UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

Voxel Cone Tracing

Bachelorarbeit

zur Erlangung des Grades eines Bachelor of Science (B.Sc.)
im Studiengang Computervisualistik

vorgelegt von

Robin Dinse
rdinse@uni-koblenz.de

Erstgutachter:	Prof. Dr. Stefan Müller Institut für Computervisualistik
Zweitgutachter:	Gerrit Lochmann, M. Sc. Institut für Computervisualistik, Arbeitsgruppe Müller

Koblenz, im Februar 2015

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

	Ja	Nein
Mit der Einstellung der Arbeit in die Bibliothek bin ich einverstanden.	☐	☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	☐	☐

.....

(Ort, Datum) (Unterschrift)

Zusammenfassung

Die vorliegende Arbeit befasst sich mit dem Echtzeit-Rendering indirekter Beleuchtung und anderer globaler Beleuchtungseffekte mit Hilfe des Voxel-Cone-Tracing-Verfahrens, das von Crassin et al. entwickelt wurde [Cra+11]. Voxel-Cone-Tracing ist eines der ersten Verfahren, die eine annäherungsweise Berechnung von dynamischen, indirekten Beleuchtungseffekten in Echtzeit ermöglichen und dabei nicht auf Offline-Berechnungen im Vorhinein beruhen. Im Gegensatz zu lokalen Beleuchtungsmodellen, wie dem Phong-Modell, ist es bei indirekter Beleuchtung notwendig, die Interreflektionen zwischen allen Oberflächen in der Szene zu berücksichtigen, was die Berechnungszeit maßgeblich erhöht. Beim Voxel-Cone-Tracing wird die Szene zunächst in eine hierarchische Voxel-Repräsentation übergeführt, die es erlaubt indirekte Beleuchtungseffekte durch das Cone-Tracing mit hohen Bildraten zu rendern. Ziel dieser Arbeit ist es, einen Renderer zu implementieren der von dem Verfahren Gebrauch macht. Abschließend werden Verbesserungsmöglichkeiten sowie Vor- und Nachteile des Verfahrens diskutiert.

Abstract

The present work deals with real-time rendering of indirect illumination and other global illumination effects using voxel cone tracing, a rendering technique that was presented by Crassin et al. [Cra+11]. Voxel cone tracing is one of the first techniques that allow for approximated calculation of fully dynamic indirect illumination effects in real-time without relying on pre-computed solutions. In contrast to local illumination models, such as the Phong model, for indirect lighting it is necessary to calculate interreflections between the surfaces in the scene which vastly increases the computational complexity. Using voxel cone tracing, the scene is converted into a hierarchical voxel representation which allows for cone tracing of secondary lighting effects at interactive frame rates. The goal of this thesis is to implement a simple GPU-based renderer using voxel cone tracing.

Contents

1 Introduction

As illustrated in Figure 1, we can distinguish several ways in which light is reflected in a scene. A fundamental distinction is whether the light is reflected only once (*direct light*) or multiple times (*indirect light*) before it reaches the eye. Early lighting models in real-time graphics were limited to direct light, because it is fast to compute. Once it is determined whether a surface point is visible from the light source, its lighting calculations are independent of other objects in the scene and can be fully described by local information such as material, normal and light direction information. The main disadvantage of this approach is, however, that it produces a flat and unrealistic appearance of the virtual scene (see Figure 2.a).

Lighting algorithms that additionally describe the light at a certain surface point as a function of other geometries in the scene are hence called *global illumination* algorithms. Since light is naturally scattered many times before reaching the eye, global illumination algorithms are necessary for synthesizing naturally looking images of a virtual scene. Moreover, global lighting effects are known to provide visual cues that help the viewer to understand the volumetric structure and the spatial relationships of objects in the scene [Sto$^+$04; AMHH08]. For these reasons, there is a high interest in developing real-time global illumination techniques, most notably in the entertainment, architecture and design industry.

In this theses we will review the voxel cone tracing technique by developing a renderer that demonstrates voxel cone tracing-based global illumination effects in real-time.

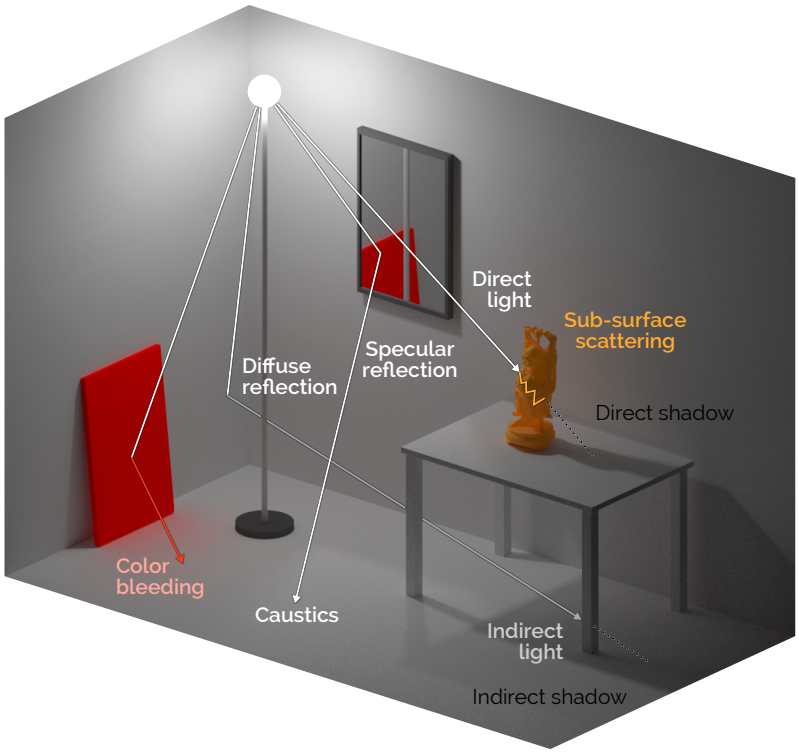


Figure 1. *Example scene with global illumination.* The diffusely reflected rays on the left wall indirectly lighten the floor under the table. The mirror on the wall is an example for specular reflection and it also casts *caustics* both on the floor and on the table which create an *indirect shadow*. Light reflected on the red canvas produces a *color bleeding* effect, and the *Happy Buddha* wax sculpture on the table (courtesy of Stanford University) is an example for light scattering in a semi-translucent object.

(a) Omitting indirect lighting results in strong contrasts between lit and shaded regions. A constant brightening term would make these regions appear unshaded and flat.

(b) *Single indirect light bounce.* Simulating a single indirect bounce lightens shaded regions considerably and enhances the realism of the resulting image.

(c) *Multiple indirect light bounces.* In many scenes, more indirect bounces have a diminishing effect since the probability of absorption increases with each bounce.

Figure 2. *Direct and indirect illumination.* (Images courtesy of PDI/DreamWorks [TL04]).

1.1 Thesis Outline

This thesis is organized in four main parts. In the following part (Chapter 2) we provide a brief introduction to the mathematical framework used for light transport calculations and we give a compact overview of rendering techniques for real-time global illumination. At the end of this part we outline the voxel cone tracing technique. Chapter 3 contains a detailed description of the implemented renderer that makes use of voxel cone tracing to produce indirect illumination, as well as soft shadows and ambient occlusion effects. In Chapter 4 we evaluate the results of the implementation and Chapter 5 summarizes the results and discusses possible ways of improvement of the implemented rendering method.

2 Theoretical Background

Lighting calculations are based on the interaction of electromagnetic radiation and matter. Depending on the circumstances, light propagation is described with different mathematical models. In increasing order of complexity, these models are either based on *rays* using geometric optics, on *electromagnetic waves*, or on *photons* described by particle physics. The wavelength of a photon (which manifests as the color) is, however, the only wave property that is always prominently noticeable to us. For this reason, even in photorealistic image synthesis it usually suffices to model light propagation with geometric optics [AMHH08].

For calculating the light propagation in a geometric scene it is useful to have several radiometric measures at hand. The most basic one is the *radiant energy* Q which is the total energy of a collection of photons and is measured in *joules*. Since light sources emit a flow of photons one considers the radiant energy per time unit, a quantity called *radiant flux* (or *power*) which is measured in *watts*:

$$\Phi = \frac{\mathrm{d}Q}{\mathrm{d}t} \quad \left[\frac{J}{s} = W\right].$$

All other quantities in radiometry are densities of flux. *Irradiance* (E) is a flux area density, a measure for the flux that is incident upon a differential surface segment from all directions. The surface segment $\mathrm{d}A$ can generally be imaginary or part of an object in the scene.

$$E = \frac{\mathrm{d}\Phi}{\mathrm{d}A} \quad \left[\frac{W}{m^2}\right].$$

The area density of flux *leaving* the surface is called *radiosity*. Similarily, we can measure the flux that is incident upon a surface from a certain direction

$\mathrm{d}\boldsymbol{\omega}$. The direction is expressed as a *solid angle* and measured in *steradians*. The flux solid angle density is called *radiant intensity*:

$$I = \frac{\mathrm{d}\Phi}{\mathrm{d}\boldsymbol{\omega}} \quad \left[\frac{W}{\mathrm{sr}}\right].$$

Finally, we define *radiance* (L) as a measure for the flux at a differential surface segment $\mathrm{d}A$ coming from or leaving towards a certain direction $\mathrm{d}\boldsymbol{\omega}$. We define this quantity as the radiant intensity per *unit projected area normal to the beam direction* $A_\perp$:

$$L = \frac{\mathrm{d}I}{dA_\perp} = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\boldsymbol{\omega}\,\mathrm{d}A_\perp} = \frac{\mathrm{d}^2\Phi}{\mathrm{d}\boldsymbol{\omega}\,\mathrm{d}A\cos\theta} \quad \left[\frac{W}{m^2\,\mathrm{sr}}\right].$$

2.1 The Rendering Equation

The interactions of light under geometric optics approximation can be expressed by the *Rendering Equation* [Kaj86] which defines the outgoing radiance L_o at a surface point $\boldsymbol{x}$ in direction $\boldsymbol{\omega}_\mathrm{o}$ as the sum of emitted radiance L_e and the reflected radiance L_r:

$$\begin{aligned} L_\mathrm{o}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{o}) &= L_\mathrm{e}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{o}) + L_\mathrm{r}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{o}) \\ &= L_\mathrm{e}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{o}) + \int_{\boldsymbol{\omega}_\mathrm{i}\in\Omega^+} L_\mathrm{i}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{i})\, f_\mathrm{r}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{i}\to\boldsymbol{\omega}_\mathrm{o})\, \langle N(\boldsymbol{x}),\boldsymbol{\omega}_\mathrm{i}\rangle^+ \,\mathrm{d}\boldsymbol{\omega}_\mathrm{i}\,. \end{aligned} \tag{2.1}$$

The reflected radiance is the weighted incident radiance L_i coming from all directions on the upper unit hemisphere Ω^+ centered at the surface point $\boldsymbol{x}$ and oriented around the surface normal $N(\boldsymbol{x})$. The vector $\boldsymbol{\omega}_\mathrm{i}$ is the negative direction of the incoming light, $\langle\cdots\rangle^+$ is a dot product that is clamped to zero, and f_r is the *bidirectional reflectance distribution function* or *BRDF*. The dot product is a weakening factor that accounts for the increasing incident area relative to the projected area perpendicular to the ray as the incident angle increases.

