REAL-TIME VISUAL SIMULATION OF VOLUMETRIC SURFACES

Ville Timonen
Master's thesis
Computer Science
University of Kuopio
The Department of Computer Science
November 2006

UNIVERSITY OF KUOPIO, the Department of Computer Science
The Degree Programme of Computer Science

Timonen, V.: Real-time visual simulation of volumetric surfaces
Master's thesis, 122 p., 1 appendix (19 p.)
Supervisor of the Master's thesis: Ph.D. Mauno Rönkkö
November 2006

Keywords: graphics, rendering, real-time, relief mapping, volumetric surfaces, translucency, transparency

The increase in reprogrammability and processing power of modern graphics hardware has advanced the real-time rendering techniques for volumetric surfaces. This thesis offers an introduction to real-time relief mapping, covering different techniques used to improve intersection searching and visual appearance. The techniques are implemented and their applicability is analysed.

The main contribution of this thesis is the presentation of a relief mapping based rendering technique capable of simulating translucent features on volumetric surfaces. Optical behaviour of translucent materials is discussed in detail, and the simulation model is gradually completed based on physical authenticity. The simulation model proves efficient in simulating large and small-scale translucent details on surfaces, even when observed from close distance, in a way that is not achievable through polygon-based methods. Real-time suitability is proven by presenting an implementation that achieves real-time framerates on commodity graphics hardware.

Preface

This thesis is made for the Department of Computer Science of University of Kuopio in autumn 2006. The production of the thesis was supervised by Mauno Rönkkö, to whom I would like to express my gratitude.

In Kuopio 08.11.2006

Author

Terms and abbreviations

Bounding box	A shape that represents the maximum volume of a surface. The simulated surface is hence bound by the bounding box.
Fragment	The smallest primitive whose appearance can be independently set. A fragment is occasionally equal to a pixel.
Fragment shader	A program that determines a color and an optional depth value of a fragment. A fragment shader is executed by a GPU. Synonyms: pixel shader, fragment program
GPU	Graphics processing unit
Index of refraction	A property of a translucent material that defines its optical behaviour. It is defined by the speed of light in vacuum relative to the speed of light in the material. Synonyms: optical density
Normal map	A map containing surface normal vectors expressed in a tangent space. It can be used as a texture by graphics hardware.
Polygon	A planar graphics primitive. Its shape is defined by the associated vertices that represent the corners of the polygon.
Tangent space	A coordinate system private to a polygon.
Texel	An element of a texture. Synonyms: texture element
Vertex	A corner of a polygon. Different properties can be assigned for a vertex, of which at least a coordinate value is mandatory.
Vertex shader	A program ran on a GPU that processes vertex data. Synonyms: vertex program
Viewing ray	Vector between a fragment and a viewing point.

Contents

1 INTRODUCTION **6**

1.1 Real-time computer graphics 7

1.1.1 Basic principles of real-time graphics 7

1.1.2 Coordinate systems 9

1.1.3 Color and normal mapping techniques 11

1.2 Graphics hardware 14

1.2.1 Hardware categories 14

1.2.2 Architecture of reprogrammable hardware 14

1.3 Approaches to simulate geometric complexity 17

1.3.1 Displacement mapping 17

1.3.2 Offset mapping 17

1.4 Thesis contribution and recent related work 21

1.4.1 Volumetric opaque surfaces 21

1.4.2 Translucency 21

2 RELIEF MAPPING **23**

2.1 Principle 24

2.1.1 Introducing height data 24

2.1.2 Applying height data 26

2.2 Intersection searching 29

2.2.1 Linear search 29

2.2.2 Binary search 31

2.2.3 Linear height field approximation 33

2.3 Advanced intersection searching 35
2.3.1 Adaptation 35
2.3.2 Distance information 37
2.4 Self-shadowing 40
2.4.1 Simple self-shadows 40
2.4.2 Soft self-shadows 42
2.5 Summary 47

3 REFLECTIVE AND REFRACTIVE SURFACES 48

3.1 Light optics 49
3.1.1 Reflection and refraction 49
3.1.2 Intensities of reflection and refraction 51
3.1.3 Light scattering 52
3.2 Implementation 54
3.2.1 Viewing ray transformations 54
3.2.2 Viewing ray tracing 55
3.2.3 Simulating scattered light 57
3.3 Environment mapping techniques 60
3.3.1 Cube maps 60
3.3.2 Perspective dependent maps 61
3.4 Summary 64

4 TRANSLUCENT FEATURES ON VOLUMETRIC SURFACES 65

4.1 Surface sampling 66
4.1.1 Relief mapping two layers 66

4.1.2 Refraction 68
4.1.3 Reflection 70
4.2 Applying the samples 75
4.2.1 Composition 75
4.2.2 Occlusion 78
4.3 Performance 82
4.3.1 Environment 82
4.3.2 Results 82
4.3.3 Analysis 84
4.4 Summary and discussion 86

5 SUMMARY AND DISCUSSION **92**
5.1 Summary 92
5.2 Discussion 94
5.3 Future work 96

REFERENCES **99**

A SHADER PROGRAM **102**
A.1 Vertex shader 102
A.2 Fragment shader 103

1 INTRODUCTION

This chapter gets the reader acquainted with real-time computer graphics as a preparation for the later chapters. A reader already familiar with implementing real-time graphics can skip the first introductory sections of this chapter.

A brief introduction to the structure of 3D scenes is given along with the common terminology. The relation of the different coordinate systems or spaces commonly used in 3D graphics is also laid out. *Color* and *normal mapping* rendering techniques are briefly discussed as examples of polygon rendering techniques.

An introduction to different types of graphics hardware used for rendering real-time graphics is given. Architecture layout and a description of modern reprogrammable graphics processing units are presented as well.

Two different techniques used to increase visual complexity of height varying surfaces are discussed: *displacement mapping* [Coo84] [CCC87] and *offset mapping* [Wel04]. These techniques along with their flaws are represented as alternative or competing techniques to the *relief mapping* technique discussed in more detail in Chapter 2.

In the end of the chapter, the contribution of this thesis is discussed in contrast with relating work.

1.1 Real-time computer graphics

1.1.1 Basic principles of real-time graphics

Real-time computer graphics refers to graphics rendered right prior to displaying it on screen. Real-time graphics is most often used in interactive applications, where the rendered scene has to react to user input immediately, and the exact graphical content cannot be known in advance. Computer and video games are probably the most well known applications of interactive real-time graphics.

Scenes in non-real-time graphics, as opposed to real-time graphics, are usually designed and constructed before the actual rendering process. Non-real-time computer graphics is usually rendered as batch jobs during a long period of time. As an example, non-real-time graphics is often used for movie effects, where the graphical content is known in advance and rendering times significantly larger than the length of the produced clip can be tolerated.

In real-time graphics, the focus is on performance, as the whole scene has to be rendered dozens of times in a second to retain fluent and pleasant visual appearance. Different rendering techniques are usually applied for real-time and non-real-time graphics, real-time techniques often trading quality for performance.

A number of different rendering primitives can be used in real-time graphics, *polygon* being the most prominent one. Polygons have proven to be a flexible and a practical way to represent 3D objects. A polygon in computer graphics represents a plane with three or more corners, i.e. *vertices*. The shape of any solid object can be represented with polygons, as demonstrated in Figure 1.

The final appearance of an object depends on what sorts of techniques are used to render the polygons. Increase in detail for graphics rendered in polygons can only be achieved either by increasing the polygon count, or by improving the polygon rendering techniques. A usual scene involves other types of objects in addition to polygonal objects, such as lights. Lights are taken into account during the rendering of polygons according to a lighting model, and are not usually

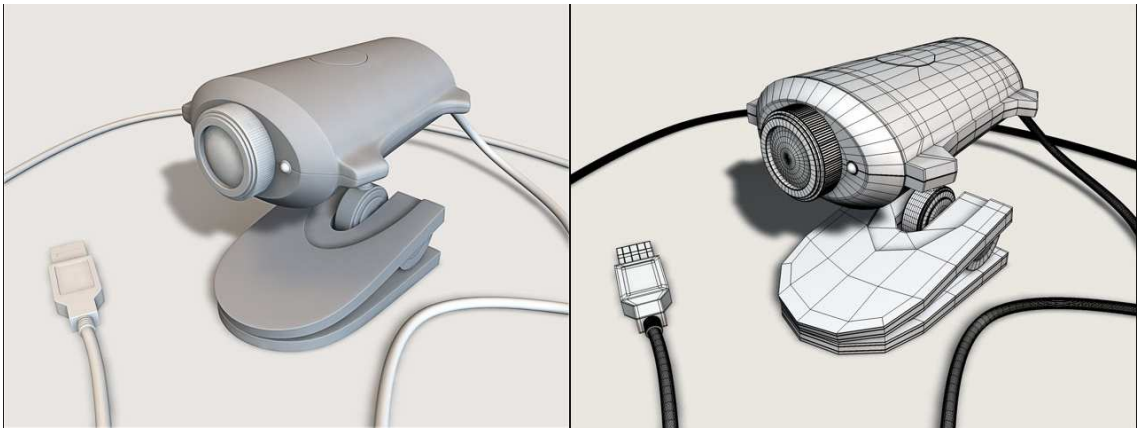

Figure 1: Wireframe representation (right) of a solid object (left) [Kje05]

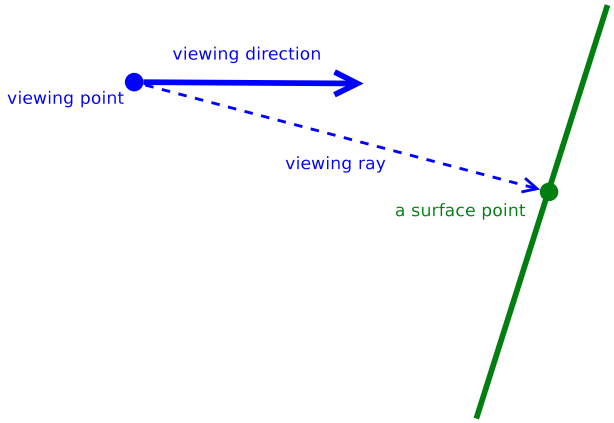

Figure 2: A viewing point, a viewing direction, and a viewing ray demonstrated

rendered independently.

A *viewing point* is the point in the scene which the scene itself is observed at. A viewing point is equal to the position of an eye or a camera in a scene. *Viewing direction* is equal to the direction of the eye or the camera, and determines the direction which the scene is viewed at from the viewing point.

Viewing ray is the vector along which a certain point in the scene is viewed at. Although one might be tempted to think that a viewing ray is equal to the viewing direction, a viewing ray is the vector between the viewing point and the point being rendered, and as such does not depend on the viewing direction. Figure 2 demonstrates the difference between the the three terms introduced here.

A polygon can be further divided into *fragments*. Usually a fragment represents the smallest primitive whose appearance can be independently set. Hence, the purpose of a polygon renderer is to determine proper values for fragments within a polygon.

1.1.2 Coordinate systems

The rendering process of a 3D scene utilizes at least 3 different *coordinate systems* or *spaces*. One is the local coordinate system of an object, i.e. the *object space*. Polygons of a static object retain static vertex coordinates, even if the position and direction of the object moves relative to other objects. *Model space* is a synonym for object space.

When the scene is composed of several objects, object space coordinates need to be transformed into a *world space* or an *eye space*. Vertex coordinates expressed in an eye space change whenever the object is moved, rotated, or scaled. The expression "eye space" refers to objects being in a space relative to the viewing point, i.e. to the eye or the camera. Whether the term "world space" or "eye space" is used, it is important to note that all objects within the scene are expressed uniformly in the same space.

As display devices today are only able to display 2D images, another coordinate system is needed to represent the visual area of a display. This is usually called a *clip space*. Only after the conversion from an eye space to a clip space, are polygons ready to be rendered.

Some rendering techniques require yet another coordinate system: a *tangent space*. A tangent space is a coordinate system for individual polygons within an object, and thus becomes the lowest-scale coordinate system introduced here. Figure 3 lists the hierarchy of the different coordinate systems. Even though coordinate system transformations are usually done from lower-scale spaces towards the clip space, it is possible – and necessary for certain polygon rendering techniques – to convert objects such as light sources and the viewing point into tangent space.

The Z-axis of a tangent space is usually chosen to be of the same direction as polygon's normal vector. In such a case it is appropriate to choose X and Y-axes so that they conform to the texture coordinate axes on the surface. Figure 4 demonstrates the tangent space axes for a polygon.

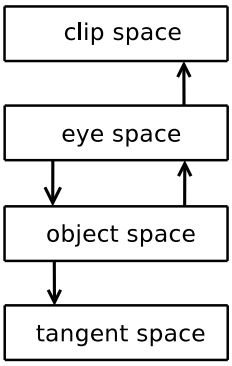

Figure 3: Hierarchy of coordinate systems

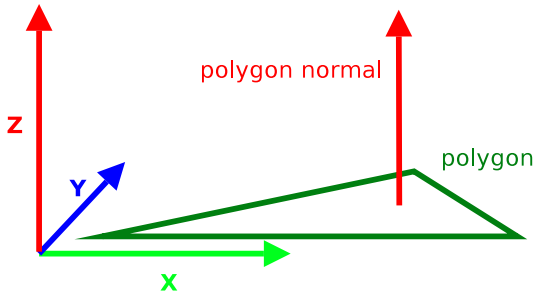

Figure 4: A tangent space for a polygon

1.1.3 Color and normal mapping techniques

When polygons are used to represent the surface of a 3D object, they are usually trying to give an illusion of a surface more complex than a plane. Since a polygon is essentially a flat surface, several different rendering techniques have been introduced to improve the illusion of complexity during the history of computer graphics.

One of the most popular techniques to improve polygon detail is *texture mapping* or *color mapping*. Color mapping incorporates the use of a color map with the polygon rendering. Instead of using a single or a per-vertex color value throughout the polygon surface, color values for individual fragments are fetched from the color map during the rendering. A more thorough exploration of different color mapping techniques can be found in a survey [WD97].

Another more recent and a popular rendering technique improving the illusion of complexity is *normal mapping*. Just as with color mapping, additional surface information is brought to the polygon via a map. Instead of color values, a *normal map* consists of fragment specific normal vectors. Instead of using per-polygon or per-vertex normal vectors, lighting equations are computed against the fragment specific normal vectors fetched during the rendering of a polygon from the normal map. Although being able to represent only rather small-scale details of the surface, this technique greatly improves the surface's ability to react properly to light sources. A more detailed introduction to normal mapping and an example implementation is shown in [HS99]. Figure 7 shows a color-mapped surface. Figure 8 shows a color-mapped surface with normal mapping.

Figure 5: A color map representing a brick wall [Clo06]

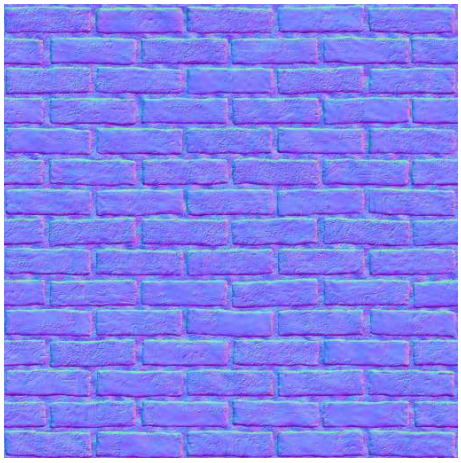

Figure 6: A normal map for a brick wall [Clo06]

Figure 7: Polygon surface with color mapping. The images are produced using the color bitmap shown in Figure 5

Figure 8: Polygon surface with color and normal mapping. The images are produced using bitmaps shown in Figures 5 and 6

1.2 Graphics hardware

1.2.1 Hardware categories

Graphics rendered in real-time is rendered to a *framebuffer* in order to be shown on screen. The framebuffer usually resides in a separate piece of hardware in a computer, designed to provide graphics for the display. This hardware is referred to as graphics hardware from now on.

The simplest kind of graphics hardware is not able to do any part of the graphics rendering itself, but instead only interfaces the framebuffer for the graphics application ran on the central processing unit (CPU) in an operating system. With such hardware, the application is responsible for rendering the entire scene and producing the final pixel values in the framebuffer. Some graphics hardware is able to do simple 2D graphics operations itself, such as alpha blending bitmaps stored in the video memory of the hardware.

Graphics hardware with 3D acceleration capabilities is able to perform most of the rendering process on its graphics processing unit (GPU). Such hardware usually renders the graphics using polygons. Unfortunately, early hardware possessing 3D acceleration capabilities was only able to use predefined rendering methods. This disposed the application developers of the ability to control most of the rendering process, forcing them to create effects using a limited set of provided hardware features. Graphics units employing 3D acceleration features were, however, significantly more efficient than CPUs, and vastly increased rendering performances were achieved.

GPUs today offer the possibility to reprogram most of the rendering pipeline. This conveniently allows programmers to implement effects and rendering techniques of their own with little limitations. Graphics still has to be rendered in polygons, but fragments within a polygon can be independently set with a customizable program.

1.2.2 Architecture of reprogrammable hardware

Knowledge of graphics hardware architecture is necessary, when reprogramming rendering pipeline on hardware supporting it. Figure 9 shows a simplified description of a rendering pipeline used

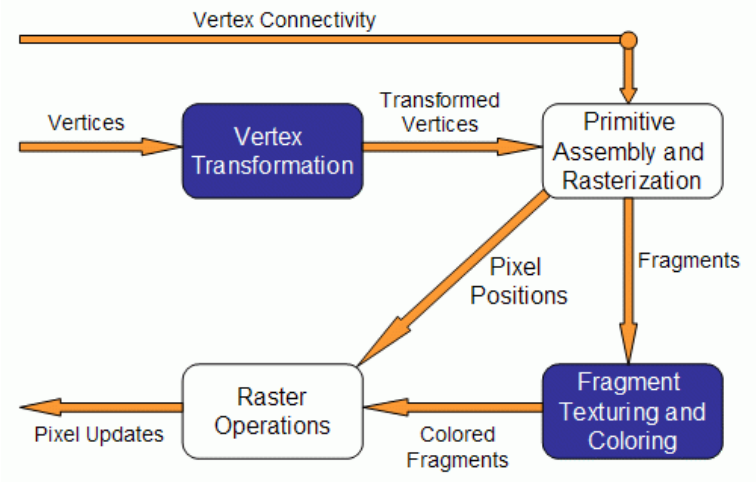

Figure 9: A simplified architectural description of a reprogrammable GPU [Fer04]

in today's reprogrammable GPUs. Boxes with blue background represent stages of the pipeline reprogrammable by the graphics application.

Vertex transformation stage processes vertex data, and a program providing this phase is called a *vertex program* or a *vertex shader*. Vertex data incoming to the stage can include a space coordinate, normal vector, color, and texture coordinate values for individual vertices. The purpose of this stage is to process the input data, provide information for the fragment program, and transform the object space vertices into the eye space. Usually per-vertex light calculations are performed in this stage. A vertex shader is not capable of accessing any texture data.

A *fragment program* or a *fragment shader* is responsible for implementing the stage `Fragment Texturing and Coloring`. A fragment shader gets its input indirectly from a vertex shader in an interpolated form between the vertices. A fragment shader determines the color value for the processed fragment, and optionally a depth value, when it does not lay on the flat polygon surface. The visibility of the fragment is not determined by the fragment program, and neither is the possible alpha blending calculated. Any effects that are applied per-fragment, however, are implemented in the fragment program. A fragment shader is able to access texturing units, and utilize texture data.

Associating fragments with pixels might help in understanding the scale of a fragment, but a fragment does not necessarily represent a pixel on the framebuffer. For example, graphics hardware

may render the scene with finer detail than in pixels, and apply post-processing techniques to compose the final appearance of a pixel on the framebuffer from several fragments. Such a technique is a form of anti-aliasing. In addition, a fragment shader has no access to the pixels in the framebuffer.

Depending on the polygon count of a scene, fragments are often significantly smaller primitives in a scene than polygons. This implies that, depending on the effects used, more fragment than vertex processing occurs for each frame. Modern hardware is geared towards fragment processing by employing more fragment processors than vertex processors, as stated in [Don05].

In OpenGL applications, fragment and vertex programs can be supplied for the graphics hardware natively through OpenGL 2.0 and through extensions for OpenGL 1.4 and later. OpenGL supports the *OpenGL Shading Language* (GLSL) for programming the rendering pipeline. A more detailed description of the rendering pipeline and GLSL can be found from [Fer04].

1.3 Approaches to simulate geometric complexity

1.3.1 Displacement mapping

In order to render complex surfaces accurately, varying height of the surface has to be taken into account. One rather trivial approach to this problem is to split the surface into smaller polygons that conform to height differences of the surface. This method cannot really be considered a rendering technique, since the produced polygons are rendered with regular methods; only the amount of polygons is increased to enhance the illusion. This approach is usually called *displacement mapping*, but the term is occasionally used for other techniques as well. A more comprehensive coverage of displacement mapping is offered in publication [Coo84], and an implementation in [CCC87].

Even though displacement mapping gives the best possible geometric equivalence to the surface being simulated, its suitability for real-time applications is poor due to the stress it applies on vertex pipelines of graphics hardware. The usual approach to solving performance issues in real-time graphics is to minimize the amount of polygons and implement desired effects via rendering techniques that are applied in the rendering phase of polygons. This is mostly because handling polygons involves more processing than is necessary for small-scale details that can be adequately simulated during the rendering of larger polygons.

If the complexity is simulated in a technique implemented in the fragment shader, computation naturally focuses on polygons near the viewing point where they cover more fragments than polygons far away from the viewing point. This is ideal since a more accurate simulation is usually in order for polygons near the viewing point. However, this is an advantage the displacement mapping does not have, decreasing its usability in real-time applications.

1.3.2 Offset mapping

A rendering technique called *offset mapping* or *parallax mapping* also takes varying height of a surface into account, and as such is able to simulate geometrically complex surfaces to a certain

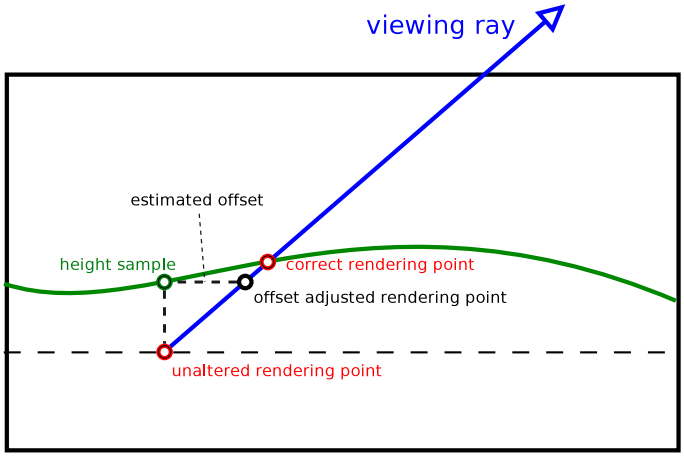

Figure 10: Correcting the sampling point by an offset approximation

degree. Figure 10 demonstrates the offset correction of a surface sampling point. When a surface that has height fluctuations is viewed along the viewing ray, the point that is observed is the `correct rendering point` instead of the `unaltered rendering point`.

The fragment in Figure 10 should be rendered as if it were the `correct rendering point`, which means that color and normal samples should be fetched from texture coordinates corresponding to this point. A height sample from the `unaltered rendering point` is used to approximate the sampling point offset along the viewing ray as Figure 10 demonstrates.

