TECHNISCHE UNIVERSITÄT DRESDEN

FACULTY OF COMPUTER SCIENCE
INSTITUTE OF SOFTWARE AND MULTIMEDIA TECHNOLOGY
CHAIR OF COMPUTER GRAPHICS AND VISUALIZATION
PROF. DR. STEFAN GUMHOLD

Minor Thesis

Point Cloud Based Interactive Global Illumination

Mirko Salm
(Mat.-No.: 3753374)

Tutor: Nico Schertler, M.Sc.
Dipl.-Medieninf. Joachim Staib

Dresden, 09.03.2015

Aufgabenstellung

Globale Beleuchtungseffekte wie indirekte Beleuchtung und Verschattung sind ein signifikanter Faktor für die Realitätsnähe von gerenderten Bildern. Außerdem tragen sie zum besseren Verständnis der Szene bei, indem bspw. die Relationen von Objekten zueinander durch Schatten deutlich werden. Obwohl eine physikalisch korrekte Berechnung der indirekten Beleuchtung möglich ist, sind solche Verfahren sehr rechenintensiv und damit nicht für interaktive Darstellungen geeignet. Ansätze, die Teile der Lichtausbreitung vorberechnen, können zur Beschleunigung beitragen, jedoch sind diese oft auf statische oder teil-statische Szenen beschränkt. Stattdessen sollen in dieser Arbeit Verfahren untersucht werden, die optisch ansprechende Ergebnisse erzielen. Dabei soll der Fokus auf Effekte der indirekten Beleuchtung gelegt werden. Nachdem ein fundierter Überblick über solche Verfahren gegeben wurde, soll ein Programm für die Visualisierung von Punktwolken mittels Voxel Cone Tracing implementiert werden. Dabei ist darauf zu achten, dass die Implementierung mit dynamischen Szenen umgehen kann und eine genügend kurze Berechnungszeit aufweist, sodass eine interaktive Visualisierung möglich ist. Die Implementierung soll in dem Umfang, den die gewählten Algorithmen erlauben, auf der Grafikkarte laufen. Abschließend ist diese bezüglich Performanz und Qualität zu evaluieren.

Teilaufgaben:

- Literaturrecherche zu globalen Beleuchtungsmodellen und deren effiziente Implementierung (Light Propagation Volumes, Voxel Cone Tracing, Virtual Point Lights, Instant Radiosity).
- Überblick über wesentliche Verfahren zur indirekten Beleuchtung.
- Erstellen oder Sammeln von geeigneten Test-Szenen unterschiedlicher Komplexität.
- Implementierung der Beleuchtungsberechnung für direkte und indirekte Beleuchtung mittels Voxel Cone Tracing über einem adaptiven Octree auf der Grafikkarte.
- Effiziente Implementierung, die eine interaktive Visualisierung ermöglicht.
- Evaluation der Performanz der Implementierung in Bezug auf Speicherbedarf und erreichter Framerate abhängig von der Szenenkomplexität.
- Evaluation der Qualität durch:
  - Plausibilität der optischen Erscheinung
  - Unterstützung der folgenden Effekte (Klassifizierung in 'ist vollständig/teilweise implementiert', 'kann noch implementiert werden', 'wird vom gewählten Verfahren nicht unterstützt'): Ambient Occlusion, Farbbluten, gerichtet diffuse (spekulare) Reflexionen.
- Analyse möglicher Probleme der Implementierung (bspw. Ausleuchtung von schmalen Gängen)

und Skizzierung von Lösungsmöglichkeiten.

Optionale Aufgaben:

- Implementierung einer Flächenlichtquelle.
- Implementierung von Light Propagation Volumes

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag dem Prüfungsausschuss der Fakultät Informatik eingereichte Arbeit zum Thema:

Point Cloud Based Interactive Global Illumination

vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den 09.03.2015

Mirko Salm

Contents

Nomenclature

BRDF	bidirectional reflectance distribution function
ESMs	exponential shadow maps
GI	global illumination
GPGPU	general purpose computing on the GPU
GPU	graphics processing unit
ISMs	imperfect shadow maps
LPVs	(cascaded) light propagation volumes
NDF	normal distribution function
RoI	range of influence (of a scene sample)
SVO	sparse voxel octree
VCT	voxel cone tracing
VPL	virtual point light
VSMs	variance shadow maps

1 Introduction

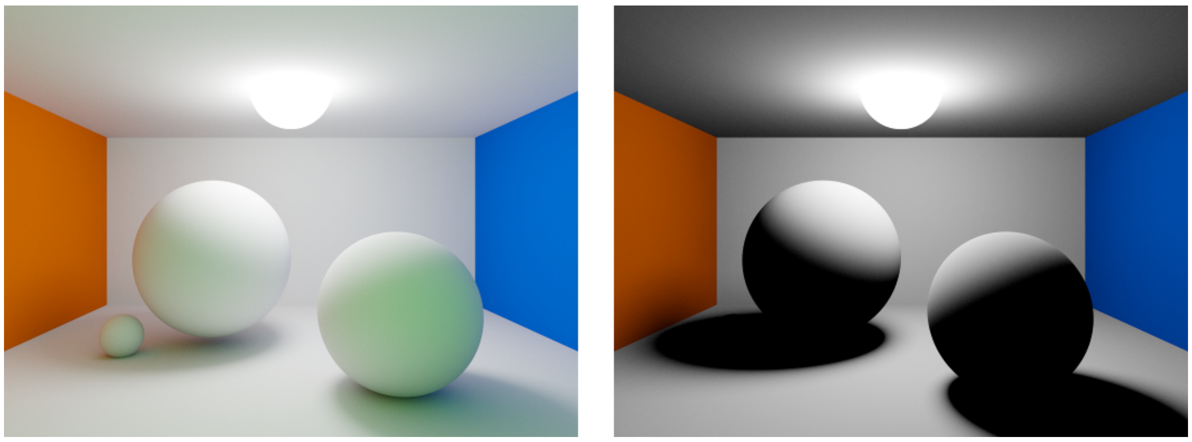


Figure 1.1: *A high quality offline rendering of a simple, virtual scene demonstrating the importance of a comprehensive light propagation simulation. Left image: complex light paths resulting from multiple reflections of light at surfaces are considered. Right image: only simple light paths that do not account for more than a single light reflection are simulated. As a result, the right image lacks important visual cues. Due to the missing light inside the shadows, it is not clear whether the spheres touch the floor or not while the small sphere is not visible at all. Furthermore, in the left image the reflected light from the vertical walls partially colors the spheres and shadows in their respective colors improving the perception of the relative spatial positioning of the objects in the scene as well as hinting the presence of a green wall behind the virtual camera.*

The computation of photo-realistic images from virtual scene descriptions belongs to the primary objectives of 3d computer graphics. An accurate simulation of light propagation is essential to produce synthetic images of natural appearance. Unfortunately, the high complexity of the underlying mathematical models makes comprehensive simulations computational expensive which is particularly problematic for interactive or even real-time applications where individual images have to be generated in a matter of milliseconds. A large amount of this complexity is induced by the complicated paths that the light can take through the scene before being detected by the virtual sensor. Therefore, real-time applications traditionally consider only simple light paths whose computation maps well to graphics hardware. As illustrated in figure 1.1, this is insufficient when attempting to achieve realistic results since many important visual cues are missing. Consequently, additional heuristics are often used to improve the appearance of interactively rendered scenes. With increasingly more potent graphics hardware, these heuristics are gradually substituted by less restrictive and more accurate techniques. In this thesis, a light propagation approach derived from the work of Crassin et al. ([CNS^{+}11]) is presented that attempts to capture many important visual effects originating from multiple light reflections while maintaining interactive frame rates. The original algorithm is modified to support a large amount of progressively built light bounces as well as to reduce memory consumption and to simplify the overall approach. In addition, an antialiasing scheme for the construction of the structure which the light propagation simulation is based on is proposed. This allows, to some degree, to build the structure from a scene sampling of adaptive density without introducing regions of degenerated quality in the resulting scene description.

1.1 Thesis Structure

First, an overview of the effects due to light propagation inside a scene as well as the underlying physical concepts and mathematical models employed for rendering purposes is given. Subsequently, previous work in the field of interactive light propagation simulation is reviewed before the algorithm developed in the context of this work is described. The general description of the approach is followed by a presentation of the implementation. At last, the achieved results are evaluated and possible improvements are outlined before a conclusion of the work is given.

2 Global illumination

This chapter provides a fundamental understanding of how light affects the appearance of a scene and how its effects can be mathematically modeled and simulated for computer graphics purposes which in turn is a central objective of the algorithm presented in this work.

In the context of computer graphics, *global illumination* (GI) [DBB06] describes the behavior of light in a virtual scene where all light events local to the objects surfaces are modeled as well as the thereby induced effects on a global scale. For example, the global implication of light being locally reflected at a red surface might be that an object occluded by the surface receives no light while a nearby object appears red due to the reflected light (see figure 2.5). In contrast, *local illumination* models only a single local light event at every surface point under consideration of the light sources and the observer while completely ignoring the geometry in the scene.

When light is emitted by light sources into a scene, it can undergo, among others, reflection, refraction, absorption, and scattering events before a fraction eventually reaches the sensor of the virtual camera, effectively forming the final image. Most prominent of these local events are diffuse and glossy light reflections. *Glossy reflections* (often also called *specular* reflections) account for the portion of light that is directly reflected and, due to imperfections, scattered at the surface of the material (see figure 2.2). In contrast, the portion of light, that actually enters the body of the material, where it then gets scattered and partially absorbed before being re-emitted at approximately the point of incidence, is modeled by the *diffuse reflection* [AMHH08]. In the following, only the *ideal* diffuse reflection which ignores imperfections at the surface is considered (see figure 2.1). The color appearance that a material exhibits under lighting is on the one hand determined by the color of the light and on the other hand by the reflection behavior of the material itself. For rendering purposes, materials are commonly classified as either metals or insulators (dielectrics) or combinations of both [Bur12][Kar13][LdR14]. Insulators usually exhibit both diffuse and glossy reflections to a certain degree. The coloring of highlights resulting from glossy reflections at the surfaces of ideal insulators are entirely determined by the color of the light since the reflections itself do not exhibit any wavelength dependencies. However, the light portion that does not undergo glossy reflections and is instead diffusely reflected changes its color according to the *albedo* which corresponds to the wavelength dependent amount of light that is re-emitted instead of being absorbed by the material [JMLH01]. In contrast, pure metals absorb all light that is entering their body and therefore do not produce diffuse reflections. Their coloring is instead entirely determined by the glossy reflections at their surfaces which are, in contrast to the glossy reflections at the surfaces of insulators, wavelength dependent.

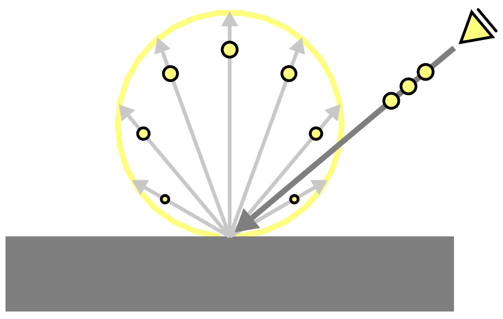


Figure 2.1: *Ideal diffuse reflection. The amount of light reflected in each direction is independent of the direction of the incoming light.*

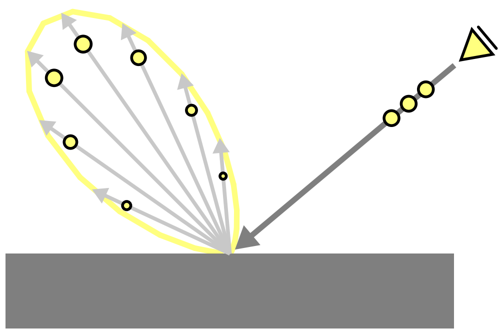


Figure 2.2: *Glossy reflection. The reflected light approximately focuses around the reflection direction of the light direction.*

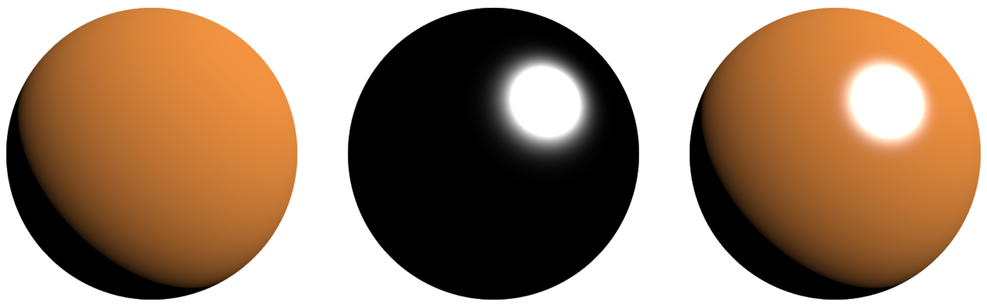


Figure 2.3: *A non-metallic sphere of brown albedo illuminated from the top-right by a white light source. From left to right: diffuse reflections, glossy reflections resulting in a highlight, and the combined effect of both. Note that these images are only intended to outline the different effects and are not physically correct rendered.*

The concept of global illumination, building upon the local effects outlined so far, is commonly divided into *direct lighting*, which considers only the first interaction of light with the scene, and *indirect lighting*, handling all successively occurring events. This distinction is on the one side made because the computation of indirect lighting is in general substantially more involved than the one of direct lighting, and on the other side, because both concepts exhibit their own distinctive effects. Shadows, for example, are a characteristic feature of direct lighting. They are the result of light being blocked by occluders before reaching the subsequently shadowed geometry. In the case of mostly diffuse indirect lighting, occluded regions tend to be rather generally darkened instead of distinctively shadowed (see figure 2.4). Scenes

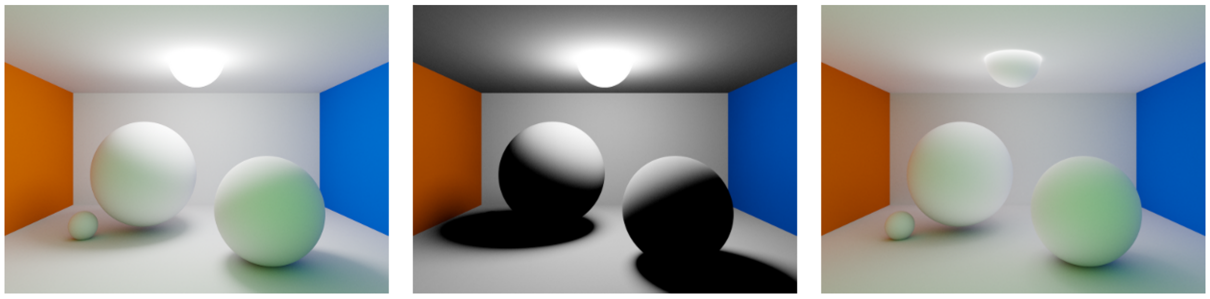


Figure 2.4: *Comparison of the characteristics of direct and indirect lighting. From left to right: direct and indirect lighting, only direct lighting, only indirect lighting. Note how the indirect lighting lacks distinct shadows and instead produces, among other effects, a very soft darkening under the spheres. This effect is often approximated by ambient occlusion [Fer04] in cases where full global illumination is too expensive.*

exclusively rendered with direct lighting often exhibit unnaturally high contrast due to the coherent nature of the primary light paths. Furthermore, the brightness of the resulting images presents only a lower bound to the real one since light that is not reflected towards the camera by the first bounce is completely ignored. Indirect lighting accounts for that by considering potential subsequent light bounces. During diffuse reflection events, parts of the light are absorbed, effectively tinting the re-emitted light in the color of the material. This gives rise to the so called *color bleeding*, where brightly lit surfaces appear to illuminate their surroundings with light of their own coloring (see figure 2.5).