The BRDF describes the reflectance properties at a surface point $\boldsymbol{x}$ when viewed from direction $\boldsymbol{\omega}_\mathrm{o}$. It is defined as the ratio of the radiance that is reflected toward $\boldsymbol{\omega}_\mathrm{o}$ and the incoming radiance from direction $\boldsymbol{\omega}_i$:

$$f_\mathrm{r}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{i}\to\boldsymbol{\omega}_\mathrm{o}) = \frac{\mathrm{d}L_\mathrm{o}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{o})}{\mathrm{d}L_\mathrm{i}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{i})\cos\theta_i\,\mathrm{d}\boldsymbol{\omega}_i} = \frac{\mathrm{d}L_\mathrm{o}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{o})}{\mathrm{d}E_\mathrm{i}(\boldsymbol{x},\boldsymbol{\omega}_\mathrm{i})}. \quad \left[\frac{1}{\mathrm{sr}}\right] \tag{2.2}$$

Physically based BRDFs must be both symmetric $f_\mathrm{r}(\boldsymbol{\omega}_\mathrm{i}\to\boldsymbol{\omega}_\mathrm{o}) = f_\mathrm{r}(\boldsymbol{\omega}_\mathrm{o}\to\boldsymbol{\omega}_\mathrm{i})$ and energy-conserving, i. e. that $\forall\boldsymbol{\omega}_\mathrm{o}\int_{\Omega^+} f(\boldsymbol{\omega}_\mathrm{i}\to\boldsymbol{\omega}_\mathrm{o})\mathrm{d}\boldsymbol{\omega}_\mathrm{i} \leq 1$. Two

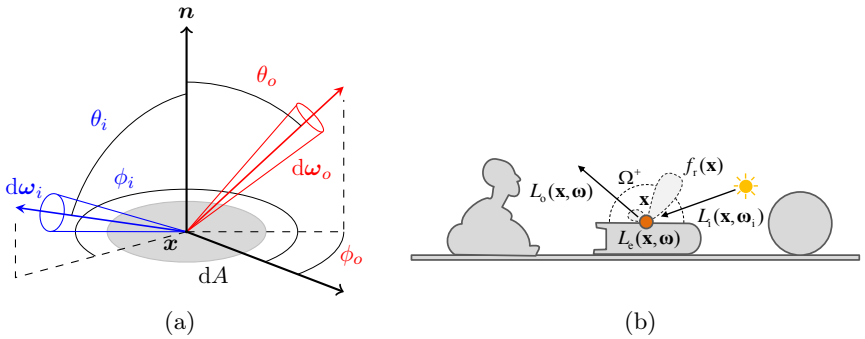


Figure 3. *Visualization of the BRDF and the Rendering Equation.* The BRDF (**a**) relates the incoming radiance from direction $\boldsymbol{\omega}_i$ with the outgoing radiance in direction $\boldsymbol{\omega}_o$ at a surface point $\boldsymbol{x}$ and thereby describes the scattering properties of a surface. θ and ϕ are the corresponding spherical coordinates of the solid angles. The illustration of the Rendering Equation (**b**) is courtesy of [Rit+12].

extreme cases of the BRDF are the *Lambertian* BRDF which is constant for any pair ($\boldsymbol{\omega}_\mathrm{i}$, $\boldsymbol{\omega}_\mathrm{o}$), and the *perfectly specular* BRDF which is a Dirac delta function in the reflected viewing direction. Many surfaces can be modelled as a combination of these two extremes [Rit+12].

All of the functions defined above can additionally be parameterized by the wavelength λ which allows modelling of colored materials and lights.

2.2 Rendering Techniques

The Rendering Equation is very difficult to compute for several reasons. One reason is that the hemisphere Ω^+ describes a continuous space of solid angles which means that infinitely many directions and intersections with the scene geometry must be regarded. Moreover, the integral in Equation 2.1 is defined in terms of itself (notice the L_i on the right-hand side) which makes it a type of *integral equation* to which no general algebraic solution is known [McG13]. For these reasons, computations of the Rendering Equation generally involve approximations.

There are several different approaches to approximating the Rendering Equation. In photo-realistic rendering the most important techniques include *photon mapping* [Jen96], finite element methods such as the *radiosity method* [Gor+84] and Monte Carlo-based methods such as *bidirectional path tracing* [LW93]. These methods produce near-ideal results but are not directly suitable for interactive applications. Instead, we make use of algorithms that are specifically tailored for highly parallel computing on GPUs. A selection of notable real-time techniques are briefly summarized in the following list.

- *Manual placement of lights.* Since direct lighting models are computationally cheap, it is still common practice that 3D artists manually place numerous point lights throughout the scene to achieve diffuse indirect illumination effects. However, while this approach enables fine-grained artistic decisions over the lighting, it is often impractical for scenes with dynamic lights and requires additional work.

- *Virtual point lights.* A common approach to fully dynamic real-time GI is based on a rendering method called *Instant Radiosity* [Kel97] in which a number of *virtual point light* are generated at positions where random light paths from the light source hit the surfaces in the scene. Secondary lighting is efficiently accumulated by calculating the radiance that comes directly from each VPL. For visibility tests, however, shadow maps need to be calculated for each VPL which is the main disadvantage of this approach.

- *Reflective shadow maps.* In this technique, shadow maps are used as the source of secondary lighting [DS05]. It is based on the observation that the directly lit surfaces are the only sources for secondary light bounces. To allow a fast computation, however, occlusion information is neglected when sampling the indirect light from the RSMs, which may result in implausible light effects.

- *Photon mapping.* This approach works in two main steps: First, rays are traced from the light sources throughout the scene. At each point where the light is reflected a photon is deposited on the surface and cached in a space-partitioning data structure. Afterwards, the photons are gathered from the visible surfaces to estimate radiance values. The GPU-based photon mapping technique by McGuire et al. [ML09] exploits the fact that both in the photon emission and in the gathering step, all rays have a common center of projection and can thus be accelerated using the rasterization pipeline. The tracing of secondary bounces and beyond, however, takes place on the CPU.

- *Screen space approaches.* These techniques work exclusively with the scene geometry that is left after the clipping is performed or even only with the front-most fragments in the framebuffer. For example, in the *screen space directional occlusion* method by Ritschel et al. [RGS09], approximated indirect lighting is efficiently calculated from framebuffers with normal, position and direct light information. The main limitation is, however, that scene information beyond the field of view is not included which may result in missing shadows and reflections.

- *Light propagation volumes.* Like voxel cone tracing, the LPVs method [KD10] is based on a volumetric data structure. First, any light emitted from surfaces and area lights is accumulated in a coarse volumetric grid. The light is then iteratively propagated throughout the scene in the LPV grid using a diffusion algorithm. Indirect light can then be sampled from the LPV grid. This approach is, however, limited to diffuse reflections and often produces noticeable light leaking.

2.3 Voxel Cone Tracing

Currently, there is a trend toward more general computations on GPUs as graphics hardware and APIs provide increasingly general functions to read and manipulate video memory and more flexible ways to invoke GPU programs. The *Image Load Store* extension exposed in OpenGL 4.2, for example, enables us to gain random read/write access to texture images and buffer memory from any stage of the pipeline. Moreover, *compute shaders* enable computations of arbitrary information on the GPU since OpenGL 4.3. Voxel cone tracing makes use of both of these extensions to calculate global illumination effects in real-time. The whole procedure works as follows:

In a proceeding step, we voxelize the scene. As in [CG12], we use the Image Load Store extension in conjunction with a framebuffer without attachments (enabled by the `ARB_framebuffer_no_attachments` extension) to write the produced fragments into a 3D texture. For each fragment, we use a diffuse, direct lighting model to compute a radiosity value which is stored in the corresponding voxel together with an opacity value. In contrast to the original method [Cra+11], the voxel data is not sparsely encoded in an octree. The consequence of this is that the memory usage is considerably higher, but it also simplifies the program.

The voxel representation is then used to approximate the integral of the incoming radiance values $L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o})$ in the Rendering Equation (2.1) for indirect light:

$$L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o}) = \int_{\boldsymbol{\omega}_\mathrm{i} \in \Omega^+} L_\mathrm{i}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i})\, f_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i} \to \boldsymbol{\omega}_\mathrm{o})\, \langle N(\boldsymbol{x}), \boldsymbol{\omega}_\mathrm{i} \rangle^+ \, \mathrm{d}\boldsymbol{\omega}_\mathrm{i} \,. \tag{2.3}$$

This is achieved by sampling the incoming radiance L_i with a small set of cone-shaped beams that are distributed across the hemisphere Ω^+. One approach of determining a radiance value in a single cone would be to perform ray marching along the cone axis by sampling the previously generated voxel map in successive steps to accumulate both radiance and opacity values. The traversal can be halted once the accumulated opacity is greater or equal to 1.