The approximation is correct for portions of the surface where height differences remain relatively small. Figure 11 demonstrates an erroneous offset approximation on a surface that fluctuates rapidly. Another problem arises when the viewing ray intersects the surface in a steep angle. In such a case, the intersection point is estimated using a height value sampled possibly so far away from the intersection point that it has no relation to the height value on the correct intersection point. The worst-case scenario is that the surface gets mapped with nearly random coordinate offsets, resulting in a discontinuous and irregular texturing. Even though the problem can be controlled to an extent by limiting the offset values as proposed in [Wel04], the visual result is not significantly better than when the height values are ignored altogether for steep viewing angles.

As the technique relies on adjusting texture coordinates independently on each fragment, reprogrammable graphics hardware is necessary for the customized fragment shader. Height values for

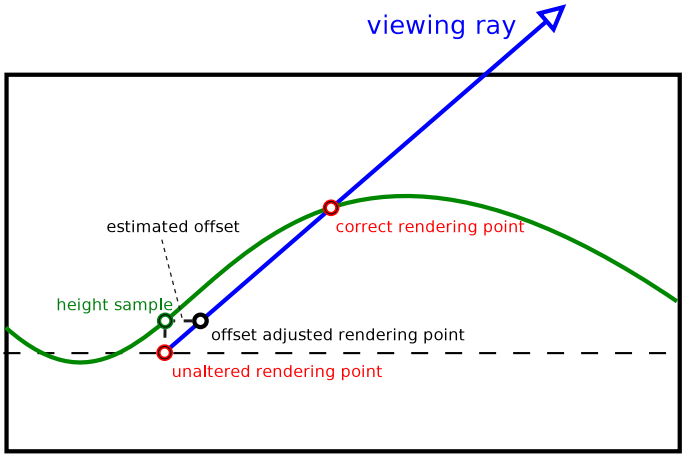

Figure 11: An erroneous offset approximation on a rapidly fluctuating surface

Figure 12: A height map for a brick wall – the image is blurred to soften the rapid height variations on bricks' edges [Clo06]

the surface can be supplied in the alpha channel of a normal map, or in a separate texturing unit. In order to calculate offsets, the viewing ray has to be expressed in the same coordinate system as the texturing coordinates, preferably in the tangent space.

Offset mapping requires little extra computation, and is a feasible technique in real-time applications on surfaces that do not fluctuate rapidly. Figure 13 demonstrates the results of offset mapping on a brick wall. A more comprehensive exploration of the offset mapping technique is represented in [Wel04].

Figure 13: A brick wall rendered using offset mapping. The image is produced using bitmaps shown in Figures 5, 6, and 12

1.4 Thesis contribution and recent related work

1.4.1 Volumetric opaque surfaces

The increase in processing power and reprogrammability delivered by the latest graphics hardware has allowed the simulation of volumetric surfaces in real-time graphics rendering. Recently, the simulation of surfaces having varying, non-planar, height has been under extensive research. Polygon rendering techniques utilizing a height map for the purpose of simulating such surfaces are known as relief mapping techniques.

In Chapter 2 of this thesis, I demonstrate relief mapping through the implementation of linear and binary searches [POC05], linear height field approximation [Tat06], hard self-shadowing [POC05], and soft self-shadowing [Tat06]. The use of distance information as proposed in [Don05] is discussed as well. Scenarios, where the techniques are applicable and efficient, are discussed. Also, advices on what to avoid and what to strive for, when employing the techniques, are offered. No new rendering techniques are proposed in this respect.

Real-time implementations for previously problematic surface properties, such as sub-surface scattering, have been proposed recently. For example, publication [BC06] demonstrates the real-time simulation of sub-surface scattering. The rendering of human skin, volumetric fog, and light scattering is discussed in [GG04]. Such surface properties are not further discussed in this thesis.

1.4.2 Translucency

Real-life objects and surfaces often consist of transparent or translucent features of some sort, thus methods for simulating translucency are necessary for the production of realistic graphics. Transparent objects or surface features have always been challenging to simulate in computer graphics. This is mostly due to the complicated visual phenomena seen in transparent surfaces, resulting from the unique behaviour of light within and especially on the boundaries of optical materials. Transparent features often interact with the environment, and such interaction is hard to simulate, as well.

Due to the problematic nature of transparency simulation, physically correct simulation models have traditionally been seen only in prerendered graphics, mostly in systems based on ray tracing. The increase in processing power of modern graphics hardware has been accompanied by the arrival of real-time implementations for transparency simulation. These simulations often concentrate on simulating large areas of optically transparent or translucent materials, ocean water being a prime example [PA01] [Bel03] [YFCF06]. These simulations are based on simulating effects that dominate on large surfaces. Approximations, such as perturbed planar reflections, are used to suit the simulated surface, as described in detail in [VIO02]. Even though being realistic enough for simulating ocean water or lakes, the techniques lack the capability to simulate small-scale transparent features within objects.

Techniques to simulate fully transparent or translucent polygonal objects in real-time have been proposed in [CW05] and [SN06]. These techniques simulate the behaviour of light in a physically correct fashion, including the scattering of light. However, these techniques are only applicable to polygonal objects that are entirely translucent, and the techniques are highly dependent on environment mapping techniques, since the sampling of reflected and refracted rays is done from the environment maps alone. Because of these characteristics, the techniques are unable to simulate small-scale transparent features within objects and surfaces.

In Chapter 4 of this thesis, I propose a technique capable of simulating large and small-scale transparent features accurately and physically correctly on solid volumetric surfaces. The technique makes use of known relief mapping techniques, and the physical behaviour of light is simulated based on known optical physics, as described in Chapter 3. The simulation of transparent features is presented illustratively, the effects being added gradually. The meaning and importance of the effects are explained and demonstrated. A full implementation is offered and its performance analysed.

2 RELIEF MAPPING

Relief mapping is a polygon rendering technique used to simulate correctly height variations on a surface with real-time frame rates. The method is more suitable for real-time applications than displacement mapping, and it does not suffer from the offset mapping's inability to scale into simulating steep height fluctuations.

Relief mapping technique takes advantage of the modern graphics hardware and requires the graphics processing pipeline to be reprogrammable. The concept of relief mapping is presented as introduced in [POC05], along with an example implementation.

Two search methods, *linear* and *binary searches*, are discussed and analysed. Also *linear height field approximation*, as presented in [Tat06], is implemented and its effectiveness evaluated. Adaptation of search intensities is briefly explored. The use of *distance information* is introduced as presented in [Don05].

Finally, techniques for hard and soft self-shadowing are described and demonstrated via implementations. The hard shadowing technique is based on [POC05] and the soft shadowing technique on [Tat06].

2.1 Principle

2.1.1 Introducing height data

A complex surface can be made to react to light sources properly by the use of normal mapping technique. This is not, however, enough to produce an accurate simulation of a height varying surface. When compared to a flat surface, height differences can be thought to distort the texture maps on the surface. Hence, a logical approach to simulate height varying surfaces is to correct texture coordinates during texture sampling according to the local height of the surface. Rendering technique taking height information into account this way is usually called *relief mapping*.

Relief map in general means a map representing height or altitude information. A relief map, used for relief mapping, works like a normal map or a color map and consists of height data. The height data is used to recalculate texture sampling coordinates on a per-fragment basis. Relief maps can be represented as single component textures with 8-bit samples in graphics hardware, and are usually stored in the unused alpha channel of normal maps. As height maps begin to represent increasingly large height differences, it might become necessary to start using larger texture samples, such as 16-bit ones, or more color components for one height value. In the latter case, the height information can no longer be carried in the alpha channel of a normal map, but instead another texture unit has to be taken into use.

Relief maps usually represent height variation from the ground level of a polygon. In addition, the height value is not an absolute height, but instead a relative one within certain limits. This also conveniently optimizes the accuracy of height data in the limited space reserved for it. The limits for maximum height variations are usually supplied externally for the fragment shader. These limits form the *bounding box* of the surface. An example can be seen from Figure 14. The bounding box defines the maximum volume or depth of the surface that the rendering method is able to simulate. Samples of the relief map represent relative height values from -1.0 to 1.0, respectively from bottom to top of the bounding box. Figure 15 is an example relief map of a brick wall. It should be noted that the monochromatic image of the relief map is only a representation – relief map samples define relative height values, not colors.

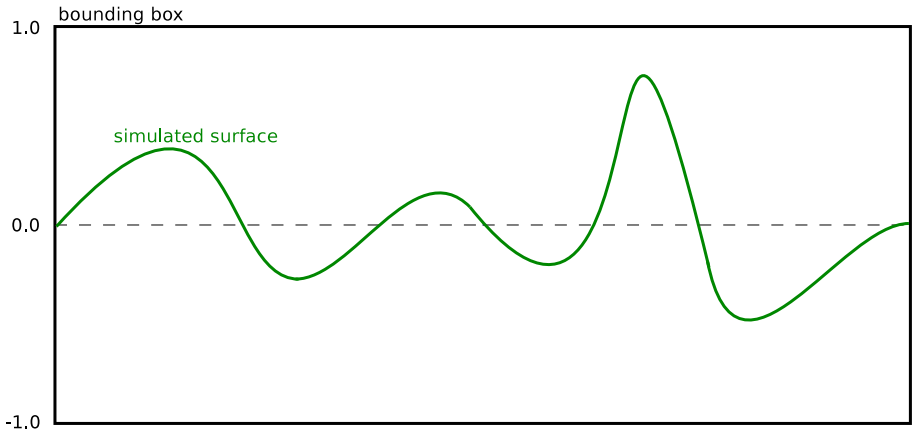

Figure 14: A bounding box

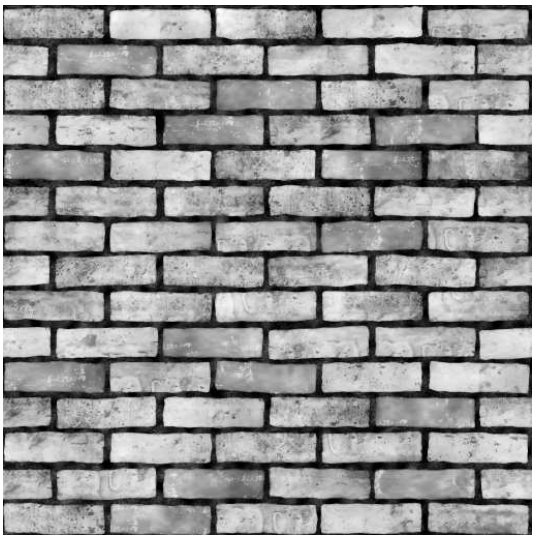

Figure 15: A relief map [Clo06]

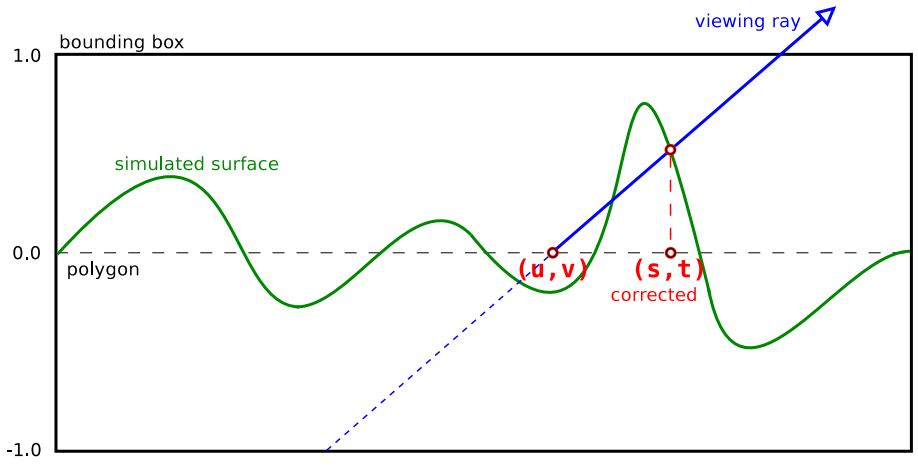

Figure 16: Correct sampling coordinates

2.1.2 Applying height data

The purpose of the height information is to aid the correction of the sampling coordinates. Figure 16 demonstrates the way sampling coordinates should be corrected. `Viewing ray` is the direction which the fragment is being viewed at and `(u,v)` is the original sampling coordinate. When the surface is being viewed along the viewing ray towards `(u,v)` the point that is perceived is the first intersection point of the height map and the viewing ray. When properly applying textures on the surface, `(s,t)` should be used as the sampling coordinate instead of `(u,v)`. Once the intersection point is found in the tangent space of the surface, the corrected texture coordinate can be obtained by simply discarding the component normal to the surface. Unfortunately, there is no trivial or computationally cheap way to derive the precise intersection point from the available surface data.

The approach to deriving the correct sampling point resembles more of the paradigm used in ray tracing. In traditional real-time computer graphics the question has always been “where in the screen does this object render to”, but in ray tracing the approach is opposite and can be described by asking the question “what objects constitute to the appearance of this point of the screen?”

Figure 18 demonstrates the increase in detail a relief mapping technique can deliver. The images on the left show a surface with normal and color mapping, and images on the right are rendered

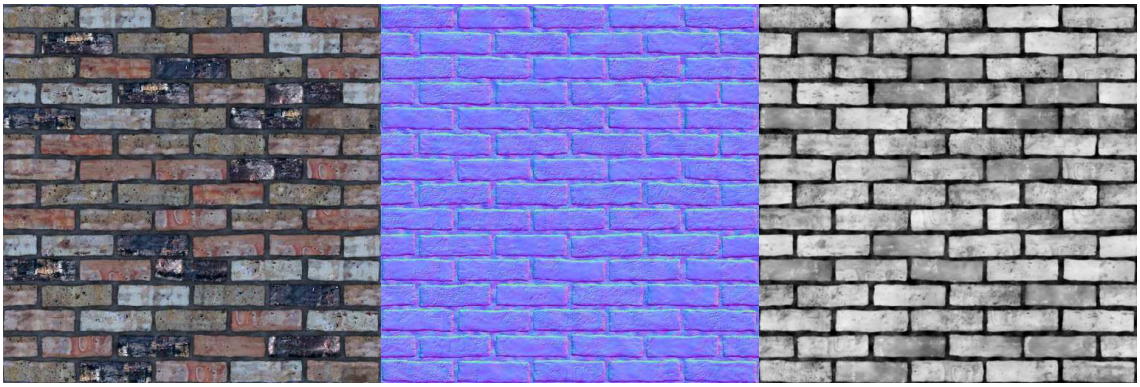

Figure 17: Color (left), normal (middle) and height (right) maps for a brick wall [Clo06]

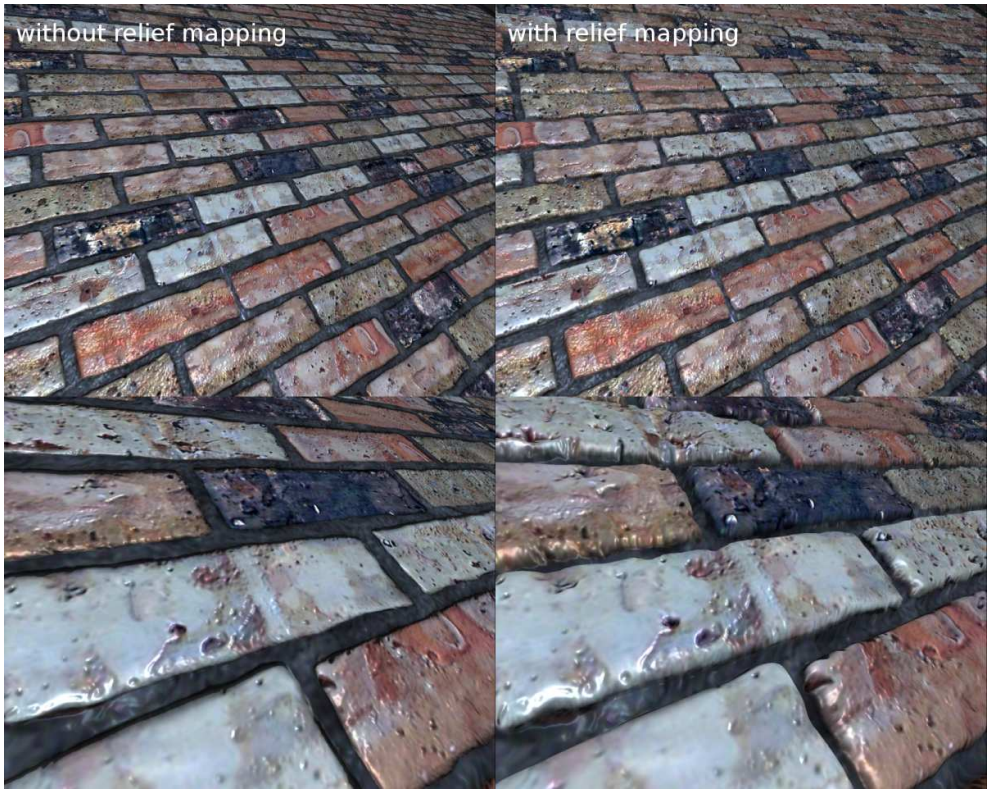

Figure 18: A normal- and color-mapped surface with and without relief mapping. The images are produced using bitmaps shown in Figure 17

using relief mapping as well.

2.2 Intersection searching

Searching the intersection point of a viewing ray and the surface is the single most important part of a relief mapping technique. It is solely responsible for both the accuracy and the time complexity of the technique. This section concentrates on different practical ways to search for the intersection point.

Search methods described here are iterative in nature and cannot find the exact intersection point in general. So far, no method being able to derive the precise intersection point has been introduced that is both practical and implementable with modern graphics hardware. The intersection problem resembles the one familiar from mathematics: finding a solution for an equation that is not analytical. Only numerical methods prove useful, but they give only approximations and require iterations.

Because the height data is arbitrary, methods have to rely on sampling the height data on different points in order to decide whether the intersection point has been reached. The logic according to which the sampling steps are selected is mostly responsible for how fast the intersection point is found within certain accuracy.

In practice, during each step, the height map is sampled and its height is compared against the height of the viewing ray on the sampling point. The texture sampling coordinate can be derived from the point along the viewing ray by discarding the component normal to the surface if the viewing ray is expressed in tangent space of the surface. The discarded component can be used directly as the height that the sampled height is compared against. The comparison needs to be done against the absolute height variation from the ground level, not the relative value stored in the height map. When two points, one below and one above the viewing ray, has been found, it is certain that there is at least one intersection point between them. This information is used to advance the approximation of the intersection point. Figure 19 illustrates this situation.

2.2.1 Linear search

Linearly searching the intersection point involves taking steps of predefined length along the view-

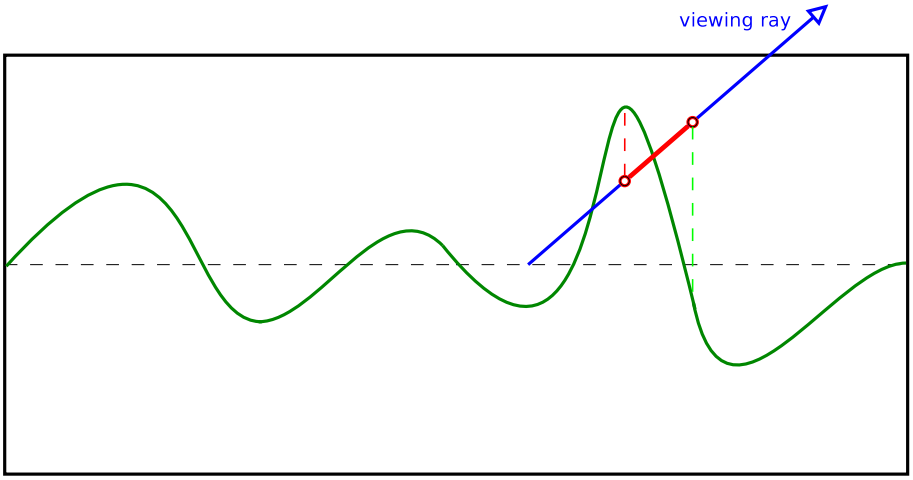

Figure 19: An intersection range

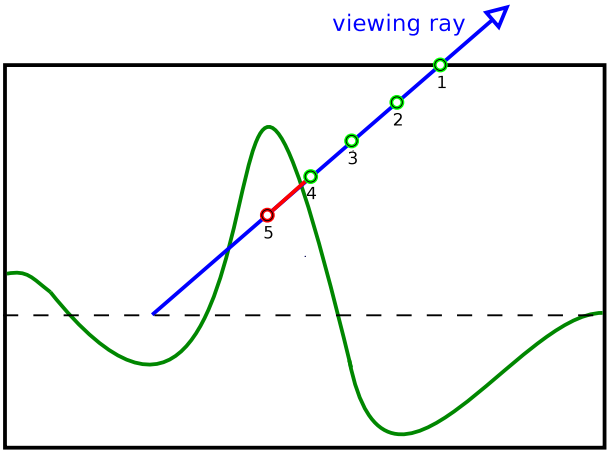

Figure 20: Progression of a linear search

ing ray. Since we have to find the first intersection point, it is only reasonable to start marching along the viewing ray from the edge of the bounding box that is closest to the viewing point. Linear search is especially suitable for starting the search, because it does not skip the first intersection point as easily as binary search. Figure 20 shows a usual progression of a linear search. The steps are equal in length and the fifth step is the first one under the surface level. This means that the viewing ray intersects the surface between points four and five. [POC05]

If a too large step size is chosen, the search might skip intersection points. Illustration of this case is shown in Figure 21. The search is able to find an intersection point, but not the first one, which

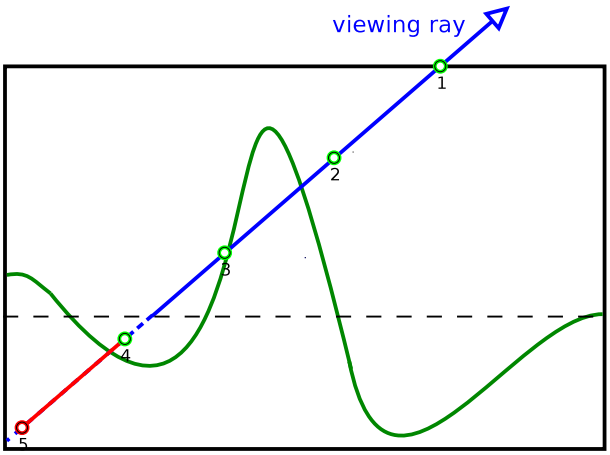

Figure 21: Linear search skipping the first intersection point

is a serious flaw. The step size should be chosen at least small enough for any serious aliasing caused by skipping not to occur. Unfortunately, it is impossible to avoid aliasing altogether no matter how small the step size is, especially on sharp edges.

Linear search should be avoided to be used as the method to find the precise intersection point. Even though iterations with linear search are computationally rather cheap, it is still a method with time complexity of O(n). Doubling the desired accuracy doubles the average number of iterations required, whereas this would require only an extra iteration with binary search. It should be noted, though, that a single iteration with binary search is computationally more expensive than with linear search. This means that doubling the amount of iterations with linear search is cheaper than adding an iteration with binary search up to a certain point. As a general guideline, when linear search is used in conjunction with binary search, it is reasonable to minimize the number of linear search iterations to the point where aliasing due to skipping of intersection points is adequately infrequent.