Another distinction is made with respect to light sources. In reality, light sources like light bulbs or the sun emit their photons from surfaces of finite extents. In computer graphics they are therefore referred to as *area lights*. However, in practice area lights are often approximated by *point lights* whose accurate simulation is substantially less expensive. Point lights, in contrast to area lights, emit all photons from a single point in space. This property allows their direct lighting contribution to be computed analytically which is in general not possible for area lights. The most commonly used point light variants include omni-directional point lights, which emit light in all directions equally, spotlights, which are additionally parametrized by a primary emittance direction and an angular falloff, and directional light sources. Directional lights provide a further approximation by principally assuming infinite distance to the point light source. Their spatial positioning can therefore be described by a simple direction, hence the name. Even though they are convenient to work with in terms of computation, point lights produce unnaturally hard shadows due to their binary visibility from any point in the scene (see figure 2.6).

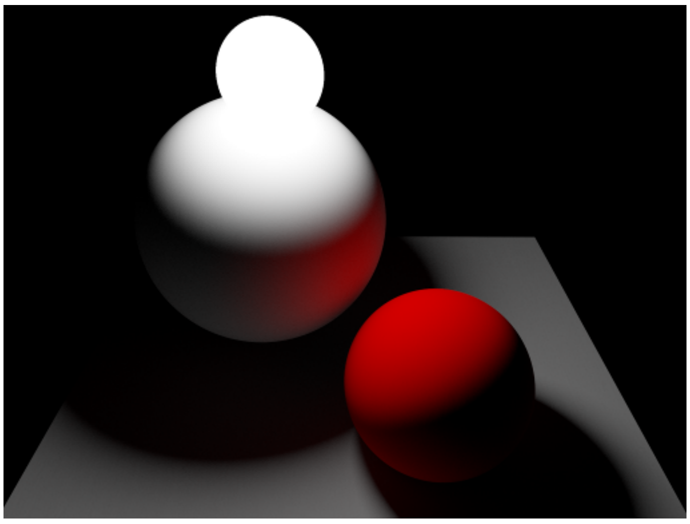


Figure 2.5: *Color bleeding. The light reflected from the right sphere partially colors the large sphere red.*

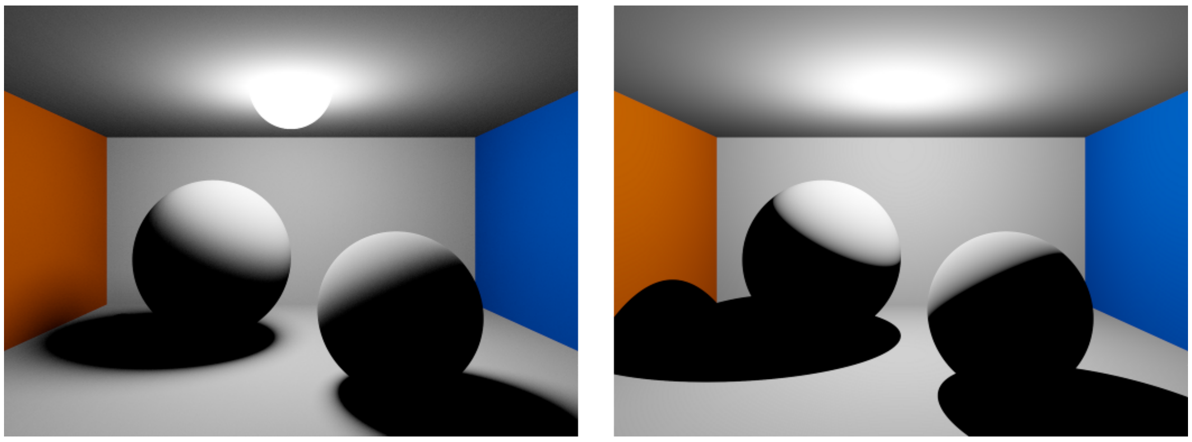


Figure 2.6: *Comparison between the direct lighting from a spherical area light (left) and from an omnidirectional point light (right). The shadows cast by the spheres in the left image feature penumbras, smooth transitions from shadowed to the illuminated regions. These gradients are not apparent in the right image due to the infinitesimal size of the point light.*

2.1 Basic Radiometry

Radiometry provides mathematical tools to model light transport under the assumption that light travels along straight paths or rays [PH10]. By largely ignoring the wave properties of the light, several less dominant effects like polarization, interference, diffraction, fluorescence, and phosphorescence can not be modeled. Since our perception of color is wavelength depended [TFCRS11], the wave character can-not be disregarded completely. All radiometric computations are therefore performed independently for a number of distinct wavelengths or for a number of representative wavelength ranges. At each pixel of the virtual sensor, the resulting values are eventually converted into a sRGB triple which in turn is interpreted by the monitor to determine the intensities of the color components of the respective pixel of the display. Moreover, the wavelength property is also relevant for the definition of *energy* since the amount of energy light holds is not only proportional to the number of photons it consists of but also to the frequency of these photons [AMHH08]. While energy constitutes the conceptional basic quantity in radiometry from which all other quantities are directly or indirectly derived, it is usually only used to determine the final color of the pixels in a rendered image due to its lack of descriptive power with respect to time. More practical quantities are therefore provided by *flux*, which measures the total amount of energy emitted by a light source per time interval and by *irradiance* which in turn is defined as the flux density with respect to area. To illustrate the relation between flux and irradiance, consider an omni-directional light source surrounded by a sphere of varying scale. While the total amount of flux measured at the surface of this sphere remains equal irrespective of the chosen radius, the received irradiance falls off inversely proportional to the squared radius since the flux is distributed over an increasingly larger area (see figure 2.7). This also explains why the brightness of a diffuse surface falls off with increasing distance to the light source. A similar situation occurs when a diffuse surface is illuminated under increasingly acute angle, which likewise distributes the energy of the incoming light over a larger surface area (see figure 2.8). This effect is sometimes confused with *Lambert's cosine law* [Lam92], which states that under ideal diffuse reflection at a surface, the amount of light re-emitted in a certain direction falls off proportional to the cosine of the angle between this direction and the normal of the surface (see figure 2.1). The directional dependent quantity referred to by Lambert's law is called *intensity* and is among others useful to describe the emission behavior of point light sources. More precisely, intensity is defined as flux density with respect to *solid angle*. Solid angle is a two dimensional analog to the concept of one dimensional angles. As a two dimensional object projects to an arc on the unit circle whose length corresponds to an angle in radians (see figure 2.9), a three dimensional solid projects to a surface patch on the unit sphere, with the area corresponding to a solid angle measured in *steradians* (see figure 2.10). In conclusion, the maximal solid angle of 4π steradians can only be subtended by an object completely enclosing the point of reference. Solid angle is also used to account for the directional dependency in the definition of *radiance*, which provides a natural quantity to simulate light transport by rays since it measures the amount of light traveling along a straight path parametrized by position and direction. Radiance is therefore defined as flux density with respect to solid angle and projected area or alternatively as intensity per projected area. The projection property of the area ensures that the differentiation is performed in a plane perpendicular to the direction in which the intensity is measured. This differentiation of intensity

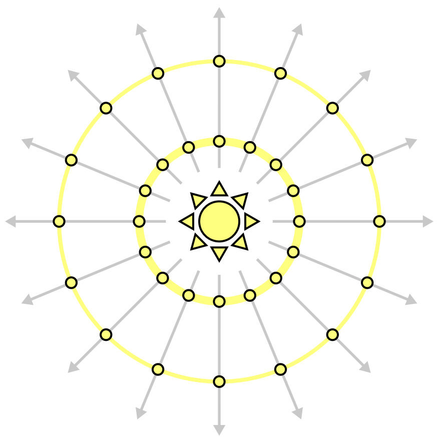


Figure 2.7: *An omni-directional light source. The flux density with respect to area (irradiance) decreases inversely proportional to the squared distance to the light source while the total flux remains constant.*

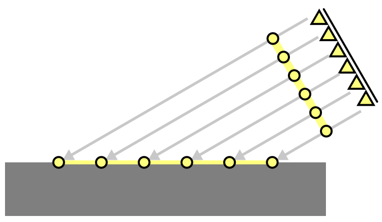


Figure 2.8: *The acute incident angle of the light distributes the photons over a larger surface area, leading to a decrease in irradiance proportional to the cosine of the incident angle.*

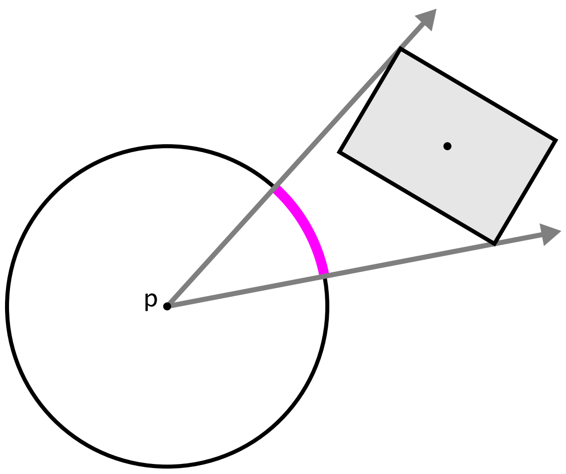


Figure 2.9: *A two dimensional solid seen from point p subtends an angle (magenta), corresponding to a distance on the unit circle.*

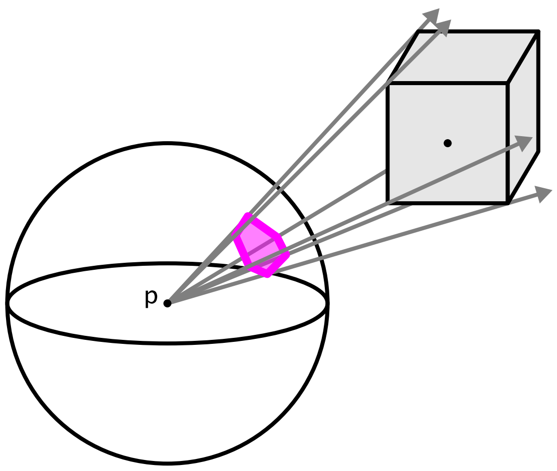


Figure 2.10: *A three dimensional solid seen from point p subtends a solid angle (magenta), corresponding to an area on the unit sphere.*

with respect to projected area can be intuitively understood as picking a single light ray from a set of rays that perpendicularly pass through the backside of a plane of reference whose normal corresponds to the direction in which the intensity is measured. Radiance can be particularly convenient to work with since all other radiometric quantities can be derived by integration from it.

2.2 The Rendering Equation

$$L_o(x, \omega_o) = L_e(x, \omega_o) + \int_\Omega f_r(x, \omega_i, \omega_o) L_i(x, \omega_i)(\omega_i \cdot n) d\omega_i \tag{2.1}$$

Based on the radiometric quantities outlined previously, the light propagation in a scene can be modeled by the *rendering equation* [ICG86][Kaj86] (equation 2.1) which constitutes the mathematical basis for global illumination under the assumption that light is only affected on interaction with surfaces but not when traveling through the space separating them. The rendering equation relates the definite radiance L_o along a given direction ω_o at some surface point x to the light L_e that the material of the surface produces on its own and to the light L_i that the whole scene directly or indirectly reflects or emits towards this point of reference and the fashion in which the material subsequently reflects the light. This relation is expressed by an integral over the hemisphere defined by the normal n at point x. The scalar product of ω_i and n accounts for the projection of the incoming light into the plane defined by the point x and normal n. The local reflection behavior at each point in the scene is modeled by the *bidirectional reflectance distribution function* (BRDF) [Nic65] f_r. The BRDF converts differential irradiance arriving from a given direction ω_i at the surface point x to an amount of reflected differential radiance in an outgoing direction ω_i of interest. In order to be considered physically plausible, a BRDF must meet certain requirements. One such an important property it must exhibit is the preservation of energy. Energy preserving BRDFs never reflect more light than arrives at the surface, or in other words, they do not produce any energy on their own. This must hold for any incoming direction and is usually ensured by a normalization constant that corresponds to the reciprocal maximum amount of totally reflected intensity. The simplest BRDF is the *Lambertian BRDF* [AMHH08] which models the ideal diffuse reflection described earlier and is depicted in figure 2.1. It consists only of the normalization constant ($\frac{1}{\pi}$) and an albedo value which accounts for the absorption behavior of the surface at the given point. More complex BRDFs, like the ones used to realize glossy reflections, however, have to additionally consider incoming and outgoing direction. Those BRDFs are not further discussed here since the approach presented in this work primarily focuses on the simulation of diffuse light propagation by approximately solving the rendering equation while using the Lambertian BRDF.

2.3 Rendering

The difficulty of computing global illumination by solving the rendering equation stems from its recursive nature. To determine the radiance at some point, the reflected radiance towards this point from everywhere in the scene must be known, which again requires knowledge of the radiance at every other point, including the point for which the radiance is supposed to be computed in the first place. A closed solution to the rendering equation does consequently not exit in general. Approaches to global illumination often substantially differ depending on the render times aimed for. Computationally heavy algorithms like *path tracing* [Kaj86] or *radiosity* [Spe93] are traditional representatives in the field of non-interactive applications. There, single images are rendered by offline processes and are not required to be available in the fraction of a second as is usually the case for interactive or real-time applications like visualizations or games. Path tracing attempts to find an approximate solution to the rendering equation by adding up light contributions of a large amount of paths that the light takes through the scene to reach the sensor of the virtual camera. The paths itself are constructed outgoing from the the image pixels into the scene. Every path segment corresponds to a straight connection between two surface points or between a pixel and a surface point in the case of the first segment. The light contribution along a path is then calculated by virtually backwards propagating the radiance every surface point emits to the screen pixel while accounting for the amount of reflected light at each surface by evaluating its corresponding BRDF. For every path, the contribution due to direct lighting is implicitly computed along the first two path segments (image pixel to first surface point, first surface point to potential light source surface), while all successive segments gather indirect light. Radiosity takes a different approach where the scene is first discretized into flat surface patches. Diffuse light propagation is then realized by gathering the incoming irradiance at each patch from every other patch and converting it to reflected radiance. This gathering process is iterated where every iteration computes an additional bounce of diffusely reflected light. Direct lighting is therefore realized by the first iteration and indirect lighting by the subsequent ones. The caching of the light information directly at the geometry instead of at the image pixels like in the case of path tracing allows to move the observer without having to recompute the global illumination. On the downside, following the original design only diffuse light propagation can be simulated. Any extensions to support more complex BRDFs would inevitably introduce view dependencies which require the observer to be fixed. The algorithm presented later in this work will apply similar ideas to construct multiple diffuse light bounces.

