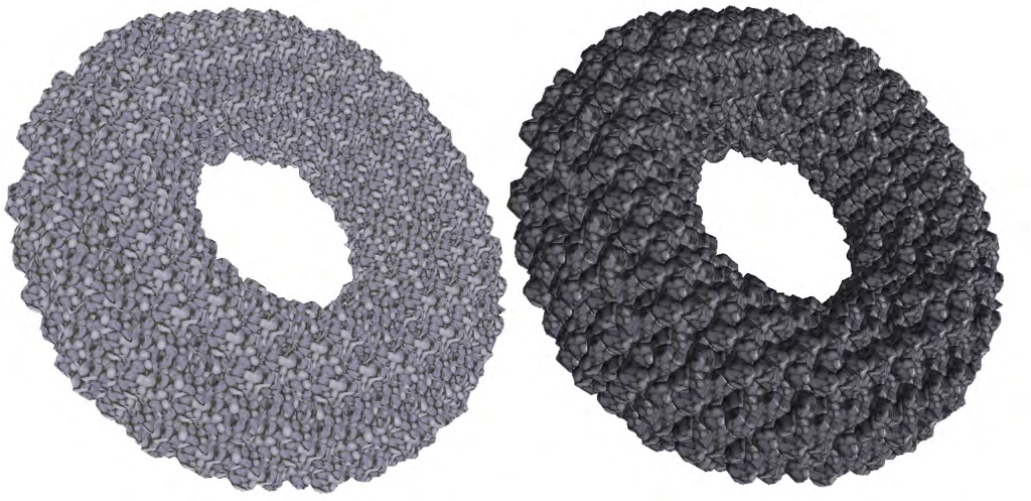

Figure 3.9: *Comparison between an unshaded cell-based torus (left) and a phong shaded, cell-based torus object (right). Both objects are colored with the stone color map from [23] Figure 3.7.*

3.2 Rendering Stage

In this chapter a ray tracing-based rendering method is proposed for rendering the cell-based object. The ray tracer uses the precomputed cell membership map (explained in Chapter 3.1), which makes rendering very efficient. Even self reflection and refraction can be easily computed by shooting further rays. The main goal of the ray tracer is to find the inside cell's boundary, which represent the fine scale details of the cell-based object.

3.2.1 Ray Tracer

This proposed rendering process is based on a ray tracing algorithm. The rendering needs the following four three-dimensional textures as input: tile map, cell membership map, color map and normal map. The tile map and cell membership map are required to localize a cell in object space and to determine its cell membership. After finding an inside cell's boundary a local illumination calculation is done which needs the color map and normal map as input. The fifth and last input for rendering is the basic mesh. By rendering the basic mesh (the rough representation of our cell-based object), the object's surface is the entry point for the ray tracing algorithm. The idea is to shoot rays from the object's surface in eye direction to hit an inside cell shown in Figure 3.10 called the "*first hit*". To find an intersection of an inside cell two simple search algorithms have to be performed. The following piece of pseudo code shows a first hit by performing a linear search with a subsequent binary search to find the inside cell's boundary:

```
1 for each pixel shoot ray in eye direction
2
3       entry point is set to the basic mesh's surface
4
5       //linear search
6       while (cellMembership == INTERSECTED)
7               stepsize along ray
8               determine cell membership
9               if (cell membership == INSIDE)
10                      store hit position
11                      break while loop
12              end
13      end
14      if (cell membership == OUTSIDE)
15              discard ray
```

```
16      end
17
18      //binary search
19      start binary search from last hit position
20      for each binary step
21              determine cell membership
22              half step size
23              if (cell membership == INSIDE)
24                      store hit position

25                      step two times backwards along ray
26              end
27              step one step forward along ray
28      end
29 end
```

Algorithm 3.1: *First hit performed by a linear search (row 6-13) with a subsequent binary search (row19 - 28) to find the inside cell's boundary.*

3.2.2 First Hit

First of all a linear search is performed to find an inside cell (shown in row 6-13 in Algorithm 3.1). As we can see in Figure 3.10 the ray entry is shown by a cyan colored dot labeled with an "E". Note that the entry point is inside an intersection cell, which is no coincidence. The fact that cells touching the object's surface is per definition an intersected cell. In the following an intersection cell, which contains the entry point will be called the "*entry cell*". The linear search is performed until an inside cell is found (green cell in Figure 3.10). In every iteration (represented by the yellow colored dots labeled with "L1" and "L2" in Figure 3.10) a constant sized step is done. The initial step size of the linear search highly depends on the angle between the shooting ray and the basic mesh's surface normal. This rule is needed to avoid artifacts by more grazing viewing angles [15]. This happens because the step size is too high and therefore missing an inside cell.
It is not always the case (as in Figure 3.10) that right after the entry cell an inside cell is followed. Especially at grazing angles often the ray steps through more than one intersection cell until it hits an inside cell shown in Figure 3.11.
But it is not guaranteed that exactly the border of the inside cell was hit. Therefore the binary search has to be performed (shown in row 19-28 in Algorithm 3.1). The idea is to half the step size with every iteration as long as the step size is smaller than a certain threshold(ideally the size of a voxel) shown in Figure 3.12. As we can see in Figure 3.12 the binary search algorithm starts at

the last position of the linear search, which has to be inside an inside cell(the cyan colored dot labeled with a “L”). In the example in Figure 3.12 we only need three iterations to hit exactly the inside cell's border(yellow colored dot labeled with a “B3”).

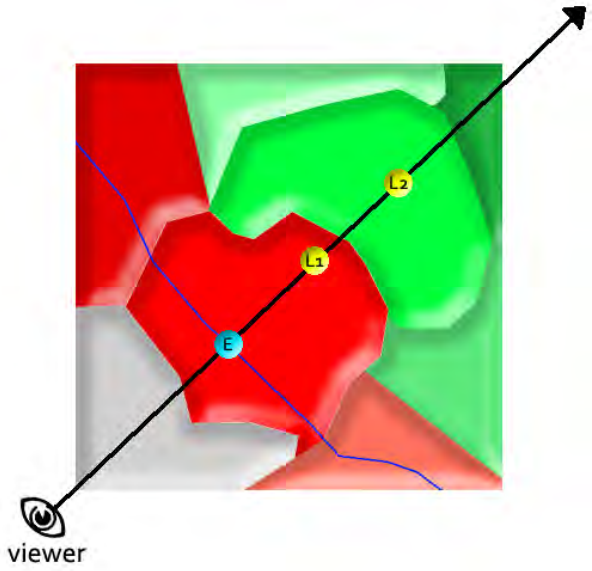

Figure 3.10: *Ray tracing performing a linear search to find an inside cell (green cells). Intersected cells are colored red and outside cells are colored gray. The blue line represents the basic mesh's surface. “E” is the entry point and “L1,”L2” are the equidistant steps along the ray.*

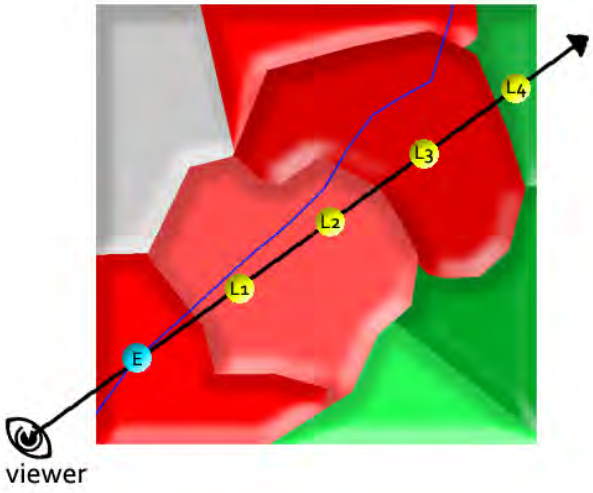

Figure 3.11: *At grazing angles the ray steps through more than one intersected cell (red cells). Four equidistant steps (yellow colored dots) are required to hit an inside cell (green cell).*

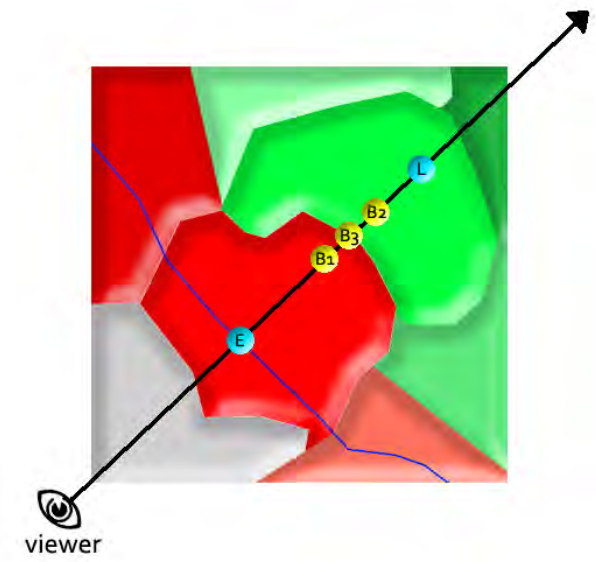

Figure 3.12: *Ray tracing performing a binary search to find the cell's boundary. Inside cells are colored green, intersected cells are colored red and outside cells are colored gray. The blue line represents the basic mesh's surface. Three iterations B1,B2 and B3 are needed to hit exactly the inside cell's border.*

Generally for the linear search the termination criteria is to hit an inside cell no matter if this inside cell is an adjacent neighbor of the intersection cell. After finding an inside cell the search algorithm switches to a binary search to finally hit the cell's boundary for illumination computation. But what happens if the ray hits an outside cell? If we take a look at Figure 3.13 we can see that the first ray hits an outside cell meaning the ray is outside the object's surface. To finally find an inside cell the ray had to step through all the outside cells until it hits the object's surface again. Due to basic mesh rendering we get entry points inside an intersected cell. This fact helps us to skip all over the outside cells such as in Figure 3.13 and immediately starting at the entry point right after the outside cells shown by the second entry point in Figure 3.13. This approach saves some computational time. In fact the idea is not to skip the outside cells, but discarding the first ray entirely (marked by the red “X” for ray 1 in Figure 3.13) if an outside cell is hit. Due to basic mesh rendering a further ray right after the outside cells (ray 2 in Figure 3.13) is shot from the object's surface pointing in the same viewing direction. To discard a ray means the search algorithm can be entirely terminated by not performing a subsequent binary search. In this case no inside cell could be hit and therefore the color information gets entirely discarded meaning that ray misses the object's surface. By finding an inside cell's boundary only a simple lookup in the precomputed normal and corresponding color map has to be done to finally evaluate the local illumination.

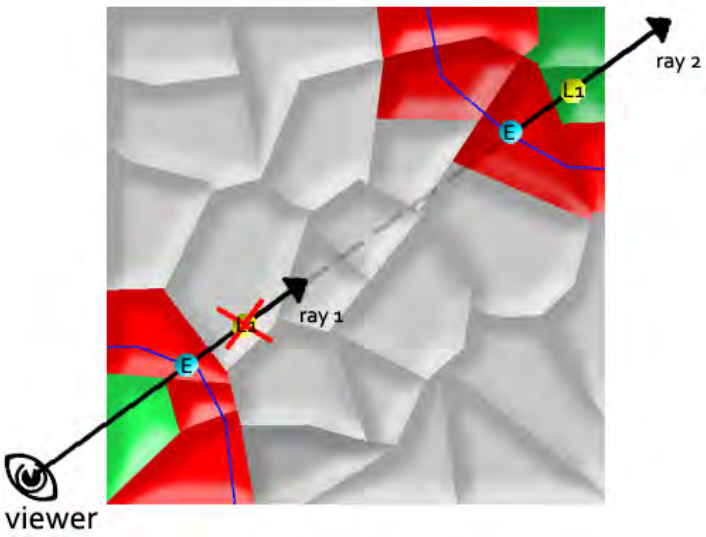

Figure 3.13: *Ray tracing performing a linear search. Since the first ray hits an outside cell (gray cells) ray 1 gets entirely discarded (shown by the red "X"). Due to basic mesh rendering a further ray (ray 2) is shot pointing in the same viewing direction. Ray 2 hits an inside cell (green cell).*

3.2.3 Specular Self-Reflection

After finding the inside cell's boundary performing a first hit (explained in Chapter 3.2.2) specular self-reflection can be easily computed by shooting further rays (reflected rays R1 and R2 shown in Figure 3.14). The law of reflection says that for specular reflection the direction of the incoming light (the incident ray) and the direction of outgoing light reflected (the reflected ray) make the same angle with respect to the surface normal. In Algorithm 3.2 a self reflection is explained. As for the first hit (explained in Chapter 3.2.2) a linear search has to be performed to find an inside cell. The main difference between the first hit and the specular self-reflection is that a linear search (shown in row 4-17 in Algorithm 3.2) is performed mostly outside the cell-based object instead inside. This fact provides an optimization by doing a "*tile skipping*" for empty tiles (shown in row 11-16 in Algorithm 3.2). Due to the precomputed membership map, which contains the membership of each cell in a tile, we can easily check if a whole tile contains solely outside cells. In case the ray intersects an empty tile (magenta highlighted square in Figure 3.15) only a box vs ray intersections has to be computed to determine the exit point (yellow colored dot labeled with a "B" in Figure 3.15) along the shooting ray. After finding an inside cell a binary search (the same as in Algorithm 3.1 row 19-28) is

performed to find the cell's boundary. In case no inside cell is hit by the linear search the ray gets entirely discarded (19-20 in Algorithm 3.2).
Specular self-reflection can have more than one reflection bounce. Since every further reflection bounce leads to more computational costs, in practice only one or maybe even two reflection bounces are sufficient to have a realistic looking specular self-reflection. More reflection bounces don't contribute so much details in the final rendering so that a higher order ray would be worth the effort. In Figure 3.16 we can see a comparison of a cell-based torus object rendered with higher order rays using an environment map. It shows that the difference between ray order one and higher orders is hardly visible.

```
1 for each specular reflection bounce shoot ray in reflection direction
2
3       determine reflection direction
4       while (cellMembership != INSIDE) //linear search
5               stepsize along ray
6               determine cell membership
7               if (cell membership == INSIDE)
8                       store hit position
9                       break while loop
10              end
11              if (cell membership == OUTSIDE) //empty tile skipping
12                      if (tile is empty)
13                              box(tile) vs ray intersection to determine exit point
14                              hit position = exit point
15                      end
16              end
17      end
18
19      if (linear search found no hit)
20              discard ray
21
22      perform binary search
23 end
```

Algorithm 3.2: *Specular self-reflection performed by a linear search (row 4-17) with a subsequent binary search (row 22) to find the inside cell's boundary. For optimizing purposes an empty tile skipping (row 11-16) is done for the linear search.*

