GLOBAL ILLUMINATION EFFECTS WITH SAMPLED GEOMETRY

A dissertation submitted to

the Budapest University of Technology and Economics

in fulfillment of the requirements

for the degree of Doctor of Philosophy (Ph.D)

by

Umenhoffer Tamás

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics

advisor
Szirmay-Kalos László

2009

Contents

1 Introduction **2**

1.1 Rendering equation . . . 2
1.2 Volumetric rendering equation . . . 3
1.3 Local illumination . . . 4
1.4 The GPU pipeline . . . 5
1.4.1 Vertex shader . . . 6
1.4.2 Clipping, rasterization and interpolation . . . 7
1.4.3 Fragment shader . . . 7
1.4.4 Merging . . . 7
1.4.5 Render-to-texture feature . . . 8
1.5 Using GPUs for global illumination . . . 8
1.6 Structure and objectives of the thesis work . . . 9
1.6.1 Diffuse material rendering . . . 9
1.6.2 Specular material rendering . . . 10
1.6.3 Participating media rendering . . . 10

2 Previous work **11**

2.1 Displacement mapping . . . 12
2.1.1 Parallax mapping . . . 13
2.1.2 Binary search . . . 14
2.1.3 Secant search . . . 14
2.1.4 Linear search . . . 15
2.1.5 Relief mapping . . . 15
2.2 Impostors . . . 15
2.3 Sampled representation of indirect illumination . . . 17
2.3.1 Light maps and shadow maps . . . 17
2.3.2 Environment mapping and its generalization with geometry information . 19
Parallax correction . . . 21
Linear search . . . 21
Secant search . . . 21
Binary search . . . 22
Single reflections or refractions using the sampled geometry . . . 22
Inter-object multiple reflections and refractions . . . 22
2.3.3 Caustic mapping . . . 23
2.4 Rendering volumetric media on the GPU . . . 25

I Diffuse Material Rendering **28**

3 Robust diffuse environment mapping **29**

3.1 The new algorithm . . . 30
3.2 Results . . . 31

3.3 Conclusions 35

4 Volumetric ambient occlusion **37**
4.1 The new model of ambient lighting 38
4.1.1 Replacing ray tracing by containment tests 38
4.1.2 Exploiting the distance to the separating surface 41
4.1.3 Noise reduction with interleaved sampling 45
4.2 Results 45
4.3 Conclusions 49

II Specular Material Rendering **51**

5 Robust specular reflections and refractions **52**
5.1 The new method of tracing secondary rays 52
5.1.1 Generation of layered distance maps 52
5.1.2 Ray-tracing of layered distance maps 53
Linear search 53
Acceleration with min-max distance values 54
5.2 Application to multiple reflections and refractions 54
5.3 Results 56
5.4 Conclusion 58

6 Caustic triangles **61**
6.1 The new caustics generation algorithm 62
6.1.1 Photon tracing pass 63
6.1.2 Caustic reconstruction pass 64
6.1.3 Camera pass 65
6.2 Optimizations 65
6.3 Results 66
6.4 Conclusions 66

III Participating Media Rendering **69**

7 Spherical billboards **70**
7.1 The new method using spherical billboards 71
7.1.1 "Gouraud shading" for spherical billboards 72
7.1.2 GPU Implementation of spherical billboards 74
7.2 Rendering explosions 74
7.2.1 Dust and smoke 74
7.2.2 Fire 75
7.2.3 Layer composition 77
7.3 Results 80
7.4 Conclusion 80

8 Hierarchical Depth Impostors **82**
8.1 The new method using particle hierarchies 82
8.1.1 Generating a depth impostor 83
8.1.2 Using depth impostors during light passes 83
8.1.3 Using depth impostors during final gathering 84
8.1.4 Objects in clouds 84
8.2 Results 85

8.3 Conclusions 85

9 Parcipating media rendering with illumination networks 88

9.1 The new method using illumination networks 88

9.1.1 Setting the parameters of the illumination network 90

9.1.2 Iterating the illumination network 90

9.2 Results 91

9.3 Conclusions 93

10 Conclusions 95

11 Thesis summary 96

Chapter 1

Introduction

In most of the fields of computer graphics our main goal is to create images that are similar to the ones we see in real life. This requires a deep understanding of how light interacts with real world objects and even with the media it travels within. All these physical rules need to be described by a mathematic formula that can be evaluated by computers. The mathematical model of light transport is the so called **rendering equation**, which is — in its pure form — so highly computation intensive that it cannot be evaluated in real-time even for simple scenarios. On the other hand, real-time rendering is an essential requirement in case of interactive graphical applications.

As we simulate light-object and light-media interactions, we need a way to store the virtual model of real world objects, which have almost infinite geometric detail. To efficiently do this we need either some kind of mathematical model or a rough simplification of the original surfaces.

This thesis work presents real-time methods that are based upon the discretization of scene geometry to address the storing problem. With sampled geometry light ray-surface intersections can be efficiently computed, which is the essential component of the advanced evaluation of the rendering equation. The presented algorithms deal with large range of illumination effects from diffuse indirect illumination to specular reflections and even participating media rendering.

1.1 Rendering equation

In scenes not incorporating **participating media** it is enough to calculate the light intensity that is measured by the **radiance** at surface points since in vacuum the transferred radiance does not change along a light ray. The radiance reflected off a surface is affected by the emission of this point, the illumination provided by other surfaces, and the optical properties of the material at this point.

Formally this dependence is characterized by a Fredholm type integral equation, which is called the **rendering equation**. Denoting the radiance of point $\vec{y}$ in direction $\vec{\omega}'$ by $L(\vec{y}, \vec{\omega}')$, the rendering equation expresses radiance $L(\vec{x}, \vec{\omega})$ at point $\vec{x}$ in direction $\vec{\omega}$ as its own **emission** $L^e(\vec{x}, \vec{\omega})$ and its reflected radiance $L^r(\vec{x}, \vec{\omega})$:

$$L(\vec{x}, \vec{\omega}) = L^e(\vec{x}, \vec{\omega}) + L^r(\vec{x}, \vec{\omega}), \tag{1.1}$$

The **reflected radiance** is the sum of contributions scattered from all incoming directions:

$$L^r(\vec{x}, \vec{\omega}) = \int_{\Omega'} L(\vec{y}, \vec{\omega}') f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cos^+ \theta'_{\vec{x}} \mathrm{d}\omega', \tag{1.2}$$

where $\vec{y}$ is the point visible from $\vec{x}$ at direction $\vec{\omega}'$, Ω' is the directional sphere, $f_r(\vec{\omega}', \vec{x}, \vec{\omega})$ is the **BRDF**, and $\theta'_{\vec{x}}$ is the incident angle between the surface normal and direction $-\vec{\omega}'$ at $\vec{x}$. If incident angle $\theta'_{\vec{x}}$ is greater than 90 degrees — i.e. the light illuminates the "back" of the surface — then the negative cosine value should be replaced by zero, which is indicated by

superscript $^+$ in $\cos^+$. **Illuminating point** $\vec{y}$ is unambiguously determined by **shaded point** $\vec{x}$ and illumination direction $\vec{\omega}'$.

The rendering equation describes the light transfer at a single representative wavelength. In fact, we evaluate the radiance spectrum at a few representative wavelengths, most conveniently at the wavelengths of red, green, and blue since this relieves us from the color matching calculation.

If we knew the radiance of all other points $\vec{y}$, then the solution of the rendering equation would be equivalent to the evaluation of an integral. Unfortunately, the radiance of other points is not known, but similar equations should be solved for them. In fact, the unknown radiance function shows up not only on the left side of the rendering equation, but also inside the integral of the right side. The rendering equation is thus an integral equation. Solving integral equations is difficult and requires a lot of computation.

1.2 Volumetric rendering equation

If light absorbing and scattering materials are present in space the rendering equation gets even more complicated. Incoming radiance will depend not only on the geometric factor but also on the absorbing and scattering properties of the media along the light path from one surface point to another. In other words we should consider how the light goes through **participating media**. The change of radiance L on path of length $\mathrm{d}s$ and of direction $\vec{\omega}$ depends on different phenomena:

- **Absorbtion** and **outscattering**: the light is absorbed or scattered out from its path when photons collide with the particles. If the probability of collision in a unit distance is τ, then the radiance changes by $-\tau L \mathrm{d}s$ due to the collisions. After collision a particle may be either absorbed or reflected with the probability of **albedo** a.

- **Emission**: the radiance may be increased by the photons emitted by the participating media (e.g. fire). This increase is $L^e \mathrm{d}s$ where L^e is the emission density.

- **In-scattering**: photons originally flying in a different direction may be scattered into the considered direction. The expected number of scattered photons from differential solid angle $\mathrm{d}\omega'$ equals to the product of the number of incoming photons and the probability that the photon is scattered from $\mathrm{d}\omega'$ to $\vec{\omega}$. The scattering probability is the product of the collision probability (τ), the probability of not absorbing the photon (a), and the probability density of the reflection direction, called **phase function** P. This function can be measured from the real world medium to be displayed, or preferably it can be an analytic expression. The choice of the scattering phase function is a compromise between realism and mathematical tractability.

  The simplest phase function is the **isotropic phase function**:

  $$P(\vec{\omega}', \vec{\omega}) = \frac{1}{4\pi}$$

  where the factor of $1/4\pi$ results from the normalization condition of a probability density function:

  $$\int_{\Omega'} P(\vec{\omega}', \vec{\omega}) \mathrm{d}\omega' = 1.$$

  A more complex anisotropic function is the **Henyey-Greenstein phase function**, which is widely used in light scattering simulation [HG40, CS92]:

  $$P(\vec{\omega}', \vec{\omega}) = \frac{1}{4\pi} \cdot \frac{3(1-g^2)(1+(\vec{\omega}' \cdot \vec{\omega})^2)}{2(2+g^2)(1+g^2-2g(\vec{\omega}' \cdot \vec{\omega}))^{3/2}}, \tag{1.3}$$

  where $g \in (-1, 1)$ is a material property describing how strongly the material scatters forward or backward. This function can be used in a wide range of different areas from

natural phenomena — like smoke or clouds — rendering to medical simulation — like simulating light scattering in human dermis or aorta.

Taking into account all incoming directions Ω', we obtain the following radiance increase due to in-scattering:

$$\tau(s)\mathrm{d}s \cdot a(s) \left(\int_{\Omega'} L(s, \vec{\omega}') P(\vec{\omega}', \vec{\omega}) \mathrm{d}\omega' \right).$$

Adding the discussed changes, we obtain the following **volumetric rendering equation** for radiance L of the ray at $s + \mathrm{d}s$:

$$L(s + \mathrm{d}s, \vec{\omega}) = (1 - \tau(s)\mathrm{d}s) L(s, \vec{\omega}) + L^e(s, \vec{\omega})\mathrm{d}s + \tau(s)\mathrm{d}s \cdot a(s) \int_{\Omega'} L(s, \vec{\omega}') P(\vec{\omega}', \vec{\omega}) \mathrm{d}\omega'. \quad (1.4)$$

This is also an integral equation, and the computation of the in-scattering term makes the evaluation difficult. Depending on the runtime and quality requirements there are different ways to incorporate the volumetric rendering equation into the rendering equation. We can choose to use the volumetric equation only at light paths from visible surface points to the eye, or at light paths from light sources to visible surface points so the shadows of participating media will show up. If the presence of opaque objects are taken into account while solving the volumetric equation shadowing and light shafts will appear. One can also choose to solve the entire volumetric equation, or can use an approximate solution with eliminating, or somehow approximating the in-scattering term.

1.3 Local illumination

Local illumination methods take a drastic approach, and approximate the radiance to be reflected by a known term $L^{in}(\vec{x}, \vec{\omega}')$, which is some approximation of the incoming radiance at point $\vec{x}$:

$$L(\vec{x}, \vec{\omega}) = L^e(\vec{x}, \vec{\omega}) + \int_{\Omega'} L^{in}(\vec{x}, \vec{\omega}') f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cos^+ \theta'_{\vec{x}} \mathrm{d}\omega', \quad (1.5)$$

The name of the method came from the fact that — if shadows are ignored — radiance calculations depend only on local properties of the surface. Visible surface points should be searched only from the camera. To approximate incoming radiance, usually **abstract light sources** are used, as they provide illumination just in a single direction for each point.

Graphics accelerator hardware uses **rasterization** and the **z-buffer algorithm** to compute primary visibility, i.e. to determine what is visible from the camera, and any further calculation is independent of the scene geometry and illumination of other surface points, so it can be highly parallelized to achieve real-time performance for complex scenes (see Section 1.4).

The local illumination model examines only one-bounce light paths and ignores multiple reflections. This clearly results in some illumination deficit, so the images will be darker than expected. More importantly, those surfaces that are not directly visible from any light sources will be completely dark. The old trick to add the missing illumination in a simplified form is the introduction of the **ambient light**.

Let us assume that there is some ambient lighting in the scene of intensity L^a that is constant for every point and every direction. According to the rendering equation (Equation 1.2), the reflection of this ambient illumination in point $\vec{x}$ and at viewing direction $\vec{\omega}$ is

$$L(\vec{x}, \vec{\omega}) = \int_{\Omega'} L^a f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cos^+ \theta'_{\vec{x}} \mathrm{d}\omega' = L^a \; a(\vec{x}, \vec{\omega}),$$

where

$$a(\vec{x}, \vec{\omega}) = \int_{\Omega'} f_r(\vec{\omega}', \vec{x}, \vec{\omega}) \cos^+ \theta'_{\vec{x}} \mathrm{d}\omega'$$

is the **albedo** of the surface. The albedo is the probability that a photon arriving at a surface from the viewing direction is not absorbed by the surface. Note that the albedo is not independent of the BRDF but can be expressed from it, and with the exception of **diffuse** surfaces, it is not constant but depends on the viewing direction. For diffuse surfaces, the correspondence of the albedo and the **diffuse reflectivity** is

$$a(\vec{x}) = \int_{\Omega'} k_d(\vec{x}) \cos^+ \theta'_{\vec{x}} \mathrm{d}\omega' = k_d(\vec{x})\pi. \tag{1.6}$$

As the ambient illumination model ignores the geometry of the scene, the resulting images are plain and do not have a 3D appearance. A physically correct approach would be the solution of the rendering equation that can take into account all factors missing in the classical ambient lighting model. However, this approach is too expensive computationally when dynamic scenes need to be rendered in real-time.

Instead of working with the rendering equation, local approaches examine only a neighborhood of the shaded point. **Ambient occlusion** [Hay02, PG04, KA06] and **obscurances** [ZIK98, IKSZ03] methods compute just how "open" the scene is in the neighborhood of a point, and scale the ambient light accordingly. Originally, the neighborhood had a sharp boundary in ambient occlusion and a fuzzy boundary in the obscurances method, but nowadays these terms refer to similar techniques. As the name of ambient occlusion became more popular, we also use this term in this thesis.

If not only the openness of the points is computed but also the average of open directions is obtained, then this extra directional information can be used to extend ambient occlusion from constant ambient light to environment maps [PG04]. In the **spectral** extensions the average spectral reflectivities of the neighborhood and the whole scene are also taken into account, thus even **color bleeding** effects can be cheaply simulated [MSC03, Bun05].

Since ambient occlusion is the "local invisibility of the sky", real-time methods rely on scene representations where the visibility can be easily determined. These scene representations include the approximation of surfaces by disks [Bun05, HJ07] or spheres [SA07]. Instead of dealing directly with the geometry, the visibility function can also be approximated [CAM08]. A **cube map** or a **depth map** [Mit07, Sai08] rendered from the camera can also be considered as a sampled representation of the scene. Since these maps are already in the texture memory of the GPU, a fragment shader program can check the visibility for many directions. The method called **screen-space ambient occlusion** [Mit07] took the difference of the depth values. **Horizon split ambient occlusion** [Sai08] generated and evaluated a horizon map on the fly.

1.4 The GPU pipeline

Though local illumination makes simplifications that cause notable quality loss in visual appearance, the rough approximations make real-time rendering possible through high parallelization and a special purpose hardware called the graphics accelerator. We often refer to graphics cards with the name **GPU** which stands for the denomination of the processors located on the accelerator cards, the **Graphics Processing Units**. In this dissertation we assume a **Shader Model 3** compatible GPU whose pipeline is shown in Figure 1.1.

Controlling the pipeline is possible from the CPU through a special **3D API**, which is in practice either **Direct3D** or **OpenGL**. 3D APIs are used to provide the input and define the output of the computation outplaced to the dedicated hardware. The functionalities of the rendering pipeline are set through **render states**, and computation is evoked with **render calls**. The output of the rendering is a color buffer which will be displayed on the monitor,

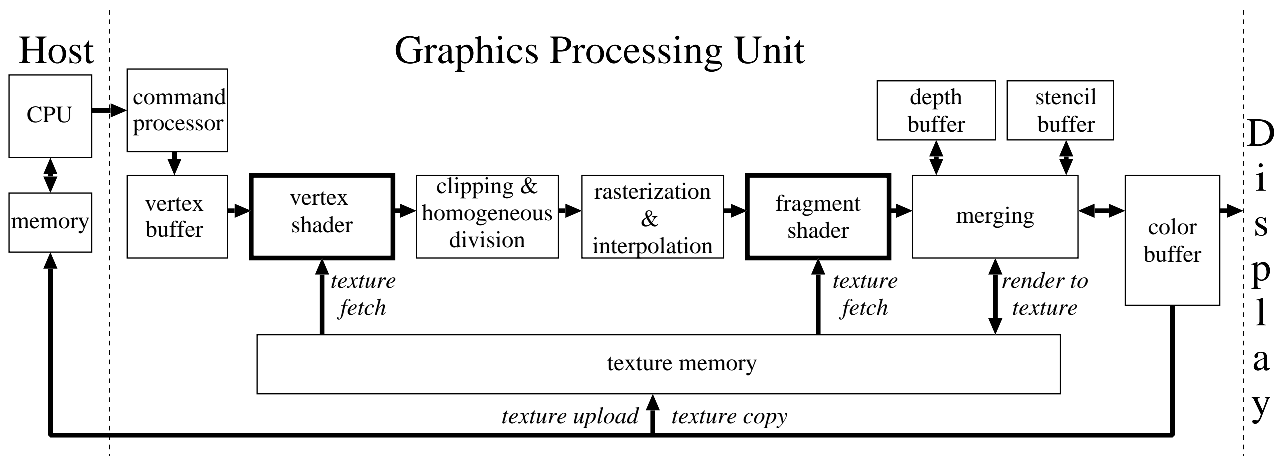


Figure 1.1: Shader Model 3 architecture.

while the input is some kind of geometry data of the virtual world. The virtual world consists of objects. Every object is defined in its own modeling space where the definition is simple. Object definitions are mathematical formulae that identify which points belong to the surface of the object. Though plenty of different models of surface definition exist, GPUs can process only triangles. As in many cases, modelling the objects is easier in other type of definitions than triangles, these descriptions should be converted to **triangle meshes**. The approximation of surfaces by triangles is called **tessellation**.

Triangles output by tessellation are defined by triplets of vertices given in modeling space. For every vertex, beside its position, local surface characteristics required for shading are also calculated. These usually include the **normal vector** of the surface at the given vertex and BRDF parameters, like diffuse or specular reflectivity or shininess. A useful property is a **texture coordinate** that references surface properties stored in one-, two- or three-dimensional arrays called **textures**. The records of triangle vertex data are stored in huge arrays called **vertex buffers**, in which three consecutive vertex records form a triangle. Except deformable geometry, vertex buffers are uploaded to the video memory of the GPU only once. Later to generate the image of one particular mesh, the CPU program should bind the right vertex buffer and issue a draw call to start the pipeline.

1.4.1 Vertex shader

The first parallelized part of the pipeline is the vertex shader that processes vertices one-by-one and usually executes transformation and per-vertex lighting. To perform shading, vertex positions and normals have to be transformed into the coordinate system where the camera and lights are specified. This is the **world space**. Transformation parameters define where the object actually resides in the virtual world, and the computation itself is referred to as the **modelling** or **world transformation**. Making this transformation time dependent, objects can be animated in the virtual world.

Where a triangle must be drawn on the screen is found by applying the **camera**, **projection** and finally **viewport** transformations after world transformation. Camera transformation translates and rotates the virtual world as it is seen in the point of view of the camera, making the z-axis look into the camera view direction. Projection transformation defines the size of the window the camera can see through, and distorts the scene to make view rays meeting in the camera become parallel if **perspective** projection is used to mimic real world cameras.

The **vertex shader** is one of the replaceable part of the pipeline, which means that its behavior can be freely programmed in a special programming language. The base functionality remains — namely it should project the triangles to the screen — but how it is performed, and

what kind of other information is also calculated is not specified.

1.4.2 Clipping, rasterization and interpolation

After projection the useful part of the virtual world lies in an axis-aligned box defined by inequalities $-1 < x < 1, -1 < y < 1, 0 < z < 1$, to which triangles are clipped. Finally these **normalized screen space** coordinates — more precisely the x and y coordinates — are transformed to screen pixel coordinates (**screen space**) to identify where exactly they should be drawn in the output buffer.

After computing final vertex colors (with or without lighting) and screen space positions, **rasterization** and **linear interpolation** are performed. During rasterization pixels which fall into the interior of the projection of the triangle are identified by a scan line triangle filling algorithm. The properties (e.g. colors and position) of the individual pixels are obtained from the vertex properties using **linear interpolation**. The main advantage of this scan line algorithm is that a property of a pixel can be obtained with a single addition from the property of the previous pixel, only an increment constant should be calculated once for each property and for each triangle.

1.4.3 Fragment shader

Pixels identified by rasterization are processed independently and parallel. For each pixel a texturing operation is performed which enables per-pixel color definitions with the use of **textures**. Textures are usually 2D arrays of color records in video memory. How these textures should be mapped onto a triangle is specified **texture coordinates** assigned to every vertex. These coordinates are also interpolated linearly within the triangles like other attributes. One can decide to reject vertex color information and use texture colors instead, or can combine these two colors (usually multiply) together to use both lighting and detailed surface color variations. An important feature of textures is that a number of filtering techniques can be applied. Textures are initialized either by the CPU code or their can be a copied from the frame buffer. Both color buffer and textures can be read back to CPU memory at any time. We should always consider that these data movings have heavy time cost.

Fragment shaders are the second replaceable part of the pipeline. Their inputs are interpolated vertex shader output values. Plenty of arithmetic, geometric and texture sampling functions can be used to calculate the final fragment color (and even fragment depth), which enables a wide variety of special shading effects that can be implemented on the GPU.

1.4.4 Merging

Pixel colors after texturing stage are not directly written to the frame buffer, but they are combined with the data already stored at the same location. This operation is called **merging**. During this stage not only color values but also additional data is used, like depth, stencil or alpha values. Besides color buffer memory a so called **depth buffer** is also maintained, that contains screen space z coordinates. Whenever a triangle is rasterized to a pixel, the color and the depth are overwritten only if the new depth value is less than the depth stored in the depth buffer, this test is called the **depth test**. As a result, we get a rendering of triangles correctly occluding each other. This process is commonly called the **depth buffer algorithm**. Depth testing can be turned off, or can be reversed so only furthest pixels will be kept. If a pixel passes the depth test not only color values but depth values are also overwritten. This **depth writing** can be turned off even if depth testing is turned on, which is useful for transparent material rendering.

Transparent surface rendering is achieved with **blending**. Color buffers can have four components, where the fourth component beside red, green and blue channels is an **alpha** channel,

which describes the transparency of a pixel. Blending is a process where final pixel color is computed from the newly computed color (source color) and the previously stored color (destination color) with some kind of **blending operator**. After the merging state the frame buffer is ready to be displayed.

1.4.5 Render-to-texture feature

Shader Model 3 GPUs introduced an important feature essential for general computing called **render-to-texture**. In the classic case the merging state works on the frame buffer, which will be displayed on the screen. The content of this buffer could be copied to a texture to be reused in later renderings. As this memory moving has a significant cost, newer GPUs can redirect the final output from the frame buffer to a given texture. The frame buffer or the texture memory, which is written by fragment shaders is called **render target**. As these targets are no longer associated with the display device, it also became possible to output values simultaneously to several textures, allowing more output data information. This feature is called **multiple render targets**. This feature also extended the number of render output formats that can be used, allowing signed and floating point representations for texture render targets, which opened the possibility to precisely store general information.

1.5 Using GPUs for global illumination

Unlike local illumination, **global illumination** methods[Cso05, Ant04] do not take drastic simplifications, and aim at solving the rendering equation in its original form. By accurately simulating real world physics, we expect the results to be more realistic.

Let us consider equation 1.2, which expresses the reflected radiance of an arbitrary point $\vec{x}$ as a function of the radiance values of illuminating points $\vec{y}$. Radiance $L(\vec{y}, \vec{\omega}')$ at illuminating point $\vec{y}$ is not known, but we can express it using the rendering equation inserting $\vec{y}$ into $\vec{x}$.

Repeating the same step, the radiance caused by single, double, triple, etc. reflections can be obtained. It means that to consider not only single bounce but also multiple bounce light paths, the integral of the rendering equation should be recursively evaluated. From mathematical point of view, global illumination rendering means the solution of the rendering equation, or alternatively, the evaluation of the sequence of high-dimensional integrals for the representative wavelengths.

When the scene is rendered by the GPU, the data passed to shaders are still only describing local geometry and materials, or global constants, but nothing about other pieces of geometry. Primitives and points are processed independently, dependence is introduced only at the last merging step by depth testing and alpha blending. This still allows only local illumination. However, when a point is shaded with a global illumination algorithm, its radiance will be the function of all other points in the scene. From a programming point of view this means that we need to access the complete scene description when shading a point. While this is granted in CPU based ray tracing systems, the stream processing architecture of current GPUs fundamentally contradicts this requirement. When a point is shaded on the GPU we have just its limited amount of local properties stored in registers, and may access texture data. Thus the required global properties of the scene must be stored in uniform registers — which have a limited size and can be refreshed only by the CPU —, or more preferably in **textures**.

If the textures must be static or they must be computed on the CPU, then the illumination computation is not making use of the processing power of the parallel hardware, and the graphics card merely presents CPU results. Textures themselves have to be computed on the GPU. The render-to-texture feature allows this: anything that can be rendered to the screen, may be stored in a texture. Such texture render targets may also require depth and stencil buffers. Along with programmability, various kinds of data may be computed to textures. They may also be stored in floating point format in the texture memory.

To use textures generated by the GPU, the rendering process must be decomposed to **passes**, where one pass may render into a texture and another may use the generated textures. Since the reflected radiance also depends on geometric properties, these textures usually contain not only conventional color data, but they also encode geometry and prepared, reusable illumination information as well.

Programmability and render-to-texture together make it possible to create some kind of processed representation of geometry and illumination as textures. The different passes transferring these information to each other are the keys to the addressing the self-dependency of the global illumination rendering problem.

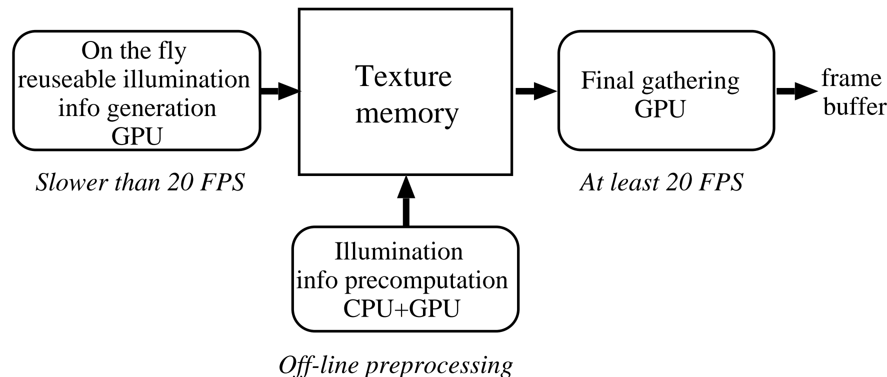


Figure 1.2: Structure of the GPU implementation of global illumination algorithms

An algorithm simulating a specific global illumination effect can be divided into two major tasks. The first is an **illumination information generation** step, in which we create some kind of information about the scene or about a part of it — like the geometry of the surrounding environment, or some kind of illumination information like incoming radiance, photon hit position, ray parameters etc. — and store it as a texture. This information will be used in the second main step, called the **final gathering step**. The final gathering step is the rendering of an object to the frame buffer using a specific shader that uses the resources created in the illumination info generation step to break the restriction of local illumination (see Figure 1.2).

While the final gathering step is executed in each frame, the illumination info generation step should be executed only if needed. This update interval can vary according to the nature of the given output **resource** (i.e. a texture or some simple variable). It can be updated in each frame, only once before the first frame (can be implemented in a separate application and run off-line), only if some conditions change (e.g. view position, view direction, object or light position), or in an interval of a few frames defined by performance tuning.

1.6 Structure and objectives of the thesis work

This thesis work first summarizes the previous work in related fields, then presents the new results in three parts. The three main parts of this work introduce methods that use sampled geometry to achieve realistic illumination effects in case of common rendering problems: diffuse and specular material rendering, and participating media rendering.

1.6.1 Diffuse material rendering

Indirect illumination is usually handled only in case of diffuse materials when the reflected radiance is independent of the outgoing direction. This reduces the dimensionality of the light transport operator making illumination pre-calculation easier. Several different methods have been developed in real-time rendering that approximate indirect illumination, which would need some kind of global illumination algorithm to be calculated exactly.

The first part introduces an improved **environment map** based diffuse indirect illumination algorithm, which provides more robust and accurate results than previous approaches [J8]. This is followed by the presentation of a new **screen space ambient occlusion** method that efficiently reduces the noise of this sampling based technique by rewriting the **ambient occlusion** formula and introducing a new integration formula [F4].

1.6.2 Specular material rendering

The best way to display ideal reflections or refractions and caustics is to use a ray-tracing algorithm, which remains an off-line solution due to its complexity. However in real-time graphics new algorithms have been developed to render similar effects. Of course, these methods have some limitations compared to ray-tracing. Self reflection may be missing, visible artifacts may appear, or caustics may have poor quality.

The second part introduces a new method to render **multiple reflections** and refractions in real-time enabling self reflections [D2]. Results can even be compared with classical ray-traced solutions. A new method for caustic rendering is also presented, which replaces photon hit splatting with caustic triangle rendering, which provides higher quality and requires less user control [J5, J9].

1.6.3 Participating media rendering

Participating media is often represented by particle systems. Particles are usually drawn as screen aligned small planes called **billboards** . As these billboards are two-dimensional objects placing them in a three-dimensional environment causes several visual problems like **clipping** and **popping artifacts**. The exact illumination calculation of participating media is also a challenging task, most particle rendering methods usually discard the scattering and light absorbtion properties of the media.

The last part introduces methods to eliminate billboard rendering artifacts [J4, D1, J3, J2], enable efficient rendering of high number of particles [J2], and to approximate light absorbtion and scattering even under changing lighting conditions [J2, J1].

Chapter 2

Previous work

GPUs has a limited number of triangles they can render in each frame maintaining real-time performance. To overcome this problem several methods — that are based on representing geometry information in textures — have been developed to reduce triangle count while trying to preserve geometric detail.

On the other hand, as the GPU processes triangles, vertices, and fragments independently of each other, the only possibility to introduce self dependency of global illumination in a GPU environment is the application of textures. Methods that use special textures to improve local illumination have been used for several years in CPU computer graphics even before programable graphics hardware showed up. The next subsections cover the most important classical techniques as well as their improvements used on the GPU.

As any extra information is stored in textures and during rendering we usually address these textures with the **texture coordinates** of the actual geometry, this mapping can play an essential role. In many cases geometry properties are sampled at their positions in **texture space**. Creating the texture coordinates of a triangle mesh means unfolding it to a plane. This can be a simple planar or spherical projection, but in many cases this mapping should meet some requirements. The most common requirements are that the geometry should be mapped to a unit square, and each triangle should have a unique mapping i.e. triangles should not overlap each other in texture space. A texture storing some information with this special mapping is usually called an **UV atlas**.

Geometry images [GGH02] use a special mapping, where no pixel in the texture atlas is wasted (the geometry is seamlessly and perfectly flattened to a texture). All the three model space position and normal coordinates of the original surface are stored in separate atlases, which can be used later to reconstruct the geometry. For three adjacent pixels of the atlases, a triangle is drawn whose vertex positions and normals are fetched from the position and normal textures, respectively. The main advantage of this technique is that highly detailed geometry can be efficiently stored and their triangle count can be hierarchically reduced, as for these textures existing image compression techniques and mipmapping can be used.

On the other hand, if the texture coordinate pair is also directly utilized to identify the represented point, storing three components for positions is not needed, a single additional coordinate is enough. The texture containing this third coordinate can store distances from a reference surface, when — according to its actual use — it is called a **height map** or **depth map**, or from a **reference point**, when it is called a **distance map**. Methods described next all exploit this compactness.

This chapter summarizes the most relevant previous work and discusses **displacement mapping**, **impostors**, **distance map** based indirect illumination approaches, and finally the **particle system** representation of participating media.

2.1 Displacement mapping

Displacement mapping decomposes the definition of the surface to a rough **macrostructure geometry** — which is a low resolution triangle mesh — and to details, which describe the difference between the rough geometry and the original surface. The vertex shader transforms only the macrostructure geometry, and the details are taken into account only when fragments are processed, that is, when color texturing takes place. However, at this stage it is too late to change the geometry, the point visible through the pixel is already defined. But as the vertex shader processed only an approximate geometry, this surface point will also be incorrect. Thus the visibility problem needs to be corrected in the fragment shader program by a **ray-tracing** like algorithm. Vertex normals are also describing only the rough mesh, so normals needed for proper lighting should also be corrected to match the bumps of the detailed surface.

The original surface normals and the displacements are stored in an UV atlases, called **normal map** and **height map**.

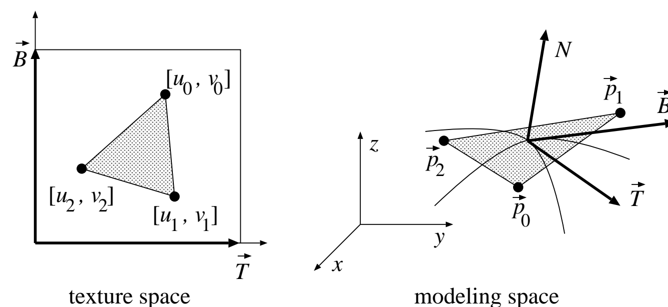


Figure 2.1: Tangent space

This information is defined in **tangent space**, which is a coordinate system attached to the surface. Its basis vectors are **normal** $\vec{N}$, **tangent** $\vec{T}$, and **binormal** $\vec{B}$ which are defined for each vertex (Figure 2.1). Normal points in the direction of the triangle normal, tangent and binormal point to the the directions where the first and second texture coordinates increase respectively. Vectors defined in this space are always relative to the triangle plane, making their definition independent of triangle orientation. Vectors can be transformed from tangent space to model space (or back) with a simple 3×3 matrix multiplication.

The **height map** stores the distance of the detailed surface and the approximating geometry along the triangle normal. Visible surface detection can be imagined as tracing rays into the height field to obtain the texture coordinates of the visible point (Figure 2.2), which are used to fetch color and normal vector data.

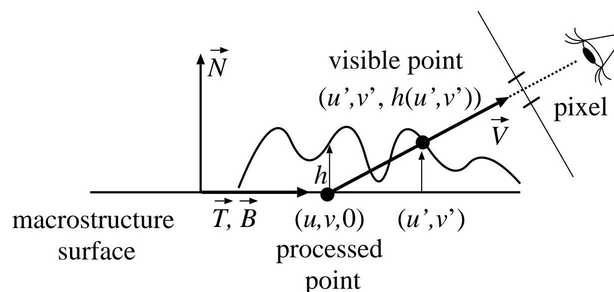


Figure 2.2: Ray tracing of the height field

Height function h is evaluated by fetching the texture memory. Let us define the ray by the processed point (u, v) and by another tangent space point where the ray intersects the maximum height plane. This point may be called **entry point** and is given as coordinates $(u_{in}, v_{in}, 1)$ in tangent space since the ray enters here the volume of possible height field intersections. Similarly,

the tangent space processed point, $(u, v, 0)$, on the other hand, can be considered as the **exit point**. With the entry and exit points the ray segment between the minimum and maximum height planes is

$$(u, v, 0)(1 - H) + (u_{in}, v_{in}, 1)H, \quad H \in [0, 1]. \tag{2.1}$$

In this representation height value H directly plays the role of the ray parameter. The equation to be solved is

$$(u', v') = (u, v)(1 - H) + (u_{in}, v_{in})H, \quad h(u', v') = H.$$

In the following the most important displacement mapping techniques are reviewed. These methods use different strategies to solve the height field ray-tracing problem.

2.1.1 Parallax mapping

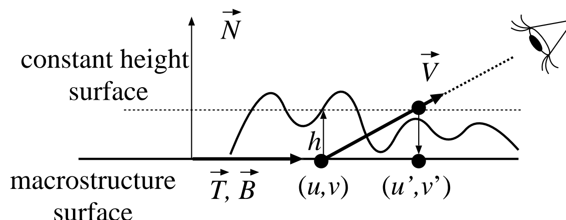


Figure 2.3: Parallax mapping

The texture coordinates are modified assuming that the height field is constant $h(u, v)$ everywhere in the neighborhood of (u, v). As can be seen in Figure 2.3, the original (u, v) texture coordinates get substituted by (u', v'), which are calculated from the direction of tangent space view vector $\vec{V} = (V_x, V_y, V_z)$ and height value $h(u, v)$ read from a texture at point (u, v). The assumption on a constant height surface simplifies the ray equation to

$$(u', v', h(u, v)) = (u, v, 0) + \vec{V}t,$$

which has the following solution:

$$(u', v') = (u, v) + h(u, v)\left(\frac{V_x}{V_z}, \frac{V_y}{V_z}\right).$$

Parallax mapping in its original form has a significant flaw. As the viewing angle becomes more grazing, offset values approach infinity. When offset values become large, the odds of (u', v') indexing a similar height to that of (u, v) fade away, and the result seems to be random.

A simple way to solve the problem of parallax mapping at grazing angles is to limit the offsets so that they never get larger than the height at (u, v) [Wel04].

Parallax mapping assumes that the surface is a constant height plane. A better approximation can be obtained if we assume that the surface is still planar, but its normal vector can be arbitrary (i.e. this surface is not necessarily parallel with the macrostructure surface). The normal of the approximating plane can be taken as the normal vector read from the normal map, thus this approach does not require any further texture lookups [MM05].

Parallax mapping makes an attempt to offset the texture coordinates toward the really seen height field point. Of course, with a single attempt perfect results cannot be expected. The accuracy of the solution, however, can be improved by repeating the correction step by a few (say 3–4) times [Pre06].