Interactive applications traditionally avoid the costly computation of dynamic global illumination altogether by only evaluating local illumination models or merely extend those by visibility terms that account for shadowing to at least simulate interactive direct illumination. While the local lighting models used in real-time rendering often do not differentiate anymore from the ones considered, for example, for film production [Bur12][Kar13], it is still necessary to handle shadow computations by dedicated algorithms like *shadow mapping* to fit the tight frame time budgets demanded by interactive or even real-time frame rates. Traditional shadow mapping [Wil78] first renders the *occluder depth*, the distance from scene geometry to a given light source as seen from the light itself, to a so called *shadow map*. In a second pass the occluder depth is projected into the scene and compared against the *receiver depth*, which likewise corresponds to the distance to the light source, but this time rendered from the perspective of the observer. If for a certain point the receiver depth turns out to be greater than the occluder depth the point is considered to be in shadow. This binary shadow test produces unnaturally hard shadows and by that exaggerates the approximation character of point lights. Percentage closer filtering [RSC87] can be used to address this problem by filtering over a set of shadow test, producing a penumbra of constant size. *Variance shadow maps* (VSMs) [DL06] and *exponential shadow maps* (ESMs) [AMS$^+$08], in contrast, belong to a subset of shadow mapping algorithms, that allow their shadow map variants to be pre-filtered to produce soft shadows by a single subsequent shadow test. VSMs replace the traditional shadow test by a probabilistic one based on an occluder depth distribution parametrized by mean and variance. ESMs, on the other hand, approximate the step function of the shadow test with exponential terms for occluder and receiver depth. It can be shown, that, under some assumption, both solutions allow to pre-convolve the occluder depth without consideration of the receiver depth. However, due to the involved approximations and assumption that do not always hold, both variants suffer from light leaking artifacts. With VSMs, light leaking occurs in regions of high depth complexity, usually originating from geometric discontinuities where mean and variance alone fail to accurately capture the actual depth distribution. ESMs, being based on a smooth approximation of the shadow test, cause light leaking artifacts primarily behind occluders but also in some border cases where the receiver depth is actually smaller than the occluder depth. A straightforward way to tackle these issues is to take the minimum of both results. Although *exponential variance shadow maps* [Lau08] provide a more sophisticated way of combining VSMs and ESMs to reduce light bleeding, a simple minimum already leads to distinct qualitative improvements and is used to simulate the direct lighting from point lights for the approach presented in this thesis.

In the following chapter, interactive solutions that additionally aim to capture the effects of dynamic indirect lighting are discussed.

3 Previous Work

Global illumination approaches like path tracing, radiosity, and *photon mapping* [Jen96], while achieving high quality results, often exhibit computation times ranging from minutes to several hours. Direct application of these solutions to interactive scenarios is therefore not feasible. A possible tradeoff is to assume static geometry, static light sources, and view independent diffuse light propagation which makes it possible to completely pre-process and store global illumination for later use at runtime. However, for many interactive applications such strong limitations are inconvenient and less restrictive solutions are desirable. In the following, several of such approaches are outlined.

Aiming for shorter computation times, *instant radiosity* [Kel97] employs the idea of virtual point lights (VPLs), which are, outgoing from primary light sources, distributed into the scene by ray casting over multiple bounces. These VPLs serve as a point based approximation of reflected radiance in the scene. During a second pass, the contribution of this radiance to the final image is evaluated by accumulating the direct light from all VPLs for each image pixel using hardware accelerated shadowing techniques like shadow volumes or shadow mapping. By doing so, instant radiosity reaches rendering times in the range of seconds to minutes, depending on scene complexity and chosen shadow algorithm. While considerably faster than aforementioned approaches, the large number of costly VPLs required to produce a single image makes instant radiosity not directly applicable to interactive scenes even when favoring the faster shadow maps over high quality shadow volumes. Furthermore, while Instant Radiosity is well suited to simulate diffuse light propagation, support for glossy materials is limited since strongly focused reflections expose the VPLs to the viewer. The derived algorithms described in the following inherit this limitation.

To reduce the number of VPLs and associated shadow maps that have to be rendered every frame, [LSK$^+$07] propose an incremental real-time capable approach for semi-static scenes and a single bounce of indirect light where only a subset of VPLs has to be updated per frame. The actual number of newly rendered shadow maps, however, strongly correlates with the light movement between frames and scene complexity and can therefore lead to inconsistent rendering quality due to the amount of VPL updates being limited by a fixed budget to ensure stable frame rates. Furthermore, since every additional light source introduces its own set of VPLs and therefore results in a significant performance penalty, only a small number of lights is feasible. On the upside, even though only static geometry is considered for VPL placement, dynamic geometry can still receive lighting from the VPLs. Nevertheless, the approach remains too restrictive for many interactive scenarios while suffering from inconsistent quality.

[RGK$^+$08] introduce *imperfect shadow maps* (ISMs) to tackle the issue of the costly shadow map rendering. Instead of rasterizing continuous geometry, ISMs employ point based rendering to accelerate shadow map generation, utilizing a sparse, splat based scene approximation. The resulting imperfect shadow maps, while rendered substantially faster then common shadow maps, inevitably suffer from imperfections in their depth description, hence the name. To improve their quality, a hole filling approach is applied that attempts to partially recover continuous depth values. Quality deficits in the ISMs are further masked during accumulation of the VPL contributions since every ISM is rendered from a unique set of scene samples, independent from the sets assigned to any other ISM. Although hundreds of ISMs can be rendered at high frame rates, their coarse nature can be problematic for nearby geometry and therefore indirect illumination over short distances. Moreover, even for dynamic scenes of relatively low complexity only interactive frame rates are achieved since the number of scene samples and ISMs has to be increased compared to static scenes to achieve temporally coherent results without flickering artifacts. Another aspect worth considering is that while ISMs map naturally to point based rendering, point clouds usually contain much more points than would be feasible to render for each ISM. Therefore the use of an importance sampled scene description can in general not be avoided. In the context of their point cloud based global illumination approach, [PW10] propose to use the original set of points for ISM rendering but nonetheless render every scene sample only once for a single randomly selected VPL.

Cascaded light propagation volumes (LPVs) [KD10], designed to fit in the tight budget of real-time applications, take a different approach compared to the previously described VPL-based techniques. Instead, a dense, volumetric representation of the scene is built around the observer where each voxel serves as a VPL. For each voxel the light emission characteristic of the associated VPL is stored as a directional distribution of intensity. Every frame the direct lighting in the scene is injected into a subset of VPLs. This injected light is subsequently propagated inside the VPL volume by an iterative diffusion-like process, potentially considering opacity information stored in a second set of grids to account for occlusion and secondary reflection events. Afterwards, the resulting spatial and directional light distribution is sampled per fragment to compute the diffusely reflected radiance due to indirect illumination. In addition, one bounce of glossy reflections can be approximated by accumulating samples along the reflection vector. Doing so effectively collects incompletely propagated light which suggests that the effectiveness of this heuristic actually diminishes with increasing quality of the light propagation. In practice, LPVs often exhibit light leaking artifacts and overall mediocre quality due to the discretizations induced by limited volume resolution and flux distribution precision. Since memory consumption of dense grids scales badly with increasing resolution, multiple nested grids of increasing scales but equal voxel counts are used to cover larger scene portions.

[CNS$^+$11] propose to discretize the scene in a sparse, hierarchical voxel grid that allows to continuously query scene information at different levels of precision while avoiding to waste large amounts of memory for empty voxels as is typically the case with dense grids. Every voxel approximates the scene geometry in its neighborhood by a set of attributes that include albedo, opacity, and a distribution of normals. The further down a voxel lives in the hierarchy, the narrower its corresponding neighborhood and therefore more accurate, although locally more restricted, its scene description. Initial to the light propagation stage, direct lighting is splatted into the leaf voxels and subsequently filtered up the hierarchy which allows incoming radiance to be queried side by side with geometry attributes over scene regions of arbitrary extent. Stepping through the volume along a straight path while sampling increasingly coarser hierarchy levels, approximates the footprint of a cone. During the rendering phase, this *voxel cone tracing* (VCT) scheme is used to estimate the incident radiance at each fragment by performing VCT starting from the fragments positions into several directions to gather the lighting information stored in the voxel structure. The gathered light is subsequently used to compute the reflected radiance at each fragment. Based on the comprehensive surface and lighting information provided by the voxel hierarchy, the cone tracing algorithm is able to produce not only the effects of diffuse light propagation but also view dependent indirect glossy reflections. Where LPVs where only capable of approximating glossy highlights of previously diffusely reflected light, this approach additionally captures the effect of twofold glossy reflections. On the down side, the high memory requirements exhibited by the voxel structure potentially exceed the budget of many interactive applications. Furthermore, even for moderate screen resolutions only near real-time frame rates are achieved. Although the voxelization of static objects is implemented as a pre-processing step, the high performance costs of dynamic voxelization and per fragment VCT remain. It is, however, possible to omit the dynamic voxelization of selected objects. While not being considered during light propagation, these objects can still receive indirect light from their surrounding.

[McL14] also proposes to use VCT to simulate the light propagation but turns back to nested, dense grids. Compared to [CNS$^+$11], the scheme is further simplified by storing diffusely reflected radiance at each voxel instead of directional distributions of incoming radiance. While twofold glossy reflections are consequently not supported, a single bounce of glossy reflected light can be added by cone tracing into the volume along the reflection vector at each fragment. The dense nature of the volume structure allows the surface voxelization to be substituted with a dense one to improve the quality of the scene approximation of coarser hierarchy levels without introducing additional memory requirements. The approach achieves real-time frame rates by the use of several auxiliary volumes to speed up the costly

VCT pass. However, these additional volumes cause high memory requirements.

4 The Algorithm

This chapter describes the global illumination approach developed in this work in general before the subsequent chapter outlines details of the implementation. The presented solution is primarily based on the work of [CNS$^+$11] and [Cra11] but also incorporates some ideas from [McL14] to simplify the approach and to reduce memory consumption. The algorithm runs primarily on the GPU and can be roughly subdivided into three phases: construction and updating of a sparse hierarchical voxel structure, light injection and propagation based on this hierarchy, and the final rendering of the scene where the global illumination information is sampled from the voxel structure and combined with analytically evaluated lighting. The voxel hierarchy is directly built from point clouds which represent a sampling of an either real or virtual scene, or a combination of both. Individual points are therefore referred to as *scene samples* in the following. Every scene sample is at a minimum defined by a position, an opacity, an albedo, and a *normal distribution function* (NDF) where the latter three are referred to as *geometry attributes*. NDFs encode distributions of normals [Fou92]. Here, each NDF corresponds to a three dimensional vector whose direction equals the mean normal while its length correlates with the variance of the encoded normals [Tok04]. A set of normals can be converted into such a NDF by a simple average. NDFs allow the scene samples to more accurately describe surface regions that exhibit variations in their normals. This is, for example, useful to encode the normals of surfaces adjacent to edges along which scene samples might be taken. The opacity, on the other hand, measures the amount of solid geometry that a scene sample represents. An opacity of 1 corresponds to completely opaque and 0 to entirely transparent. Scene samples can additionally store a radiance which allows them to describe surfaces of area lights. Each point cloud is the result of an arbitrary sampling process. This could, for example, be a GPU rasterizer based scene sampling as is proposed by [CNS$^+$11] or an irregular sampling of the scene geometry generated on the CPU or even by a laser scanner. The here described approach assumes that the scene sample attributes are kept in no particular order in a structure of arrays in global GPU memory. It would be, however, theoretically possible to omit these buffers in the case that the sampling is generated on the fly.

The following two sections first outline how the scene itself is rendered and how the direct lighting is handled. Afterwards, all voxel structure related computations are described in detail. The last section of this chapter deals with the rendering of the final image.

4.1 Scene geometry rendering

While the light propagation is simulated based on the voxel structure, the final rendering as well as the computation of shadow maps still requires the original geometry of the scene to be rendered in some way. An object is either rendered as a triangle based mesh or as spherical *surfels* (surface elements) positioned at its scene sample locations (see figure 4.1). In general, for every object the faster method is preferred. If an object is exclusively represented by its point cloud and no triangulation exists, as might be the case when working with laser scanner data, the surfel based approach is applied in any case. The spherical surfels are rendered with a simple ray casting approach, ignoring perspective distortion. While the missing distortion results in slightly incoherent depth ordering, the heavy view angle dependent inconsistencies that typically occur with flat, observer aligned geometry, are avoided. On the downside, the resulting surface representation is not smooth since it is composed of intersecting spheres. If a smooth result is desired, this simple approach, however, can be substituted by a more involved one like a splat based blending [PJW12].

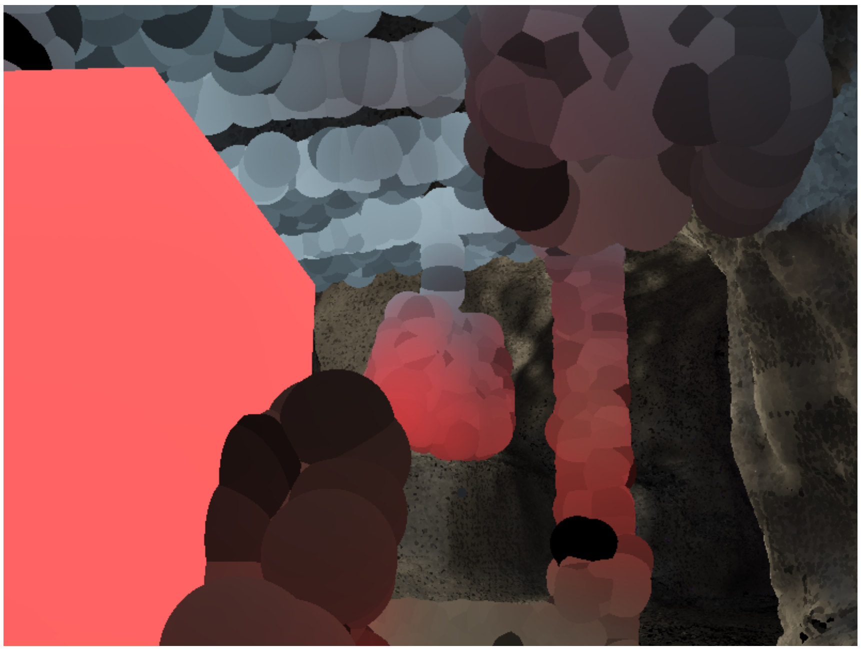


Figure 4.1: *A section of a scene that consists of a large point cloud rendered with spherical surfels and a pink emissive cube rendered as a triangle based mesh.*

4.2 Direct illumination

Both point and area lights are supported by the proposed algorithm. Area lights are realized by emissive scene samples that can represent surfaces of arbitrary shape. Their direct lighting is implicitly simulated by the voxel structure based light propagation. Consequently, no additional steps need to be performed. In contrast, direct illumination from point lights is evaluated analytically using a simple local illumination model while accounting for occlusion by a minimum of VSM and ESM as explained in the *Rendering* section of the *Global illumination* chapter. The shadow maps are pre-filtered with a 5x5 taps disk shaped kernel to create approximate penumbras. Directly computing the first light bounce results in a much higher quality than could be achieved by a voxel structure based simulation. The evaluation of direct lighting of point lights needs to be performed twice, once during the light injection stage and a second time for the final rendering of the scene as will be explained later. The local illumination is based on a simple Lambertian BRDF since the algorithm focuses on diffuse light propagation. Therefore, a position, an albedo, and a normal have to be provided additionally to the light source specific parameters to compute the direct lighting from point lights at a given position.