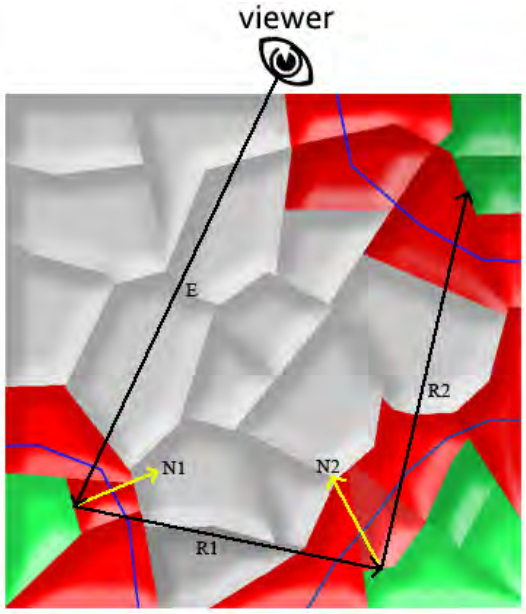

Figure 3.14: *Specular self-reflection with two bounces. E is the viewing direction, which is the incident ray for the first bounce. R1 is the corresponding reflected ray. In this example an additional bounce is performed. R1 is the incident and R2 the reflected ray, respectively. N1 and N2 are the surface normals.*

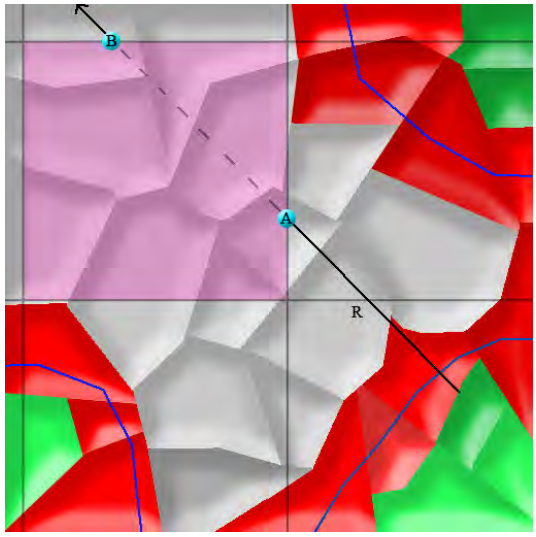

Figure 3.15: *Empty tile skipping is done by computing a box(tile) vs ray intersection to determine the exit point "B"."A" is the corresponding entry point. The magenta colored square represents the empty tile, which contains solely outside cells.*

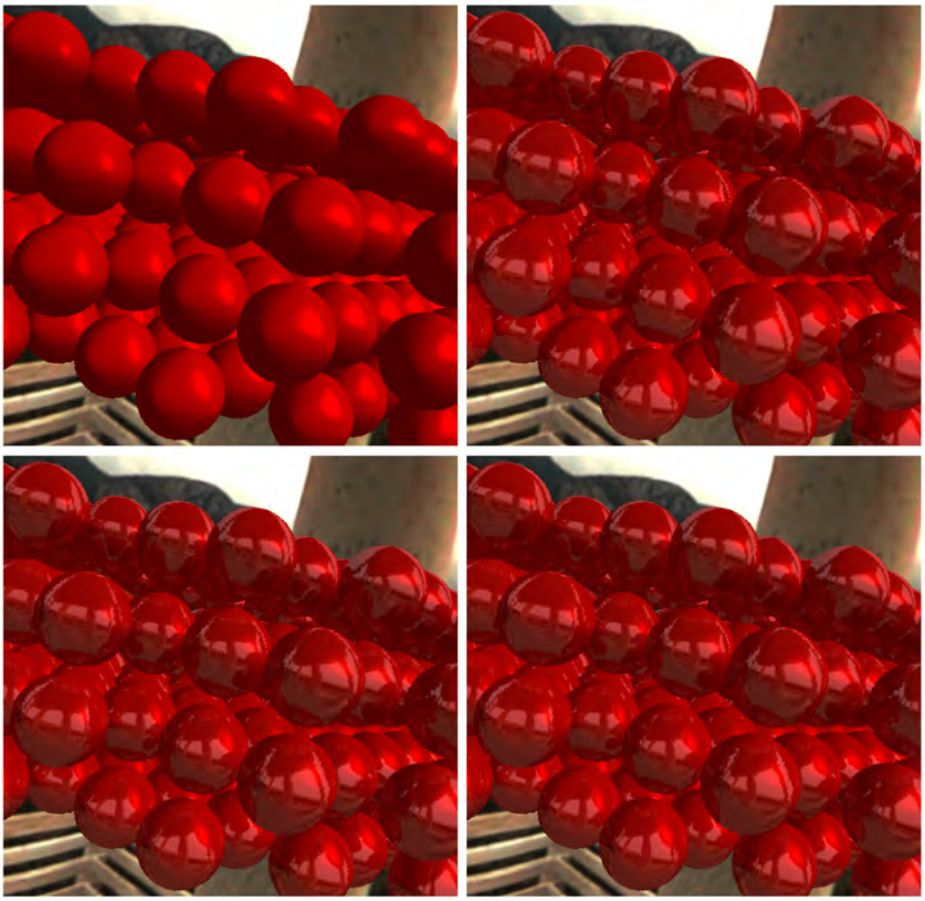

Figure 3.16: *A close-up view of a cell-based torus rendered with specular self-reflection and environment mapping. Traversing higher order reflection ray: zero (top left), one (top right), two (bottom left) and three (bottom right).*

3.2.4 Refraction

Refraction can be easily computed just like specular self-reflection explained in Chapter 3.2.3 by shooting further rays right after the first hit. Instead of shooting further rays in reflection direction this time a refraction direction is computed by Snell's law. Snell's law is a formula used to describe the relationship between the angle of incidence θ_1 and refraction θ_2:

$$\sin\theta_2 = \frac{n_1}{n_2}\sin\theta_1$$

When light penetrates from one media into another the direction of the incident light gets bend. The amount of bending the light direction depends on the media refractive index n_1 and n_2 .

In Figure 3.17 the first refraction R1 occurs when the viewing ray E penetrates into the cell-based object. Ray R1 travels inside the object until it leaves the object by hitting the inside cell's boundary. To determine the inside cell's boundary efficiently a modified linear search has to be performed as we have seen for the specular reflection. Due to ray marching is done in the inside (instead outside) the object the termination criteria for the cell membership has to be changed (as we can see in Algorithm 3.3 row 4 and row 7). Hence a tile skipping can be also done for tiles, which contain solely inside cells (shown in Algorithm 3.3 row 12-15). After the ray hits the cell-based object's boundary two possible refractions are possible. First, the ray can be refracted in such a way so that the ray travels again in the inside of the cell-based object shown by R2 in Figure 3.17. In this case the same search algorithm (row 4-17 in Algorithm 3.3) as mentioned before is performed due to ray marching is done inside the cell-based object. The second possibility is that the ray leaves the cell-based object (ray R3 in Figure 3.17) so that a ray marching is performed at the outside of the cell-based object. This algorithm (Algorithm 3.2 row 4-22) was mentioned before in the specular self-reflection explained in Chapter 3.2.3. Refraction is repeated until no cell-based object's surface can be hit anymore. At the end of this refraction process we get a ray direction, which could be used to lookup into an environment map (just like for specular self-reflection). Results of refraction with an environment map can be seen in Figure 3.18.

```
1 for each refraction bounce shoot ray in refraction direction
2       determine refraction direction
3       if (next step along ray == INSIDE)
4               while (cellMembership == INSIDE) //linear search
5                       stepsize along ray
6                       determine cell membership
7                       if (cell membership == INTERSECTED)
8                               stepsize backwards along ray
9                               store hit position
10                              break while loop
11                      end
12                      if (tile contains solely inside cells) //tile skipping
13                              box(tile) vs ray intersection to determine exit point
14                              store exit point as hit position
15                      end
16              end
17              perform binary search
```

```
18      end
19      else //same search algorithm just like for specular self-reflection
20              (see row 4-22 in Algorithm 3.2)
21      end
22 end
```

Algorithm 3.3: *Refraction performed by a linear search (row 4-16) in the inside of the cell-based object with a subsequent binary search (row 17) to find the inside cell's boundary. For optimizing purposes an tile skipping (row 12-15) for tiles containing solely inside cells is done for the linear search. Search algorithm for refraction rays outside the cell-based object are explained in Algorithm 3.2 in row4-22.*

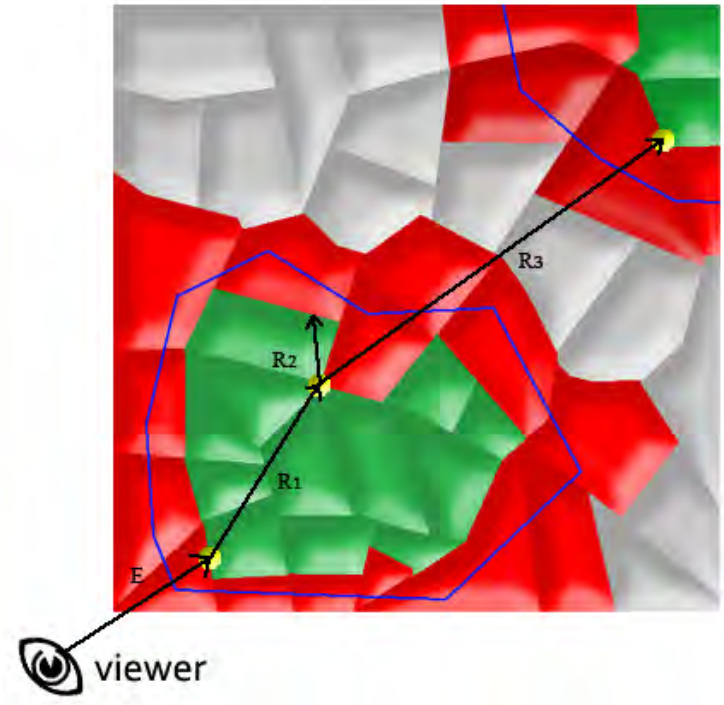

Figure 3.17: *Refraction with two possible ray paths R2 and R3. E represents the viewing vector, which penetrates into the cell-based object. At the first hit position (yellow dot) a refraction vector R1 is shot. At the second hit position two possible refraction vectors are shown. R2 refracts into the cell-based object. R3 leaves the cell-based object.*

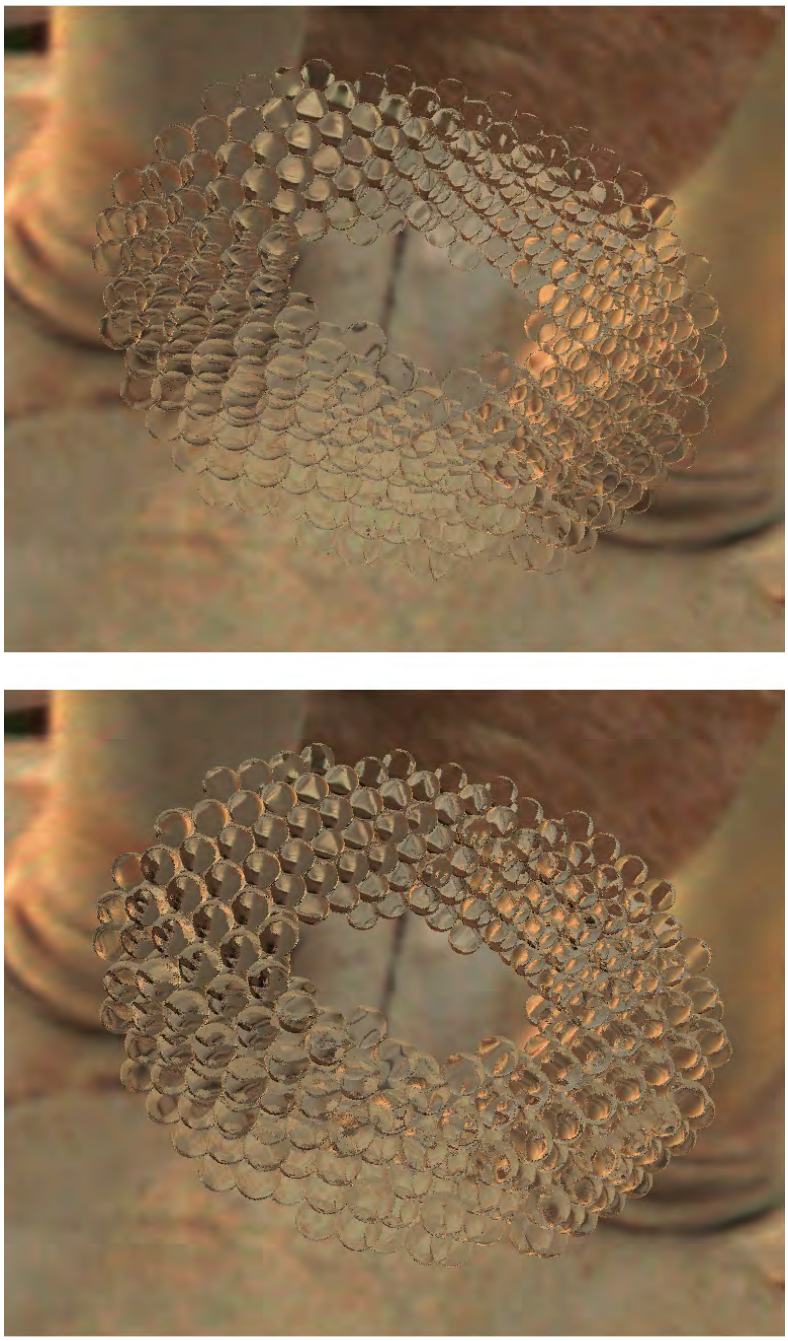

Figure 3.18: *A cell-based torus rendered with refraction. In this rendering an environment map is used. The image above is rendered with a lower refraction index than the image at the bottom.*

3.3 Summary

In summary in the preprocessing stage the object-space is split into tiles. Each tile contains the same tileable cell collection. After precomputation each cell in object-space is tagged with one of the following cell membership information: inside, outside or intersected and stored in the cell membership map. Localization of a cell is done by two indices, the tile index and the cell index. Especially for tileable cells a cell splitting has to be done, which results to a new indexing. The color map has to be tileable to get nice results in the rendering. The normal map is computed by calculating edge normals on the cell's boundary. In the rendering stage a ray-tracer is performed using a linear with a subsequent binary search to hit an inside cell's boundary to calculate a local illumination. Additionally features such as specular self reflection and refraction can be easily computed by shooting further rays. For ray tracing some optimizations such as discarding viewing rays and empty tile skipping are discussed to improve the rendering speed.

„An algorithm must be seen to be believed„

Donald Knuth

Chapter 4

Implementation

With the fundamental understanding of the basic concept of a cell-based object representation described in Chapter 3 we can analyze the preprocessing stage from a technical point of view. During the preprocessing stage the following textures are precomputed: tile map, membership map and normal map. In this chapter each texture generation is explained in detail. In Chapter 4.1 we will discuss various ways to generate a tilemap. In Chapter 4.2 and 4.3 some index modifications to the tile map has to be done to meet certain requirements for the next precomputational stage the cell membership determination, which is explained in Chapter 4.4 and 4.5. Chapter 4.6 explains how to generate the corresponding normal map. The last Chapter 4.7 presents three tools to model and render the cell-based objects.