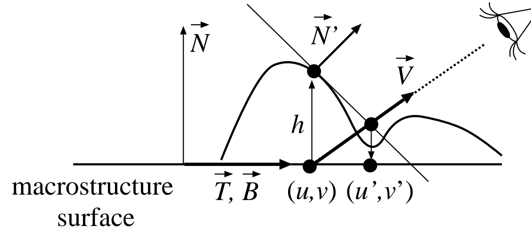


Figure 2.4: Parallax mapping taking into account the slope

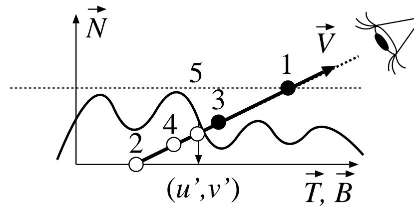


Figure 2.5: Binary search that stops at point 5

2.1.2 Binary search

Suppose we have two guesses on the ray that enclose the real intersection point since one guess is above while the other is below the height field. Points on the minimum and maximum height values, i.e. points defined by height parameters $H_{\min} = 0$ and $H_{\max} = 1$, surely meet this requirement.

Binary search halves the interval $(H_{\min}, H_{\max})$ containing the intersection in each iteration step putting the next guess at the middle of the current interval [POC05, PO05]. Comparing the height of this guess and the height field, we can decide whether or not the middle point is below the surface. Then we keep that half interval where one endpoint is above while the other is below the height field.

The binary search procedure quickly converges to an intersection but may not result in the first intersection that has the maximum height value.

2.1.3 Secant search

Binary search simply halves the interval potentially containing the intersection point without taking into account the underlying geometry. The **secant method** [YJ04, RSP06], on the other hand, assumes that the surface is planar between the two guesses and computes the intersection between the planar surface and the ray. It means that if the surface were really a plane between the first two candidate points, this intersection point could be obtained at once. The height field is checked at the intersection point and one endpoint of the current interval is replaced keeping always a pair of end points where one end is above while the other is below the height field. As has been pointed out in [SKALP05] the name “secant” is not precise from mathematical point of view since the secant method in mathematics always keeps the last two guesses and does not care of having a pair of points that enclose the intersection. Mathematically the root finding scheme used in displacement mapping is equivalent to the **false position method**. We note that it would be worth checking the real secant algorithm as well, which almost always converges faster than the false position method, but unfortunately, its convergence is not guaranteed in all cases.

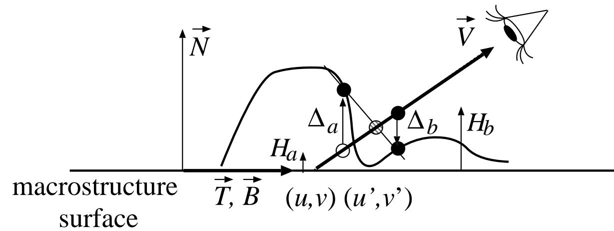


Figure 2.6: The secant method

2.1.4 Linear search

Linear search, i.e. **ray-marching** finds a pair of points on the ray that enclose the possibly first intersection, taking steps of the same length on the ray between the entry and exit points (Figure 2.7). Ray marching is an old method of rendering 3D volumetric data [Lev90].

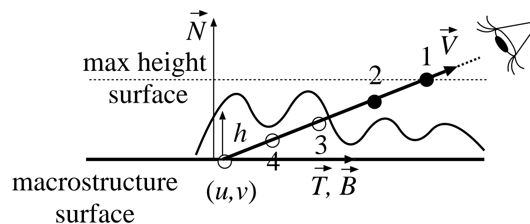


Figure 2.7: Linear search that stops at point 3

Using linear search alone results in stair-stepping artifacts unless the steps are very fine. To make linear search really safe, each texel needs to be visited, which is slow if the texture resolution is high. Linear search was used in **steep parallax mapping** [MM05].

2.1.5 Relief mapping

Combined iterative methods combine a safe or quasi-safe technique to provide robustness and an unsafe method that is responsible for finding the intersection quickly. The safe method should only find a rough approximation of the first intersection and should make sure that from this approximation the unsafe method cannot fail.

This combined approach was used in **relief mapping** [OBM00, Oli00, POC05, PO05], which uses a two phase root-finding approach to locate the intersection of the height field and the ray. The first phase is a linear search, i.e. ray-marching, which finds a pair of points on the ray that enclose the possibly first intersection. The second phase refines these approximations by a binary search.

Ray marching can also be improved combined with a series of geometric intersections, according to the **secant method**. The first pass of these algorithms is also a linear search. This kind of combined approach was first proposed by Yerex [YJ04] and independently in **parallax occlusion mapping** [BT04, Tat06a, Tat06b], and showed up in **interval mapping** [RSP06] too.

Figure 2.8 compares the different bump mapping and displacement mapping techniques.

2.2 Impostors

A classical use of textures to represent geometry are impostors. These are previously prepared semi-transparent images of complex objects stretched onto a rectangular object placed in the 3D space which replaces the original geometry. In order to avoid the shortening of the visible image when the user looks at it from grazing angles, the impostor plane is always turned towards the

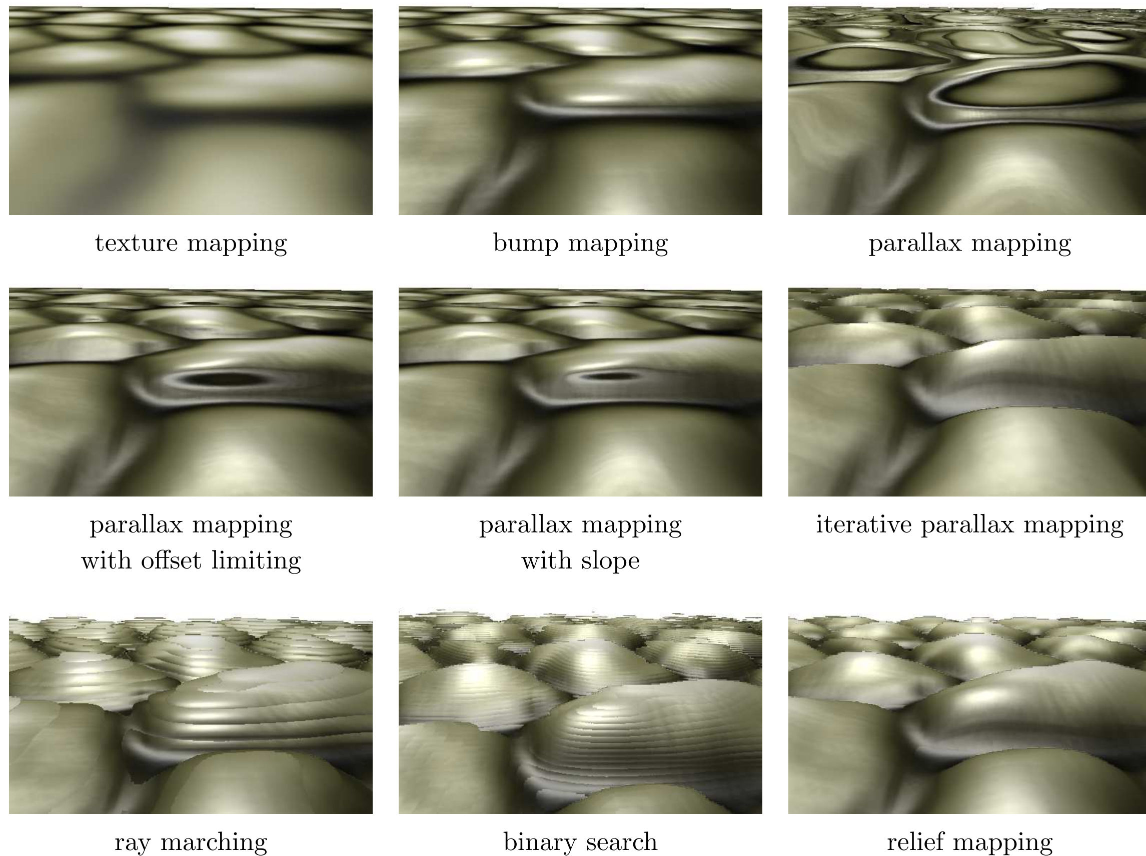


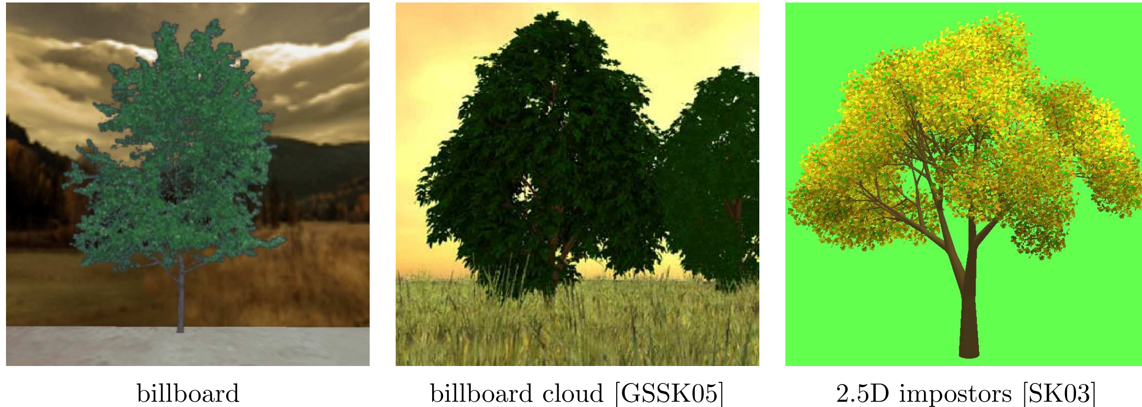
Figure 2.8: Comparison of mapping techniques assuming normal displacement sizes

camera. However the object always looks the same no matter from where we look at it. This missing parallax effect makes the replacement too easy to recognize. In order to handle this problem, we can pre-render a finite set of images from a few views, and present the one closest in alignment with the actual viewing direction, or we can re-render the impostor image at a desired frequency. This last method is called **dynamically generated impostors**, and is based on the observation that image data can be re-used as long as the geometric and photometric errors remain below a given threshold. Dynamically generated impostors distribute the problem of rendering complex geometry between several frames to increase average performance. The main problem of impostor based techniques is that they do not have real three-dimensional extents, which makes correct occlusion handling problematic.

Billboard clouds [DDSD03] also use several images to replace a complex 3D object. A billboard cloud is built by choosing a set of planes that capture well the geometry of the input model, and by projecting the triangles onto these planes to compute the texture and transparency maps associated with each plane of the billboard. Unlike billboards, these impostors are not rotated when the camera moves, thus the expected parallax effects are provided. Billboard clouds are displayed by rendering each polygon independently. The key of the method is how the planes are defined, and how the triangles are clustered and projected onto the planes. Billboard clouds can be used efficiently for smooth surfaces, complex, combined models, and even for crown of trees [GSSK05].

In case of impostors an extra depth information can avoid occlusion artifacts as the actual camera transformation and the position of the impostor plane is enough to reconstruct the points of the sampled object. [Sch97] introduced **nailboards** that are used like classic impostor billboards, but they also contained distance information from the billboard plane, which solved the occlusion problem. This idea was used in [SK03] under a name **2.5D impostors**, which

was a complete GPU implemented version. It was successfully used to render trees with high number of leaves. The crone was divided into blocks of leaves, and each block used the same dynamically generated impostor with depth information. The technique was successfully applied to handle occlusions between tree branches and leaf block impostors giving the illusion as each leaf had its own geometry. [PMDS06] used depth complemented impostors to calculate accurate multiple reflections of complex objects. The billboard cloud method was also extended with depth information [MJW07].



billboard | billboard cloud [GSSK05] | 2.5D impostors [SK03]

Figure 2.9: Comparison of different impostor techniques used for tree rendering.

An other clever use of textures with geometry information is **deferred shading** [HH04, BL08], which was developed to overcome the problem of lighting a scene with many light sources. The main concept behind deferred shading is to gather all information of the visible surface points that are needed by lighting. Deferred shading transforms geometry only once and performs lighting calculation for each light source separately and combines their results. To do this first geometry information is captured from the camera: a so called **fat buffer** or **geometry buffer** is stored. This buffer is a collection of textures containing usually position, normal, color and texture coordinate information of surface points visible from the camera. During lighting, no geometry is processed only the fat buffer is used to reconstruct the necessary information in each pixel. Each lighting pass computes the contribution of one light source and the result of these passes are added together. Deferred shading overcomes the problem of limited instruction count that can be used by shaders, and can greatly increase performance for complex scenes as shading is performed only for finally visible points that have passed depth test after the entire scene is drawn.

2.3 Sampled representation of indirect illumination

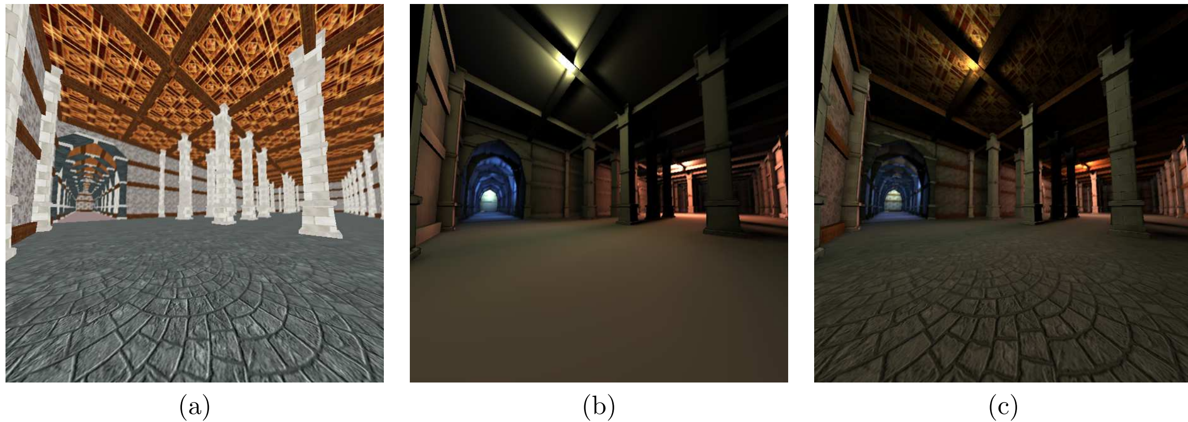
As mentioned before in Section 1.5, textures are our only tools to resolve the self dependency of global illumination in a GPU environment. Methods that involve special textures to improve local illumination have been used for several years in CPU computer graphics even before programable graphics hardware showed up. The next subsections cover the most important classical techniques as well as their improvements used on the GPU.

2.3.1 Light maps and shadow maps

Light maps store incoming radiance values for surface points. The surface is sampled in its texture space so radiance values are stored in an UV atlas called **light map**. Light maps are usually computed once in a pre-processing step, so to determine incoming radiance even ray-tracing or any kind of global illumination software can be used. As all lighting calculation is done off-line a single texture read is enough for a shaded surface point which makes the

method extremely fast, but this also gives its main disadvantage namely it is suitable only for static scenes under static lighting conditions, and supports only materials with view independent BRDF models (diffuse objects only).

Light maps are still widely used in games on static geometry because of their effectiveness (Figure 2.10). During the past few years several extensions have been developed to encode the directionality of incoming radiance allowing more complex BRDF models and normal mapping [MMG06]. Methods to support changing lighting conditions have also been developed [LLM06], of course all these improvements go at the expense of performance.



(a) (b) (c)

Figure 2.10: Use of light maps: surface color (a), incoming radiance read from a light map (b), final image using color and radiance information(c).

The most effective way to enhance the impression of real world lighting is to simply add shadows to the local illumination model. Global illumination and ray tracing (and also light mapping) techniques implicitly contain shadowing (and even soft shadowing) features. Shadow generation in an incremental rendering pipeline however needs special algorithms. The most widely used shadowing technique is **shadow mapping** [Wil78], which also uses geometry information stored in textures.

Shadows occur when an object called **shadow caster** occludes the light source from another object, called **shadow receiver**, thus prevents the light source from illuminating the shadow receiver. To detect occluding geometry, the shadow mapping technique uses a depth image called **shadow map**. This depth image is taken from a camera centered at the light source, looking at the light source direction. Thus, the shadow map is a sampled representation of shadow caster geometry where sampled depth values are distances from the light camera plane. During shadow computation shaded surface points (which are visible from the avatar's camera) are projected back onto the light camera plane and their distances are compared to the previously stored distance values. Shadowing occurs if the stored distance is smaller than the distance of the shaded point meaning that an occluder surface point exist between the light source and the shaded point.

Shadow maps are easy to generate, and can be effectively used even with special filtering techniques required for eliminating aliasing artifacts and to mimic soft shadowing effects (Figure 2.11). As shadow mapping works for moving light sources and objects, it is often used in real-time applications for dynamic geometry. We should note that each light source requires its own shadow map, thus the number of lights that can be present is limited. Shadow mapping techniques also completely neglect multiple light bounces, thus indirect illumination is not supported. Shadow maps and light maps are often used simultaneously, calculating the shadows of dynamic objects with shadow mapping but for the static scene global illumination, information is fetched from light maps.

Figure 2.11: Soft shadows of dynamic objects with shadow mapping [J6]

2.3.2 Environment mapping and its generalization with geometry information

The effects of dynamic objects on static geometry is usually handled with shadow mapping. This technique can also be used to compute the shadows cast by static geometry on dynamic objects. However these effects do not take multiple light bounces into account, which is essential for reflections and refractions but also greatly increases rendering quality in case of diffuse and glossy objects.

The idea of approximating the indirect illumination by a finite set of **virtual lights** was born in the context of global illumination algorithms, such as in **instant radiosity** [Kel97], and has been used in **Monte Carlo** algorithms [War94, SWZ96, WBS03, WFA$^+$05].

For the computation of diffuse interreflections on the GPU, Dachsbacher [DS05] considered shadow map lexels as virtual lights, while Lazányi [SKL06] assigned virtual lights to texels of an environment map. Indeed, if only two-bounce indirect illumination is considered, a shadow map lexel identifies the point which is directly illuminated by the light source. Such points may indirectly illuminate other points, so can be considered as virtual lights.

Virtual light source algorithms suppose that virtual lights are point sources, and thus use the point-to-point form factor (also called geometry factor) when the irradiance is computed. Unfortunately, point-to-point form factor is numerically unstable since it goes to infinity when the virtual light gets close to the shaded point, which results in bright spikes making the position of the virtual lights clearly visible (see figures 3.3 and 3.4).

To address the numerical instability, in [SKL06] virtual lights were supposed to be small disks, which made the geometry factor bounded. However, this approximation is still quite far from being precise when the shaded point is close to the virtual light and the shaded point is not in the normal direction from the virtual light source.

The indirect illumination from static environment to dynamic objects is often handled by **environment mapping**. Similar to light maps, the environment mapping technique stores incoming illumination to a texture, but sampling is not performed on surface points but on incoming directions. As a pre-rendering step we capture the color information of surrounding static environment to an appropriate texture called the **environment map**. A natural way of storing the direction dependent information is an angular mapped texture, but in practice a more GPU friendly possibility is to parameterize the directional space as sides of a cube centered

at the origin of the dynamic object. Current GPUs have a built in support to group six images of the six faces of a cube together and address them with a 3 dimensional direction vector. This data structure is called a **cube map**.

When rendering ideally refractive or reflective dynamic objects the environment map is looked up from a reflection or refraction direction respectively. However to support glossy and diffuse materials many lookups should be made as their BRDF is not a Dirac-delta like in the case of ideally reflections and refractions. These costly lookups can be saved if the environment map is pre-convolved with the actual BRDF [RH01]. However this is a costly process which can only be done in a pre-processing step, making the dynamic refreshment of environment maps impossible.

Environment mapping is a very effective technique to compute specular effects, but it has a main drawback. When storing incoming radiance we sample only a directional domain and do not take surface locations into account, i.e. we assume that the dynamic objects is point like, or in other worlds the environment is considered to be infinitely far away. This assumption usually does not hold, which raises the error for surface points away from the cube map center and closer to the surrounding environment.

One possible solution is to use multiple environment maps [GSHG98, ZHL$^+$05], which can be compressed using **spherical harmonics** [RH01, KAMJ05] or **wavelets** [ZHL$^+$05].

Another solution takes the geometry of the environment and the dynamic object into account. As the position of the shaded point is available only the environment geometry should be stored. This can be done with a **distance environment map** or **distance impostor** [SKALP05], which is generated just like a classical cube map, but it stores distance values from the cube map center in addition to the incoming radiance.

Section 2.1 showed techniques that performed ray-tracing in height fields with various searching algorithms. This concept can be reused in case of distance environment maps. With special searching methods intersection with the environment can be computed for light rays with arbitrary start position and direction. With this ray-tracing capability in our hands environment map based reflections and refractions will be much more accurate (see Figure **??**).

In the remaining part of this section the ray-tracing algorithm in a single distance map layer is discussed, using the notations of Figure 2.12. Let us assume that center $\vec{o}$ of our coordinate system is the reference point and we are interested in the illumination of point $\vec{x}$ from direction $\vec{R}$.

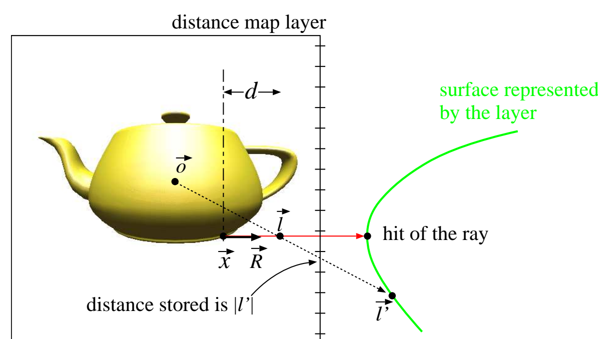


Figure 2.12: Tracing a ray from reflection point $\vec{x}$ at direction $\vec{R}$. When we have a hit point approximation $\vec{l}$ on the ray, the distance and the radiance of point $\vec{l'}$ will be fetched from the cube map of reference point $\vec{o}$.

The illuminating point is thus on the ray of equation $\vec{x} + \vec{R}d$, where d is the ray parameter. The accuracy of an arbitrary approximation d can be checked by reading the distance of the environment surface stored with the direction of $\vec{l} = \vec{x} + \vec{R}d$ in the cube map ($|\vec{l'}|$) and comparing

it with the distance of approximating point $\vec{l}$ on the ray ($|\vec{l}|$). If $|\vec{l}| \approx |\vec{l}'|$, then we have found the intersection. If the point on the ray is in front of the surface, that is $|\vec{l}| < |\vec{l}'|$, the current approximation is an **undershooting**. On the other hand, the case when point $\vec{l}$ is behind the surface ($|\vec{l}| > |\vec{l}'|$) is called **overshooting**. Ray parameter d of the ray hit can be found by a simple approximation or by an iterative process.

Parallax correction

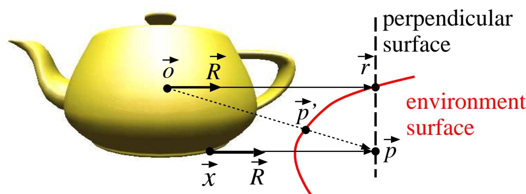


Figure 2.13: Parallax correction assumes that the surface is perpendicular to the ray and replaces approximation $\vec{r}$ of the environment mapping by the intersection of the ray of origin $\vec{x}$ and direction $\vec{R}$ with this perpendicular surface.

Classical environment mapping would look up the illumination selected by direction $\vec{R}$, that is, it would use the radiance of point $\vec{r}$ (Figure 2.13). This can be considered as the first guess for the ray hit. To find a better second guess, we assume that the environment surface at $\vec{r}$ is perpendicular to ray direction $\vec{R}$. In case of perpendicular surface, the ray would hit point $\vec{p}$ with ray parameter d_p:

$$d_p = |\vec{r}| - \vec{R} \cdot \vec{x}. \tag{2.2}$$

If we used the direction of point $\vec{p}$ to lookup the environment map, we would obtain the radiance of point $\vec{p}'$, which is in the direction of $\vec{p}$ but is on the surface. If the accuracy is not sufficient, the same step can be repeated, resulting in an iterative process [Wym05, WD06b, SP07].

Linear search

The possible intersection points are on the half-line of the ray, thus the intersection can be found by marching on the ray, i.e. checking points $\vec{l} = \vec{x} + \vec{R}d$ generated with an increasing sequence of parameter d and detecting the first pair of subsequent points where one point is an overshooting while the other is an undershooting [Pat95].

The complete search can be implemented by drawing a line and letting the fragment program check just one texel [KBW06]. However, in this case it is problematic to find the first intersection from the multiple intersections since fragments are processed independently. In their implementation, Krüger et al. solved this problem by rendering each ray into a separate row of a texture, and found the first hits by additional texture processing passes, which complicates the algorithm, reduces the flexibility, and prohibits early ray termination.

Secant search

Secant search can be started when there are already two guesses of the intersection point, provided, for example, by a linear search [D2], by taking the start and the end of the ray [SKAL05], or by pairing the end of the ray with the result of the parallax correction [SKALP05]. Let us denote the ray parameters of the two guesses by d_p and d_l, respectively. The corresponding two points on the ray are $\vec{p}$ and $\vec{l}$, and the two points on the surface are $\vec{p}'$ and $\vec{l}'$, respectively (Figure 2.14).

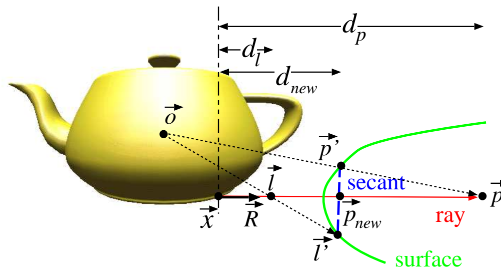


Figure 2.14: Refinement by a secant step. The new approximation $\vec{p}_{new}$ is the intersection of the ray and the line segment of overshooting approximation $\vec{p}'$ and undershooting approximation $\vec{l}'$.

Computing the ray parameter at the intersection of the ray and line segment $\vec{p}'$ and $\vec{l}'$, we obtain:

$$d_{new} = d_l + (d_p - d_l)\frac{1 - |\vec{l}|/|\vec{l}'|}{|\vec{p}|/|\vec{p}'| - |\vec{l}|/|\vec{l}'|}. \tag{2.3}$$

The point specified by this new ray parameter gets closer to the real intersection point. If a single secant step does not provide accurate enough results, then d_{new} can replace one of the previous approximations d_l or d_p, and we can proceed with the same iteration step. If we keep always one overshooting and one undershooting approximations, the method is equivalent to the false position root finding algorithm.

Binary search

Hu et al. [HQ07] and Oliveira et al. [OB07] used binary search steps to refine the approximations obtained by a linear search similarly to popular displacement mapping algorithms. Binary search simply halves the interval of the ray parameter:

$$d_{new} = \frac{d_l + d_p}{2}.$$

Since it does not use as much information as the secant search its convergence is slower.

Single reflections or refractions using the sampled geometry

In order to render a scene with an object specularly reflecting its environment, we need to generate the depth or distance map of the environment of the specular object. This requires the rendering of all objects but the reflective object six times from the reference point, which is put close to the center of the reflective object. Then non-specular objects are rendered from the camera position in a normal way. Finally, the specular object is sent through the pipeline, setting the fragment shader to compute the reflected ray, and to approximate the hit point as a cube map texel. Having identified the texel corresponding to the hit point, its radiance is read from the cube map and is weighted by the **Fresnel** function. To simulate refracted rays, just the direction computation should be changed from the law of reflection to the **Snellius-Descartes law** of refraction.

Inter-object multiple reflections and refractions

Cube map based methods computing single reflections can straightforwardly be used to obtain multiple specular inter-reflections of different objects if each of them has its own cube map [NC02]. Suppose that the cube maps of specular objects are generated one by one. When the cube map of a particular object is generated, other objects are rendered with their own shader

programs. A diffuse object is rendered with the reflected radiance of the direct light sources, and a specular object is processed by a fragment shader that looks up its cube map in the direction of the hit point of the reflection (or refraction) ray. When the first object is processed the cube maps of other objects are not yet initialized, so the cube map of the first object will be valid only where diffuse surfaces are visible. However, during the generation of the cube map for the second reflective object, the color reflected off the first object is already available, thus **diffuse surface – first reflective object – second reflective object** paths are correctly generated. At the end of the first round of the cube map generation process a later generated cube map will contain the reflection of other reflectors processed earlier, but not vice versa. Repeating the cube map generation process again, all cube maps will store double reflections and later rendered cube maps also represent triple reflections of earlier processed objects. Cube map generation cycles should be repeated until the required reflection depth is reached.

If we have a dynamic scene when cube maps are periodically re-generated anyway, the calculation of higher order reflections is not more expensive computationally than rendering single reflections. In each frame cube maps are updated using other objects' cube maps independently of whether they have already been refreshed in this frame or only in the previous frame (Figure 2.15). The reflection of a reflected image might come from the previous frame — i.e. the latency for degree n inter-reflection will be n frames — but this delay is not noticeable at interactive frame rates.

Figure 2.15: Inter-object reflections in a car game. Note the reflection of the reflective car on the beer bottles, and vice versa [J6].

2.3.3 Caustic mapping

Caustics show up as high frequency patterns on diffuse or glossy surfaces, formed by light paths originating at light sources and visiting mirrors or refracting surfaces. A caustic is the concentration of light [Jen01, TS00, WS03].

Light paths starting at the light sources and visiting specular reflectors and refractors until they arrive at diffuse surfaces need to be simulated to create caustic effects. Theoretically, such paths can be built starting the path at the light source and following the direction of the light (light or **photon tracing**), or starting at the receiver surfaces and going opposite to the normal light (visibility ray tracing). If light sources are small, then the probability that visibility ray tracing finds them is negligible, thus visibility ray tracing is inefficient to render general caustics.

General and effective caustic generation algorithms have two phases [Arv86], where the first phase identifies the terminal hits of light paths using some kind of photon ray tracing, and the second projects caustic patterns onto the receiver surfaces.

In GPU based caustic algorithms, the photon tracing part requires the implementation of some ray-tracing algorithm on the GPU. One effective solution to GPU ray tracing was introduced in Section 2.3.2. This environment map based technique suits very well to the needs of the photon tracing part of caustic generation.

In the first phase, putting the view plane between the light and the refractor (Figure 6.2), the scene is rendered from the point of view of the light source, and the terminal hits of caustic paths are determined. The location of the terminal hits are stored in pixels of the render target called the **photon hit location image**.

From discrete photon hits a continuous caustic pattern needs to be reconstructed and projected onto the **caustic receiver** surfaces. During reconstruction a photon hit should affect not only a surface point, but also a surface neighborhood where the power of the photon is distributed. A neighborhood consists of points that are in the same direction from the caustic generator object and are all visible from the **caustic generator**. In order to eliminate light leaks, this neighborhood information should be preserved in the space where photon hits are stored and caustics are reconstructed.

There are several alternatives for spaces to represent the location of a photon hit, having different advantages and disadvantages:

3D grid [PDC+03]: Similarly to classical **photon mapping** photon hits can be stored independently of the surfaces in a regular or adaptive 3D grid. When the irradiance of a point is needed, hits that are close to the point are obtained. If the surface normal is also stored with the photon hits, and only those hits are taken into account which have similar normal vectors as the considered surface point, then light leaks can be minimized and photons arriving at back faces can be ignored. Unfortunately, GPUs are not good in managing adaptive data structures needed by this approach, so such approaches are not effective.

Texture space [GD01, SKALP05, CSKSN05]: Considering that the reflected radiance caused by a photon hit is the product of the BRDF and the power of the photon, and the BRDF is most conveniently fetched according to the texture coordinates, one straightforward possibility to identify a photon hit is the texture coordinates of that surface point which is hit by the ray. A pixel of the photon hit location image stores the two texture coordinates of the hit position and the luminance of the power of the photon. Since the texture space neighborhood of a point visible from the caustic generator may also include occluded points, light leaks might occur.

Screen or image space [LC04, WD06b]: A point in the scene can be identified by the pixel coordinates and the depth when rendered from the point of view of the camera. This screen space location can also be written into the photon hit location image. If photon hits are represented in image space, photons can be splat directly onto the image of the diffuse caustic receivers without additional transformations. However, the BRDF of the surface point cannot be easily looked up with this representation, and we should modulate the rendered color with the caustic light, which is only an approximation. This method is also prone to creating light leaks.

Ray space [IDN02, EAMJ05, KBW06]: Instead of the hit point, the ray after the last specular reflection or refraction can also be stored. When caustic patterns are projected onto the receiver surfaces, the first hit of these rays need to be found to finalize the location of the hit, which is complicated. Thus these methods either ignore visibility [IDN02, EAMJ05] or do not apply filtering [KBW06].

Shadow map space [SP07]: In the coordinate system of the shadow map, where the light source is in the origin, a point is identified by the direction in which it is visible from the

light source. An advantage of this approach is that rendering from the light's point of view is needed by shadow mapping anyway. The drawbacks are the possibility of light leaks and that caustics coming from the outside of the light's frustum are omitted.

Most of the GPU based algorithms reconstruct the continuous caustic pattern from discrete hits with placing semi-transparent quadrilaterals with a filtering texture, this technique is called photon **splatting** [WS03, SKALP05, SP07, WD06b, WD06a].

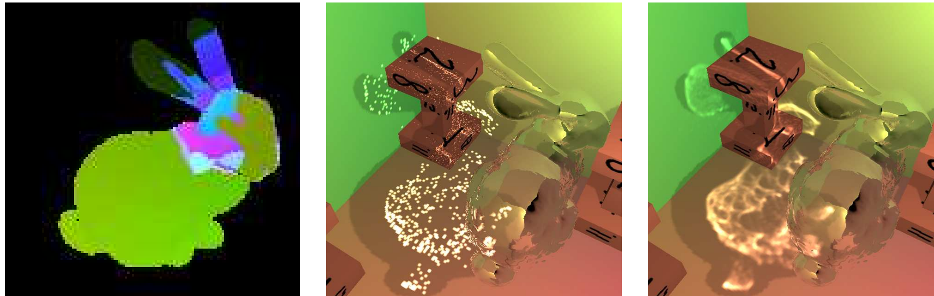


Figure 2.16: A photon map, a room without photon hit positions and a room with splat photons [SKALP05].

2.4 Rendering volumetric media on the GPU

Light scattering is caused by small particles in the air. As we cannot examine all the billion particles one-by-one, some kind of discretization of the continuous medium is needed, which allows us to replace the differentials of the volumetric rendering equation by finite differences. The two most common representations of volumetric media are grids and particle systems.

The most obvious way to discretize a volume is to sample it on a **regular grid**. **Cartesian grids** have the advantage that neighborhood relations are straightforward, they can be stored in a two- or three- dimensional array, and the mapping between an arbitrary point in space and its array indices is easy to calculate.

Besides Cartesian grids several other types of grids can be chosen that may suit our needs better for a given situation. Examples are the **face centered (FCC)** or the **body centered (BCC)** cubic lattices [BB08], which sample the space more evenly at higher dimensions. These grids can also be stored in a two-dimensional array or texture, however, neighborhood relations and the mapping of array indices are not so straightforward.

GPUs have a special texture type, which is a three dimensional array with support for tri-linear interpolation. Scattering media can be uploaded to a **3D texture** and can be rendered efficiently with hardware support [Wei06].

One of the most widely used GPU methods for texture based volume rendering is **texture slicing**. This method renders view aligned polygons which are generated by the CPU and clipped against the bounding box of the volume. Rasterization generates fragments on the polygon slices, producing sampling positions through the volume. These sampling positions are addressed by 3D texture coordinates attached to the vertices of the polygons. This technique can be efficiently implemented even in the fixed function pipeline. Polygons should be rendered in order (e.g. from back to front), z-buffer algorithm should be turned off, and blending should be enabled.

With the evolution of pixel shaders a common technique — that has been used for long in CPU volume visualization — could be reimplemented on the GPU, namely **ray marching**. Ray marching traverses view rays from the camera through each pixel into the volume. For each ray, the volumetric rendering equation is evaluated, with equidistance sampling along the view ray

leading to a Riemann sum approximation. As sample positions usually differ from grid positions a tri-linear sampling filter should be used. Only the front faces of the volume boundary need to be rendered, the sampling is performed by the fragment shader. GPUs that support loops and branching in fragment shaders (Shader Model 3 compliant GPUs) are most suitable for GPU ray marching.

Besides the many advantages of sampling the media on grids it has a main disadvantage, namely it is optimal for media that fills the sampled space evenly. In many cases most of the grid points contain no important information as the scattering media has a free formed shape or fills up only a small part of the sampled space. To capture enough information in these cases, the grid resolution should be heighten, which makes the sampling too memory consuming and wasteful and even unduly raises computation time. However, when neighbor information is needed e.g. in case of fluid simulation or light scattering simulation or in case of distributed computing, grid based techniques can be a good choice [J13, J7, J10, J12].

To overcome the problems of uniform sampling, a different representation can be used, namely **particle systems**. The particle system model of the volume corresponds to an adaptive discretization, when we assume that scattering can happen only at N discrete points called particles. We assume that particles are sampled randomly, preferably from a distribution proportional to collision density τ, and we do not require them to be placed at grid points [GRWS04]. Let us assume that particle p represents its spherical neighborhood of diameter Δs_p, and introduce its **opacity** as $\alpha_p = 1 - e^{-\tau_p \Delta s_p}$, its **emission** as $E_p = L^e \Delta s_p$, its **incoming radiance** by I_p, and its **outgoing radiance** by L_p. The **discretized volumetric rendering equation** at particle p is then:

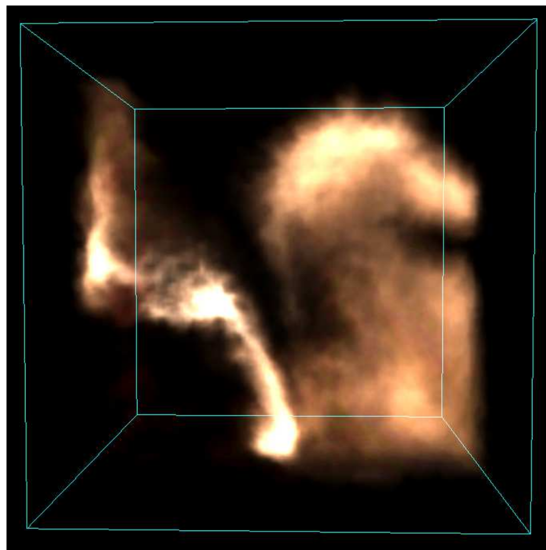
$$L_p(\vec{\omega}) = (1 - \alpha_p) I_p(\vec{\omega}) + E_p(\vec{\omega}) + \alpha_p a_p \int_{\Omega'} I_p(\vec{\omega}') P_p(\vec{\omega}', \vec{\omega}) \mathrm{d}\omega'.$$

In homogeneous media, albedo a and phase function P are the same for all particles. In inhomogeneous media, these parameters are particle attributes [REK$^+$04].

Particles are usually rendered with small screen aligned quadrilaterals called billboards. These quads consist of two triangles and can be treated by graphics hardware just like any other triangle meshes. This rendering method is closely related to impostor rendering described in Section 2.2.

Billboard vertex data should be refreshed in each frame, as their properties change in time. They are also usually sorted according to their distances from the current camera position. Vertices can be rotated towards the camera on the CPU, or with newer hardware more preferably by the vertex shader on the GPU. As they represent volumetric media and not opaque objects alpha blending and special depth testing should be set in the graphics pipeline (see Section 1.4.4). Because their performance and flexibility particle systems are the most commonly used representations of volumetric media (like smoke, fire or clouds) in real-time virtual reality applications.