4.3 Hierarchical voxel structure

The simulation of light propagation over multiple bounces is based on an alternative volumetric representation of the scene. This voxel structure, in contrast to the original geometry, allows to continuously sample geometry attributes and radiance in the scene. By building a mipmap pyramid over the leaf voxels, the base structure is turned into a hierarchical scene representation that is subsequently exploited to simulate the diffuse light propagation. To reduce memory consumption, only voxels that represent non-empty scene portions are explicitly stored, which, on the downside, makes sampling and updating much more involved compared to a dense structure.

The following subsections first address the underlying design of the voxel structure and its construction before the individual steps of the actual light propagation simulation are described subsequently.

4.3.1 Description

The voxel hierarchy is realized as a *sparse voxel octree* (SVO). Every node of this octree represents a non-empty portion of the scene and might reference up to eight *child nodes* (see figure 4.2) which in turn serve as a more precise, although more locally restricted scene description. Starting from the *root node* which represents the hole scene, the nodes are successively subdivided into child nodes until either a given tree *depth* is reached or the associated scene portion of the current node is empty. The deeper the octree the more precise its scene representation on the lowest level. The nodes on this lowest level are in the following referred to as *leaf nodes* (in contrast to the common definition of 'leaf nodes'). The terms *node* and *voxel* are used interchangeably to some degree due to the volumetric scene description associated with the nodes. The tree structure is stored in a linearly organized *SVO-buffer* that is pre-

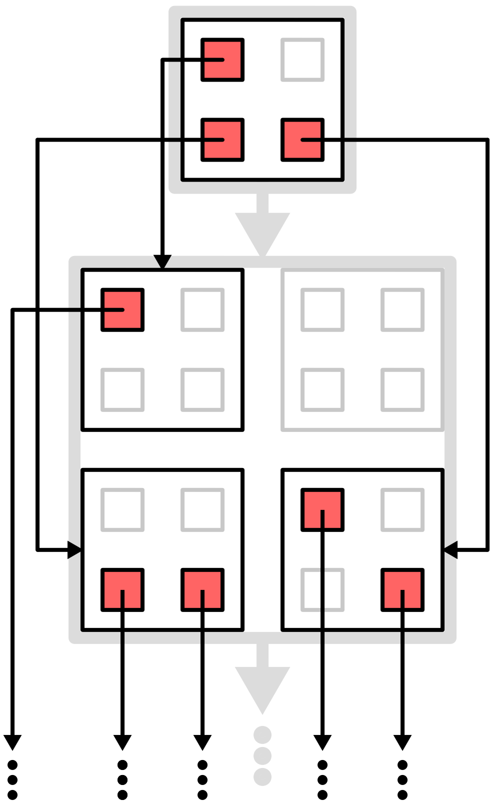


Figure 4.2: *2d depiction of the first two SVO levels. Every non-empty node (red) references up to 2x2 child nodes (2x2x2 in 3d). The thick, gray frames and arrows illustrate the tree level hierarchy.*

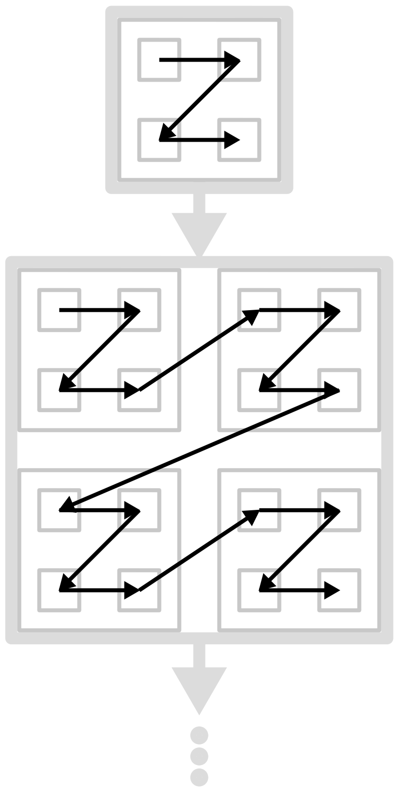


Figure 4.3: *2d depiction of the morton order of the nodes on the first two SVO levels. Recursively applying the Z-order curve to child nodes results in a linear enumeration of all nodes.*

allocated in global GPU memory. To improve data coherency and compactness, child nodes are grouped together in *tiles* of eight nodes. This way, a parent node can reference all its children by a single *tile pointer*. In this context a pointer refers to an element-wise memory offset relative to the beginning of the buffer that contains the target element. Due to the tile based layout, the root node is not explicitly stored in memory since a corresponding tile does not exist. Instead, its children grouped together in the pre-allocated *root tile* form the highest available tree level. Additional to the tile pointer, every node also stores a couple of bit flags used during the construction phase and six *neighbor pointers* which are useful to quickly visit spatially adjacent nodes without having to traverse the SVO. The memory layout of the SVO is depicted in figure 4.4. Traversing the structure to find a specific node, however, can often

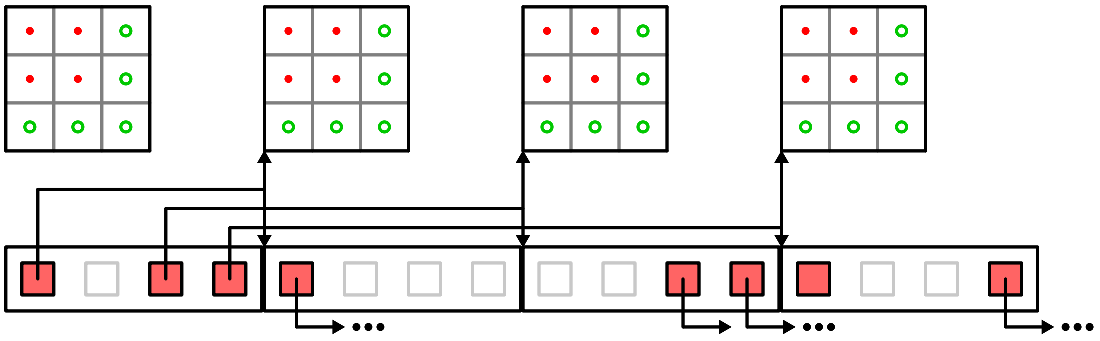


Figure 4.4: *The linear layout of the nodes and tiles inside the SVO-buffer (bottom row). The buffer contains the first two SVO levels as depicted in figure 4.2. Each tile consists of 4 nodes (8 in 3d). The tile pointer of each node also points to a brick in the brick pool (top row). The root tile (the first from the left) references its brick implicitly. No tile is allocated for the empty node in the root tile.*

not be avoided due to the sparse nature of the tree. The traversal of the SVO starting from the root tile down to a target node requires knowledge about the *local id* of the child node that has to be visited next on each level. This local id is an integer in the range $[0..7]$ since every node can reference up to eight child nodes. In each tile, the eight nodes are enumerated in *morton order* [Mor66]. This means that the three binary components of the node's position in its tile are concatenated and interpreted as a single integer which corresponds to the local id of the node (see figure 4.5). By not only considering the local id of each individual node but also the ids of all their predecessors, it becomes apparent that the nodes are consequently enumerated in morton order with respect to the whole tree and not just in their parent's tiles (see figure 4.3). The traversal down multiple tree levels, therefore, requires a sequence of local ids. This sequence of bit triples is called the *morton key* of the target node. The morton key of a node is computed by interleaving the bits of the components of its tree-local position. The coordinates of this position are integers that range from 0 to $2^l - 1$ with l being the tree level on which the node resides. For example, a leaf node in a tree of depth 8 would have a tree-local position with coordinates in the range $[0..511]$. Interleaving the bits of the coordinates then results in a single node index within a linear enumeration, the morton key (see figure 4.6).

The local scene description associated with every node is based on a number of voxel attributes. In the case of the leaf nodes, these attributes include RGB values for *reflected* radiance and the geometry

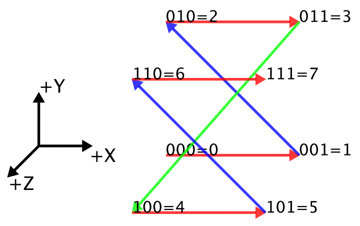
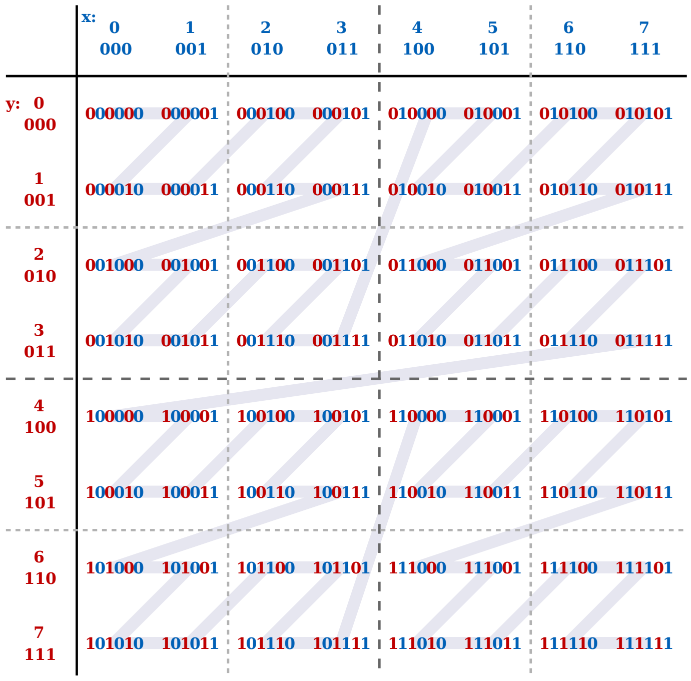


Figure 4.5: *The local ids of nodes inside a tile. Concatenating the three binary components of the position of each node results in the respective ids.*



	x: 0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
y: 0 000	000000	000001	000100	000101	010000	010001	010100	010101
1 001	000010	000011	000110	000111	010010	010011	010110	010111
2 010	001000	001001	001100	001101	011000	011001	011100	011101
3 011	001010	001011	001110	001111	011010	011011	011110	011111
4 100	100000	100001	100100	100101	110000	110001	110100	110101
5 101	100010	100011	100110	100111	110010	110011	110110	110111
6 110	101000	101001	101100	101101	111000	111001	111100	111101
7 111	101010	101011	101110	101111	111010	111011	111110	111111

Figure 4.6: *2d depiction of the construction of morton keys by interleaving the bits of the individual components of the tree-local positions of the nodes on the third tree level. [Epp10]*

attributes of the scene samples, i.e., opacity, albedo, and NDF. In practice, only the mean normal of the NDF is used. The mean normal can be derived from the NDF by normalization and is simply referred to as the *normal* of a node, sample, or fragment in the following. By storing NDFs instead of plain normals it is ensured that linear combinations of multiple nodes result in the correct mean normals which is important for trilinear sampling. In contrast to leaf nodes, non-leaf nodes store only *outgoing* radiance and opacity which are written into them during the mipmap construction phase as explained later. The difference between reflected and outgoing radiance is that outgoing radiance includes the amount of light the geometry might emit on its own additionally to any reflected light. Outgoing radiance is only temporarily apparent in the leaf voxels and is overwritten with reflected radiance during the light propagation phase. The voxel attributes are not directly stored in the SVO-buffer but are instead written to 3d textures where they are grouped together in *bricks* of 2x2x2 voxels. These bricks are directly referenced by the tile pointers that also reference their corresponding tiles in the SVO-buffer. This results in linear addresses for the bricks which are in turn translated to 3d texture coordinates when reading or writing the voxel attributes. This translation is necessary since 3d texture sizes are rather limited in each dimension [D3Db] which makes a sequential ordering of a large amount bricks along a single dimension inside the textures impossible. Contrary to the linear ordering of nodes inside tiles, the voxel positions inside the bricks correspond to their actual relative spatial positioning. This brick scheme allows the fast dedicated trilinear sampling of the GPU to be used to continuously query voxel attributes. However, a boundary of redundant voxels around every brick is necessary to ensure correct sampling between voxels that reside in different bricks. These *boundary voxels* mirror the adjacent voxels of the neighbor bricks which are only spatially adjacent but not necessarily in memory. Building a complete boundary would increase the brick size from 2x2x2 to 4x4x4 and therefore induce an increase in memory consumption by a factor of 8. Instead, only positive boundaries are added leading to a 3x3x3 voxels layout and only an increase in memory consumption by a factor of 3.375 (see figure 4.7). When a sample query falls on the negative boundary of a brick, it is simply moved to the appropriate sampling position in an adjacent brick where the positive boundary consists of the cloned voxels of the original brick (see figure 4.8). To guarantee that the necessary neighbors in negative directions exist, additional empty *border tiles* are added where required.