4.1 Tile Map

In Chapter 3.1.3 we defined the term cell tile, which contains a collection of cells stored in a three-dimensional texture called the tile map. This cell tile gets tiled over the object space to create the fine scaled details on a cell-based object. The basic idea is to create a three-dimensional texture, which contains cell indices. Each cell is defined by an unsigned integer. In Figure 3.2 we can see a tile map containing eight cells meaning that the cell index range goes from 1 to 8. Note that some cells get tiled over the corners and edges. Since the tile map in Figure 3.2 is based on a Voronoi diagram (explained in Chapter 2.4.2) each cell has a immediate adjacent cell. That is no empty space is defined in this texture. However it is also possible to create a tile map containing empty space defined by the cell index 0. In Figure 4.1 we can see a tile map with empty space defined by the black area. In this tile map we created spheres with a certain radius. Each sphere represents a cell in the tile map. In Figure 4.2 a slice of a three-dimensional tile map is shown containing four cells. Each pixel in this slice is tagged with an unsigned integer representing a cell index (index range from 1 to 4) or emtpy space (index 0).

Defining each cell by hand could be a tedious task because every slice of the three-dimensional tile map has to be created by hand with a painting tool. To make life easier the tile map should be created automatically. For testing purpose we choose a Voronoi diagram (explained in Chapter 2.4.2) for generating a tile map. We randomly set some seed points in three-dimensional space and generated a three-dimensional Voronoi diagram given by Euclidean distance. In Figure 3.2 we can see a computed tile map out of eight randomly set seed points This tile map has a resolution of 64x64x64 voxels. Instead of the Euclidean distance we can use another distance function, which produces various interesting Voronoi diagrams as we have seen in Figure 2.27. Certainly there are many procedural texture generation methods, which could be used to create a tile map. Some of them we have been described in the related work in Chapter 2.4. There are some practical methods for texture design [20] to build procedural textures.

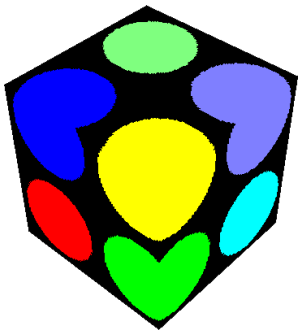

Figure 4.1: *A tile map containing eight spherical cells (colored dots). The black area represents the empty space.*

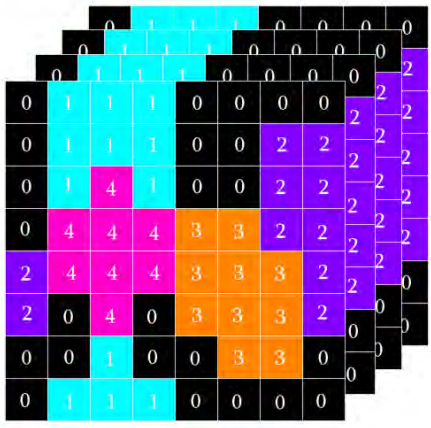

Figure 4.2: *Tile map slices tagged with a cell index 1 to 4. Index 0 represents empty space.*

4.2 Cell Splitting

Assuming that we have chosen a procedural method for building a tile map, such as a Voronoi diagram, we have to do some modifications to the tileable cells itself. In such a map there are some cells, which fit entirely into the tile map (magenta colored cell tagged with the cell index 4 in Figure 4.2) and some other cells, which get tiled over the tile map's boundaries (the purple colored cell tagged with cell index 2 in Figure 4.2). Every split cell in this tile has to be locatable by a split cell index as explained in Chapter 3.1.6. If the tiled cell is tagged with the same cell index, we have to clip this cell on the tile map's boundary and assign each split part of the tileable cell with a new cell index somehow. In the three dimensional space a cell can be split up to eight split cells. Producing eight split cells is only possible if and only if the cell gets tiled over the corners as we can see for the cyan colored cell in Figure 3.2. Accordingly we need for every tileable cell a placeholder for eight possible split cells.

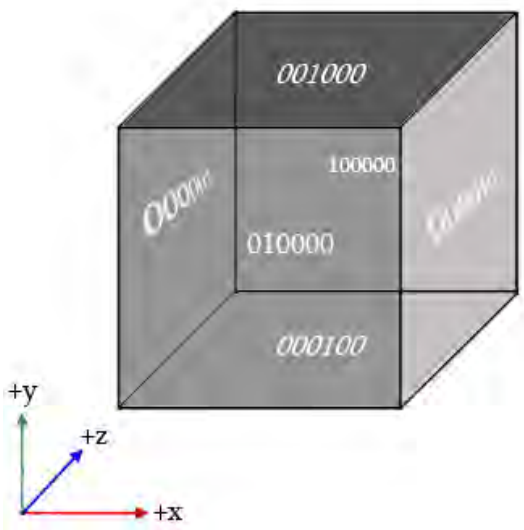

Figure 4.3: *A three-dimensional texture's boundary tagged with a six bit binary code.*

If we take the example from Figure 4.2 the tile map's slice contains two tileable cells with index 1 and 2. Each of these tileable cells have to be split into two split cells. To determine which cells are tiled over a boundary, we have to tag every cell, which comes in contact with a boundary, with a so-called “face code”. Since a three-dimensional texture map has a cubic shape each face side can be tagged with a face code, which is realized by a six bit binary code shown

in Figure 4.3. The split cell algorithm needs the following three three-dimensional textures:

1.) **tileMap:** Is the input for the split cell algorithm.
2.) **splitCellMap1**: A three-dimensional texture, which is initialized with the face codes.
3.) **SplitCellMap2**: Another three-dimensional texture, which is empty meaning all voxels are set to zero.

The data type (unsigned integer) and the resolution of both split cell maps are the same as for the tile map. In Algorithm 4.1 we can see the pseudo code for a split cell algorithm. The tile map is set as input for the split cell shader (row1 in Algorithm 4.1). A sort of “ping pong” swapping between the two split cells splitCellMap1 and splitCellMap2 is done after each split cell shader pass (row 3-11 in Algorithm 4.1). Ping pong swapping in this context means that one split cell map is set as input and the other as output for the split cell shader.
The main idea of the split cell shader (pseudo code shown in Algorithm 4.2) is to go through every voxel and apply an OR-operator (row 13 in Algorithm 4.2) on the corresponding neighboring voxels (26 directions since we are working with three-dimensional texture maps) of the same cell. At the end of the whole process every cell is tagged with the corresponding face code, meaning that for each cell we know exactly every touching boundary.

```
1  set tileMap as input texture
2
3  for each pass
4        set splitCellMap1 as input texture
5        set splitCellMap2 as output texture
6        do split cell shader
7
8        set splitCellMap1 as output texture
9        set splitCellMap2 as input texture
10       do split cell shader
11 end
```

Algorithm 4.1: *Split cell algorithm.*

```
1  for each voxel
2     cCellIndex = tile map's cell index on current position
3     cSplitCellIndex = get split cell index on current position
4
5     //empty space
6     if(cCellIndex == 0)
7       return 0;
8
9     for each neighboring voxel
10      nCellIndex = tile map's cell index on neighboring position
11      if(nCellIndex == cCellIndex)
12         nSplitCellIndex = get split cell index on neighboring position
13         cSplitCellIndex =  cSplitCellIndex | nSplitCellIndex;
14      end
15    end
16
17    return cSplitCellIndex;
18 end
```

Algorithm 4.2: *The split cell shader.*

An example of an input tile map and split cell map initialized with face codes is shown in Figure 4.4. In this example only two passes (shown in Figure 4.5) of the split cell shader are needed to get the final result. The number of passes depends on the resolution of the input maps. Assuming that NxNxN is the resolution of the input map the algorithm needs N passes to assure correct results.

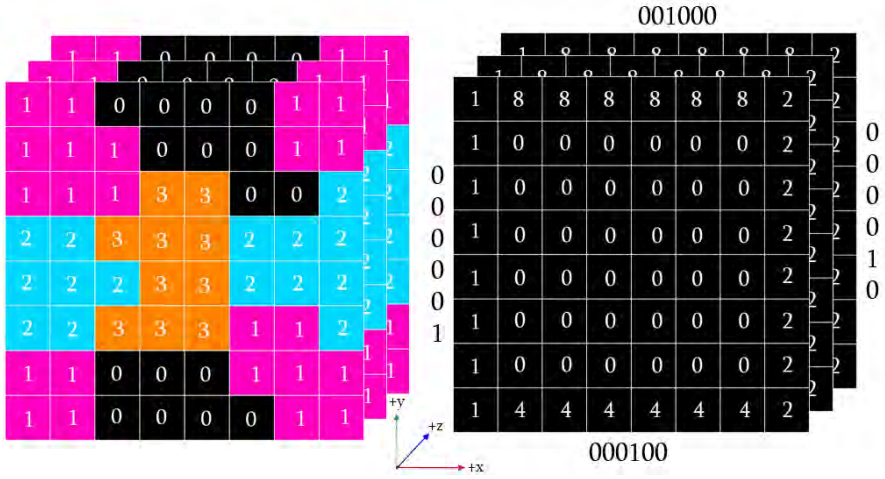

Figure 4.4: *The two input maps for the split cell shader. The tile map (left) containing three cells and empty space. The split cell map (right) initialized with face codes.*

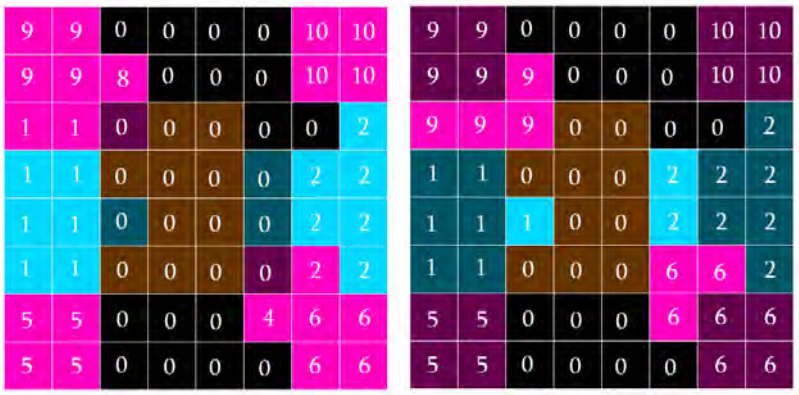

Figure 4.5: *Two passes of the split cell algorithm for the example in Figure 4.4. First pass is on the left and the second and last pass is on right. The highlighted pixels represent the pixels modified in the current pass.*

4.3 Split Cell Index Determination

As we can see in Figure 4.5 every split cell has a corresponding face code. But for a correct cell membership map indexing (explained in Chapter 3.1.6) we cannot adopt the face codes in our resulting split cell map (shown in Figure 4.5 right). With the knowledge that every cell needs a placeholder of eight split cell indices we define for each cell an ascending sequence, meaning that the first cell has the index range from 1 to 8, the next cell the index range from 9 to 15 and so on. Hence the only task is to convert the resulting face codes into an ascending split cell index. The result of conversion for the example from Figure 4.5 can be found in Figure 4.6.

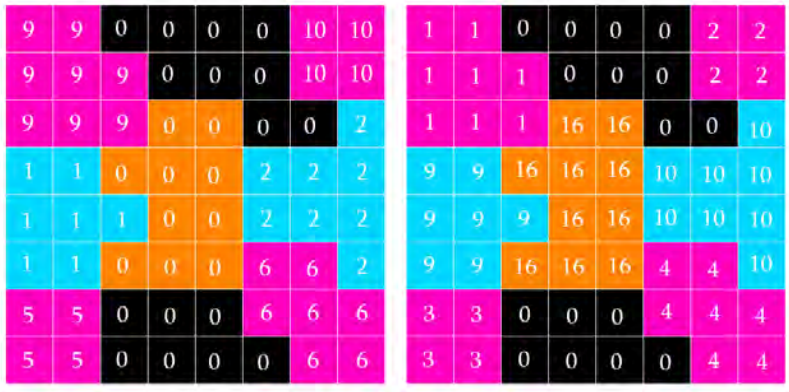

Figure 4.6: *Conversion of face codes (left) into an ascending split cell index (right) needed for a correct cell membership map indexing.*

At the end of the split cell index conversion two new important textures are created, the face code map (in Figure 4.6 left) and the split cell map (in Figure 4.6 right). Both textures are important for the cell membership determination.

4.4 Cell Membership Determination

The basic concept of cell membership determination is explained in Chapter 3.1.5. At the end of this process the cell membership map contains for each split cell in object space a membership information (inside, outside and intersected). The main implementation idea of the cell membership determination is to calculate the split cell membership slice by slice (see Algorithm 4.3). As input for the cell membership determination a voxelization of the basic mesh (mentioned in Chapter 3.1.4) is required. In Figure 4.7 on the left we can see a slice of a voxelized object. Notice the black and grey areas. The grey area (tagged with a 1) represents the inside and the black area (tagged with 0) the outside of the basic mesh. In the following the set of all slices will be called the "inside outside map". The resolution of the inside outside map depends on two parameters calculated by the resolution of the split cell map times the number of tiles. Since the resolution of the split cell map in Figure 4.7 on the right is 8x8 and since the split cell map gets tiled over the object space two times in every direction (x, y, and z) the resulting resolution of the inside outside map in Figure 4.7 on the left has to be 16x16. On modern graphics cards the maximum texture size is 8192x8192. Hence a high precision of the inside outside voxelization is possible. Assuming the maximum texture memory size is 1 GB the cell membership map contains 406x406x406 tiles at most, due to the following formula:

$$\text{number of tiles} = \sqrt[3]{\frac{\text{max texture memory size in bytes}}{\text{face code in bytes}}} = \sqrt[3]{\frac{1024^3}{16}} = 406$$

In this case the resolution of the split cell map is 20x20x20 pixels calculated by the above mentioned formula:

$$\text{split cell map resolution} = \frac{\text{inside-outside map resolution}}{\text{number of tiles}} = \frac{8192}{406} = 20$$

If this split cell map contains only one cell, which would make sense for a resolution of 20x20x20 pixels, this cell based object can have up to 406 cells (due to 406 tiles) in one coordinate axis (or up to 406^3 = 66923416 cells in the whole object space). Since only the inside cells in the cell-based ray tracing are rendered usually only half of the possible cells gets rendered depending on the basic mesh. On the whole we can assume that the following ratio between inside, outside and intersected cells exists:

inside:outside:intersected = 3:2:1

```
1       for each slice
2               create inside outside slice
3               calculate split cell membership
4       end
5
6       merge split cell membership information
```