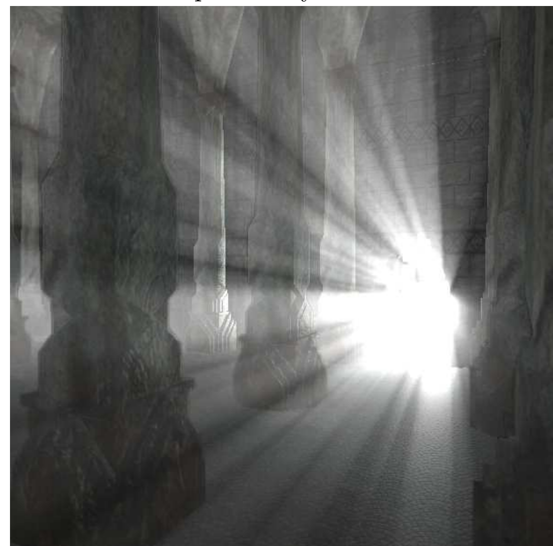
Some special effects like fog or underwater fog do not require complex sampling as they have roughly uniform properties over the entire scene. Their contribution is affected only by the distance from the camera to the shaded point, thus they can be calculated after the run of fragment shader in the merging state. GPUs have a built in support for a fog merging operator. This method simplifies the volumetric rendering equation as it assumes constant inscattering along the view ray (i.e. no objects can occlude incoming light), thus can simulate only absorbtion and forward scattering. To take into account occluding geometry when producing **shafts of light**, complex **merging** operator should be used, which can be implemented as a **post processing** effect [J17, J16].



texture slicing [J10]

particle system

fog



light shafts[J17]

Figure 2.17: GPU volumetric media rendering techniques.

Part I

Diffuse Material Rendering

Chapter 3

Robust diffuse environment mapping

The rendering equation integrates in the domain of input directions Ω'. If a differential surface $\mathrm{d}y$ is visible from illuminated point $\vec{x}$, and the angle between direction $\omega'_{\vec{y}\to\vec{x}}$ from $\vec{y}$ to $\vec{x}$ and the surface normal at $\vec{y}$ is $\theta_{\vec{y}}$, then this differential surface appears in solid angle

$$\mathrm{d}\omega' = \frac{\mathrm{d}y \cos^+ \theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2},$$

that is, it is proportional to the size of the illuminating surface, it depends on its orientation, and inversely proportional to the square of the distance. If differential surface $\mathrm{d}y$ is occluded, then solid angle $\mathrm{d}\omega'$ is zero. To model visibility, we can introduce a **visibility indicator** $v(\vec{x},\vec{y})$ that is 1 if the two points see each other and zero otherwise. With these, an equivalent form of the transport operator uses an integration over the **set of surface points** S:

$$L^r(\vec{x},\vec{\omega}) = \int_S L(\vec{y},\vec{\omega}'_{\vec{y}\to\vec{x}}) f_r(\vec{\omega}'_{\vec{y}\to\vec{x}},\vec{x},\omega) G(\vec{x},\vec{y}) \mathrm{d}y. \tag{3.1}$$

In this equation

$$G(\vec{x},\vec{y}) = v(\vec{x},\vec{y}) \frac{\cos^+ \theta'_{\vec{x}} \cos^+ \theta_{\vec{y}}}{|\vec{x}-\vec{y}|^2} \tag{3.2}$$

is the **geometric factor**, where $\theta'_{\vec{x}}$ and $\theta_{\vec{y}}$ are the angles between the surface normals and direction $\omega_{\vec{y}\to\vec{x}}$ that is between $\vec{y}$ and $\vec{x}$.

Final gathering, i.e. the computation of the reflection of the indirect illumination toward the eye, is one of the most time consuming steps of realistic rendering. The evaluation of this integral usually requires many sampling rays from each shaded point. Classical ray-casting would find for all shaded point $\vec{x}$ and direction $\vec{\omega}'$ the visible surface point $\vec{y}$.

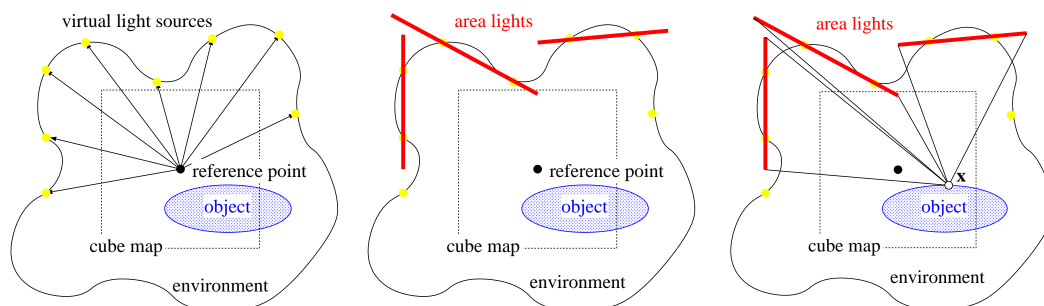


Figure 3.1: Diffuse/glossy final gathering. Virtual lights correspond to cube map texels. These point lights are grouped to form large area lights by downsamling the cube map. At shaded point **x**, the illumination of the area lights is computed without visibility tests.

The first idea to speed up this process is to separate dynamic and static objects and reuse illuminating points when the illumination on the dynamic object is computed. It means that first we obtain a set of virtual lights that represent the indirect illumination and calculate just the single reflection of these lights during final gathering (Figure 3.1). Assuming that the same set of illuminating points are visible from each shaded point, **self-shadowing** effects are ignored, and apply just a simple test based on the normal vector and the illumination direction to determine whether the virtual light may illuminate the point. However, while shadows are crucial for direct lighting, shadows from indirect lighting are not so visually important. Thus the user finds this simplification acceptable.

Unfortunately, this simplification alone cannot allow real time frame rates. The evaluation of the reflected radiance at a shaded point still requires the evaluation of the irradiance and the orientation angle, and the multiplication with the diffuse reflectance for all directions. In order to further increase the rendering speed, we propose to carry out as much computation globally for virtual lights, as possible. Intuitively, global computation means that the sets of virtual light sources are replaced by larger homogeneous **area light sources**. Since the total area of these lights is assumed to be visible, the reflected radiance can be analytically evaluated once for a whole set of virtual light sources.

In this chapter we address the numerical instability of virtual lights and eliminate the need of storing normal vectors at these lights. The basic idea is that we always consider four virtual lights that are close to each other and assume that the surface is roughly planar between them. Thus instead of point sources we have a set of homogeneous area light sources. Since self-shadowing is ignored, the reflection of these area light sources can be computed analytically, in a numerically stable way.

3.1 The new algorithm

The first step of the algorithm is the generation of virtual lights that may illuminate the given dynamic object. To find these points, the scene is rendered from reference point $\vec{o}$ of the dynamic object, and the resulting images are put into an environment map. Not only the radiance of the visible points is evaluated but the distance between the reference point and the visible surface is found and stored. In each texel of the environment map[**?**] a potential virtual light source is visible.

In order to estimate the integral of the rendering equation, the set of surface points visible from the reference point is partitioned according to the texels of the environment map. The surface between four points visible from the texel corners is approximated by a quadrilateral, thus the environment is assumed to be a list of quadrilaterals $S_i, i = 1, \ldots, N$. After partitioning, the irradiance is expressed by the following sum:

$$I(\vec{x}) = \sum_{i=1}^{N} \int_{S_i} L(\vec{y}) \cdot \frac{\cos^+ \theta_{\vec{x}} \cdot \cos^+ \theta_{\vec{y}}}{|\vec{x} - \vec{y}|^2} \mathrm{d}y.$$

Since $\mathrm{d}y \cos^+ \theta_{\vec{y}} / |\vec{x} - \vec{y}|^2$ is the projection of area $\mathrm{d}y$ onto the surface of a unit hemisphere placed around $\vec{x}$, and factor $\cos^+ \theta_{\vec{x}}$ represents a further projection from the sphere onto the tangent plane, the irradiance integral can also be evaluated on the tangent plane of the surface at $\vec{x}$:

$$I(\vec{x}) = \sum_{i=1}^{N} \int_{P_i} L(\vec{y}_p) \mathrm{d}y_p$$

where P_i is the double projection of the visible surface, first onto the **hemisphere** then onto the **tangent plane**. Let us assume that $L(\vec{y}_p)$ is linear. In this case the irradiance is the product of the area of P_i and the average of the radiances stored at the four corner texels:

$$I(\vec{x}) = |P_i| \cdot \frac{L_i^1 + L_i^2 + L_i^3 + L_i^4}{4}$$

Let us consider a single term of this sum representing the radiance reflected from S_i.

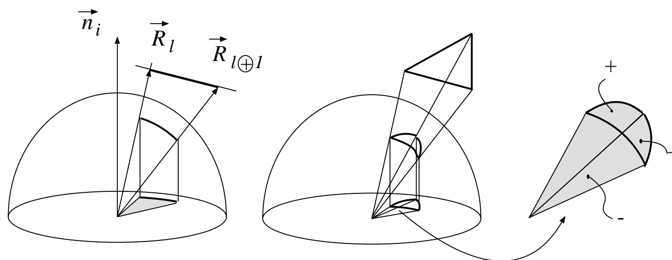


Figure 3.2: Hemispherical projection of a planar polygon

Area $|P_i|$ is in fact the **polygon-to-point form factor** [BRW89]. Consider only one edge line of the polygon first, and two subsequent vertices, $\vec{R}_l$ and $\vec{R}_{l\oplus 1}$, on it (Figure 3.2) [SKe95]. The hemispherical projection of this line is a half great circle. Since the radius of this great circle is 1, the area of the sector formed by the projections of $\vec{R}_l$ and $\vec{R}_{l\oplus 1}$ and the center of the hemisphere is simply half the angle of $\vec{R}_l$ and $\vec{R}_{l\oplus 1}$. Projecting this sector orthographically onto the equatorial plane, an ellipse sector is generated, having the area of the great circle sector multiplied by the cosine of the angle of the surface normal $\vec{n}_i$ and the normal of the segment $(\vec{R}_l \times \vec{R}_{l\oplus 1})$.

The area of the doubly projected polygon can be obtained by adding and subtracting the areas of the ellipse sectors of the different edges, as is demonstrated in Figure 3.2, depending on whether the projections of vectors $\vec{R}_l$ and $\vec{R}_{l\oplus 1}$ follow each other clockwise when looking at them from the direction of the surface normal. This sign value provided by the dot product of the cross product of the two vertex vectors and the normal vector. Finally, the double projected polygon is a summation:

$$\sum_{l=0}^{L-1} \frac{1}{2} \cdot \text{angle}(\vec{R}_l, \vec{R}_{l\oplus 1}) \cdot \left(\frac{(\vec{R}_l \times \vec{R}_{l\oplus 1})}{|\vec{R}_l \times \vec{R}_{l\oplus 1}|} \cdot \vec{n}_i \right). \tag{3.3}$$

The proposed algorithm first computes an environment cube map from the reference point and stores the radiance and distance values of the points visible in its pixels. We usually generate $6 \times 256 \times 256$ pixel resolution cube maps. Then the cube map is downsampled to have `M`×`M` pixel resolution faces (`M` is 4 or even 2). One texel of the low-resolution cubemap represents an area light source. Note that both radiance and distance values are averaged, thus finally we have larger lights having the average radiance of the small lights and placed at their average position.

During final gathering for each texel in each cube face we should calculate the virtual light source's exact location and area and evaluate it's contribution to the reflected radiance. Algorithm 1 shows the fragment shader pseudo code used to calculate diffuse indirect shading.

3.2 Results

In order to demonstrate the results, we took a simple environment consisting of a colored cubic room. The first set of pictures shows the Stanford bunny model inside the room (Figure 3.3). The images of the first column were rendered by the traditional environment mapping technique for diffuse materials where a precalculated **convolution** enables us to determine the irradiance at the reference point with a single lookup. This method has correct results only at the reference point and cannot deal with the position of the object, thus the bunny looks similar everywhere. The second column shows the **point-to-point form factor**. Although this method have pleasing results for points other than the reference point, it has artifacts for points too close to the virtual light sources. The right column shows our polygon-to-point form factor

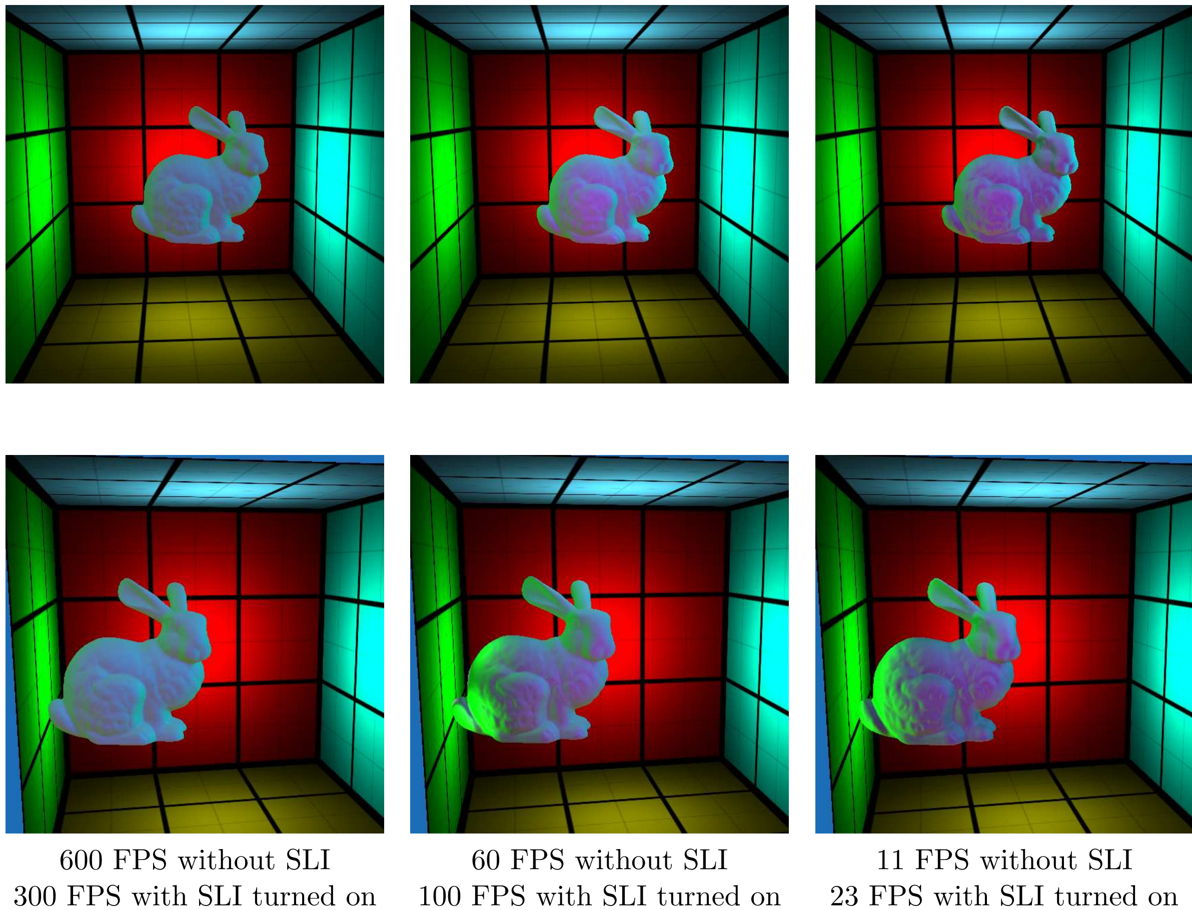
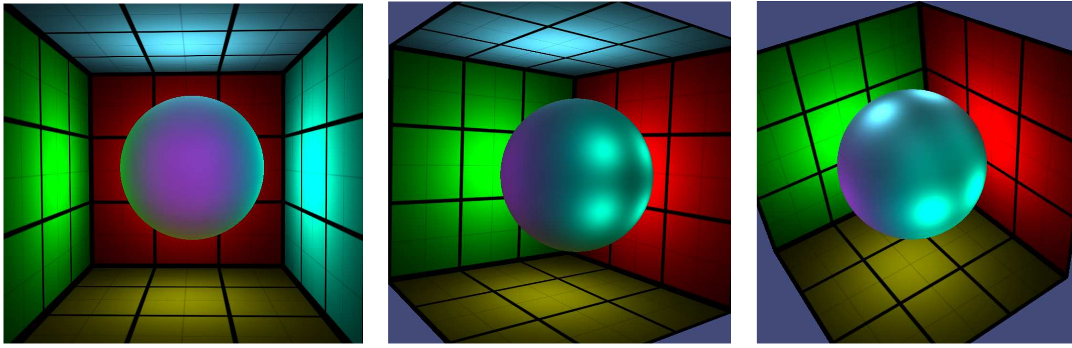
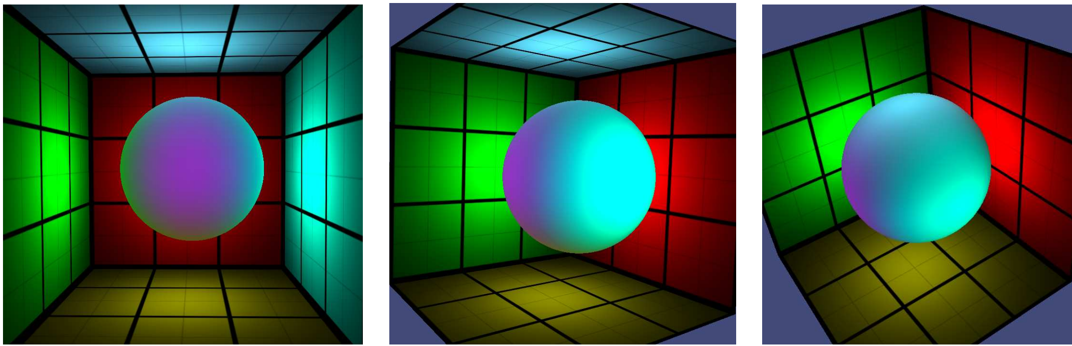


Figure 3.3: *The Stanford bunny model (about 25000 vertices and 50000 faces) in a colored box. On the right images diffuse reflections are calculated with only environment map convolution and one environment lookup. On the center images point to point form factor while on the left images polygon to point form factor is used. The simple one environment map only produces pleasable results for points that are close to the environment reference point. The point to point form factor method can handle greater distances, but as the shaded surface gets close to the walls it gets unstable (see the bright green spikes on the bunny's tail and back). The polygon to point form factor has good results for any object positions. Configuration: 700x700 resolution on NVIDIA GeForce 7950 GX2 with SLI support, AMD Athlon64 Dual 4600+ processor.*



150 FPS without SLI, 185 FPS with SLI turned on

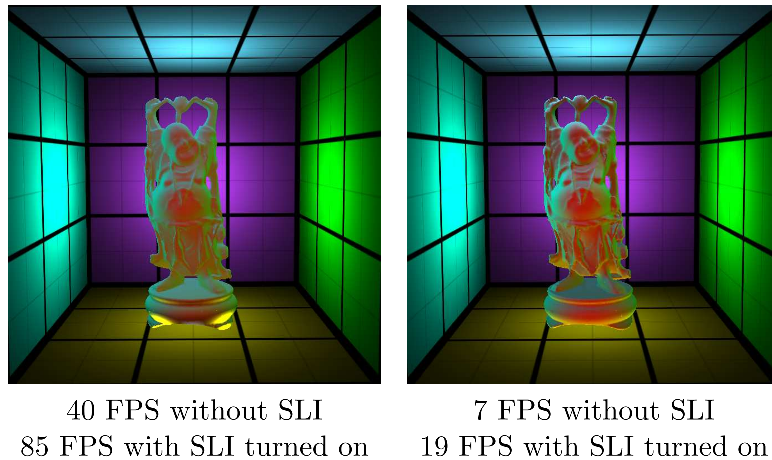


32 FPS without SLI, 63 FPS with SLI turned on

Figure 3.4: *Sphere (about 2300 vertices and 2300 faces) in a colored box. Upper row with point to point form factor, bottom row with our polygon to point form factor. As the object gets closer to the walls the virtual light sources become visible in case of point to point form factor, while the polygon to point form factor method does not have this artifact. Configuration: 700x700 resolution on NVIDIA GeForce 7950 GX2 with SLI support, AMD Athlon64 Dual 4600+ processor.*

Algorithm 1 Diffuse surface point shading

$indirectIllum \leftarrow 0$
for each cube map face **do**
  for each texel **do**
    $triangleArea \leftarrow 0$
    Calculate texel corner directions L_1, L_2, L_3, L_4 from cube map center
    Read texel corner distances d_1, d_2, d_3, d_4
    $L_1 = L_1 \cdot d_1,\ L_2 = L_2 \cdot d_2,\ L_3 = L_3 \cdot d_3,\ L_4 = L_4 \cdot d_4$
    Calculate texel corner directions R_1, R_2, R_3, R_4 from shaded point
    for each texel quad edge **do**
      $texelContribution \leftarrow$ evaluate equation 3.3
      $triangleArea += max(0, texelContribution)$
    end for
    $texelColor \leftarrow$ color of cube map texel
    $indirectIllum += triangleArea \cdot texelColor$
  end for
end for
return $indirectIllum$



40 FPS without SLI
85 FPS with SLI turned on

7 FPS without SLI
19 FPS with SLI turned on

Figure 3.5: *Diffuse Stanford Buddha (about 35000 vertices and 67000 faces) in a box. Left image with point to point form factor, right image with our polygon-to-point form factor method. Configuration: 700x700 resolution on NVIDIA GeForce 7950 GX2 with SLI support, AMD Athlon64 Dual 4600+ processor.*

method which has correct results for arbitrary point positions. The difference between point to point and polygon-to-point factors can clearly be seen in the second set of pictures (Figure 3.4). The first row shows point to point while the second row shows polygon-to-point form factor results. While the two methods show similar results when the object is in the center of the room there is a significant improvement in case of the polygon-to-point factor method when the sphere gets close to the walls.

The third set of pictures (Figure 3.5) shows the Stanford Buddha model in the room with point to point (left) and polygon-to-point form factors (right). The images on the left show visual errors at surface points near the environment (see the keystone of the statue). In Figure 3.7 the Buddha was placed in a scientific laboratory. This environment was implemented in an axctual game engine called **Ogre3D**.

The speed measurements show that evaluating the polygon-to-point form factor has high computational costs, which can make it hard to integrate it into a real-time application. We should note that all the discussed images and measurements were taken with the use of a 4×4 resolution downsampled cube map. Using a smaller sized cube map can greatly improove perfomance without significant image quality loss (Figure 3.6). We should also note that recent graphics hardware like the NVIDIA GeForce GTX 260 has such high computational power that even higher resolution cube maps can be used in real-time. Table 3.1 and 3.2 shows speed mesurements on newer hardware and with lower cubemap resolutions.

4×4 **cube map resolution**			
GPU	classic	point to point	polygon to point
GeForce 7900 GT	600 FPS	60 FPS	11 FPS
GeForce GTX 260	850 FPS	100 FPS	50 FPS

Table 3.1: *Speed comparison of the different diffuse environment mapping methods on different graphics hardware using* 4×4 *downsampled cube map resolution.*

2×2 **cube map resolution**			
GPU	classic	point to point	polygon to point
GeForce 7900 GT	600 FPS	150 FPS	80 FPS
GeForce GTX 260	850 FPS	400 FPS	150 FPS

Table 3.2: *Speed comparison of the different diffuse environment mapping methods on different graphics hardware using* 2×2 *downsampled cube map resolution.*

3.3 Conclusions

This chapter presented a GPU based method for computing diffuse reflections of the incoming radiance stored in environment maps. The environment map is considered as a definition of large area light sources whose reflections are obtained analytically without checking self-shadowing. The presented method runs in real-time and provides visually pleasing results.

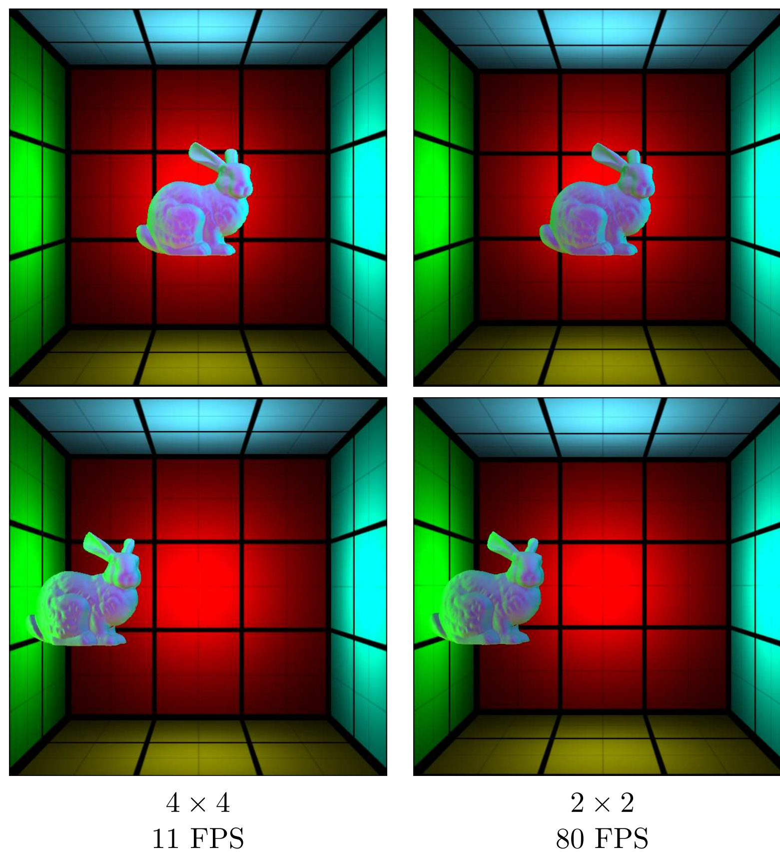


Figure 3.6: *Speed and quality comparison of different cube map resolutions. On the right images* 4×4*, on the left images* 2×2 *resolution was used. Configuration: 700x700 resolution on NVIDIA GeForce 7900 GT.*

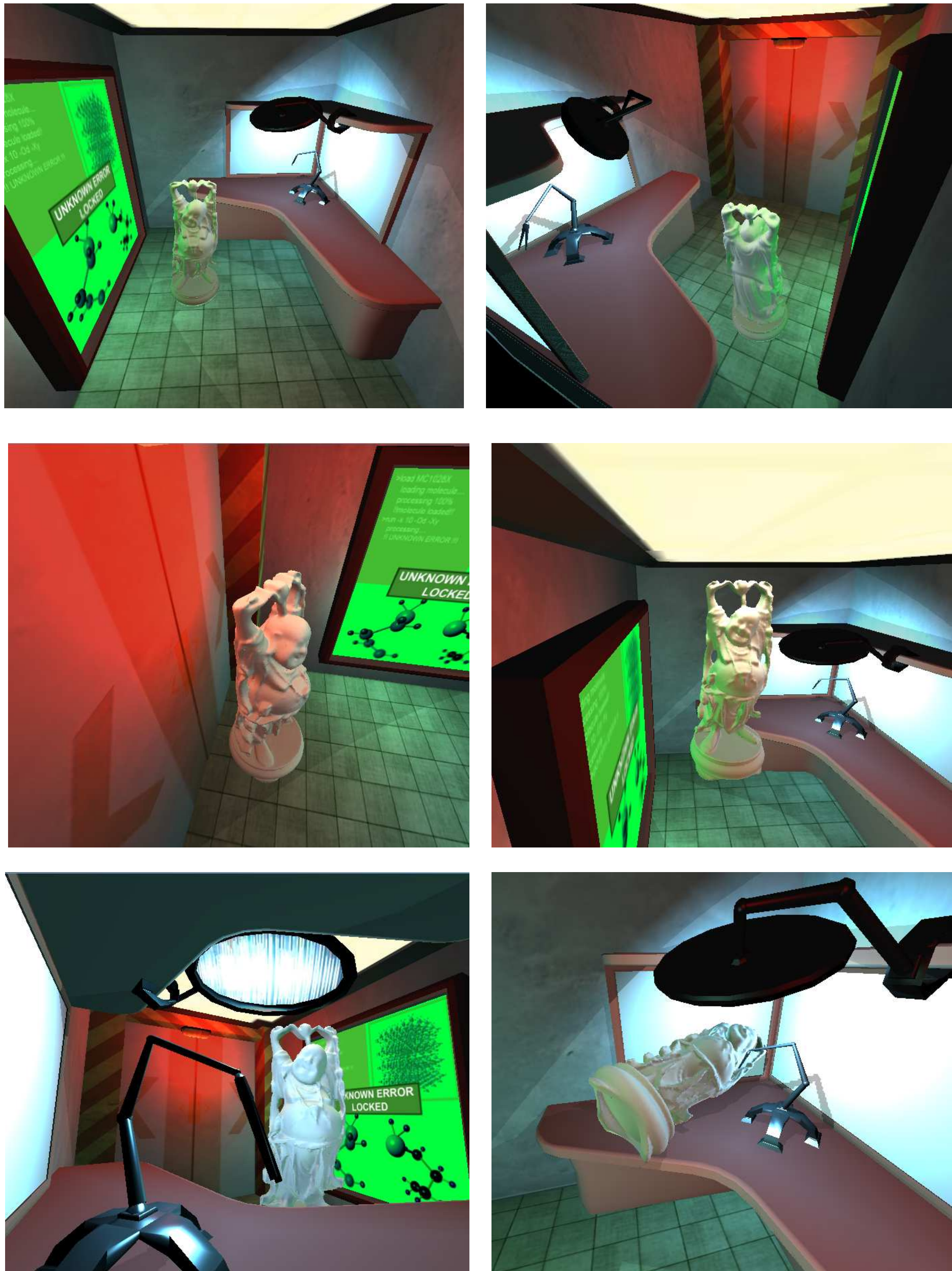


Figure 3.7: *Screen shots from Buddha in a laboratory (the whole scene with the Buddha is about 38000 vertices and 70000 faces). Ogre3D implementation. 15 FPS, 800x600 resolution on NVIDIA GeForce 7800GT, AMD Athlon 64 3500+ processor.*

Chapter 4

Volumetric ambient occlusion

This chapter focuses on the fast computation of the reflection of the ambient light[1]. We shall assume that the primary source of illumination in the scene is a homogeneous sky light source of radiance L^a. For the sake of simplicity, we consider only diffuse surfaces. According to the rendering equation, **reflected radiance** L^r in **shaded point** $\vec{x}$ can be obtained as:

$$L^r(\vec{x}) = \frac{a(\vec{x})}{\pi} \int_{\Omega'} L^{in}(\vec{x}, \vec{\omega}) \cos^+ \theta' \mathrm{d}\omega', \tag{4.1}$$

where Ω' is the set of directions in the hemisphere above $\vec{x}$, $L^{in}(\vec{x}, \vec{\omega}')$ is the **incident radiance** from direction $\vec{\omega}'$, $a(\vec{x})$ is the **albedo** of the surface, and θ' is the angle between the surface normal and illumination direction $\vec{\omega}$. If incident angle θ' is greater than 90 degrees, then the negative cosine value should be replaced by zero, which is indicated by superscript $^+$ in $\cos^+$.

If no surface is seen from $\vec{x}$ at direction $\vec{\omega}'$, then shaded point $\vec{x}$ is said to be **open** in this direction, and incident radiance L^{in} is equal to ambient radiance L^a. If there is an occluder nearby, then the point is called **closed** at this direction and the incident radiance is the radiance of the occluder surface. The exact determination of this radiance would require a global illumination solution, which is too costly in real-time applications. Thus, we simply assume that the radiance is proportional to the ambient radiance and to a factor expressing the **openness** — also called **accessibility** — of the point. The theory of classical ambient occlusion considers an occluder to be "nearby" if its distance is smaller than a predefined threshold R. However, such an uncertain property like "being close" is better to handle by a **fuzzy measure** $\mu(d(\vec{\omega}'))$ that defines how strongly direction $\vec{\omega}'$ belongs to the set of open directions based on distance d of the occlusion at direction $\vec{\omega}'$. In other words, this fuzzy measure expresses how an occluder at distance d allows the ambient lighting to take into effect. Relying on the physics analogy of the non-scattering participating media, a possible fuzzy measure could be $\mu(d) = 1 - \exp(-\tau d)$ where τ is the **absorption coefficient** of the media [IKSZ03, AMS⁺08]. Unfortunately, this interpretation requires the distance of far occlusions as well, while the classical ambient occlusion does not have to compute occlusions that are farther than R, localizing and thus simplifying the shading process.

Thus, for practical fuzzy measures we use functions that are non-negative, monotonously increasing from zero and reach 1 at distance R. The particular value of R can be set by the application developer. When we increase this value, shadows due to ambient occlusions get larger and softer.

Using the fuzzy measure of openness, the incident radiance is

$$L^{in}(\vec{x}, \vec{\omega}') = L^a \mu(d(\vec{\omega}')),$$

[1]Should the scene contain other sources, e.g. directional or point lights, their effects can be added to the illumination of the ambient light.

thus the reflected radiance (Equation 4.1) can be written in the following form:

$$L^r(\vec{x}) = a(\vec{x}) \cdot L^a \cdot O(\vec{x}),$$

where

$$O(\vec{x}) = \frac{1}{\pi} \int_{\Omega'} \mu(d(\vec{\omega}')) \cos^+ \theta' \mathrm{d}\omega'. \tag{4.2}$$

is the **ambient occlusion** representing the local geometry. It expresses how strongly the ambient lighting can take effect on point $\vec{x}$. The computation of ambient occlusion requires the distances $d(\vec{\omega}')$ of occluders in different directions, which are usually obtained by computationally expensive ray-tracing.

This chapter proposes an ambient occlusion algorithm where the directional integral of Equation 4.2 is replaced by a volumetric one that can be efficiently evaluated. The resulting method

- runs at high frame rates on current GPUs,
- does not require any pre-processing and thus can be applied to general dynamic models,
- can provide smooth shading with just a few samples due to partial analytic integration and interleaved sampling.

These properties make the algorithm suitable for real-time rendering and games.

4.1 The new model of ambient lighting

The evaluation of the directional integral in the ambient occlusion formula (Equation 4.2) requires rays to be traced in many directions, which is rather costly and needs complex GPU shaders. Thus, we transform this directional integral to a volumetric one that can be efficiently evaluated on the GPU. The integral transformation involves the following steps:

1. Considering its derivative instead of the fuzzy membership function, we replace the expensive ray tracing operation by a simple **containment test**. This replacement is valid if neighborhood R is small enough to allow the assumption that a ray intersects the surface at most once in interval $[0, R]$.
2. Factor $\cos^+ \theta'$ in Equation 4.2 is compensated by transforming the integration domain mapping the hemisphere above the surface to the tangent sphere. This makes ambient occlusion depend on the volume of that portion of the tangent sphere which belongs to the "free" space that is not occupied by objects.

4.1.1 Replacing ray tracing by containment tests

Let us assume that the surfaces subdivide the space into an **outer part** where the camera is and into **inner parts** that cannot be reached from the camera without crossing a surface. We define the **characteristic function** $\mathcal{I}(\vec{p})$ that is 1 if point $\vec{p}$ is an outer point and zero otherwise. The boundary between the inner and outer parts, i.e. the surfaces, belong to the outer part by definition (Figure 4.1).

The form of the indicator function $\mathcal{I}(\vec{p})$ depends on the type or representation of the surfaces (Figure 4.2).

1. **Implicit surfaces** are defined by equation $f(\vec{p}) = 0$. In this case $\mathcal{I}(\vec{p})$ is 1 if $f(\vec{p})$ is positive or zero (the point is outside of the object or on its surface), and zero otherwise.

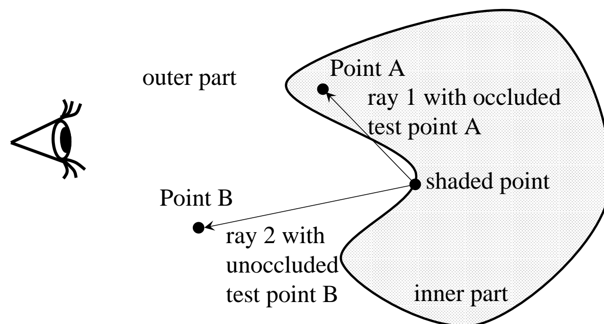


Figure 4.1: Replacing ray tracing by containment tests. If a test point (Point B) along a ray starting at the shaded point being in the outer region or on the region boundary is also in the outer region, then the ray either has not intersected the surface or has intersected at least two times. Supposing that the ray is short enough and thus may intersect the surface at most once, the condition of being in the outer region is equivalent to the condition that no intersection happened.

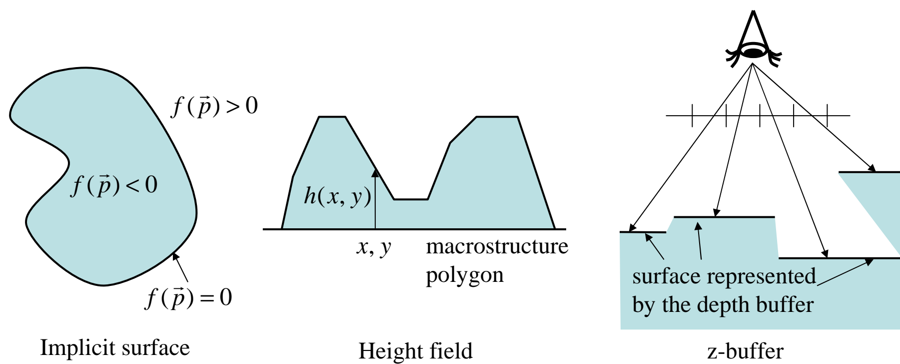


Figure 4.2: Three examples for the definition of the characteristic function. Outer regions where $\mathcal{I} = 1$ are blank, inner regions where $\mathcal{I} = 0$ are filled.

2. **Height fields** represent a particularly important special case of implicit surfaces. A height field is formed by points (x, y, z) satisfying equation $z = h(x, y)$. Since the eye is usually "above" the height field, the characteristic function should indicate the case when for point $\vec{p} = (p_x, p_y, p_z)$, relation $p_z \geq h(p_x, p_y)$ holds. Note that a polygon of a **displacement mapped** mesh can also be considered as a height field in the tangent space of the macrostructure polygon.

3. As in **screen-space ambient occlusion** methods [Mit07, Sai08], the content of the z-buffer can also provide an inner–outer distinction. If a point is reported to be occluded by the depth map, then surfaces separate this point from the eye, thus its characteristic value is zero. If the point passes the depth test, then it is in the same region as the eye, so it gets indicator value 1.