Due to the bricks being directly referenced by the tile pointers of the nodes in the SVO-buffer (see figure 4.4), every node that references a tile of child nodes automatically holds a reference to a brick of associated child voxels. The *leaf tiles*, containing the leaf nodes, are not explicitly allocated in the SVO-buffer since their associated bricks are already referenced by their parent nodes. Instead, a *leaf node bit field* referenced by the tile pointer of the parent indicates for every implicit leaf tile which leaf nodes are existent. This bit field is referenced through the same pointer that the parent of the leaf tile

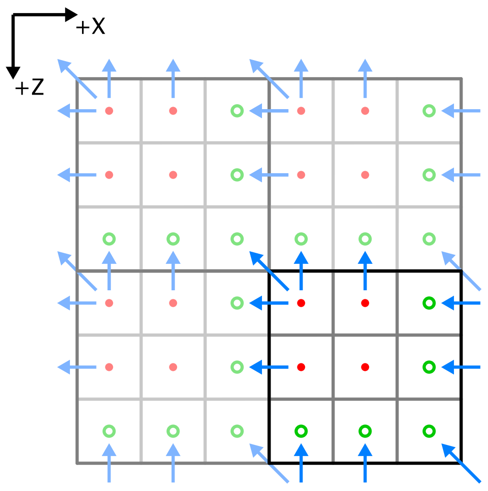


Figure 4.7: *2d depiction of the brick layout. Every brick (black squares) consists of 3x3 voxels (dots and circles, 3x3x3 in 3d). 5 of those are redundant boundary voxels (green circles, 19 in 3d) which are copies of the spatially adjacent voxels (red dots). The arrows illustrate this mapping.*

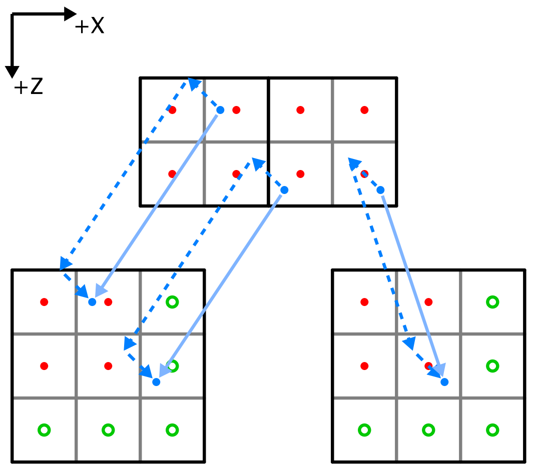


Figure 4.8: *2d depiction of the brick pool sampling scheme. The image shows two spatially adjacent tiles (top) and their respective bricks (bottom) which reside at random positions in the brick pool. The blue dots and arrows illustrate the mapping from sampling positions in space to bricks. Half-voxel-cell offsets ensure that the samples fall in the appropriate bricks and are consequently surrounded by the proper voxels (illustrated by the dashed arrows). This scheme moves the sample in the middle to the brick of its neighbor tile. If this neighbor tile does not exist after the construction phase, it is added as a border tile in the case that one of the two adjacent voxels of the right tile is non-empty. The final sampling positions of the other two samples are unaffected by the offset scheme.*

also uses to refer to the brick containing the leaf voxels.

The 3d textures that store the bricks are in the following referred to as *brick pools*. There are four different brick pools in total: the geometry pool, the radiance pool, the mipmap pool, and the intermediate pool. The mipmap pool keeps outgoing radiance and opacity values of the non-leaf nodes while both geometry pool and radiance pool exclusively contain attributes of the leaves. The geometry pool stores albedos, opacity values and NDFs of the leaf voxels while their reflected radiance is kept in the radiance pool. The radiance pool also provides backup capabilities required for dynamic updates of the SVO. Voxel attributes are often temporarily written to the intermediate pool which can become necessary due to GPUs being currently not able to simultaneously read from and write to the same texture in a single pass except for restrictive texture formats [D3Da]. Voxels with lower opacity are supposed to be weighted lower than those with a higher opacity value in a trilinear sample. This is accounted for by pre-multiplying albedo, NDF, and radiance with the opacity. The un-multiplied value of a sample is then recovered by division by the opacity of the sample itself which usually only becomes necessary in cases where the opacity value is guaranteed to be non-zero. Since the contributions of the scene samples are additively accumulated in the leaf nodes, the opacity can exceed the physically plausible upper bound of 1 which implies a completely opaque node. In practice the opacity of a trilinear sample is therefore clamped to 1 for usage other than the retrieval of the un-multiplied albedo, NDF, and radiance.

4.3.2 General purpose computations

All SVO related computations described in the following are, to some degree, realized with the help of general purpose computing on the GPU (GPGPU). Consequently, a short overview of how these computations are handled by the algorithm appears appropriate. Frameworks like CUDA [CUD] and OpenCL [Ope] make GPGPU capabilities accessible outside of graphics APIs. In the here considered case, however, the general purpose computations are part of a graphics application which uses a graphics API anyway. Therefore, shaders are used to construct and update the SVO as well as to simulate the light propagation. Since graphics APIs currently provide no dedicated functionality to perform atomic additions on resource formats other than `int` and `uint` without using the output merger stage of the graphics pipeline [D3Da][OGL], general purpose computations are emulated by traditional shaders instead of being handled by compute shaders which are specifically designed for GPGPU tasks in graphics contexts but have no access to the output merger. Atomic additions become necessary when writing geometry attributes and radiance to the brick pools, which are using floating point formats. The GPGPU emulation is realized by performing (indirect) draw calls without having vertex or index buffers bound to the pipeline. The number of desired threads is then passed by the CPU or GPU to the draw call as the number vertices that is supposed to be processed. The vertex shader functions only as a pass-through stage that forwards vertex ids to the following stages. The actual computations are then performed by geometry or fragment shaders based on the bound resources and the vertex ids which are conceptually interpreted as thread ids. For other computations that do not atomically write to the brick pools, like the construction of the spatial base structure of the SVO, compute shaders could be used instead.

4.3.3 Construction

The construction of the SVO is separated into two major phases. First, the spatial base structure is built inside the SVO-buffer from the positions of the scene samples. Afterward, the geometry attributes are written to the bricks referenced by the nodes in the SVO-buffer. The radiance of emissive samples is added later on a frame-wise basis during the radiance injection stage.

To construct the spatial structure in the SVO-buffer, for every scene sample a thread is created that first computes the morton key of the leaf node to which the sample contributes. Based on the morton key, every thread then traverses the SVO starting from the pre-allocated root tile. Every time a thread encounters an empty non-leaf node, the node gets subdivided. A node is considered to be existent or non-empty when it references a tile of child nodes and a brick in the brick pool. Subdivision of a node therefore involves allocating a tile of child nodes and writing the tile pointer to the node. Allocations inside the SVO-buffer are based on the incrementation of an atomic counter that all threads share. To allocate a new tile when subdividing a node, a thread increments the counter and uses the returned value as tile pointer. Race conditions between different threads that try to subdivide the same node are prevented by two atomically accessed bits. When a thread visits a node it first checks the states of the *exist bit* and the *locked bit*. Depending on these states, three different situations can occur. If the node does not exist (exist bit not set) and is not locked (locked bit not set), the thread sets the locked bit, subdivides the node, sets the exist bit, and continues its traversal. In the case that the node is currently being subdivided by another thread (exist bit not set but locked bit set), the thread writes its morton key and the reached position in the SVO-buffer to a *queue buffer* in preparation of deferred re-execution and terminates. The third case occurs when the exist bit is set, that is, the node is existent. Then, the thread simply continues its traversal, ignoring the locked bit. When a thread reaches a leaf node parent, it sets the appropriate bit in the leaf node bit field referenced by the tile pointer to indicate that the leaf node exists. If the leaf node parent itself does not exist yet, it gets subdivided similarly to the preceding nodes. However, the allocation of leaf tiles uses its own dedicated atomic counter since their voxel attributes are stored in different brick pools (geometry and radiance pool) than the ones of non-leaf tiles (mipmap pool) and therefore require a separate address space.

This traversal/subdivision procedure is repeated for the prematurely terminated threads which are put asleep in the queue buffer. In the worst case, a thread is repeatedly put asleep while descending the tree one level per iteration. The upper bound of iterations required to ensure that the queue buffer is emptied and the spatial structure of the SVO is completed, therefore, is given by the tree depth minus one (the first level is pre-allocated). In practice often only a couple of iterations are required to empty the queue buffer. The number of iterations depends, among others, on how the scene samples are distributed in the scene. A more widely spread, even distribution results in less collisions than a set of samples that contains clusters. Since GPUs are currently unable to autonomously issue draw or dispatch calls, the maximal amount of iterations is performed, potentially resulting in some minor overhead for iterations that do not generate any threads.

The neighbor pointers for every node are determined by simply traversing the tree top down to the adjacent nodes. Every node searches its three potential neighbors in the positive directions. If a neighbor is found, its address is set as the respective neighbor pointer in the reference node while the address of the reference node itself is written to the neighbor as neighbor pointer in the negative direction. This way, all neighbor pointers in positive as well as negative directions are set without requiring atomic operations.

To complete the spatial base structure of the SVO, the border tiles which are required to ensure correct trilinear filtering need to be added. From the perspective of a single tile there are up to 26 tiles surrounding it. Seven of these tiles are neighbors in negative directions. This includes three *direct neighbors* (-X, -Y, -Z) which can be visited by use of the neighbor pointers in the parent node. The other four tiles are *indirect neighbors* (-XY, -XZ, -YZ, -XYZ). If one of these seven tiles does not exist, it has to be added as a border tile because a trilinear sample taken in the brick associated with the reference tile can potentially be moved to a brick of one of these neighbor tiles due to the lack of negative redundant boundary voxels (see figure 4.8). The most straightforward way to add these border tiles is to create seven threads for each tile which then traverse the SVO and check whether the necessary tiles exist and, if not, allocate them. This approach, however, generates a large number of redundant threads that try to allocate the same border tiles since many tiles share their direct and indirect neighbors with others. By exploiting the neighbor pointers the number of this redundancy can be greatly reduced. The most obvious optimization is to simply check whether a direct neighbor already exists with the help of the neighbor pointers. This simple approach can be extended to visiting indirect neighbors by following two or three neighbor pointers. Furthermore, depending on which nodes are actually existent in the reference tile, it might not even be necessary to allocate certain tiles. In the extreme case where only the node with the local id 7 (binary relative position 1, 1, 1) exists in the tile (see figure 4.9), no border tiles have to be added at all because the empty nodes already provide a natural boundary of completely transparent voxels in the negative directions. Therefore, in a first pass a bit field indicating for each tile which boundary tiles are required is generated depending on which nodes in the tile exist. During a second pass these bit fields are used to make assumptions about what border tiles the direct and indirect neighbors of a reference tile try to allocate. Based on these assumptions the number of generated threads is considerably reduced. After allocation of the border tiles, the neighbor pointers of their parent nodes have to be set. This time, two passes for every border tile are necessary; one for the positive and one for the negative neighbor pointers. Otherwise, in border cases where a non-border tile is adjacent to a border tile in a negative direction, the respective neighbor pointers would not be set for both tiles.

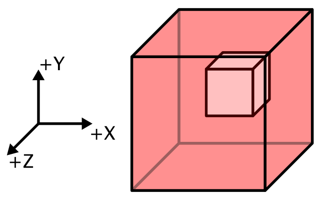


Figure 4.9: *A tile (large cube) that contains only the node with the local id 7 (small cube). No border tiles have to be added to ensure correct trilinear sampling since the empty nodes already provide a seam of black auxiliary voxels in the negative directions. In the positive directions auxiliary voxels are provided by the redundant boundary voxels in any case.*

After finishing the construction of the spatial base structure, the albedo, opacity and NDF values are additively written to the bricks referenced by the leaf nodes parents. The attributes are first rendered to the intermediate pool and are copied to the geometry pool during a *completion* step which writes the redundant boundary voxel with the help of the negative neighbor pointers (see figure 4.7). Thanks to the border tiles it is always ensured that all existing indirect neighbors can be accessed over multiple neighbor pointers.

At this point it is possible to trilinearly sample the geometry attributes at every point in the scene. For static parts of the scene, the construction described so far is performed only once at initialization time. Partial dynamic updates to the structure are realized on a frame-wise basis following a similar scheme. A more detailed explanation is provided later in the *Dynamic updates* subsection.

4.3.4 Antialiasing

Up to now every scene sample exclusively contributes to the leaf node in whose cell its position falls or with other words to the leaf node closest to its position. This binary assignment can lead to aliasing in the scene description built during the construction process. The induced artifacts become especially apparent for dynamic geometry. To tackle this problem, a cubical *region of influence* (RoI) aligned with the main axes of the SVO is assigned to every scene sample. The maximal size of this RoI corresponds to the size of the leaf node cells and shall be defined as 1. The amount a scene sample contributes to a leaf node is then determined by the intersection volume of the sample's RoI and the cell of the leaf node (see figure 4.10). Following this scheme, every scene sample writes to up to eight leaf nodes while the contributions correlate with the distances between samples and nodes. For each leaf node that a scene sample tries to allocate, a new thread is created on the fly by generating additional point primitives in a geometry shader preceding the fragment shader that handles the construction of the SVO. A meaningful RoI size for a scene sample could, for example, correspond to the minimum distance to the next sample normalized by division by the leaf node cell size of the SVO and clamped to 1. By using the maximum norm as distance metric instead of the euclidean norm, this approach fills up the spaces between the scene samples without producing overlaps in the RoIs. For a dense scene sampling, this can result in small RoIs which in turn lead to small contributions by the individual samples. For a volumetric sampling, this does not display a problem. However, in the case of a surface sampling, the overall contribution of all samples to the leaf nodes approaches zero with increasing sampling density. To compensate for this effect, the opacity values of the samples are allowed to become larger than 1. This is still physically plausible as long as the values stay under the reciprocal of the volumes enclosed by the RoIs since the opacity values now actually represent opacity *densities*. The opacity of a sample is computed by integrating over parts of its RoI which corresponds to the intersection volume calculations performed to determine the partial contributions to individual leaf nodes by the sample.

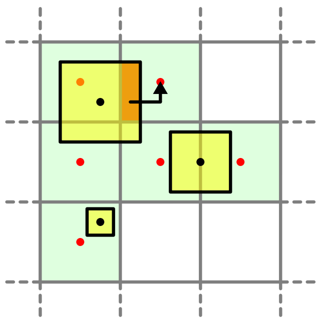


Figure 4.10: *2d depiction of the RoI based antialiasing scheme. The intersection areas (volumes in 3d) of the RoIs (yellow squares) and the voxel cells (green squares) determine the contributions of each scene sample (black dots) to the individual leaf voxels (red dots). For example, the orange area corresponds to the amount the topmost sample contributes to the voxel pointed out by the arrow.*

4.4 Light injection and propagation

To simulate the propagation of light with the help of the SVO, the radiance from the light sources is injected into the leaf nodes each frame. Afterwards, a hierarchical presentation of this radiance and the opacity is built. By exploiting the resulting mipmap, the diffusely reflected radiance at each leaf node is propagated into the scene. These three stages are described in more detail in the following.

4.4.1 Radiance injection

In preparation of the light propagation phase, the light contribution from all light sources is additively rendered to the radiance pool which at that point already contains propagated radiance from previous frames. This process differs depending on the light source type. Area lights are realized by scene samples that keep an RGB value of emitted radiance additionally to their geometry information. This radiance values are directly rendered to the radiance pool, following the same approach of how the geometry attributes are written to the geometry pool. In the case of point light sources, the direct lighting is evaluated at every leaf node following the approach depicted earlier in the *direct illumination* section. The necessary albedos and normals are provided by the geometry attributes of the leaf nodes. Alternatively, it would also be possible to apply a splatting scheme where every shadow map texel of each light is treated as a *photon*, a small light package that is, by traversing the SVO just like a scene sample during the construction phase, added to the radiance of the leaf voxel in whose cell it falls. In cases where only a small portion of leaf voxels is affected by direct lighting from point lights, the overhead that the gathering approach produces by evaluating the direct lighting for completely shadowed leaf voxels would be avoided. On the down side, if the photon coverage is not high enough, voxels that should be illuminated can be missed and consequently receive no light. It might be, however, possible to reduce this artifact by applying the RoI based antialiasing scheme to the photons as well.

Before the reflected radiance is added to the radiance in the brick pool, it is additionally multiplied by the opacity of the node to ensure correct weighting during trilinear sampling.