Algorithm 4.3: *Cell membership determination.*

After computing the inside outside voxelization (see Algorithm 4.3 row 2) it has to be determined whether split cells are totally inside, outside or even intersecting the basic mesh (see Algorithm 4.3 row 3). For this a new data structure is defined, which holds the cell membership information the so-called "*cell membership map*". The resolution of the cell membership map is depending on the number of tiles. If N is the number of tiles in one direction the resolution of the cell membership map is set to NxNxN. A tiling in three-dimensional object space is shown in Figure 4.8. Due to the fact that a cell can be tagged as "*inside"*, "*outside"* or "*intersected"* we need only two bits per split cell. In our implementation the following bit codes are defined:

Binary code	Cell membership tag
00	OUTSIDE
01	INSIDE
10	INTERSECTED
11	UNSET

Table 4.1: *Binary codes for cell membership information.. In the following the colors will represent the corresponding cell membership tags for better visualization.*

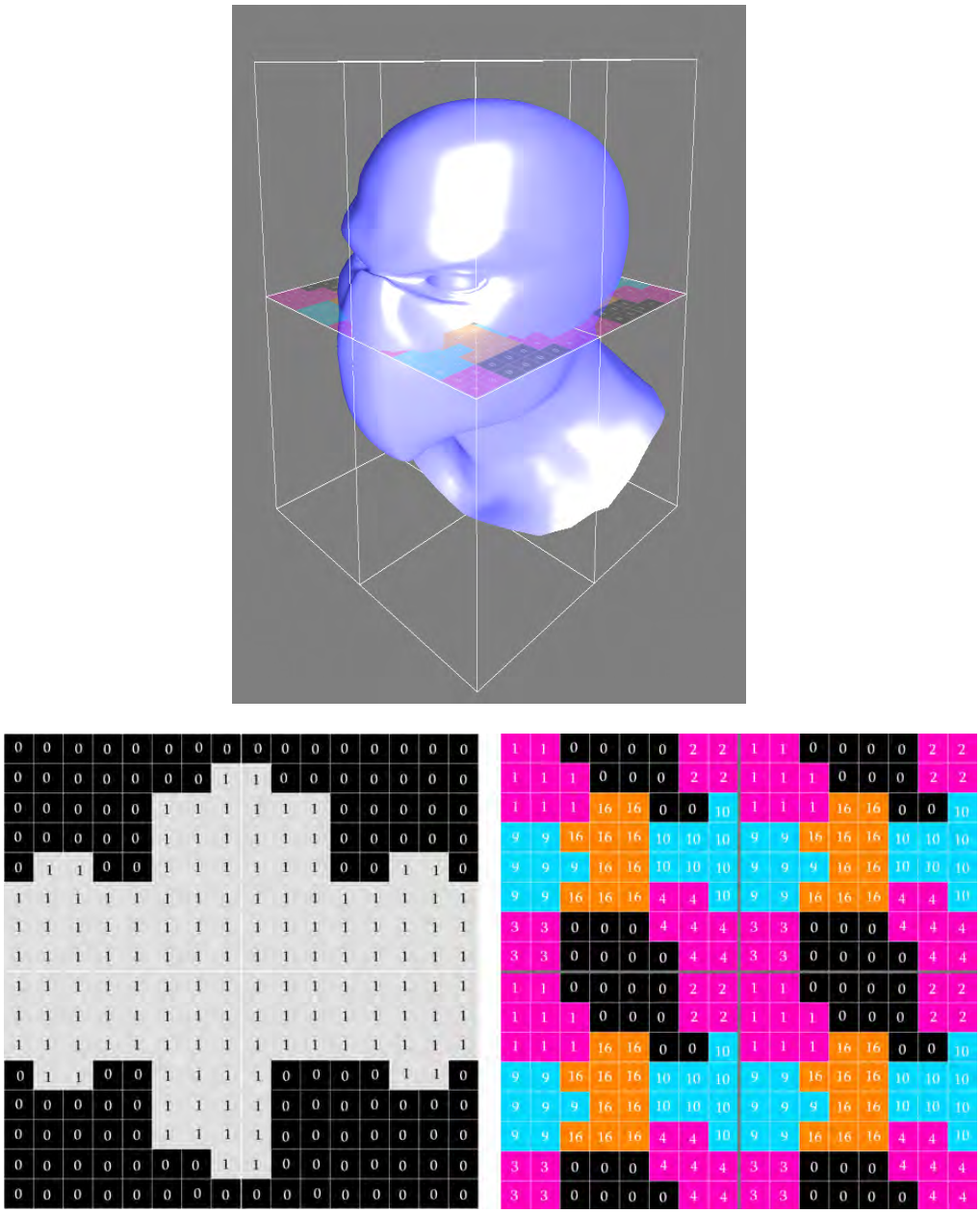

Figure 4.7: *top: shows the corresponding slice in object space. bottom left: voxelized basic mesh, bottom right: split cell map tiled over object space.*

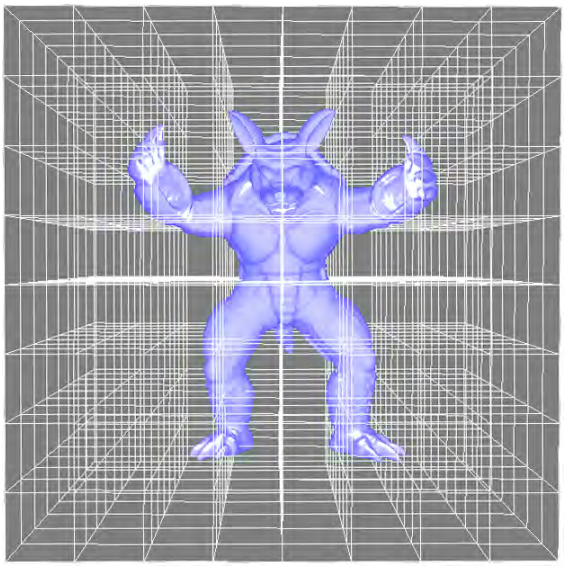

Figure 4.8: *Object space divided into 8x8x8 equal distant cubes. Each cube represents a tile.*

The bit code 11 as in Table 4.1 represents an "*unset*" voxel, which is important for initialization. On modern graphics hardware it is possible to store a 128-bit value per voxel in a three-dimensional texture. Consequently it is possible to store 64 split cell membership information per tile. In the Chapter 4.2 it was mentioned that a cell splitting is needed for cells, which are tiled over the tile maps corner or edges. Due to the fact that a tiled cell could be split into eight split cells at most and the split cells are tagged with an ascending split cell index we get the following bit allocation shown in Figure 4.9. In the following the 128-bit code per tile will be called the "*tile bit code*". That is the cell membership map is stored as a three-dimensional texture with a 32-bit RGBA channel. Each 32-bit channel can store 16 split cells (each split cell needs two bit). Hence the cell membership is nothing else than a list of tile bit codes.

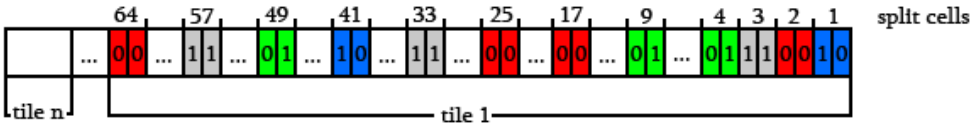

Figure 4.9: *Bit allocation for the cell membership map. A tile is a 128-bit code containing 64 2-bit split cells. Each 2-bit code represents a split cell membership tag.*

So far due to the inside outside voxelization only the cell membership of one voxel is calculated. However we need the membership of the whole cell. This is given by the algorithm explained in Algorithm 4.4.

```
1 initialize membership map with the UNSET tag
2 for each voxel in the inside outside map
3
4         calculate cell and tile coordinates from current voxel position
5         get split cell index
6
7         if (empty space)
8                 continue (jump back to row 3);
9         end
10
11        membership = get split cell tag from cell membership map
12
13        if (membership == INTERSECTED)
14                continue (jump back to row 3);
15        end
16
17        newMembership = get tag from inside outside slice
18
19    if( (newMembership != membership) && (membership != UNSET) )
20                newMembership = INTERSECTED;
21    end
22
23        overwrite existing cell membership with new one
24  end
25 end
```

Algorithm 4.4: *Split cell membership determination.*

First of all every split cell in the cell membership map gets initialized with the "unset" tag (see row 1 in Algorithm 4.4). The idea is to go through every slice of the inside outside map pixel by pixel (see row 2 in Algorithm 4.4). Due to the knowledge of the tile's size two coordinates are calculated from the current voxel position (see row 4 in Algorithm 4.4) by the following formula:

tile coordinate = voxel position **modulo** tile's size
cell coordinate = voxel position – tile coordinate

With the cell coordinate a simple lookup in the split cell map returns the split cell index (see row 5 in Algorithm 4.4). If the split cell index is equal to zero the voxel is inside an empty space. Hence no further computation is needed and therefore the algorithm jumps to the next voxel (see row 7-8 in Algorithm 4.4). As mentioned in Chapter 3.1.6 a split cell localization is realized by two coordinates, the tile and cell coordinate. The tile bit code contains the cell membership for each split cell inside a corresponding tile. To get the tile bit code a lookup in the cell membership map with the corresponding tile coordinates has to be done (see row 3 in Algorithm 4.5). Additionally some bit comparisons (explained in Algorithm 4.5) are done to get the corresponding split cell membership tag (see row 11 in Algorithm 4.4). The input for the bit comparison is the split cell index (range from 1 to 64). As mentioned before the cell membership is stored as a three-dimensional texture with a 32-bit RGBA channel. Hence the tile bit code has 128-bit storing 64 split cells (16 split cells in one channel). The corresponding channel is computed by dividing the split cell index by 16 (Algorithm 4.5 row 2). Next the bit position inside the channel is computed (shown in Algorithm 4.5 row 1). To get tile bit code only a lookup in the membership map with the corresponding tile coordinate is done. Knowing the channel and the bit position in this channel it is very easy to get the split cell membership out of the tile bit code (see Algorithm 4.5 row 5-16).

```
1  shift = (splitCellIndex & 15) << 1;
2  channel = splitCellIndex >> 4;
3  tileBitCode = get tile code
4
5  if ( channel == 0 )
6          splitCellMembershipTag =  ((tileBitCode.r & (3<<shift)) >> shift);
7  end
8  if ( channel == 1 )
9          splitCellMembershipTag =  ((tileBitCode.g & (3<<shift)) >> shift);
10 end
11 if ( channel == 2 )
12         splitCellMembershipTag =  ((tileBitCode.b & (3<<shift)) >> shift);
13 end
14 if ( channel == 3 )
15         splitCellMembershipTag =  ((tileBitCode.a & (3<<shift)) >> shift);
16 end
```

Algorithm 4.5: *Bit comparisons to get split cell membership from cell membership map.”&” = bitwise AND. “<<” = shift left. “>>” = shift right.*

The final step is to compare the voxel's membership from the inside outside map (see row 17 in Algorithm 4.4) with the already written membership information in the cell membership map. By defining the following rules we can determinate the membership of a whole split cell:

Membership 1	Membership 2	Resulting Membership
INSIDE	INSIDE	INSIDE
OUTSIDE	OUTSIDE	OUTSIDE
INSIDE	OUTSIDE	INTERSECTED
OUTSIDE	INSIDE	INTERSECTED

Table 4.2: *Rules for combining cell memberships. Column 1 and 2 are the inputs and the third column the output for this operation.*

The result after the split cell membership determination explained in Algorithm 4.4 for the input maps from Figure 4.7 can be seen in Figure 4.10. The resulting membership map is shown in Figure 4.11. Note that this membership map has only the split cell membership computed without taking the whole cell into account. Therefore a split cell membership merger explained in the next Chapter 4.5 has to be done. The split cell membership merger is the last step to determine the final cell membership map.

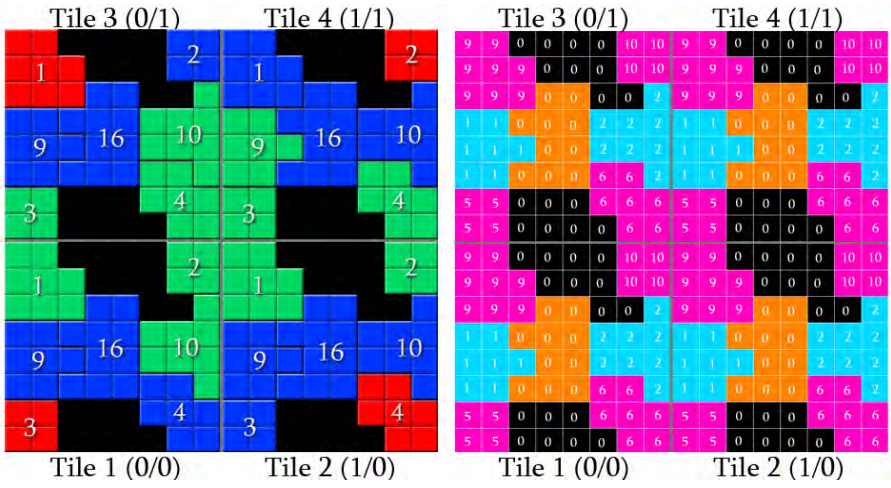

Figure 4.10: *Left: The result of the split cell membership determination from the example in Figure 4.7. Split cells are tagged with INSIDE(green), OUTSIDE(red) and INTERSECTED(blue). The numbers represent the split cell index in a tile. Right: The corresponding face code map from the example in Figure 4.7 tiled over the object space. This map shows which split cells belong to which cells. Tile coordinates (x/y) can be found right after the tile label.*

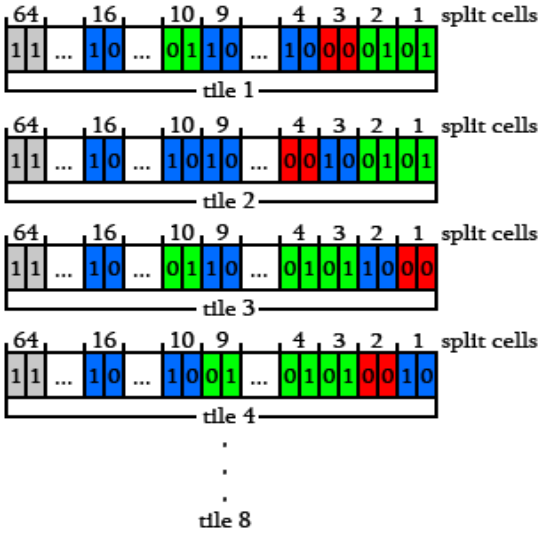

Figure 4.11: *The resulting cell membership map from example in Figure 4.7. Each row represents a tile bit code containing 64 split cell memberships. Split cells are tagged with INSIDE(green), OUTSIDE(red), INTERSECTED(blue) and UNSET(grey). The binary code is defined in Table 4.1.*

4.5 Split Cell Membership Merger

At this point we only know the membership of each split cell. The last step is to merge the split cell membership information to determine the cell membership of a whole cell (build of up to eight split cells). In our implementation we store a lookup table (see Figure 4.12), which contains for every split cell a corresponding face code and additionally know which split cells belong to which cell. The split cell membership merger is done on the CPU. In this process (explained in Algorithm 4.6) only the cell membership map is needed. In Algorithm 4.6 we go through every corresponding split cell of a cell and merge the corresponding split cell memberships (see row 7-10 in Algorithm 4.6) by the same rules (see Table 4.2) as mentioned in Chapter 4.4. After calculating the proper membership for the current cell, all corresponding split cells has to be set to the calculated cell's membership in the membership map (see row 13-15 in Algorithm 4.6).
For a better understanding in the following a specific split cell is localized by the tile coordinate tx and ty and the split cell index si by the tuple (tx/ty)(si). For example in Figure 4.10 on the left the split cell 10 in tile 2 is localized by the tuple (1/0)(10). If we look at Figure 4.10 we can see that split cell (0/0)(10) and (1/0)(9) belong to the cyan colored cell. Split cell (0/0)(10) is tagged as INSIDE and split cell (1/0)(9) is tagged as INTERSECTED. Since we are using the rules

of Table 4.2 the whole cell should be tagged as INTERSECTED. The question is how do we know in which neighboring tile we have to lookup to find the neighboring split cell? In the previous Chapter 4.4 we explained the face code, which was used to determine, in which edge or corner the cell gets tiled. With this face code it is very easy to determine the corresponding split cells (see row 4-5 in Algorithm 4.6) of a tiled cell lying in neighboring tiles by doing some bit comparison explained in Algorithm 4.7.