2.2.2 Binary search

With *binary search* the steps are chosen by halving the range containing an intersection point. In other words, the new step is chosen by taking the average of the last two points that bound an intersection point. If binary search is used as the first search technique, there is a high risk of

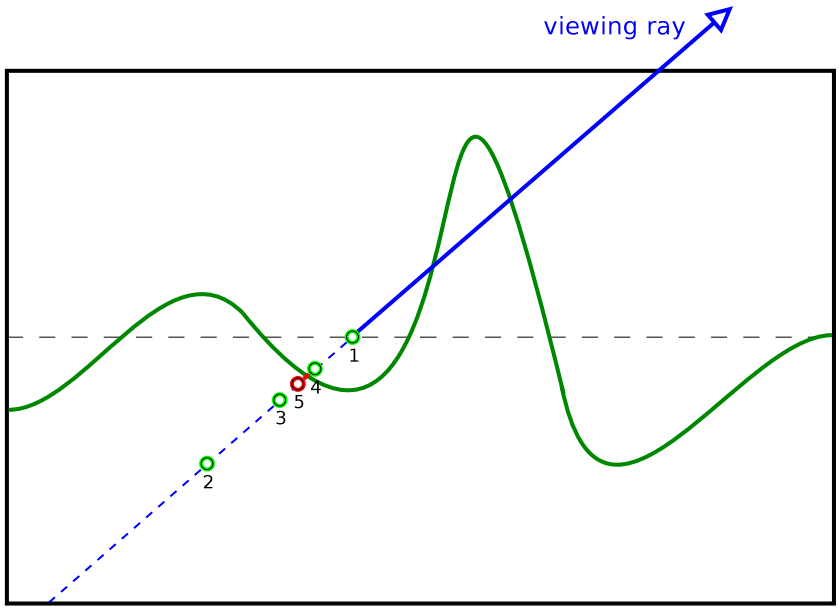

Figure 22: Binary search skipping the first intersection point

skipping an intersection point. An illustration of such a case is shown in Figure 22. The search proceeds to close in on the wrong intersection point.

Binary search is highly vulnerable to finding a wrong intersection point within a range that has more than one intersection point. The search is best suited for searching a range known to have only the needed intersection point. Usually this range is acquired via linear search. Unlike with linear search, the accuracy with binary search doubles on each iteration. However, iterations with binary search are computationally costlier than with linear search.

An effective combination of linear and binary searches could have a maximum of 8 iterations of linear search followed by 6 iterations of binary search. This way linear search achieves a resolution of $\frac{1}{8}$ of the height of the bounding box, and binary search is able to narrow the resolution for an intersection point down to $\frac{1}{8} \cdot \frac{1}{2^6} = \frac{1}{512}$. Assuming that height map samples were 8 bits in size, the height could get 256 different equally spaced values between the limits of the bounding box. In this perspective, the $\frac{1}{512}$ resolution of the used search can be considered sufficient. Exactly this kind of a search was used in the rendering of the surface in Figure 18.

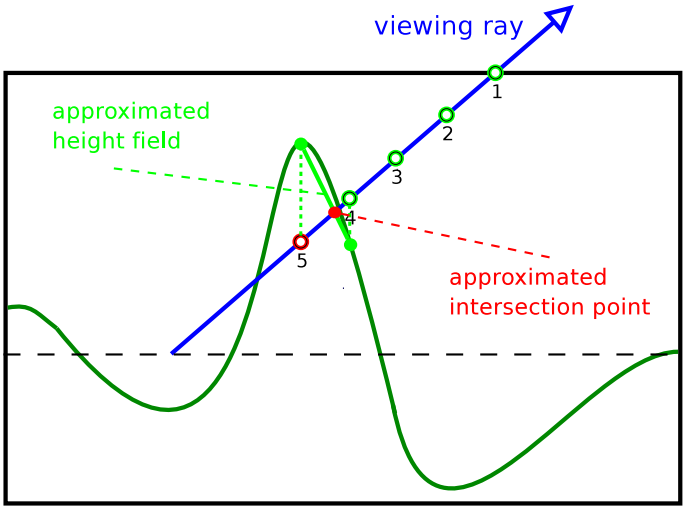

Figure 23: Linearly approximating a height field

2.2.3 Linear height field approximation

For best visual results, it might not be smart to use the latest sampling step as the intersection point when the search is finished. The approaches mentioned so far only make use of the fact that two sampling steps are below and above the surface. In addition to this information, also height values from these points are known. The height information does not require extra height map sampling, since it was already sampled during the intersection search, and it can be utilized in approximating the height field between the points.

If the surface can be assumed not to fluctuate arbitrarily between the last two sampling steps, a *linear approximation* can be made from the surface by interpolating the heights between the last two sampling points. The approximation is demonstrated in Figure 23. This assumption is true for most surfaces, and in any case, it is usually a better approach than using the latest sampling point. When the intersection of the viewing ray and the interpolated line is used as the intersection point, smoother visual results are achieved, as discussed in [Tat06]. This can be seen from Figure 24, where a surface is rendered using 4 iteration linear search followed by 2 iterations of binary search with and without linear height field approximation. Notable aliasing artefacts arise due to the inaccuracy of the search, as can be seen from the image on the left. Linear approximation can improve the visual quality significantly, as the image on the right demonstrates.

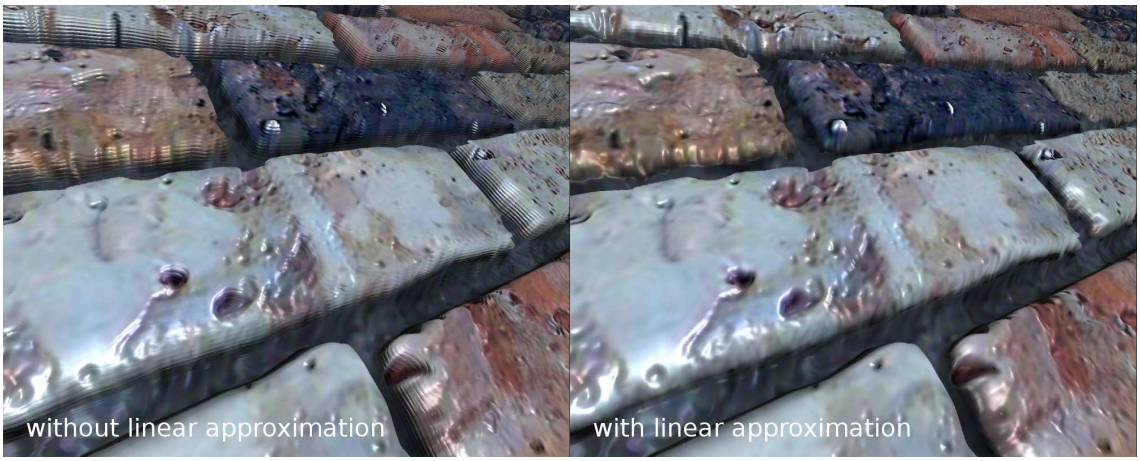

Figure 24: Visual results of linear approximation. The images are produced using bitmaps shown in Figure 17

The intersection point against the interpolated line has to be calculated only once: at the end of the search. This makes the method especially useful when used in conjunction with searches that use large number of iterations. The first reason for this is that the linear approximation does not lose its efficiency as the search grows more accurate. The second reason is that the static time complexity of the linear approximation diminishes in comparison to the time complexity of the search as it grows longer.

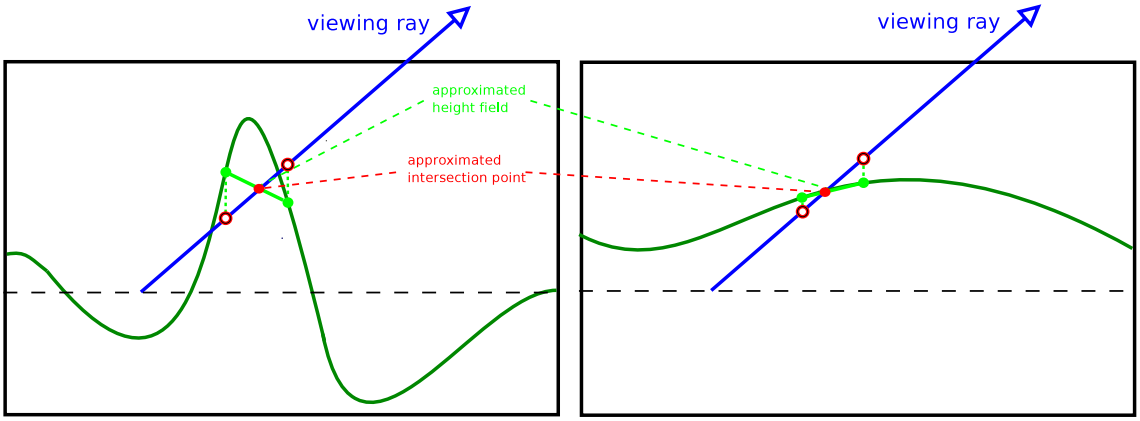

Figure 25: Height fluctuation of a surface affecting search accuracy

2.3 Advanced intersection searching

2.3.1 Adaptation

The appropriate search intensity depends on several factors, one of which is the surface height fluctuation. If surface height fluctuates rapidly, a more exhaustive search is needed to be certain that the encountered intersection point is really the first one, and that no skipping occurs. This affects the number of appropriate linear search iterations. In addition, if linear height field approximation is used, more closely spaced samples are needed from a highly fluctuating surface for the approximation to be accurate. Figure 25 shows approximations for two different surfaces with equally spaced samples. The approximation is more accurate for the surface that is more flat. This in part affects how far a binary search, which is responsible for the final intersection range, should be iterated. In general, intensities of both linear and binary searches should be dependent on the height fluctuation of the surface, so that less exhaustive searches are used on more flat surfaces.

The above problem can be solved by supplying the fragment shader with information that tells how intense a search should be used for different parts of the surface. The idea of externally supplying the number of linear search iterations for surfaces is presented in [Tat06]. The idea can be used for both linear and binary searches, since they are both dependent on the surface complexity the same way, as stated in the previous paragraph. For added flexibility, the search intensity information can be carried in textures the same way as normal, color and height information.

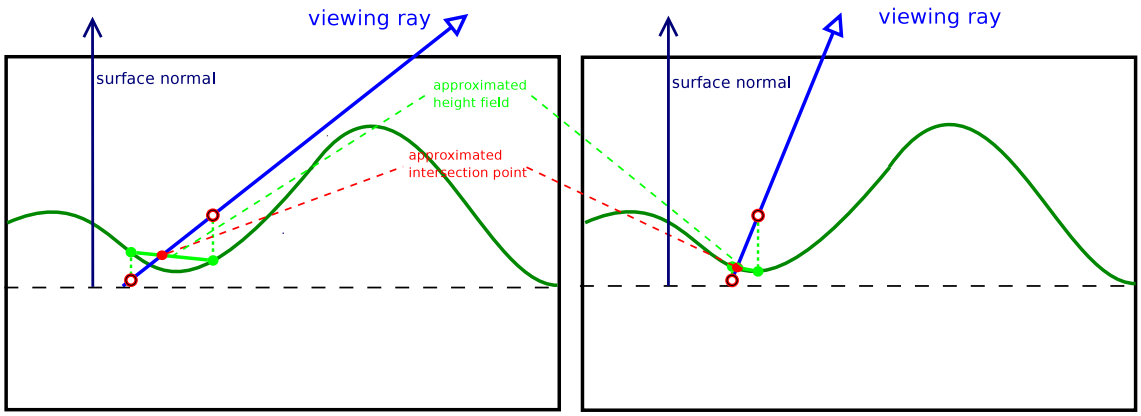

Figure 26: The angle of the viewing ray affecting search accuracy

It is usually appropriate to make the search intensity dependent on the perspective as well. The approach to adapting search intensity according to the surface distance and the angle between surface normal and viewing ray was presented in [Tat06]. Objects farther from the viewing point usually do not need as accurate simulation. When a surface is facing the viewing ray in a steep angle, a more accurate search is needed than when the viewing ray is close to the surface normal.

Figure 26 shows the same surface with different perspectives i.e. with different viewing rays. If the viewing ray faces the surface in a steep angle, more intersection points are likely to exist along the search path. This affects the appropriate number of linear search iterations in the same way, as when the surface height fluctuates rapidly. When the viewing ray is close to the surface normal, even long distances between sampling steps result in rather small distances along the polygon surface, and therefore they result in small deviations in corrected texture coordinates and linear surface approximations. Because of this, the intersection range does not have to be narrowed down as far, and not as many iterations are required with the binary search either. Intensities of both linear and binary searches should be made dependent on the angle between the surface normal and the viewing ray so that more exhaustive search is used when the angle is large.

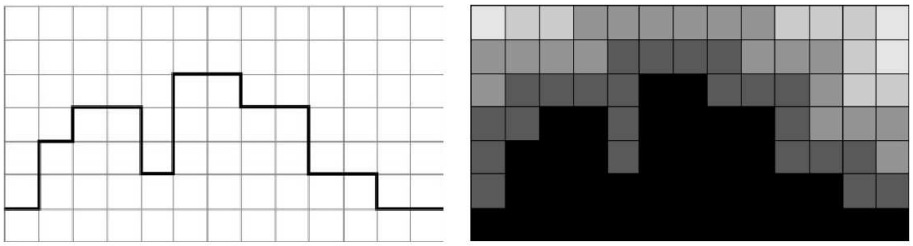

Figure 27: A 1D surface and its 2D distance map [Don05]

2.3.2 Distance information

Choosing proper steps during intersection searching is an important issue with relief mapping. Techniques explored before have to be balanced between performance and visual quality determined by how intense search is used. The technique introduced next is able to determine proper step length without the risk of skipping an intersection point.

Relief mapping's approach to rendering surfaces resembles more of that familiar from ray tracing. Because of this, it might not come as a surprise that techniques originally developed for ray tracing prove useful with relief mapping as well. One of these is *sphere tracing*, a technique presented in [Har96].

This technique introduces the surface with new type of data that tells how long steps can be taken from a certain point in the vicinity of the surface while still being certain that no intersection point is skipped. This data can be called a *distance map* and it stores the smallest distance to the surface from points in the bounding box. Figure 27 shows a visualisation of a distance map for a 1D surface. The image on the left shows a 1D height map for a surface, and the image on the right shows its distance map. Black texels tell that they are already under the surface, and the color of the grey texels tell the lowest distance to the surface, lighter texels indicating larger distance. Distance maps are discussed in [Don05].

Unfortunately, distance map has an extra dimension compared to the surface it is applied on. In Figure 27 for example, a 2D distance map is needed for a 1D surface. This feature becomes more significant when the used height map is 2D, which is usually the case, since we need to use 3D distance maps that often come with space complexity problems.

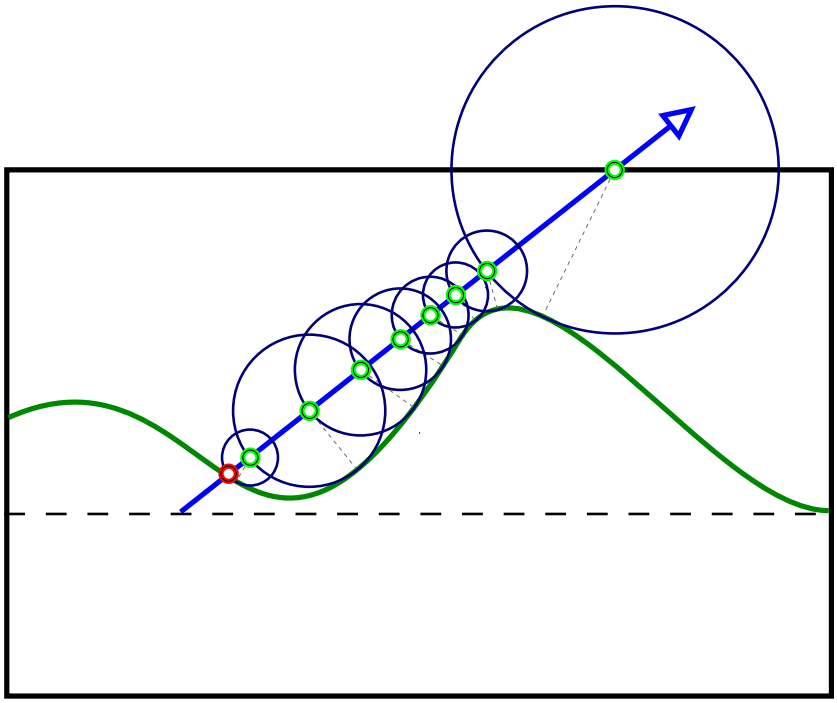

Figure 28: Progression of an intersection search utilizing distance information

The idea behind utilizing the distance information is to choose the step size for the next step marched along the viewing ray to be equal to the distance in the distance map at the point of the current step. Since the distance map value represents the distance to the closest point in the surface, no matter what the direction of the viewing ray is, an intersection point will not be skipped. Figure 28 shows a demonstration of an intersection search utilizing distance information. The search can be ended, when a certain number of iterations is reached, or when a predefined threshold distance is achieved. Intersection searching using distance maps is discussed in [Don05].

In addition to being the only technique introduced so far that can completely eliminate the risk of intersection skipping, the technique is also able to choose steps rather efficiently. If the real intersection point is close to being the nearest surface point, the search method converges on the correct intersection point in only a few iterations. However, if the surface gets close to the viewing ray without intersecting, the method may use excessive number of iterations without making any significant progress towards the intersection point.

The distance map is usually precomputed and supplied as a 3D texture much like normal, height

and color maps. An algorithm presented in [Dan80] is able to create a distance map in time complexity of O(n), where n represents the number of texels in the distance map.

There are a few problems involved in using this technique, space complexity being the major one. Since the distance map has to be 3D for a usual 2D surface, it requires a significant amount of memory. Fortunately good results can be achieved by using a texture that is only 16 or 32 texels deep [Don05], but it still makes the distance map 16 or 32 times larger than the height map, if used with the same 2D resolution. Texture compression available in modern graphics hardware can be used to remedy the space complexity problem to a certain extent, but will not be discussed here in any closer detail.

Another problem arises, when animated height maps are used. The distance map would have to be recomputed against every change of the height map, which is computationally impossible to implement for real-time applications. Animated height maps can be used to represent lively surfaces, such as water.

2.4 Self-shadowing

2.4.1 Simple self-shadows

Shadows play an essential role in visually realistic computer graphics. Numerous different shadowing techniques have been introduced that are usable in real-time rendering. Methods that rely on geometrically creating shadows by using shadow polygons that are clipped over object polygons, are applicable to relief-mapped surfaces as long as depth values for fragments are corrected during relief mapped rendering. Assigning depth values is discussed in [POC05].

This and other similar techniques are able to cast shadows from polygons only. When in-polygon geometry is introduced via a technique, it is the technique's responsibility to produce correct shadows on the surface cast by the surface. Fortunately producing hard self-shadows on relief mapped surfaces during the rendering is rather trivial.

The idea behind shadowing is to determine for each fragment whether it is lit by a light source or not. It is not lit, if a part of the surface is between the fragment and the light source. This can be easily determined by testing whether there is an intersection point between the light source and the point of surface being rendered. If the light source direction is expressed in tangent space, we can march along the light ray from the intersection point the same way as when searching the intersection point for the viewing ray. The most important difference with the search is that we only need to discover whether an intersection point exists; we do not need to know its exact location. [POC05]

Figure 29 demonstrates the logic according to which the lighting of the fragment is determined. Point `1` in the surface is behind part of the surface as seen by the light source, which means it is in a shadow. Point `2` instead is lit, and no intersection point can be found between the light source and the point. Because the precise intersection point does not have to be known, linear search can be used alone. Figure 30 shows a screen capture of an implementation of self-shadowing described above. Eight-step linear search is used to search the intersection.

This technique can be extended to environments having several dynamic lights. In such case,

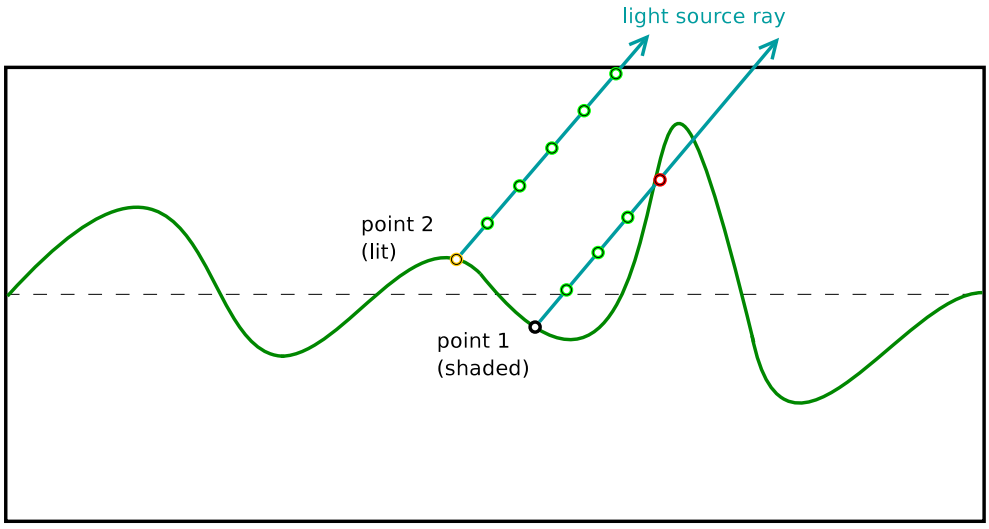

Figure 29: Two surface points, one lit and one in a shadow

Figure 30: A surface with and without hard self-shadows. The images are produced using bitmaps shown in Figure 17

the intersection test has to be done against every light source. This makes the technique's time complexity linear to the number of light sources. The shadow intensity can be determined by the number of light sources reaching the surface.

2.4.2 Soft self-shadows

The simple self-shadows described in the previous section are only accurate for infinitely small light sources, and thus have sharp edges. Light sources in real life are never points without dimensions. They usually have dimensions either by nature, like the sun, or by surrounding environment, like a lamp with a lampshade. In addition, indirect light sources, such as windows, can be represented as light sources with definite dimensions. When rendering surfaces in the vicinity of such light sources, fragments cannot be rendered as being simply lit or in a shadow, since they receive varying amounts of light.

The lighting technique described here determines the amount of light received by the fragment from volumetric light sources. The idea of incorporating this kind of lighting with relief mapping was introduced in [Tat06]. The technique can be implemented with the iterations used for searching hard shadows.

The amount of light received for a certain fragment depends on the surface point along the light ray that blocks the light most. Figure 31 demonstrates how a surface point blocks the visibility for a fragment. We can express the *blocking ratio* of a certain surface point by the ratio

$$\frac{h}{d} \tag{1}$$

which is a convenient choice for calculating the final light amount, as seen in the next paragraph. This is not precisely correct, since h does not represent the closest – i.e. normal – distance to the light ray, but it is usually a very good approximation. The error is insignificant in comparison to other error factors, such as the limitations of the search. In practice, the light ray can be marched with linear search to find the surface point with the smallest blocking ratio. The smaller the ratio is, the smaller the amount of light received by the fragment. Only the smallest ratio has to be

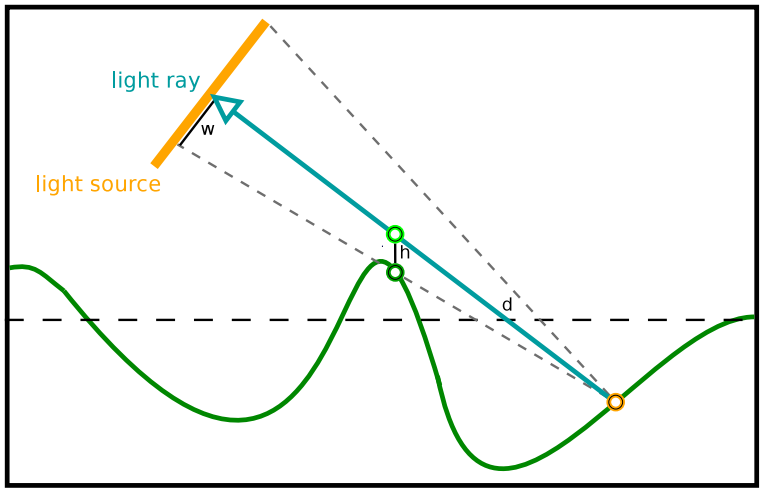

Figure 31: A surface feature partially occluding a light source

known, since this ratio solely determines the amount of light received.