4.4.2 Mipmapping

At this point, a hierarchical presentation of radiance and opacity in the form of a mipmap is built outgoing from the leaf nodes by successively filtering these values to nodes of higher tree levels. During each iteration, the nodes of the next higher tree level average the radiance and opacity of their child nodes and write the result to the respective bricks in the mipmap pool. This process is repeated until the nodes in the root tile, which provide the coarsest scene description, are filled.

After completion of the mipmap bricks, i.e., writing their boundary voxels, outgoing radiance and opacity can be sampled at any position at different resolutions. The lower the resolution of a sample, i.e., the higher the mipmap level it is taken on the larger the scene region it approximates. Continuous sampling between mipmap levels is realized by taking two trilinear samples at the closest discrete mipmap levels and blending them together to a single quadrilinear sample.

To improve the quality of the pre-integration provided by the mipmap, the isotropic voxels produced by the simple average described so far are substituted with anisotropic ones. Such an anisotropic voxel consists of six values instead of one, as is the case for isotropic voxels. Each of these six values corresponds to the appearance of a voxel when viewed along one of the SVO's main axes in positive or negative direction. The computation of each value for a given voxel starts with a compositing of child voxel pairs along the direction the value represents, i.e., -X, +X, -Y, +Y, -Z, or +Z. The applied compositing scheme corresponds to a simple alpha blending with pre-multiplied values [PD84]. Afterwards, the four results of this compositing step are averaged to the final value (see figure 4.11). Anisotropic voxels, therefore, provide a directional pre-integration in addition to a spatial one. During sampling, the six values, each stored in a separate brick, are weighted and blended together based on a given direction (see figure 4.12).

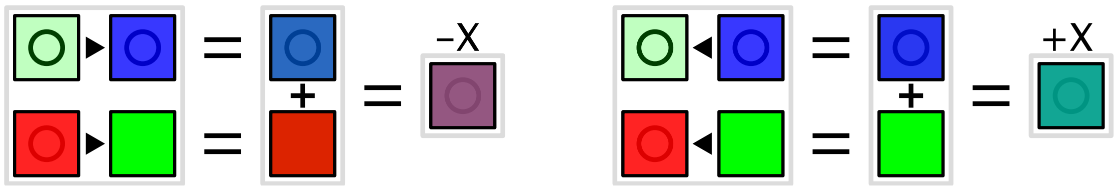


Figure 4.11: *2d depiction of the anisotropic filtering scheme. 4 isotropic child voxels (8 in 3d) are pairwise blended together along the main axes (only -X and +X are shown). For each axis, the resulting 2 values (4 in 3d) are subsequently averaged which accounts for the spatial pre-integration. The visibility of the circles illustrates the opacity values.*

When computing the value for a given direction from a set of child voxel that themselves are anisotropic, only their values that correspond to that direction are used. For example, the +Z value of an anisotropic voxel is exclusively computed from the +Z values of its anisotropic children. The first mipmapping step, however, has to compute anisotropic voxels from isotropic ones. Therefore, the NDFs of the isotropic leaf voxels are used to determine how much their radiance and opacity values contribute to each of their six virtual directions. One possible approach would be to set the contribution for each direction whose dot product with the voxel normal is greater zero to 1 and to 0 in the other cases. The approach used in the implementation uses a falloff based on the cosine between normal and direction that starts by 1 where the cosine is 1 and goes to 0 for a cosine of -1. This way, directions roughly orthogonal to the normal are not assumed to be totally transparent. The chosen approach for this first mipmapping step can have

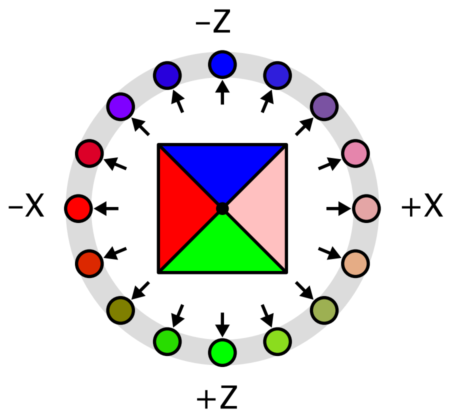


Figure 4.12: *2d depiction of the directional sampling of an anisotropic voxel. The 4 directional values (6 in 3d) are blended together based on given sampling directions. In practice only 2 values contribute to each sample (3 in 3d).*

significant impact on the result of the light propagation simulation.

4.4.3 Diffuse light propagation

The outgoing radiance stored in every leaf node is now propagated to every other leaf node in the SVO where it then contributes to the newly estimated reflected radiance. In practice, this propagation is realized as a gathering process at every leaf node. At a given leaf node, the incoming radiance is determined by integrating the outgoing radiance stored in every other leaf node over the hemisphere defined by the normal of this reference node. A traditional approach to numerically estimate this integral is to send a number of rays into the scene to gather the radiance at their intersection points with the geometry. However, since the algorithm here works on a volumetric approximation of the scene geometry, intersections between scene and rays can not be computed. Instead, the discrete volumetric rendering approach described in [Max95] that accounts for absorption and emission at every voxel is used to evaluate the radiance gathered along rays. Max first presents an ordinary differential equation that describes how the energy of light changes at every point in space when traveling along a ray in a volumetric medium that absorbs but also emits light. He then derives a general solution that computes the final radiance at the target position of the ray and shows that the approximation of the involved integral by a Riemann sum results in the familiar back-to-front compositing scheme which in turn can be transformed into the front-to-back compositing (algorithm 1).

While this model is usually employed to render participating media, here it serves to approximate partial contributions and occlusions of radiance along rays due to solid geometry. This ray based approach,

Algorithm 1 Front-To-Back-Compositing

```
1: procedure ComposeFrontToBack
2:     $fRadiance \leftarrow 0$   {the final radiance is built in this value}
3:     $fOpacity \leftarrow 0$   {the final opacity is built in this value}
4:     $smpPosition \leftarrow rayStartPosition$
5:     $i \leftarrow 0$
6:     while $i < stepCount$ do
7:         $smpPosition \leftarrow smpPosition + rayDirection$   {step along the ray}
8:         $smpRadiance \leftarrow SampleRadianceAtPosition(smpPosition)$
9:         $smpOpacity \leftarrow SampleOpacityAtPosition(smpPosition)$
10:        $fRadiance \leftarrow fRadiance + smpRadiance * (1 - fOpacity)$
11:        $fOpacity \leftarrow fOpacity + smpOpacity * (1 - fOpacity)$
12:        $i \leftarrow i + 1.$
13:    $fRadiance \leftarrow fRadiance + bgRadiance * (1 - fOpacity)$   {consider background radiance}
```

however, either produces noisy results or becomes quickly too costly depending on the number of rays used. Instead, the rays are substituted with cones by subdividing the hemisphere into a handful of conic sections (four to ten) whose sub-integrals are approximately estimated by voxel cone tracing (VCT) [Cra11]. Directly computing the amount of radiance every leaf node emits into a given cone is not feasible. Therefore, the previously built mipmap is used to approximate the radiance contribution of leaf nodes inside the cone by a number of quadrilinear samples of different resolutions taken while stepping along the cone's axis and blended together by front-to-back compositing (see figure 4.13). The negated cone axis also provides the direction for the sampling of the anisotropic voxels. To keep the shader complexity low, only the anisotropic voxels from the mipmap are considered since the isotropic leaf voxels require special handling. As a result, details only apparent in the leafs might be missed. To include the leafs in the cone tracing process, the compositing loop can be split into two separate loops where the first one builds quadrilinear samples from leaf and non-leaf voxels and the second one deals with non-leaf voxels exclusively as usual. Another advantage of ignoring the leaf voxels is that the radiance bricks do not need to be completed after radiance injection.

To calculate the reflected radiance under the assumption of a Lambertian BRDF, the contributions of the individual cones are cosine weighted, summed up and multiplied with the albedo of the node. The multiplication by the normalization factor of the BRDF and the reciprocal probability density, however, is substituted with a multiplication by an empirical factor to compensate, to some degree, the errors introduced by the approximation. This empirical factor is determined by comparing the overall brightness

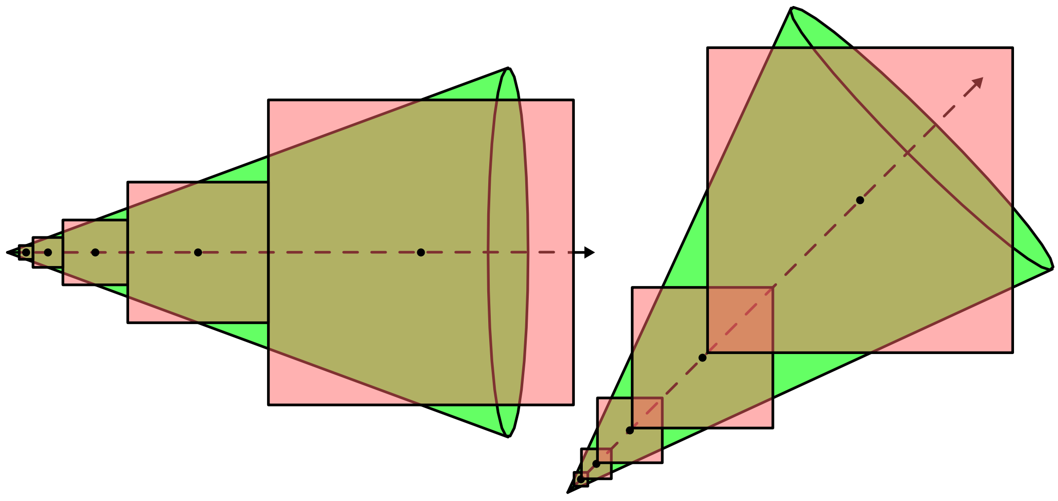


Figure 4.13: *The VCT sampling scheme. The footprint of the cone (green) is approximated by samples (black dots) of different resolutions taken along the cone's main axis (dashed arrow). The cone is partially under- as well as overestimated by the footprints of the samples (red squares). The approximation error also depends on the direction of the axis.*

of the final rendering with that of a high-quality reference rendering. The factor can heavily differ depending on the approach chosen to virtually convert isotropic leaf voxels to anisotropic ones during the first mipmapping iteration, as described in the *Mipmapping* subsection. For example, if no directional dependencies are derived for the opacity values from the NDFs during this earlier stage, a larger normalization to brighten up the result becomes necessary now due to self occlusion issues induced in exchange for reduced light leaking artifacts.

To verify that the voxel based approach is able to produce plausible results in the first place, it is possible to replace the cone tracing with a slow ray based variant that works exclusively on the leaf node level. This ray based variant produces results with a brightness similar to the reference rendering without any empirical adjustments (see figure 4.14).

After estimating the reflected radiance for a given node, the original radiance is overwritten. This injection/propagation procedure is evaluated every frame. Doing so progressively builds up multiple light bounces over time where every light bounce is completely re-computed every frame (see figure 4.15). As a result of distributing the light propagation over multiple frames, the n-th bounce lags $n - 1$ and $n - 2$ frames behind the current states of the area and point lights, respectively.

In preparation of the final rendering, the radiance bricks are completed to ensure correct trilinear sampling.

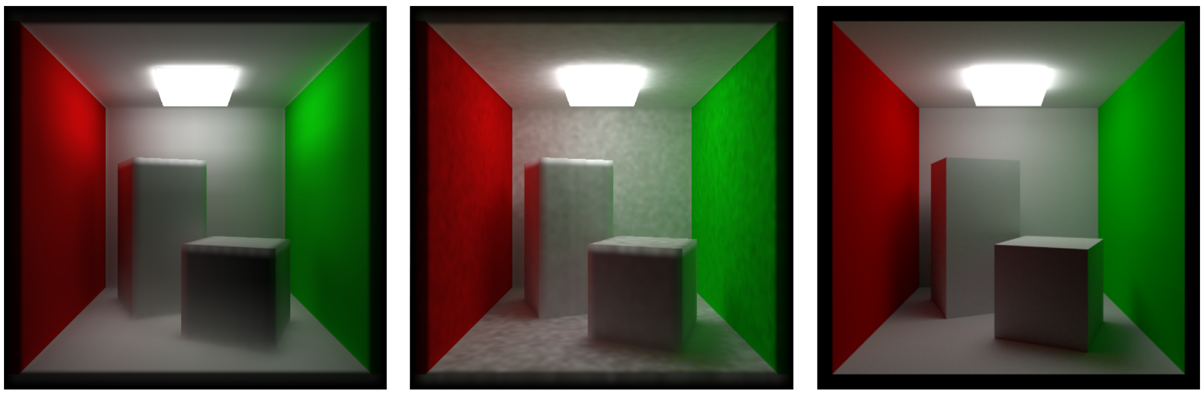


Figure 4.14: *Left to right: VCT result with adjusted normalization, voxel ray tracing test rendering without adjustments, path traced offline rendering used as reference to empirically adjust the normalization of the VCT rendering. The slow ray based test rendering exhibits a similar brightness as the reference. Shadows missing in the VCT result are apparent in the ray based rendering due to the higher integration quality. No tonemapping is used for the comparison.*

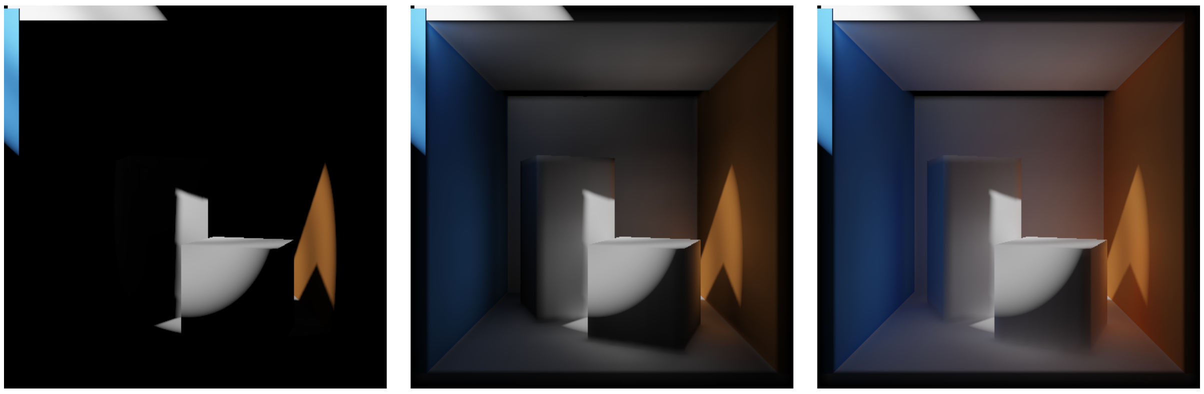


Figure 4.15: *Comparison of different numbers of light bounces. Left to right: only direct lighting (one bounce), two bounces (one direct, one indirect), many progressively built bounces. The right most result exhibits the most natural appearance.*

4.5 Dynamic updates

Scene samples that dynamically change their positions or other attributes are inserted into the same SVO-buffer and brick pool as the static ones on a frame-wise basis. Doing so prevents the necessity to sample twice from two separate structures for every query which would significantly increase the cost inflicted by the voxel cone tracing. While the basic construction algorithm as described in the *Construction* section remains the same, some occurring issues require to be handled specifically. For one, since the dynamic scene samples can potentially write to already existing nodes that contain static information, a backup of the affected geometry bricks needs to be made. Another problem arises in consequence of the non-deterministic allocation of bricks which requires the radiance bricks to be zeroed at the end of each frame. As a result, during the light propagation phase dynamic objects act as diffusers only for the light injected in the current frame. The simulation of multiple light bounces at dynamic leaf voxels, therefore, requires their reflected radiance to be extracted at the end of each frame before being zeroed and re-injected in the next frame during the radiance injection stage. This process of extracting radiance from the brick pool into a linear backup buffer at every dynamic scene sample works inversely to inserting geometry or radiance information into the pool. At the end of the frame, all information added by dynamic scene samples is removed from the SVO-buffer, the geometry backup is loaded into the geometry pool and radiance bricks affected by dynamic samples are set to zero. Doing so recovers the state of the SVO as it was before the dynamic update.