Note that in all these three cases, not only are we able to classify a point as inner or outer, but we can also determine the distance between point (p_x, p_y, p_z) and the surface along axis z. When the content of the z-buffer defines the separation, the z-direction is the viewing direction in clipping space. Reading depth value z^* with the p_x, p_y coordinates of the point, the distance between the point and the surface can be expressed as $z^* - p_z$. If the characteristic function is defined by height field $h(x, y)$, then the distance along axis z is $p_z - h(p_x, p_y)$. Finally, when the surface is defined by implicit equation $f(x, y, z) = 0$, we can use Taylor's approximation to estimate the distance between point (p_x, p_y, p_z) and point (p_x, p_y, z^*) that is on the surface:

$$f(p_x, p_y, p_z + (z^* - p_z)) = 0 \quad \Rightarrow \quad z^* - p_z \approx -\frac{f(p_x, p_y, p_z)}{\partial f / \partial z}.$$

In order to evaluate the ambient occlusion using containment tests, we express it as a three dimensional integral. For a point at distance d, fuzzy measure $\mu(d)$ can be found by integrating its derivative from 0 to d since $\mu(0) = 0$. Then the integration domain can be extended from d to R by multiplying the integrand by a **step function** which replaces the derivative by zero when the distance is greater than d:

$$\mu(d) = \int_0^d \frac{\mathrm{d}\mu(r)}{\mathrm{d}r} \mathrm{d}r = \int_0^R \frac{\mathrm{d}\mu(r)}{\mathrm{d}r} \epsilon(d - r) \mathrm{d}r,$$

where $\epsilon(x)$ is the step function, which is 1 if $x \geq 0$ and zero otherwise. Note that this formulation requires generalized derivatives for classical ambient occlusion since its membership function is a step function, and the derivative of the step function is a **Dirac-delta** (Figure 4.3).

Substituting this integral into the ambient occlusion formula, we get

$$O(\vec{x}) = \frac{1}{\pi} \int_{\Omega'} \int_0^R \frac{\mathrm{d}\mu(r)}{\mathrm{d}r} \epsilon(d - r) \cos^+ \theta' \mathrm{d}r \mathrm{d}\omega'. \quad (4.3)$$

Let us consider a ray of equation $\vec{x} + \vec{\omega}' r$ where shaded point $\vec{x}$ is the origin, $\vec{\omega}'$ is the direction, and distance r is the ray parameter. Indicator $\epsilon(d - r)$ is 1 if the distance of the intersection d is larger than current distance r and zero otherwise, that is, it shows whether or not intersection has happened. If we assume that the ray intersects the surface at most once in the R-neighborhood, then the condition that $\vec{x} + \vec{\omega}' r$ is in the outer part, i.e. $\mathcal{I}(\vec{x} + \vec{\omega}' r) = 1$, also shows that no intersection has happened yet. Thus, provided that only one intersection is possible in the R-neighborhood, indicators $\epsilon(d - r)$ and $\mathcal{I}(\vec{x} + \vec{\omega}' r)$ are equivalent. Replacing step function $\epsilon(d - r)$ by indicator function $\mathcal{I}$ in Equation 4.3, we obtain

$$O(\vec{x}) = \frac{1}{\pi} \int_{\Omega'} \int_0^R \frac{\mathrm{d}\mu(r)}{\mathrm{d}r} \mathcal{I}(\vec{x} + \vec{\omega}' r) \cos^+ \theta' \, \mathrm{d}r \mathrm{d}\omega'. \quad (4.4)$$

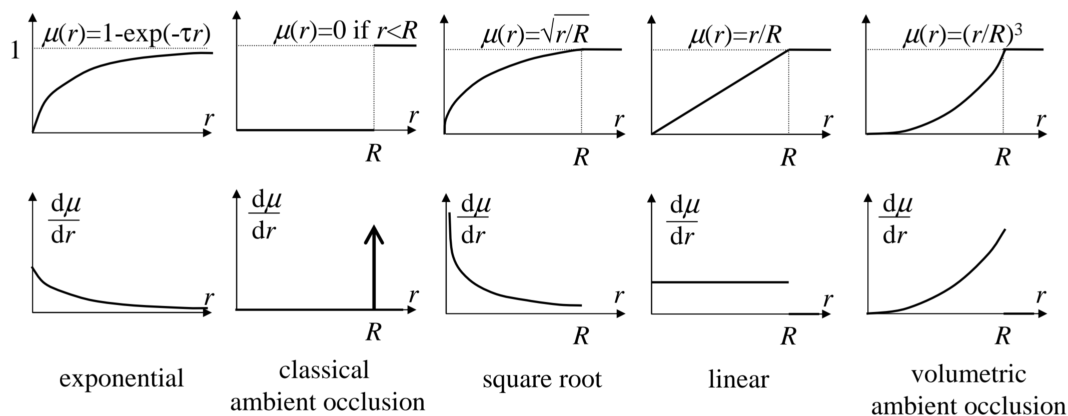


Figure 4.3: Example fuzzy membership functions and their derivatives. Note that the exponential function [IKSZ03, AMS+08] can be given a physical interpretation but it requires global visibility computations, while other functions need visibility checks only in a neighborhood of radius R. Classical ambient occlusion [PG04] uses a non-fuzzy separation. The localized square root is a good compromise between the global exponential and the non-fuzzy separation [MSC03]. We also included the membership function of the volumetric ambient occlusion that is proposed in the next subsection.

One possibility for the estimation of this integral is the Monte Carlo quadrature. According to the concept of **importance sampling** we use cosine distributed sample directions $\vec{\omega}_i$ and distance samples r_i following density $\mathrm{d}\mu(r)/\mathrm{d}r$, that are obtained by transforming uniformly distributed samples ξ_i as $r_i = \mu^{-1}(\xi_i)$. Note that in our case the number of samples n is small and the random samples should be generated only once, thus we can hardwire samples $(X_i, Y_i, Z_i) = \vec{\omega}_i r_i$ into the shader code computing

$$O(\vec{x}) \approx \frac{1}{n}\sum_{i=1}^{n} \mathcal{I}\left(\vec{x} + X_i\vec{T} + Y_i\vec{B} + Z_i\vec{N}\right). \tag{4.5}$$

where $\vec{T}$, $\vec{B}$, and $\vec{N}$, are the tangent, binormal, and normal vectors, respectively. We call this method the **containment test based algorithm**. This method would approximate the quadrature averaging n binary numbers, thus the average would have **binomial distribution** with mean $O(\vec{x})$ and standard deviation $\sqrt{O(\vec{x})(1 - O(\vec{x}))/n}$. Ambient occlusion $O(\vec{x})$ is in $[0, 1]$ and the standard deviation reaches its maximum $1/\sqrt{4n}$ when the ambient occlusion value is $1/2$. In order to reduce the maximum standard deviation (i.e. the error) below 0.01 in a pixel, we need 2500 random samples, which are too many for real-time applications.

Thus, we need a better estimate for the ambient occlusion integral that requires less samples to achieve the same accuracy. We combine three techniques to reach this goal. Most importantly, the integral is re-formulated to reduce the variation of the integrand and to allow the incorporation of all available information into the quadrature. In particular, we use the distance to the separating surface as such additional information. Secondly, we replace statistically independent random samples by low-discrepancy samples following the **Poisson-disk distribution**. Finally, we use interleaved sampling, i.e. exploit the samples obtained in the neighboring pixels to improve the estimate in the current pixel.

4.1.2 Exploiting the distance to the separating surface

Inspecting Equation 4.4 we can observe that the ambient occlusion is a double integral inside a hemisphere where the integrand includes factor $\cos^+\theta'$, thus directions enclosing a larger angle

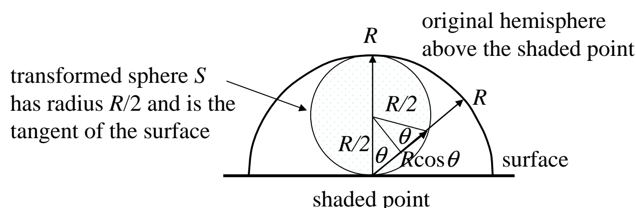


Figure 4.4: Transforming a hemisphere of radius R by shrinking distances by $\cos\theta$ results in another sphere of radius $R/2$.

with the surface normal are less important. Instead of the multiplication, the effect of this cosine factor can also be mimicked by reducing the size of the integration domain proportionally to $\cos^+\theta$. Note that this is equivalent to the original integral only if the remaining factors of the integrand are constant, and can be accepted as an approximation in other cases:

$$O(\vec{x}) \approx \frac{1}{\pi}\int\limits_{\Omega'}\int\limits_{0}^{R\cos^+\theta'} \frac{\mathrm{d}\mu(r)}{\mathrm{d}r}\mathcal{I}(\vec{x}+\vec{\omega}'r)\mathrm{d}r\mathrm{d}\omega'.$$

Transforming a hemisphere by shrinking distances in directions enclosing angle θ with the surface normal by $\cos\theta$ results in another sphere, which is denoted by S (Figure 4.4). This new sphere has radius $R/2$ and its center is at distance $R/2$ from shaded point $\vec{x}$ in the direction of the surface normal. While the original hemisphere had the shaded point as the center of its base circle, the surface will be the tangent of the new sphere at shaded point $\vec{x}$.

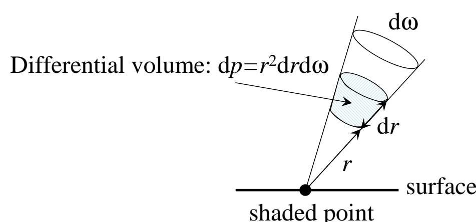


Figure 4.5: Differential volume swept when r changes by dr and direction $\vec{\omega}$ is in $d\omega$.

In order to replace integrals over directions and distances by a volumetric integral, let us examine the volume swept when distance r changes by $\mathrm{d}r$ and direction $\vec{\omega}'$ varies in solid angle $\mathrm{d}\omega'$ (Figure 4.5). During this, we sweep a differential volume $\mathrm{d}p = r^2\mathrm{d}r\mathrm{d}\omega'$. Thus, we can express the ambient occlusion as a **volumetric integral** in S instead of a double integral of directions and distances in the following way:

$$O(\vec{x}) \approx \frac{1}{\pi}\int\limits_{\vec{p}\in S} \frac{\mathrm{d}\mu(r(\vec{p}))}{\mathrm{d}r}\frac{1}{(r(\vec{p}))^2}\mathcal{I}(\vec{p})\mathrm{d}p$$

where $r(\vec{p})$ is the distance of point $\vec{p}$ from the shaded point. If we set the fuzzy membership function such that $\mathrm{d}\mu(r)/\mathrm{d}r$ is proportional to r^2 (last column of Figure 4.3), then the ambient occlusion integral becomes just the volumetric integral of the membership function. This observation leads us to a new definition of the openness of a point, which we call the **volumetric ambient occlusion** and denote by $V(\vec{x})$ to distinguish it from ambient occlusion $O(\vec{x})$. The volumetric ambient occlusion is the relative volume of the unoccluded part of the **tangent**

sphere S. Formally, the volumetric ambient occlusion function is defined as:

$$V(\vec{x}) = \frac{\int_S \mathcal{I}(\vec{p})\mathrm{d}p}{|S|}, \tag{4.6}$$

where $|S| = 4(R/2)^3\pi/3$ is the volume of the tangent sphere, which makes sure that the volumetric ambient occlusion is also in $[0,1]$. Figure 4.6 compares the computation of volumetric ambient occlusion to ray-tracing based methods and to the containment test based algorithm.

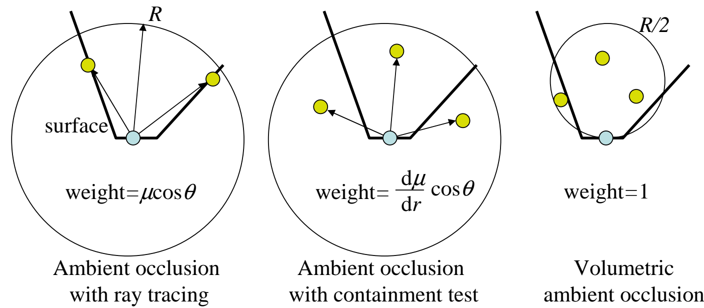


Figure 4.6: Comparison of the ambient occlusion computation with ray-tracing, containment test, and with the volumetric interpretation. Note that these methods differ both in sample generation and weighting.

When evaluating volumetric integral $\int_S \mathcal{I}(\vec{p})\ dp$, we can take advantage of the distances to the separating surface. The volume is computed as a sum of "pipes" (Figure 4.7). The axes of these pipes are parallel to axis z and they have the same cross section area. The pipes are limited by either the separating surface or the surface of the tangent sphere.

Let us denote the center of the tangent sphere by (x_c, y_c, z_c) and consider a disk of radius $R/2$ around this point that is perpendicular to axis z. We sample n uniformly distributed points (x_i, y_i) in a unit radius disk, and transform them onto the considered disk of the tangent sphere, that is perpendicular to direction z. A transformed point has coordinates $(x_i R/2, y_i R/2, z_c)$ and is called **sample** in Figure 4.7. A line crossing the ith sample point and being parallel with axis z enters the sphere at $z_i^{entry} = z_c - \frac{R}{2}\sqrt{1 - x_i^2 - y_i^2}$ and exits it at $z_i^{exit} = z_c + \frac{R}{2}\sqrt{1 - x_i^2 - y_i^2}$. The points on this line belong to the outer region when their z coordinates are less than z^*, where z^* represents the intersection of this line with the surface.

The length of traveling in the outer, i.e. unoccluded part of the sphere is $\Delta z_i = z_i^* - z_i^{entry}$ when $z_i^{entry} \le z_i^* \le z_i^{exit}$ (Case A in Figure 4.7). The traveled distance is $\Delta z_i = z_i^{exit} - z_i^{entry}$ if the surface is behind the sphere (Case B). If $z_i^* < z_i^{entry}$, then $\Delta z_i = 0$ since this part of the sphere is occluded (Case C).

If n sample points are uniformly distributed on the disk of radius $R/2$, then a line is associated with $(R/2)^2\pi/n$ area of the disk. Thus, we get the following approximation of the volume of the unoccluded part of the tangent sphere:

$$\int_S \mathcal{I}(\vec{p})\mathrm{d}p \approx \frac{R^2\pi}{4n}\sum_{i=1}^{n}\Delta z_i. \tag{4.7}$$

The fragment shader gets samples (x_i, y_i) as a constant array and estimates the volumetric ambient occlusion of the point as

$$V(\vec{x}) = \frac{\int_S \mathcal{I}(\vec{p})\mathrm{d}p}{|S|} \approx \frac{3}{2Rn}\sum_{i=1}^{n}\Delta z_i = \frac{1}{F}\sum_{i=1}^{n}\Delta z_i. \tag{4.8}$$

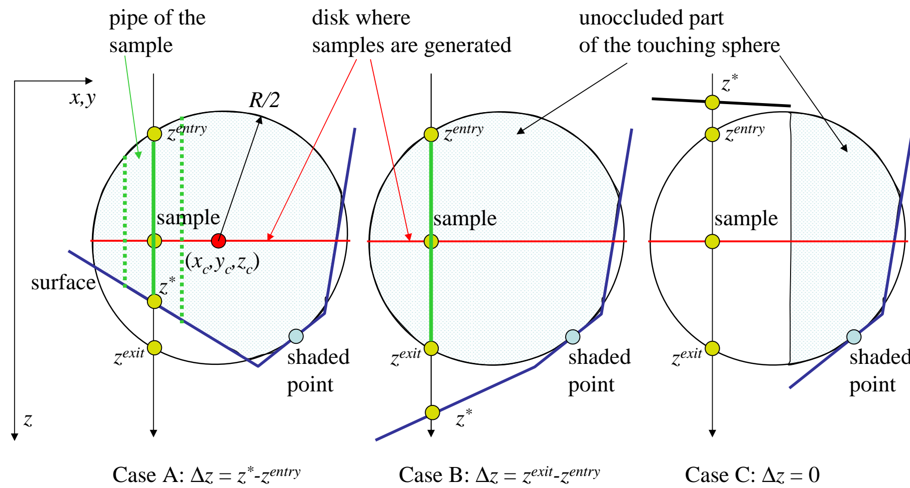


Figure 4.7: Computation of the volumetric ambient occlusion. The volume of the unoccluded part of the tangent sphere of radius $R/2$ is approximated by uniformly sampling n points called **sample** on the disk that is perpendicular to axis z. Each sample represents the same base area $(R/2)^2\pi/n$ of the disk. Such a base area is the cross section of a "pipe" inside the unoccluded part of the sphere. The volume of the pipe associated with a sample is the base area multiplied with the length of the line segment that is in the unoccluded part of the tangent sphere. The line crossing the sample point enters the sphere at z coordinate z^{entry}, exits it at z^{exit}, and intersects the surface at z^*. To obtain length Δz of the line segment that is in the unoccluded part of the sphere, we identify three main cases. In Case A the line–surface intersection is in the sphere. In Case B the surface is behind the sphere. In Case C the surface is in front of the sphere.

Algorithm 2 Volumetric screen space ambient occlusion

$V \leftarrow 0$
for each sample direction **do**
  Calculate sample location on sample disk
  Calculate enter and exit points z_i^{enter}, z_i^{exit}
  Calculate unoccluded part $\Delta z_i = min(z_i^{exit}, z_i^*) - min(z_i^{enter}, z_i^*)$
  $V += \Delta z_i$
end for
$V /= F$
return V

Note that the normalization constant F is the ratio of the tangent sphere's volume $|S| = 4(R/2)^3\pi/3$ and $R^2\pi/(4n)$, which can be analytically computed. However, instead of the exact value of $|S|$, it is worth approximating it with the same samples as used to compute the volume of the unoccluded part (Equation 4.7), which results in the following constant

$$F = R\sum_{i=1}^{n}\sqrt{1 - x_i^2 - y_i^2}.$$

In this case, the approximations of the volume of the unoccluded part and the volume of the tangent sphere have correlated error. Thus, when their ratio is computed, the error of the volume of the unoccluded part is reduced, as proposed by **weighted importance sampling** [PS66]. Algorithm 2 shows the pseudo code of the volumetric ambient occlusion algorithm, that should be run by a fragment shader.

4.1.3 Noise reduction with interleaved sampling

The quasi-Monte Carlo quadrature has some error in each pixel, which depends on the particular samples used in the quadrature. If we used different quasi-random numbers in neighboring pixels, then dot noise would show up. Using the same quasi-random numbers in every pixel would make the error correlated and replace dot noise by "stripes". Unfortunately, both stripes and pixel noise are quite disturbing. In order to reduce the error without taking excessive number of samples, we apply **interleaved sampling** [KH01] that uses different sets of samples in the pixels of a 4×4 pixel pattern, and repeat the same sample structure periodically. The 16 different sample sets can be obtained from a single set by a rotation around the surface normal vector by random angle α. The rotation is executed in the fragment shader that gets 16 $(\cos\alpha, \sin\alpha)$ pairs in addition to quasi-random samples (x_i, y_i). The errors in the pixels of a 4×4 pixel pattern are uncorrelated, and can be successfully reduced by a low-pass filter of the same size. Thus, interleaved sampling using a 4×4 pixel pattern multiplies the effective sample number by 16 but has only the added cost of a box-filtering with a 4×4 pixel window. When implementing the low-pass filter, we also check whether or not the depth difference of the current and the neighbor pixels exceeds a given limit. If it does, then the neighbor pixel is not included in the averaging operation.

4.2 Results

The proposed methods have been implemented in DirectX/HLSL environment and their performance has been measured on an NVIDIA GeForce 8800 GTX GPU at 800×600 resolution.

Figure 4.8 compares Crytek's screen-space ambient occlusion[2], the containment test based method (Equation 4.5), the volumetric ambient occlusion (Equation 4.8), and a reference that is obtained with the software ray-tracer of MentalRay. We note that screen-space ambient occlusion takes samples on the whole sphere and is not based on the original ambient occlusion formula, thus it assigns middle gray even for completely unoccluded surfaces, giving an unrealistically dark touch to the image. Note that both the containment test based and the volumetric methods are faster than Crytek's SSAO, and the speed is sensitive to the size of the neighborhood in all cases (Figure 4.12). This sensitivity is due to the degraded cache efficiency of z-buffer accesses when the large R requires distant samples to be fetched. This degradation can be avoided by reducing the z-buffer resolution by an additional z-buffer filtering before the ambient occlusion computation. In terms of quality the volumetric ambient occlusion is the closest to the ray-traced reference. The quality degradation with respect to the ray-traced result is mainly due to the assumption that in the R-neighborhood at most one intersection can happen. This assumption limits the simultaneous consideration of close and distant occlusions, which reduces the level of details in the final image. This is the price we should pay for real-time rendering.

[2] http://en.wikipedia.org/wiki/Screen_Space_Ambient_Occlusion

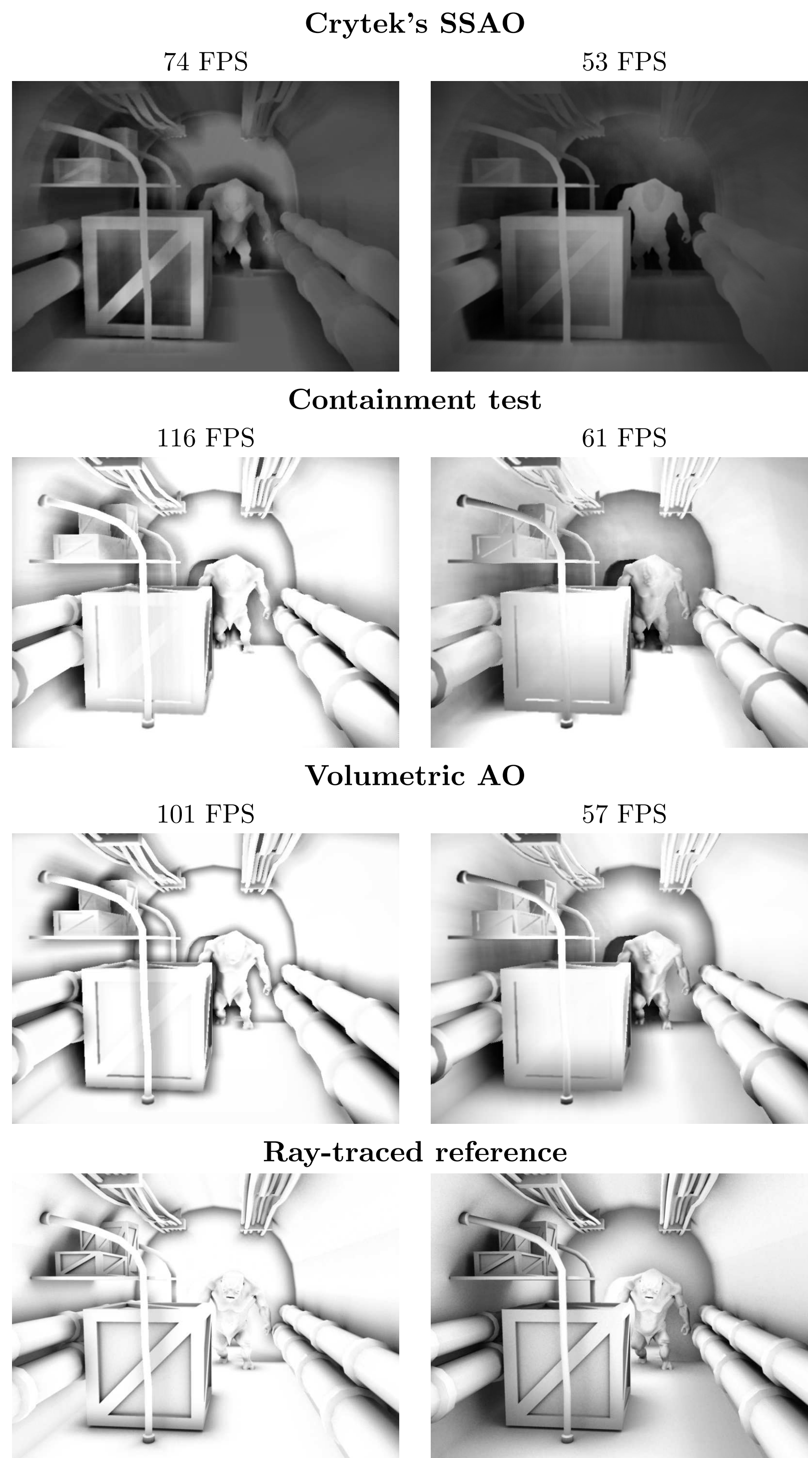


Figure 4.8: Comparison of Crytek's screen-space ambient occlusion, the containment test based method, volumetric ambient occlusion, and a reference that is obtained with ray-tracing. Images in the upper and lower rows are rendered with a smaller ($R = 3$) and a larger ($R = 9$) neighborhood, respectively. We used 32 samples per pixel in all cases. We incorporated membership function $\mu = (r/R)^3$ in the containment test based and in the ray-tracing methods, which is the same membership function that is implicitly used by volumetric ambient occlusion.

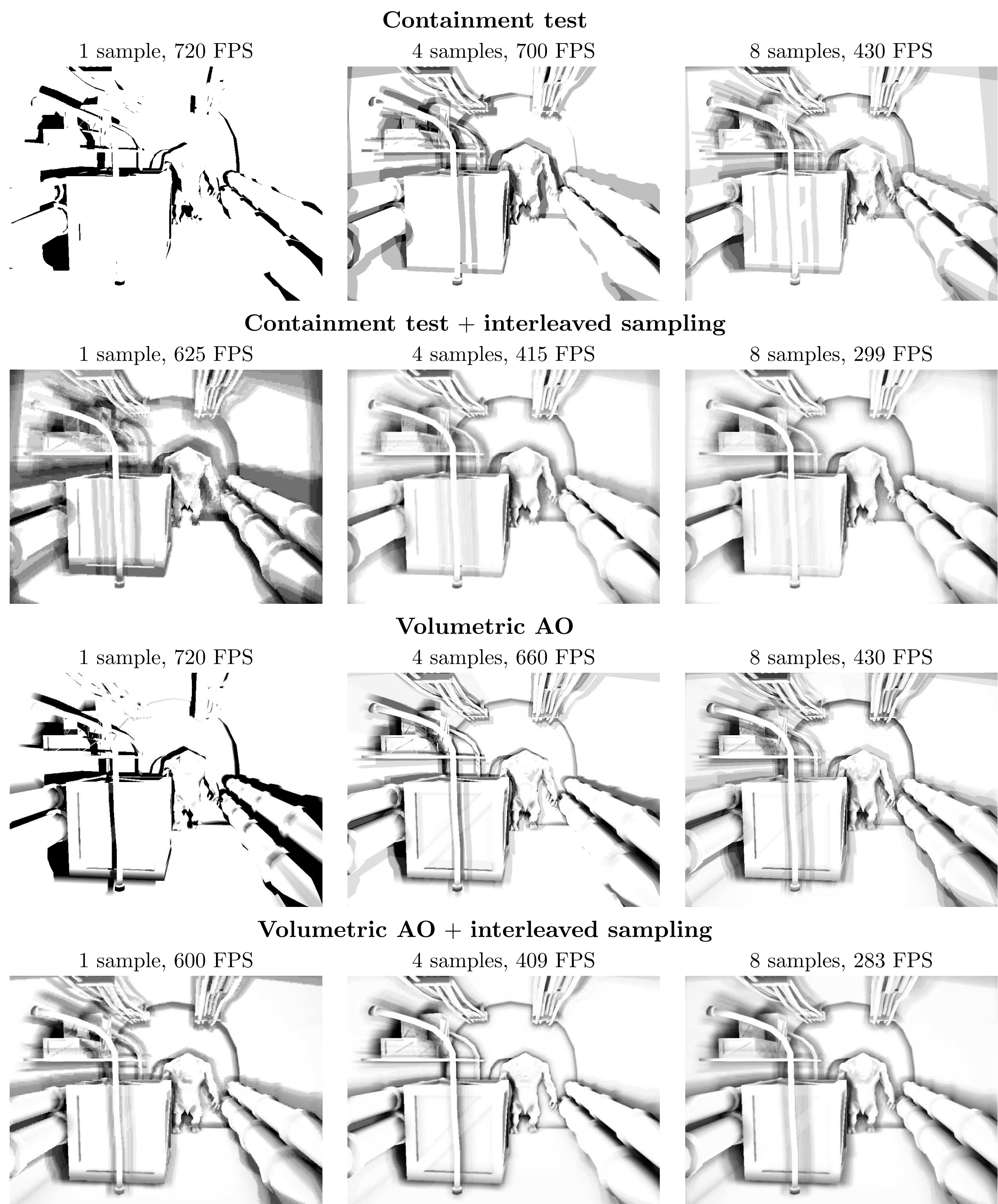
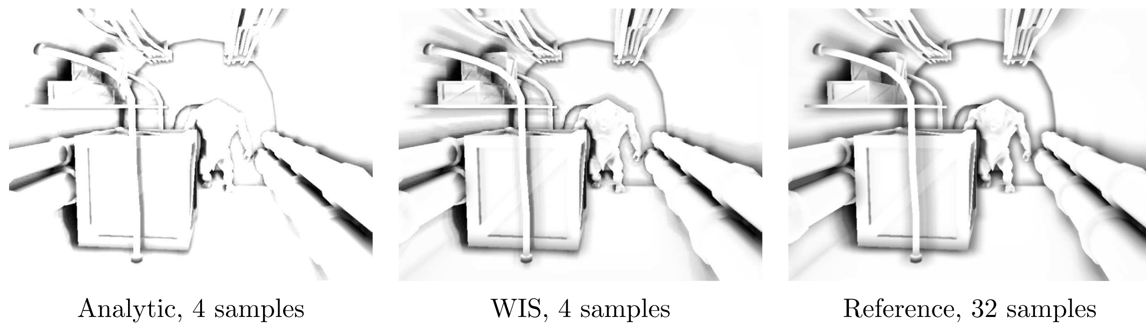


Figure 4.9: Comparison of the containment test based and the volumetric ambient occlusion (AO) using different numbers of samples per pixel and turning interleaved sampling on and off.

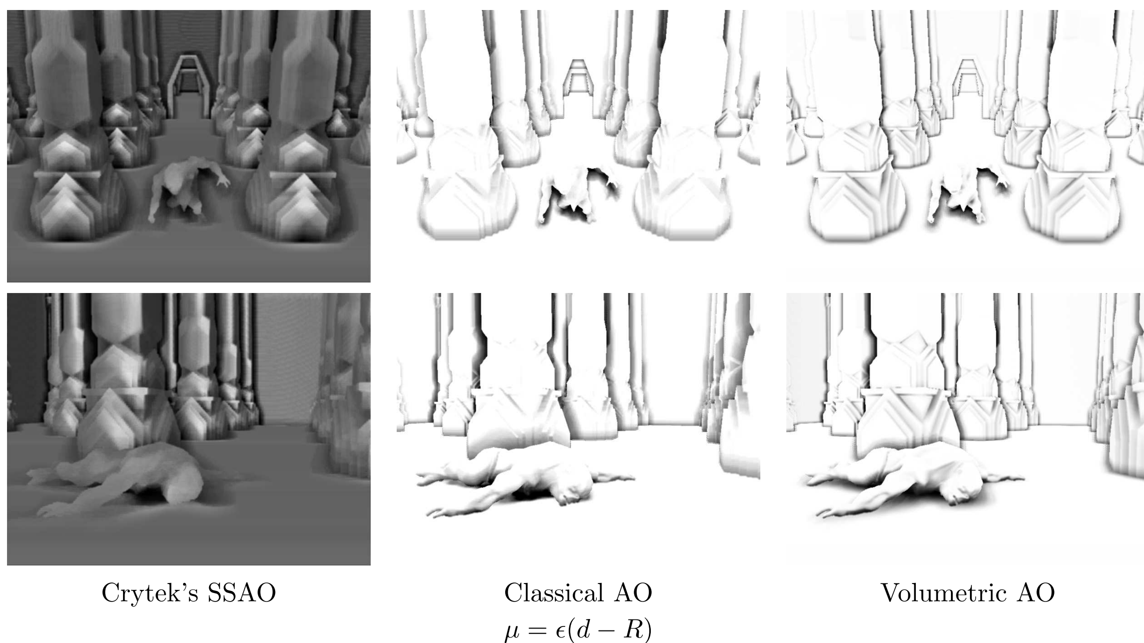
Figure 4.9 evaluates the performance-quality tradeoff for the containment test based and the volumetric ambient occlusion algorithms, and also shows the effect of interleaved sampling. As the volumetric approach evaluates a part of the integration analytically, its results are smoother when just a few samples per pixel are computed. The additional interleaved sampling helps eliminating sampling artifacts in all cases, but it also has a cost and mildly blurs the image. Figure 4.12 shows how the number of samples influence the performance of the screen space ambient occlusion algorithm.



Analytic, 4 samples | WIS, 4 samples | Reference, 32 samples

Figure 4.10: Effects of the proposed weighted importance sampling (WIS). Note that the image rendered with weighted importance sampling from 4 samples is much closer to the reference image generated from 32 samples.

Figure 4.10 shows the power of weighted importance sampling. We rendered the left image with the analytic formula (Equation 4.8) and the middle image with weighted importance sampling taking only 4 samples per pixel. Note that the image obtained with weighted importance sampling is closer to the reference rendered with 32 samples.



Crytek's SSAO | Classical AO $\mu = \epsilon(d - R)$ | Volumetric AO

Figure 4.11: Comparison of Crytek's screen space ambient occlusion, the containment test based ambient occlusion using the step membership function, and volumetric ambient occlusion. We used as few as 8 samples per pixel. The scene is built of 70138 triangles. Crytek's SSAO is a little slower than the other two algorithms that run at 250 FPS.

Figure 4.11 demonstrates the differences of Crytek's screen-space ambient occlusion, the classical ambient occlusion obtained with containment tests, and our new volumetric ambient occlusion. We took only $n = 8$ samples per pixel in all cases, and frame rates were also similar (about 250 FPS). As the volumetric ambient occlusion method needs to integrate a lower variation integrand, it provides smoother and better results than screen-space ambient occlusion or containment test based methods.

In our implementation the characteristic function may be a product of two characteristic functions. One separates the inner and outer parts according to the z-buffer. The other handles displacement mapped polygons considering them as height fields defined in tangent space. The z-buffer based classification is responsible for occlusions of other objects, while the height field based classification handles self-shadowing. Note that this way the ambient occlusion value can be obtained for displacement mapped surfaces even if the depth value is not modified in the fragment shader, and the accuracy problems of the z-buffer are also eliminated.

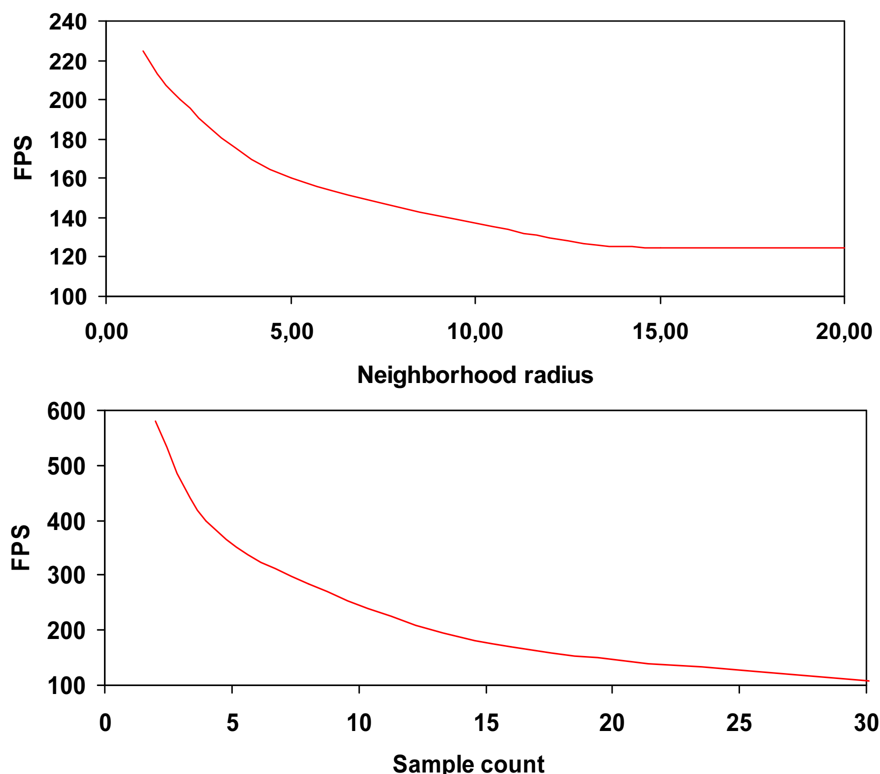


Figure 4.12: The effect of the neighborhood distance and the number of samples on performance.

Figure 4.13 compares images rendered with only the z-buffer based characteristic function and when height field occlusions are also considered, and also shows the final image due to environment map lighting. The volumetric ambient occlusion is computed with $n = 12$ samples per pixel. Note that our ambient occlusion computation is just slightly more expensive than environment lighting, but significantly enhances the details both on the level of macrostructure geometry and on the level of displacement maps.

4.3 Conclusions

This chapter proposed a fast method for the computation of ambient occlusion. The new approach is based on rewriting the ambient occlusion integral to evaluate the volume of the unoccluded part of the tangent sphere, and on its accurate approximation taking advantage of

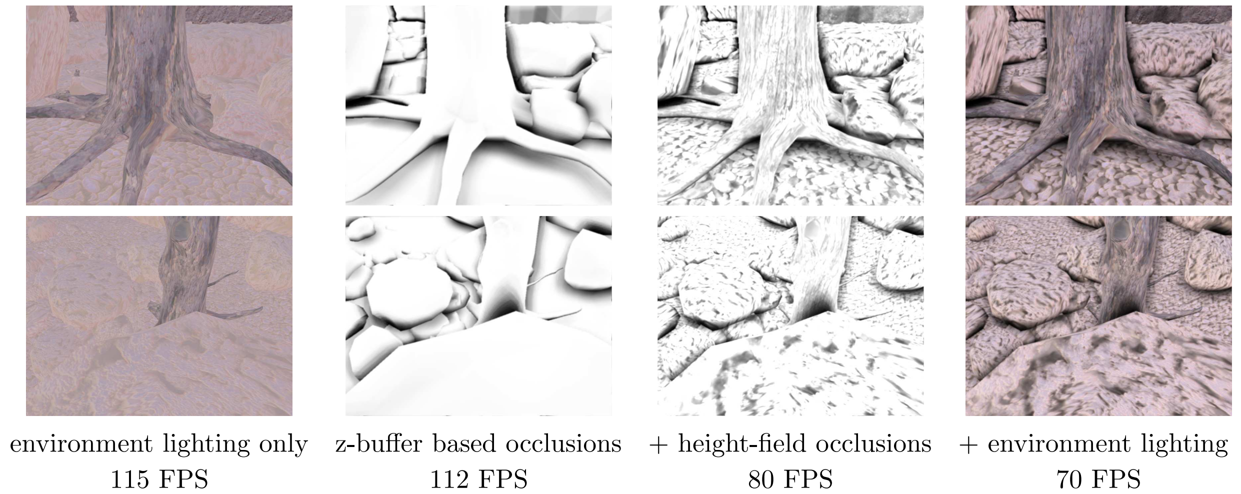


Figure 4.13: A tree with stones rendered by volumetric ambient occlusion taking 12 samples per pixel. The scene is defined by 109784 triangles.

uniform sequences transformed to mimic the new integrand. We also examined the correspondence of the new volumetric ambient occlusion formula to the classical obscurances or ambient occlusion. However, we have to emphasize that the volumetric ambient occlusion can also be considered as a new definition for the openness or accessibility of a point. The important advantage of the new formula is that it can be evaluated more accurately with the same number of samples, thus, it is more appropriate in real-time systems where low noise results are needed with just a few cheap samples. The method also works for dynamic geometry with dynamic height fields and displacement maps. It does not require pre-processing and runs at high frame rates, since it only needs as few as 8-10 samples per pixel.