In Figure 31, there are two parallel triangles sharing a common hypotenuse and the other leg. One triangle consisting of legs w and l where l represents the length of the light ray, and the other triangle consisting of legs h and d where h is an approximation mentioned in the previous paragraph. Equation

$$\frac{h}{d} = \frac{w}{l} \tag{2}$$

can be written for the leg ratios, due to the fact that the triangles are parallel. Since the blocking ratio given by equation (1) is already known, w can be solved as

$$w = \frac{h}{d} \cdot l \tag{3}$$

If w equals or exceeds the light source radius, the surface point being rendered receives light from the entire light source. If $-w$ equals or exceeds the light source radius, the surface point does not receive any light from the light source directly. In general, the amount of light is

$$\frac{h}{d} \cdot \frac{l}{r} + 0.5 \tag{4}$$

where r is the radius of the light source. This value should be clipped to a range of [0,1], and it

Figure 32: A surface with hard and soft self-shadows. The images are produced using bitmaps shown in Figure 17

represents the relative amount of light received from a light source.

Figure 32 demonstrates the results. Shadows rendered using this technique properly react to light source dimensions and to light distance. Shadows sharpen, when the light source gets farther away or when the light source gets smaller in radius. More importantly, the shadows react properly to surface features occluding the light source partially. The light source must be simplified to being rectangular in shape and facing the surface, but it is close enough approximation for most types of light sources. Extending the technique into multiple lights can be done by computing the light calculations for each light and combining the results the same way as with hard shadows.

Compared to shadowing with hard shadows, this technique is slightly more expensive computationally, because the blocking ratio has to be calculated in each search iteration, and we cannot stop iterating when an intersection point is confirmed to exist. In addition, this technique is highly

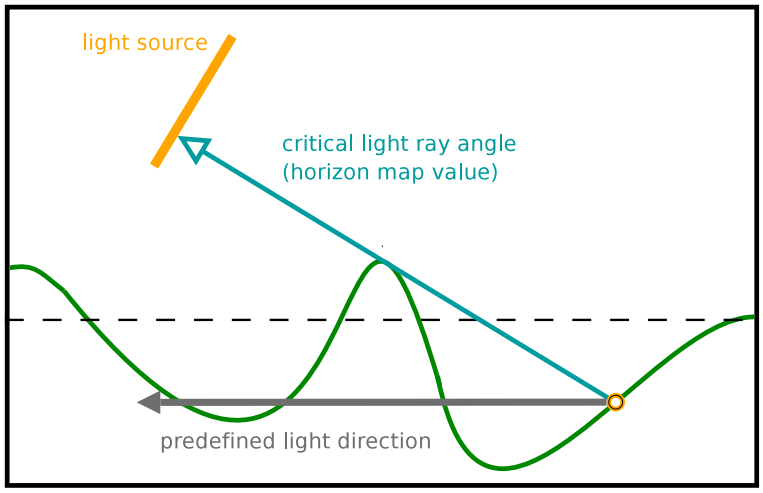

Figure 33: Using horizon maps to determine light visibility

dependent on the number of iterations used, much like with hard shadows. Intensifying the search trades performance for quality.

There are other shadowing techniques that are applicable to relief-mapped surfaces as well. One of them is *horizon mapping*. Horizon mapping does not share the same principle as relief mapping and the shadowing techniques described previously. It requires precomputed multi-dimensional horizon maps, and it is not dependent on the number of sampling iterations the same way as the previously discussed techniques. The logic according to which fragments are shadowed is similar to the techniques discussed in this section, but instead of focusing on computing the blocking ratio in real-time, horizon data is precomputed and stored into a 3D horizon map for certain predefined light ray directions.

Horizon map can be thought to contain the horizon altitude or direction information for each point of the height map for a predefined number of compass bearings i.e. light directions. This information can be utilized the same way as the blocking ratio, except that the horizon map value describes the critical angle of the light ray where the ray hits the surface. This equals to a blocking ratio of 0. We can then compare the critical angle to the real angle of a light and to its size. Figure 33 demonstrates this case. Usually 8 to 32 predefined light directions are used and horizon values are interpolated in between.

Horizon mapping suffers from the same problems as relief mapping with distance information; the horizon map is large and can only be applied on static height maps. An article about horizon mapping with 3D horizon maps using reprogrammable graphics hardware is presented in [For02].

2.5 Summary

In this chapter, we described the way geometric complexity can be efficiently and accurately visualized with the use of relief mapping polygon rendering technique. The rendering technique is based on changing the texturing coordinates per-fragment by finding the intersection point of the surface and the viewing ray.

Linear search is most suitable for starting the intersection search to avoid skipping of intersection points. Linear search accuracy should be chosen so that the aliasing due to intersection point skipping is acceptable, but the eventual accuracy should be refined with binary search instead of linear search. Linear approximation of the intersection point can be justifiably applied when its computational cost is small in comparison to the searches, i.e. when a large number of search iterations are used.

Adaptive techniques can be used to concentrate more exhaustive searches on problematic portions of the surface, such as portions with rapid height fluctuations or portions viewed at steep viewing angles. Adapting the searches for surface features usually requires extra information about the complexity of the surface.

Distance information can be utilized to greatly enhance the search accuracy and time complexity, but it requires multi-dimensional distance information maps that might come with space complexity problems. In addition, distance information maps are not suitable for animated or dynamic height maps.

Relief-mapped surfaces can occlude light from themselves, which can be simulated by the use of hard self-shadowing. Soft self-shadowing that depends on light size and distance is also implementable for relief-mapped surfaces, and produces realistic-looking soft shadows with decent accuracy.

3 REFLECTIVE AND REFRACTIVE SURFACES

In this chapter, optical behaviour of transparent and translucent materials is discussed. The focus is on characteristics that apply, when simulating appearance of a surface that has transparent or translucent features.

The progression of a light ray upon *reflection* and *refraction* is described, and vector form equations are given to aid in efficient calculation of reflected and refracted light ray vectors. Equations to compute reflection and refraction intensities according to the *Fresnel terms* are given as well.

Light scattering and *absorption*, while light travels in a translucent material, are discussed, and methods for implementing static and dynamic light scattering and absorption are offered. Implementational considerations for sampling the reflected and refracted light rays are presented also.

In the end of the chapter, two *environment mapping* techniques are discussed that can be used to simulate the external environment of a surface: *cube mapping* and *view dependent mapping*. An introduction to cube mapping can be found from [Kil99] and to view dependent mapping from [VIO02].

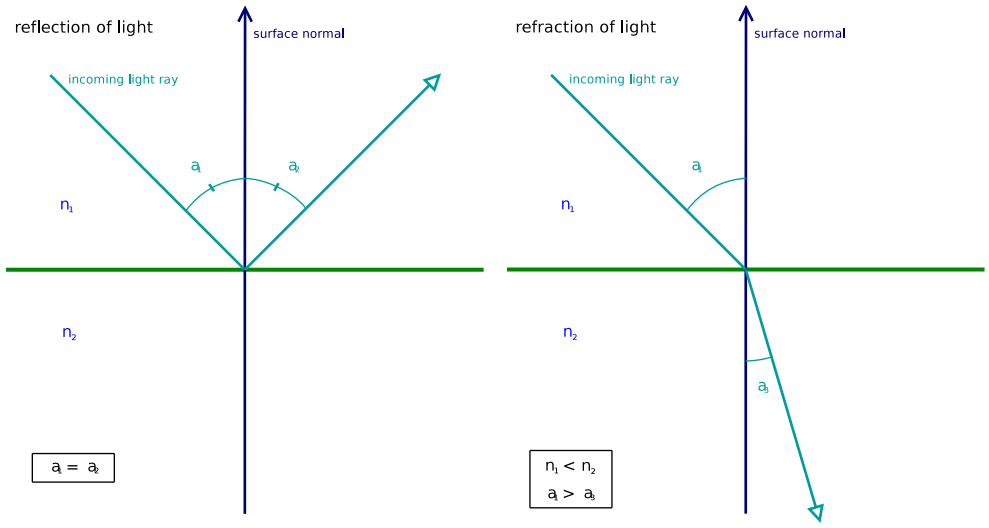

Figure 34: Reflection and refraction of light

3.1 Light optics

3.1.1 Reflection and refraction

Not all surfaces behave visually as if they were solid. Surfaces can be fully or partially transparent having varying *indices of refraction*. In this section, we discuss optical behaviour of transparent surfaces according to their refraction indices. Refraction index n for a material is defined by the ratio of the speed of light c in vacuum to the speed v in the material as [YF03]:

$$n = \frac{c}{v} \tag{5}$$

When a ray of light arrives at the boundary of two materials with differing indices of refraction, it can behave in two ways; it can bounce off from the junction or pass through. If the light ray bounces off from the junction, i.e. gets *reflected*, it proceeds at the same angle compared to the junction normal as it came in. Figure 34 in the left demonstrates this case. Another behaviour, called *refraction*, occurs, when the light passes through the material junction. In this case, the speed and the direction of the light ray changes as it surpasses the junction. When visually simulating surfaces with varying indices of refraction, especially the change in the direction of the light has to be taken into account in order to produce visually realistic results. Light tends to bend

towards surface normal when it passes into a material with a higher index of refraction. [YF03]

Equation [YF03]:

$$n_1 sin(a_1) = n_2 sin(a_3) \tag{6}$$

applies for the indices of refraction n_1 and n_2 of the materials and for the light angles a_1 and a_3. Figure 34 in the right demonstrates the case where light passes through the junction of materials. If n_1 is larger than n_2, a boundary angle exists, when

$$a_3 = 0^\circ \tag{7}$$

where the light is on the verge of being able to pass through the junction. From this we get that when

$$a_1 \geq arcsin(n_2/n_1) \tag{8}$$

no light can pass through and all light is reflected.

Index of refraction for a certain material varies for different wavelengths of light, an effect known as *dispersion* or *chromatic dispersion*. This has the consequence, among others, that refracted light rays for different colors are not parallel. If white light consisting of several wavelengths undergoes refraction, light rays of different colors spread out in different angles from the junction level. The variation of refraction index is material-specific, and no trivial formula for calculating refraction indices for different colors can be presented. Water for example has a refraction index of 1.329 (red) to 1.344 (violet) for different visible wavelengths. [YF03]

When simulating the effects of dispersion in computer graphics, it is usually adequate to use different indices of refraction for each color components, red, green, and blue. The visual influence of diffraction can vary from unnoticeable to significant, and is sometimes important enough to be simulated.

3.1.2 Intensities of reflection and refraction

When light arrives at the junction of two materials with differing indices of refraction, it can be reflected or refracted. In addition to being fully reflected or refracted, light can also be partially reflected and partially refracted at the same time. In this case, the sum of the intensities of the reflected and refracted rays must equal to the intensity of the incident ray, i.e. no loss of light energy occurs.

The ratios of refracted and reflected rays depend on the *polarization* of the light, and are defined by the *Freznel equations*. For s-polarized light the intensity coefficient for reflected ray is given by the equation [Bli77]:

$$R_s = \left(\frac{sin(a_3 - a_1)}{sin(a_3 + a_1)} \right)^2 = \left(\frac{n_1 cos(a_1) - n_2 cos(a_3)}{n_1 cos(a_1) + n_2 cos(a_3)} \right)^2 \tag{9}$$

and for p-polarized light the equation is [Bli77]:

$$R_p = \left(\frac{tan(a_3 - a_1)}{tan(a_3 + a_1)} \right)^2 = \left(\frac{n_1 cos(a_3) - n_2 cos(a_1)}{n_1 cos(a_3) + n_2 cos(a_1)} \right)^2 \tag{10}$$

with angles a_1 and a_3 as shown in Figure 34. The Freznel equations predict, for example, that surfaces tend to be more reflective when viewed at steep angles. This holds true for everyday observations, such as for the reflectivity of water.

Light used in real-time computer graphics can often be simplified to being unpolarized, which consists of equal amounts of s and p-polarized light. In such a case, the intensity coefficient R for reflected ray is the average of the two differently polarized light coefficients from equations (9) and (10) as given by the equation [Bli77]:

$$R = \frac{R_s + R_p}{2} \tag{11}$$

Since the intensities of reflected and refracted rays must add up to the intensity of the incident ray,

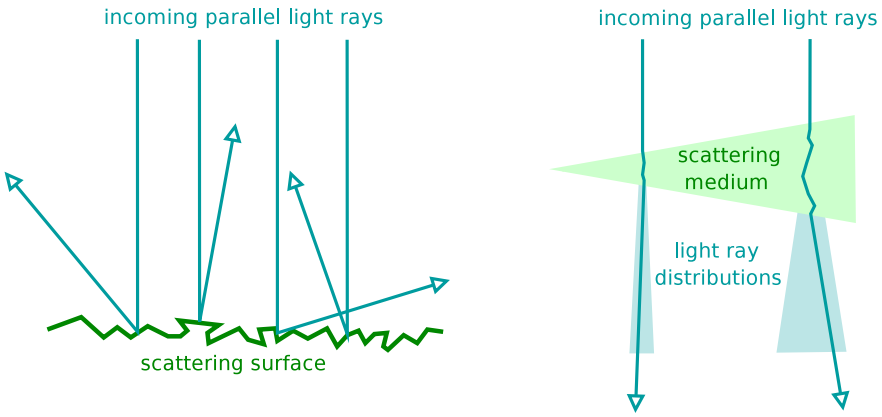

Figure 35: Light scattering from a surface (left) and in a medium (right)

the intensity coefficient T for refracted ray is given by

$$T = 1 - R \tag{12}$$

A more comprehensive introduction to optics can be found from [BW99].

3.1.3 Light scattering

Light, in practice, does not always follow its calculated path with mathematical precision. Light can scatter from its estimated path due to irregularities and impurities of the medium it travels in or of the surface it reflects from. Scattering of light from a surface can be mostly responsible for the lighting behaviour of the surface and thus the appearance, diffuse lighting being a good example of this. Occasionally scattering is restricted to an extent where reflections are distinguishable yet not sharp. The same effect can be seen when light scatters while travelling through a medium. If the material is uniform, the amount of scattering depends on the distance travelled in the medium. Figure 35 demonstrates the scattering of light on a surface and through a material. [YF03]

Since the irregularities and impurities responsible for light scattering are usually microscopic, they can be simulated as approximations. The perceived appearance of scattering resembles that of softening or blurring, which is leads to an assumption that simulating scattering can be done via image manipulation techniques such as blurring. Scattered reflections can be implemented

with a static degree of blurring, but scattering inside translucent objects needs to be simulated with different degrees of blurring, depending on the thickness of the material where light passes through it. [CW05]

Translucent material can also absorb light. If uniform material allows half of the light intensity pass through in a distance of d_0, half of this intensity gets through after the remaining of the light passes through another distance of d_0. From this we can conclude that light intensity decreases in half for every d_0 travelled in the material. Thus absorbed light intensity for distance d is given by the equation

$$I = 1 - \left(\frac{1}{2}\right)^{d/d_0} \tag{13}$$

The material can also have a hue, which means that it absorbs some wavelengths or colors more than others. This can be implemented by multiplying the absorption intensity by a color vector.

3.2 Implementation

3.2.1 Viewing ray transformations

Light changes direction upon reflection or refraction. When a surface, that has reflective or refractive behaviour, is viewed along the viewing ray, the appearance is partially or fully determined by the transformed viewing ray. Since the observed appearance is actually light, laws of refraction and reflection can be applied directly, as light follows its calculated path in both ways. In order to compose the appearance of a point, both reflected and refracted rays have to be traced.

In addition to knowing the viewing ray, also the surface normal at the point of reflection or refraction has to be known. If in-polygon details are not simulated, i.e. no relief or normal mapping techniques are applied, per-vertex normal vectors are adequate. If this is not the case, normal vectors fetched from the normal map can be used in the ray transformations. Indices of refraction for the mediums need to be supplied as well.

Assuming that the viewing ray vector and the normal are expressed in the same space, preferably in the tangent space, the reflected ray $\mathbf{v_{refl}}$ is given by the equation [Shi02]:

$$\mathbf{v_{refl}} = \mathbf{v} + 2(\mathbf{v} \cdot \mathbf{n})\mathbf{n} \tag{14}$$

where $\mathbf{v}$ is the viewing ray and $\mathbf{n}$ is the surface normal. Since the dot product

$$\mathbf{v} \cdot \mathbf{n} = cos(a_1) \tag{15}$$

where a_1 is the same as in (6), equation (14) can be written equally as

$$\mathbf{v_{refl}} = \mathbf{v} + 2cos(a_1)\mathbf{n} \tag{16}$$

In order for the equations to work both $\mathbf{v}$ and $\mathbf{n}$ must point in the same direction, that is towards the surface or out of it.

Refracted ray $\mathbf{v_{refr}}$ can also be derived in a vector form [Shi02]:

$$\mathbf{v_{refr}} = \frac{n_1(\mathbf{v} - \mathbf{n}(\mathbf{v} \cdot \mathbf{n}))}{n_2} - \mathbf{n}\sqrt{1 - \frac{n_1^2(1 - (\mathbf{v} \cdot \mathbf{n})^2)}{n_2^2}} \tag{17}$$

or written equally as [Shi02]:

$$\mathbf{v_{refr}} = \left(\frac{n_1}{n_2}\right)\mathbf{v} + \left(cos(a_3) - \frac{n_1}{n_2}cos(a_1)\right)\mathbf{n} \tag{18}$$

where n_1 and n_2 are the indices of refraction and a_1 and a_3 the ray angles as given by equation (6). It should be noted that $cos(a_3)$ can be written as [Shi02]:

$$cos(a_3) = \sqrt{1 - \left(\frac{n_1}{n_2}\right)^2 (1 - cos^2(a_1))} \tag{19}$$

which means that with the help of equation (15) no trigonometric functions need to be used.

When equations (9) and (10) need to be known as well, equation (18) is usually preferred over (17), because the the cosines can be recycled. If refracted vector alone needs to be known, and computing the cosines is not necessary, equation (17) can be used directly for efficiency.

Incoming light is also affected by refraction and reflection, and therefore light source intensities and directions should be corrected for physically correct simulation. For example, if part of a surface is occluded by translucent material, a fraction of the incoming light gets reflected away and weakens the intensity of the light arriving at the surface. The rest of the light changes direction according to the laws of refraction before illuminating the surface.

3.2.2 Viewing ray tracing

In addition to knowing the transformed viewing ray, a precise or an approximate end point of the ray has to be sampled in order to compose the visual appearance of the fragment. A trivial case of tracing a transformed viewing ray happens when the transformed ray can be traced within a polygon. Consider a case in Figure 36, where two-layer surface with refractive upper layer is rendered as a single polygon. After the first surface junction point is derived using relief mapping,

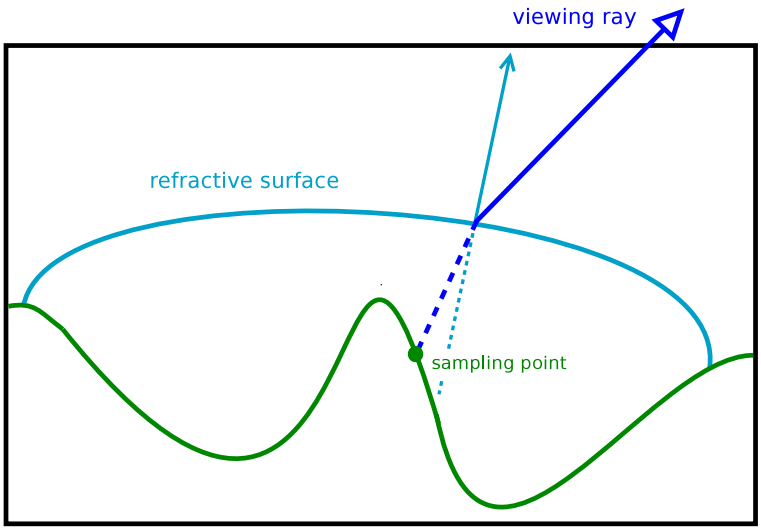

Figure 36: Refraction within a layered surface

for example, the refracted viewing ray can be traced the same way as the original ray. Such a trace yields precise results within the limits of the search, and no approximations need to be done.

In some cases, the ray intersects the boundary of the translucent material more than once. The problem with such cases is, that if the boundary is both refractive and reflective, as often is the case, the viewing ray splits in two each time a junction of materials is passed and rays to trace increase rapidly. This can be seen from Figure 37. In surface point 1 the viewing ray splits into reflected and refracted rays. As the refracted ray reaches surface point 2 it undergoes internal reflection and refracts partly and escapes the surface. The refracted ray once again meets the surface in point 3 and undergoes reflection and refraction. Rays diverging from point 1 have less intensity than the incident ray, as is the case with points 2 and 3 as well.

Sometimes even one additional viewing ray that has to be traced increases computation significantly. Often sorts of limitations to these kinds of cases have to be done. For example, rays can be traced until their intensities are small enough to be discarded. Another option is simply to ignore the additional intersections along the way of a viewing ray.

When light reflects from a surface, it bounces away, and the resulting viewing ray can rarely be sampled within the same surface. Instead, the end point of the transformed viewing ray most likely resides on the surface of an entirely different polygon. Figure 38 demonstrates this case.

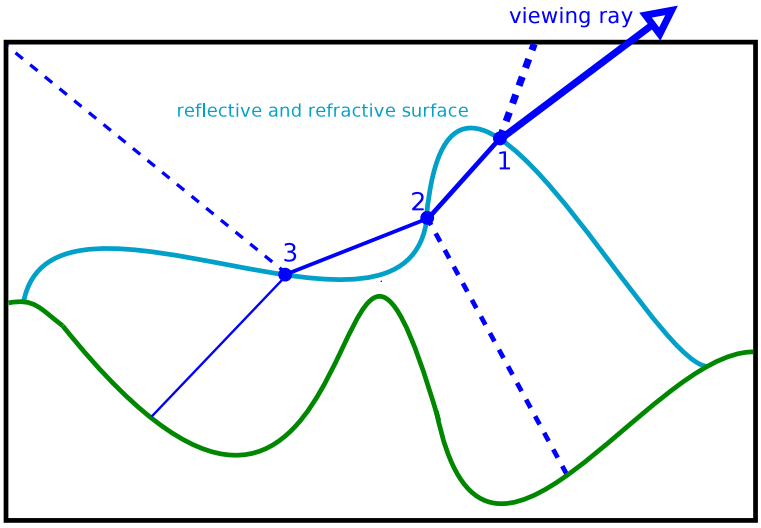

Figure 37: A sketch of how light rays split as a result of multiple intersections

Computing intersection points for a viewing ray against other polygons in a scene is computationally very expensive. In addition, if the tracing is done in fragment shader, direct access to polygon data is not available, although this problem can be avoided to some extent by encoding the polygon data into a texture. Due to the expensiveness of the computation alone, this approach is not suitable for real-time rendering with current hardware in most applications.