4.6 Final rendering

The final rendering of the scene uses deferred shading [ST90] to prevent an overhead due to shading computations performed for occluded fragments and to reduce the amount of shader permutations. Contrary to *forward shading*, deferred shading does not perform the fragment-wise shading computations directly after the fragments are generated by the rasterizer, i.e., when the scene geometry is rendered. Instead, for every fragment a number of attributes that are necessary for the shading computations are written to a *G-buffer*. In general, these attributes vary depending on the particular use case. Here they include an albedo, a linear depth from which the world position of the fragment can be reconstructed, a NDF, and an emitted radiance. A second pass then computes the actual shading for every screen pixel based on the attributes in the G-buffer, the light sources, and the reflected radiance stored in the leaf voxels of the SVO. During this shading pass, first the analytical lighting from the light sources is evaluated. For area lights this boils down to adding the emitted radiance stored in the G-buffer to the flux of every screen pixel. Point lights, in contrast, add their direct lighting contribution which is evaluated as described in the *Direct illumination* section. Then, the reflected radiance stored in the leaf voxels of the SVO is trilinearly or tricubically sampled at the fragment positions and likewise added to the flux. Additionally, it is possible to compute a bounce of glossy reflections from previously diffusely propagated light by sending a single cone starting from the fragment's position along the reflection vector into the SVO (see figure 4.16). To cheaply simulate the integration over time which converts the flux of each screen pixel to energy, the flux is multiplied with an empirical exposure duration. The resulting energy is at last *tonemapped* and *gamma-encoded* [TFCRS11] to map more values into a displayable range and to account for the conversion from RGB space to sRGB space, respectively.

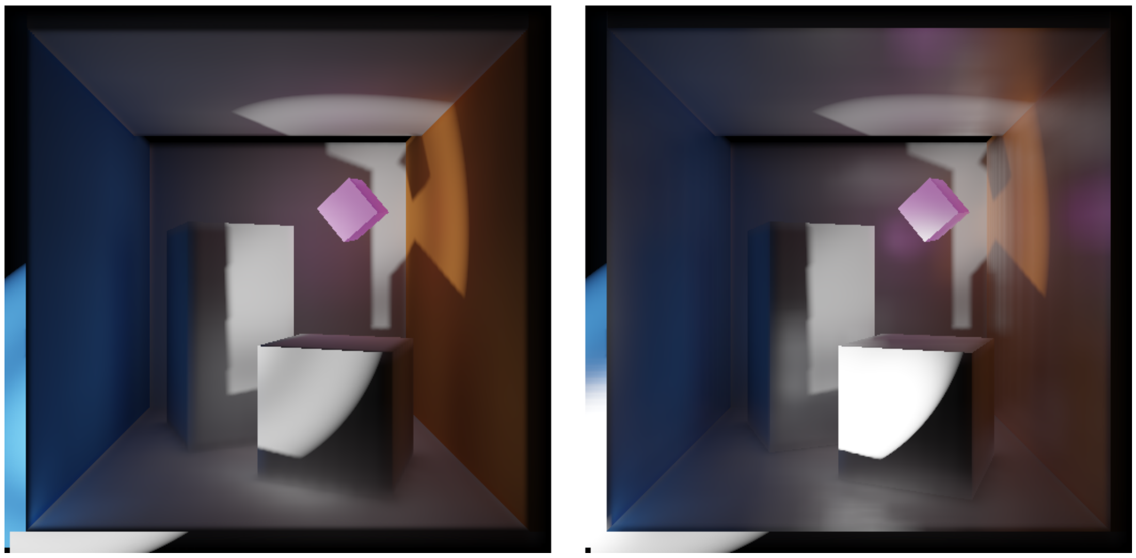


Figure 4.16: *Comparison between the default rendering mode which exclusively produces diffuse reflections (left) and an additional glossy bounce (right) based on the outgoing radiance stored in the mipmap.*

5 Implementation

The implementation of the algorithm detailed in the previous chapter is presented in the following. First, a short overview of the user level features of the application is given, followed by a description of the underlying software design and the presentation of the profiling results.

5.1 Application overview

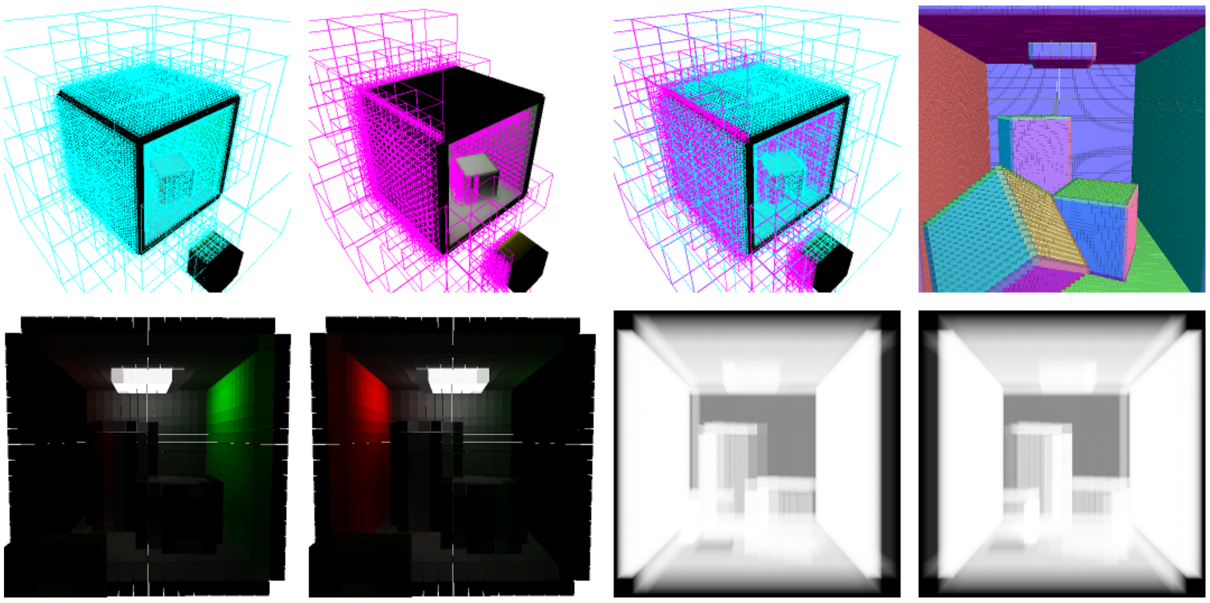


Figure 5.1: *Various visualizations for debugging purposes. Top row, left to right: octree structure, border tiles, octree and border tiles, scene sample RoIs and NDFs. Bottom row, left to right: radiance of a selected mipmap level for -X and +X, opacity of the same level for -X and +X.*

The application provides three different scenes to chose from: CornellBox, CornSnailBox, and CaveCloud. CornellBox is a Cornell box-like [CBo] scene featuring a white area light and a yellow cube that can be moved and rotated by the user. The CornSnailBox is a similar scene but instead of an area light it uses a spotlight of adjustable orientation as primary light source. Unlike the Cornell box it has an opening in the ceiling that allows the spotlight to illuminate the scene from the upper back. It additionally contains a small dim area light that can be moved. The CaveCloud scene features a point cloud (about

1.4 million points) of a portion of a real cave. The cave is illuminated by two moving spotlights and two area light sources; one small and static, the other large and moveable (see figure 5.2).

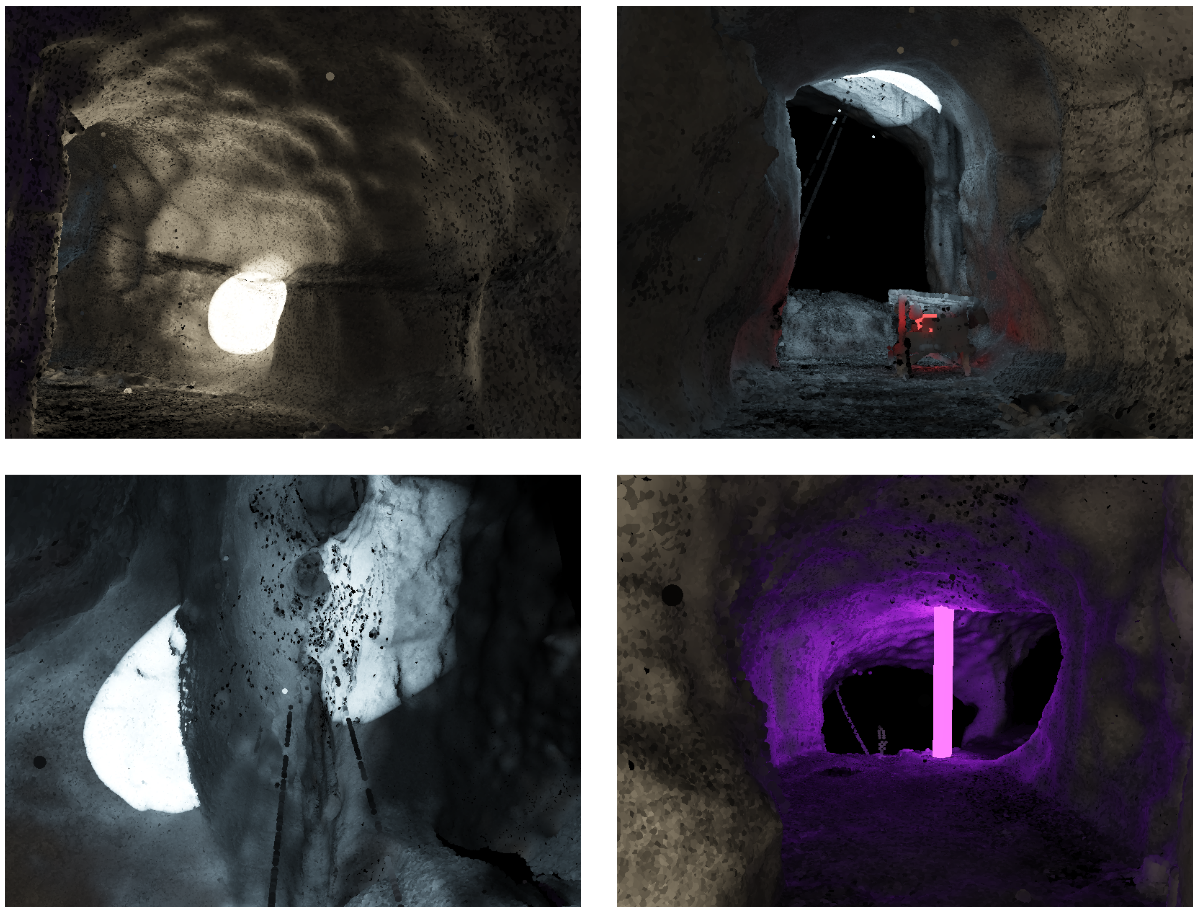


Figure 5.2: *Results of the CaveCloud scene.*

Since the cave scene has relatively low albedo values, the implementation allows them to be blended towards white by linear interpolation for the light propagation simulation to better show off the indirect lighting (see figure 5.3). It is also possible to manually adjust the exposure duration to brighten up the result which in contrast to the albedo adjustment is physically plausible but less suitable to adjust the intensity falloff of the multiple light bounces. For comparison purposes, the radiance pool can be optionally cleared every frame to suppress the accumulation of multiple light bounces, resulting in a single bounce of indirect lighting for spotlights and direct lighting only for area light sources. Glossy reflections and tricubic filtering can also be toggled on and off. Furthermore, the application provides various options to visualize different parts of the SVO and the underlying scene sampling (see figure 5.1). Depending on the scene, the observer either orbits around the center (CornellBox and CornSnailBox) or can move freely (CaveCloud).

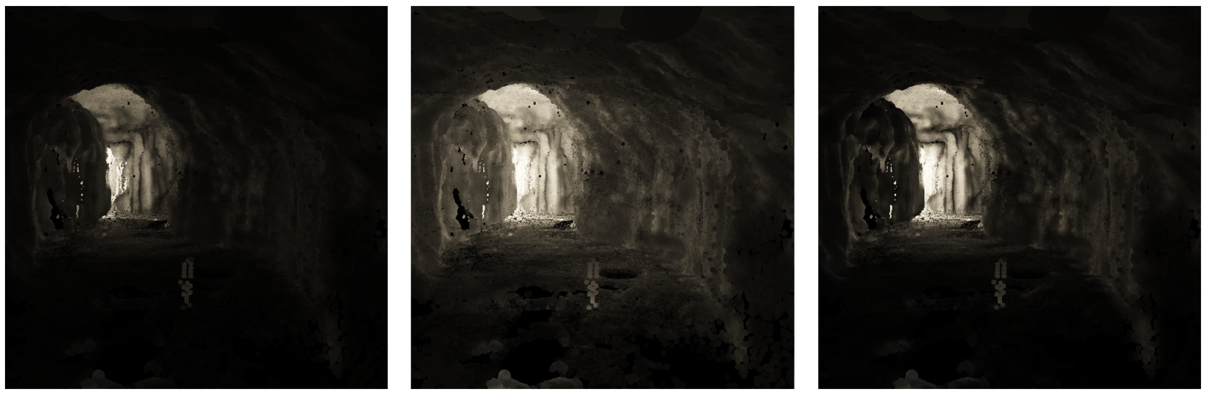


Figure 5.3: *Albedo adjustment. Left to right: no adjustment, with albedo adjustment of 0.33, with adjustment but only a single bounce (note how little light reaches the floor in the foreground).*

5.2 Software overview

This section provides a short overview of the general structure of the implementation and its most important classes and procedures.