Part II

Specular Material Rendering

Chapter 5

Robust specular reflections and refractions

Section 2.3.2 showed how to render **single** reflections and refractions using environment maps. In this chapter we propose a simple and robust ray tracing algorithm running on the GPU for tracing secondary rays, which can be used to compute real-time, **multiple** and refractions. The geometry is stored in **layered distance maps**. Note that this map is the sampled representation of the scene geometry. Thus, when a ray is traced, the intersection calculation can use this information instead of the triangular meshes. Since the layered distance map is generated on the fly and refreshed regularly, the method does not rely on preprocessing and can cope with dynamic scenes and deformable objects. The algorithm is simple to implement, efficient, and accurate enough to handle self reflections as well. In our approach ray tracing may support searches for the second, third, etc. hit points of the path, while the first point is identified by rasterization.

5.1 The new method of tracing secondary rays

The proposed ray tracing algorithm works on the geometry stored in layered distance maps. A single layer of these layered distance maps is a cube map, where a texel contains the material properties, the distance, and the normal vector of the point that is in the texel's direction. The material property is the reflected radiance for diffuse surfaces and the Fresnel factor at perpendicular illumination for specular reflectors. For refractors, the index of refraction is also stored as a material property. The distance is measured from the center of the cube map, which is called the **reference point**.

5.1.1 Generation of layered distance maps

The computation of distance maps is very similar to that of classical environment maps. The only difference is that not only the color, but the distance and the normal vector are also calculated and stored in additional cube maps.

Since the distance is a non linear function of the homogeneous coordinates of the points, correct results can be obtained only by letting the pixel shader compute the distance values.

The set of layers representing the scene could be obtained by **depth peeling** [Eve01, LWX06] in the general case. However, in our case correct depth order is not required between subsequent layers. Since the proposed searching step is more efficient if the distance values are smooth in a single layer, for the closer part of the scene, layers are organized to maintain the smoothness of the distance values instead of assigning the closest points to the same layer. On the other hand, occluded layers with larger distance values could not be visible from any points close to the reference point. Thus the reference point is put close to the reflective objects and the far part of the scene is replaced by a single distance map (Figure 5.1). To make a distinction between

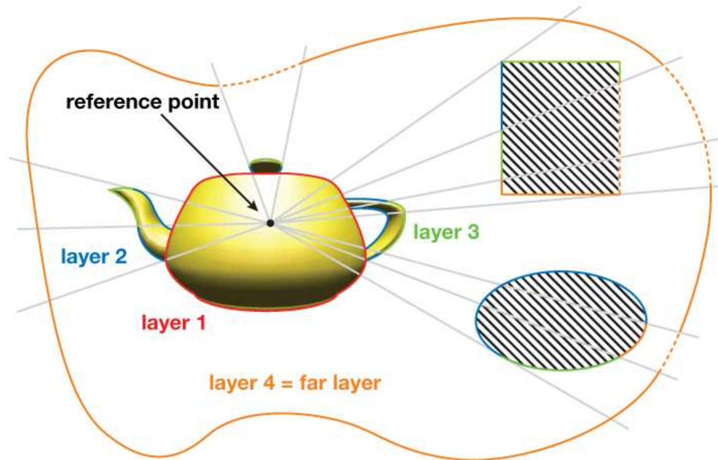


Figure 5.1: A layered distance map with 3+1 layers

close and far parts of the scene, objects close to the reference point are rendered using depth peeling into several layers, while objects far from the reference points are rasterized into a single distance map. As a consequence of this, incomplete layers may be generated, where a texel may also contain no visible point.

5.1.2 Ray-tracing of layered distance maps

The new algorithm for ray-tracing in layered distance map is based on the algorithms used for single layer ray-tracing described in section 2.3.2. However as layers can be incomplete, special considerations must be made. First a safe search method should be used to find an undershooting and an overshooting point, which provides the two endpoint of a continuous surface on which an unsafe but fast secant search can be performed.

Linear search

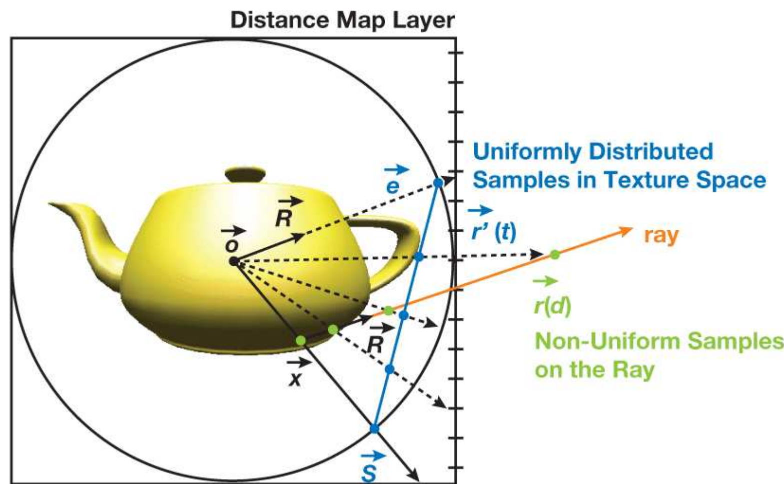


Figure 5.2: Cube map texels visited when marching along ray $\vec{x} + \vec{R}d$.

Let us use the notations on figure 5.2, and the ray to be traced with equation $\vec{x} + \vec{R}d$. The

basics of linear search in distance maps was described in section 2.3.2, however definition of the increments of ray parameter d needs special considerations since it is not worth checking the same sample many times while ignoring other samples. Unfortunately, making uniform steps on the ray does not guarantee that the texture space is uniformly sampled. As we get farther from the reference point, constant length steps on the ray correspond to decreasing length steps in texture space. This problem can be solved by marching along a line segment that looks identical to the ray from the reference point, except that its two end points are at the same distance [D2]. The end points of such a line segment can be obtained by projecting the start of the ray ($\vec{r}(0)$) and the end of the ray ($\vec{r}(\infty)$) onto a unit sphere, resulting in new start $\vec{s}$ and end point $\vec{e}$, respectively (Figure 5.2).

The intersection algorithm searches these texels, making uniform steps on the line segment of $\vec{s}$ and $\vec{e}$:

$$\vec{r}' = \vec{s}(1-t) + \vec{e}t, \quad t = 0, \Delta t, 2\Delta t, \ldots, 1.$$

The correspondence between ray parameter d and line parameter t can be found by projecting $\vec{r}'$ onto the ray, which leads to the following formula:

$$d(t) = \frac{|\vec{x}|}{|\vec{R}|} \frac{t}{1-t}.$$

This algorithm finds a pair of subsequent undershooting and overshooting points in a single layer, advancing on the ray making uniform steps Δt in texture space. Step size Δt is set proportionally to the length of line segment of $\vec{s}$ and $\vec{e}$, and also to the resolution of the cube map. By setting texel step `Dt` to be greater than the distance of two neighboring texels, we speed up the algorithm but also increase the probability of missing the reflection of a thin object. At a texel, the distance value is obtained from the alpha channel of the cube map. As linear search finds a pair of undershooting and overshooting points additional secant steps can be used to refine the result (see Section 2.3.2).

Acceleration with min-max distance values

To speed up the intersection search in a layer, we compute a minimum and a maximum distance value for each distance map. When a ray is traced, the ray is intersected with the spheres centered at the reference point and having radii equal to the minimum and maximum values. The two intersection points significantly reduce the ray space that needs to be marched. This process moves the start point s and end point e closer, so fewer marching steps would provide the same accuracy. See Figure 5.3.

5.2 Application to multiple reflections and refractions

Light may get reflected or refracted on an ideal reflector or refractor several times. If the hit point of a ray is again on the specular surface, then reflected or refracted rays need to be computed and ray-tracing should be continued, repeating the same algorithm recursively. Generally, searching multi-layer distance maps is equivalent to searching every layer with the discussed single layer algorithms and then selecting the closest intersection.

The set of layers representing the scene could be obtained by **depth peeling** [SKP98, Eve01, LWX06] in the general case. However, in many cases a much simpler strategy may also work.

If we simulate only multiple refractions on a single object, then the reference point can be put to the center of the refractor, and a single cube map layer can be assigned to the refractor surface, and one other for the environment, thus two layers can solve the occlusion problem (Figure **??**). On the other hand, in case of a reflecting surface, we may separate the specular surface from its environment, place the reference point close to the specular object, and assign one layer for the front faces of the specular surface, one for the back faces, and a third layer for the environment (Figure 5.6).

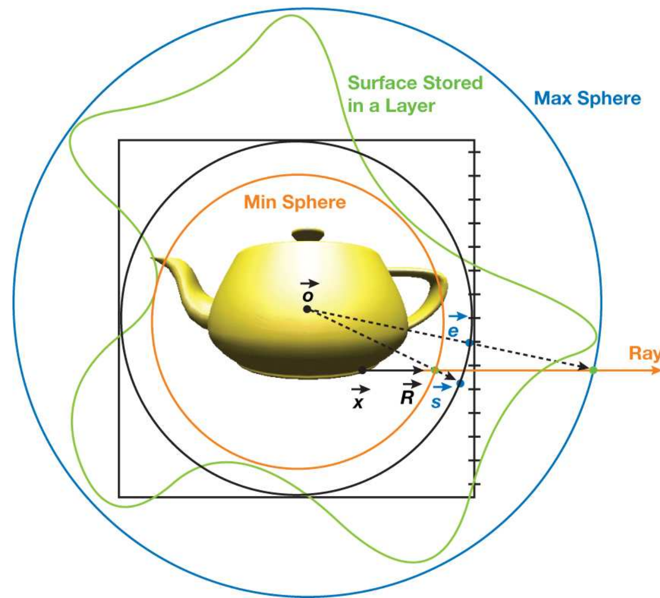


Figure 5.3: Min-max acceleration

The computation of the reflected or refracted ray requires the normal vector at the hit surface, the Fresnel function, and also the index of refraction in case of refractions. These attributes can also be stored in distance maps. We use two cube maps for each layer. The first cube map includes the material properties, i.e. the reflected color for diffuse surfaces or the Fresnel function at perpendicular illumination for specular surfaces, and the index of refraction. The second cube map stores the normal vectors and the distance values. To distinguish ideal reflectors, refractors, and diffuse surfaces, the sign of the index of refraction is checked. Negative, zero, and positive values indicate a reflector, a diffuse surface, and a refractor, respectively. Algorithm 3 shows the pseudo code of the recursive layered distance map ray tracing function supporting both reflections and refractions.

Algorithm 3 Ray-trace

```
for each cube map layer do
   Find undershooting and overshooting points with linear search
   Refine intersection point x_l with secant search
end for
Find closest intersection x = min(x_0, .., x_l)
Read normal and surface properties at x
if diffuse surface then
   return surface color
else
   if max ray depth reached then
      return zero
   else
      Calculate reflection direction Rl and refraction direction Rf
      Calculate Fresnel function F
      return F · RayTrace(x, Rl) + (1 − F) · RayTrace(x, Rf)
   end if
end if
```

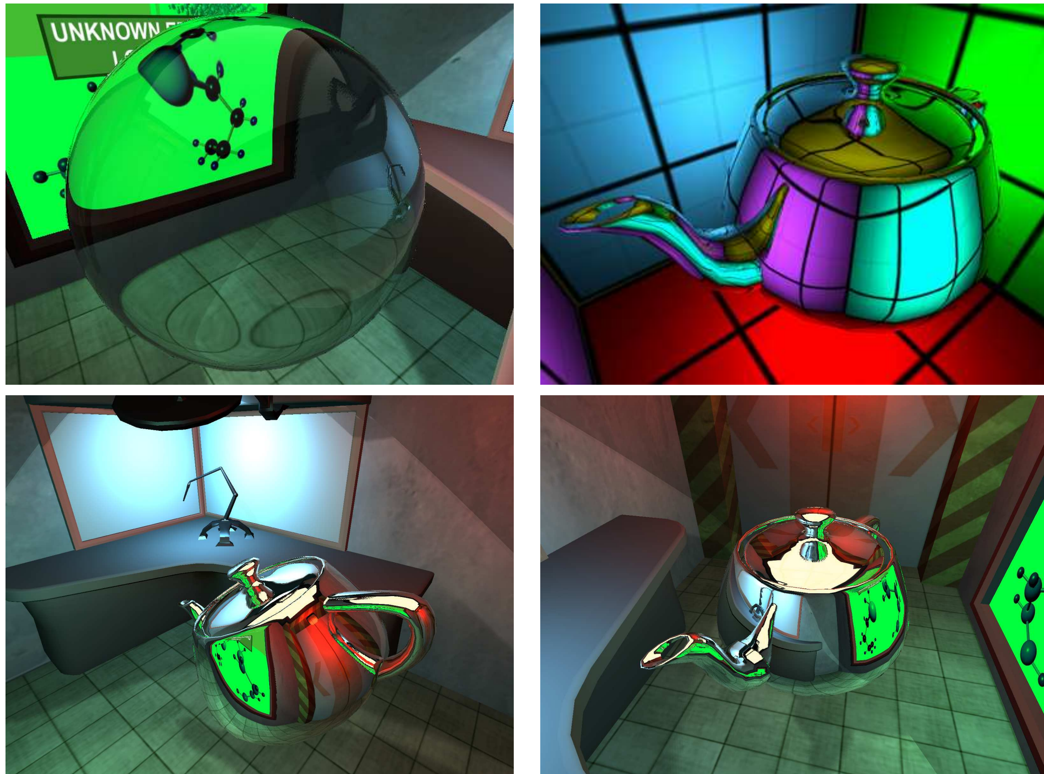


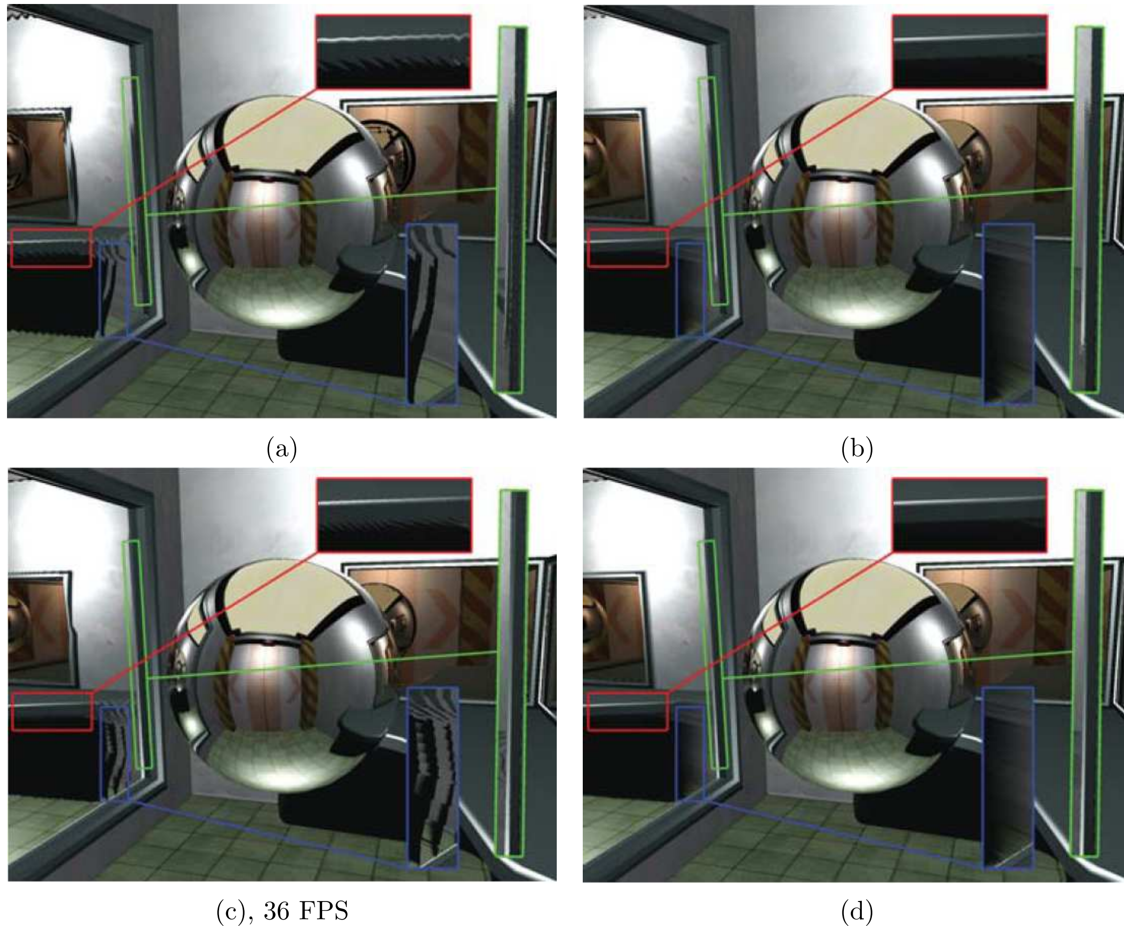
Figure 5.4: Multiple refractions and reflections when the maximum ray depth is four.

5.3 Results

Our algorithm was implemented in a DirectX 9 HLSL environment and tested on an NVIDIA GeForce 8800 GTX graphics card. To simplify the implementation, we represented the reflective objects by two layers (containing the front and the back faces, respectively) and rasterized all diffuse surfaces to the third layer. This simplification relieved us from implementing the depth-peeling process and maximized the number of layers to three. To handle more-complex reflective objects, we need to integrate the depth-peeling algorithm. All images were rendered at 800×600 resolution. The cube maps had $6 \times 512 \times 512$ resolution and were updated in every frame. We should carefully select the step size of the linear search and the iteration number of the secant search because they can significantly influence the image quality and the rendering speed. If we set the step size of the linear search greater than the distance of two neighboring texels of the distance map, we can speed up the algorithm but also increase the probability of missing the reflection of a thin object. If the geometry rendered into a layer is smooth (that is, it consists of larger polygonal faces), the linear search can take larger steps and the secant search is able to find exact solutions with just a few iterations. However, when the distance value in a layer varies abruptly, the linear search should take fine steps to avoid missing spikes of the distance map, which correspond to thin reflected objects.

Another source of undersampling artifacts is the limitation of the number of layers and of the resolution of the distance map. In reflections we may see parts of the scene that are coarsely, or not at all, represented in the distance maps because of occlusions and their grazing angle orientation for the center of the cube map. Figure 5.5 shows these artifacts and demonstrates how they can be reduced by appropriately setting the step size of the linear search and the iteration number of the secant search. Note how the stair-stepping artifacts are generally eliminated by adding secant steps. The thin objects zoomed-in on in the green and red frames require fine linear steps because, if they are missed, the later secant search is not always able to quickly

correct the error. Note that in Figure 5.5b, the secant search was more successful for the table object than for the mirror frame because the table occupies more texels in the distance maps. The aliasing of the reflection of the shadowed area below the table in the left mirror of Figures 5.5a and 5.5c, which is zoomed-in on in the blue frame, is caused by the limitation of distance map layers to three. This area is not represented in the layers, so not even the secant search can compute accurate reflections (Figures 5.5b and 5.5d). We can address these types of problems by increasing the number of layers.



(a) (b)

(c), 36 FPS (d)

Figure 5.5: Aliasing artifacts when the numbers of linear/secant steps are maximized to 15/1, 15/10, 80/1, and 80/10, respectively. (a) Max 15 linear and 1 secant steps. (b) Max 15 linear and 10 secant steps. (c) Max 80 linear and 1 secant steps. (d) Max 80 linear and 10 secant steps.

Figure 5.6 shows images of a reflective teapot in a box, rendered with the proposed method and with Maya (for comparison), limiting the maximum number of reflections in a path to one, two, and three, respectively. Note the high similarity between the GPU-rendered and software-rendered images. It is also worth mentioning that the frame rate decreases very little when the maximum number of reflections increases. This scenario indicates the excellent dynamic branching performance of the latest NVIDIA cards and also shows that early ray termination can improve performance, even in GPU solutions. Figure 5.7 compares multiple reflections and refractions in more complex scenes rendered by the proposed method and Maya. Finally, figure 5.8 shows images of a more complex scene containing a reflective sphere and a mirror, when the maximum ray depth variable is incremented from one to eight. The assignment of all diffuse surfaces to a single layer causes minor blurring artifacts where view-dependent occlusions occur (for example, in the reflection of the handle of the lamp on the mirroring sphere). We could

eliminate these artifacts by using more layers, but at the cost of a more complex implementation and lower rendering speed.

5.4 Conclusion

In this chapter, we presented a robust algorithm to trace rays in scenes represented by layered distance maps. The method is used to compute single and multiple reflections and refractions on the GPU. An important advantage of tracing rays in rasterized geometric representations instead of directly implementing classic ray tracing is that these methods can be integrated into game engines, benefit from visibility algorithms developed for games [WB05], and use the full potential of the latest graphics cards. This method is particularly effective if the geometry rendered into a layer is smooth (that is, consists of larger polygonal faces). In this case, the linear search can take larger steps and the secant search is able to find exact solutions, taking just a few iterations. However, when the distance value in a layer varies abruptly, the linear search should take fine steps to prevent missing thin, reflected objects. Note that a similar problem also arises in relief mapping, and multiple solutions have been developed [PM07, Don05]. In the future, we would consider integrating similar solutions into our technique.

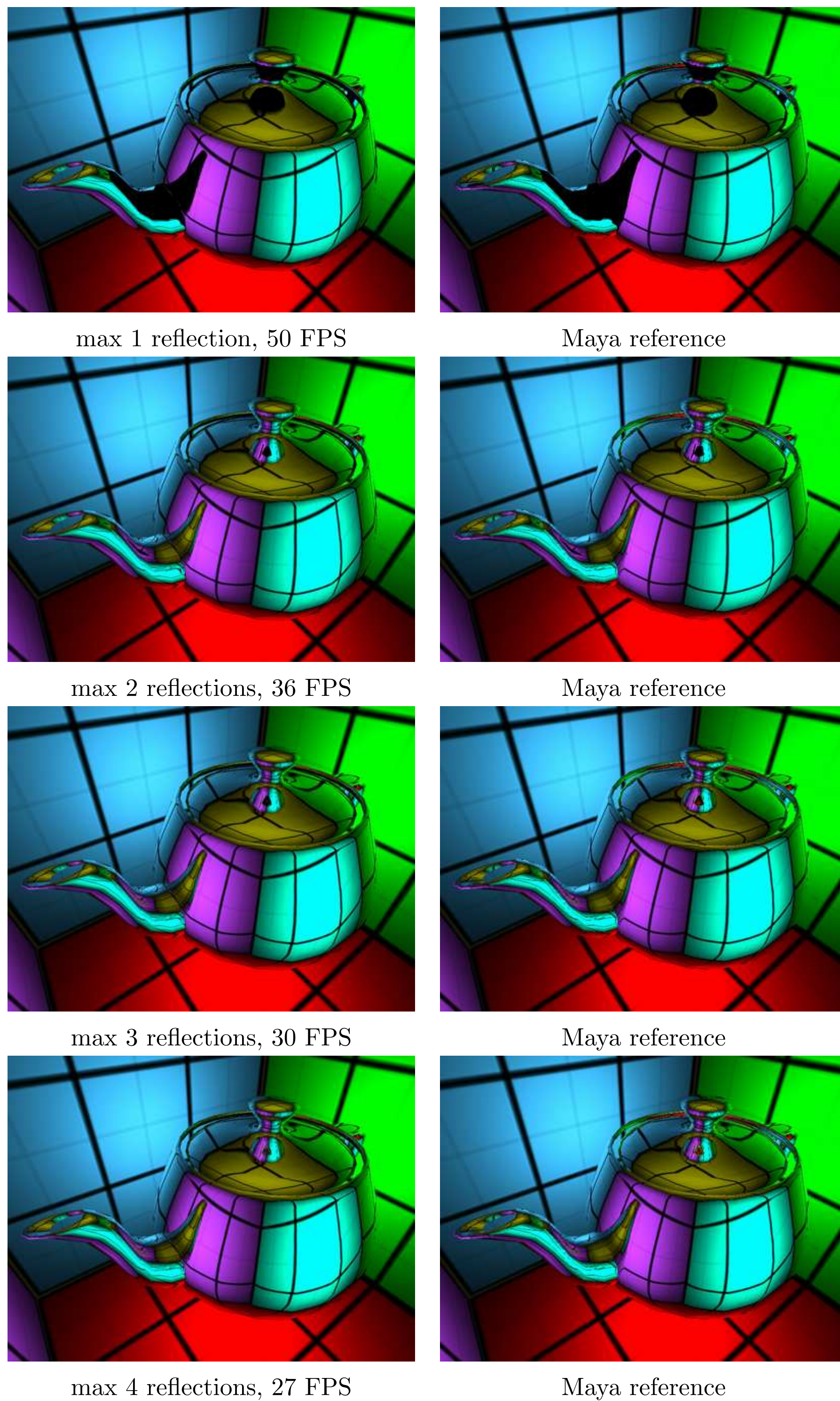


Figure 5.6: Single and multiple reflections on a teapot, and comparisons with the software ray tracer of the Maya 7 renderer

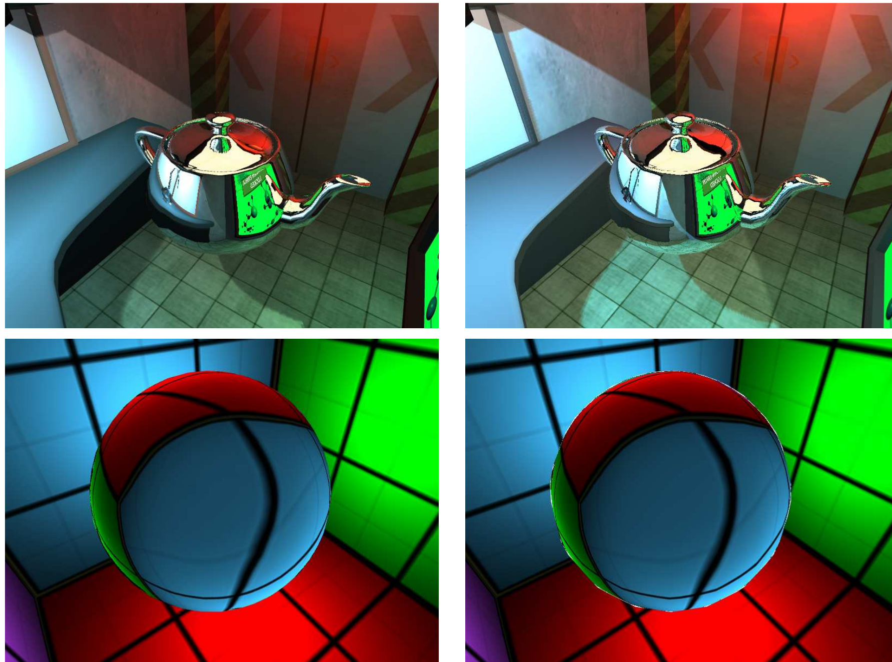


Figure 5.7: Comparing multiple reflections and refractions rendered by the proposed method (left) and Maya (right)

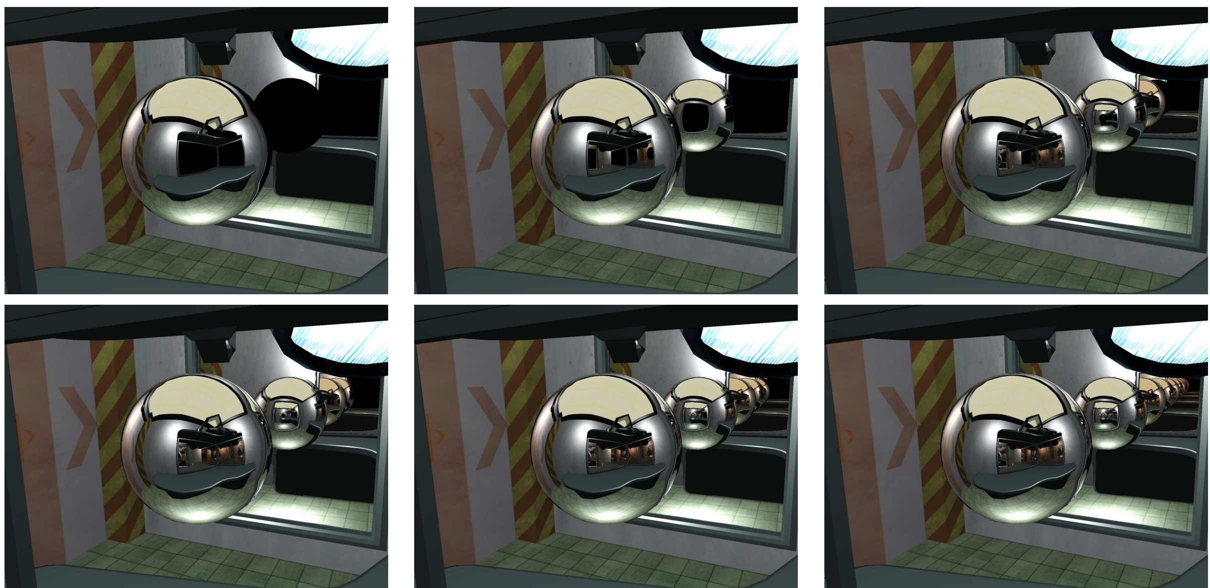


Figure 5.8: Multiple reflections incrementing the maximum depth variable from one up to eight

Chapter 6

Caustic triangles

Light paths starting at the light sources and visiting specular reflectors and refractors until they arrive at diffuse surfaces need to be simulated to create caustic effects. In GPU based caustic algorithms, the photon tracing part requires the implementation of some ray-tracing algorithm on the GPU. When a ray is traced, the complete scene representation needs to be processed. Since the shader processors of GPUs may access just the currently processed varying item (vertex, primitive or fragment), global parameters, and textures, this requirement can be met if the scene geometry is stored in textures.

Photon hits form a discrete representation of a continuous phenomenon. The distribution of these photon hits are very uneven, since the photon density follows the high frequency caustic patterns. Thus the reconstruction filter and the number of traced photons need to be carefully selected. If the number of traced photons is low, then undersampling effects occur. If this number is high, then the algorithm would be slow.

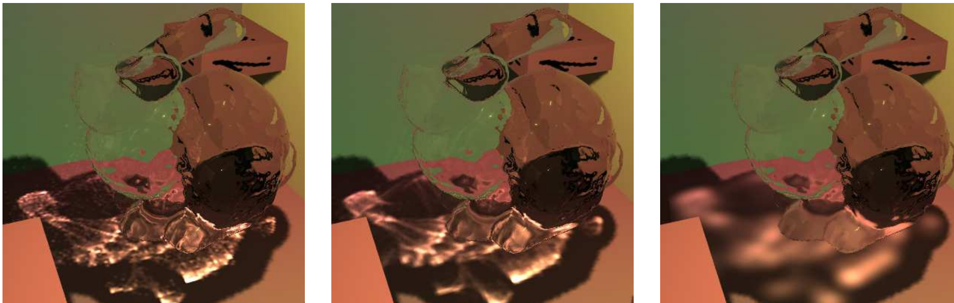


Figure 6.1: The problems of splatting. The left image was rendered with too small splatting filter, the right one with too large. To obtain the middle image, the splatting size was manually optimized.

Most of the GPU based algorithms reconstructed the continuous caustic pattern from discrete hits with photon **splatting** [WS03, SKALP05, SP07, WD06b, WD06a]. However, photon splatting has drawbacks. The size of splats should be carefully selected in order to preserve high frequency details but to eliminate dot patterns (Figure 6.1). Finding a uniformly good splat size is impossible due to the very uneven distribution of the photon hits. On the other hand, splatting does not take into account the orientation of the receiver surface, which results in unrealistically strong caustics when the surface is lit from grazing angles. The reconstruction problems of splatting can be reduced by adaptive size control [WD06a] or by hierarchical methods [Wym07], but these approaches either always work with the largest splat size or with multiple hierarchical levels, which reduces their rendering speed.

In addition to splatting, another possibility for reconstruction is to take three neighboring hits, assume that they form a caustic triangle or caustic beam, and this beam is intersected

with the caustic surfaces. Evolving on an original idea from Nishita and Nakamae [NN94] for rendering underwater caustics based on the beam-tracing approach, Iwasaki, Dobashi and Nishita [IDN02, IDN03] developed methods for the fast rendering of refractive and reflective caustics due to water surfaces, which were subdivided into triangular meshes. At each water surface mesh vertex, the refracted vector is computed. They call the volume defined by sweeping the reflected or refracted vector at each vertex of a caustic generator triangle the illumination volume. Caustics patterns are displayed by drawing the intersection areas between all of the illumination volumes and the receiver surfaces, and by accumulating the intensities of light reaching the intersection area. Unfortunately, this method treated neither shadows nor the case of warped volumes which can occur in beam tracing Ernst et al. [EAMJ05] solved some of these problems and also presented a caustic intensity interpolation scheme to reduce aliasing, which resulted in smoother caustics. However, this algorithm also ignored occlusions, so was unable to obtain shadows.

To attack the problems of previous GPU based methods, we propose a new algorithm that

- stores photon hits and reconstructs caustic patterns in the cube map space of the caustic generator and takes into account visibility information when projecting reconstructed caustic patterns onto the surfaces in order to avoid light leaks,
- renders caustic triangles instead of splatting to reduce undersampling artifacts.

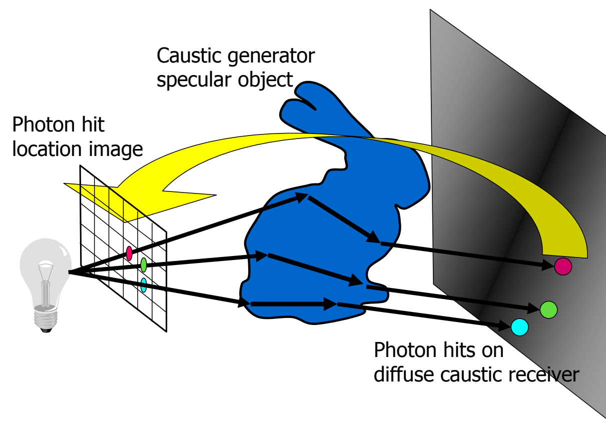


Figure 6.2: Caustics generation renders into the photon hit location image where each pixel stores the location of the photon hit with respect to some coordinate system.

6.1 The new caustics generation algorithm

As concluded during the review of the previous work, a critical issue of all caustic algorithms is the selection of the coordinate system where photon hits are stored and the definition of the algorithm that reconstructs the continuous caustic pattern from these discrete hits.

In order to support artifact free filtering and reconstruction, in this space two points should be close if they are seen in similar directions from the point of view of the caustic generator object. Furthermore, we need a way to separate those points which can be hit by rays leaving the caustic generator objects from those points which cannot be hit due to occlusions.

Let us note that the coordinate system of the distance map used to trace rays leaving the caustic generator meets these requirements. Points that are represented in neighboring texels are seen in similar direction from the center of the cube map, and are potentially affected by neighboring photon paths. On the other hand, comparing the distance of the point from the center of the cube map to the stored distance of the cube map layers, we can decide whether a point is occluded or visible from the center of the caustic generator object.

A point is identified by the direction in which it is visible from the reference point, i.e. the texel of the cube map, and also by the distance from the reference point. An appropriate neighborhood for filtering is defined by those points that are projected onto neighboring texels taking the reference point as the center of projection, and having similar distances from the reference point as stored in the distance map. Note that this approach is very similar to classical shadow mapping and successfully eliminates light leaks.

The caustic generation algorithm has multiple passes. See Figure 6.3. Firstly, the photon tracing pass builds the photon hit location image, followed by the caustic reconstruction pass which reconstructs the caustic patterns on the cube map face of the distance map and, finally, the camera pass, which projects the caustic patterns onto the caustic receiver surfaces.

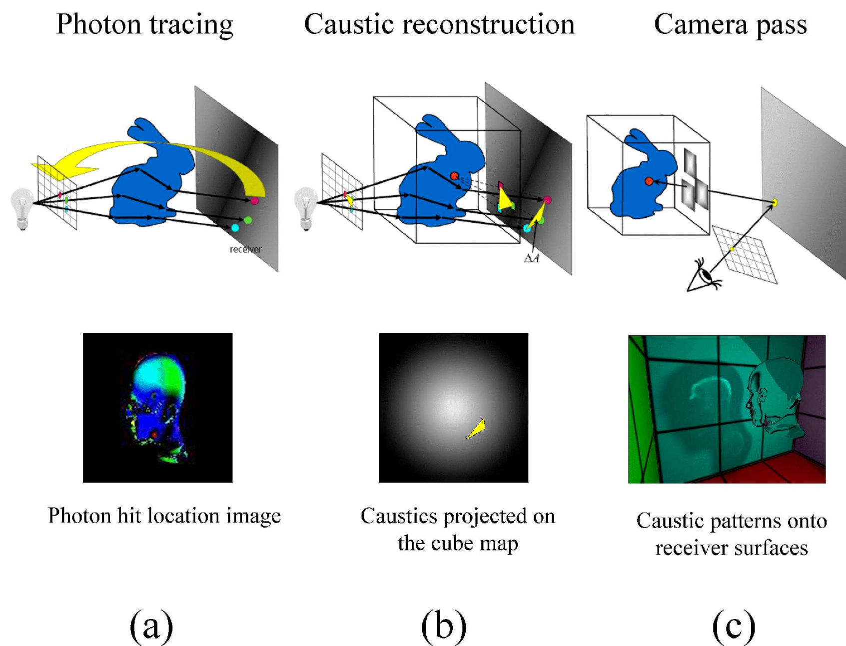


Figure 6.3: Overview of the new caustics generation algorithm: Firstly, the Photon tracing pass (a), followed by the Caustic reconstruction pass (b) and finally, the Camera pass(c).

6.1.1 Photon tracing pass

The caustic generator algorithm first identifies those caustic generators that are visible from the light sources and may cast caustics onto caustic receivers that are visible from the camera. Theoretically, one layered distance map is enough for the whole scene for ray-tracing, but due to accuracy reasons we generate a separate distance map for each of these caustic generators if they are far from each other.

Then each caustics generator is rendered from the point of view of each light source (Figure 6.2). The shaders are set so that the vertex shader transforms the caustic generator to clipping space, and also to the coordinate system of the cube map by first applying the modeling transform, then translating to the reference point. When the fragment shader processes a point on the caustic generator, the light ray defined the light source position and the current pixel is traced, possibly multiple times, until a diffuse or glossy surface is found. The direction of this point with respect to the center of the cube map and the power of the photon are written into the processed fragment. The power is obtained by multiplying the power going through a pixel of the light's camera and the Fresnel factors along the path (or the complement of the Fresnel if refraction happens). To distinguish fragments where the caustic generator is not visible,

the render target is initialized with negative power values.

6.1.2 Caustic reconstruction pass

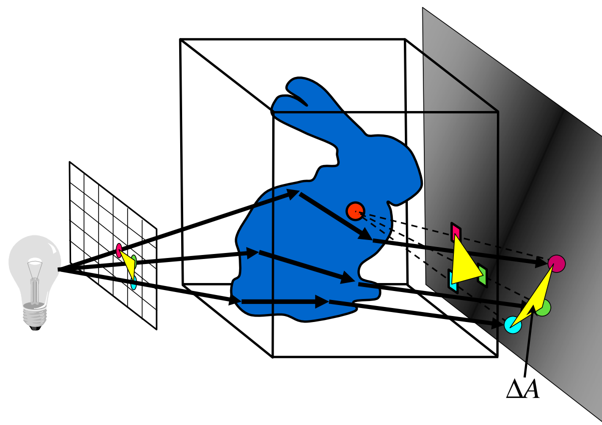


Figure 6.4: Computation of the irradiance at the vertices of caustic triangles. The same caustic triangle is shown on the photon hit location image, on the light cube map face, and projected onto the caustic receiver.