Reflections from the surface do not necessarily need to be precisely correct in order to give the surface a believable appearance. *Environment mapping* techniques can be used to map the environment so that it can be sampled without exhausting intersection searching with the surrounding polygon meshes. Also refracted viewing rays that escape the surface can be sampled with environment mapping techniques. Fully translucent objects, such as ones made entirely of translucent material, are highly dependent on how the environment is simulated.

3.2.3 Simulating scattered light

If light scatters from a surface or in a material in a uniform manner, it is distributed in the environment of its unaltered path, most of the light ending up near the center path. As discussed in [CW05], the distribution is expressed by the *Gaussian distribution*. Tracing several scattered light rays for a certain viewing ray is computationally too expensive. A more practical approach for

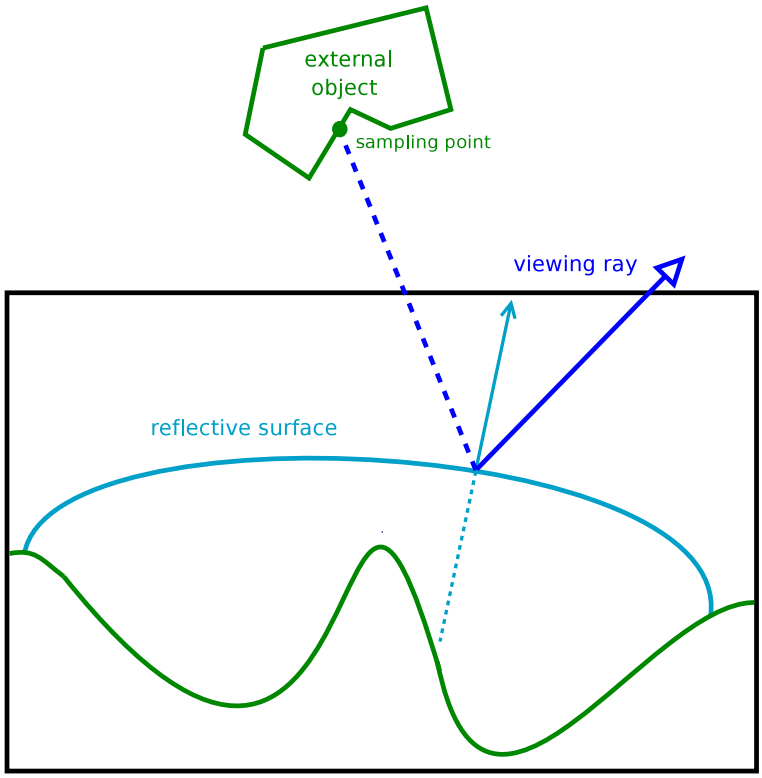

Figure 38: Reflected ray escaping the surface into environment

real-time applications is to apply *Gaussian blur* filtering to the textures that are used to sample the viewing rays. Gaussian blurring is not computationally cheap either, but the color maps can be precomputed, and then used as textures in real-time. [CW05]

Intensity of the scattering during reflection depends on the surface properties. Therefore, if the surface is uniform, the reflected environment can be blurred with a static degree of blur. However, if the scattering occurs within a uniform material, different levels of blurred environments need to be constructed, as light scatters more as it travels longer distance in the material.

Dynamic level of blurring can be implemented by interpolating the precomputed key environment maps to achieve arbitrary levels of blur between the maps. Since the blurred images represent less detail, lower resolutions suffice for the key environment maps. It is suggested in the article [CW05], that proper choices for the resolutions are

$$2^n \times 2^n, n \in \mathbb{Z}^+ \tag{20}$$

This gives resolutions 1x1, 2x2, 4x4...256x256, and 512x512, for example. In addition, the according Gaussian blur strengths have to be calculated for the resolutions. These have to be further related to medium thicknesses in order to choose proper resolutions and interpolation ratios during rendering. This aspect is more thoroughly explored in article [CW05].

A problem in approximating scattering using this approach is that the environment maps cannot be blurred in real-time efficiently. A consequence is that the technique is able to simulate only static environments. A change in the environment yields either incorrect appearance for the object when changes are not taken into account, or expensive computation when the environment maps are regenerated in real-time.

3.3 Environment mapping techniques

3.3.1 Cube maps

Cube mapping is an environment mapping technique capable of simulating the environment as a whole. A cube map consists of six textures representing the environment from a certain point, every texture representing one direction. As the name implies, a textured cube simulates the environment of the center point of the cube.

The perceived color value for every possible direction can be sampled from the cube map. Unfortunately the simulation is only precisely accurate for the center point of the cube, unless the environment is actually infinitely far away, which is rarely the case. Usually cube maps are used to represent the environment for objects of finite size, which makes the cube map a distorted approximation for points outside the vicinity of the center of the object. The distortion is proportional to the distance from the center to the simulated point, and inversely proportional to the average distance of the environment. In practice, this means that cube mapping is accurate for objects that are small in comparison to the environment.

A cube map can be generated by rendering the surrounding scene using six different perspectives: forward, backwards, left, right, up and down. In vector form these equal to viewing direction vectors of $(1,0,0), (-1,0,0), (0,1,0), (0,-1,0), (0,0,1),$ and $(0,0,-1)$ from the desired point which becomes the center of the cube. A problem with this technique is its obviously high computational cost. Without alterations, the technique has to render the scene six times for every cube map used in the scene. Figure 39 demonstrates how the six textures map into the environment of an object.

When cube maps are used for refracted or reflected appearance, they do not need to be as accurate or as detailed as the rest of the scene in order to deliver the same level of realism. In addition to the fact, that refractions and reflections from an uneven surface are far less distinguishable than the rest of the scene, they are usually significantly smaller in size than the entire scene. Usually it is appropriate to trade visual quality of the cube map for performance, as rendering the scene six times over for every cube map in each frame is usually unacceptable.

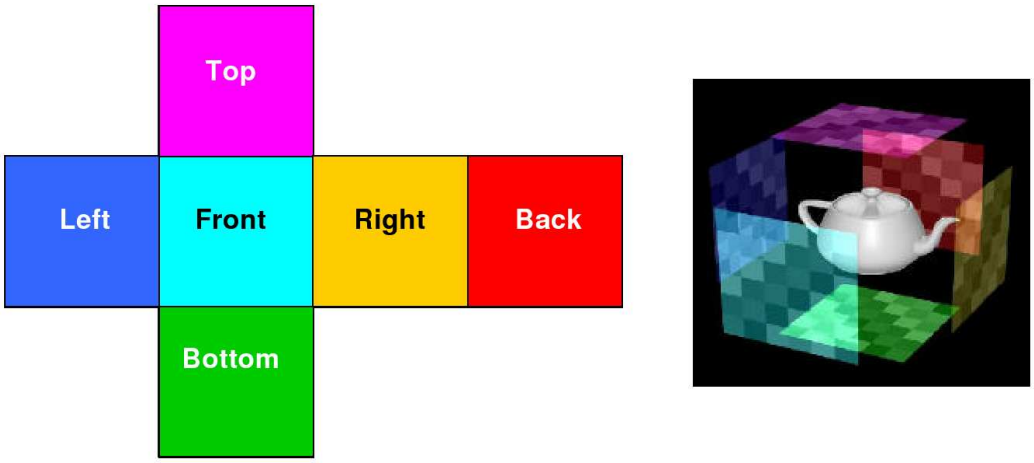

Figure 39: Mapping of the environment into six textures of a cube map [Kil99]

In order to decrease rendering time, cube maps can be rendered with a significantly smaller resolution than the scene. In addition, cube maps do not necessarily need to be rerendered each frame if the environment remains relatively static. It should be noted, that in the cube map's point of view movement of the object equals to the movement of the environment, and hence calls for an update of the cube map.

A single cube map can be applied for several different objects if they are close enough to each other compared to the environment. Also, the cube maps can be rendered with lesser details and simpler rendering routines. For example, relief mapping, self-shadowing, and even normal mapping can be often left out when rendering cube maps. Another introduction to cube mapping can be found from [Kil99].

3.3.2 Perspective dependent maps

If an environment needs to be mapped for a surface that is planar and represents only small-scale variations and unevenness, such as ripples in water, the environment can be simulated with a direct projection. Such a simulation is done by rendering the environment unaltered as seen by the viewing point into a texture, which can be then used for refractions. Such an environment map is usually referred to as a *refraction map*. *A reflection map* can be rendered similarly using a reflected viewing point. Figure 40 shows perspective dependent reflection and refraction maps

generated for a water surface. An implementation with this type of environment mapping can be seen in [VIO02].

Reflection and refraction environment maps are mapped onto a planar surface by adjusting sampling coordinates according to alterations in viewing ray as it undergoes reflection or refraction. The problem with this mapping technique is that it does not represent the entire environment, but only a portion visible to the rendering perspective. This limits the usage to surfaces that have only small perturbations, as highly perturbed viewing rays can escape the simulated environment or land on occluded parts of the environment. Moreover, surfaces that are not planar cannot be simulated with this technique, as the unaltered environment maps are flat.

In addition, distance from the reflective or refractive layer to the environment has to be known along with the perturbed viewing ray in order to calculate precise sampling points. This information is not in general available and thus the dependence of the refracted medium depth and the intensity of sampling coordinate perturbations are not visible, which constitutes to the inaccuracy of the simulation. This can be seen, for example, in the boundaries of simulated water in Figure 40 where the distortion of the refraction map should diminish as water depth reaches zero, but it does not.

Aside to these flaws, perspective dependent environment maps have some advantages over cube maps. They can be used for surfaces that are large relative to their environment without any inaccuracy caused by the size. If the environment maps are dynamically updated, the environment has to be rendered only once as opposed to the six times required by cube mapping. It should be noted, that perspective dependent environment maps need to be updated if the viewing point or the viewing direction changes significantly, which is not the case with perspective independent environment mapping techniques, such as cube mapping.

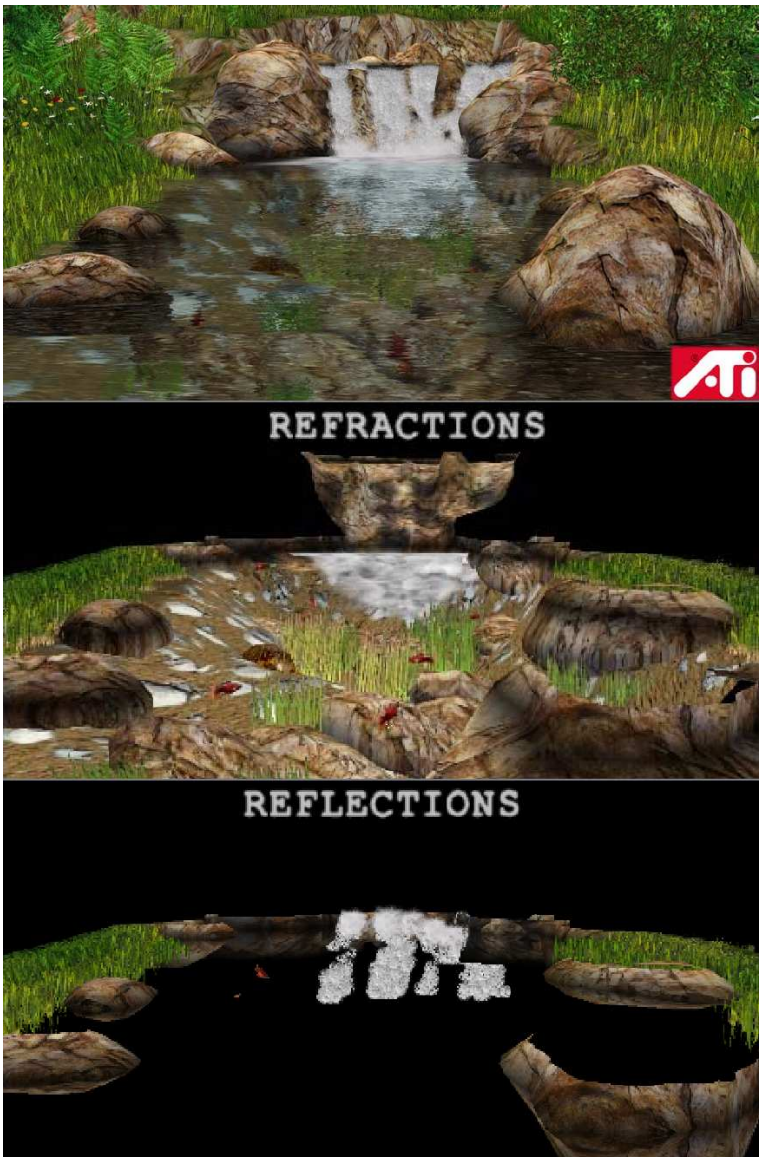

Figure 40: A scene with a planar rippling water layer and according refraction and reflection maps [VIO02]

3.4 Summary

In this chapter, we have discussed the way light reflects and refracts, and a way to apply this information into viewing rays making it possible to render optical materials in a physically correct fashion. Intensity calculations have also been covered, including Fresnel terms and chromatic dispersion.

An equation to calculate light absorption was given. Light scattering can be implemented by Gaussian blurring the textures or environment maps used to sample the scattered light ray. The appropriate blurring degree is determined in [CW05]. Dynamic level of blurring can be achieved by interpolating pregenerated key maps. Due to computational costs the Gaussian blur cannot be applied in real-time.

Two different environment mapping techniques were explored. Cube mapping is a suitable mapping technique when the environment is relatively distant to the object. Regenerating cube map requires a construction of 6 texture maps, and as such is rather time consuming for environments that change or for objects that move considerably relative to their environment.

Perspective dependent environment maps are suitable for planar surfaces and situations where physical correctness is not required. In addition to these limitations, any change in the viewing point or direction needs to be compensated by regenerating the view dependent environment map.

4 TRANSLUCENT FEATURES ON VOLUMETRIC SURFACES

The following chapter discusses a novel implementation for visual simulation of geometrically complex surfaces including translucent features. Implementational considerations are given along with an example implementation.

Simulating two-layer surfaces using relief mapping techniques are discussed and demonstrated. The upper layer is defined as being translucent and having a larger index of refraction than the surrounding environment. Refraction of light is then applied on the upper layer. Absorption of light is also simulated within the translucent layer.

Reflections are discussed and demonstrated along with different sampling techniques. Refraction and reflection intensities according to the Fresnel terms are analysed and demonstrated, and reflection and refraction samples are composed according to the intensities. *Specular reflections* are simulated using an adjustment to suit the reflection model.

Shadows that the translucent layer casts differ from the ones cast by solid features of the surface. These shadows – parts of the surface occluded by the translucent layer – are discussed and implemented. Hued occlusions are also covered. In the end of the chapter, performance of the implementation is analysed in detail. The applicability of the proposed rendering technique is compared against other similar techniques, as well.

4.1 Surface sampling

4.1.1 Relief mapping two layers

With the use of relief mapping and knowledge of how light behaves in materials with varying optical densities, it is possible to efficiently simulate double-layer surfaces that consist of a translucent upper layer and a solid lower layer with optically denser material in between. Such simulations are ideal for surfaces with full or partial glass coatings, surfaces with drops of water or some other liquid on them or even surfaces with puddles and ponds of water, for example.

The layers can be represented with a normal and a height map for each surface. If the layers are close to each other through the entire surface, they can be adequately simulated with regular height maps. Different bounding boxes are needed for the layers if either surface is significantly deeper or elevated compared to the other. If distance between the layers occasionally gets large, as might be the case when simulating ponds for example, more accurate height maps need to be used. *High dynamic range* (HDR) textures with 16-bit floating-point samples can be used in such cases for added precision.

An additional texture unit needs to be used for the normal map of the translucent layer. Height map for the translucent layer can be supplied in the alpha channel of the normal map, or in a separate texture unit with the height map for the solid layer. If HDR textures are used, the latter case is the only reasonable choice.

Just as with regular relief mapping, the first intersection point along the viewing ray has to be searched. As the first intersection point might reside on either of the layers, both height maps are sampled until it is known for sure which layer intersects the viewing ray first. If the layers are so close together at a certain fragment that they cannot be distinguished from one another using the search, it should be assumed that the solid layer comes first, as this will decrease the computation time for the fragment. It should be noted that fewer samples need to be fetched in the beginning of the search if both height maps are in the same texture unit.

Figure 43 demonstrates relief mapping on two layers. Upper layer is rendered without texturing

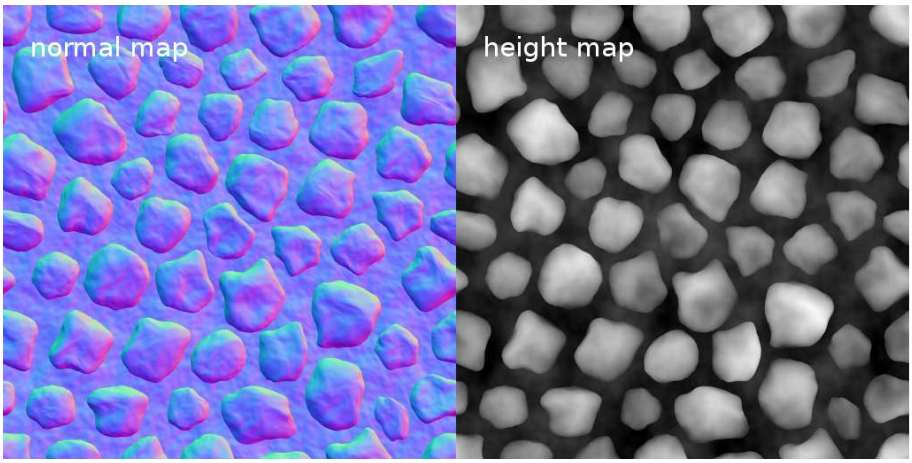

Figure 41: A normal map (left) and a height map (right) for the upper layer of the surface [Clo06]

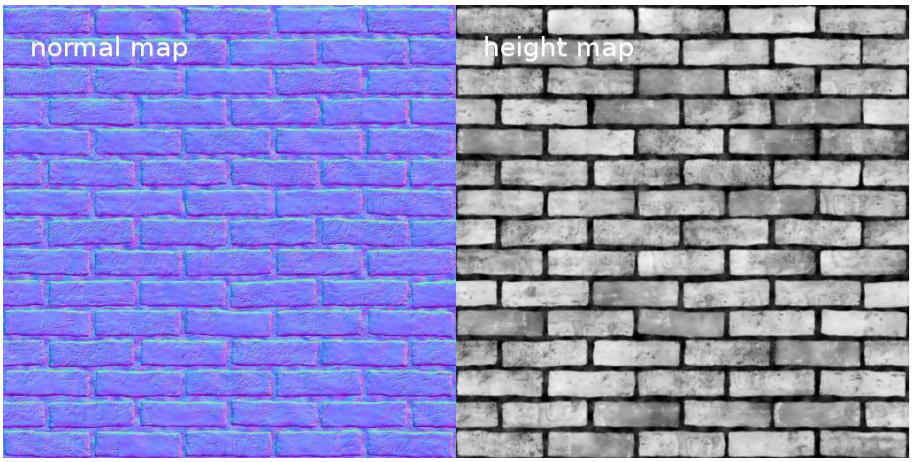

Figure 42: A normal map (left) and a height map (right) for the lower layer of the surface [Clo06]

in red using normal and height maps seen in Figure 41. The lower layer is rendered in blue using normal and height maps shown in Figure 42. The intersection is searched using 8 steps of linear search followed by 6 iterations of binary search and linearly approximating the intersection point between the two last steps. The lower layer is preferred in the case of layer levels being closer together than the resolution of the search. Bounding box limits for the upper layer are from -0.5 to 0.5, and for the lower layer from -0.25 to -0.05.

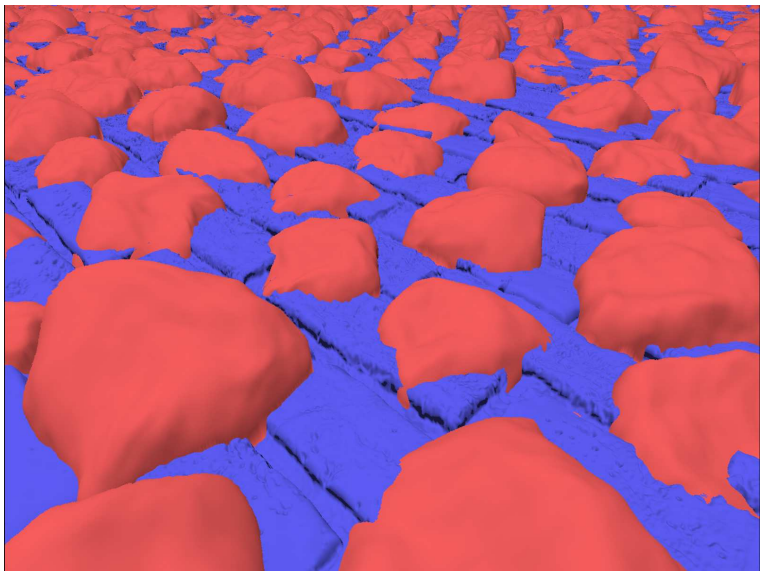

Figure 43: Using relief mapping on a two-layer surface. The image is produced using bitmaps shown in Figures 41 and 42

4.1.2 Refraction

When light hits the boundary of materials with differing optical density, which happens at the intersection point of the upper translucent layer, it changes direction according to equation (18). Normal vector for the translucent layer can be sampled from the related normal map at the position of the intersection point. Optical density of the material between the layers can be externally supplied for the fragment program. After the viewing ray has been transformed according to laws of refraction, it can be traced to the perceived intersection point.

If most refracted rays hit the solid layer without intersecting the translucent layer again, the intersection point can be searched against the solid layer alone. Tracing the reflected ray directly to the solid layer also avoids the problem of increasing viewing rays that have to be traced due to several reflections and refractions, which increases the computation time significantly. Ignoring the rest intersection points is not physically correct, but its impact on quality is small compared to the gain in performance.

Figure 44 shows the surface from Figure 43 with applied refraction. The optical density between the upper and the lower layer is set at 1.62, which is the average optical density of glass. The

Figure 44: Simulated refraction for a layer of glass on a surface – portions of the surface affected by refraction are indicated by a static red hue. The image is produced using bitmaps shown in Figures 17 and 42

surrounding environment is air with optical density of 1.00. Refracted viewing rays are traced to the solid layer using a 6-step linear search followed by 5 iterations of binary search. Linear approximation for the intersection point is used as well.

If the solid layer can be simplified to being nearly planar, intersection with a plane representing the surface can be calculated to avoid the time consuming search. This acceleration technique is discussed in detail later along with reflections, as the inherent inaccuracy of the technique is not as visible in reflections.

The simulated translucent material can also have a hue. Hue results from absorption of different wavelengths of light, i.e. colors. As light emitted from the surface travels in the absorbing medium, it gets dimmer as the path travelled grows longer according to the intensity factor from equation (13). The path travelled can be derived by taking the length of the refracted viewing ray between the layers. To simulate absorption of different colors, the intensity factor can be multiplied with a color vector representing the amount of absorption per color. Colors that have a value of 0 do not lose their intensity through the multiplication.

Figure 45 represents optical medium with blue hue, i.e. absorption of red and green, with two

Figure 45: Absorption of red and green along the path of refracted rays, making the glass blue hued – the image on the right absorbs light more. The images are produced using bitmaps shown in Figures 17 and 42

levels of absorption intensity. The image on the right is rendered with d_0 from equation (13) half than d_0 for the image on the left, making it absorb red and green colors more. Color absorption also affects rays passing through the glass medium, making shadows hued. This is simulated later.