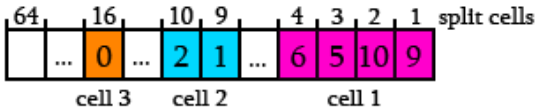

Figure 4.12: *Lookup table stores face codes for every split cell. This lookup table is generated from the example from Figure 4.10. Magenta cell 1 contains 4 tiled split cells with face codes 5,6,9 and 10. Note that cell 3 contains only a split cell tagged with face code 0, consequently this cell was not tiled.*

```
1 for each tile
2       for each split cell
3               currentMembership = get current split cell membership
4               get face code
5               get corresponding neighboring split cells out of face code
6               for each corresponding neighboring split cell
7                       neighboringMembership = get split cell membership
8                       if( (neighbouringMembership == INTERSECTED) ||
9                           (currentMembership != neighbouringMembership) )
10                              currentMembership = INTERSECTED;
11                      end
12              end
13              for each corresponding neighboring split cell
14                      set currentMembership for split cell in membership map
15              end
16      end
17 end
```

Algorithm 4.6: *Cell membership determination by merging the split cell membership information.*

```
1 currentTileCoord = get current tile coordinates
2 currentFaceCode = get face code of current split cell
3 for each corresponding neighboring split cell
4       neighboringFaceCode = get face code of corresponding  split cell
5       directions = currentFaceCode | neighbourFaceCode
6       xDir =  directions & 3
7       yDir = (directions & (3<<2)) >> 2
8       zDir = (directions & (3<<4)) >> 4
9       xShift = yShift = zShift = 0;
10      if(xDir == 3)
11              xShift = 2( currentFaceCode & 3) - 3
12      end
13      if(yDir == 3)
14              yShift = 2*(( currentFaceCode & (3<<2)) >> 2) - 3
15      end
16      if(zDir == 3)
17              xShift = 2*(( currentFaceCode & (3<<4)) >> 4) - 3
18      end

19      newTileCoordinate = currentTileCoord + (xShift,yShift,zShift)
20      store newTileCoordinate in a list
21 end
```

Algorithm 4.7: *Bit comparisons to get the tile coordinate of the corresponding neighboring split cells.”&” = bitwise AND. “|” = bitwise OR.“<<” = shift left. “>>” = shift right.*

The bit comparison in Algorithm 4.7 is best explained by the following example. Note that in this example we only take the two-dimensional case into account. Assuming we take the same split cells as mentioned before, split cell 10 and 9 from Figure 4.10 and assuming our current tile coordinate (tx,ty) is (1/0). Due to the lookup table (see Figure 4.12) we know that split cell 9 has as it's neighbor split cell 10. In Table 4.3 we can see the results for the specific example to get the right neighboring tile coordinate out of the face code explained in Algorithm 4.7.

Binary Code	Decimal Number	Row# in Algorithm 4.7	Comment
0001	1	Row 2	currentFaceCode of Split Cell 9
0010	2	Row 4	neighboringFaceCode of split cell 10
0011	3	Row 5	directions
0011	3	Row 6	xDir
0000	0	Row 7	yDir
-	-1	Row 11	xShift
-	0	Row 9	yShift

Table 4.3: *Intermediate results of Algorithm 4.7 from the example in Figure 4.10.*

neighboring tile coordinate (tx+xShift,ty+yShift) = (1-1,0+0) = (0/0) row 19

Accordingly if the current tile coordinate for split cell 9 is (1,0) the neighboring split cell 10 is found in the neighboring tile with coordinate (0,0). A comparison between two membership maps before and after the split cell merger process are shown in Figure 4.13.
The split cell membership merger is the last step to determine a proper cell membership map. Finally a cell-based object representation is described by the tile and cell membership map, which are necessary for cell localization in the rendering stage.

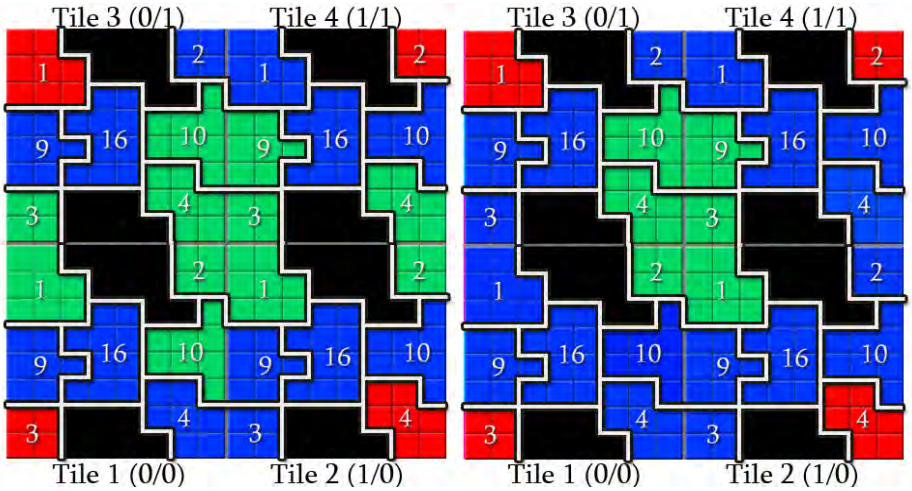

Figure 4.13: *The cell membership map from the example in Figure 4.10 before (left) and after (right) split cell merger process. Note that only two cells are totally inside.*

4.6 Normal Map

As mentioned in Chapter 3.1.8 in our implementation we precompute normals on the cell's boundary. An edge normal for a cell is calculated in the following way. With the tile map as input we go through every voxel in this map (row 1 in Algorithm 4.8). The idea is to generate for every voxel up to 26 normals pointing to the neighboring position (row 5 in Algorithm 4.8). It is important, that only normals with a different cell index are taken into account (row 4 in Algorithm 4.8). Out of these calculated normals a mean normal has to be computed, normalized and stored into the three-dimensional normal map. The normal map has the same resolution as the tile map (row 10-12 in Algorithm 4.8). In our implementation the normal map is a 3 channel 32 bit floating point texture.

```
1 for each voxel in the tile map
2       initialize finalNormal with zeros
3       for each neighboring voxel
4               if (current voxel has a different cell index as neighboring voxel)
5                       newNormal = neighboring position - current position
6                       normalize newNormal
7                       finalNormal = finalNormal + newNormal
8               end
9       end
10      calculate mean value out of finalNormal
11      normalize finalNormal
12      store finalNormal in normal map
13 end
```

Algorithm 4.8: Calculation of the edge normal out of the tile map.

4.7 Tools

For a prove of concept a DirectX 10 implementation was created using HLSL shading and C++ as programming language. We programmed two preprocessing tools for texture precomputation (tile map, membership map and normal map) and a rendering program, which displays the cell-based object with a ray tracer (explained in Chapter 3.2). In the first precomputation program it is possible to create a tile, color and a normal map procedurally. In Figure 4.14 we used a three-dimensional Vornoi diagram for generating all three maps. In the second tool the basic mesh, the tile map and the number of tiles per axis are needed as input to generate a corresponding cell membership map seen in Figure 4.15.

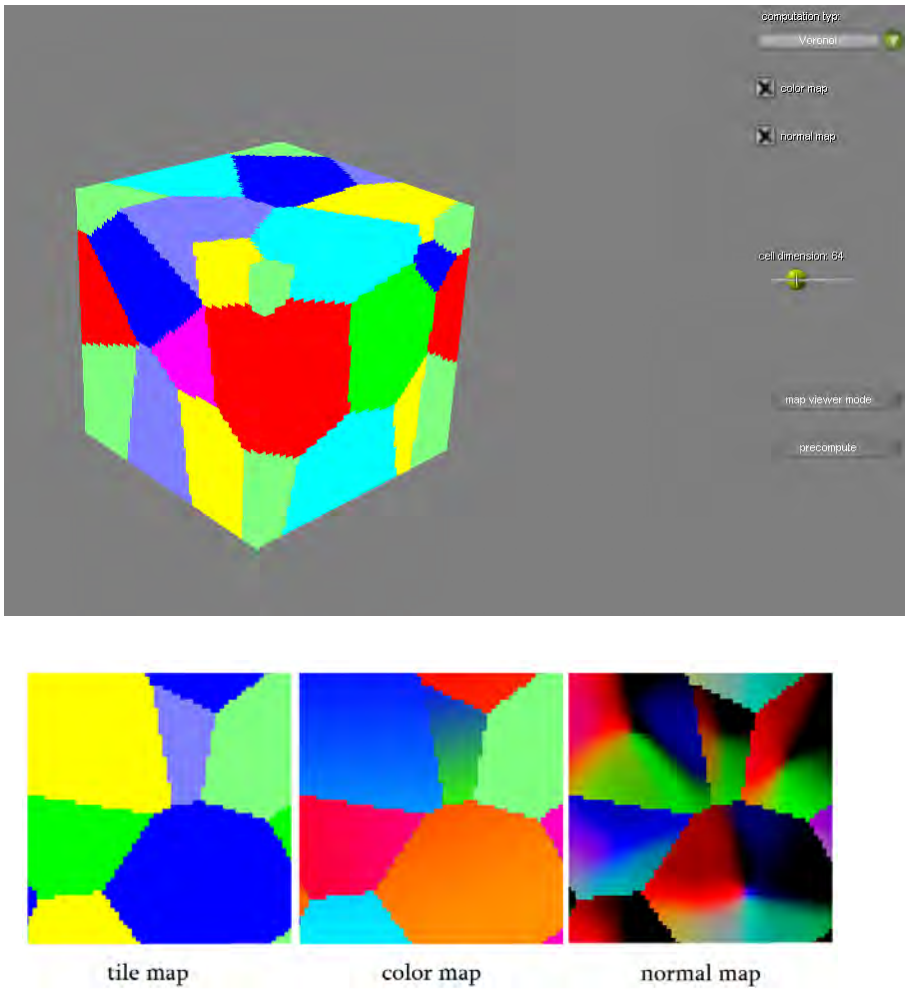

Figure 4.14: *Precomputation tool to generate a tile, color and normal map procedurally. This is an example of a Voronoi diagram with eight randomly set seed points.*

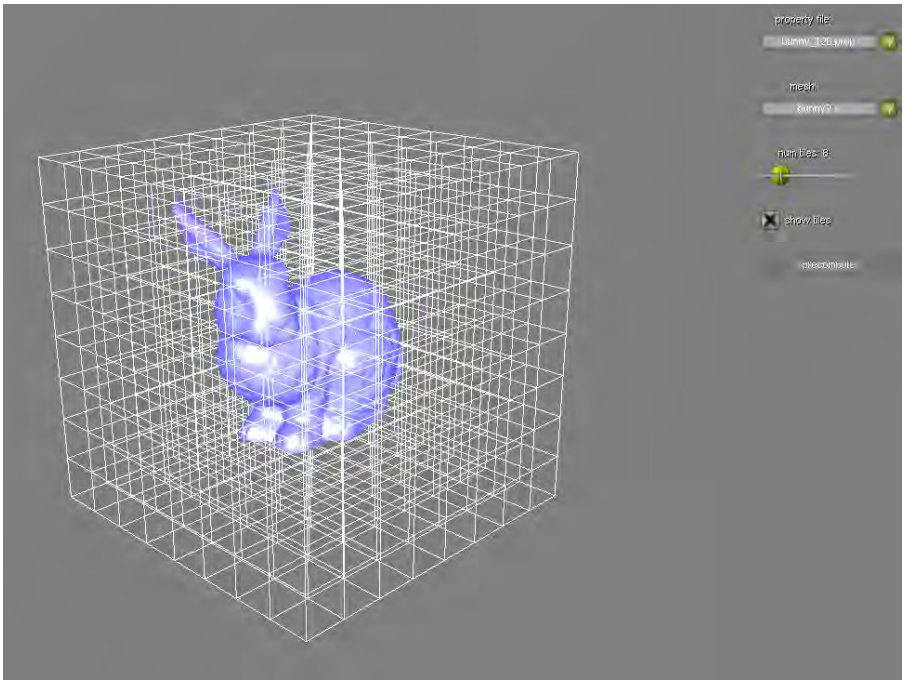

Figure 4.15: *Precomputation tool to generate a membership map. Inputs for this tool are the basic mesh (blue object), the corresponding tilemap and the number of tiles per axis.*

The rendering program puts all precomputed maps together and renders the cell-based object with three different ray tracing modes: first hit, specular self-reflection and refraction (shown in Figure 4.16). The first hit mode uses only primary rays. The two other modes additionally shoot further rays to render more interesting effects.