We propose caustic patterns to be reconstructed on the cube map face of the distance map, and then projecting the caustic patterns onto the caustic receiver surfaces (Figure 6.4).

During reconstruction neighboring photon hits are found in adjacent texels of the photon hit location image. These texels contain all information (direction and distance) to determine the vertices of the caustic triangle. A caustic triangle is drawn onto the surface of the cube map, where the vertex locations are set as the directions stored in the photon hit location image. We send two times as many triangles as pixels the photon hit location image has (Figure 6.5) down the pipeline. We can avoid the expensive GPU to CPU transfer of this image, if vertices are sent with dummy coordinates, or with photon hit location image coordinates, and the vertex shader sets the world or cube map space coordinates according to the content of the photon hit location image. This operation requires at least Shader Model 3 GPUs that allow the vertex shader to access textures. If a pixel of the photon hit location image has negative power, i.e. it is invalid since the caustic generator is not visible in this pixel, then the corresponding vertex is placed outside the clipping region. This way we can exploit the clipping hardware to eliminate invalid triangles and vertices.

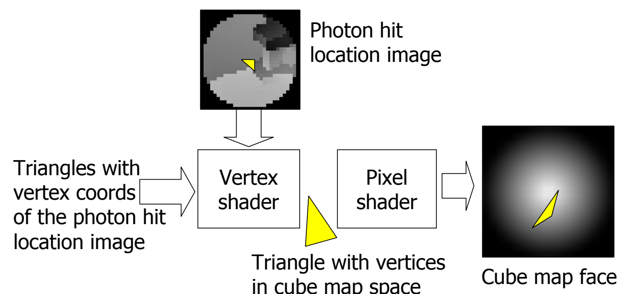


Figure 6.5: Rendering to the light cube map.

The irradiance at a vertex can be computed as the ratio of power $\Delta\Phi$ arriving at neighborhood of area ΔA, i.e. as $I = \Delta\Phi/\Delta A$. Area ΔA is approximated from the areas of projected caustic triangles adjacent to this vertex (Figure 6.4). Since we use Gouraud shading, i.e. linear

interpolation between vertices, power distribution area ΔA is computed as the half of the total areas of triangles attached to this vertex.

The compositing operator should be set to "add" since the contributions of different photon hits should be added.

The resulting cube map with irradiance values in its texels is called the **light cube map**(or **caustic cube map**), which can be considered as a direction dependent point light source that is responsible for adding the caustic contribution.

6.1.3 Camera pass

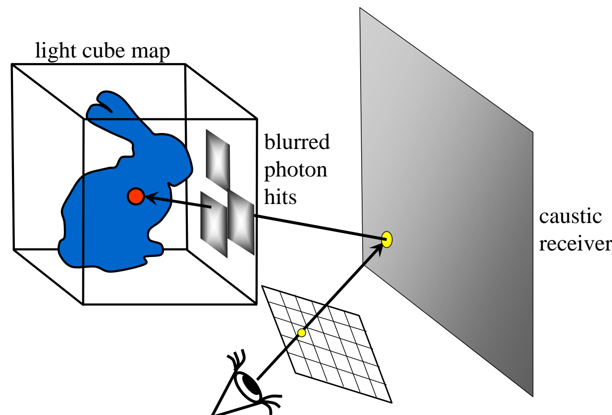


Figure 6.6: Light projections in the final camera pass.

During the final camera pass, the light cube map acts as an additional light source (Figure 6.6). The texels of the cube map already have distance information, thus this cube map is also a shadow map. When a point is processed by the fragment shader during final gathering, the direction between the center of the light cube map and the shaded point is obtained. The light cube map is looked up in this direction. If the distance stored in the light cube map texel is similar to the distance to the shaded point, then the irradiance stored in the texel is multiplied by the BRDF of the shaded point and result is added to the reflection of other light sources. Note that not only the irradiance but also the direction of the illumination is available during this computation thus glossy BRDFs can also be correctly processed. Algorithm 4 shows the pseudo code of the entire caustics generation process.

6.2 Optimizations

In the presented method the area computation of a triangle is repeated if a vertex is reused in other triangles. So we can implement a separate pass to render just a single quad at the resolution of the photon hit location image. This pass computes the area of triangles adjacent to a single vertex.

This step can also be utilized to execute hierarchical reconstruction similarly to [Wym07], but unlike in Wyman's approach not the photon hits but caustic triangles are merged together. The hierarchical reconstruction can address the oversampling problem, i.e. that caustic triangles may be very small in the focus (i.e. smaller than a texel of the light cube map or a pixel on the screen). It is worth combining small triangles into a single triangle adding up their power, using the geometry shader of **Shader Model 4** GPUs. We consider the implementation of this mipmap based approach as an important future work.

Another important area of further study is the representation of high frequency caustics into the process: now, represented caustics are limited by the resolution of both the photon hit location image and the cube map surface. Although the usage of triangles built from the

Algorithm 4 Caustic triangles

```
for each caustic caster do
  Create cube map storing distance values from caster object center
  Create photon hit location image
      Place camera in light source and orient to caster
      Use ray-tracing to identify hit point location
      Write hit point direction from caster center
  Create caustic cube map placed in caster center
      Setup triangles for adjacent texels in photon location image
      Read vertex positions from photon location image
      Place vertices with invalid photon hits outside clipping region
      Calculate adjacent triangle areas
      Calculate vertex intensity
      Draw triangle with additive blending
end for
for each caustic receiver do
  for each caustic caster do
    Calculate direction between shaded point and caster center
    Read distance stored in distance cube map
    Compare with distance of the shaded point
    if distances are similar then
      Read caustics intensity from caustic cube map
      Add caustics intensity to shaded color
    end if
  end for
end for
```

photon hit location image generates patterns without noticeable artifacts, small details could be lost. A possible workaround this problem is to employ an adaptive or hierarchical solution, as mentioned before. In our experiments, we have not seen any serious problem with respect to the dependency on the cubemap resolution (we used maps of $6 \times 512 \times 512$, see below).

6.3 Results

The discussed algorithm has been implemented in DirectX/HLSL environment and integrated into the Ogre3D game engine. Figure 6.8 shows the photon hit location image, the projected photon hits visualized by point primitives, and the result with caustic triangles. We used a 128×128 pixel resolution photon hit location image, and a Cubemap of $6 \times 512 \times 512$. Figure 6.7 shows a glass head in the “laboratory” scene. Since the caustic cube map can be looked up not only when the primary ray hits a surface but also when secondary reflective or refractive rays are traced, the proposed method can produce the reflections or refractions of caustics as well. Note that even with shadow, reflection, and refraction computation, the method runs at 60 FPS on an NV6800GT GPU. Table 6.1 shows speed measurements with different photon hit location image resolution and caustic cube map resolution settings. We should note that on recent hardware (like the NVidia GTX 260) we can count on a tripled speedup of the algorithm.

6.4 Conclusions

We proposed a robust algorithm to compute caustics by tracing rays in distance maps. Unlike splatting-based methods our caustic algorithm does not exhibit problems depending on a splatting filter since caustic triangles provide continuous patterns without overblurring even at

Caustic cube map resolution	**Photon hit location image resolution**			
	32×32	64×64	128×128	256×256
64×64	300 FPS	175 FPS	63 FPS	16FPS
128×128	290 FPS	170 FPS	62 FPS	16FPS
256×256	280 FPS	165 FPS	60 FPS	16FPS
512×512	260 FPS	155 FPS	60 FPS	16FPS

Table 6.1: *The effect of photon hit location image resolution and caustic cube map resolution on performance (NV6800GT).*

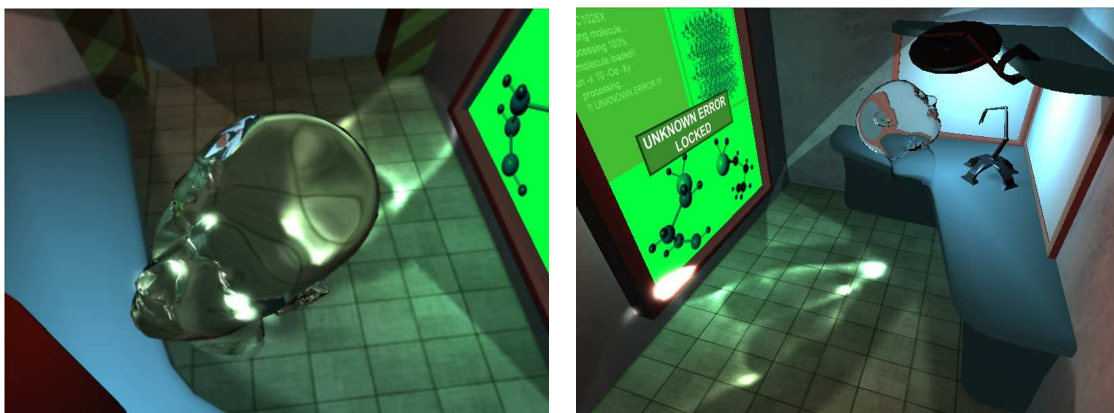
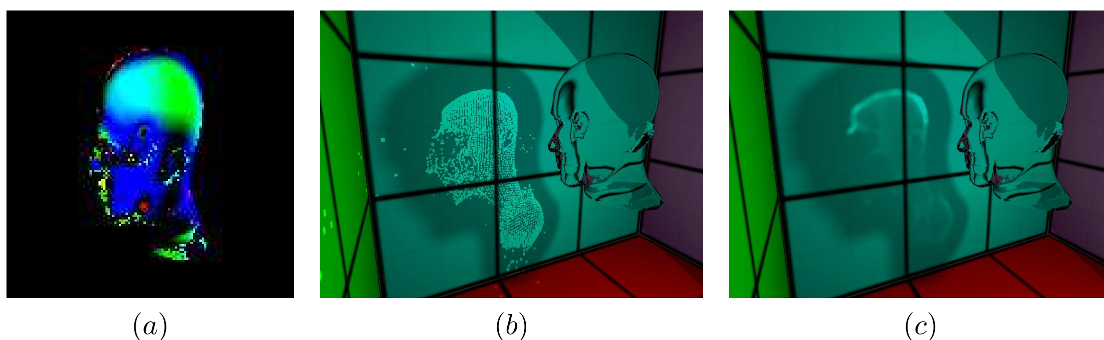


Figure 6.7: Real-time caustics caused by a glass head and rendered by the proposed method rendered at 60 FPS on an NV6800GT GPU.



(*a*) (*b*) (*c*)

Figure 6.8: (a) The 128×128 resolution photon hit location image, (b) photon hits rendered as points, and (c) the caustics as a collection of transparent triangles rendered at 80 FPS on an NV6800GT GPU.

moderate resolutions. Also, unlike [EAMJ05], the proposed approach can correctly take into account occlusions and warped volumes, which allows accurate shadow computations. On the other hand, the method presented here inherits some of the limitations of distance map based ray tracing: the number of search iterations should be carefully chosen to guarantee that artifacts in the ray tracing stage are minimized.

Part III

Participating Media Rendering

Chapter 7

Spherical billboards

Particle system rendering methods usually splat particles onto the screen [Ree83, WLMK02]. Splatting substitutes particles with semi-transparent, camera-aligned rectangles, called **billboards** [Sch95].

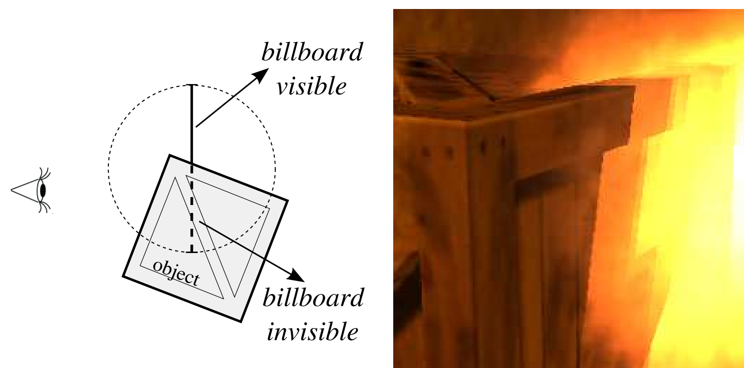


Figure 7.1: Billboard clipping artifact. When the billboard rectangle intersects an opaque object, transparency becomes spatially discontinuous.

Billboards are planar rectangles having no extension along one dimension. This can cause artifacts when billboards intersect opaque objects making the intersection of the billboard plane and the object clearly noticeable (Figure 7.1). The core of this **clipping artifact** is that a billboard fades those objects that are behind it according to its transparency as if the object were fully behind the sphere of the particle. However, those objects that are in front of the billboard plane are not faded at all, thus transparency changes abruptly at the object–billboard intersection.

On the other hand, when the camera moves into the media, billboards also cause **popping artifacts**. In this case, the billboard is either behind or in front of the front clipping plane, and the transition between the two stages is instantaneous. The former case corresponds to a fully visible, while the latter to a fully invisible particle, which results in an abrupt change during animation (Figure 7.2).

The issue of billboard clipping artifacts has been addressed in [HL01, Har02] where the splitting of the intersected billboards is proposed. Instead of using the z-buffer to identify the visible fragments, a back to front rendering is executed, which draws two billboards instead of a single intersected billboard, one before, and the other after the object rendering. While this approach eliminates artifacts in case of a single object, it gets unmanageable for many included objects.

If billboard pixels are either fully visible or fully opaque, then another possibility exists to mix billboards and conventional 3D objects. To avoid incorrect clipping, billboards are equipped with per pixel depth information as happens, for example in nailboards [Sch97] and 2.5D impostors

Figure 7.2: Billboard popping artifact. Where the billboard gets to the other side of the front clipping plane, the transparency is discontinuous in time (the figure shows two adjacent frames in an animation).

[Szi05, J2].

In this chapter we propose a novel solution for including objects into the participating medium without billboard clipping and popping artifacts. The basic idea of spherical billboards is introduced in Section 7.1. Unlike nailboards, the proposed method can also handle semi transparent billboards needed for **natural phenomena**, such as fire, smoke, clouds, and explosions. Instead of associating depth textures to billboards, we obtain two depth values analytically, one for the front, and another one for the back surfaces, and compute the opacity according to the real distance the light travels inside the particle.

The proposed idea is used then to render realistic explosions in real-time in Section 7.2. In our explosion rendering system fire and smoke particles are animated and colored in a physically plausible way similarly to [NFJ02, Ngu04].

7.1 The new method using spherical billboards

Billboard clipping artifacts are solved by calculating the real path length a light ray travels inside a given particle since this length determines the opacity value to be used during rendering. The traveled distance is obtained from the spherical geometry of particles instead of assuming that a particle can be represented by a planar rectangle. However, in order to keep the implementation simple and fast, we still send the particles through the rendering pipeline as quadrilateral primitives, and take into account the spherical shape only during fragment processing.

To find out where opaque objects are during particle rendering, first these objects are drawn and the resulting **depth buffer** storing camera space z coordinates is saved in a **texture**.

Having rendered the opaque objects, particle systems are processed. The particles are rendered as quads perpendicular to axis z of the camera coordinate system. These quads are placed at the farthest point of the particle sphere to avoid unwanted front plane clipping. Disabling depth test is also needed to eliminate incorrect object–billboard clipping.

When rendering a fragment of the particle, we compute the interval the ray travels inside the particle sphere in camera space. This interval is obtained considering the saved depth values of opaque objects and the camera's front clipping plane distance. From the resulting interval we can compute the opacity for each fragment in such a way that both fully and partially visible particles are displayed correctly, giving the illusion of a volumetric medium. During opacity computation we assume that the density is uniform inside a particle sphere.

For the formal discussion of the processing of a single particle, let us use the notations of Figure 7.3. We consider the fragment processing of a particle of center $\vec{P} = (x_p, y_p, z_p)$, which is rendered as a quad perpendicular to axis z. Suppose that the current fragment corresponds

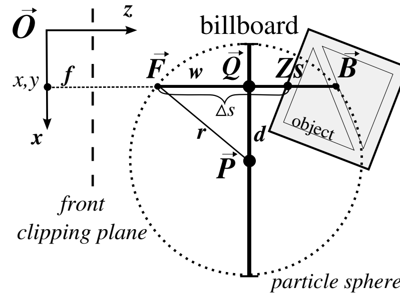


Figure 7.3: Computation of length Δs the ray segment travels inside a particle sphere in camera space.

to the visibility ray cast through point $\vec{Q} = (x_q, y_q, z_q)$ of the quadrilateral. Although visibility rays start in the origin of the camera coordinate system and thus form a perspective bundle, for the sake of simplicity, we consider them as being parallel with axis z. This approximation is acceptable if the perspective distortion is not too strong. If the visibility ray is parallel with axis z, then the z coordinates of $\vec{P}$ and $\vec{Q}$ are identical, i.e. $z_q = z_p$.

The radius of the particle sphere is denoted by r. The distance between the ray and the particle center is $d = \sqrt{(x - x_p)^2 + (y - y_p)^2}$. The closest and the farthest points of the particle sphere on the ray from the camera are $\vec{F}$ and $\vec{B}$, respectively. The distances of these points from the camera can be obtained as

$$|\vec{F}| \approx z_p - w, \qquad |\vec{B}| \approx z_p + w, \tag{7.1}$$

where

$$w = \sqrt{r^2 - d^2}. \tag{7.2}$$

The ray travels inside the particle in interval $[|\vec{F}|, |\vec{B}|]$.

Taking into account the front clipping plane and the depth values of opaque objects, these distances may be modified. First to eliminate popping artifacts, we should ensure that $|\vec{F}|$ is not smaller than the front clipping plane distance f, thus the distance the ray travels in the particle before reaching the front plane is not be included. Secondly we should also ensure that $|\vec{B}|$ is not greater than Z_s which is the stored object depth at the given pixel, thus the distance traveled inside the object is not considered.

From these modified distances we can obtain the real length the ray travels in the particle:

$$\Delta s = min(Z_s, |\vec{B}|) - max(f, |\vec{F}|). \tag{7.3}$$

7.1.1 "Gouraud shading" for spherical billboards

Assuming that the density is homogeneous inside a particle and using equation **??** to obtain the respective opacity value correspond to the piece-wise constant finite-element approximation. While constant finite-elements might be acceptable from the point of view of numerical precision, their application results in annoying visual artifacts.

Figure 7.4 depicts the accumulated density ($\int_{\Delta s_j} \tau(s) \mathrm{d}s$) and the respective opacity as a function of ray–particle distance d, assuming constant finite-elements. Note that at the contour of the particle sphere ($d = 1$) the accumulated density and the opacity become zero, but they do not diminish smoothly. The accumulated density has a significant derivative at this point, which makes the contour of the particle sphere clearly visible for the human observer.

This artifact can be eliminated if we use piece-wise linear rather than piece-wise constant finite-elements, that is, the density is supposed to be linearly decreasing with the distance

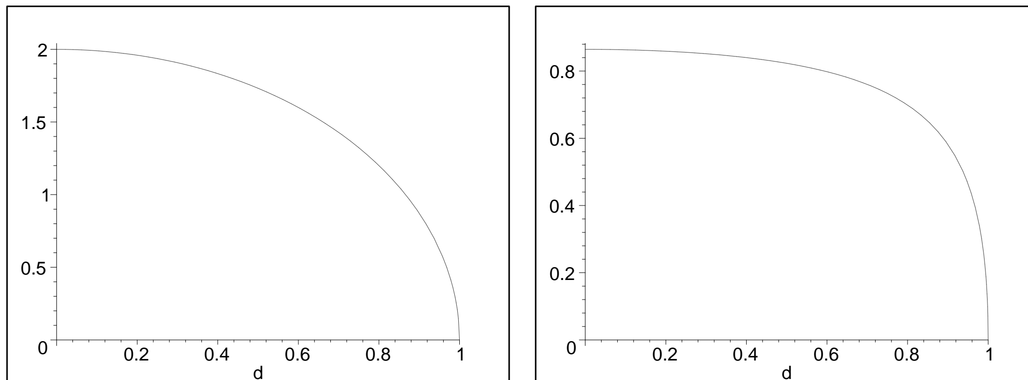


Figure 7.4: The accumulated density of a ray (left) and its seen opacity (right) as the function of the distance of the ray and the center in a unit sphere with constant, unit density.

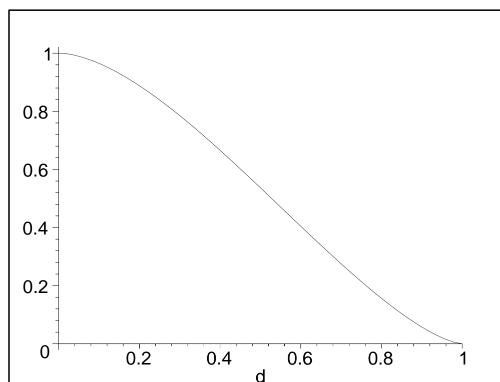


Figure 7.5: The accumulated density of a ray as the function of the distance of the ray and the center in a unit sphere with density function linearly decreasing with the distance from the particle center.

from the particle center. The accumulated density computed with this assumption is shown by Figure 7.5. Note that this function is very close to a linear function, thus the effects of piece-wise linear finite elements can be approximated by modulating the accumulated density by this linear function. It means that we use the following formula to obtain the opacity of particle j:

$$\alpha_j \approx 1 - e^{-\tau_j (1-d/r_j)\Delta s_j} \tag{7.4}$$

, where d is the distance between the ray and the particle center, and r_j is the radius of this particle. Note that this approach is very similar to the trick applied in radiosity methods. While computing the patch radiosities piece-wise constant finite-elements are used, but the final image is presented with Gouraud shading, which corresponds to linear filtering.

7.1.2 GPU Implementation of spherical billboards

The evaluation of the length the ray travels in a particle sphere and the computation of the corresponding opacity can be efficiently executed by a custom fragment shader program. The fragment program gets some of its inputs from the vertex shader: the particle position in camera space, the shaded billboard point in camera space, the particle radius, and the screen coordinates of the shaded point to address the depth map. Some uniform parameters should also be given, including the density, and the camera's front clipping plane distance. Algorithm 5 shows the pseudo code of the spherical billboard algorithm, that can be used in a fragment shader to calculate opacity.

Algorithm 5 Spherical billboard

```
alpha ← 0
Calculate distance d from particle center
if d < particleRadius then
   Calculate w using equation 7.2
   Calculate enter and exit points on sphere using equation 7.1
   Read scene depth
   Calculate ray length inside sphere using equation 7.3
   alpha ← evaluate equation 7.4
else
   Discard pixel
end if
return alpha
```

With this simple calculation the shader program obtains the real ray segment length and computes opacity of the given particle that controls blending of the particle into the frame buffer. The consideration of the spherical geometry during fragment processing eliminates clipping and popping artifacts (see Figure 7.6).

7.2 Rendering explosions

An explosion consists of dust, smoke, and fire, which are modeled by specific particle systems. Dust and smoke absorb light, fire emits light. These particle systems are rendered separately, and the final result is obtained by compositing their rendered images.

7.2.1 Dust and smoke

Smoke particles are responsible for absorbing light in the fire. These particles typically have low albedo values ($a = 0.2, \tau = 0.4$). High albedo ($a = 0.9, \tau = 0.4$) dust that is swirling in the air, on the other hand, is added to improve the realism of the explosion (Figure 7.7).

Figure 7.6: Particle system rendered with planar (left) and with spherical (right) billboards.

When rendering dust and smoke we assume that these particles do not emit radiance so their emission term is zero. To calculate the in-scattering term, the length the light travels in the particle sphere, the albedo, the density, and the phase function are needed. We use the Henyey-Greenstein phase function [HG40, CS92] (equation 1.3). We have found that setting g to constant zero gives satisfactory results for dust and smoke.

The real length the light travels inside a smoke or dust particle is computed by the proposed spherical billboard method.

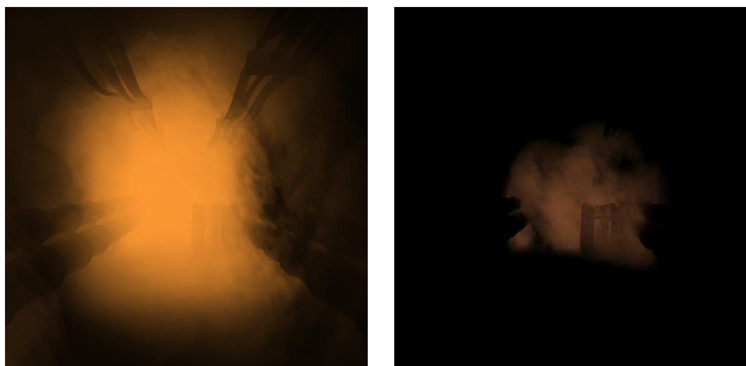


Figure 7.7: High albedo dust and low albedo smoke.

In order to maintain high frame rates, the number of particles should be limited, which may compromise high detail features. To cope with this problem, the number of particles is reduced while their radius is increased. The variety needed by the details is added by perturbing the opacity values computed by spherical billboards. Each particle has a unique, time dependent perturbation pattern. The perturbation is extracted from a grey scale texture, called **detail image**, which depicts real smoke or dust (Figure 7.8). The perturbation pattern of a particle is taken from a randomly placed, small quad shaped part of this texture. This technique has been used for a longer time by off-line renderers of the motion picture industry [AG00]. As time advances this texture is dynamically updated to provide variety in the time domain as well. Such animated 2D textures can be obtained from real world videos and stored as a 3D texture since inter-frame interpolation and looping can automatically be provided by the graphics hardware's texture sampling unit [Ngu04].

7.2.2 Fire

Fire is modelled as a **black-body** radiator rather than participating medium, i.e. the albedo is zero, so only the emission term is needed. The color characteristics of fire particles are deter-

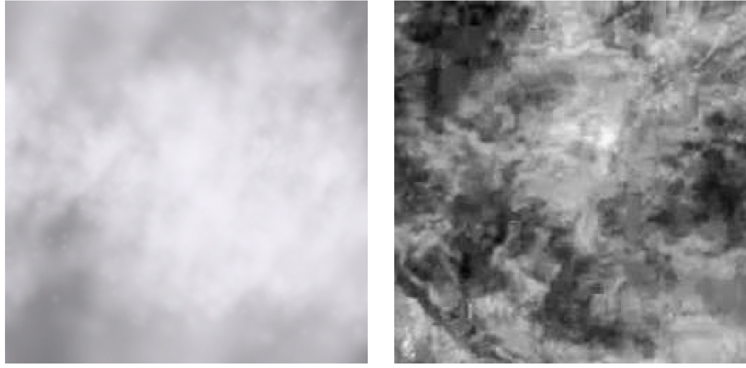


Figure 7.8: Images from real smoke and fire video clips, which are used to perturb the billboard fragment opacities and temperatures.

mined by the physics theory of black-body radiation. For wavelength λ, the emitted radiance of a black-body can be computed by **Planck's formula**:

$$L_{e,\lambda}(x) = \frac{2C_1}{\lambda^5(e^{C_2/(\lambda T)} - 1)}$$

where $C_1 = 3.7418 \cdot 10^{-16} Wm^2$, $C_2 = 1.4388 \cdot 10^{-2} m^\circ K$, and T is the absolute temperature of the radiator [SH81]. Figure 7.9 shows the spectral radiance of black-body radiators at different temperatures. Note that the higher the temperature is, the more blueish the color gets.

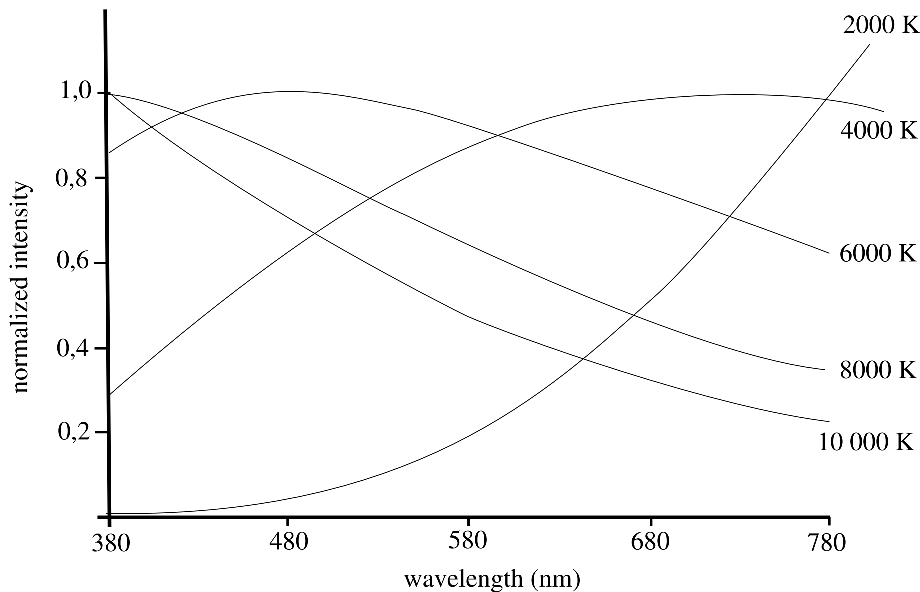


Figure 7.9: Black-body radiator spectral distribution

For different temperature values, the RGB components can be obtained by integrating the spectrum multiplied by the color matching functions. These integrals can be pre-computed and stored in a texture. To depict realistic fire, the temperature range of $T \in [2500^\circ K, 3200^\circ K]$ needs to be processed (see Figure 7.10).

The opacity of fire particles is calculated by the spherical billboards.

Figure 7.10: Black-body radiator colors from $0°K$ to $10000°K$. Fire particles belong to temperature values from $2500°K$ to $3200°K$.

High detail features are added to fire particles similarly to smoke and dust particles. However, now not the opacity, but the emission radiance should be perturbed. We could use a color video and take the color samples directly from this video, but this approach would limit the freedom of controlling the temperature range and color of different explosions. Instead, we decided to store the temperature variations in the detail texture (Figure 7.8), and its stored temperature values are scaled and are used for color computation on the fly. A randomly selected, small quadrilateral part of a frame in the detail video is assigned to a fire particle to control the temperature perturbation of the fragments of the particle billboard. The temperature is scaled and a bias is added if required. Then the resulting temperature is used to find the color of this fragment in the black-body radiation function.

7.2.3 Layer composition

To combine the particle systems together and with the image of opaque objects, a layer composition method has been used. This way we should render the opaque objects and the particle systems into separate textures, and then compose them. This leads to three rendering passes: one pass for opaque objects, one pass for dust, fire, and smoke, and one final pass for composition. The first pass computes both the color and the depth of opaque objects.

One great advantage of rendering the participating medium into a texture is that we can use floating point blending. Another advantage is that this render pass may have smaller resolution rendering target than the final display resolution, which considerably speeds up rendering since blending needs a huge amount of pixel processing power to overdraw a pixel many times (see Figure 7.11). Table 7.1 shows how the ratio of screen resolution and particle render target resolution influence performance. On recent hardware even with high screen and render target resolutions considerably high frame rates can be achieved.

GPU screen resolution	**Render target ratio**			
	1	**1/2**	**1/4**	**1/8**
GeForce 6800GT 512×512	30 FPS	40 FPS	50 FPS	60 FPS
GeForce GTX260 512×512	330 FPS	750 FPS	750 FPS	750 FPS
GeForce GTX260 1024×768	130 FPS	345 FPS	730 FPS	750 FPS

Table 7.1: *Speed comparison of different screen resolutions, render target resolutions and graphics hardware.*

To enhance realism, we also simulated heat shimmering that distorts the image [Ngu04]. This is done by rendering particles of a noisy texture. This noise is used in the final composition as u, v offset values to distort the image. With the help of multiple render targets, this pass can also be merged in the pass of rendering of fire particles.

The final effect that could be used through composition is motion blur, which can easily be

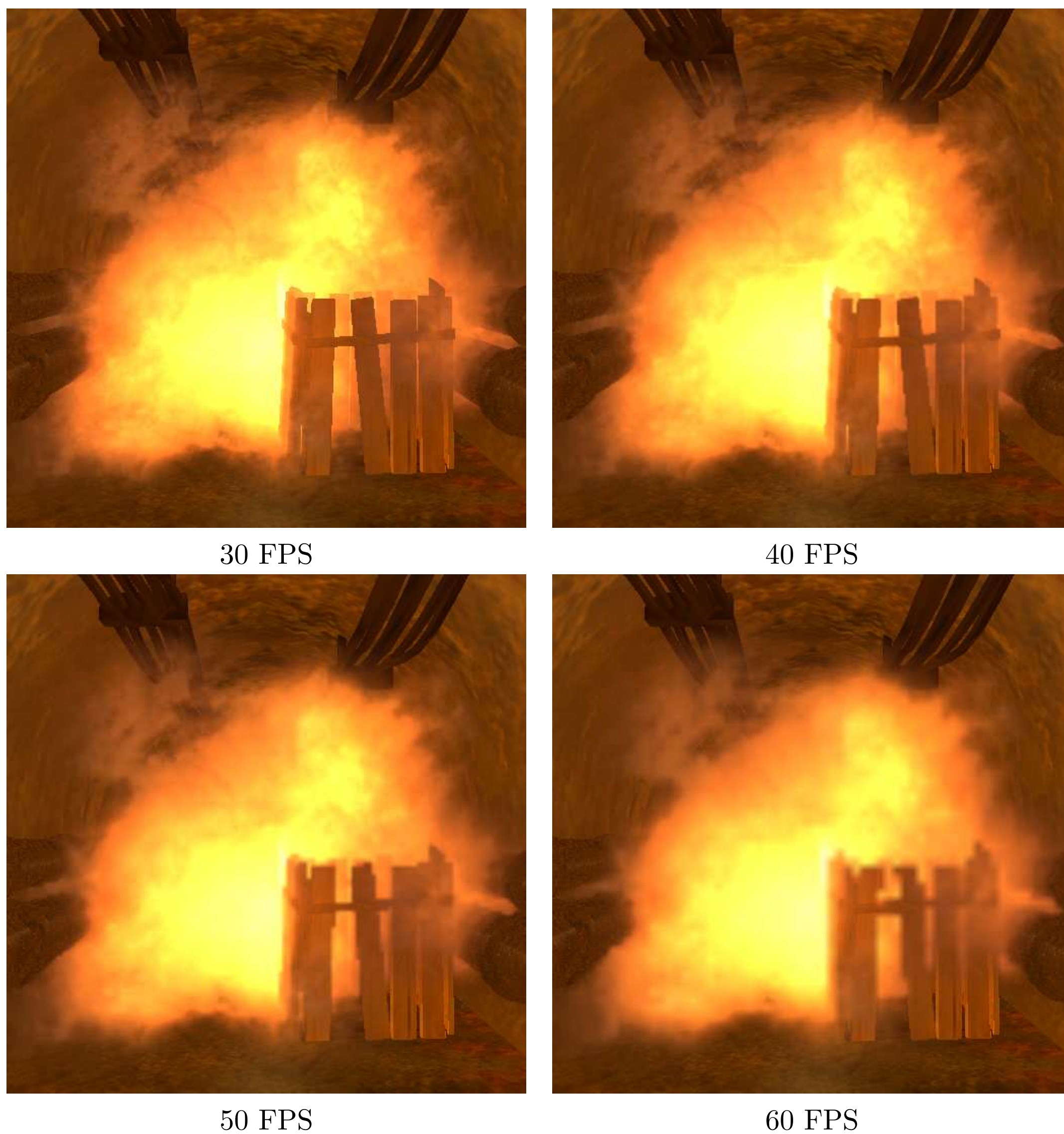


Figure 7.11: Particles rendered to render targets of different resolutions (GeForce 6800GT). Upper left: particle render target with screen resolution. Upper right, lower left, lower right: render target with half, quarter and one eights of the screen resolution.

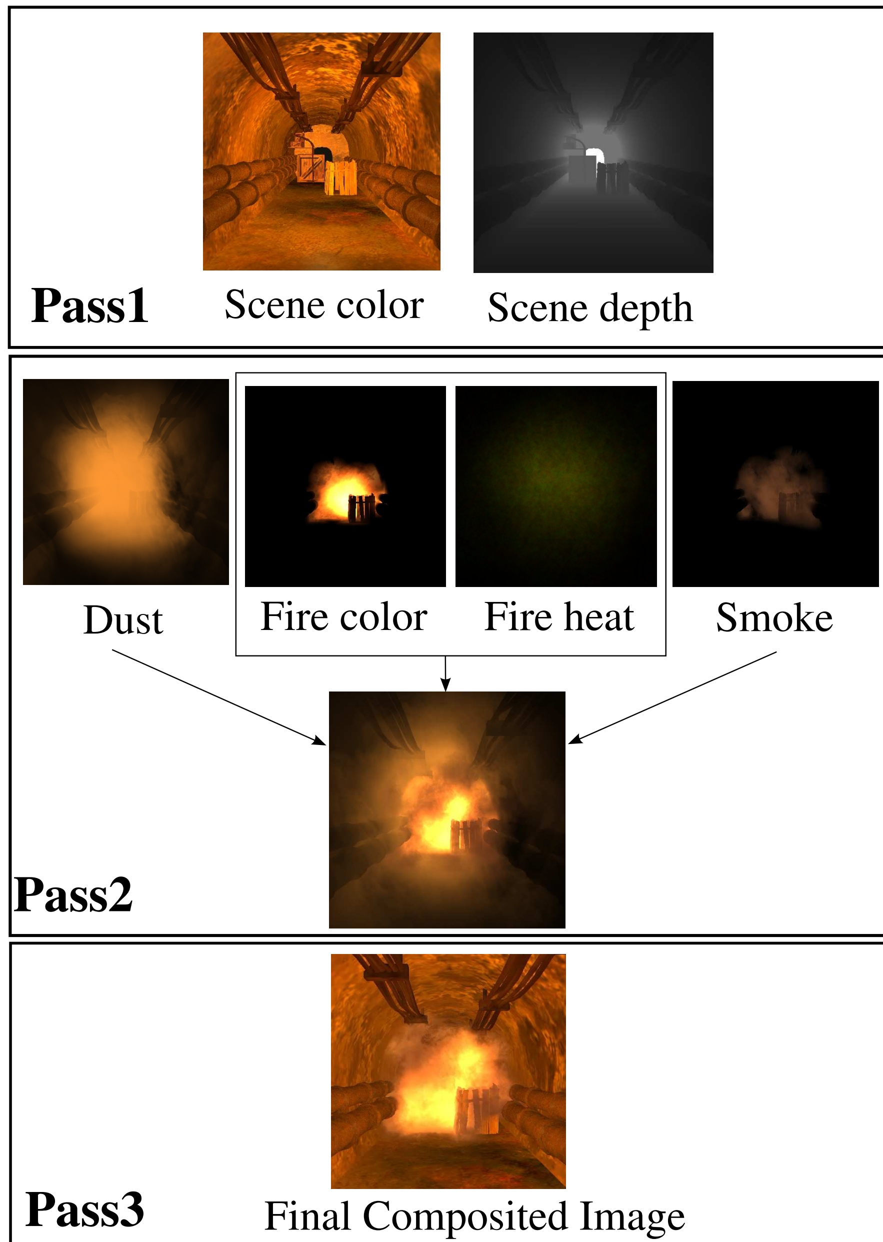


Figure 7.12: Rendering algorithm

done with blending, letting the new frame fade into previous frames. The complete rendering process is shown in Figure 7.12.