4.1.3 Reflection

In addition to refraction, portion of light reflects from the boundary of mediums with different indices of refraction. The reflected ray is given by equation (14). Unlike with refraction, majority of the reflected rays escape the surface and end up somewhere in the environment. Because of this, environment mapping techniques need to be applied to sample the reflecting rays properly.

If the surface is partially covered with translucent material, such as the example surface, cube mapping is a viable technique for sampling the environment. Some of the reflected viewing rays, however, do not escape the surface but end up near the vicinity of the beginning of the reflected viewing ray vector. Due to the cube mapping's inherent inability to simulate parts of the environment close to the simulated feature, local reflections appear random and unrealistic.

Local reflections can be satisfied by sampling the surface directly, as demonstrated by Figure 46. For this, the intersection point of the reflected viewing ray against the surface needs to be known. Relief mapping techniques can be used to search the precise intersection point, but reflections are

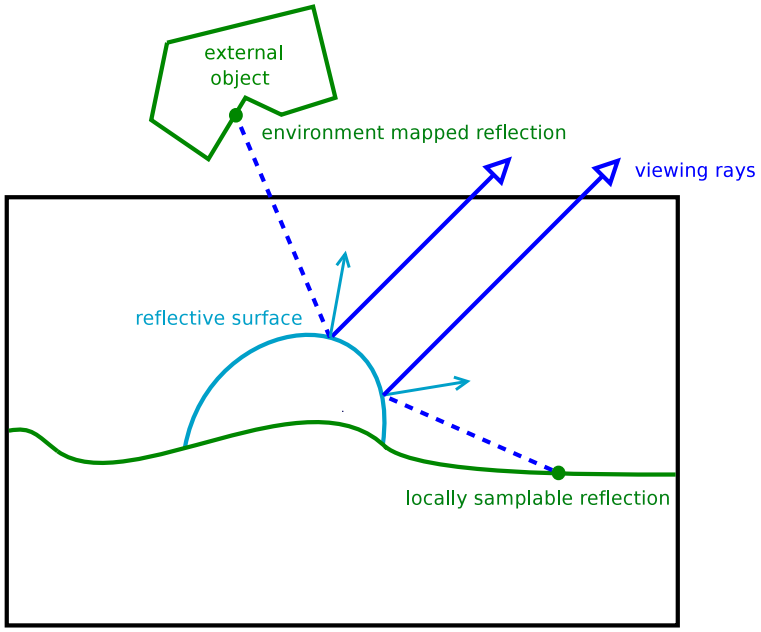

Figure 46: Local and environment reflections

not usually distinguishable enough to justify the computational intensity of the search. In addition, portion of the reflected viewing rays tend to intersect the surface in very steep angles, which leads to serious artefacts.

The solid layer of the surface can be assumed planar, and the intersection point can be calculated against the surface plane. All points $\mathbf{p}$, that satisfy equation

$$(\mathbf{p} - \mathbf{s}) \cdot \mathbf{n} = 0 \tag{21}$$

make up a plane in a 3D space. The equation should be familiar from basic vector mathematics. In the above equation $\mathbf{s}$ is a point somewhere on the plane and $\mathbf{n}$ is the plane's normal vector.

Points $\mathbf{p}$ along the reflected viewing ray are given by equation

$$\mathbf{p} = \mathbf{i} + t\mathbf{v} \tag{22}$$

where $\mathbf{v}$ is the reflected viewing ray, $\mathbf{i}$ is the intersection point on the translucent layer where the viewing ray begins and t is a scalar greater than or equal to 0. When $\mathbf{p}$ from equation (22) is

placed in equation (21), t can be solved as

$$t = \frac{(\mathbf{s} - \mathbf{i}) \cdot \mathbf{n}}{\mathbf{v} \cdot \mathbf{n}} \tag{23}$$

If the plane equation is to be applied for surfaces expressed in tangent space, then

$$\begin{cases} \mathbf{n}_x = \mathbf{n}_y = 0 \\ \mathbf{n}_z = 1 \\ \mathbf{s}_x = \mathbf{i}_x \\ \mathbf{s}_y = \mathbf{i}_y \\ \mathbf{s}_z = h \end{cases} \tag{24}$$

where h is the static height of the solid layer. Note that $\mathbf{s}$ can be chosen as any point on the surface. With these choices the equation (23) can be simplified to

$$t = \frac{h - \mathbf{i}_z}{\mathbf{v}_z} \tag{25}$$

By placing the known t into equation (22) the precise intersection point can be solved, and furthermore the proper sampling point as $(\mathbf{p}_x, \mathbf{p}_y)$. Before sampling, it should be checked that the sampling point lands within the surface, i.e. sampling point is bound by the texturing coordinates of the polygon. If it does not, the ray does not intersect the surface, and it has to be sampled using environment mapping techniques such as cube mapping. It should be noted that $\mathbf{v}_z < 0$ is a preliminary condition for the intersection point to exist, which can be checked before calculating the sampling point to reduce computing time for rays escaping the surface.

Figure 48 shows portions of the translucent layer that can be sampled locally in red and portions needing environment mapping in blue. Figure 49 demonstrates the rendering of reflections on the surface. In the left reflections are rendered using environment mapping alone. The environment simulates a cloudy sky, with light blue and white shades, as shown in Figure 47. In the right, the suggested technique for sampling local reflections is used.

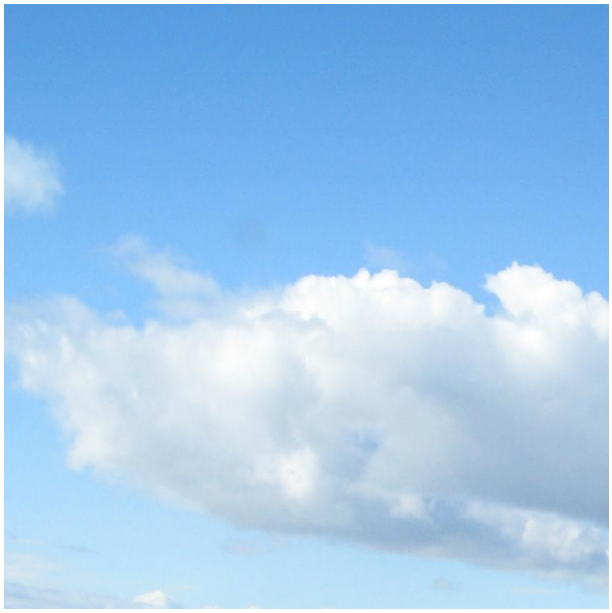

Figure 47: A texture representing a cloudy sky is used for environment mapping [Par05]

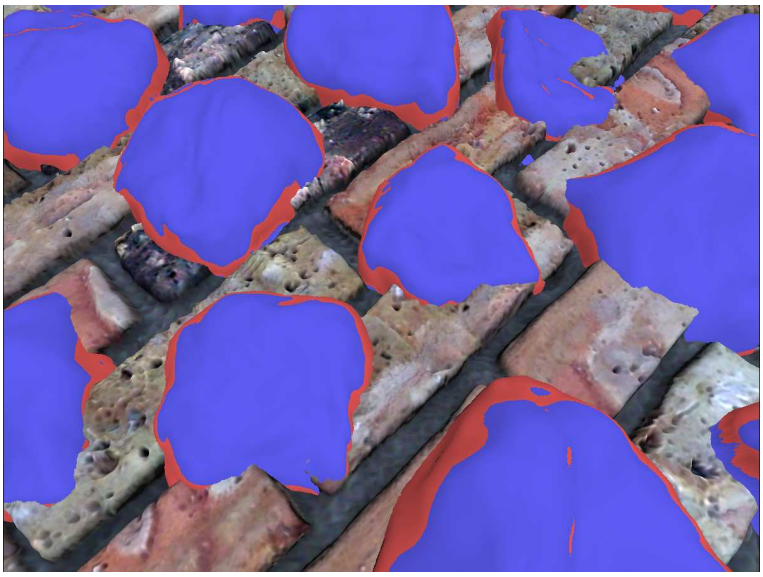

Figure 48: Portions of surface reflections that can be sampled locally (red) and portions that need to be sampled using environment mapping (blue). The image is produced using bitmaps shown in Figures 17 and 42

Figure 49: Using local sampling for reflections. The image is produced using bitmaps shown in Figures 17, 42 and 47

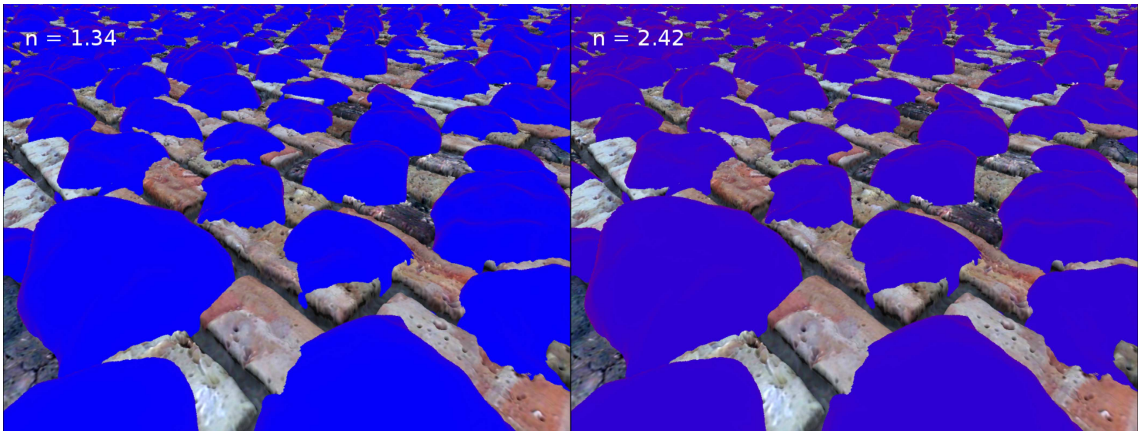

Figure 50: Intensities of refraction (blue) and reflection (red) on water (left) and diamond (right) surfaces. The images are produced using bitmaps shown in Figures 17 and 42

4.2 Applying the samples

4.2.1 Composition

The final appearance of a fragment is determined by both reflected and refracted rays. The intensities according to which the sampled rays should be blended are given by equations (11) and (12). Figure 50 demonstrates the intensities of the rays on different parts of the translucent layer of the surface. Refraction intensity is indicated with a blue color, and reflection with a red. The greater is the portion of color, the greater the intensity. Two different mediums are simulated: water and diamond, with indices of refraction of 1.34 and 2.42, respectively.

Reflections are stronger in parts of the surface, where the viewing ray intersects the surface in a steep angle. Reflections are also stronger in a medium with larger index of refraction. Refractions however dominate within real life mediums that have indices of refraction less than 2.5. As the steepness of the viewing ray also plays an important role, reflections become more domineering for example in large water areas when observed from a low altitude.

Composed appearance of reflection and refraction can be seen in Figure 51 for mediums from Figure 50. The medium also absorbs light, which affects refraction samples.

Non-scattering surfaces, like the surfaces discussed previously, also reflect light sources. The usual

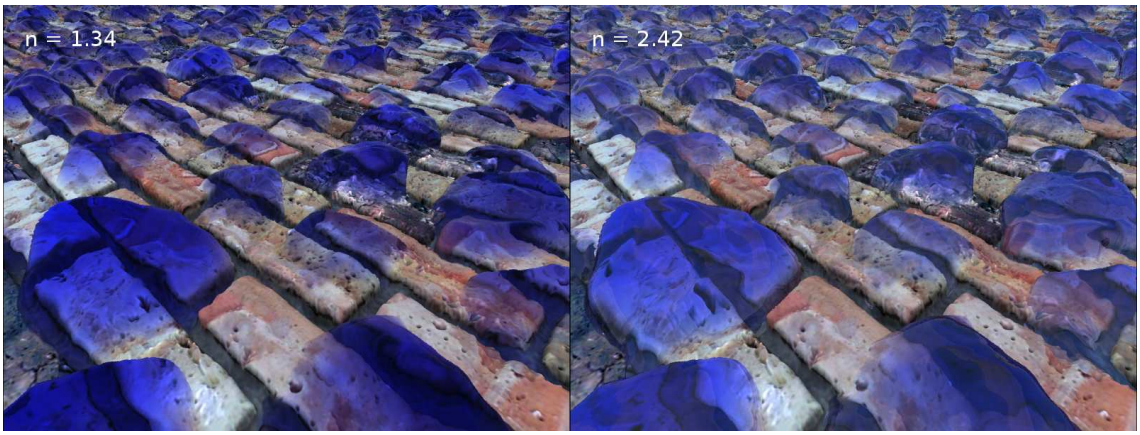

Figure 51: Composed reflections and refractions for water (left) and diamond (right) surfaces. The images are produced using bitmaps shown in Figures 17, 42 and 47

method for computing light reflections, using *Phong shading* [Bli77], produces light intensity in the range from 0.0 to 1.0. This is appropriate for solid surfaces since it represents the final value on the screen, which will be clamped to this range in any case.

Unfortunately, light reflections for the transparent layer cannot be rendered the same way as specular reflections on ordinary surfaces. Light sources usually correspond to intensities greater than 1.0 on the color scale, and even after partial reflection, the intensity can stay well above 1.0. However, if the reflection including light sources with maximum intensity of 1.0 is multiplied by the reflection intensity, reflections from light sources appear dim with an intensity below 1.0. Light source reflections could be applied to surfaces after the environment reflections, but it would not be physically correct as light source reflections depend on the reflection intensity the same way as environment reflections.

A solution to overcome this problem is to multiply the intensity of specular reflections by a factor representing the relative strength of the light source. Figure 52 shows specular reflections alone, rendered using unaltered Phong shading on the translucent layer. These reflections are then rendered with a proper reflection intensity along the environment reflections in the image on the left in Figure 53. In the image on the right, specular reflections are multiplied by 10 before applying them on the reflections, which results in more realistic light source reflections. The simulated medium is glass with an index of refraction of 1.62.

Figure 52: Specular reflections on the translucent layer. The image is produced using bitmaps shown in Figures 17 and 42

Figure 53: Unaltered specular reflections (left) and intensified specular reflections (right) after applying reflection intensity for the glass surface. The images are produced using bitmaps shown in Figures 17, 42 and 47

Another solution for this is to apply *high dynamic range rendering* (HDRR) for lighting throughout the entire rendering process. In this lighting model light sources are given higher color values from the beginning, and lighting calculations are done in a higher range exceeding the traditional limit of 1.0. The result is precise lighting without the need to resort to compensations or approximate multiplications of light sources.

4.2.2 Occlusion

Translucent material affects light that passes through it. Therefore, the material can be thought to cast shadows as the solid layer does. Areas being occluded by the translucent layer usually have non-uniform shadows unlike parts being occluded by the solid layer.

When light passes through the boundary of materials with differing indices of refraction, portion of the light gets reflected instead of maintaining full intensity. This decrease in light intensity can be calculated with equation (12). Also, if the translucent material has a hue, i.e. it absorbs light, light intensity is further decreased relative to the length travelled in the absorbing medium.

Figure 54 shows a progression of a light ray passing through a translucent layer. A portion of the light intensity reflects from point 1, allowing a decreased intensity to pass through the first intersection point. Between points 1 and 2 light intensity is further decreased if the material absorbs light. In point 2, light intensity is again decreased because of the internal reflection.

The method is suitable for calculating light for fragments that are under the translucent layer as well. In such a case no internal reflection usually occurs, and the alight fragment is used as a surface sample for the refracted viewing ray. In general this lighting model should be used for every sample of the solid layer. However, if no relief mapping technique is used for the local sampling of reflections, it might be appropriate not to use this lighting method for the reflected surface samples as an optimization.

If the solid layer intersects the light ray for a fragment, no light can reach the surface at all. The effect is equivalent to producing hard self-shadows for the solid layer. Unless the solid layer is flat enough for no self-occlusion to occur, the light ray intersections with the solid layer should be

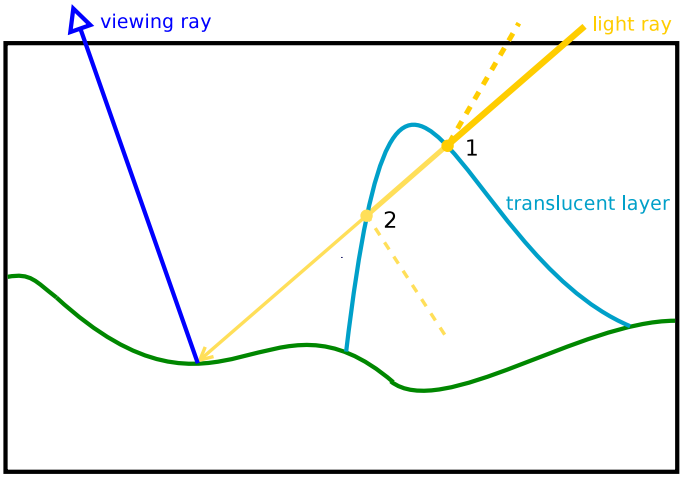

Figure 54: Light intensity decreases as it passes through a translucent layer

checked for, as it requires only little extra computation for the search.

One method to implement the occlusion calculation is to march through the light ray and multiply the incident light ray with a per-color attenuation vector for each surface feature affecting light propagation. As the order in which the vectors are multiplied does not affect the result, the light ray can be searched in either direction.

It is practical to start the search from the fragment with a linear search and use binary searching for refining the encountered intersection points. If the translucent material absorbs light, points where the ray enters the material should be stored in order to calculate the length travelled within the material when the ray escapes the material. If the ray intersects the solid layer, no light can pass through and the search can be ended immediately. On the intersection points with the translucent layer, intensity calculations are done according to the surface normal gotten from the normal map on the refined intersection point.

Figure 55 demonstrates a two-layer surface with and without shadows. Borders of the shadows occluded by the translucent layer are more intense than the middle portions of the shadows. This effect is correct as more light is reflected where the incident ray comes in a steep angle into the surface layer, and less light gets through. In points, where full internal reflection occurs according to (7), no light can get through.

Figure 55: A two-layer surface with and without occlusion of light. The images are produced using bitmaps shown in Figures 17, 42 and 47

Figure 56: A two-layer blue hued surface with and without occlusion of light. The images are produced using bitmaps shown in Figures 17, 42 and 47

Figure 56 demonstrates a surface shown in Figure 55 with a material that absorbs red and green light, making the medium blue hued. The solid layer under the translucent layer appears to be more intensely hued on the image with applied occlusions. If light absorption is not taken into account, incident light does not lose its intensity and arrives at the solid layer with full intensity and without a hue. This is incorrect as light should lose part of its intensity on its way to the solid layer, and further during its path along the viewing ray from the solid layer. Image on the right simulates a more realistic lighting.

Unfortunately, the discussed lighting model is not physically precise. It can only simulate point lights, which are either infinitely small or infinitely far away, and hence no soft shadows can

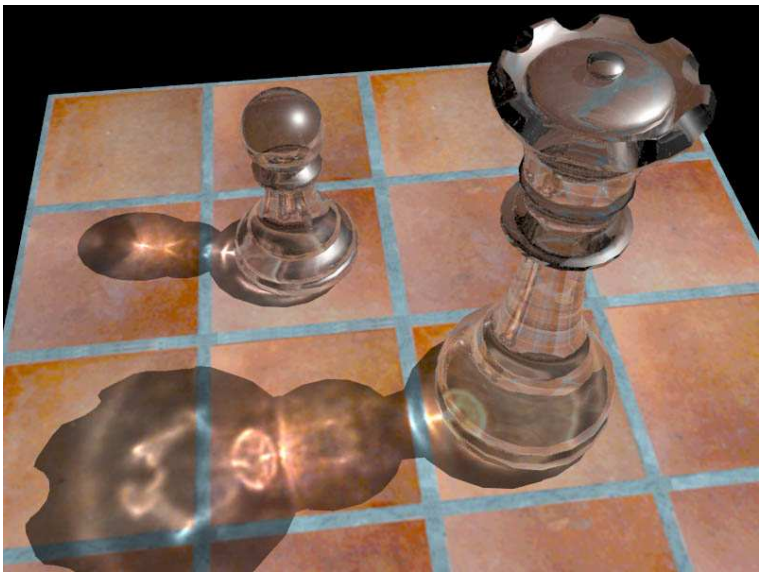

Figure 57: Non-real-time ray traced translucent objects with correct lighting [Oki06]

be produced. When light undergoes refraction, not only does its intensity change, but also its direction. A fragment program's purpose is to determine a value for a certain fragment, and therefore several different light rays would have to be traced in order to determine the amount of light received for the fragment.

Tracing the light ray directly would only determine whether this light ray from infinite number of possibilities arrives at this fragment, and such information is useless on its own. One possibility is to trace a number of light rays from the fragment and calculate a proportion of light arriving at the fragment. The more light rays are traced, the more accurate is the simulation. While such a simulation might be usable for prerendered graphics in ray tracing applications, it is far too exhaustive to be used for real-time applications, as even one light ray search is rather exhaustive.

Fortunately, light intensity is calculated physically correct, so the only practical problem lies in the light distribution. Figure 57 demonstrates a ray traced image consisting of translucent objects. Light intensity is not distributed evenly in the shadows, but instead lighter parts appear where more light rays converge. Usually this does not affect the average light intensity for a shadow. It should be noted, that even though most of the light from the middle of the shadows is focused on relatively small areas, the edges of the shadows are darker, as can also be seen from Figure 55.

4.3 Performance

4.3.1 Environment

With real-time graphics, performance is always an important issue. In this section a detailed performance analysis of the previously discussed implementation is given. Even though performance of different implementations may vary significantly, it is possible to get an idea what sort of a performance impact comes with the different techniques.

The shader program is written in OpenGL Shading Language (GLSL), and it is listed in Appendix A. NVIDIA's GeForce 7600GS (G73 architecture) graphics processing unit is used for the execution of the shader at a resolution of 800x600. The application environment is a GNU/Linux operating system with the NVIDIA's OpenGL 2.0 implementation using Linux x86 driver version 1.0-8774.

4.3.2 Results

The rendering techniques have been taken into use consecutively on top of each other and then benchmarked. The first technique benchmarked is a simple normal- and color-mapped surface seen on the left in Figure 18, and the final technique can be seen on the right in Figure 56.

In Table 1, the time required to render an entire frame is shown in milliseconds for each technique including the previous techniques applied so far. In addition, the corresponding frames per second ratio is shown. To illustrate the performance of each technique independently, difference in the rendering time before and after applying the rendering technique is shown.

Table 1: Performance measurements

Description	time [ms]	Δtime [ms]	fps
Normal and color mapping	4.37		229

Relief mapping		**26.0**	
24-step linear search	22.8	18.4	44.0
6-step binary search	29.1	6.35	34.4
Linear approximation	30.3	1.23	33.0
Two-layer relief mapping	49.7	**19.3**	20.1
Reflection/refraction calculations	50.4	**0.70**	19.9
Refraction search		**16.1**	
8-step linear search	59.6	9.23	17.8
5-step binary search	64.3	4.71	15.6
Linear approximation	66.5	2.20	15.0
Reflections		**15.6**	
Environment mapping	68.6	2.07	14.6
Local sampling	75.2	6.59	13.3
Local sample lighting	72.7	-2.47	13.8
Corrected specular lighting	82.1	9.45	12.2
Occlusion search		**78.2**	
16-step linear search	114	31.9	8.77
5-step binary search	160	46.3	6.24
Occlusion intensity	226	**65.2**	4.43
Hued light absorption		**38.7**	
Refraction	261	35.0	3.84
Occluded light	264	3.70	3.79

4.3.3 Analysis

The results are highly dependent on the shader implementation and the GPU used. For example, implementing lighting for local samples decreases the rendering time even though more GLSL code is compiled. This might have something to do with NVIDIA's GLSL compiler, or GPU's ability to schedule operations in a more efficient fashion.