This approach, however, would introduce aliasing, and the quantity of subsamples required to eliminate the aliasing would be impractical for real-time purposes.

At this point, one can draw an analogy to 2D texture minification, since in both cases it is necessary to integrate large regions of a texture while keeping aliasing effects at a minimum. A commonly used technique for this problem is to precompute a multi-resolution pyramid of the original texture, a so called *mipmap* [Wil83], by repeatedly downsampling the texture by a factor of two. This can be done, for example, by averaging 2×2 pixel blocks at a time in each mip image and storing the result in the corresponding mip level above. We can then sample the pre-filtered texture at an appropriate mip level instead of relying on a multitude of samples.

The same principle can be applied to 3D textures by regarding $2 \times 2 \times 2$ blocks during the downsampling. The radiance value of a cone is then determined by stepping along the axis of the cone and sampling the pre-filtered voxel data at a mip level with a voxel size that corresponds to the current cone diameter. The radiance and opacity values are interpolated between adjacent texels in the 3D texture and between two mip levels, which results in smooth transitions.

Voxel cone tracing enables us to approximate arbitrary BRDFs f_r (Equation 2.3) by arranging multiple cones with different apertures and weights on the hemisphere Ω^+. A specular BRDF, for example, can be represented by a single narrow cone in the direction of the viewing vector reflected about the normal, and a diffuse BRDF can be achieved with a set of uniformly distributed wide cones ($\sim$5–12 suffice). Especially when using wide cones, this approach allows very fast, approximated evaluation of the lighting integral because the sampling rate can be quickly decreased when stepping through coarser mip levels of the voxel representation. Regarding the solid angle we integrate over, the runtime behavior is thus converse to that of ray tracing. Moreover, relying on a data structure with a fixed resolution also decouples the cone tracing from the geometric complexity of the scene.

In summary, voxel cone tracing allows us to calculate $\mathsf{LD\{S|D\}E}$ light paths in real-time and LDDE with particularly high efficiency. Furthermore, by accumulating only the opacity values of the voxels, the same technique can be employed for soft shadows and ambient occlusion effects which is explained in detail in the next chapter.

3 Implementation

The voxel cone tracing renderer which was implemented in the course of this thesis, is written in the *C++11* programming language and uses the *OpenGL 4.4* graphics API as well as the *OpenGL Shading Language* (GLSL) in version 420 (shader model 5.0). For the scene and material management we make use of the graphics framework *CVK* (provided by Arbeitsgruppe Müller) which is based on the *Open Asset Import Library* (Assimp 3.1.1). We extend parts of the CVK framework to satisfy requirements, e.g. for special framebuffer configurations and for recompilation of shader programs for debugging purposes. Furthermore, we make use of the *AntTweakBar* library (in version 1.16) to provide a simple user interface with options for adjusting several parameters at runtime. The *GLFW library* (version 3.0.4) is used for creating the OpenGL context and handling user input, and we leverage the *OpenGL Mathematics Library* (GLM 0.9.5.4) for GLSL-compliant arithmetics and linear algebra functions and classes.

The main functionality of the program is realized in three singleton objects called `Renderer`, `Context` and `Profiler`. The `Renderer` singleton contains the rendering loop and takes care of the initialization of the shaders, textures, framebuffers and the scene graph. The `Context` singleton encapsulates parts of the user interface, initializes the OpenGL context and the AntTweakBar. The `Profiler` singleton takes care of the time measurements which are obtained using `GL_TIME_ELAPSED` queries that are sent to the GPU at each major stage of the rendering engine. To reduce the variance, the measurements are smoothed with an exponentially weighted moving average. For a simplified overview of the implementation see Figures 4 and 5.

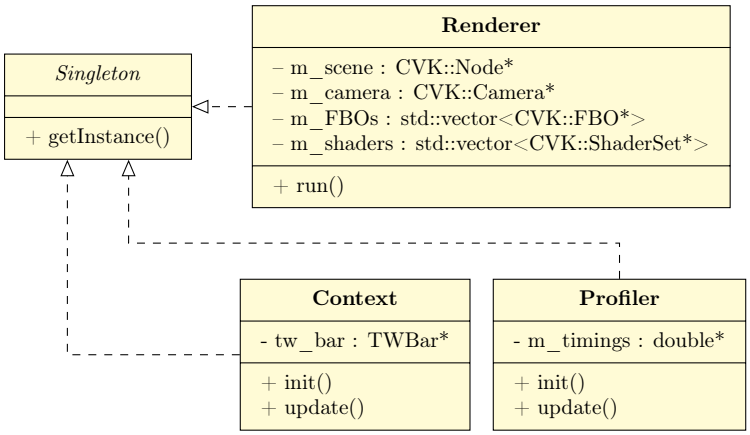


Figure 4. *Simplified class diagram of the basic application structure.*

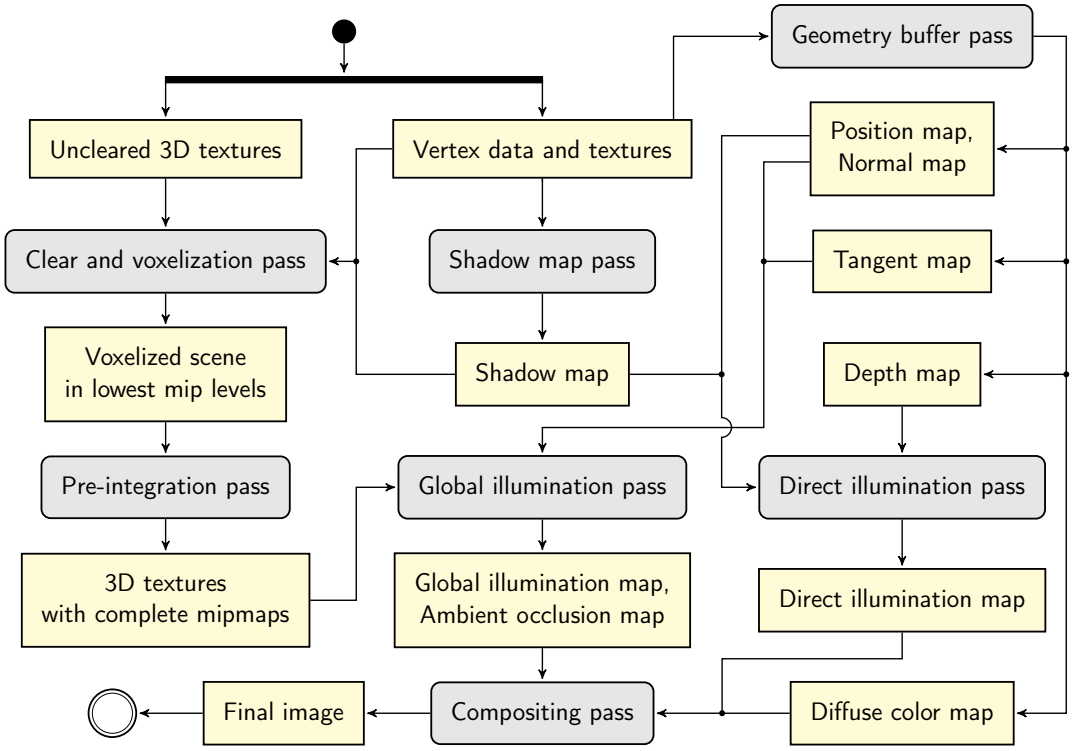


Figure 5. *Data flow diagram of the rendering procedure.*

The presented implementation[1] is based on a deferred rendering approach in which the scene geometry is first rendered to separate buffers for diffuse color, normal, tangent and depth information. Both, during the lighting and compositing stages the shader programs access the geometry buffers instead of re-rasterizing the scene. This way, the overdraw can be saved at a small expense to memory space, which is particularly effective during the expensive voxel cone tracing operations. When the geometry buffer is complete, the shadow map is generated by rendering the scene from the view of the scene light (only a single spot light is supported). In the next step we use the shadow map and the geometry buffer to calculate direct lighting with the *Blinn-Phong* lighting model [Bli77]. Alternatively, voxel cone tracing can be used at a later stage of the renderer to trace soft shadows. After these preceding steps, we voxelize the scene and use the voxel representation to render global illumination effects with cone tracing, which is described in detail in the following sections.

[1]The source code is available at `https://github.com/dinse/VCTGI`.

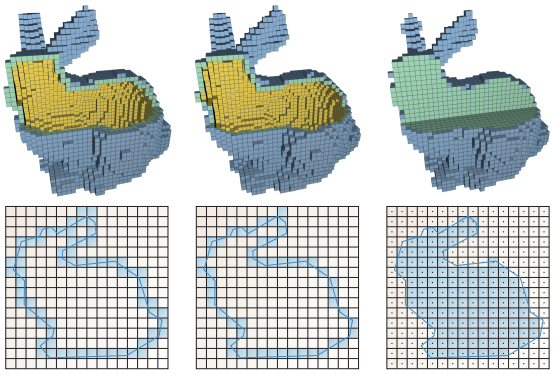


Figure 6. *Examples of different types of voxelization.* The left pair shows a 26-separating (or *conservative*) voxelization, the middle pair a 6-separating (or *thin*) voxelization and the right pair a solid voxelization. In a conservative voxelization, all voxels that are touched by the geometry are regarded. (The illustrations are courtesy of [SS10].)

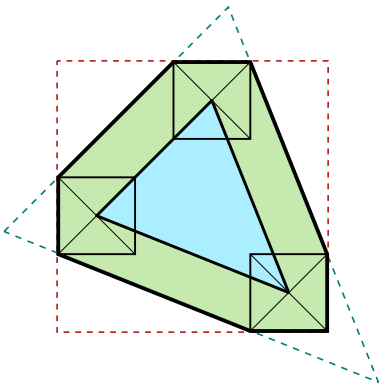


Figure 7. *Conservative triangle rasterization.* The triangle is enlarged to ensure that each pixel cell it touches, produces a fragment. The axis-aligned bounding box and the enlarged triangle are highlighted with dashed lines.