7.3 Results

The presented algorithm has been implemented in OpenGL/Cg environment on an NV6800GT graphics card. The dust, the fire, and the smoke consist of 16, 115, and 28 animated particles, respectively. Note that these surprisingly small number of larger particles can be simulated very efficiently, but thanks to the opacity and temperature perturbation, the high frequency details are not compromised. The modeled scene consists of 16800 triangles. The scene is rendered with per pixel Phong shading. During particle simulation we detected collisions between particles and a simplified scene geometry. Our algorithm offers real-time rendering speed (40 FPS) with high details (see Figure 7.13). The frame rate strongly depends on the number of overridden pixels. For comparison, the scene without the particle system is rendered at 70 FPS, and the classic billboard rendering method would also run at about 40 FPS. This means that the performance lost due to the more complex spherical billboard calculations can be regained by decreasing the render target resolution during particle system drawing.

7.4 Conclusion

This chapter proposed to consider particles as spheres rather than planar billboards during fragment processing, while still rendering them as billboards, which eliminated billboard clipping and popping artifacts. The paper also introduced an efficient method to display explosions composed of fire, smoke, and dust. The discussed method also used post rendering effects and runs at high frame rates.

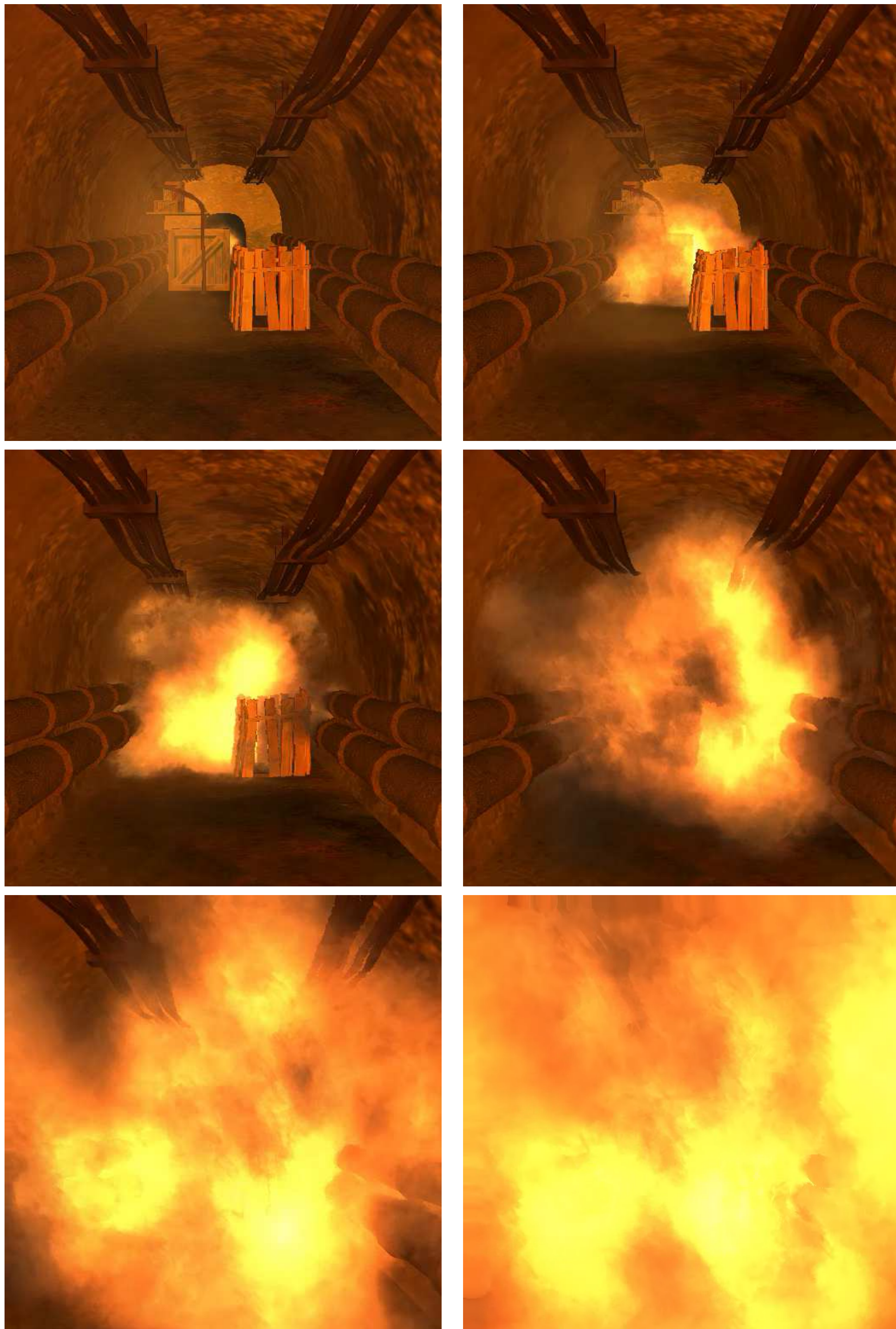


Figure 7.13: Rendered frames from the animation sequence.

Chapter 8

Hierarchical Depth Impostors

The **in-scattering** term requires the evaluation of a directional integral at each particle j. Real-time methods usually simplify this integral and consider only the directions of the light sources in these integrals, and thus allowing only attenuation and forward scattering [Har02]. The incoming radiance for all particles can be effectively evaluated executing a light pass for each light source. In a light pass, particles are sorted along the light direction, and their opacity billboards are rendered one by one in this order. Before rendering a particle, the color buffer is read back to obtain the light attenuation at this particle, then the opacity texture of the particle is combined with the image to prepare the light attenuation for the subsequent particle.

The incoming radiance of a particle due to a given light source is computed from the light source intensity and the opacity accumulated so far.

If we know the incoming radiance of particles, the volume can efficiently be rendered from the camera using alpha blending. The in-scattering term of a particle is obtained by multiplying the albedo and the phase function with the incoming radiance values due to the different light sources, which is attenuated according to the total opacity of the particles that are between the camera and this particle. This requires the sorting of particles in the view direction before sending them to the frame buffer in back to front order. At a given particle, the evolving image is decreased according to the opacity of the particle and increased by its in-scattering term.

In this chapter we propose the application of particle hierarchies to reduce the computational burden of rendering. On the lower level, particles are grouped into blocks, that are rendered once for the current viewing or lighting direction, then the role of the individual particles are taken by these blocks. To eliminate the read-backs of the color buffer, we compute the attenuation at discrete depth samples during light passes.

8.1 The new method using particle hierarchies

To reduce the computational burden of rendering, particle hierarchies are formed, and the full system is built of smaller similar blocks. As most natural phenomena shows self-similarity, we can use this approximation in most cases.

A single **particle block** represents particles that are close to each other. Before rendering for a given direction, the image of the particles of a block is determined from this direction, and then we use these images instead of individual particles. The image is called **depth impostor**. A pixel of the depth impostor stores information that is needed about the particles which project onto this pixel, particularly their total opacity, their minimum (front) and maximum (back) depths. During rendering an impostor pixel acts as a “super-particle” that concentrates all those particles of the block, which are projected onto it. The total opacity is used in the radiance transfer, while the depths are taken into account to eliminate artifacts caused by objects included in the volume. The idea of approximate particle systems with blocks and **double layered depth impostors** is was an extension of **2.5D impostors** originally used to render crown of trees [SK03](section 2.2).

Replacing particles by blocks, the computation burden can be reduced significantly. If we want a system with N particles, we can build a block of b particles and instance it N/b times. The hierarchical approach needs only $b + N/b$ calculations during simulation and color computation, in contrast to N calculations of the non-hierarchical method.

8.1.1 Generating a depth impostor

To generate a depth impostor representing a block for a particular direction (either light or camera), the particles of the block are rendered and the total opacity, and front and back depths are computed for each pixel on the GPU. These depth impostors are first generated for particle spheres and then for volume blocks. The opacity texture of a particle is pre-defined, and the depth textures of a particle sphere can be created analytically evaluating the depths of a sphere in the preprocessing phase.

The total opacity of a block could be determined using alpha blending of the opacity textures of the particles. The depth values of the block, however, require a different operation. A simple approach would generate the front and back depth textures of a block by rendering particle the front and back textures of the particles, overwriting the fragment depth value in the fragment shader, and letting the z-buffer to find the minimum and the maximum in two different passes.

However, this simple approach needs three rendering passes for each particle block. Fortunately, it is also possible to execute the three different calculations in a single rendering pass if depth testing is replaced by alpha testing and with enabling special blending operators of the GPUs. The layers of a depth impostor are shown in Figure 8.1.

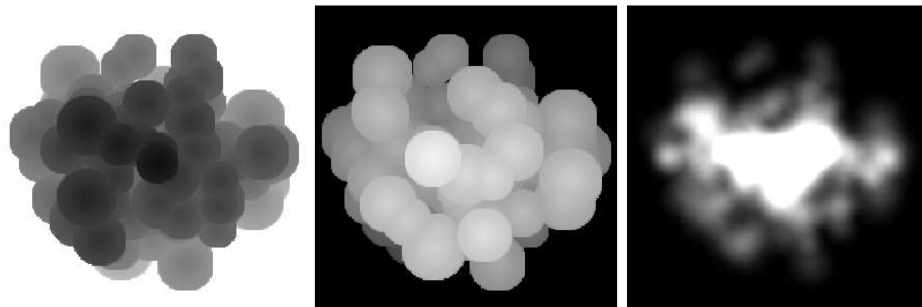


Figure 8.1: Depth impostor layers of a block: front depth, back depth, and accumulated opacity

8.1.2 Using depth impostors during light passes

Rendering participating media consists of a separate light pass for each light source determining the approximation of the in-scattering term caused by this particular light source, and a final gathering step. Handling light volume interaction on the particle level would be too computation intensive since the light pass requires the incremental rendering of all particles and the read back of the actual result for each of them. To speed up the process, we render particle blocks one by one, and separate light–volume interaction calculation from the particles.

During a light pass we classify particle blocks into groups according to their distances from the light source, and store the evolving image in textures at given sample distances. These textures are called **light slices**. The first texture will display the accumulated opacity of the first group of particle blocks, the second will show the opacity of the first and second groups of particle blocks and so on. The required number of depth samples depends on the particle count and the cloud shape. For a roughly spherical shape and relatively few particles (where overlapping is not dominant), even four depths can be enough. Figure 8.2 shows this technique with five light slices. Four slices can be computed simultaneously if we store the slices in the color channels of one RGBA texture. For a given particle, the vertex shader will decide in

which slice (or color channel) this particle should be rendered. The vertex shader sets the color channels corresponding to other slices to zero, and the pixel is updated with alpha blending.

8.1.3 Using depth impostors during final gathering

During final rendering we obtain the depth impostors of the blocks for the viewing direction, sort them and render them one after the other in back to front order. The in-scattering term is obtained from the sampled textures of the slices that enclose the pixel of the block (Figure 8.2).

The accumulated opacity of the slices can be used to determine the radiance at the pixels of these slices and finally the reflected radiance of a particle between the slices. Knowing the position of the particles we can decide which two slices enclose it. By linear interpolation between the values read from these two textures we can approximate the attenuation of the light source color. Harris used a similar technique in [HBSL03], where he stored these slices in a 3D texture called oriented light volume. In order to obtain a better multiple scattering approximation, the radiance of those pixels of both enclosing slices are taken into account, for which the phase function is not negligible.

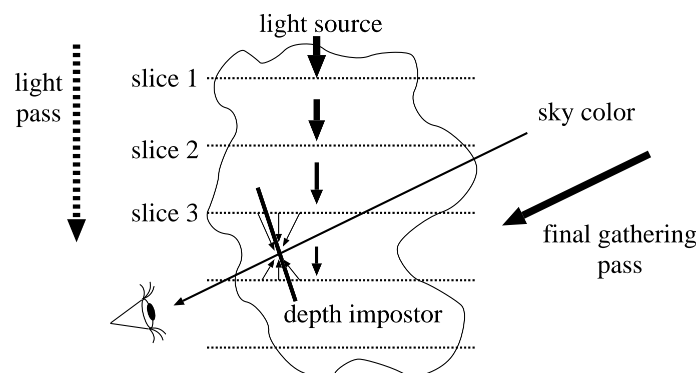


Figure 8.2: Final gathering for a block

8.1.4 Objects in clouds

As we are working with billboards, this method suffers from the main problem of billboard rendering described in Section 7, namely clipping artifacts. This problem is solved using the extension of the particle block, i.e. the interval of the block in the depth direction, which is stored in impostor texels. This idea is similar to the **spherical billboard** method (Section 7), but in this case the particles cannot be treated as spheres and no analytic expression can describe their density distribution. Our solution is a volumetric extension of the 2.5D impostor method described in [SK03].

In order to attack this problem, first we render all objects of the scene and save the depth buffer in a texture. Then the particle blocks are rendered one by one, enabling depth test but disabling depth write. When rendering a particle block, we compute the interval the ray travels inside the block adding the depth value of the block center to the front and back depth values of the depth impostor. This interval is compared with the value storing the depth of the visible object. If the object depth value is outside the interval, then the object is either fully visible or fully occluded by the particle sphere, thus we can rely on the z-buffer and alpha-blending hardware to compute the correct result. However, when the interval of the block encloses the depth of the object, only a part of the volume block occludes the object. In this case, the opacity of the particle block is scaled according to the relative distance between the front depth of the particle block and the object, and the depth interval of the particle block. This scaling

Figure 8.3: Problems caused by objects in a volume rendered as billboards. Where the billboard plane intersects the object, transparency becomes discontinuous.

corresponds to the assumption that the density is uniform inside a block. The results are shown in Figure 8.4. Algorithm 6 shows the pseudo code of the rendering algorithm.

Figure 8.4: Volume rendered with depth impostors eliminating billboard clipping artifacts

8.2 Results

The presented algorithm has been implemented in OpenGL/Cg environment on an NV6800GT graphics card. The animated cloud of Figure 8.5 consists of 8000 particles grouped to different number of blocks, and is illuminated by a directional light. The rendering speed increases as we put more particles in a single block until 40 blocks. For higher number of particles per block, rendering gets slower, because of the block computation overhead (Figure 8.6). We should note that on recent hardware (like the NVidia GTX 260) the performance of the algorithm doubles.

8.3 Conclusions

This chapter proposed to build particle clouds of blocks. A block itself represents many particles, and is defined by a depth impostor. We also applied depth sampling to compute self-shadowing and multiple forward scattering quickly. As a combined effect of these improvements, the presented method renders realistically shaded dynamic smoke or cloud formations under changing lighting conditions at high frame rates, taking advantage of the GPU. On the other hand, the inclusion of front and depth information into the depth impostors eliminated billboard clipping artifacts when the volume contains 3D objects.

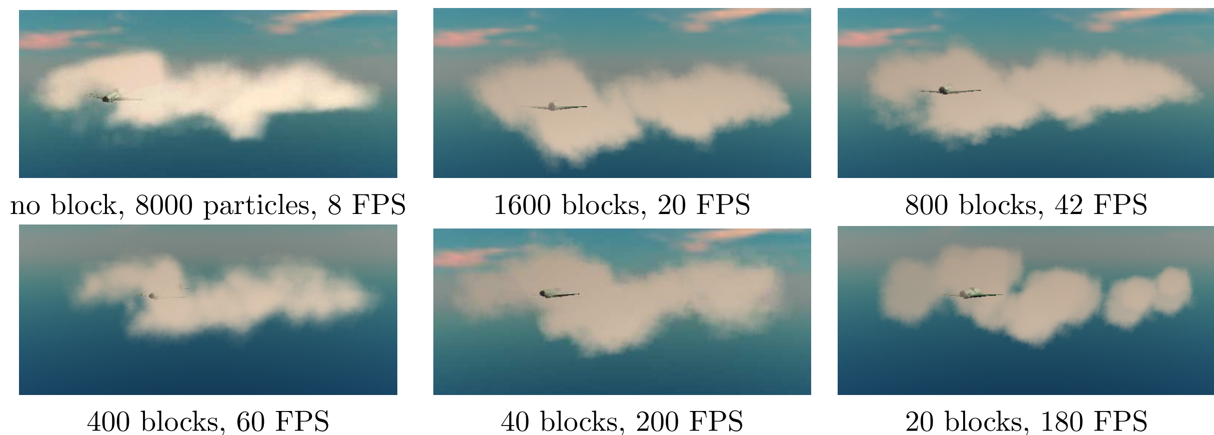
Algorithm 6 Participating media rendering with depth impostors

```
Store scene depth

Define ortogonal projection oriented to view direction and fitted to particle block
Store minimum depth values of the particle block
Store maximum depth values of the particle block
Store total opacity of the particle block

Define ortogonal projection oriented to light direction and fitted to the particle system
for each light slice do
  Clear with white color
  for each particle in this slice do
    Render with alpha blending and black color using block opacity
  end for
end for

for Each particle do
  Render block-sized screen aligned quad
  for Each drawn fragment do
    Read opacity from impostor image
    Read minimum depth from impostor image
    Read maximum depth from impostor image
    Read scene depth
    Calculate ray length inside block
    Calculate opacity from ray length
    Identify adjacent light slices
    Read attenuated light intensities from light slices
    Interpolate between slices to obtain incoming radiance
    Calculate reflected radiance according to computed opacity
  end for
end for
```



no block, 8000 particles, 8 FPS	1600 blocks, 20 FPS	800 blocks, 42 FPS
400 blocks, 60 FPS	40 blocks, 200 FPS	20 blocks, 180 FPS

Figure 8.5: Images of animated volumes of 8000 particles organized in different number of blocks

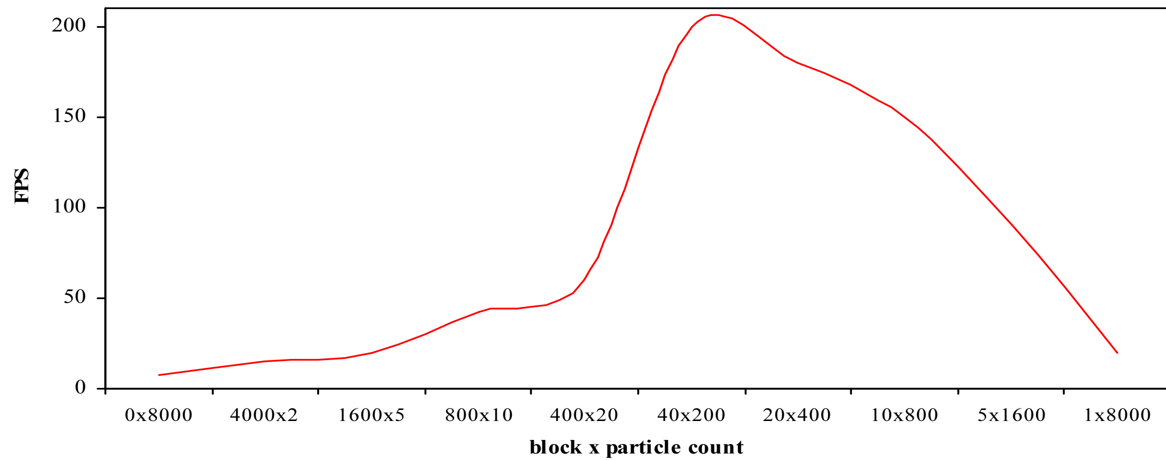


Figure 8.6: The effect of grouping different number of particles into different number of blocks.

Figure 8.7: Swirling cloud rendered at 180 FPS

Figure 8.8: Swirling cloud crossed by a plane rendered at 180 FPS

Chapter 9

Parcipating media rendering with illumination networks

We solve the discretized volumetric rendering equation by iteration. The volume is represented by a set of randomly sampled particle positions. Suppose that we have an estimate of particle radiance values (and consequently, of incoming radiance values) at iteration step $n-1$. The new particle radiance in iteration step n is obtained by substituting these values to the right side of the discretized volumetric rendering equation:

$$L_p^n(\vec{\omega}) = (1-\alpha_p)I_p^{n-1}(\vec{\omega}) + E_p(\vec{\omega}) + \alpha_p a_p \int_{\Omega'} I_p^{n-1}(\vec{\omega}')P_p(\vec{\omega}',\vec{\omega})\, \mathrm{d}\omega'. \tag{9.1}$$

This iteration is convergent if the opacity is in $[0,1]$ and the albedo is positive and less than 1, which is always the case for physically plausible materials.

In order to calculate the directional integral representing the inscattering term of equation 9.1, we suppose that D random directions $\vec{\omega}_1,\ldots,\vec{\omega}_D$ are obtained from uniform distribution of density $1/(4\pi)$, and the integral is estimated by Monte Carlo quadrature:

$$\int_{\Omega'} I_p(\vec{\omega}')P_p(\vec{\omega}',\vec{\omega})\, \mathrm{d}\omega' \approx \frac{1}{D}\sum_{d=1}^{D} I_p(\vec{\omega}'_d)P_p(\vec{\omega}'_d,\vec{\omega})4\pi.$$

Note that replacing the original integral by its approximating quadrature introduces some error in each iteration step, which accumulates during the iteration. This error can be controlled by setting D according to the albedo of the participating media since if the error of a single step is ϵ and the albedo is a, then the error is bound by $\epsilon/(1-a)$.

9.1 The new method using illumination networks

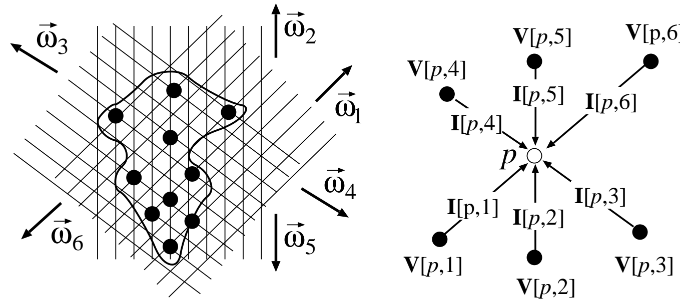


Figure 9.1: Illumination and visibility networks

If we use the same set of sample directions for all particles, then the incoming radiance and therefore the outgoing radiance are needed only at these directions during iteration. For a single particle p, we need D incoming radiance values $I_p(\vec{\omega}_d)$ in $\vec{\omega}_1, \ldots, \vec{\omega}_D$, and the reflected radiance needs to be computed exactly in these directions. In order to update the radiance of a particle, we should know the indices of the particles visible in sample directions, and also the distances of these particles to compute the opacity. This information can be stored in two-dimensional arrays **I** and **V** of size $N \times D$, indexed by particles and directions respectively (figure 9.1). Array **I** is called the **illumination network** and stores the incoming radiance values of the particles on the wavelengths of red, green, and blue. Array **V** is the **visibility network** and stores index of visible particle vp and opacity α for each particle and incoming direction, that is, it identifies from where the given particle can receive illumination (figure 9.2).

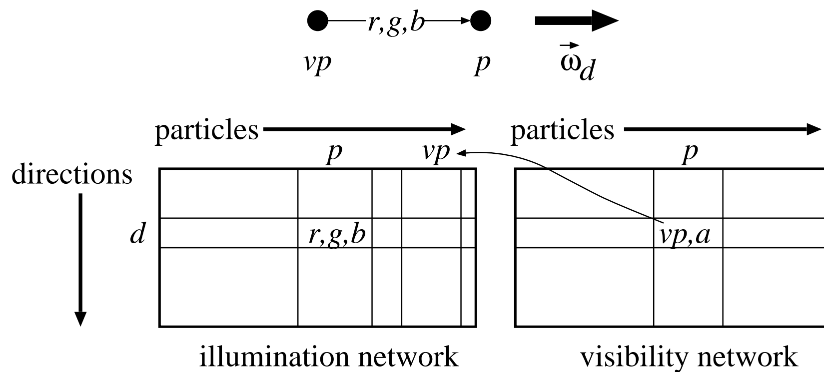


Figure 9.2: Storing the networks in arrays

In order to handle emissions and the direct illumination of light sources, we use a third array **E** that stores the sum of the emission and the reflection of the direct illumination for each particle and discrete direction. This array can be initialized by rendering the volume from the point of view of the light source and identifying those particles that are directly visible. At a particular particle, the discrete direction closest to the illumination direction is found, and the reflection of the light source is computed from the incoming discrete direction for each outgoing discrete direction.

Visibility network **V** expressing the visibility between particles is constructed during a preprocessing phase (Figure 9.3). The bounding sphere of the volume is constructed and then D uniformly distributed points are sampled on its surface. Each point on the sphere defines a direction aiming at the center of the sphere, and also a plane perpendicular to the direction. A square window is set on this plane to include the projection of the volume, and the window is discretized to $M \times M$ pixels.

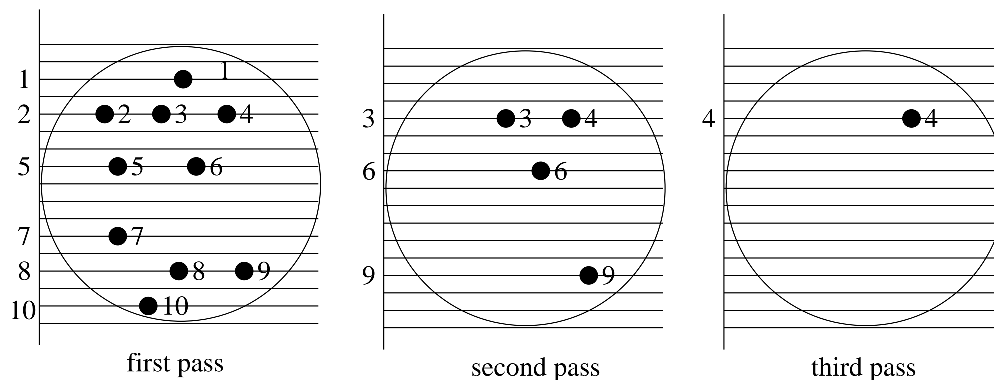


Figure 9.3: Constructing the visibility network

When a particular direction is processed, particles are orthographically projected onto the window, and rendered using a standard z-buffer algorithm, having set the color of particle p equal to its index p (Figure 9.3). The contents of the image and depth buffers are read back to the CPU memory, and the indices and depths of the visible particles are stored together with the pixel coordinates. The particles that were visible in the preceding rendering step are ignored in the subsequent rendering steps. Repeating the rendering for the remaining particles and reading back the image and depth buffers again, we can obtain the indices of those particles which were occluded in the previous rendering step. Pairing these indices to those previously obtained ones which have the same pixel coordinates, we can get the pairs of particles that occlude each other in the given direction. On the other hand, the difference of the depth values is the distance of the particles, from which the opacity can be computed. Repeating the same step until the image is empty, we can build lists of particles that are projected onto the same pixel. Subsequent pairs of these lists define a single row of array $\mathbf{V}$ corresponding to this direction (figure 9.2). Executing this algorithm for all predefined directions, the complete array can be filled up.

Note that multiple z-buffer steps carry out a **depth peeling** procedure. Since this happens in the preprocessing step, its performance is not critical. However, if we intend to modify the illumination network during rendering in order to cope with animated volumes, then the performance should be improved. Fortunately, the depth peeling process can also be realized on the GPU as suggested by [Eve01, Hac04].

9.1.1 Setting the parameters of the illumination network

The illumination network depends on radius R of the bounding sphere, number of discrete directions D, and on resolution of the windows $M \times M$. These parameters are not independent, but must be set appropriately taking into account the density of the medium as well. Since point rendering is used during the projection onto the window of size $2R \times 2R$ and of resolution $M \times M$, the projected area of a particle is implicitly set to $A = 4R^2/M^2$. The solid angle in which a particle at point $\vec{x}$ is seen from another particle at $\vec{y}$ is $\Delta\omega = A/|\vec{x} - \vec{y}|^2$. Approximating the maximum distance by the expected free run length $1/\tau$, we get $\Delta\omega \geq A\tau^2$.

If we do not want to miss particle interactions due to the insufficient number of sample directions, solid angle $\Delta\omega$ should not be smaller than the solid angle assigned to a single directional sample, which is $4\pi/D$. Substituting the implicit projected area of a particle, we obtain that the number of sample directions and the resolution of the window should meet $D(R\tau)^2/\pi \geq M^2$. The number of sample directions is typically 128, factor $R\tau$ is the expected number of collisions while the light travels through half of the volume, which usually ranges in 10–100, thus maximum resolution M is in 60–600, i.e. it is usually quite small.

9.1.2 Iterating the illumination network

The solution of the global illumination problem requires the iteration of the illumination network. A single step of the iteration evaluates the following formula for each particle $p = 1, \ldots, N$ and for each incoming direction $i = 1, \ldots, D$:

$$\mathbf{I}[p,i] = (1 - \alpha_{\mathbf{V}[p,i]})\mathbf{I}[\mathbf{V}[p,i],i] + \mathbf{E}[\mathbf{V}[p,i],i] + \frac{4\pi\alpha_{\mathbf{V}[p,i]}a_{\mathbf{V}[p,i]}}{D}\sum_{d=1}^{D}\mathbf{I}[\mathbf{V}[p,i],d]P_{\mathbf{V}[p,i]}(\vec{\omega}'_d, \vec{\omega}_i).$$

Interpreting the two-dimensional arrays of the emission, visibility and illumination maps as textures, the graphics hardware can also be exploited to update the illumination network. The first texture is visibility network $\mathbf{V}$ storing the visible particle in red and the opacity in green channels, the second stores emission array $\mathbf{E}$ in the red, green, and blue channels, and the third texture is the illumination map, which also has red, green and blue channels. Note that in practical cases number of particles N is about a thousand, while number of sample direction D is typically 128, and radiance values are half precision floating point numbers, thus the total size of these textures is quite small ($1024 \times 128 \times 8 \times 2$ bytes $= 2$ Mbyte).

Figure 9.4: Cloud illuminated by two dynamic directional light sources (the first is left-up, the second is bottom-right) and sky illumination, and rendered by an animated camera at 26 FPS.

In the GPU implementation a single iteration step is the rendering of a viewport sized, textured rectangle, having set the viewpoint resolution to $N \times D$ and the render target to the texture map representing the updated illumination network. A pixel corresponds to a single particle and single direction. The pixel shader obtains the visibility network, the emission array, and the illumination map from textures. The phase function is implemented by a texture lookup of prepared values allowing different phase functions to be easily incorporated [REK$^+$04]. In our implementation we assume that opacity is precomputed and stored in the visibility texture, but albedo are constant for all particles. Should it not be the case, the albedo could also be looked up in a texture.

When no other particle is seen in the input direction, then the incoming illumination is taken from the sky radiance (`sky`). In this way not only a single sky color, but sky illumination textures can also be used [PSS99, REK$^+$04].

For each particle and for each direction, the fragment shader finds opacity and the visible particle, takes its emission or direct illumination, and computes its radiance as the sum of the direct illumination and the reflected radiance values for its input directions.

The illumination network provides a view independent radiance representation. When the final image is needed, we can use a traditional participating media rendering method, which sorts the particles according to their distance from the camera, splats them, and adds their contributions with alpha blending.

When the outgoing reflected radiance of a particle is needed, we compute the reflection from the sampled incoming directions to the viewing direction. Finally the sum of particle emission and direct illumination of the external lights is interpolated from the sample directions, and is added to the reflected radiance. Algorithm 7 shows the pseudo code of the illumination networks algorithm.

9.2 Results

The proposed algorithm has been implemented in OpenGL/Cg environment and run on an NV6800GT graphics card. We compute one iteration in each frame, and when the light sources move, we take the solution of the previous light position as the initial value of the iteration, which results in fast convergence.

The results are shown in Figures 9.4, 9.5, and 9.6. Where it is not explicitly stated, the cloud model consists of 1024 particles, and 128 discrete directions are sampled. With these settings the typical rendering speed is about 26 frames per second, and is almost independent of the number of light sources and of the existence of sky illumination. The albedo is 0.9, and the expected number of photon–particle collisions ($2R\tau$) is 20, and material parameter g is 0. Figures 9.4 shows how two external light sources illuminate the cloud. In figure 9.5 we can follow the evolution of the image of the same cloud after different iteration steps, where we can observe the speed of convergence. Figure 9.6 describes how the number of sample directions

Algorithm 7 Illumination networks

```
Precomputation:
for each particle do
    for each direction do
        Find visible particle
        Compute opacity from distance to visible particle
        Store opacity and index of visible particle
    end for
end for

Iteration:
Initialize direct illumination map with particle emission values
for each light source do
    for each direction do
        for each particle do
            Calculate the reflection of direct illumination and add to direct illumination map
        end for
    end for
end for
for number of iterations do
    for each particle do
        for each direction do
            I ← 0
            Read visible particle and opacity
            Read direct illumination and add to I
            for each direction do
                Read illumination from illumination map and calculate reflected radiance
                Add to I
            end for
            Store I in illumination map
        end for
    end for
end for

Final render:
for each rendered particle do
    I ← 0
    for each direction do
        Read incoming radiance
        Calculate reflected radiance to camera
        Add to I
    end for
    Render particle with color I
end for
```

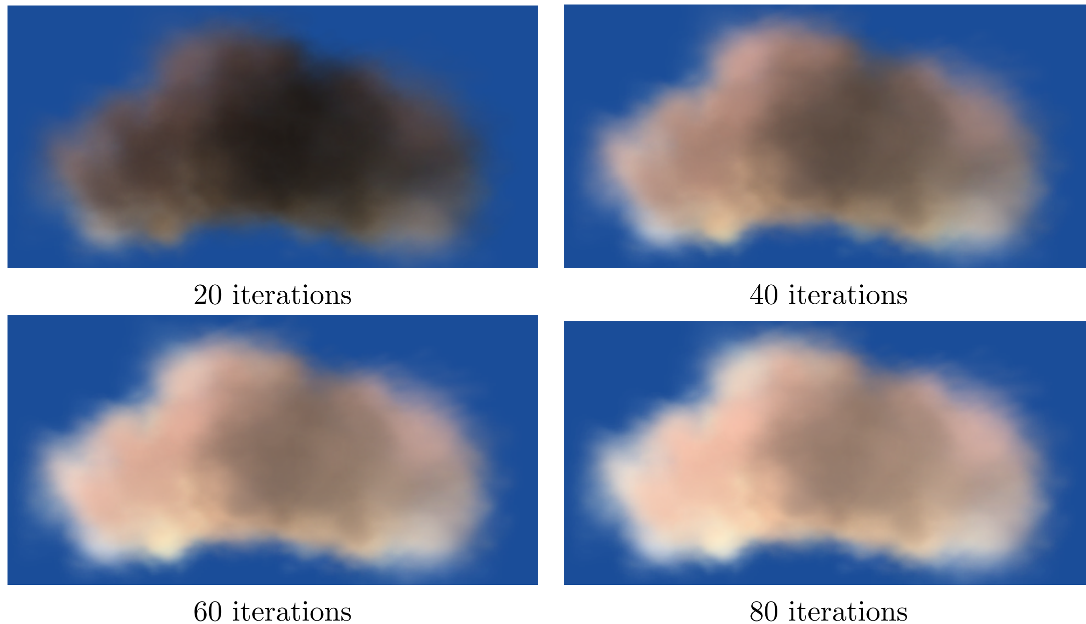


Figure 9.5: A cloud illuminated by two directional lights rendered with different iteration steps

and the number of particles affect the image quality and the rendering speed. Note that using 128 directions and 512 particles we can obtain believable clouds at interactive frame rates, and the method is not too sensitive to the number of discrete directions. Changing the number of particles, however, has more significant impact on the image.

The proposed method has some limitations in case of shader model 3 graphics cards, namely the available texture resolution is maximized in 4096×4096, which limits the number of simulated particles to 4096. This limit can be overcome with a little modification of the storage of the two dimensional arrays. However as it can be seen from the speed measurements of figure 9.6 around only 2000 particles can be simulated on such hardware in real-time. Fortunately recent graphics hardware (like GeForce GTX 260) not only has three or four times more computational power, but also offers higher resolution textures, which enables real-time simulation of even 8000 particles using 64 discrete directions.

9.3 Conclusions

This chapter presented a global illumination algorithm for participating media, which works with a prepared visibility network, and maintains a similar illumination network. These networks are two-dimensional arrays that can be stored as textures and managed by the graphics hardware. The resulting algorithm can render multiple scattering effects at high frame rates.

The current implementation assumes that the volume is static. To cope with evolving volumes, such as clouds in wind, fire, or smoke, we plan to gradually update the illumination network, re-evaluating the visibility just in a single direction at a time, and thus amortizing the cost of the building the data structure during animation. We also plan to extend the method for hierarchical particle systems to handle complex phenomena.

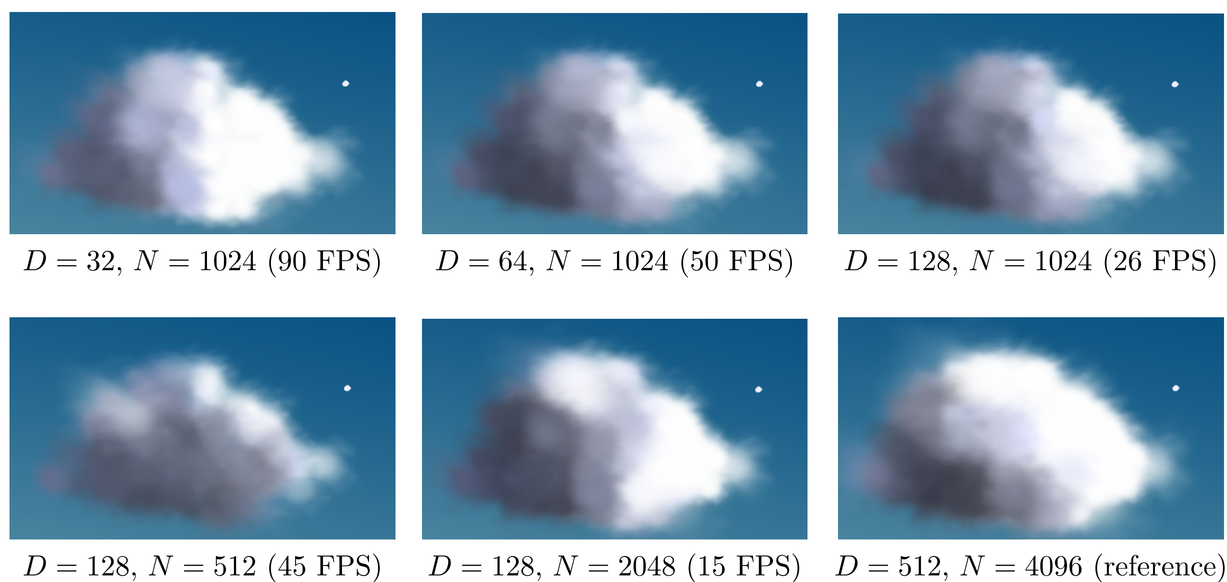


Figure 9.6: Effect of number of discrete directions D and number of particles N on the image quality and on rendering speed. The light source is in the direction of the white point in the upper right part of the image.