The results also show that light absorption implemented for refractions take almost ten-fold time compared to absorption implemented for occluded light. Experiments showed that implementing either of the technique increased computation time significantly, and adding the other increased computation time only slightly, hence the techniques can be considered equally time consuming. Evaluating the techniques independently from the results may lead to erroneous conclusions.

Searches for occluded light seem to increase computation time considerably. This is mostly because linear search cannot be interrupted if an intersection point is found – unless it is with the solid layer – and the search iteration count grows to the maximum. Binary search is used for every intersection point instead of the first, and can be ran more than once, especially in the areas occluded by the translucent layer. Other than that, the searches were implemented the same way as with relief mapping that shows a significantly lower computation time increase.

In summary, the iteration count for the linear search should be minimized to a point where shapes of surface features do not suffer noticeably from intersection point skipping. Linear approximation is always advisable on two-layered surfaces due to its static, and hence relatively small, time complexity. The iteration count for the binary search should be selected, after the linear search, so that the surface appears smooth.

The same rules apply to the searches used to trace refractions, but the searches generally require less iterations to achieve the same level of visual quality. The local sampling of reflections as a technique is rather cheap for the delivered realism, and should be left unemployed only if all reflections end up in the environment.

Simulating the occlusion of light involves computationally exhaustive searches, and should be avoided whenever the maximum performance is pursued. Applying a hue to materials is computa-

tionally cheap, when the light occlusion is omitted; when the light occlusion is applied, the extra cost is small relative to the entire rendering process. In general, a hue should be applied whenever the realism of the surface so requires.

4.4 Summary and discussion

In this chapter, an effective method for rendering two-layered surfaces with the accuracy of regular relief mapping was presented. Refraction and reflection were simulated as if the upper layer was translucent material with indices of refraction larger than the environment's. A comprehensive model to simulate light absorbing material was introduced. Relative refraction and reflection intensities were demonstrated with the conclusion that refractions dominate on materials with realistic indices of refraction.

A method for sampling reflections locally from the surface was presented. The method is especially suitable for sampling reflections on surface features in conjunction with a cube-mapped environment, and produces accurate reflections of the surface.

Specular reflections were intensified in order to produce visually realistic light source reflections. Hard self-shadowing technique proposed in [POC05] was extended to suit the simulation of light occlusion. The occlusion method produces visually plausible shadows for both solid and translucent features of a surface, including a possible hue resulting from a light absorbing translucent medium. A detailed performance analysis was given at the end of the chapter for the example implementation, including rendering time costs for individual techniques.

The rendering method is especially suitable for rendering small-scale details on surfaces that are fully or partially covered by a layer of optically denser material. Such surfaces include surfaces covered with raindrops, or other liquid, including colored liquids and solid materials, such as glass or diamond. An accurate simulation of sweaty or bloody skin is possible as well.

The performance measurements indicated that the technique is capable of achieving real-time framerates on commodity graphics hardware, when light occlusion is not applied. Light occlusion, in the proposed form, is computationally costly, but can be simulated in real-time on high-end hardware. The performance measurements were done for searches that are more than adequate for most surfaces, thus lightening the searches will likely improve performance without affecting the quality noticeably. The method does not rely on precomputation, the environment mapping techniques excluded. Hence it is suitable for simulating dynamic or animated surfaces.

Figure 58: A color map used to represent ground [Clo06]

The translucency rendering technique for polygonal objects proposed in [CW05] is dependent on precomputed geometry information, in addition to being highly dependent on the environment mapping techniques, and does not suit for the simulation of animated objects. The rendering model simulates light scattering in a form directly applicable to the technique represented in this thesis.

The translucency rendering technique for ice, as presented in [SN06], resorts to simplifications suitable for icy materials, but not for optical materials in general. In any case, both of the techniques [CW05] [SN06], as techniques for polygonal objects in general, are meant for the rendering of complete objects, as the technique proposed here is better suited for simulating details within a surface on a polygon.

Larger scale features, such as ponds and puddles of water on ground can be simulated using the technique, as well. Figures 60 and 61 demonstrate the rendering of a water layer without light occlusion. The simulation of water dynamics is performed in the graphics application, outside the GPU. It should be noted, that the simulation model is suitable for simulating both small and large-scale water areas. In Figure 63 a pool of water is rendered using light occlusion. The simulation produces shadows on the bottom of the pool.

When simulating large water areas, such as oceans, simulation models designed specifically for

Figure 59: A color map for a terrain made of stone [Clo06]

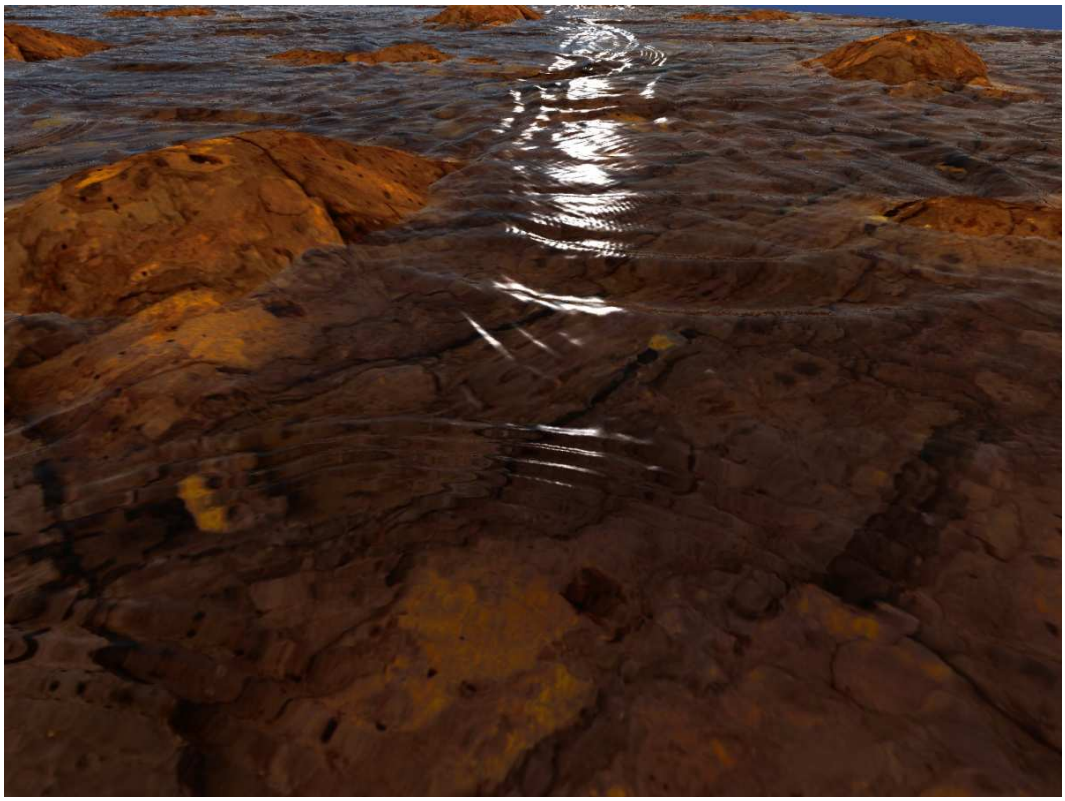

Figure 60: A simulation of a clear water layer. The image is produced using bitmaps shown in Figures 47 and 58

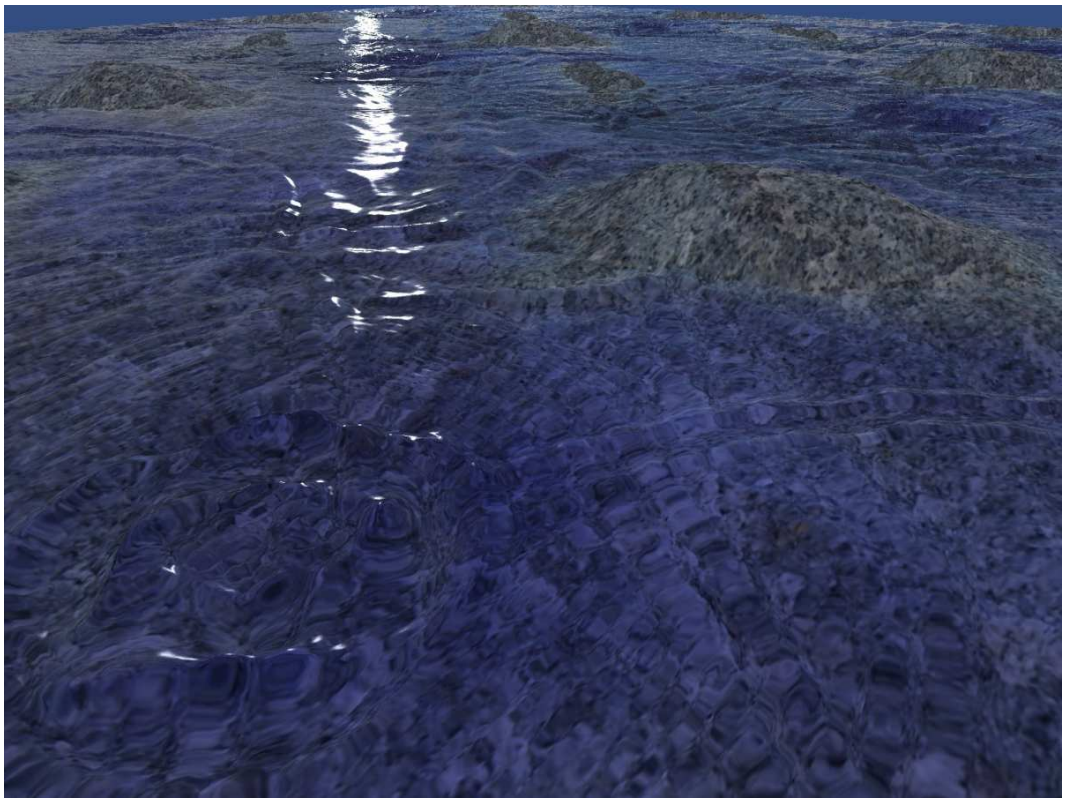

Figure 61: A simulation of a hued water layer. The image is produced using bitmaps shown in Figures 47 and 59

Figure 62: A color map for tileable blocks [Clo06]

Figure 63: A simulation of clear water on a pool with light occlusion applied. The image is produced using bitmaps shown in Figures 47 and 62

water might prove faster, and more suitable, than the technique proposed here. Techniques such as the ones presented in [PA01], [Bel03], and [YFCF06] specialize on simulating the effects that dominate on large water areas. For example, it might not be necessary to simulate refractions with costly physically correct models in order to produce realistic water simulation, as simple texture perturbations look equally convincing for deep water where the bottom of the water layer is not distinguishable anyway. In addition, the deployment of polygonal objects in the vicinity of the water is easier and more seamless for techniques that represent the water as a polygon mesh. The technique proposed in this thesis does not, as such, support refractions and reflections for polygonal objects interacting with the water, except through the environment mapping.

Thin water layers tend to need correct simulation of refraction in order to appear realistic. If refractions do not conform to the depth of the water, authenticity of the simulation suffers, and the water volume might be hard to perceive correctly. The proposed technique is suitable for rendering small-scale and thin water layers, as refractions and local reflections are simulated correctly.

During the writing of this thesis, an implementation [BD06] was published that simulates water volumes mostly the same way as the independently developed technique presented here. The water layer is represented as two relief-mapped layers, and reflections and refractions are sampled using relief mapping on a fragment-basis. A lighting model based on *photon mapping* is used instead of computing the light occlusion as presented in this thesis. For water volumes of significant depth, the light ray refractions become noticeable on the bottom of the water layer, forming shapes of converged light. This effect is called *caustics* – a phenomenon that the light occlusion model is unable to simulate. Considering the way caustics is simulated, the implementation presented in [BD06] can be considered superior to the one presented by this thesis for the simulation of water volumes.

The technique presented in this thesis is at its best in simulating small-scale transparent or translucent features on polygon surfaces. The technique allows the simulation of layered materials with user-defined index of refraction for the translucent layer.

5 SUMMARY AND DISCUSSION

5.1 Summary

Introduction to real-time computer graphics was given in the beginning of this thesis. A few techniques commonly used to increase visual complexity of surfaces representing uneven or varying height were briefly discussed. Main contributions of this thesis were discussed and analysed in respect to previous similar work, in the first chapter, as well.

In the second chapter, part of the thesis contribution was presented as an analysis of relief mapping rendering techniques. The relief mapping techniques were implemented and their applicability was analysed. Relief mapping can efficiently and scalably simulate surfaces that represent height variations, both subtle and rapid. Shadowing techniques for self-shadowing relief mapped surfaces were explored and demonstrated as well.

In the third chapter, simulation of reflections and refractions were discussed and methods for real-time implementation were suggested. Real-time simulation of light scattering and absorption during refractions was discussed. Sampling techniques to be used with reflections, such as environment mapping, were discussed as well.

Simulation models for large-scale water, such as oceans, are common in real-time graphics, but due to several simplifications and specializations they are unable to simulate small-scale details of optical materials in general. Recently, rendering techniques for simulating transparent and translucent polygonal objects have been introduced, such as the one proposed in [CW05]. These techniques are usually highly dependent on environment mapping techniques – a characteristic that limits their usage to fully transparent objects in wide environments.

The main contribution of this thesis is the presentation of a rendering technique capable of simulating small-scale translucent features on relief-mapped surfaces with great detail, as demonstrated in the fourth chapter. Relief mapping was applied on surfaces with two-layers, the lower layer representing a solid material and the upper layer representing a translucent medium. A demonstration of real-time implementation of physically realistic reflections and refractions on the translucent

layer were given. Self-occlusion of light on the surface was implemented for both the solid and the translucent layer. Even though the technique can be also used to simulate water areas, there is a recently published technique [BD06] that is more suitable for the rendering of water volumes.

The technique presented in this thesis achieves real-time framerates on today's hardware, and its accuracy is determined by the employed relief mapping methods. This means that it does not resort to approximations, except with the simulation of environment through environment mapping techniques, and it can deliver detail only limited by the level of performance that needs to be maintained.

5.2 Discussion

Increasing detail of objects in real-time graphics will most probably continue as a development of fragment shaders. Effects implemented in fragment shaders have a few significant advantages over effects implemented in vertex shaders or within the application.

If extra detail is brought to the object via an increased polygon count, no native dynamic adjustment in the level of detail occurs. Details in objects go through the same processing no matter how far from the viewing point the object resides. Objects far away from the viewing point usually require significantly less detail to appear as realistic as objects near the viewing point covering a larger portion of the screen. Effects implemented in fragment shader concentrate processing relative to their rendered area on display, providing native load balancing.

In addition, vertex processing is often needlessly heavy when used for small-scale details. For example, representing an elevated point on a surface as a vertex requires a 3-component coordinate value, usually represented with 32-bit floating-point elements, whereas a single 8-bit texture sample used by relief mapping might be as effective.

Also, if techniques similar to the presented rendering of translucency on volumetric surfaces are implemented, intersection tracing against polygon meshes – as opposed to a height map – can be extremely exhaustive, especially when the scene is large and consists of a large number of polygons. Graphics hardware manufacturers seem to agree with the trend in which computing shifts towards fragments, as more fragment processing power is introduced with modern GPUs compared to vertex processing power.

The implementation for rendering translucent features on volumetric surfaces presented in this thesis produces visually plausible results, but is computationally costly compared to effects employed in today's 3D games, for example. Shader programs are often heavily optimized, and the final optimization is done with a lower level programming language than GLSL, usually in an *assembly programming language*. Such optimizations were not done for the presented implementation. With careful optimization, performance is likely to be at least doubled, and achieving performance levels three or four times the current might be possible.

The graphics hardware on which the implementation was evaluated is far from the fastest on the market today. As the implementation represents methods likely to be seen applied in the future, the technique might be better evaluated on today's high-end equipment. Running the implementation on a NVidia's GeForce 7900GTX SLI configuration, for example, would likely yield a performance six times the performance of our reference benchmarks on NVidia's GeForce 7600GS configuration. With optimizations and ran on high-end hardware, the technique can easily achieve high enough frame rates to be employed on interactive graphics engines.

5.3 Future work

The intersection point searches, used in relief mapping techniques, can be further enhanced by applying intelligent algorithms that adjust step sizes during the search. Heuristic search algorithms can play an important role in rendering techniques that apply searches with numerous iterations. For example, height samples retrieved during the searches can be used to estimate how the surface behaves, if the nature of the surface is known in advance. Search results from the neighbouring fragments might also be of use when starting a search, as well as an approximation for the intersection point similar to the one used in offset mapping.

It is possible to extend the rendering of volumetric surfaces into surfaces representing nearly arbitrary shapes by the use of additional surface layers. The use of multiple surface layers with relief mapping was researched in [PO06] and [POC05]. The techniques discussed in the publications are applicable to translucent layers as well, and extendable into surfaces composed of more than two layers.

Methods for real-time implementation of light scattering were discussed in chapter 3, and the methods are directly applicable to the implementation in chapter 4. There are types of light scattering that cannot be simulated in this manner. For example, the observed turquoise color of water results from both light absorption, which turns light turquoise, and light scattering from suspended matter, which allows the light entering the water be seen by the observer. In addition to light scattering, matter can also absorb photons into excited atomic states, and release them in a different direction or in a photon of different energy and wavelength, i.e. color. Such properties and their effect on the observed appearance of optically complex materials should be evaluated in the context of this rendering technique.

Materials that show uncommon optical behaviour, such as soap bubbles or metals of different kind, could also be rendered using techniques discussed in this thesis, and might be worth further research. Sub-surface scattering, and similar light behaviour within volumes of translucent material that were not covered in this thesis, can likely be simulated using derivatives of the proposed technique.

Also, if an extremely accurate simulation of light scattering is required, the techniques used to travel through optical materials can be extended into simulating the scattering of individual light rays. Sub-pixel detail would have to be employed for this, perhaps through known anti-aliasing techniques, where several fragment samples are computed for a single frame-buffer pixel. These techniques would be extremely exhaustive and far from applicable in real-time graphics at the moment, but might be of use in special cases requiring extremely accurate light scattering simulations.

The lighting models presented in this thesis compute lighting in the same pass as the rest of the graphics is being rendered. When translucent volumetric features of notable depth are simulated, caustics play an essential role in the behaviour of incident light. As the amount of light received by a specific fragment may depend on several light rays, most of which do not point directly towards the fragment, caustics cannot be simulated efficiently using a traditional one pass rendering. A solution to this is to render lighting in an extra pass, prior to the actual rendering. In this pass, the surface is rendered from the perspective of the light source, as opposed to the viewing point. Instead of tracing viewing rays, light rays are traced, and the end points can be stored to a texture, which can be used afterwards as a light map in the actual rendering pass.

This way light source reflections and refractions can be correctly rendered, providing an implementation of caustics simulation. This type of lighting models are often referred to as *photon mapping* techniques. The performance measurements indicate, that relief mapping on two layers, combined with the tracing of reflections and refractions, is computationally cheaper than the exhaustive light occlusion simulation. This means that using an extra pass for light rendering, that applies the relief mapping techniques and ray transformations, probably yields a better performance than the suggested simulation for light occlusion. Research on what types of implementations are best suited for the simulation of caustics is needed.

Interaction with polygonal objects needs further research, as well. Even though external polygonal objects can be trivially clipped with the volumetric surfaces by employing Z-buffer correction presented in [POC05], another problem arises when interaction with translucent features is required. There is no trivial way to produce a correct refracted or reflected image of an object through a

translucent medium of arbitrary shape. The problem is somewhat similar to the problem of simulating an environment undistorted.

References

[BC06] F. Banterle and A. Chalmers. A fast translucency appearance model for real-time applications. In *SCCG 2006 - Spring Conference on Computer Graphics*. ACM SIGGRAPH, April 2006.

[BD06] L. Baboud and X. Décoret. Realistic water volumes in real-time. In *Eurographics Workshop on Natural Phenomena*. Eurographics, 2006.

[Bel03] V. Belyalev. Real-time simulation of water surface. In *GraphiCon-2003, Conference Proceedings*, pages 131–138. MAX Press, 2003.

[Bli77] J. Blinn. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques*, pages 192–198, New York, NY, USA, 1977. ACM Press.

[BW99] M. Born and E. Wolf. *Principles of Optics: Electromagnetic Theory of Propagation, Interference and Diffraction of Light*. Cambridge University press, 1999.

[CCC87] R. Cook, L. Carpenter, and E. Catmull. The reyes image rendering architecture. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 95–102, New York, NY, USA, 1987. ACM Press.

[Clo06] B. Cloward. Texture resources. Web-page, referenced 12.10.2006, 2006. `http://www.bencloward.com/resources_textures.shtml`.

[Coo84] R. Cook. Shade trees. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, pages 223–231, New York, NY, USA, 1984. ACM Press.

[CW05] B. Chan and W. Wang. Geocube – gpu accelerated real-time rendering of transparency and translucency. *Visual Computer*, 21(8-10):579–590, 2005.

[Dan80] P. Danielsson. Euclidean distance mapping. *Computer Graphics and Image Processing*, 14:227–248, 1980.

[Don05] W. Donnelly. *Per-Pixel Displacement Mapping with Distance Functions*, pages 123–136. Addison Wesley Professional, 2005.

[Fer04] A. Fernandes. Glsl tutorial. Web-page, referenced 18.10.2006, 2004. `http://www.lighthouse3d.com/opengl/glsl/index.php?ogldir1`.

[For02] T. Forsyth. Self-shadowing bump map using 3d texture hardware. *J. Graph. Tools*, 7(4):19–26, 2002.

[GG04] J. Greg and S. Green. Real-time animated translucency, 2004.

[Har96] J. Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *Visual Computer*, 12(10):527–545, 1996.

[HS99] W. Heidrich and H. Seidel. Realistic, hardware-accelerated shading and lighting. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 171–178, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co.

[Kil99] M. Kilgard. Perfect reflections and specular lighting effects with cube environment mapping. Technical report, 1999.

[Kje05] A. Kjellberg. Rendosity cinema 4d forum speed modeling session #2. Web-page, referenced 16.10.2006, 2005. `http://www.cartesiuscreations.com/galleries/speedmodelings/speed1.shtml`.

[Oki06] Okino Computer Graphics. Caustics rendering. Web-page, referenced 09.10.2006, 2006. `http://www.okino.com/conv/features/caustics/chess_pieces_by_rcl.jpg`.

[PA01] S. Premoze and M. Ashikhmin. Rendering natural waters. *Computer graphics forum*, 20(4):189–200, 2001.

[Par05] S. Partanen. A camera picture. Web-page, referenced 12.10.2006, 2005. `http://www.partanen.net/tmp/taustoja/Norja_taustakuva.jpg`.

[PO06] F. Policarpo and M. Oliveira. Relief mapping of non-height-field surface details. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 55–62, New York, NY, USA, 2006. ACM Press.

[POC05] F. Policarpo, M. Oliveira, and J. Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games*, pages 155–162, New York, NY, USA, 2005. ACM Press.

[Shi02] Peter Shirley. *Fundamentals of Computer Graphics*. A. K. Peters, Ltd., Natick, MA, USA, 2002.