3.1 Voxelization

Analogously, to the rasterization of triangles in two dimensions, there are several ways in which a three-dimensional rasterisation, or voxelization, can be performed. One can distinguish, for example, solid from surface voxelization, *6-* from *26-separating* voxelization and whether it regards anti-aliasing [SS10]. See Figure 6 for visualizations of these properties. In the presented implementation we perform a thin voxelization without addressing aliasing. The lack of anti-aliasing will be especially noticeable as temporal aliasing on moving objects, but it simplifies the voxelization procedure considerably.

We implement the voxelization method described in [CG12] which is based on the rasterization pipeline, the Image Load Store extension and orthogonal projection. The entire scene is re-voxelized in each frame. The voxels are stored in a 3D texture that is generated using `glTexStorage3D` which specifies an uninitialized image for each mip level in one operation. Afterwards, we clear the lowest mip level of the texture using `glClearTexImage`. Higher mip levels may remain uninitialized at this point, because they will be written to during the pre-integration stage. Back face culling and depth testing are disabled, with the result that the triangles produce fragments independent of their orientation and do not interfere with one another. The frame buffer resolution is set in correspondence with the voxel grid of the lowest mip level. In the first step, we determine the direction of projection along one of the

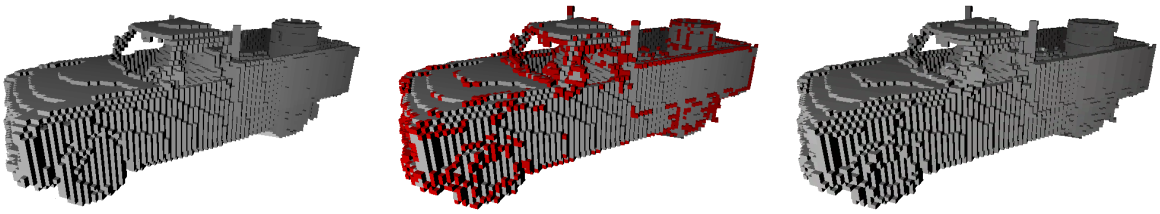


Figure 8. *Voxelization with conservative triangle rasterization.* The left picture shows the voxelized truck without conservative rasterization, while on the right picture the conservative rasterization is enabled. In the middle the differences between the two approaches are highlighted.

three major axes in which the surface area of the triangle is maximized. This calculation takes place in the geometry shader. The resulting projection also maximizes the number of fragments that are produced during the rasterization. It can be shown that using this projection, the voxelized surfaces will not yield any holes and will result in a 6-separating or *thin* surface voxelization [SS10]. The maximizing direction can quickly be determined by finding the maximal component of the absolute normal vector of the triangle. Once the direction of projection is determined, the triangles are projected accordingly and emitted from the geometry shader. In the fragment shader we then light the fragment with a direct lighting model using the earlier generated shadow map and write the resulting radiosity value along with an opacity in the alpha component to the voxel map.

One issue that must be carefully considered is that hardware-based rasterization computes the point-in-triangle tests only for the *fragment centers*, which may result in small triangles being omitted. One way to avoid gaps due to small triangles is to rasterize them *conservatively*. Analogously to the conservative *voxelization* (Fig. 6), conservative rasterization ensures that all fragment cells that the triangle touches result in a fragment shader invocation. This is achieved by enlarging the triangle in the geometry shader so that each edge is shifted outward by the semi-diagonal of a pixel (Fig. 7). The residue at the corners due to the enlargement is clipped in the fragment shader by passing an axis-aligned bounding box which is extended by half the size of a pixel cell in all directions. See Figure 8 for the result.

Another implementation issue is the way in which the lighting information is written to the 3D texture. Since multiple fragments can easily fall into the same voxel location and since the order in which the fragments are generated is not deterministic, flickering may occur. This can be avoided by averaging all values that are written to a voxel. As in [CG12], this is achieved using a

simple moving average which is calculated as

$$\mathrm{A}_{n+1} = \frac{n\,\mathrm{A}_n + x_{i+1}}{n+1}.$$

The fastest way to write to texture images from a shader is to use the `imageStore` operation. This is, however, not suitable for the averaging because the changes made with `imageStore` are not guaranteed to become visible for other threads during the rendering pass. Hence, we need to make use of *atomic* image store operations which are also provided by the Image Load Store extension. One limitation of this approach is that in OpenGL 4.4 atomic image operations only support 32-bit integer values. For this reason, it is necessary to bind the images in the `R32UI` format, in which the color and alpha components are represented by four concatenated unsigned 8-bit integers. To avoid storing the number of samples n in a separate texture, we rely on a 4-bit integer value which is encoded in the least significant bits of the voxel values. Minor disadvantages of this approach are that it reduces the color depth of the voxels by a factor of two and that at most 16 samples can be taken into account, which may lead to flickering when voxelizing highly detailed geometry.

Since the cone tracing pass depends on linear interpolation of the voxel values, it would be advantageous to leverage hardware-based interpolation for this task. To circumvent color leaking from voxels with zero opacity during linear interpolation, it is necessary to store the color components of the voxel values pre-multiplied with the alpha component in the form $(ra, ga, ba, a)^T$ [PD84].

3.2 Pre-Integration

The multi-resolution representation of the scene is generated using a compute shader. Compute shaders are an addition to the GLSL language since OpenGL 4.3 and they allow us to use the GPU for computing arbitrary information. Compute shaders are invocated within a *work group* of many compute shader invocations which in turn is situated in a 3-dimensional space, called *compute space*. Each shader invocation can be identified with a 3D vector within the compute space called `gl_GlobalInvocationID`. We use this ID to address the voxel space.

Using a single texture for the mipmapping would result in a loss of light direction information. To reduce implausible lighting results, the mipmapped radiosity values are thus stored in six textures, one for each of the six directions along the coordinate axes. For illustrations of the light leaking problem see Figures 11(b) and 12. The pre-integration is achieved by composing four voxels

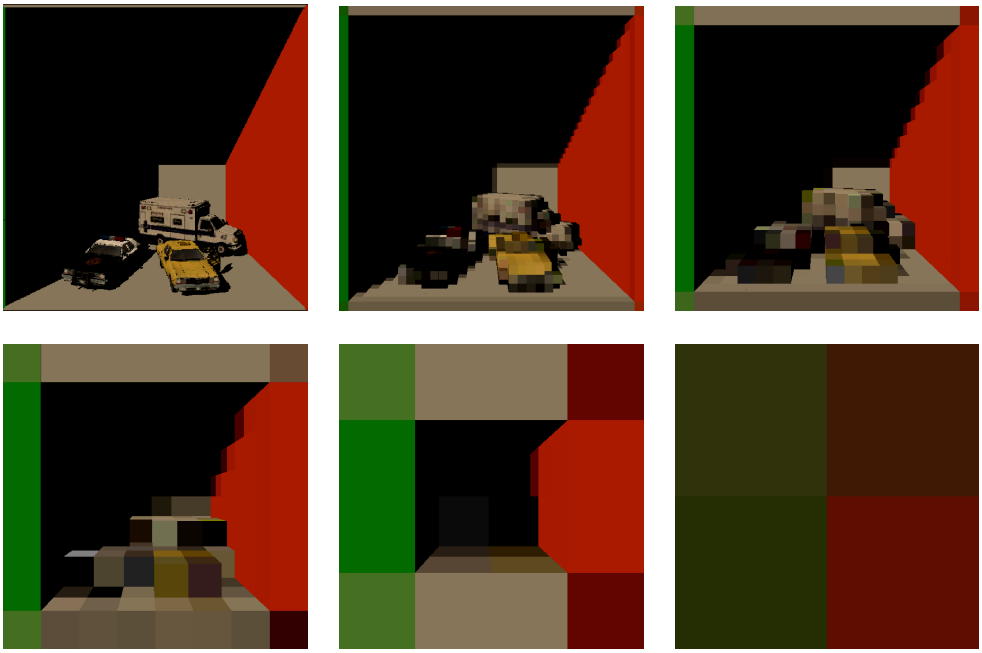


Figure 9. *Visualisation of the mip levels of the directly lit and voxelized scene.* The top left figure shows the full resolution of 256^3 voxels. Furthermore, in reading direction the mip levels 3 through 7 are shown.

at a time, averaging the result and writing it to one of the six 3D textures. To maintain the visibility information of the composed voxels when viewed from one of the six directions, it is necessary to perform a single step of volumetric integration [Cra+11; Max95]. The composed radiosity and opacity value c' and α' for a given pair of voxels c_1 (front voxel) and c_2 (back voxel) with corresponding opacities α_1 and α_2 is calculated as:

$$c' := \alpha_1 \cdot c_1 + (1 - \alpha_1) \cdot \alpha_2 \cdot c_2 \tag{3.1}$$

$$\alpha' := \alpha_1 + (1 - \alpha_1) \cdot \alpha_2 \,. \tag{3.2}$$

See Figure 9 for the result of the pre-integration stage.