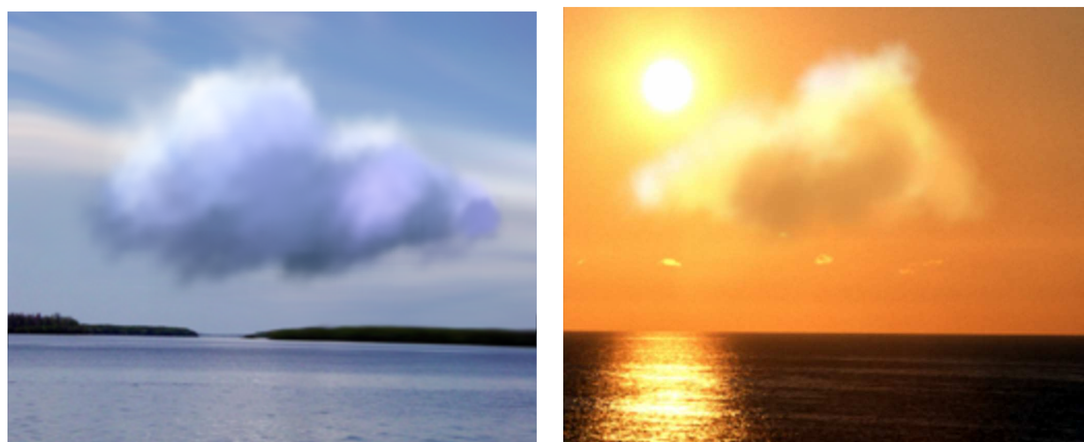


Figure 9.7: Globally illuminated clouds rendered with illumination networks.

Chapter 10

Conclusions

We have seen that efficient global illumination algorithms can be implemented on graphics hardware that can simulate one particular illumination effect in real-time. We proposed a robust environment map based method to calculate indirect diffuse illumination, and introduced a new ambient occlusion formula to compute self occlusion of the geometry that can be used in diffuse ambient lighting. These two algorithms can be used side-by-side to enhance each other, combining the color bleeding effect from the environment map based method and the self shadowing of the ambient occlusion method.

We discussed a robust environment map based method to render multiple reflections and refractions, and introduced a new triangle based caustic generation algorithm. These two algorithms are also completions of each other, the photon tracing part of caustic generation can be enhanced with tracing multiple refractions, which results in more accurate caustic patterns.

We also introduced solutions to eliminate billboard artifacts using spherical billboards. We extended this efficient method to render blocks of particles without artifacts, which enables efficient rendering of particle systems with huge amount of particles. We also gave a high speed solution to simulate light absorbtion in these particle systems. Finally we proposed an iteration method to compute multiple scattering under changing light conditions in static participating media with moderate particle count.

Though these algorithms shown pleasable results in the main fields of illumination and run real-time on current hardware, we can not lie back and think that our problems with real-time global illumination are solved. We should look for further enhancing possibilities.

In diffuse illumination we should consider using multiple environment map layers, or compute multiple indirect light bounces. We can also consider to use environment based methods for simulating subsurface scattering. We should also exploit frame-to-frame coherency, multiple depth layers and color information in screen space ambient occlusion techniques.

The speed of the robust search method used in ray tracing of multiple reflections and refractions can be enhanced with the use of special data structures (similar to cone maps for a possible example), which were efficiently used in displacement mapping.

We should explore particle based and grid based fluid simulation techniques, and rewrite the illumination networks algorithm to regular grids. We should examine the possibility of a more efficient reimplementation of the proposed algorithms on new architectures like Shader Model 4, or CUDA.

Chapter 11

Thesis summary

Thesis Group 1. Diffuse material rendering

Robust diffuse environment mapping

I have shown that treating environment texels as rectangular area light sources, polygon to point form factor can be implemented on the GPU, which makes environment map based diffuse indirect illumination calculation more accurate than form factors used before. The indirect light receiver object can be large compared to its environment and can move close to the sampled geometry still maintaining good visual quality. I have also made comparisons with a global illumination software to prove the relevancy of the new algorithm [J8].

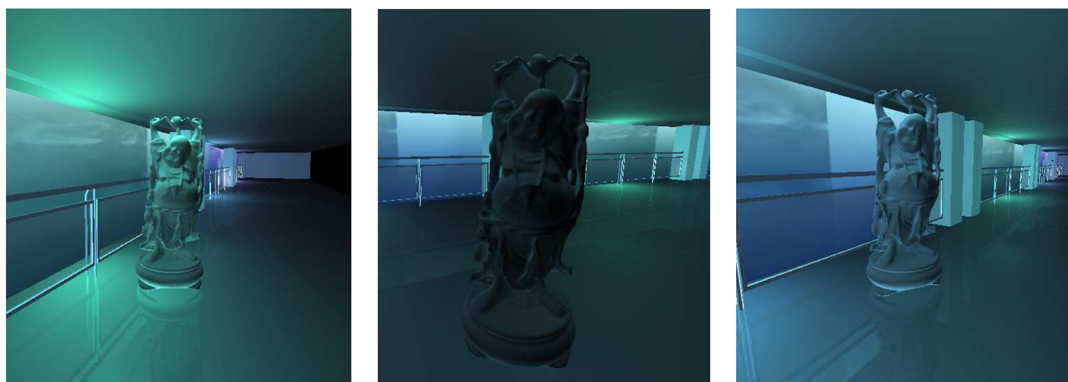


Figure 11.1: Robust diffuse environment mapping.

Volumetric screen space ambient occlusion

Screen space ambient occlusion techniques sample a depth map taken from the camera to calculate the openness of the surfaces. The quality is highly influenced by the number of samples. More samples provide smoother, less noisy results. I have introduced a new integration method that is based on a novel interpretation of ambient occlusion that measures how big portion of the tangent sphere of the surface belongs to the set of occluded points. The integrand of the new formula has low variation, thus can be estimated accurately with a few samples. The new method provides much smoother results with the same number of samples and has the same computing requirements as the classical solutions [F4].

Figure 11.2: Volumetric ambient occlusion.

Thesis Group 2. Specular material rendering

Environment map based robust multiple refections and refractions

Environment map based reflection and refraction rendering techniques usually suffer from inaccuracy or visible artifacts. Neither they support multiple ray bounces. I have proposed a method that uses multiple layers of environment maps and a safe searching algorithm to reduce visible artifacts, make intersection calculation more accurate, and to enable multiple reflections and refractions. The developed algorithm runs entirely on the GPU at interactive frame rates and provides results close to a ray traced reference [D2].

Figure 11.3: Multiple reflections.

Caustic triangles

I have proposed a caustic generation technique that uses triangle primitives instead of splatted sprites. The new algorithm provides just as high frame rates as classical splatting, but frees up the user from tuning caustic attributes like size and transparency. It can also produce sharper caustic patterns with fewer photon count [J5, J9].

Thesis Group 3. Participating media rendering

Spherical billboards

I have proposed an analytic method to eliminate the main drawbacks of billboard rendering including clipping and popping artifacts. Treating each particle as a sphere, the exact length that a light ray travels inside the particle can be computed and transparency can be changed respectively. I have also presented a complete rendering method that uses spherical billboards and composition to render explosion with dust, fire, smoke, and heat shimmering [J4, D1, J3].

Figure 11.4: Explosion with spherical billboards in a car game.

Hierarchical particle systems for volumetric media

I have shown that hierarchically built particle systems with distance impostors can be efficiently used to simulate high number of particles. I have proposed a real time illumination method that exploits the hierarchical property and special hardware possibilities to simulate light absorbtion in scattering media under changing light conditions and with animated particles [J2].

Figure 11.5: Hierarchical particle systems.

Illumination networks for particle systems

I have introduced a novel method that can simulate light scattering in static participating media by reusing light scattering paths generated with global ray bundles. The proposed method can simulate multiple scattering computed by iteration that spreads illumination in the media. Light path information is stored and refreshed in 2D arrays that can be efficiently implemented on graphics hardware. Changing light conditions and even light sources inside the media can be simulated real-time. I have shown that this method can be used to render realistic clouds with moderate particle count [J1].

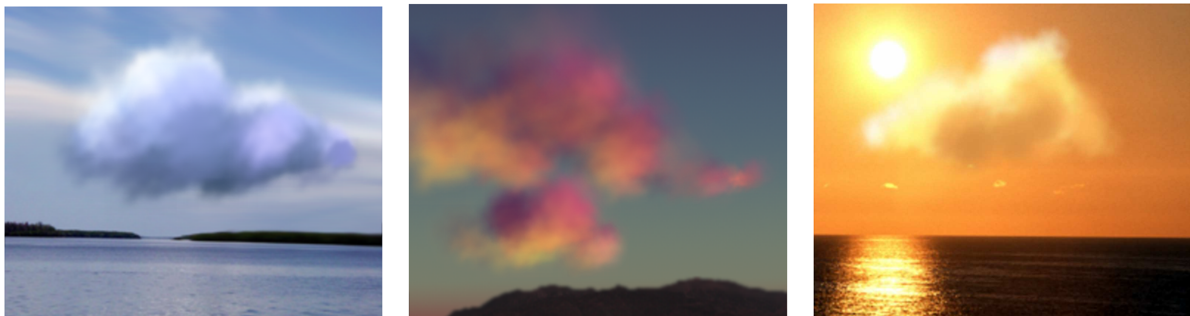


Figure 11.6: Globally illuminated clouds rendered with illumination networks.

Publications

[D1] Tamás Umenhoffer, László Szirmay-Kalos, Gábor Szijártó: Spherical Billboards for Rendering Volumetric Data, in ShaderX^5: Advanced Rendering Techniques (editor: Wolfgang Engel), Charles River Media, Hingham, Massachusetts, pp 275-287, 2006. Citations: 3.

[D2] Tamás Umenhoffer, Gustavo Patow, László Szirmay-Kalos: Robust Multiple Specular Reflections and Refractions, in GPU Gems 3 (editor: Hubert Nguyen), Addison-Wesley Professional, pp 387-408, 2007. Citations: 4.

[D3] Balázs Tóth, László Szirmay-Kalos, Tamás Umenhoffer: Efficient Post-Processing with Importance Sampling, in ShaderX^7: Advanced Rendering Techniques (editor: Wolfgang Engel), Charles River Media, Hingham, Massachusetts, pp 259-276, 2009.

[F1] László Szirmay-Kalos, Tamás Umenhoffer: Displacement Mapping on the GPU - State of the Art, in Computer Graphics Forum 27:(6), pp. 1567-1592, 2008. Citations: 10. IF: 1.107.

[F2] Tamás Umenhoffer, László Szirmay-Kalos, László Szécsi, Balázs Tóth, Mateu Sbert: Global Illumination in Games with Multi-scale PRT, in Computer Graphics & Geometry 10:(2), pp. 2-24, 2008.

[F3] László Szirmay-Kalos, Tamás Umenhoffer, Gustavo Patow, László Szécsi, Mateu Sbert: Specular Effects on the GPU - State of the Art, in Computer Graphics Forum 26:(1), pp. 1-24, 2009. IF: 1.107.

[F4] László Szirmay-Kalos, Tamás Umenhoffer, Balázs Tóth, László Szécsi, Mateu Sbert: Volumetric Ambient Occlusion, in IEEE Compute Graphics and Applications 29:(4), pp. 1-13, 2009. IF: 1.398.

[J1] László Szirmay-Kalos, Mateu Sbert, Tamás Umenhoffer: Real-Time Multiple Scattering in Participating Media with Illumination Networks, in Eurographics Rendering Symposium. Konstanz, Germany, 2005. pp. 277-282. Citations: 9.

[J2] Tamás Umenhoffer, László Szirmay-Kalos: Real-time Rendering of Cloudy Natural Phenomena with Hierarchical Depth Impostors, in Eurographics short paper. Dublin, Ireland, 2005. pp. 65-68. Citations: 1.

[J3] Tamás Umenhoffer, László Szirmay-Kalos: Rendering Fire and Smoke with Spherical Billboards, in III. Hungarian Conference on Computer Graphics and Geometry. Budapest, Hungary, 2005. pp. 24-29.

[J4] Tamás Umenhoffer, László Szirmay-Kalos, Gábor Szijártó: Spherical Billboards and their application for Rendering Explosions, in Graphics Interface. Quebec, Canada, 2006. pp. 57-64. Citations: 6.

[J5] Tamás Umenhoffer, Gustavo Patow, László Szirmay-Kalos: Caustic Triangles on the GPU, in IV. Hungarian Conference on Computer Graphics and Geometry. Budapest, Hungary, 2007. pp. 125-128.

[J6] Balázs Tóth, Tamás Umenhoffer: Global Illumination in Games, in IV. Hungarian Conference on Computer Graphics and Geometry. Budapest, Hungary, 2007. pp. 108-116.

[J7] Tamás Umenhoffer: Interactive Distributed Fluid Simulation on the GPU, in IV. Hungarian Conference on Computer Graphics and Geometry. Budapest, Hungary, 2007. pp. 26-32.

[J8] Tamás Umenhoffer, László Szirmay-Kalos: Robust Diffuse Final Gathering on the GPU, in WSCG. Plzen, Czech Republic, 2007.

[J9] Tamás Umenhoffer, Gustavo Patow, László Szirmay-Kalos: Caustic Triangles on the GPU, in Proceedings of Computer Graphics International: CGI. Istanbul, Turkey, 2008. pp. 222-228. Citations: 2.

[J10] Tamás Umenhoffer, László Szirmay-Kalos: Interactive Distributed Fluid Simulation on the GPU, in MIPRO 2008: Grid and Visualization Systems. Opatija, Croatia, 2008. pp. 236-242. Citations: 1.

[J11] Tamás Umenhoffer, László Szirmay-Kalos, László Szécsi, Balázs Tóth, Mateu Sbert: Partial Multi-Scale Precomputed Radiance Transfer, in Spring Conference on Computer Graphics. Budmerice,Slovakia, 2008. pp. 87-94.

[J12] László Szirmay-Kalos, Gábor Liktor, Tamás Umenhoffer, Balázs Tóth: Fast Approximation of Multiple Scattering in Inhomogeneous Participating Media, in Eurographics Short Papers. Munich, Germany, 2009. pp. 53-56.

[J13] László Szirmay-Kalos, Gábor Liktor, Tamás Umenhoffer, Balázs Tóth, Shree Kumar, Glenn Lupton: Parallel Solution to the Radiative Transport, in Eurographics Symposium on Parallel Graphics and Visualization. Munich, Germany, 2009. pp. 95-102.

[J14] Tamás Umenhoffer, Balázs Tóth, László Szirmay-Kalos: An Inexpensive Ambient Lighting Model, in 7th Conference of the Hungarian Association for Image Processing and Pattern Recognition (KEPAF2009). Budapest, Hungary, 2009. pp. 1-9.

[J15] Tamás Umenhoffer, Balázs Tóth, László Szirmay-Kalos: Efficient Methods for Ambient Lighting, in Spring Conference on Computer Graphics. Budmerice, Slovakia, 2009. pp. 99-106. Paper 9.

[J16] Balázs Tóth, Tamás Umenhoffer: Real-time Volumetric Light-shafts in Participating Media, in 7th Conference of Hungarian Association for Image Processing and Pattern Recognition (KEPAF2009). Budapest, Hungary, 2009. pp. 1-6.

[J17] Balázs Tóth, Tamás Umenhoffer: Real-time Volumetric Lighting in Participating Media, in Eurographics Short Papers. Munich, Germany, 2009. pp. 57-60.

Bibliography

[AG00] A. A. Apodaca and L. Gritz. *Advanced RenderMan: Creating CGI for Motion Picture.* Academic Press, 2000.

[AMS+08] Thomas Annen, Tom Mertens, Hans-Peter Seidel, Eddy Flerackers, and Jan Kautz. Exponential shadow maps. In *GI '08: Proceedings of graphics interface 2008*, pages 155–161, Toronto, Ont., Canada, Canada, 2008. Canadian Information Processing Society.

[Ant04] Gy. Antal. *Global Illumination Methods for Architectural Scenes.* PhD thesis, ELTE, Budapest, 2004.

[Arv86] J. Arvo. Backward ray tracing. In *SIGGRAPH '86 Developments in Ray Tracing*, 1986.

[BB08] Csébfalvi Balázs and Domonkos Balázs. Pass-band optimal reconstruction on the body-centered cubic lattice. In *Vision, Modeling, and Visualization*, pages 71–80, 2008.

[BL08] Tóth Balázs and Szirmay-Kalos László. Deferred shading in distributed visualization. In *MIPRO 2008, GVS*, pages 295–300, 2008.

[BRW89] D.R. Baum, H.E. Rushmeier, and J.M. Winget. Improving radiosity solutions through the use of analytically determined form-factors. *Computer Graphics (SIGGRAPH '89 Proceedings)*, pages 325–334, 1989.

[BT04] Zoe Brawley and Natalya Tatarchuk. Parallax occlusion mapping: Self-shadowing, perspective-correct bump mapping using reverse height map tracing. In Wolfgang Engel, editor, *ShaderX 3*. Charles River Media, Cambridge, MA, 2004.

[Bun05] M. Bunnel. Dynamic ambient occlusion and indirect lighting. In M. Parr, editor, *GPU Gems 2*, pages 223–233. Addison-Wesley, 2005.

[CAM08] Petrik Clarberg and Tomas Akenine-Möller. Exploiting Visibility Correlation in Direct Illumination. *Computer Graphics Forum (Proceedings of EGSR 2008)*, 27(4):1125–1136, 2008.

[CS92] W. Cornette and J. Shanks. Physically reasonable analytic expression for single-scattering phase function. *Applied Optics*, 31(16):31–52, 1992.

[CSKSN05] Sz. Czuczor, L. Szirmay-Kalos, L. Szécsi, and L. Neumann. Photon map gathering on the GPU. In *Eurographics short papers*, pages 117–120, 2005.

[Cso05] F. Csonka. *Véletlen bolyongáson és sztochasztikus iteráción alapuló globális illuminációs módszerek kombinációja.* PhD thesis, ELTE, Budapest, 2005.

[DDSD03] X. Décoret, F. Durand, F. Sillion, and J. Dorsey. Billboard clouds for extreme model simplification. In *SIGGRAPH '2003 Proceedings*, pages 689–696, 2003.

[Don05] William Donnelly. Per-pixel displacement mapping with distance functions. In Matt Pharr, editor, *GPU Gems 2*, pages 409–428. Addison-Wesley, 2005.

[DS05] C. Dachsbacher and M. Stamminger. Reflective shadow maps. In *SI3D '05: Proc. of the 2005 Symp. on Interactive 3D Graphics and Games*, pages 203–231, 2005.

[EAMJ05] M. Ernst, T. Akenine-Moller, and H. W. Jensen. Interactive rendering of caustics using interpolated warped volumes. In *Proceedings Graphics Interface*, pages 87–96, 2005.

[Eve01] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA Corporation, 2001.

[GD01] X. Granier and G. Drettakis. Incremental updates for rapid glossy global illumination. *Computer Graphics Forum (Eurographics '01)*, 20(3):268–277, 2001.

[GGH02] Xianfeng Gu, Steven J. Gortler, and Hugues Hoppe. Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361, 2002.

[GRWS04] R. Geist, K. Rasche, J. Westall, and R. Schalkoff. Lattice-boltzmann lighting. In *Eurographics Symposium on Rendering*, 2004.

[GSHG98] Greger G., P. Shirley, P. Hubbard, and D. Greenberg. The irradiance volume. *IEEE Computer Graphics and Applications*, 18(2):32–43, 1998.

[GSSK05] I. Garcia, M. Sbert, and L. Szirmay-Kalos. Leaf cluster impostors for tree rendering with parallax. In *Eurographics Conference. Short papers.*, 2005.

[Hac04] T. Hachisuka. Final gathering on GPU. In *ACM Workshop on General Purpose Computing on Graphics Processors*, 2004.

[Har02] M. Harris. Real-time cloud rendering for games. In *Game Developers Conference*, 2002.

[Hay02] L. Hayden. Production-ready global illumination. Technical report, SIGGRAPH Course notes 16, 2002. http://www.renderman.org/RMR/Books/ sig02.course16.pdf.gz.

[HBSL03] M. Harris, W. Baxter, T. Scheuermann, and A. Lastra. Simulation of cloud dynamics on graphics hardware. In *Eurographics Graphics Hardware'2003*, 2003.

[HG40] G. Henyey and J. Greenstein. Diffuse radiation in the galaxy. *Astrophysical Journal*, 88:70–73, 1940.

[HH04] S. Hargreaves and M. Harris. Deferred shading. Technical report, http://download.nvidia.com/ developer/presentations/ 2004/6800_Leagues/ 6800_Leagues_Deferred_Shading.pdf, 2004.

[HJ07] J. Hoberock and Y. Jia. High-quality ambient occlusion. In Hubert Nguyen, editor, *GPU Gems 3*, pages 257–274. Addison-Wesley, 2007.

[HL01] M. Harris and A. Lastra. Real-time cloud rendering. *Computer Graphics Forum*, 20(3), 2001.

[HQ07] W. Hu and K. Qin. Interactive approximate rendering of reflections, refractions, and caustics. *IEEE TVCG*, 13(1):46–57, 2007.

[IDN02] Kei Iwasaki, Yoshinori Dobashi, and Tomoyuki Nishita. An efficient method for rendering underwater optical effects using graphics hardware. *Computer Graphics Forum*, 21(4):701–711, 2002.

[IDN03] Kei Iwasaki, Yoshinori Dobashi, and Tomoyuki Nishita. A fast rendering method for refractive and reflective caustics due to water surfaces. *Computer Graphics Forum (Eurographics '03)*, 22(3):601–609, 2003.

[IKSZ03] A. Iones, A. Krupkin, M. Sbert, and S. Zhukov. Fast realistic lighting for video games. *IEEE Computer Graphics and Applications*, 23(3):54–64, 2003.

[Jen01] Henrik Wann Jensen. *Realistic Image Synthesis Using Photon Mapping.* AK Peters, 2001.

[KA06] J. Kontkanen and T. Aila. Ambient occlusion for animated characters. In *Proceedings of the 2006 Eurographics Symposium on Rendering*, 2006.

[KAMJ05] A. W. Kristensen, T. Akenine-Moller, and H. W. Jensen. Precomputed local radiance transfer for real-time lighting design. In *SIGGRAPH 2005*, 2005.

[KBW06] J. Krüger, K. Bürger, and R. Westermann. Interactive screen-space accurate photon tracing on GPUs. In *Eurographics Symposium on Rendering*, pages 319–329, 2006.

[Kel97] A. Keller. Instant radiosity. In *SIGGRAPH '97 Proceedings*, pages 49–55, 1997.

[KH01] A. Keller and W. Heidrich. Interleaved sampling. In *Rendering Techniques 2001 (Proceedings of the 12th Eurographics Workshop on Rendering)*, pages 269–276, 2001.

[LC04] B. D. Larsen and N. Christensen. Simulating photon mapping for real-time applications. In *Eurographics Symposium on Rendering*, pages 123–131, 2004.

[Lev90] M. Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, 9(3):245–261, 1990.

[LLM06] Szécsi László, Szirmay-Kalos László, and Sbert Mateu. Light animation with precomputed light paths on the gpu. In *GI 2006*, pages 187–94, 2006.

[LWX06] B. Li, L.-Y. Wei, and Y.-Q. Xu. Multi-layer depth peeling via fragment sort. Technical Report MSR-TR-2006-81, Microsoft Research, 2006.

[Mit07] Martin Mittring. Finding next gen — CryEngine 2. In *Advanced Real-Time Rendering in 3D Graphics and Games Course - Siggraph 2007*, pages 97–121. 2007.

[MJW07] Stephan Mantler, Stefan Jeschke, and Michael Wimmer. Displacement mapped billboard clouds. Technical Report TR-186-2-07-01, Institute of Computer Graphics, TU Vienna, January 2007.

[MM05] Morgan McGuire and Max McGuire. Steep parallax mapping. In *I3D 2005 Poster*, 2005. http://www.cs.brown.edu/research/graphics/games/ SteepParallax/index.htm.

[MMG06] Jason Mitchell, Gary McTaggart, and Chris Green. Shading in valve's source engine. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*, pages 129–142, New York, NY, USA, 2006. ACM.

[MSC03] A. Méndez, Mateu Sbert, and Jordi Catá. Real-time obscurances with color bleeding. In *SCCG '03: Proceedings of the 19th spring conference on Computer graphics*, pages 171–176, New York, NY, USA, 2003. ACM.

[NC02] K. Nielsen and N. Christensen. Real-time recursive specular reflections on planar and curved surfaces using graphics hardware. *Journal of WSCG*, 10(3):91–98, 2002.

[NFJ02] D. C. Nguyen, R. Fedkiw, and H. W. Jensen. Physically based modeling and animation of fire. In *ACM SIGGRAPH 2002*, 2002.

[Ngu04] H. Nguyen. Fire in the vulcan demo. In R. Fernando, editor, *GPU Gems*, pages 359–376. Addison-Wesley, 2004.

[NN94] T. Nishita and E. Nakamae. Method of displaying optical effects within water using accumulation buffer. In *SIGGRAPH'94 Proceedings*, 1994.

[OB07] Manuel M. Oliveira and Maicon Brauwers. Real-time refraction through deformable objects. In *I3D '07: Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 89–96. ACM Press, 2007.

[OBM00] Manuel M. Oliveira, Gary Bishop, and David McAllister. Relief texture mapping. In *SIGGRAPH 2000 Proceedings*, pages 359–368, 2000.

[Oli00] Manuel M. Oliveira. *Relief Texture Mapping*. PhD thesis, University of North Carolina, 2000.

[Pat95] G. Patow. Accurate reflections through a z-buffered environment map. In *Proceedings of Sociedad Chilena de Ciencias de la Computacion*, 1995.

[PDC+03] T. Purcell, C. Donner, M. Cammarano, H. W Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 41–50, 2003.

[PG04] M. Pharr and S. Green. Ambient occlusion. In *GPU Gems*, pages 279–292. Addison-Wesley, 2004.

[PM07] Fabio Policarpo and Oliveira Manuel. Relaxed cone stepping for relief mapping. In Hubert Nguyen, editor, *GPU Gems 3*. 2007.

[PMDS06] V. Popescu, C. Mei, J. Dauble, and E. Sacks. Reflected-scene impostors for realistic reflections at interactive rates. *Computer Graphics Forum (Eurographics'2006)*, 25(3):313–322, 2006.

[PO05] Fábio Policarpo and Manuel M. Oliveira. Rendering surface details with relief mapping. In Wolfgang Engel, editor, *ShaderX 4: Advanced Rendering Techniques*. Charles River Media, 2005.

[POC05] Fabio Policarpo, Manuel M. Oliveira, and Joao Comba. Real-time relief mapping on arbitrary polygonal surfaces. In *ACM SIGGRAPH 2005 Symposium on Interactive 3D Graphics and Games*, pages 155–162, 2005.

[Pre06] M. Premecz. Iterative parallax mapping with slope information. In *Central European Seminar on Computer Graphics*, 2006. http://www.cescg.org/CESCG-2006/papers/TUBudapest-Premecz-Matyas.pdf.

[PS66] M. Powell and J. Swann. Weighted importance sampling — a Monte-Carlo technique for reducing variance. *Inst. Maths. Applics.*, 2:228–236, 1966.

[PSS99] A. Preetham, P. Shirley, and B. Smits. A practical analytic model for daylight. In *SIGGRAPH '99 Proccedings*, pages 91–100, 1999.

[Ree83] W. T. Reeves. Particle systems - techniques for modelling a class of fuzzy objects. In *SIGGRAPH '83 Proceedings*, pages 359–376, 1983.

[REK+04] K. Riley, D. Ebert, M. Kraus, J. Tessendorf, and C. Hansen. Efficient rendering of atmospheric phenomena. In *Eurographics Symposium on Rendering*, pages 374–386, 2004.

[RH01] R. Ramamoorthi and P. Hanrahan. An efficient representation for irrandiance environment maps. *SIGGRAPH 2001*, pages 497–500, 2001.

[RSP06] Eric A. Risser, Musawir A. Shah, and Sumanta Pattanaik. Interval mapping poster. In *Symposium on Interactive 3D Graphics and Games (I3D)*, 2006.

[SA07] Perumaal Shanmugam and Okan Arikan. Hardware accelerated ambient occlusion techniques on GPUs. In *Proceedings of the 2007 Symposium on Interactive 3D graphics*, pages 73–80, 2007.

[Sai08] Miguel Sainz. Real-time depth buffer based ambient occlusion. In *Games Developers Conference '08*. 2008.

[Sch95] G. Schaufler. Dynamically generated impostors. In *I Workshop - Virtual Worlds - Distributed Graphics*, pages 129–136, 1995.

[Sch97] G. Schaufler. Nailboards: A rendering primitive for image caching in dynamic scenes. In *Eurographics Workshop on Rendering*, pages 151–162, 1997.

[SH81] R. Siegel and J. R. Howell. *Thermal Radiation Heat Transfer*. Hemisphere Publishing Corp., Washington, D.C., 1981.

[SK03] G. Szijártó and J. Koloszár. Hardware accelerated rendering of foliage for real-time applications. In *Spring Conference of Computer Graphics '03*, 2003.

[SKAL05] L. Szirmay-Kalos, B. Aszódi, and I. Lazányi. Ray-tracing effects without tracing rays. In Wolfgang Engel, editor, *ShaderX 4: Lighting & Rendering*. Charles River Media, 2005.

[SKALP05] L. Szirmay-Kalos, B. Aszódi, I. Lazányi, and M. Premecz. Approximate ray-tracing on the GPU with distance impostors. *Computer Graphics Forum (Eurographics '05)*, 24(3):695–704, 2005.

[SKe95] L. Szirmay-Kalos (editor). *Theory of Three Dimensional Computer Graphics.* Akadémia Kiadó, Budapest, 1995. http://www.iit.bme.hu/˜szirmay.

[SKL06] L. Szirmay-Kalos and I. Lazányi. Indirect diffuse and glossy illumination on the GPU. In *SCCG 2006*, pages 29–35, 2006.

[SKP98] L. Szirmay-Kalos and W. Purgathofer. Global ray-bundle tracing with hardware acceleration. In *Rendering Techniques '98*, pages 247–258, 1998.

[SP07] M. Shah and S. Pattanaik. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics (TVCG)*, 13(2):272–280, 2007.

[SWZ96] P. Shirley, C. Wang, and K. Zimmerman. Monte Carlo techniques for direct lighting calculations. *ACM Transactions on Graphics*, 15(1):1–36, 1996.

[Szi05] G. Szijártó. 2.5 dimensional impostors for realistic trees and forests. In Kim Pallister, editor, *Game Programming Gems 5*, pages 527–538. Charles River Media, 2005.

[Tat06a] 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. ACM Press, 2006.

[Tat06b] N. Tatarchuk. Practical parallax occlusion mapping with approximate soft shadows for detailed surface rendering. In Wolfgang Engel, editor, *ShaderX 5*, pages 75–105. Charles River Media, Boston, 2006.

[TS00] C. Trendall and A. Stewart. General calculations using graphics hardware, with application to interactive caustics. In *Rendering Techniques 2000*, pages 287–298, 2000.

[War94] G. Ward. Adaptive shadow testing for ray tracing. In *Rendering Workshop '94*, pages 11–20, 1994.

[WB05] M. Wimmer and J. Bittner. Hardware occlusion queries made useful. In *GPU Gems 2*, pages 91–108. Addison-Wesley, 2005.

[WBS03] I. Wald, C. Benthin, and P. Slussalek. Interactive global illumination in complex and highly occluded environments. In *14th Eurographics Symposium on Rendering*, pages 74–81, 2003.

[WD06a] C. Wyman and C. Dachsbacher. Improving image-space caustics via variable-sized splatting. Technical Report UICS-06-02, University of Utah, 2006.

[WD06b] C. Wyman and S. Davis. Interactive image-space techniques for approximating caustics. In *Proceedings of ACM Symposium on Interactive 3D Graphics and Games*, 2006.

[Wei06] Daniel Weiskopf. *GPU-Based Interactive Visualization Techniques (Mathematics and Visualization)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[Wel04] T. Welsh. Parallax mapping with offset limiting: A perpixel approximation of uneven surfaces. Technical report, Infiscape Corporation, 2004.

[WFA+05] B. Walter, S. Fernandez, A. Arbree, K. Bala, M. Donikian, and D. P. Greenberg. Lightcuts: A scalable approach to illumination. In *SIGGRAPH 2005*, 2005.

[Wil78] L. Williams. Casting curved shadows on curved surfaces. In *Computer Graphics (SIGGRAPH '78 Proceedings)*, pages 270–274, 1978.

[WLMK02] X. Wei, W. Li, K. Mueller, and A. Kaufman. Simulating fire with texture splats. In *IEEE Visualization '02*, 2002.

[WS03] M. Wand and W. Straßer. Real-time caustics. *Computer Graphics Forum (Eurographics '03)*, 22(3):611–620, 2003.

[Wym05] C. Wyman. Interactive image-space refraction of nearby geometry. In *GRAPHITE'05*, pages 205–211, 2005.

[Wym07] C. Wyman. Hierarchical caustic maps. Technical Report UICS-07-04, University of Utah, 2007.

[YJ04] K. Yerex and M. Jagersand. Displacement mapping with ray-casting in hardware. In *Siggraph 2004 Sketches*, 2004.

[ZHL+05] K. Zhou, Y. Hu, S. Lin, B. Guo, and H.-Y. Shum. Precomputed shadow fields for dynamic scenes. In *SIGGRAPH 2005*, 2005.

[ZIK98] S. Zhukov, A. Iones, and G. Kronin. An ambient light illumination model. In *Proceedings of the Eurographics Rendering Workshop*, pages 45–56, 1998.

Index

2.5D impostors, 16, 70, 82, 84
3D API, 5
3D texture, 75, 84

absorbtion, 3, 26, 74
abstract light sources, 4
albedo, 5, 74
ambient light, 4
ambient occlusion, 5, 10, 37, 38
area light source, 30

billboard, 10, 16, 70, 82
billboard clipping artifact, 10, 70, 84
billboard cloud, 16, 17
billboard popping artifact, 10, 70
binary search, 14, 22
black-body, 75
blending, 7, 74, 77, 80, 83
blending operator, 8
body centered lattice, 25
BRDF, 2

camera, 6
Cartesian grid, 25
caustic cube map, 65
caustic generator, 24, 62
caustic mapping, 23
caustic receiver, 24, 61
caustic triangles, 61, 62
clipping, 7, 64
color bleeding, 5
composition, 77
containment test, 38, 43
convolution, 31
cube map, 5, 20, 52

deferred shading, 17
depth buffer, 7, 71
depth buffer algorithm, 7
depth impostor, 82, 83
depth map, 5, 11
depth peeling, 54, 90
depth test, 7
diffuse, 5, 9, 19
diffuse reflectivity, 5
Dirac-delta, 20, 40
Direct3D, 5
discretized volumetric rendering equation, 26
displacement mapping, 11, 12, 40, 49, 95
distance impostor, 20
distance map, 11, 52, 62
double layered depth impostors, 82
dynamically generated impostor, 16

emission, 2, 3
entry point, 12
environment map, 10, 19, 24, 52
environment mapping, 19
exit point, 13

face centered lattice, 25
false position method, 14
fat buffer, 17
final gathering, 9, 29, 83
fluid simulation, 95
fragment shader, 7
Fresnel, 22, 52, 55, 63
fuzzy measure, 37

geometric factor, 29
geometry buffer, 17
global illumination, 8
GPU, 5

height field, 40, 49
height map, 11, 12
hemisphere, 30, 41
Henyey-Greenstein phase function, 3, 75
hierarchical depth impostor, 82

illuminating point, 3
illumination information generation, 9
illumination network, 88, 89
implicit surfaces, 38
impostor, 11, 15
in-scattering, 3, 26, 75, 82–84
indirect illumination, 9, 29
instant radiosity, 19
inter-object reflection, 22
interleaved sampling, 45
isotropic phase function, 3

layered distance map, 52–54, 63, 95
light cube map, 65
light map, 17
light paths, 23
light shafts, 26
light slices, 83
linear interpolation, 7, 84
linear search, 15, 21, 54
local illumination, 4

macrostructure geometry, 12, 49

merging, 7
modelling, 6
Monte Carlo, 19, 88
multiple reflections, 10, 52, 58
multiple render targets, 8

nailboard, 16, 70
natural phenomena, 71, 82
normal map, 12
normalized screen space, 7

obscurances, 5
Ogre3D, 34, 66
OpenGL, 5
outscattering, 3
overshooting, 21, 53

parallax correction, 21
parallax mapping, 13
participating media, 2, 3, 37
particle, 25
particle block, 82
particle system, 11, 25, 26, 70
perspective, 6
phase function, 3, 75
photon hit location image, 24, 64
photon mapping, 24
photon tracing, 23, 61, 63
Planck's formula, 76
point-to-point form factor, 31
Poisson-disk distribution, 41
polygon-to-point form factor, 31
projection, 6

radiance, 2
rasterization, 4, 7
ray marching, 25
ray-marching, 15, 21
ray-tracing, 12, 20, 24, 43, 53, 61, 62
reference point, 11, 30
reflected radiance, 2, 30, 37
regular grid, 25, 95
relief mapping, 15
render call, 5
render state, 5
render target, 8, 77, 91
render-to-texture, 8, 9
rendering equation, 2, 30

screen space, 7
screen space ambient occlusion, 10
screen-space ambient occlusion, 5, 40
secant method, 14, 15
secant search, 14, 21, 53
self-shadowing, 30, 49
set of surface points, 29
shaded point, 3
Shader Model 3, 5, 8, 26, 64
Shader Model 4, 65, 95
shadow, 62
shadow caster, 18
shadow map, 18, 24
shadow mapping, 18
shadow receiver, 18
single reflection, 22, 30
single reflections, 52
Snellius-Descartes law, 22
spectral, 5
spherical billboard, 70, 71, 75
spherical harmonics, 20
splatting, 25, 61
steep parallax mapping, 15
step function, 40
subsurface scattering, 95

tangent plane, 30
tangent space, 12
tangent sphere, 42
tessellation, 6
texture, 6–8, 11, 15, 61, 71, 90
texture coordinate, 6, 7, 11
texture slicing, 25
texture space, 11, 24
triangle meshes, 6

undershooting, 21, 53
UV atlas, 11

vertex buffer, 6
vertex shader, 6, 12, 26
viewport, 6
virtual light source, 19, 30
visibility indicator, 29
visibility network, 89
volumetric ambient occlusion, 37, 42
volumetric integral, 42
volumetric rendering equation, 4

wavelets, 20
weighted importance sampling, 45
world space, 6
world transformation, 6

z-buffer, 83
z-buffer algorithm, 4

Nomenclature

Ω	domain of directions
θ	outgoing angle
θ'	incident angle
$\vec{\omega}$	outgoing direction
$\vec{\omega}'$	incoming direction
$\vec{B}$	surface binormal direction
$\vec{L}$	light direction
$\vec{N}$	surface normal direction
$\vec{R}$	reflected direction
$\vec{T}$	surface tangent direction
$\vec{V}$	view direction
$\vec{x}$	shaded surface point
$\vec{y}$	visible surface point
d	distance
f_r	bidirectional refection distribution function (BRDF)
h	height
L	radiance
L^e	emitted radiance
L^r	reflected radiance
$P(\vec{\omega}', \vec{\omega})$	phase function
t	line parameter