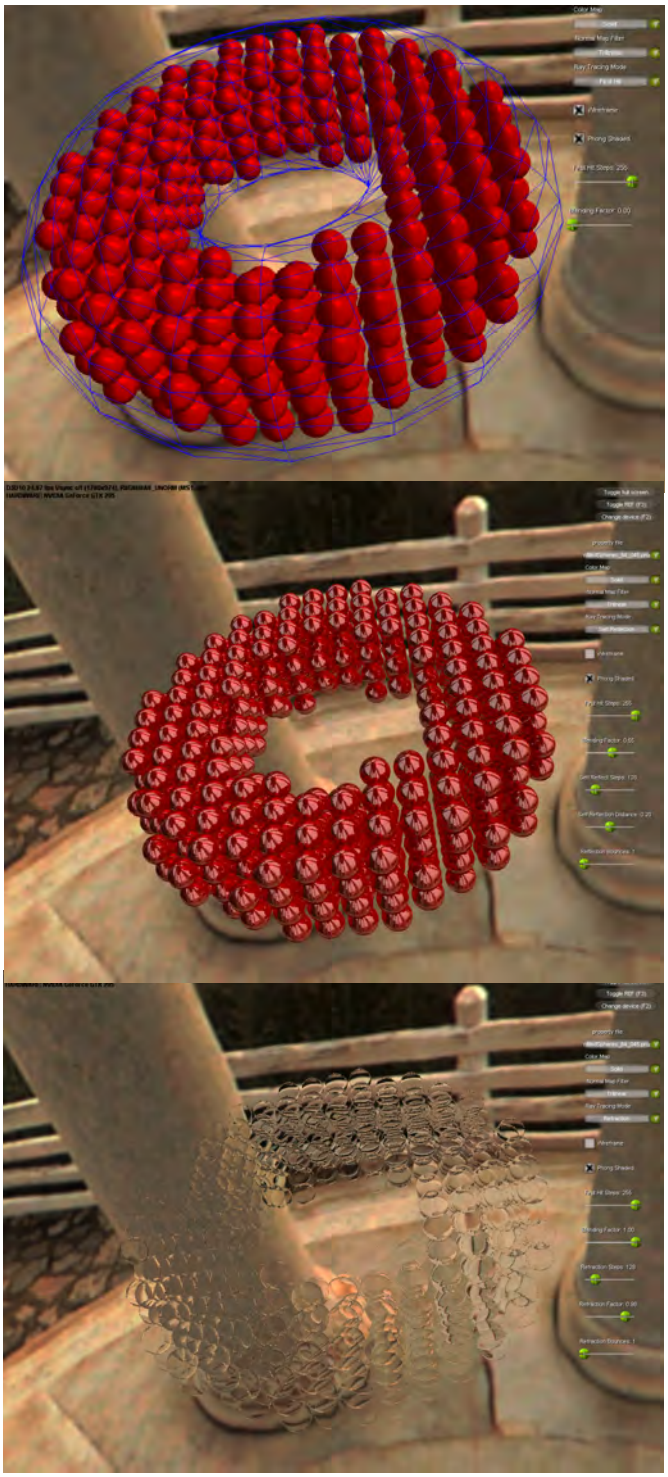

Figure 4.16: *Rendering program to demonstrate the ray tracing for cell-based objects. It is possible to switch between three rendering modes. From top to bottom: first hit, specular self-reflection and refraction.*

„By three methods we may learn wisdom: first, by reflection, which is noblest; second, by imitation, which is easiest; and third, by experience, which is the most bitter.„

Confucius

Chapter 5

Results and Discussion

All images in this chapter were produced on a Intel Core 2 PC at 2.4 GHz CPU with 2 GB RAM and a Geforce GTX 295 GPU with 2x896 MB video RAM. Since using ray tracing for displaying the cell-based objects the performance highly depends on the output resolution. In order to keep a basis for comparison all images were rendered with an output resolution of 1024x768.

5.1 Memory Consumption

In the precomputation stage the user is able to choose the cell map resolution and the number of tiles. A higher cell map resolution is needed for close-up views (see Figure 5.1 bottom). A higher number of tiles increases the amount of cells. In Table 5.1 we can see four various parameter setups to produce a high-resolution cell-based object. High resolution in this context means using a high precision inside-outside map (8192x8192) for voxelization as mentioned in Chapter 4.4.

User parameters		**Texture memory size**				
Cell map resolution	**Number of tiles**	**Cell map**	**Color map**	**Normal map**	**Cell membership map**	**Total**
256x256x256	32x32x32	16 MB	8 MB	256 MB	512 KB	280 MB
128x128x128	64x64x64	2 MB	8 MB	32 MB	4 MB	46 MB
64x64x64	128x128x128	256 KB	8 MB	4 MB	32 MB	44 MB
32x32x32	256x256x256	32 KB	8 MB	512 KB	256 MB	264 MB

Table 5.1: *Four parameter setups for a cell-based object with the corresponding texture memory consumption. The color maps [23] have a 32-bit RGBA channel with a resolution of 128x128x128.*

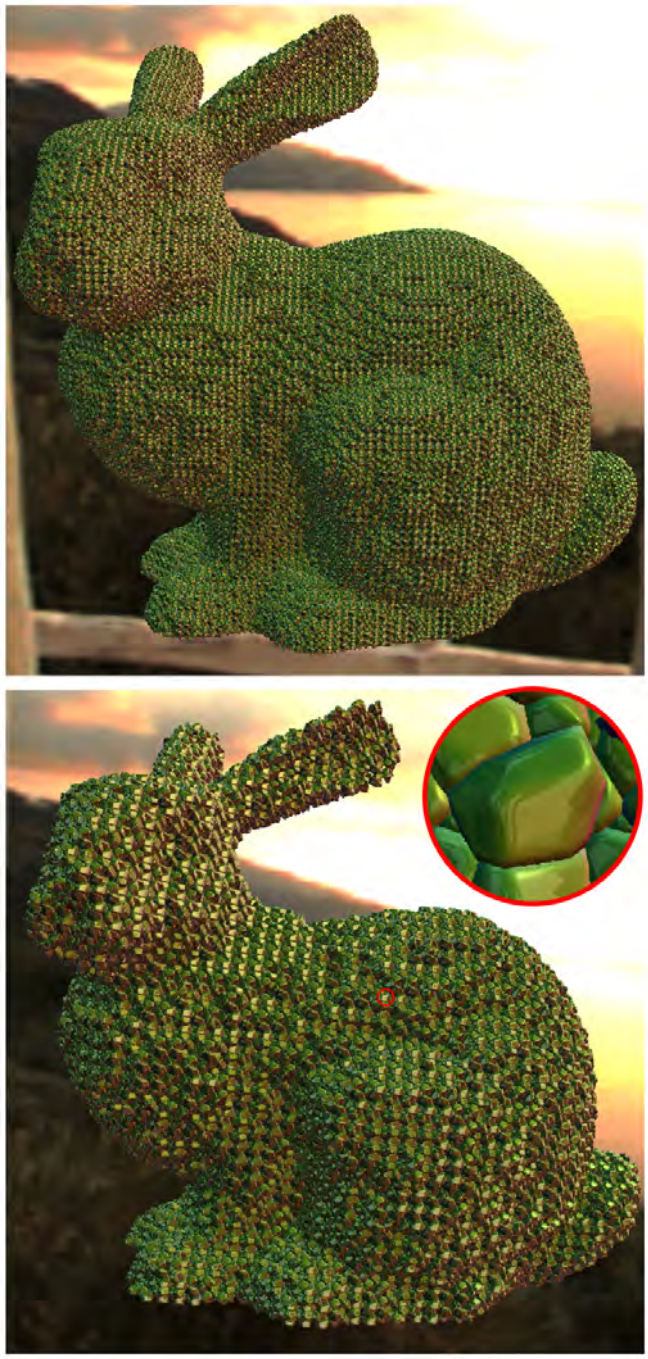

Figure 5.1: *A high resolution cell-based Stanford Bunny is displayed by an ray tracer using a specular self-reflection. Top: cell map resolution 64x64x64 and number of tiles are 128x128x128. Bottom: cell map resolution 128x128x128 and number of tiles are 64x64x64. Note the the specular self-reflection in close-up view.*

Two resulting cell-based objects from the parameter setup from Table 5.1 (row 2 and 3) are shown in Figure 5.1. Note the high frequent, repetitive surface details (Figure 5.1 top). In this example high frequent means that the object has about 256 cells per axis. This value is estimated based on the fact that the object-space is split into 128x128x128 tiles and each tile comprises 8 cells. By taking only one dimension into account a cell resolution of 128 tiles times 2 cells per axis results into 256 cells per axis. These two cell-based Stanford Bunnies need 46 MB (Figure 5.1 bottom) and 44 MB (Figure 5.1 top) texture memory, respectively. The texture size highly depends on the two mentioned parameters the cell map resolution and the number tiles. Hence a high resolution parameter setup leads to high image quality at the costs of high memory consumption.

5.2 Computational Time

The frames per second highly depends how much the cell-based object covers the screen. The larger the covered area the more pixels have to be processed by the ray tracer inducing a lower frame rate. However, for comparison the cell-based Stanford Bunny from Figure 5.1 (bottom) is rendered with three different rendering modes: first hit, specular self-reflection and refraction (mentioned in Chapter 3.2). Each rendering mode is rendered from different zooming positions to achieve various pixel coverages. In Table 5.2 the exponential dependency between pixel coverage and frame rate is shown. Since the specular self-reflection and refraction shoot higher order rays, meaning that a ray bounces more than once between the object's surface, these two techniques need more computational time than the first hit rendering mode.

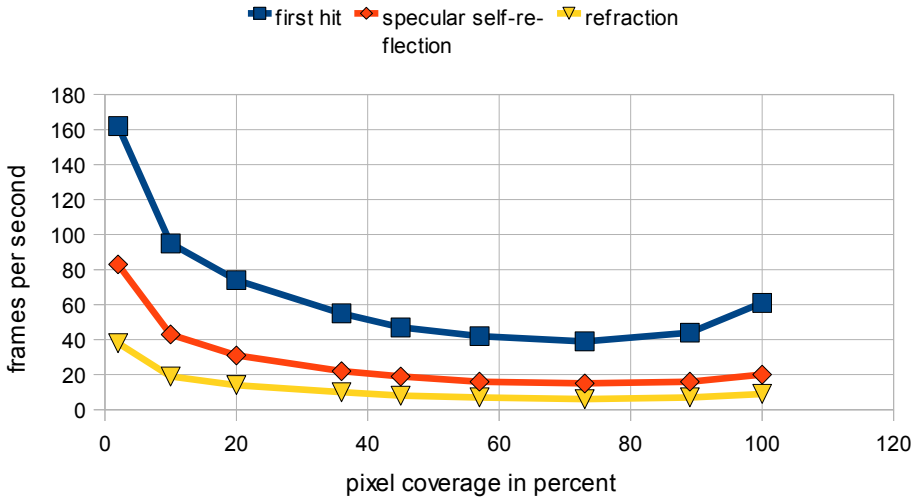

Table 5.2 *Stanford Bunny rendered with three different rendering modes: first hit, specular self-reflection and refraction. Note the exponential dependency between pixel coverage and the frame rate.*

Note in Table 5.2 that for close-up views, having a pixel coverage about 90 to 100 percent, the frame rate goes up again. This happens because a smaller amount of cells is visible and therefore needing less computational time. Consequently the second factor, which influences the frame rate is the number of visible cells. In Table 5.3 four close-up views with a pixel coverage of 100 percent are compared to show the influence of visible cells on the frame rate.

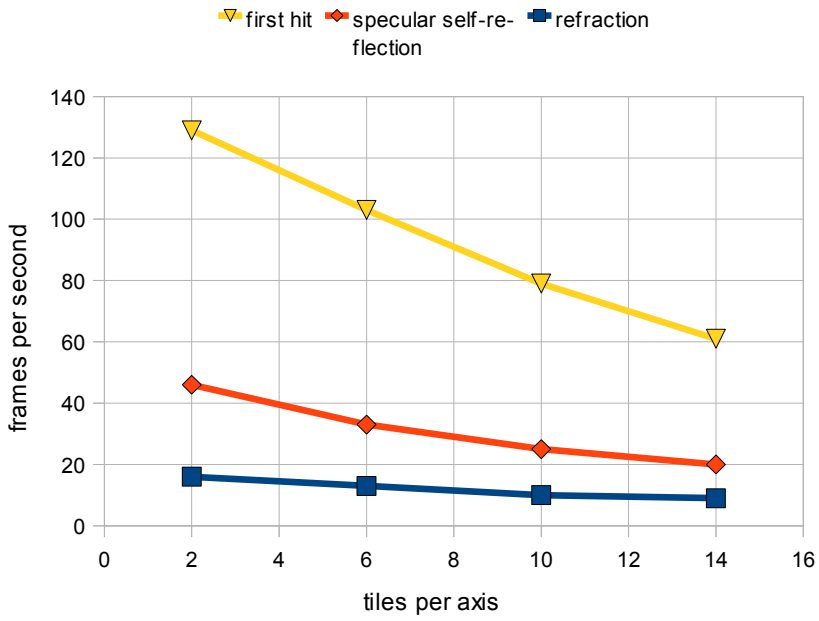

Table 5.3: *Four different close-up views with a pixel coverage of 100 percent. Note the dependency between number of visible tiles(cells) and the frame rate.*

5.3 Visual Quality

Beside the pixel coverage and the visible number of cells there are other parameters, which have an influence on the frame rate too. How precise the boundary of a cell is hit by a ray tracer is depending on the step size for the linear search. A large step size makes the ray tracing fast but on the other hand increases the probability to miss a cell's boundary especially for grazing angles. Hence a small step size enhances the visual quality at cost of lower frame rates. In Figure 5.2 an object consists of spherical cells is rendered with various step sizes. Note that the cells near the observer have a finer, concentric silhouette with decreasing step size. In contrast for distant cells the finer ray traversal have no distinguishable improvement for the visual quality. The resulting frame rates for this example in Figure 5.2 are shown in Table 5.4. This table shows that a finer step size needs more computational time.