3.3 Voxel Cone Tracing

A single cone gathers a radiance value from the hierarchical voxel representation of the scene using the classical emission-absorption optical model [Max95]. Assuming a diffusely reflecting surface, the BRDF f_r in the Rendering Equation 2.1 is constant for all pairs of incoming and outgoing directions as explained in Section 2. Hence, the reflected radiance L_r at a surface point $\boldsymbol{x}$ in

the Rendering Equation can be rewritten as

$$\begin{aligned} L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o}) &= \int\limits_{\boldsymbol{\omega}_\mathrm{i} \in \Omega^+} L_\mathrm{i}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i})\, f_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i} \to \boldsymbol{\omega}_\mathrm{o})\, \langle N(\boldsymbol{x}), \boldsymbol{\omega}_\mathrm{i} \rangle^+ \, \mathrm{d}\boldsymbol{\omega}_\mathrm{i} \\ &= \frac{\rho}{\pi} \int\limits_{\boldsymbol{\omega}_\mathrm{i} \in \Omega^+} L_\mathrm{i}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i})\, \langle N(\boldsymbol{x}), \boldsymbol{\omega}_\mathrm{i} \rangle^+ \, \mathrm{d}\boldsymbol{\omega}_\mathrm{i}, \end{aligned}$$

where ρ is called *albedo* and describes the reflectivity of the surface. Next, we partition the integral into n cones and it is assumed that in each cone the incoming radiance is constant, which allows us to factor out L_i:

$$L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o}) = \frac{\rho}{\pi} \sum_{k=1}^{n} \int\limits_{\boldsymbol{\omega}_\mathrm{i} \in \Omega_k^+} L_\mathrm{i}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{i})\, \langle N(\boldsymbol{x}), \boldsymbol{\omega}_\mathrm{i} \rangle^+ \, \mathrm{d}\boldsymbol{\omega}_\mathrm{i} \quad \overset{L_i \text{ const. for } k}{\Leftrightarrow}$$

$$\begin{aligned} L_\mathrm{r}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{o}) &= \frac{\rho}{\pi} \sum_{k=1}^{n} L_\mathrm{k}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{k}) \int\limits_{\boldsymbol{\omega}_\mathrm{i} \in \Omega_k^+} \langle N(\boldsymbol{x}), \boldsymbol{\omega}_\mathrm{i} \rangle^+ \, \mathrm{d}\boldsymbol{\omega}_\mathrm{i} \\ &= \frac{\rho}{\pi} \sum_{k=1}^{n} L_\mathrm{k}(\boldsymbol{x}, \boldsymbol{\omega}_\mathrm{k})\, W_k. \end{aligned}$$

The weights W_k sum up to π for a diffuse surface [Rau13]. The incoming radiance L_k of a cone is obtained by splitting the cone into successive elements and perform a volumetric *front-to-back* accumulation which uses the same blending of the currently accumulated value and the next sample as used during the pre-integration in Equation 3.2 [HN12]. For glossy BRDFs we trace a single narrow cone in the reflected viewing direction (see Figure 10).

The mip level of a sample at tracing distance d is determined by $\log_2(2r)$, where r is the current radius of the cone at the sample. The six-directional radiosity values are modulated with the absolute components of the normalized cone direction vector. At each step the sampling step size is adapted to the cone diameter of $2r$.

Besides rendering of indirect illumination with arbitrary BRDFs we can use voxel cone tracing for other global illumination effects such as ambient occlusion. The AO value at a surface point can be interpreted as an accessibility value [Mil94]. In an outdoor scene with a diffusely radiating, overcast sky the AO value at a surface point can be thought to depend on the solid angle that is subtended by visible sky. This value can be used to shade the surface point, which results in a diffuse lighting effect where creases and niches are darkened and openly exposed features of the scene appear bright. In an indoor scene only objects within a certain radius are taken into account and the walls are assumed to play the role of diffuse radiators [Cra+11].

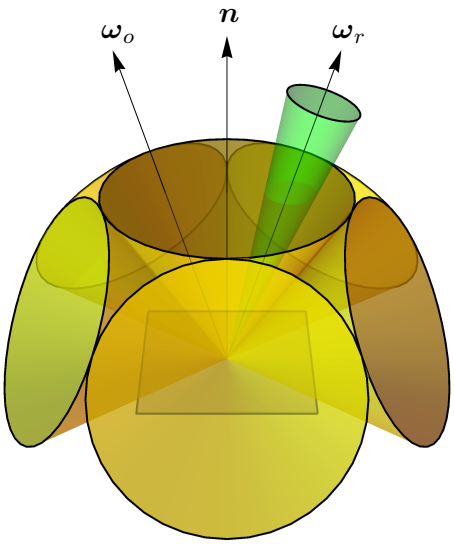


Figure 10. *Illustration of seven cones that are used to approximate a diffuse BRDF with a glossy component.* The six large cones are arranged on the vertices of an icosahedron and capture diffusely reflected incoming radiance; the thin cone is traced along the reflected viewing direction $\boldsymbol{\omega}_r$ to gather specularly reflected light.

AO can be very cheaply calculated during the voxel cone tracing of the diffuse indirect lighting and hence there is no need for an additional rendering stage. It uses the same opacity blending model as described above but each sample is weakened by the distance d and a falloff factor λ so that remote regions do not contribute to the accumulated occlusion information. At each sample from the voxel map with opacity α the AO sample for the current cone is updated according to

$$\mathrm{ao} \leftarrow \mathrm{ao} + \frac{(1 - \mathrm{ao}) \cdot \alpha}{1 + d \cdot \lambda}.$$

Another global illumination effect that can easily be implemented with voxel cone tracing is the *penumbra* or *soft shadow* effect. This is achieved by tracing cones from the fragment toward the light source, accumulating only the opacclusion information and shading the fragment according to the resulting visibility value. For results of this effect see Figure 17.

One problem with using a voxel representation of the scene arises from the fact that the surfaces become inevitably thicker. When starting the cone tracing at a surface point it is very likely that the first samples coincide with the voxelized surface of the point itself. Moreover, the problem is amplified by the thickening effect at grazing angles as illustrated in Figure 11(c). To avoid self-intersection it is hence necessary to shift the apex of the cone at least by the space diagonal of a voxel outward along the normal.

Algorithm 1 *Cone tracing algorithm with empty space skipping.*

Input:

pos	Position of the cone apex in voxel space coordinates.
dir	Direction vector of the cone.
aperture	Diameter of the cone at unit cone axis length.
voxelMap[6]	Contains 6-directional voxel data: $[+X, -X, +Y, -Y, +Z, -Z]$
voxelRes	Number of voxels along one axis of the voxel grid.

Output:

coneSample	The gathered radiance.

```
1:  visibleFace ← ivec4(0)
2:  visibleFace.x ← (dir.x < 0) ? 0 : 1    ▷ ... ? ... : ... is the ternary operator.
3:  visibleFace.y ← (dir.y < 0) ? 2 : 3
4:  visibleFace.z ← (dir.z < 0) ? 4 : 5
5:
6:  dst ← .0
7:  diameter ← .0
8:  coneSample ← vec4(.0)
9:  samplePos ← pos
10: while coneSample.a < 1 do
11:     mipLevel ← max(0, log2(diameter · voxelRes))
12:     voxelSample ←
            abs(dir.x) · textureLod(voxelMap[visibleFace.x], pos, mipLevel)
            + abs(dir.y) · textureLod(voxelMap[visibleFace.y], pos, mipLevel)
            + abs(dir.z) · textureLod(voxelMap[visibleFace.z], pos, mipLevel)
13:
14:     coneSample ← coneSample · coneSample.a
            + (1 − coneSample.a) · voxelSample · voxelSample.a
15:
16:     skipDst ← .0                                   ▷ Empty space skipping
17:     if voxelSample.a < ϵ then
18:         parentVoxel ← findLargestEmptyParentVoxel(samplePos, mipLevel)
19:         skipDst ← distanceToBackPlanes(parentVoxel, samplePos, dir)
20:     end if
21:
22:     dst ← dst + max(diameter, skipDst)
23:     diameter ← dst · aperture
24:     pos ← pos + dst · dir
25: end while
26: return coneSample
```

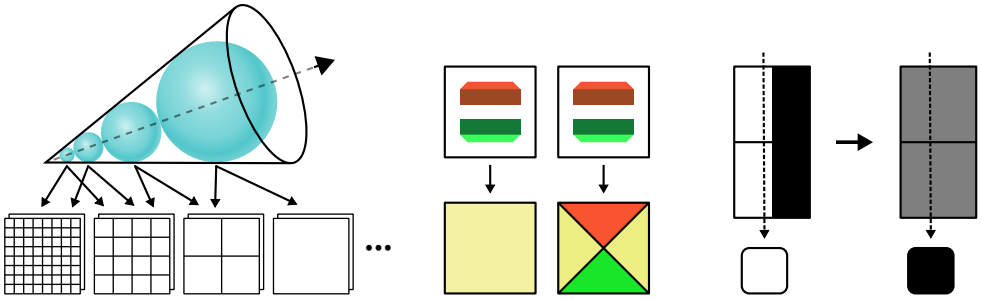


Figure 11. *Visualisation of several aspects of voxel cone tracing.* Figure **(a)** shows the successive accesses to different mip levels of the hierarchical voxel representation as the cone diameter increases. Figure **(c)** exemplifies the *red-green wall problem* that occurs when directional light information is lost due to non-directional averaging (left). Using anisotropic voxels (right) this problem can be mitigated. Figure **(c)** shows the thickening effect that occurs at grazing angles along surfaces. Since the average opacity of each block containing the actual scene geometry is 50% in the direction of the ray (left), the accumulated opacity through the averaged blocks along the same ray gives a divergent result (right).

Another problem is that while wide cones for diffuse light strongly benefit form the voxel tracing approach, the tracing of thin cones is effectively equivalent to an expensive ray marching operation through a 3D texture. In the sparse voxel octree-based approach, empty space is skipped when the tracing procedure encounters a childless node in the octree stucture in order to counteract this problem. Equivalently, we can perform a search of the largest empty parent voxel in the mipmap when we encounter an empty voxel. A problem with this skipping approach is, however, that samples that are situated in an empty voxel may still yield non-empty values when the interpolation of neighboring non-empty voxels are taken into account. This results in artifacts, as discussed in [Cra11], and is not resolved in the presented implementation (see Figure 18). Algorithm 1 shows at which point in the cone tracing procedure the space skipping is performed.