[SN06] S. Seipel and A. Nivfors. Efficient rendering of multiple refractions and reflections in natural objects. *SIGRAD 2006*, 19(1), 2006.

[Tat06] N. Tatarchuk. Dynamic parallax occlusion mapping with approximate soft shadows. In *SI3D '06: Proceedings of the 2006 symposium on Interactive 3D graphics and games*, pages 63–69, New York, NY, USA, 2006. ACM Press.

[VIO02] A. Vlachos, J. Isidoro, and C. Oat. *Rippling reflective and refractive water*. Addison-Wesley, 2002.

[WD97] F. Weinhaus and V. Devarajan. Texture mapping 3d models of real-world scenes. *ACM Comput. Surv.*, 29(4):325–365, 1997.

[Wel04] T. Welsh. Parallax mapping with offset limiting: A per-pixel approximation of uneven surfaces. Technical report, 2004.

[YF03] H. Young and R. Freedman. *University Physics with Modern Physics*. Pearson Addison Wesley, 2003.

[YFCF06] C. Yung-Feng and C. Chun-Fa. Gpu-based ocean rendering. Technical report, 2006.

A SHADER PROGRAM

A.1 Vertex shader

```
/*
 * Copyright (C) 2006   Ville Timonen
 */

// This data is passed to the shader from the application.
attribute vec3 eyedir_passed;
attribute vec3 lightpos_passed;

// Light properties for the surface. Ambient is separate from global ambient.
// Light's own ambient changes according to attenuation, so it is not constant
// as global ambient is.
varying vec4 diffuse, ambient, global_ambient;

// These are interpolated for the fragment shader.
varying vec3 lightpos;
varying vec3 eyedir;


void main() {
        // Assigning the interpolated values.
        eyedir = eyedir_passed;
        lightpos = lightpos_passed;

        // These values depend on the light source and the surface material.
        diffuse = gl_FrontMaterial.diffuse * gl_LightSource[0].diffuse;
        ambient = gl_FrontMaterial.ambient * gl_LightSource[0].ambient;
        // Global ambient doesn't depend on any specific light source.
        global_ambient = gl_LightModel.ambient * gl_FrontMaterial.ambient;

        // Object vertex to eye space.
        gl_Position = ftransform();
```

```
    // Texture coordinates are the same for every texture unit.
    gl_TexCoord[0] = gl_MultiTexCoord0;
}
```

A.2 Fragment shader

```
/*
 * Copyright (C) 2006   Ville Timonen
 */

// We have to declare the same interpolated variables here in fragment shader
// as we declared in the vertex shader.
varying vec4 diffuse, ambient, global_ambient;
varying vec3 lightpos;
varying vec3 eyedir;

// 2D textures
uniform sampler2D tex_unit, t_normal_unit, normal_unit,
                  s_height_unit, t_height_unit;

// The environment map as a cube map
uniform samplerCube cube_unit;

// Bounding box height
const float relative_height = 0.05;

// Portion of the bounding box occupied by the layers
// Prefer calling them s (surface) and t (transparent) as in surface_heights.s
const vec2 surface_heights = vec2(0.2, 1.0);
// The base level of the bounding box for the layers
const vec2 surface_levels = vec2(-0.25, -0.5);

// Normalized data
vec3 eyenorm = normalize(eyedir);
```

```
vec3 lightnorm = normalize(lightpos);
float lightdist = length(lightpos);

// These are customizable
const float optical_density1 = 1.0;
const float optical_density2 = 1.6; // Represents glass

// The search intensities
// [0] = linear search, [1] = binary search
const int intersection_search[2] = { 24, 6 };
const int refraction_search[2] = { 8, 5 };
const int occlusion_search[2] = { 16, 5 };

// Color absorbtion
const vec3 absorb_color = vec3(1.0, 1.0, 0.0);
const float absorb_half = 0.25;



// This gives out the point of linear approximation for an intersection point
vec3 linearly_approximate(
                    in vec3 intersection_range1,
                    in vec3 intersection_range2,
                    in vec2 height_samples)
{
        return intersection_range1 + ((intersection_range2 - intersection_range1) *
                    // The following is the "multiplier"
                    ((intersection_range1.z - height_samples[0]) /
                    (height_samples[1] - height_samples[0] -
                     intersection_range2.z + intersection_range1.z)));
}



// This is used exclusively for transparent layer intersection refinement
vec3 refine_tlayer_int(
                    in vec3 intersection_range1,
                    in vec3 intersection_range2)
{
```

```
vec3 half_point;
float surface_sample;
vec2 surface_samples;
// Whether the surface samples got fetched
bool s_got_set = false;                                                    70
bool t_got_set = false;

// Refining the search with binary search
for (int i = 0; i < occlusion_search[1]; ++i) {

        // Taking the average
        half_point = (intersection_range1 + intersection_range2)*0.5;

        surface_sample = texture2D(t_height_unit, half_point.xy)*
                surface_heights.t + surface_levels.t;                      80

        // Branching depending on where the intersection is
        if (surface_sample < half_point.z) {
                intersection_range1 = half_point;
                surface_samples.s = surface_sample;
                s_got_set = true;
        } else {
                intersection_range2 = half_point;
                surface_samples.t = surface_sample;
                t_got_set = true;                                          90
        }
}

// If we don't have both limits sampled already, we sample
if (!s_got_set)
        surface_samples.s = texture2D(t_height_unit, intersection_range1.xy)*
                surface_heights.t + surface_levels.t;
if (!t_got_set)
        surface_samples.t = texture2D(t_height_unit, intersection_range2.xy)*
                surface_heights.t + surface_levels.t;                      100

return linearly_approximate(intersection_range1, intersection_range2,
```

```
                        surface_samples);
}


// This is used only for sampling the light intensity
float refraction_intensity(
                in vec3 normal,
                in float odens1,                                    110
                in float odens2)
{
        vec2 cosa;

        // Optical density ratio (index of refraction ratio)
        float odens_ratio = odens1 / odens2;

        // The cosines come handy, again
        cosa[0] = dot(-lightnorm, normal);
        cosa[0] = clamp(cosa[0], 0.0, 1.0);                         120
        cosa[1] = sqrt(1 - odens_ratio*odens_ratio*(1 - cosa[0]*cosa[0]));

        vec2 intensities = vec2(
                        (odens1 * cosa[0] - odens2 * cosa[1]) /
                        (odens1 * cosa[0] + odens2 * cosa[1]),

                        (odens1 * cosa[1] - odens2 * cosa[0]) /
                        (odens1 * cosa[1] + odens2 * cosa[0]));

        intensities *= intensities;                                 130
        intensities[0] = (intensities[0] + intensities[1])*0.5;

        // This is the final intensity
        return 1.0 - intensities[0];
}


// The light ray is marched through using this function, and the
// resulting per-color light vector is resulted.
```

```
vec4 get_light_vector(                                                    140
        in vec2 sampling_point)
{
    vec3 light_vector = vec3(1.0, 1.0, 1.0);

    // The idea is to travel through the light path to map all the occlusions.
    // We go to the end of the bounding box
    vec3 intersection_range1 = vec3(
            lightpos.x / lightpos.z * relative_height,
            lightpos.y / lightpos.z * relative_height,
            0.5);                                                         150

    vec3 stepsize = intersection_range1 / occlusion_search[0] * 2;

    // The first step should not be taken from the edge,
    // but already one step forward.
    float sample_height = texture2D(s_height_unit, sampling_point)*
        surface_heights.s + surface_levels.s; // This is the ground level
    intersection_range1 = vec3(sampling_point, sample_height);
    intersection_range1 += stepsize;
                                                                          160
    // We need to set the condition at start
    bool under = false;
    vec3 exit_point;

    float transparent_height = texture2D(t_height_unit, sampling_point)*
        surface_heights.t + surface_levels.t;
    if (transparent_height > sample_height) { // Inside the translucent layer?
        under = true;
        exit_point = vec3(sampling_point, sample_height);
    }                                                                     170


    vec2 surface_samples;

    // We automatically bail out when we hit our rendering point.
    // We include one extra step so the search will always end in
```

```
// an escape from the transparent layer
while (intersection_range1.z < 0.5+stepsize.z) {
        surface_samples.s = texture2D(s_height_unit, intersection_range1.xy)*
                surface_heights.s + surface_levels.s;
        surface_samples.t = texture2D(t_height_unit, intersection_range1.xy)*
                surface_heights.t + surface_levels.t;

        // If the surface layer gets penetrated
        if (surface_samples.s > intersection_range1.z) {
                light_vector = 0.0;
                break;
        }
        if (under && surface_samples.t < intersection_range1.z) {
                // This happens when we were under the surface and got out of it
                // First we want to know the precise intersection point
                vec3 ipoint = refine_tlayer_int(intersection_range1,
                                intersection_range1-stepsize);

                // We are dimming
                float dim_multiplier = pow(.5,
                                length(ipoint - exit_point)/absorb_half);
                light_vector *= vec3(1.0, 1.0, 1.0) -
                        absorb_color * (1.0 - dim_multiplier);

                // In addition reflection intensity is calculated
                vec3 normal = (texture2D(t_normal_unit, ipoint.xy));
                // We rather have it normalized to avoid length calculations
                normal -= vec3(0.5, 0.5, 0.5);
                normal.y = -normal.y;
                normal = normalize(-normal);

                // Applying the result into the light ray
                light_vector *= refraction_intensity(normal,
                                optical_density1, optical_density2);
                under = false;

        } else if (!under && intersection_range1.z < surface_samples.t) {
```

```
                        // We previously were outside and now we're in
                        // First we want to know the precise intersection point
                        vec3 ipoint = refine_tlayer_int(intersection_range1-stepsize,
                                        intersection_range1);
                        exit_point = ipoint;
                        under = true;

                        // Now we again calculate the refraction intensity
                        // for this boundary excession
                        vec3 normal = texture2D(t_normal_unit, ipoint.xy);
                        // We rather have it normalized to avoid length calculations
                        normal -= vec3(0.5, 0.5, 0.5);
                        normal.y = -normal.y;
                        normal = normalize(normal);

                        // Applying the result into the light ray
                        light_vector *= refraction_intensity(normal,
                                        optical_density2, optical_density1);
                }

                intersection_range1 += stepsize;
        }

        return vec4(light_vector, 1.0); // Alpha is 1.0 always
}

// Used to fetch an assembled surface sample
vec4 assemble_surface_sample(
                in vec2 sampling_point)
{
        // Lighting depends on the normal
        vec3 normal = texture2D(normal_unit, sampling_point);
        normal -= vec3(0.5, 0.5, 0.5);
        normal.y = -normal.y;
        normal = normalize(normal);
```

```
        // The actual material color
        vec4 material_color = texture2D(tex_unit, sampling_point);

        float diffuse_intensity = max(dot(normal, lightnorm), 0.0);

        // The base level is global ambient
        vec4 color = global_ambient;

        // Now if the lighting should be added (facing the light)..
        if (diffuse_intensity > 0.0) {                                          260
                // Now that light should be calculated,
                // we first calculate the attenuation.
                // Attenuation is, according to OpenGL specification,
                // 1/(a+bd+cd^2), where d is distance, and a, b and c are constant,
                // linear and quadratic attenuation factors,
                // given by OpenGL and defined by the OpenGL application.
                float attenuation = 1.0 / (
                                gl_LightSource[0].constantAttenuation +
                                gl_LightSource[0].linearAttenuation * lightdist +
                                gl_LightSource[0].quadraticAttenuation * lightdist*lightdist);  270

                // Attenuation affects every component of the light.
                color += (diffuse * diffuse_intensity + ambient) * attenuation *
                        get_light_vector(sampling_point); // Occlusions
        }

        return material_color * color;
}

                                                                                280
// This is used for searching the first encounter with either of
// the two surfaces.
// Returns which surface is first (0 both, 1 transparent, 2 solid)
int search_intersection(
                out vec3 intersection_range1,
                out vec3 intersection_range2,
                in int steps)
```

```
{
    // We go to the end of the bounding box.
    intersection_range2 = vec3(                                          290
                    eyedir.x / eyedir.z * relative_height,
                    eyedir.y / eyedir.z * relative_height,
                    0.5);

    vec3 stepsize = -intersection_range2 / steps * 2;

    // The first should not be taken from the edge,
    // but already one step forwards.
    intersection_range2 += stepsize;
    // We need to start from the current sampling point.                 300
    intersection_range2.xy += gl_TexCoord[0].st;

    vec2 surface_samples;
    int encounter = 0; // This to avoid a probable bug in the GLSL compiler
    // We escape from the inside
    while (true) {
        surface_samples.s = texture2D(s_height_unit, intersection_range2.xy)*
                surface_heights.s + surface_levels.s;
        surface_samples.t = texture2D(t_height_unit, intersection_range2.xy)*
                surface_heights.t + surface_levels.t;                    310

        // If either of the surface gets penetrated, we escape.
        if (surface_samples.s > intersection_range2.z) {
            if (surface_samples.t > intersection_range2.z) {
                // Both penetrated
                break;
            }
            encounter = 2;
            break;
        }                                                                320
        if (surface_samples.t > intersection_range2.z) {
            encounter = 1;
            break;
        }
```

```
                intersection_range2 += stepsize;
        }

        // The range is now from upper limit to lower limit:
        // intersection_range1 - intersection_range2
        intersection_range1 = intersection_range2 - stepsize;                    330
        return encounter;
}


// Binary seach refinement for a range
int refine_range(
                inout vec3 intersection_range1,
                inout vec3 intersection_range2,
                out vec2 height_samples,
                in int iterations,                                               340
                in int encounter)
{
        int i;

        vec2 surface_samples;

        vec3 half_point;
        // If the upper surface is not yet revealed,
        // we need to keep double sampling until we know
        // This doesn't loop unless encounter isn't known.                       350
        for (i = 0; !encounter && i < iterations; ++i) {
                half_point = (intersection_range1 + intersection_range2) * 0.5;
                surface_samples.s = texture2D(s_height_unit, half_point.xy)*
                        surface_heights.s + surface_levels.s;
                surface_samples.t = texture2D(t_height_unit, half_point.xy)*
                        surface_heights.t + surface_levels.t;

                if (surface_samples.s > half_point.z) {
                        // At least first surface got penetrated,
                        // thus lower limit gets raised                          360
                        intersection_range2 = half_point;
```

```
                // If the transparent layer did not get penetrated,
                // we know that the solid layer is first
                if (surface_samples.t < half_point.z)
                        encounter = 2;

                // Both layers were penetrated, we're continuing.
        } else {
                // Solid layer did not get penetrated, if transparent layer
                // did not get penetrated either, we move range1 and proceed
                if (surface_samples.t < half_point.z)
                        intersection_range1 = half_point;
                else {
                        // Now we know that transparent layer is first,
                        // thus range2 gets moved and we escape
                        encounter = 1;

                        // At least the solid layer is passed.
                        intersection_range2 = half_point;
                }
        }
}

// If it's a tie, we'll prefer using the solid layer as the encounter
// for an optimization
if (!encounter)
        encounter = 2;

// Now we need to either skip further sampling or
// continue sampling the other surface.         Let 'i' decide.
for (; i < iterations; ++i) {
        half_point = (intersection_range1 + intersection_range2) * 0.5;

        // We only sample the other surface
        if (encounter == 1)
                height_samples[0] = texture2D(t_height_unit, half_point.xy)*
                        surface_heights.t + surface_levels.t;
```

```
                else
                        height_samples[0] = texture2D(s_height_unit, half_point.xy)*
                                surface_heights.s + surface_levels.s;

                if (height_samples[0] < half_point.z)
                        // Penetration
                        intersection_range1 = half_point;
                else
                        intersection_range2 = half_point;
        }

        // Now the final range is refined and the sampled surface is known.
        if (encounter == 1) { // Transparent layer
                // We can avoid sampling if we use value already sampled..
                // not doing it at the moment
                height_samples[0] = texture2D(t_height_unit, intersection_range1.xy)*
                        surface_heights.t + surface_levels.t;
                height_samples[1] = texture2D(t_height_unit, intersection_range2.xy)*
                        surface_heights.t + surface_levels.t;
        } else { // Solid layer
                height_samples[0] = texture2D(s_height_unit, intersection_range1.xy)*
                        surface_heights.s + surface_levels.s;
                height_samples[1] = texture2D(s_height_unit, intersection_range2.xy)*
                        surface_heights.s + surface_levels.s;
        }

        return encounter;
}


// With this we can sample the surface for refraction
vec4 sample_refraction(
                in vec3 intersection_point,
                in vec3 viewing_ray)
{
        // This procedure resembles a lot of the intersection searching
```

```
// We go to the end of the bounding box.
vec3 intersection_range2 = vec3(
                    viewing_ray.x / viewing_ray.z * relative_height,
                    viewing_ray.y / viewing_ray.z * relative_height,
                    0.5);                                                    440

vec3 stepsize = -intersection_range2 / refraction_search[0] * 2;

// The first should not be taken from the edge,
// but already one step forwards.
intersection_range2 += intersection_point;
intersection_range2 += stepsize;

vec2 surface_samples;
                                                                             450
// Then we loop for the linear search
while (true) {
        surface_samples.t = texture2D(s_height_unit, intersection_range2.xy)*
                    surface_heights.s + surface_levels.s;

        if (surface_samples.t > intersection_range2.z)
                    break; // We hit the surface

        intersection_range2 += stepsize;
}                                                                            460
vec3 intersection_range1 = intersection_range2 - stepsize;

vec3 half_point;
float surface_sample;
bool s_got_set = false;

// Refining the search with binary search
for (int i = 0; i < refraction_search[1]; ++i) {
        half_point = (intersection_range1 + intersection_range2)*0.5;
                                                                             470
        surface_sample = texture2D(s_height_unit, half_point.xy)*
                    surface_heights.s + surface_levels.s;
```

```
                if (surface_sample < half_point.z) {
                        intersection_range1 = half_point;
                        surface_samples.s = surface_sample;
                        s_got_set = true; // This means that 's' represents a proper value
                } else {
                        intersection_range2 = half_point;
                        surface_samples.t = surface_sample;                    480
                }
        }

        // It is possible that surface_samples.s was never refreshed,
        // in this case it has to be sampled again (actually for the first time)
        if (!s_got_set)
                surface_samples.s = texture2D(s_height_unit, intersection_range1.xy)*
                        surface_heights.s + surface_levels.s;

        // Linear approximation                                                490
        intersection_range1 = linearly_approximate(intersection_range1,
                        intersection_range2, surface_samples);

        // Dimming the refraction if the absorbtion model is used
        float dim_multiplier = pow(.5,
                        length(intersection_range1 - intersection_point)/absorb_half);
        vec4 colorleft = vec4(vec3(1.0, 1.0, 1.0) -
                        absorb_color * (1.0 - dim_multiplier), 1.0);

        return assemble_surface_sample(intersection_range1.xy) * colorleft;    500
}


// This is also self-explanatory, sampling a reflection
vec4 sample_reflection(
                in vec3 ray,
                in vec3 point,
                in vec3 normal)
{
```

```
bool local_sample = false;                                          510
vec2 texpos;

// Deciding whether to sample locally or not
if (ray.z < 0.0) { // Possible
        local_sample = true;

        // 't' is the scalar in the vector
        float t = -(point.z - (0.5 * surface_heights.s + surface_levels.s)) /
                ray.z;
        // The texturing position                                   520
        texpos = point.xy + (ray.xy * t * relative_height);

        // We'll see if it lands on the actual surface
        if (texpos.x < 0.0 || texpos.x > 4.0 ||
                        texpos.y < 0.0 || texpos.y > 4.0)
                local_sample = false; // Runs out of the surface
}

vec4 env_color;
                                                                    530
// Sampling it either locally or from environment map.
if (local_sample)
        env_color = texture2D(tex_unit, texpos);
        // It can, alternatively, be sampled as refractions.
        // Avoided, at the moment, as an optimization
        //env_color = sample_refraction(point, ray);
else
        env_color = textureCube(cube_unit, ray);


                                                                    540
// Light calculation
vec3 half = normalize(eyenorm + lightnorm);
float specular_intensity = max(dot(normal, half), 0.0);

float diffuse_intensity = max(dot(normal, lightnorm), 0.0);
```

```
        vec4 color = global_ambient;

        // Now if the lighting should be added (facing the light)..
        if (diffuse_intensity > 0.0) {                                          550
                // This is done the same way as before, read the comments
                float attenuation = 1.0 / (
                                gl_LightSource[0].constantAttenuation +
                                gl_LightSource[0].linearAttenuation * lightdist +
                                gl_LightSource[0].quadraticAttenuation * lightdist*lightdist);

                color += (diffuse * diffuse_intensity + ambient) * attenuation;
                // NOTE: As an optimization, occlusions are not searched.
        }
                                                                                560
        // Applying the lighting, including the intensified specular reflection
        return env_color * color +
                gl_LightSource[0].specular * pow(specular_intensity, 400.0) * 10.0;
}


// This is used to compose the entire appearance of a transparent layer pixel
vec4 compose_transparent(
                in vec3 intersection_point)
{                                                                               570
        float optical_density_ratio = optical_density1 / optical_density2;
        vec3 normal = (texture2D(t_normal_unit, intersection_point.xy));
        // We rather have it normalized in order to avoid length calculations
        normal -= vec3(0.5, 0.5, 0.5);
        normal.y = -normal.y;
        normal = normalize(-normal);

        vec2 cosa;
        cosa[0] = dot(-eyenorm, normal);
        cosa[0] = clamp(cosa[0], 0.0, 1.0);                                     580
        cosa[1] = sqrt(1 - optical_density_ratio*optical_density_ratio*(1 - cosa[0]*cosa[0]));

        // The refracted viewing ray
```

```
        vec3 refr_vray =
                (-eyenorm * optical_density_ratio) +
                (normal * (cosa[1] - optical_density_ratio*cosa[0]));

        // The reflected viewing ray
        vec3 refl_vray =
                -eyenorm -                                                    590
                (normal * cosa[0] * 2);

        // Sampling the rays
        vec4 refraction_sample = sample_refraction(intersection_point, refr_vray);
        vec4 reflection_sample =
                sample_reflection(refl_vray, intersection_point, -normal);

        // At this point it would seem proper to compute intensities
        vec2 intensities = vec2(
                        (optical_density1 * cosa[0] - optical_density2 * cosa[1]) /   600
                        (optical_density1 * cosa[0] + optical_density2 * cosa[1]),

                        (optical_density1 * cosa[1] - optical_density2 * cosa[0]) /
                        (optical_density1 * cosa[1] + optical_density2 * cosa[0]));

        intensities *= intensities;
        intensities[0] = (intensities[0] + intensities[1])*0.5;
        intensities[1] = 1.0 - intensities[0];

        return refraction_sample * intensities[1] +                           610
                reflection_sample * intensities[0];
}


// Here it all starts
void main() {
        vec3 intersection_range1, intersection_range2;
        vec2 height_samples;
        int encounter;
                                                                              620
```

```
        // Linear search
        encounter = search_intersection(intersection_range1, intersection_range2,
                        intersection_search[0]);

        // Binary refinement
        encounter = refine_range(intersection_range1, intersection_range2,
                        height_samples, intersection_search[1], encounter);

        // Linear approximation
        vec3 intersection_point = linearly_approximate(intersection_range1,
                        intersection_range2, height_samples);

        // Branching:   translucent layer sampling or solid layer sampling
        vec4 color;
        if (encounter == 1)
                color = compose_transparent(intersection_point);
        else
                color = assemble_surface_sample(intersection_point.xy);

        // The final appearance
        gl_FragColor = color;
}
```