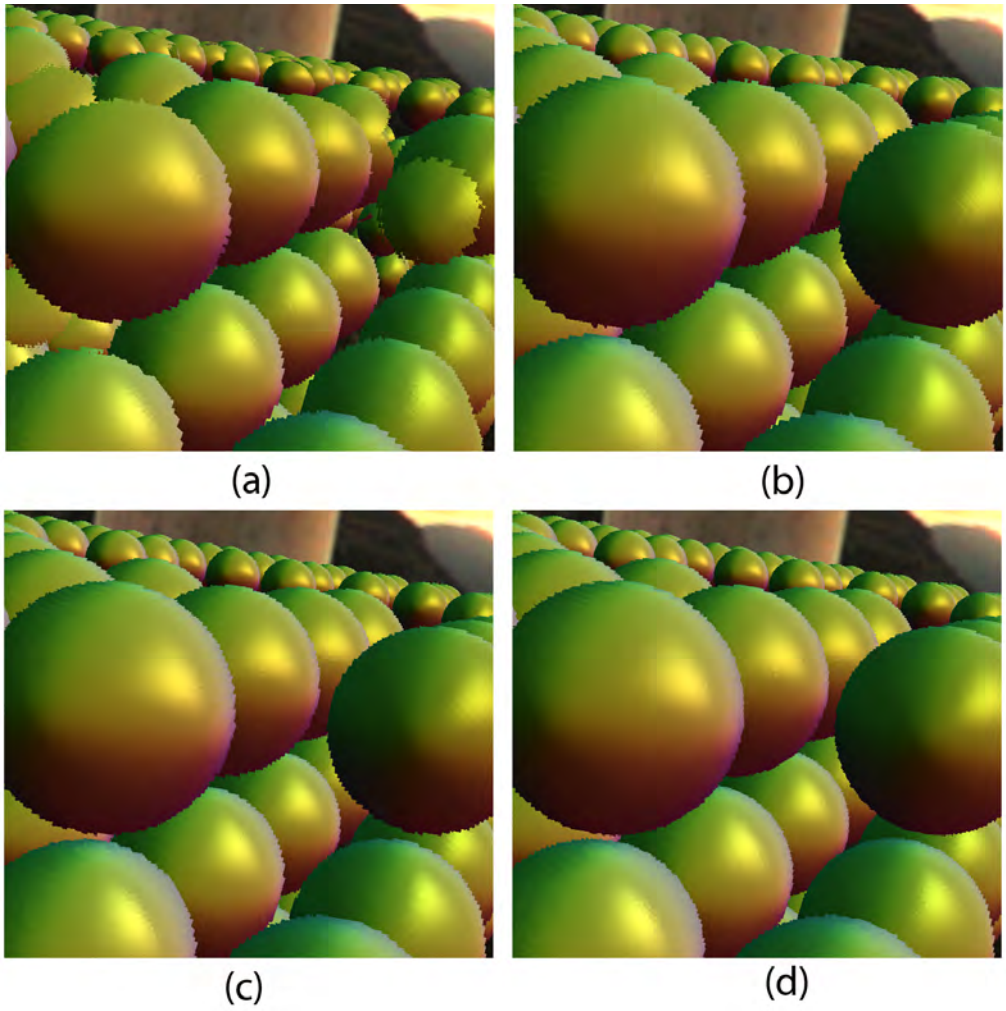

Figure 5.2: *A close-up view of an object consisting of spherical cells rendered with four different step sizes. Step sizes 1/16 (a), 1/64 (b), 1/256 ((c)) and 1/512 (d) are used for the first hit rendering. A step size with value of 1 equates to the length of a tile. Note that a finer step size results in higher visual quality.*

Not only for the first hit but also for higher order rays such as for specular self-reflection and refraction the step size influences the image quality. Figure 5.3 shows that a smaller step size induces a finer concentric silhouettes in the specular self-reflection on the spherical cells. Note that if the step size is too large the outer parts of the cells disappear since the ray misses the cell's boundary.

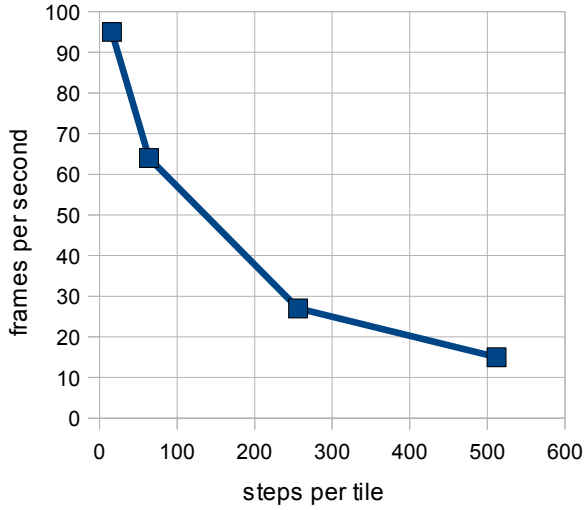

Table 5.4: *The corresponding frame rate to the four different step sizes from example in Figure 5.2. Note that a finer step size (more steps per tile) induces a lower frame rate.*

Surfaces with a high reflectivity need more than one reflection bounce for rendering a realistic specular self-reflection. As mentioned in Chapter 3.2.3 every further reflection bounce leads to more computational costs. In Figure 5.4 higher order rays are rendered for specular self-reflection. Table 5.5 shows the resulting frame rates for the higher order rays from example in Figure 5.4.

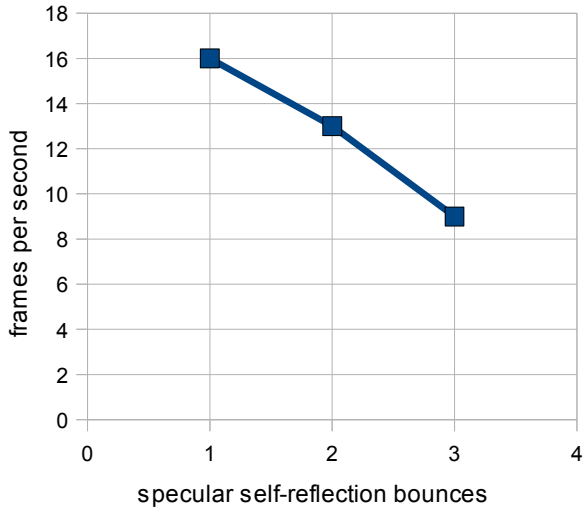

Table 5.5: *The corresponding frame rate to the three different specular self-reflection order in Figure 5.4. Note that higher order rays need more computational time.*

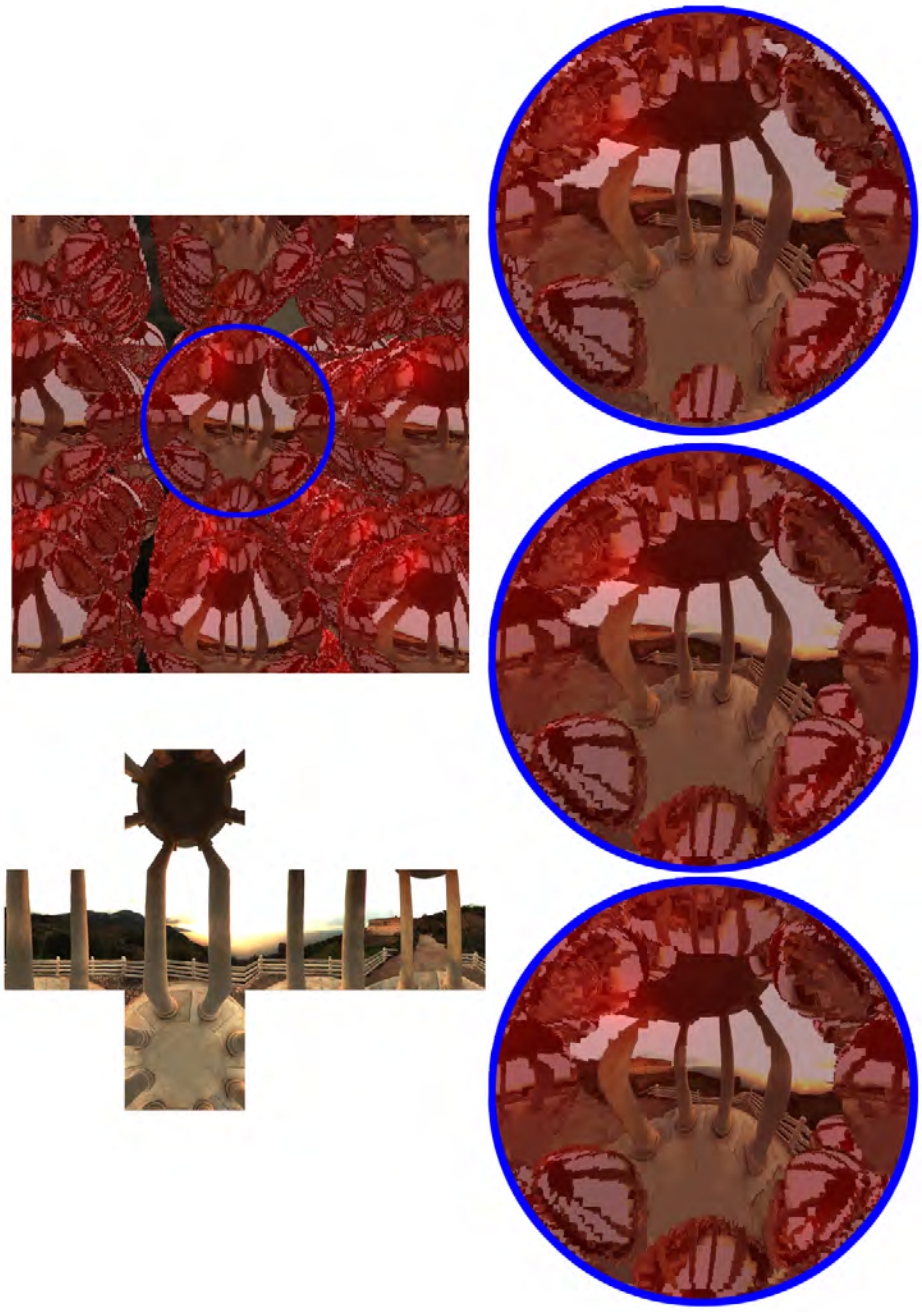

Figure 5.3: *A close-up view of an object consisting of spherical cells rendered with three different step sizes. Step sizes 1/16 (a), 1/64 (b), 1/256 ((c)) are used for the specular self-reflection. A step size with value of 1 equates to the length of a tile. Note that a finer step size results in higher visual quality.*

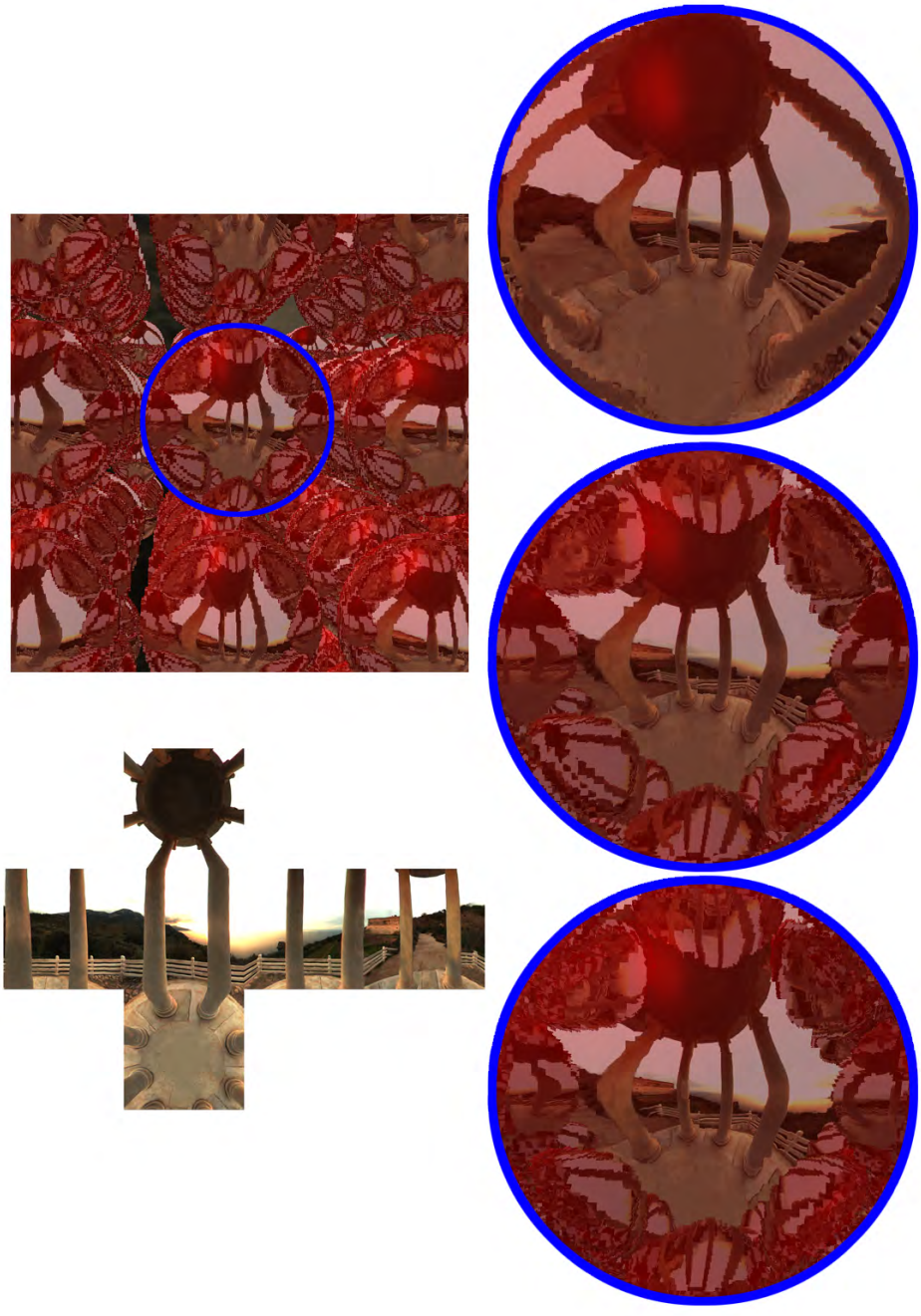

Figure 5.4: *A close-up view of an object consisting of spherical cells rendered with three different specular self-reflecting ray orders. The ray order goes from one to three from top to bottom.*

As mentioned before a higher cell map resolution is a crucial factor for the visual quality of the cell-based object. Especially in close-up views shown in Figure 5.5 a higher cell map resolution is needed.

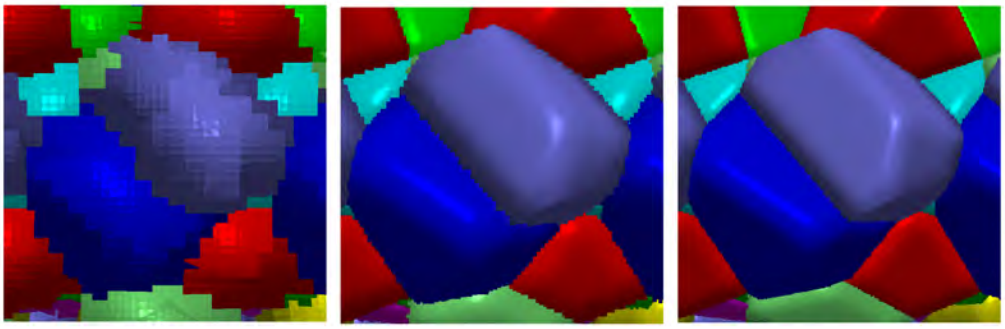

Figure 5.5: *A close-up view of a cell-based object rendered with the same cell map but with three different cell map dimensions. Note that the visual quality highly depends on the cell map resolution. Cell dimensions from left to right: 16, 64, 128.*

In Figure 5.6 a cell-based ninja head is rendered with and without specular self-reflection. The linear step size for first order rays is set to 1/512. The specular self-reflection is rendered only with one reflection bounce and the linear step size for the second order reflection rays is 1/128. This object is rendered with a cell map resolution of 128x128x128 and the object space is split into 64x64x64 tiles. Consequently the cell-based ninja head needs about 34 MB texture memory. The computational time for viewpoints with a pixel coverage of 100 percent and by rendering all cells is for the first hit rendering about 67 and for the self-reflection rendering about 29 frames per second. For close up views by rendering about 3 tiles (containing 8 cells per tile) per axis the frame per second for the first hit is about 129 and for the specular self-reflection about 39. In Figure 5.7 the same cell-based ninja head from example in Figure 5.6 is rendered with the same parameter setup but with different texture maps [23]. The cell-based dragon model from Figure 5.8 is rendered with a cell map resolution of 64x64x64 and the object space is split into 128x128x128 tiles. The parameter setup for the ray tracing remains the same as mentioned for the examples from Figure 5.6. All provided features (parallax, occlusions and silhouettes) and effects (specular self-reflection and refraction) are pointed out in Figure 5.9.