3.4 Compositing

Since the cone tracing is a very expensive operation, for the diffuse indirect illumination it is only carried out every fourth pixel by reducing the side lengths of the framebuffer by one half during the voxel cone tracing pass. As in [Pan14], we reduce the side lengths of the framebuffer by one half once more and use fourfold supersampling with a rotated grid pattern ($RGSS$) to reduce aliasing effects. In the compositing stage the individual subsamples are

accessed using the `NV_explicit_multisample` extension. The upsampling is achieved by bilateral low-pass filtering [Kop+07] using a 5x5 truncated Gaussian kernel and an additional weight based both on the depth value and surface normal from the geometry buffer. Let $I(\boldsymbol{p})$ be the result image at position $\boldsymbol{p}$, $I_\downarrow$ the buffer with the sparsely sampled diffuse indirect light, $F_\downarrow$ a window centered at $\boldsymbol{p}$ rounded to the coordinates of $I_\downarrow$, f the 5x5 Gaussian kernel, D the depth, N the normal buffer and g a Gaussian to smooth the dissimilarities in the depth and normal buffer, then the *joint bilateral filter* is given by:

$$I(\boldsymbol{p}) = \frac{1}{w_{\boldsymbol{p}}} \sum_{\boldsymbol{p}_\downarrow \in F_\downarrow} I_\downarrow(\boldsymbol{p}_\downarrow)\, f\big(\boldsymbol{p} - \boldsymbol{p}_\downarrow\big)\, g_{_N}\big(N(\boldsymbol{p}) - N(\boldsymbol{p}_\downarrow)\big)\, g_{_D}\big(D(\boldsymbol{p}) - D(\boldsymbol{p}_\downarrow)\big)$$

and the weight w_p is:

$$w_{\boldsymbol{p}} = \sum_{\boldsymbol{p}_\downarrow \in F_\downarrow} f\big(\boldsymbol{p} - \boldsymbol{p}_\downarrow\big)\, g_{_N}\big(N(\boldsymbol{p}) - N(\boldsymbol{p}_\downarrow)\big)\, g_{_D}\big(D(\boldsymbol{p}) - D(\boldsymbol{p}_\downarrow)\big)\,.$$

Due to the edge preserving effect of the joint bilateral filtering (see Figure 14) the sparse sampling of the diffuse indirect illumination only has a small effect on the resulting image since diffuse illumination is naturally a low-frequency effect. To some extent the filtering can even improve the image quality because it smoothes artifacts from the cone tracing. The same upsampling procedure is performed for the ambient occlusion map.

Finally, all previous results are combined in one image. The global illumination and direct illumination maps are added together and then multiplied with the diffuse map of the geometry buffer. The ambient occlusion map is multiplied with the resulting image. Each of these compositings has an additional blend factor which can be manipulated via the GUI at runtime. See Figure 15 for an overview of all interim results of the rendering procedure.

In a last step we take care of aliasing effects due to the rasterization. Since deferred shading prevents us from using hardware anti-aliasing techniques (supersampling), the *fast approximate anti-aliasing* algorithm (*FXAA*) is employed [Lot09]. *FXAA* computes an approximate gradient from the luminosity values of four neighboring pixels. Based on a simple edge detection, the pixels are then blurred with up to four samples along a direction that is perpendicular to the gradient. See Figure 13 for the result.

Figure 12. *Light leaking inside the ambulance.* Light leaking may occur when voxels are sampled that are averaged over both lit and unlit regions. This is in particular a result of the wide cones which are used for sampling the diffuse indirect light. The expected result is a dark interior as there are no light sources inside the ambulance.

Figure 13. *Anti-aliasing post-processing.* Depicted is the trunk of the truck in the car scene provided along with the source code. The left picture shows FXAA compared to the original image on the right.

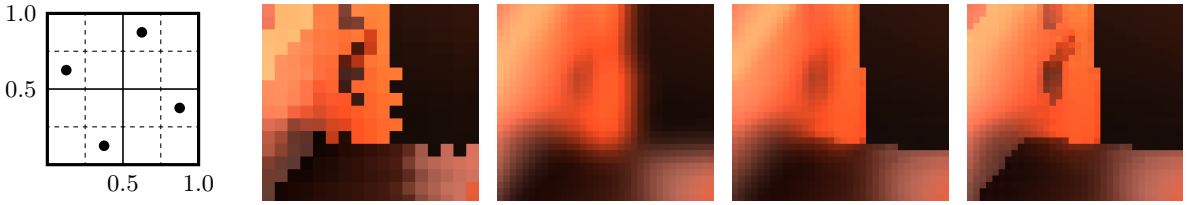


Figure 14. *Rotated sample grid and joint bilateral upsampling of the sparsely cone traced indirect lighting.* Figure **(a)** shows the pattern used to reduce aliasing effects due to the sparse cone tracing samples. The subdivision lines indicate the half and full sampling resolution. Figure **(b)** shows a small sparsely sampled part of the car scene (34×31 pixels). The black region lies in the background and the orange region consists of two orthogonally joint surfaces with several surface details. The ripples are a result of presenting the samples from the rotated grid in an axis-aligned grid. In Figure **(c)** a simple 5×5 Gaussian kernel is applied at the full resolution (68×62 pixels). Figure **(d)** shows the result of the 5×5 bilateral filtering taking only depth information into account and **(e)** includes both depth and normal information. The last three figures are slightly modified as the implementated algorithm caused subtle artifacts along the edges.

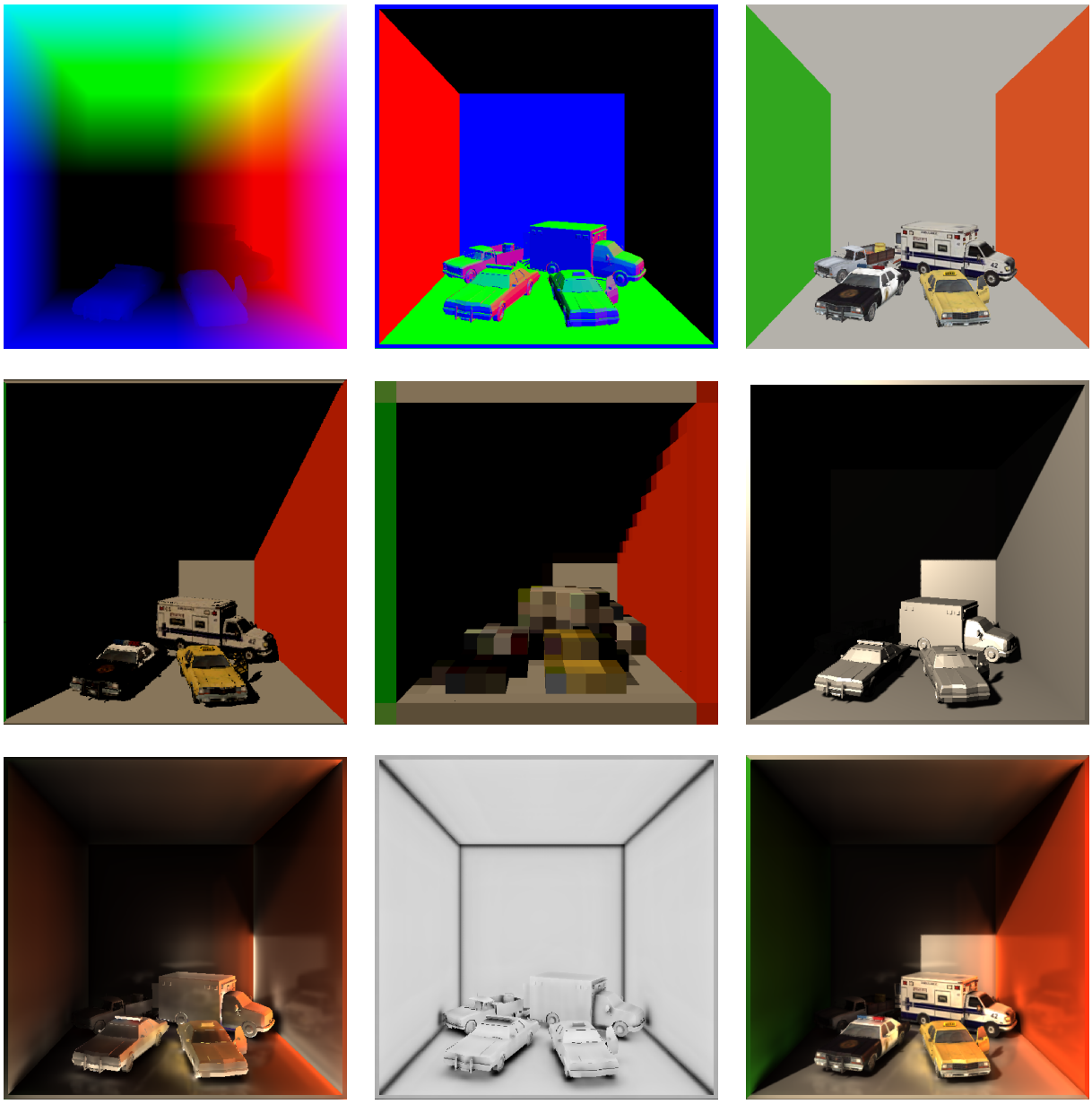


Figure 15. *Overview of all rendering stages.* The top row depicts the geometry buffer, including the position, normal and diffuse color maps. In the second row two mip levels of the voxel representation, as well as the direct light map are shown. In the bottom row the indirect map (both diffuse and specular indirect lighting), the ambient occlusion map and the final image are shown.

4 Results

In this section we will present measurements of execution time, as well as memory usage and discuss the image quality and address possible improvements of the implemented rendering technique. All of the measurements are performed on a *Pentium 4* computer with 2.7 Ghz clocking, *Linux Mint 17 Qiana* (32 bit) and 2 GB of primary memory. The GPU is a NVIDIA GeForce GTX 670 with 2 GB of GDDR5 video memory. The viewport size is set to 800×600 pixels. We use the car scene which is provided along with the source code in all measurements.