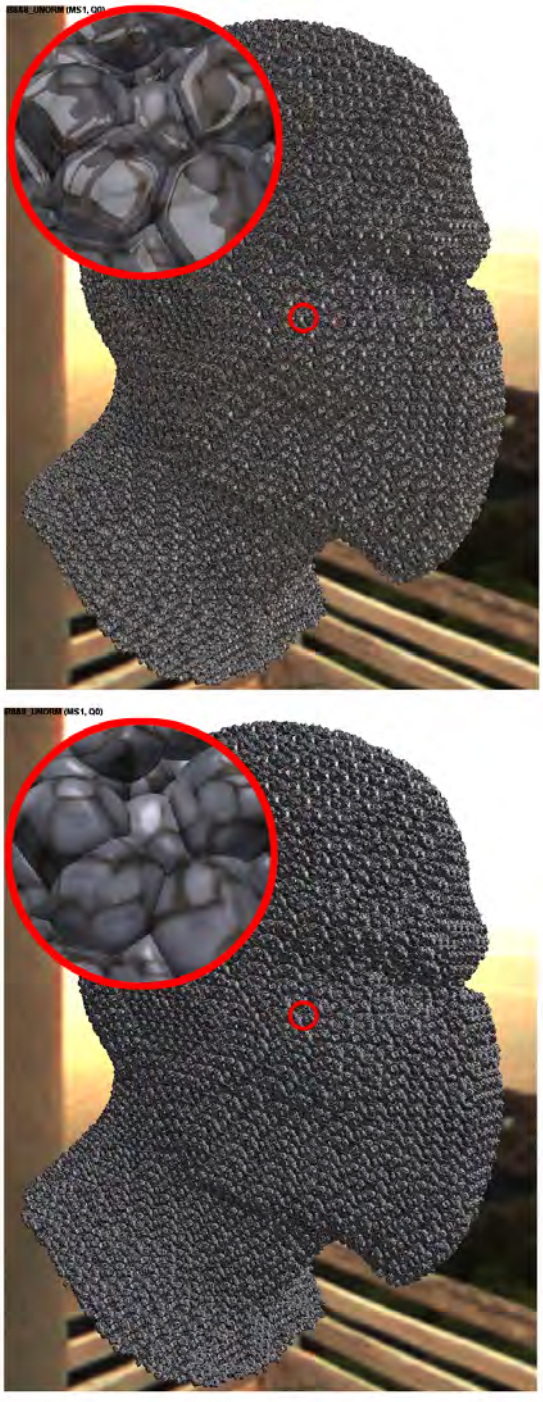

Figure 5.6: *A comparison between a cell-based ninja head rendered with(top) and without(bottom) specular self-reflection. The cell dimension is 128x128x128 and the object space is split into 64x64x64 tiles. The three-dimensional cobble stone texture is from [23]*

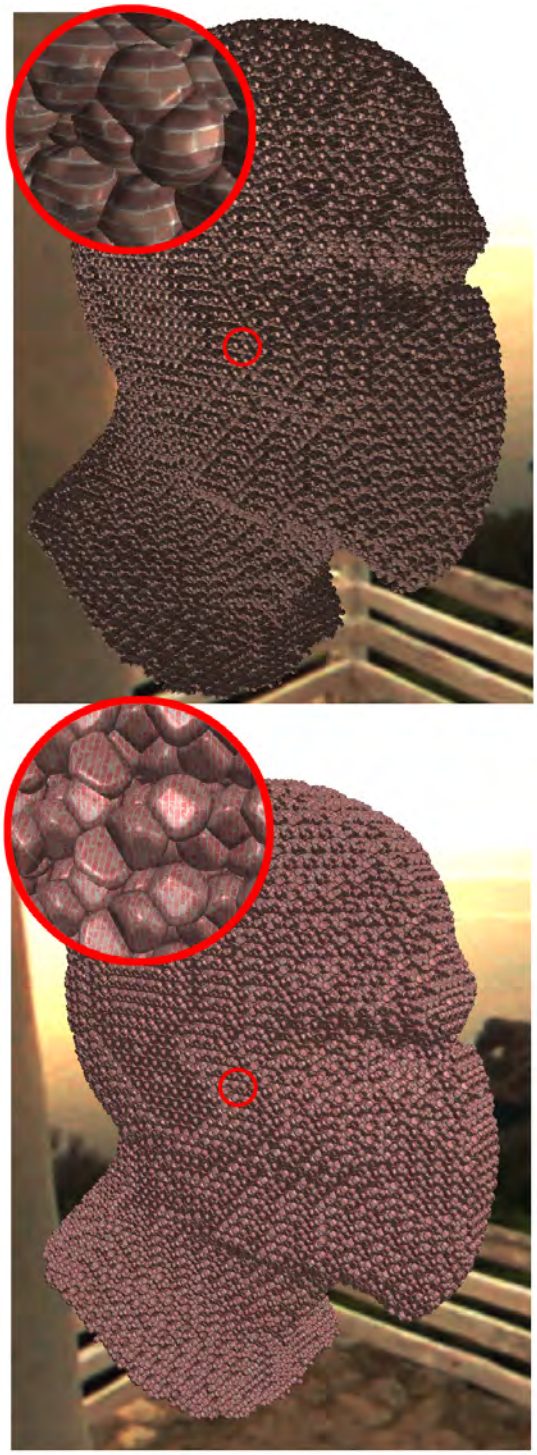

Figure 5.7: *A cell-based ninja head rendered with two different textures [23] . The cell dimension is 128x128x128 and the object space is split into 64x64x64 tiles.*

Figure 5.8: *A cell-based dragon model rendered with specular self-reflection (top) and refraction (bottom). The cell dimension is 64x64x64 and the object space is split into 128x128x128 tiles. The color texture is taken from [23] .*

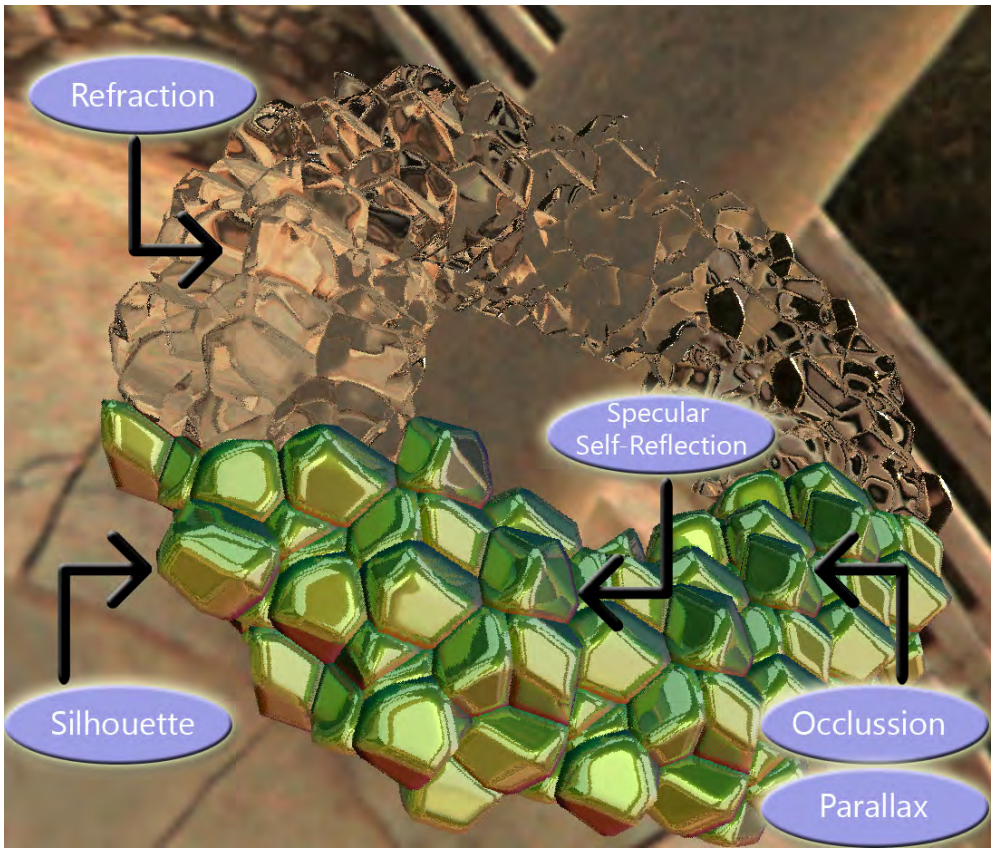

Figure 5.9: *A cell-based torus rendered with specular self-reflection (bottom) and refraction (top). Note the correct parallax, occlusions and silhouettes.*

„Nobody can go back and start a new beginning, but anyone can start today and make a new ending.„

Maria Robinson

Chapter 6

Conclusion

6.1 Summary

First this thesis gave a survey of mesostructured rendering techniques and cell-based texturing. Additionally every mesostructured rendering technique was categorized by its provided features. This thesis proposed an cell-based approach to model and display objects with a repetitive surface structure with a high visual quality. First the basic cell-based datastructure was explained and how to precompute all the needed information stored in 3D textures. For displaying the cell-based objects a ray tracer was proposed. To achieve more interesting looking surfaces the cell-based ray tracer was extended by two sophisticated effects: the specular self-reflection and refraction. With the fundamental understanding of the basic concept the technical point of view was introduced by showing the implementation details about the precomputational process. Last but not least the cell-based approach was analyzed based on the computational time, memory consumption and visual quality by showing some high resolution results.

6.2 Future Work

We proved that effects such as specular self-reflection and refraction could be easily integrated to the ray tracer displaying cell-based objects. Due to ray tracing other sophisticated effects such as caustics, translucency, self-shadowing and ambient occlusion to name a few could be a possible enhancement for the cell-based ray tracer.

In this thesis only static cell-based objects were discussed. An enhancement could be to provide also deformable cell-based objects for animations. We tested cell-based ray tracing on animated objects and discovered that for extreme deformations surfaces disappear. This happens because the rays should be wrinkled somehow depending on the deformation.
The precomputation time is depending on the cell map resolution and the number of tiles. Changing these parameters changes the look of the resulting cell-based object. For a high resolution setup the precomputation needs more than some hours. To achieve a smooth workflow in the modeling stage the precomputation has to be optimized to provide a fast response to model changes.

List of Figures

List of Tables

List of Algorithms

Bibliography

[1] Ed Catmull, "*A Subdivision Algorithm for Computer Display of Curved Surfaces"*, Ph.D. Thesis, University of Utah, 1974.

[2] L. Szirmay-Kalos and T. Umenhoffer, *"Displacement Mapping on the GPU - State of the Art",* Computer Graphics Forum, 2008.

[3] J. F. Blinn: "*Simulation of Wrinkled Surfaces*", In Proceedings SIGGRAPH 78, pp. 286-292, 1978.

[4] http://www.voronoi.com/wiki/index.php?title=Voronoi_Applications

[5] T. Akenine-Möller, E. Haines: "*Real-Time Rendering*", Second Edition, A.K. Peters Ltd., 2002.

[6] http://jerome.jouvie.free.fr/OpenGl/Projects/Shaders.php

[7] http://www.surlybird.com/tutorials/TangentSpace/index.html

[8] N. L. Max: "*Horizon mapping: shadows for bump-mapped surfaces*", The Visual Computer 4, pp.109–117. 1988.

[9] T. Kaneko, et al: "*Detailed Shape Representation with Parallax Mapping*", In Proceedings of ICAT 2001, pp. 205-208. 2001.

[10] T. Welsh: "*Parallax mapping with offset limiting: A per pixel approximation of uneven surfaces*", Tech. rep., Infiscape Corporation. 2004.

[11] R. L. Cook: "*Shade trees*", In SIGGRAPH '84 Proceedings, ACM Press, pp. 223–231. 1984.

[12] R. L. Cook, L. Carpenter, E. Catmull: "*The reyes image rendering architecture*", In Computer Graphics (SIGGRAPH '87 Proceedings), pp. 95–102. 1987.

[13] L. Wang, X. Wang, X. Tong, S. Lin, S. Hu, B. Guo, H. Shum: "*View-Dependent Displacement Mapping*", ACM Trans on Graphics, Vol. 22, No. 3, 2003 (Siggraph '03 Proceedings). 2003.

[14] F. Policarpo, M. M. Oliveira: "*Relief Mapping of Non-Height-Field Surface Details*", ACM SIGGRAPH 2006 Symposium on Interactive 3D Graphics and Games, Redwood City, CA, pp. 55-62. March 2006.

[15] F. Policarpo, M. M. Oliveira, J. Comba: "*Real-Time Relief Mapping on Arbitrary Polygonal Surfaces*", ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games, Washington, DC, pp. 155-162. April 2005.

[16] M. M. Oliveira, F. Policarpo: "*An efficient representation for surface details*", Tech. rep. RP-351, Universidade Federal do Rio Grande do Sul. 2005.

[17] S. D. Porumbescu, B. Budge, L. Feng, K. I. Joy: "*Shell maps*", ACM Transactions on Graphics 24, pp. 626–633. 2005.

[18] Peachey, Darwyn R., "*Solid Texturing of Complex Surfaces*", Computer Graphics, Vol. 19, #3, pp 279-286, 1985

[19] S. Worley, *"A Cellular Texture Basis Function",* In Proc. of SIGGRAPH 1996, ACM Press, 291–294, 1996

[20] S. Worley, "*Practical Methods for Texture Design*", Texture and Modelling: A Procedural Approach, Third Edition, pp 179 – 201, 2003

[21] F. Nielsen, "*An Interactive Tour of Voronoi Diagrams on the GPU*", Shader X6 Advanced Rendering Techniques, pp 539 – 556, 2008

[22] K. Crane, I. Llamas, S. Tariq, "*Real-Time Simulation and Rendering of 3D Fluids*", GPU Gems 3, pp 633 – 675, 2007

[23] http://johanneskopf.de/publications/solid/textures/index.html

[24] K.Engel, M.Hadwiger, J.M.Kniss, A.E.Lefohn, C.R.Salama, D.Weiskopf. "*Real-time Volume Graphics*". ACM SIGGRAPH 2004 , pp 108-138, 2004

[25] www.alexeilebedev.com/macrotips/orange.jpg

[26] R. S. GLANVILLE "*Texture Bombing*" In GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics, Addison-Wesley, 2004

[27] J. Legakis, J. Dorsey, S. Gortler "*Feature-Based Cellular Texturing for Architectural Models*", Proc. of the 28th annual conference on Computer graphics and interactive techniques, pp 309 – 316, 2001

[28] http://www.autodesk.com

[29] A. Kaufman, D. Cohen, R. Yagel, "*Volume Graphics*", IEEE Computer, Vol. 26, No.7, pp 51-64, July 1993