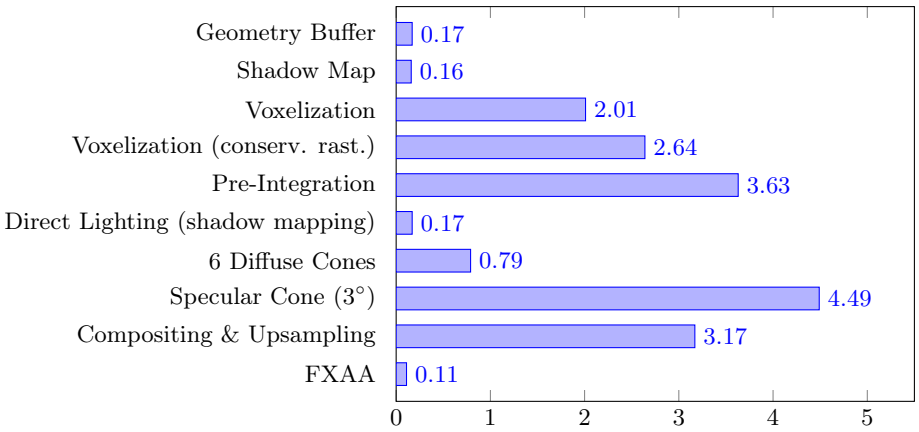


Table 1. *Performance evaluation of all rendering stages.* The timings are in milliseconds and measured with the car scene at 256^3 voxel resolution. Empty space skipping is enabled in the specular cone tracing algorithm.

Table 1 shows the timings of each rendering stage. Note that the voxelization and pre-integration take up a substantial amount of time. Taken together, they amount to roughly 40% of the total execution time. The render time of the stages in which only the scene geometry is rasterized (the geometry buffer and shadow mapping stages) and the post processing stage are negligible by comparison. Also note that specular cones take an order of magnitude more computing time than diffuse cones (see also Table 4). Besides that, we can observe a $\sim 25\%$ slowdown due to conservative rasterization. Using sparse sampling we could, however, reduce the execution time for the cone tracing stage by roughly two thirds.

As Table 2 shows, the 3D textures for the voxel maps are very memory consuming. The 256^3 voxel map requires almost a gigabyte of video memory, while [Cra+11] achieve a 512^3 resolution at the same memory demands but using a sparse octree-based voxel map. In the presented implementation a lot

Resolution	Mem. Usage (MB)
32^3	174
64^3	378
128^3	658
256^3	939

Table 2. *Statistics of the video memory usage.*

of memory could be saved by removing the lowest mip level from five of the six textures since they are identical. The reason for this is that the radiosity values are stored isotropically during the voxelization. This is necessary since the viewing direction dependent modulation of the directional voxel values (as explained in Section 3.3) would result in noticeable fluctuations in brightness, especially on flat, axis-aligned surfaces as shown in Figure 16. The reduction to one image for the lowest mip level would also accelerate the voxelization significantly, since currently the atomic averaging is used for all six texture storages. During the pre-integration a lot of texture accesses could be saved as well.

Another aspect that could be improved on is that the bilateral filtering for the upsampling of the sparse cone tracing samples is a naive implementation which takes around 3 ms using a 5x5 Gaussian kernel (see Table 1). A 3x3 kernel cuts down the execution time considerably but it produces a considerably worse image quality. For this reason, the sparse cone tracing is at the 600×800 resolution only worth doing when a large portion of the visible surfaces is traced with very thin specular cones. Moreover, the blurred sparse samples remain at the resolution of the sparse samples. To achieve smooth transitions at the viewport resolution, the sparse samples need to be interpolated. However, hardware based interpolation cannot be leveraged for this task because `NV_explicit_multisample` used for RGSS restricts the texture accesses to nearest-neighbor filtering. The upsampling algorithm could possibly generally be optimized by fetching 4 texels at a time with `textureGather`.

	Diffuse cones	Specular (aperture=0°)
No skipping	0.82 ms	25.03 ms
Skipping	0.88 ms	9.62 ms
Speedup	0.93	2.6

Table 3. *Statistics of empty space skipping.*

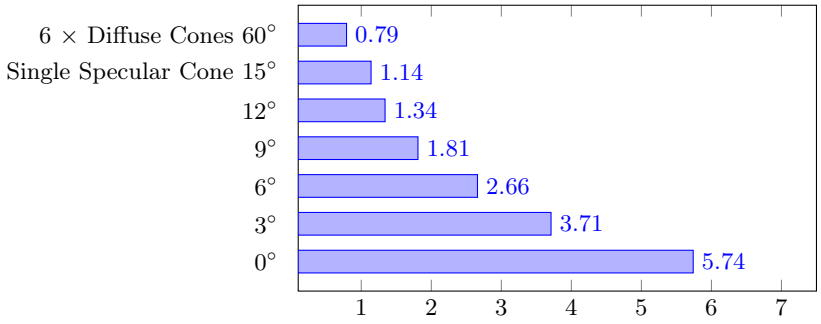


Table 4. *Performance evaluation of the specular light tracing with various apertures.* Each of the diffuse cones has an aperture of about 60°. Empty space skipping is enabled.

The implemented empty space skipping procedure yields a twofold speedup for highly specular cones. However, it slows the diffuse cone tracing slightly down, as table Table 3 shows. Therefore, it is only enabled for specular cone tracing. The skipping could possibly be enhanced by pre-calculating the mip level of the largest empty parent voxel of each empty voxel. This way, texture accesses could be reduced during the cone tracing. The mip level information could be stored, for example, in the least significant bits of the voxel values which are also used for the averaging calculations during the voxelization (see section 3.1). For full color depth an additional 3D texture storage could be allocated for these tasks (e.g. using the `R8UI` format). However, this would introduce more texture accesses.

For searching the largest empty parent voxel it appears to be faster to use a top-down approach compared to a bottom-up search. This, however, remains to be examined more thoroughly.

As explained in section 3.1, conservative rasterization prevents gaps between adjacent triangles, yet a 6-separating non-anti-aliased voxelization does not prevent light leaking through thin geometries. This effect occurs in particular when a thin cone takes samples at the finest voxel resolution and simply steps through the surface where adjacent voxels only share a vertex or an edge (6-separating). It is especially noticeable during shadow tracing with very thin cones as shown in Figure 17. To prevent light leaking in these scenarios, either a solid voxelization, a 26-separating surface voxelization or a thickening approach of the surfaces as in [Pan14] is needed. An anti-aliased thin voxelization and a higher sampling rate might also help to reduce this problem.

Lastly, the implemented rendering algorithms are not yet free of artifacts (see Figure 18), and many small inefficiencies can be found throughout the

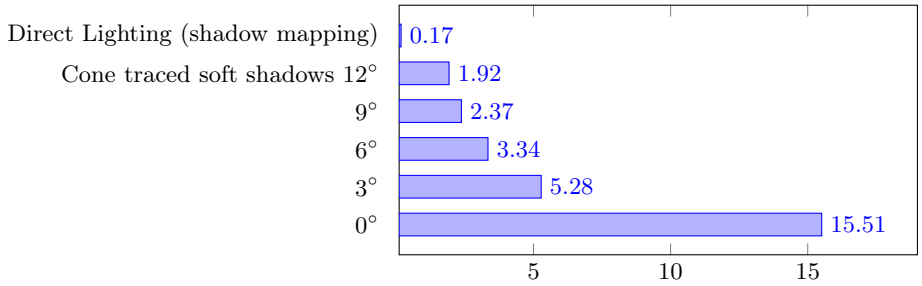


Table 5. *Performance evaluation of the shadow cone tracing.* The timings are in milliseconds and measured with the car scene with a 256^3 voxel resolution. The cone apertures are indicated by degree values. Empty space skipping and sparse sampling are not implemented in the shadow cone tracing algorithm.

implementation. For example, some vertex shader programs still include the calculation of the view-projection matrices and transposed inverse matrices, which could instead be pre-computed on the CPU. That way expensive matrix operations would not need to be carried out for each processed vertex resp. triangle. Besides that, we could adobt optimizations from [RB13] for the implemented voxilization technique. In particular, we could switch to the proposed fragment-parallel voxelization for large triangles for a better thread utilization on the GPU. Moreover, we could adopt the proposed coordinate swlizzling technique instead of using tranformation matrices for the triangle rotations during the voxelization.

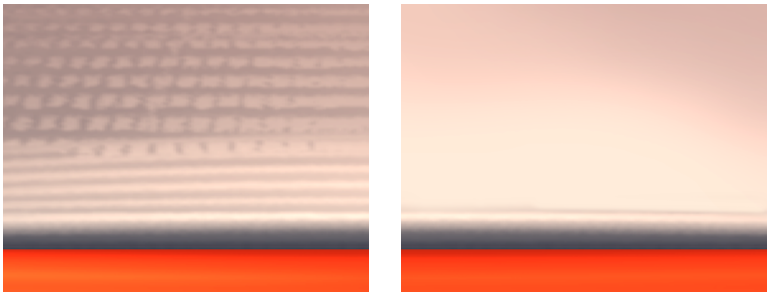
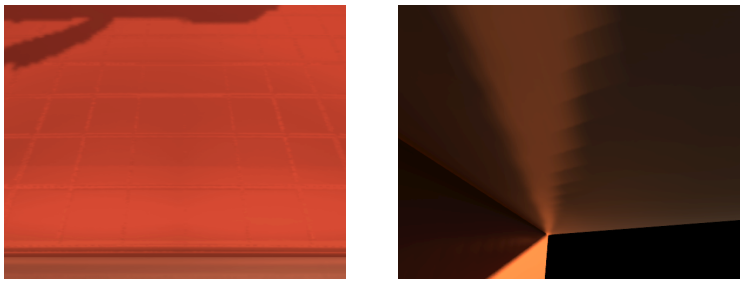


Figure 16. *Artifacts from using anisotropic voxels at the lowest mip level.* The left picture shows rippled surfaces as the result. The right picture illustrates the result with anisotropic voxels at the lowest mip level as it is implemented. The dark band is a result of the ambient occlusion effect.

Figure 17. *Cone traced direct shadows with different cone apertures.* In reading direction, the shadows in the first three pictures are traced with cone apertures of 9°, 6° and 3°. Note that the individual voxels become noticeable at 6° near the siren of the police car. On the third picture one can notice light leaking near the taxi door, due to thin voxelization with non-conservative triangle rasterization and a large step size. On the bottom left picture the conservative rasterization is enabled, yet some leaking remains due to the thin voxelization. On the fifth picture the aperture is set to 0° and the last picture shows shadow mapping with percentage closer filtering for comparison. See Table 5 for time measurements.



(a) (b)

Figure 18. *Cone tracing artifacts.* Figure **(a)** shows a grid pattern that occurs in reflections due to empty space skipping. The banding in Figure **(b)** results from clipped cones that extend beyond the boundaries of the voxel map. Solutions to this problem might include to add another sample exactly where the cone is clipped or to shift the last sample to this point.

5 Conclusion and Future Work

In this thesis we have reviewed and implemented the voxel cone tracing technique for real-time rendering of global illumination effects which was developed by C. Crassin et al. [Cra+11]. We have started with a short introduction into the mathematical and physical background of image synthesis in Chapter 2. After that, we have thoroughly described the implementation of the technique in Chapter 3. In Section 3.1 we have explained the implemented conservative rasterization based voxelization technique which is similar to [CG12] but does not rely on a sparse data structure. Furthermore, we have implemented voxel cone tracing of both diffuse and specular reflections of indirect light as described in Section 3.3. In the same section we have also touched upon how voxel cone tracing can be used for ambient occlusion and soft shadow effects, how empty space skipping is implemented and how self-intersections are handled. Section 3.4 we have described how we upsample the sparse cone tracing samples using joint bilateral filtering and how all intermediate results are composed. Finally, in Section 4, we have presented and discussed the results.

There are serveral aspects of the implemented renderer that can provide opportunities for improvement and/or future research. As we have discussed in Section 4, a main critical issue of the technique and in particular of the implementation provided in this thesis, is the high memory demand of the voxel maps. To mitigate this problem, we could employ a compression scheme such as *sparse voxel octrees* [Cra+11] or *directed acyclic graphs* (DAGs) [KSA13]. Alternatively or additionally, we could rely on a cascaded approach like the *clipmap* presented in [Pan14] in which the voxel map is centered around the camera and regions remote from the camera are voxelized at a lower resolution to save memory. For hardware native texture compression we could possibly make use of the `ARB_sparse_texture` and `EXT_sparse_texture2` extensions which are exposed in NVIDIA's Maxwell architecture[2].

Two additional performance issues result from the voxelization and pre-integration stages which take up a substantial amount of the render time. In [CG12] the impact of these stages is alleviated by voxelizing the full scene only once and updating subregions of the voxel structure as needed. In the clipmap approach they minimize the voxel map updates during camera movements by conserving the overlapping regions of the voxel map between two frames. The observed slowdown due to conservative rasterization could be reduced by using hardware native conservative rasterization which is standardized in `NV_conservative_raster` and also exposed in the Maxwell architecture.

[2] `https://developer.nvidia.com/content/maxwell-gm204-opengl-extensions` (February 10, 2015)

A fourth major issue is light leaking as a result of sampling large voxels with coarsely approximated directional light information. A possible solution might be to use orthogonal basis functions (such as spherical harmonics) at the coarse mip levels as proposed in [Rau13]. This could enable a more detailed description of the underlying scene geometry compared to the six-directional approach, while still allowing a fast evaluation of the regions of space the voxels represent.

In conclusion, we have reviewed a state-of-the-art rendering technique that constitues an efficient alternative to other global illumination approaches. It will be interesting to see whether the high memory demands and the light leaking can be further improved in future research.

References

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Naty Hoffman. *Real-Time Rendering, Third Edition*. Taylor & Francis, 2008. ISBN: 9781439865293. URL: `http://www.realtimerendering.com/`.

[Bli77] James F Blinn. "Models of light reflection for computer synthesized pictures". In: *ACM SIGGRAPH Computer Graphics*. Vol. 11. 2. ACM. 1977, pp. 192–198.

[CG12] Cyril Crassin and Simon Green. "Octree-based sparse voxelization using the GPU hardware rasterizer". In: *OpenGL Insights* (2012), pp. 303–318.

[Cra11] Cyril Crassin. "GigaVoxels: a voxel-based rendering pipeline for efficient exploration of large and detailed scenes". PhD thesis. PhD thesis, Université de Grenoble, 2011.

[Cra+11] Cyril Crassin et al. "Interactive indirect illumination using voxel cone tracing". In: *Computer Graphics Forum*. Vol. 30. 7. Wiley Online Library. 2011, pp. 1921–1930.

[DS05] Carsten Dachsbacher and Marc Stamminger. "Reflective shadow maps". In: *Proceedings of the 2005 symposium on Interactive 3D graphics and games*. ACM. 2005, pp. 203–231.

[Gor+84] Cindy M Goral et al. "Modeling the interaction of light between diffuse surfaces". In: *ACM SIGGRAPH Computer Graphics*. Vol. 18. 3. ACM. 1984, pp. 213–222.

[HN12] Eric Heitz and Fabrice Neyret. "Representing appearance and pre-filtering subpixel data in sparse voxel octrees". In: *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*. Eurographics Association. 2012, pp. 125–134.

[Jen96] Henrik W Jensen. "Global illumination using photon maps". In: *Rendering Techniques' 96*. Springer, 1996, pp. 21–30.

[Kaj86] James T Kajiya. "The rendering equation". In: *ACM Siggraph Computer Graphics*. Vol. 20. 4. ACM. 1986, pp. 143–150.

[KD10] Anton Kaplanyan and Carsten Dachsbacher. "Cascaded light propagation volumes for real-time indirect illumination". In: *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games*. ACM. 2010, pp. 99–107.

[Kel97] Alexander Keller. "Instant radiosity". In: *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*. ACM Press/Addison-Wesley Publishing Co. 1997, pp. 49–56.

[Kop+07] Johannes Kopf et al. "Joint Bilateral Upsampling". In: *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2007)*. Vol. 26. 3. Association for Computing Machinery, Inc., 2007. URL: http://research.microsoft.com/apps/pubs/default.aspx?id=78272.

[KSA13] Viktor Kämpe, Erik Sintorn, and Ulf Assarsson. "High Resolution Sparse Voxel DAGs". In: *ACM Transactions on Graphics* 32.4 (July 7, 2013). URL: highResolutionSparseVoxelDAGs.pdf.

[Lot09] T Lottes. *FXAA (Whitepaper)*. Tech. rep. NVIDIA, 2009. URL: http://developer.download.nvidia.com/assets/gamedev/files/sdk/11/FXAA_WhitePaper.pdf.

[LW93] Eric P Lafortune and Yves D Willems. "Bi-directional path tracing". In: *Proceedings of CompuGraphics*. Vol. 93. 1993, pp. 145–153.

[Max95] Nelson Max. "Optical models for direct volume rendering". In: *Visualization and Computer Graphics, IEEE Transactions on* 1.2 (1995), pp. 99–108.

[McG13] Morgan McGuire. *The Graphics Codex*. v. 2.8. Apple Inc., 2013.

[Mil94] Gavin Miller. "Efficient algorithms for local and global accessibility shading". In: *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. ACM. 1994, pp. 319–326.

[ML09] Morgan McGuire and David Luebke. "Hardware-accelerated global illumination by image space photon mapping". In: *Proceedings of the Conference on High Performance Graphics 2009*. ACM. 2009, pp. 77–89.

[Pan14] Alexey Panteleev. "Practical Real-Time Voxel-Based Global Illumination for Current GPUs". Talk at the GPU Technology Conference 2014. 2014. URL: http://on-demand-gtc.gputechconf.com/gtcnew/on-demand-gtc.php?searchByKeyword=SG4114.

[PD84] Thomas Porter and Tom Duff. "Compositing digital images". In: *ACM Siggraph Computer Graphics*. Vol. 18. 3. ACM. 1984, pp. 253–259.

[Rau13] Randall Rauwendaal. "Voxel Based Indirect Illumination using Spherical Harmonics". PhD thesis. Oregon State University, Aug. 2013. URL: http://hdl.handle.net/1957/42266.

[RB13] Randall Rauwendaal and Mike Bailey. "Hybrid Computational Voxelization Using the Graphics Pipeline". In: *Journal of Computer Graphics Techniques (JCGT)* 2.1 (2013), pp. 15–37. ISSN: 2331-7418. URL: http://jcgt.org/published/0002/01/02/.

[RGS09] Tobias Ritschel, Thorsten Grosch, and Hans-Peter Seidel. "Approximating dynamic global illumination in image space". In: *Proceedings of the 2009 symposium on Interactive 3D graphics and games*. ACM. 2009, pp. 75–82.

[Rit+12] Tobias Ritschel et al. "The state of the art in interactive global illumination". In: *Computer Graphics Forum*. Vol. 31. 1. Wiley Online Library. 2012, pp. 160–188.

[SS10] Michael Schwarz and Hans-Peter Seidel. "Fast Parallel Surface and Solid Voxelization on GPUs". In: *ACM Transactions on Graphics* 29.6 (Proceedings of SIGGRAPH Asia 2010) (Dec. 2010), 179:1–179:9.

[Sto+04] William A Stokes et al. "Perceptual illumination components: a new approach to efficient, high quality global illumination rendering". In: *ACM Transactions on Graphics (TOG)*. Vol. 23. 3. ACM. 2004, pp. 742–749.

[TL04] Eric Tabellion and Arnauld Lamorlette. "An approximate global illumination system for computer generated films". In: *ACM Transactions on Graphics (TOG)*. Vol. 23. 3. ACM. 2004, pp. 469–476.

[Wil83] Lance Williams. "Pyramidal parametrics". In: *ACM Siggraph Computer Graphics*. Vol. 17. 3. ACM. 1983, pp. 1–11.

Internet resources were last accessed on February 10, 2015.