The application is written in C$^\sharp$ and HLSL using Direct3D 11 as graphics API. The D3D API is accessed through SharpDX [Sha], a DirectX binding for C$^\sharp$. Additionally, the Coon framework is used as a base for the project. Coon works on top of SharpDX and simplifies resource creation, provides commonly used functionality like parametrized view matrices and streamlined shader (re-)compilation, and manages the main loop of the application. The main loop is realized in the static `Coon.MainLoop` class and regularly calls the delegates `MainLoop.UpdateEvent()` and `MainLoop.RenderEvent()` which can be set by the application using statement lambdas [Lam]. `UpdateEvent()` contains the program logic like camera and object movement. `RenderEvent()`, on the other side, handles the graphic API calls. This separation is motivated by Coon's use of fixed time steps for integrations involved in the program logic. All classes and functionality described in the following live in the namespace `GIVoxels` outside of the Coon framework. The method Run() in the static class `GIVoxelsMain` provides the entry point of the program. It first initializes `Coon.Core`, creates necessary resources, sets Coon's `UpdateEvent()` and `RenderEvent()` to statement lambda that contain the appropriate functionality and eventually calls `Core.Run()` to start the main loop.

The functionality that realizes the SVO based global illumination simulation is encapsulated in the `GI_SVO` class. Its constructor compiles the necessary shaders if they do not exist yet and allocates the SVO-buffer and helper buffers which, among others, provide required queues and structures that simplify the access to nodes of specific tree levels. The functionality of the most important methods of `GI_SVO` is outlined in the following.

`BuildStaticPart()` is called to (re-)initialize the SVO base structure by building it from the static scene samples.

`BuildDynamicPart()` is the first method to be called every frame. It updates the static structure by inserting the dynamic scene samples and copies the affected geometry bricks to the backup section of the brick pool. The radiance re-injection of previously extracted radiance from dynamic voxels is also handled here.

`EvalFrameWiseComputations()` first handles the light injection, then builds the mipmap and finally simulates the light propagation by cone tracing (or by ray tracing for testing purposed).

`InjectDirectLightForGI()` and **`InjectDirectLightForRefGI()`** are virtual methods which are called by `EvalFrameWiseComputations()` to handle the injection of direct light from point lights for the default rendering and the ray tracing based test rendering, respectively. These methods are implemented in the derived `GIV_SVO` class.

`RemoveDynamicPart()` is called at the end of the frame after the final rendering which is handled outside the SVO class. It extracts the reflected radiance stored at dynamically affected leaf voxels and reverses all changes to the SVO induced by `BuildDynamicPart()`.

These methods are called in the appropriate order in `RenderEvent()`.
After the call to `EvalFrameWiseComputations()`, the scene geometry is rasterized in a geometry pass that fills the G-buffer. The G-buffer is then used to render the final image to the back buffer during the deferred shading pass.

Scenes are represented by classes that inherit from `SceneBase`. `SceneBase` is an abstract class that consists of a `GIV_SVO` instance, lights and lists of objects that store scene samples. A class that encapsulates a scene sampling must implement one of four available interfaces. The list in which its instances are stored depend on the interface the class implements. These interfaces are: `IGIVoxelsStaticObj`, `IGIVoxelsStaticEmiObj`, `IGIVoxelsDynamicObj`, and `IGIVoxelsDynamicEmiObj`. Which one of these a scene sampling class implements depends on whether its sampling is static or dynamic and whether its samples emit light on their own or not. The `GIV_SVO` instance is constructed from these four lists and various settings like position, tree depth and render mode (GI/RefGI).

The implementation currently provides two scene sampling classes, `PointCloud` and `CubeBase`. During the construction of a `PointCloud` instance, a '.cloud'-file is loaded which contains the scene sample positions, NDFs, and albedos. The binary cloud format follows a simple structure. The first value in the file is the count of samples (32 bit integer). Thereafter follow the scene sample attributes in array-of-structures fashion: position (3 x 1 single precision float), NDF (3 x 1 single precision float), albedo in sRGB (3 x 8 bit unsigned integer). Positions and NDFs reside in a right-handed coordinate system with the y-axis pointing upwards. The RoIs are hard coded to a single empirical value inside the class. The CubeBase class, on the other hand, provides a regular sampling of the surface of a cuboid. The RoIs are derived from the adjustable sampling density to create a continuous surface representation with overlaps as small as possible.

The only point light source type supported at the moment are spotlights which are implemented in the `SpotLights` class and its nested `Light` class. Spotlights were chosen over omni-directional point lights for being faster, easier to implement, and because they are more suitable to show off the indirect lighting due to the focused nature of their direct lighting. Grouping all spotlights together in a single class makes it easier to organize the rendering of their shadow maps to two shared texture arrays which in turn allow to sample the VSM and ESM of a particular light by an id when evaluating its direct lighting in a shader. The shadow maps are rendered at the beginning of the frame before `BuildDynamicPart()` of the SVO object is called.

Generic buffers like the SVO-buffer, the helper buffers and the scene sample attribute buffers are accessed by unordered access views for read/write purposes in the shaders and by shader resource views for read-only access. Some buffers like the SVO-buffer and the queue buffers have also a hidden atomic counter attached to them that are used to manage allocations. Textures like the brick pools and the G-buffer are accessed by shader resource views for reading purposes and are exclusively written by the render output unit of the graphics pipeline.

5.2.1 Resource formats and sizes

The following list gives an overview of the formats and sizes of important resources. The formats are specified in a per-element fashion.

Scene sample position array: *4 x 1 single precision float*; the element count corresponds to the scene sample count. Layout per element: position X, position Y, position Z, RoI.

Scene sample NDF array: *4 x 1 single precision float*; the element count corresponds to the scene sample count. Layout per element: NDF X, NDF Y, NDF Z, unused.

Scene sample albedo array: *4 x 1 single precision float*; the element count corresponds to the scene sample count. Layout per element: albedo R, albedo G, albedo B, opacity.

Scene sample radiance array: *4 x 1 single precision float*; the element count corresponds to the scene sample count. Layout per element: radiance R, radiance G, radiance B, opacity. The redundant opacity value prevents the need to additionally read from the albedo array during the light injection stage. In practice, the application does not implement a scene sampling class that makes use of an extra radiance buffer but instead injects the same radiance for every sample by using an appropriate shader.

SVO-buffer and helper buffers: *1 x 1 32 bit unsigned integer* (re-interpreted in the shader); the element counts are estimated heuristically from three manually specified values (maxStaticLeafNodeCount, maxDynamicLeafNodeCount, fragmentQueueCapacity).

Brick pools: *4 x 1 half precision float*; their sizes are derived from the estimated maximal brick count which is itself derived from the manually specified values. The size in each individual dimension (X, Y, and Z) is a power of two and for each of the four pools the same.

Shadow maps: *2 x 1* and *1 x 1 single precision float* for VSMs and ESMs, respectively; their sizes must match for all lights and has to be the same in both dimensions. They are hard coded in `SceneBase` to 256x256.

G-buffer: albedo: *4 x 8 bit unsigned normalized integer (sRGB)*, linear depth: *1 x 1 single precision float*, NDF: *4 x 1 half precision float*, emitted radiance: *4 x 8 bit unsigned normalized integer*; their sizes correspond to the framebuffer resolution which in turn equals the output window or screen resolution for windowed and full screen mode, respectively. The RGB value of the emitted radiance is actually split into a 'direction' part and a 'length' part by division by the largest element. The 'length' scalar is stored in the alpha channel of the NDF buffer while the 'direction' is kept in the RGB-channels of the radiance buffer. In the deferred shading pass the original value is then recovered by multiplying the direction component with the length. This prevents the need of another large floating point format for the radiance buffer and utilizes the otherwise unused alpha channel of the NDF buffer.

5.3 Profiling results

In this section, the outcomes of the profiling are presented. The discussion and evaluation of this results follow in the subsequent chapter.

The tables 5.1, 5.2, 5.3, and 5.4 show the profiling results of the CornellBox, CornSnailBox and CaveCloud scenes. The upper part contains render times in milliseconds for various sections of the render process. These times are collected by averaging the measurements of 1000 rendered frames from static camera perspectives (see figure 5.4). The lower part shows the allocated memory of important resources in mega bytes. For every scene, all values are collected for two different framebuffer resolutions (800x600 and 1600x900) and two SVO tree depths (7 and 8). Tree depth 7 is considered to be the default depth for the CornellBox and CornSnailBox scenes while the CaveCloud scene uses a depth of 8 per default (these depth values are used throughout this work for all images). In all cases, four cones

out res.	800x600		1600x900	
SVO depth	7	8	7	8
net	11.4	50.8	12.0	51.8
static	20.4	89.7	20.4	91.1
dynamic	1.2	3.1	1.1	3.2
inj./mip	2.6	16.1	2.7	16.2
VCT	5.7	25.2	5.7	25.4
l.prop	9.7	47.2	9.8	47.5
shadows	-	-	-	-
g-pass	0.2	0.2	0.5	0.5
ds	0.2	0.4	0.5	0.7
final	0.5	0.6	1.0	1.2
ds(c)	2.7	3.4	5.4	6.5
ds(g)	7.6	9.5	15.7	19.3
ds(c/g)	10.8	11.9	21.9	23.5
SVO-buffer	2.1	8.7	2.1	8.7
SVO helpers	2.2	8.6	2.2	8.6
brick pools	64.0	256.0	64.0	256.0
SVO	68.3	273.2	68.3	273.2
G-buffer	11.0	11.0	33.0	33.0

Table 5.1: *CornellBox profiling results.*

out res.	800x600		1600x900	
SVO depth	7	8	7	8
net	10.9	47.4	11.4	47.9
static	19.5	85.2	19.3	84.8
dynamic	0.8	1.4	0.8	1.4
inj./mip	2.7	16.0	2.7	16.0
VCT	5.4	23.6	5.3	23.5
l.prop	9.5	45.2	9.4	45.1
shadows	0.1	0.1	0.1	0.1
g-pass	0.2	0.2	0.4	0.4
ds	0.3	0.4	0.7	0.8
final	0.5	0.6	1.1	1.3
ds(c)	2.9	3.7	5.8	6.9
ds(g)	7.8	9.7	15.9	19.6
ds(c/g)	11.1	13.9	22.6	27.9
SVO-buffer	2.1	8.7	2.1	8.7
SVO helpers	2.2	8.6	2.2	8.6
brick pools	64.0	256.0	64.0	256.0
SVO	68.0	273.2	68.3	273.2
G-buffer	11.0	11.0	33.0	33.0

Table 5.2: *CornSnailBox profiling results.*

are used in the VCT pass. All cuboids are rasterized as triangles; only the point cloud in the CaveCloud scene is rendered with surfels. The tests were performed on a Intel Core2 Quad Q9550 (2.83 GHz) and a NVIDIA GeForce GTX 560 Ti. The following list explains the meaning of the individual values.

net: render time of the complete frame with trilinear sampling in the deferred shading pass.

static: time required for the construction of the static base structure of the SVO. This value is not averaged over 1000 render frames but instead is the average of 10 construction times. It is also not contained in the **net** value.

dynamic: time required for the dynamic update of the SVO including the reversal of the induced changes at the end of the frame.

out res.	800x600		1600x900	
SVO depth	7	8	7	8
net	30.6	41.8	44.7	55.9
static	158.8	248.9	158.4	249.4
dynamic	0.6	0.7	0.6	0.7
inj./mip	1.4	5.0	1.4	5.1
VCT	1.7	7.8	1.7	7.8
l.prop	3.5	14.5	3.5	14.5
shadows	7.8	7.9	7.8	7.8
g-pass	18.2	18.3	31.5	31.6
ds	0.5	0.5	1.3	1.3
final	18.7	18.7	32.8	32.9
ds(c)	3.1	3.6	8.7	9.9
ds(g)	-	-	-	-
ds(c/g)	-	-	-	-
SVO-buffer	1.8	2.9	1.8	2.9
SVO helpers	1.8	3.0	1.8	3.0
brick pools	64.0	128.0	64.0	128.0
SVO	67.6	134.0	67.6	134.0
G-buffer	11.0	11.0	33.0	33.0

Table 5.3: *CaveCloud(1) profiling results.*

out res.	800x600		1600x900	
SVO depth	7	8	7	8
net	32.9	45.3	50.7	62.0
static	158.7	248.0	158.0	248.8
dynamic	0.6	0.7	0.6	0.7
inj./mip	1.4	5.0	1.4	5.0
VCT	1.7	7.8	1.7	7.8
l.prop	3.5	14.8	3.5	14.5
shadows	7.8	7.8	7.8	7.8
g-pass	20.5	21.8	37.6	37.6
ds	0.5	0.5	1.2	1.3
final	21.0	22.2	38.8	38.9
ds(c)	3.1	3.6	9.0	10.2
ds(g)	-	-	-	-
ds(c/g)	-	-	-	-
SVO-buffer	1.8	2.9	1.8	2.9
SVO helpers	1.8	3.0	1.8	3.0
brick pools	64.0	128.0	64.0	128.0
SVO	67.6	133.9	67.6	133.9
G-buffer	11.0	11.0	33.0	33.0

Table 5.4: *CaveCloud(2) profiling results.*

inj./mip: time requirement for the light injection and the subsequent mipmap construction.

VCT: runtime of the cone tracing pass.

l.prop: the simulation time of the complete light propagation process including **inj./mip**, **VCT**, and the subsequent completion of the radiance bricks in preparation of the deferred shading pass.

shadows: runtime of the pass that renders the shadow maps.

g-pass: render time of the geometry pass which fills the G-buffer in preparation of the deferred shading pass.

ds: render time of the deferred shading pass which writes the final image to the back buffer. Trilinear sampling is used to sample the radiance.

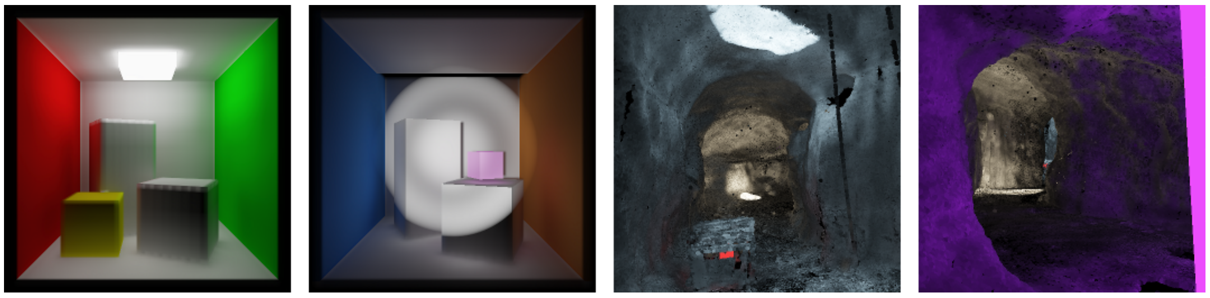


Figure 5.4: *The scenes and perspectives from which the profiling data was collected. From left to right: CornellBox, CornSnailBox, CaveCloud(1), CaveCloud(2).*

final: runtime of the final rendering (sum of **g-pass** and **ds**).

ds(c): render time of the deferred shading pass when using tricubic sampling.

ds(g): render time of the deferred shading pass when using trilinear sampling and glossy reflections. Not collected for the CaveCloud scene due to induced artifacts and generally little benefit.

ds(c/g): render time of the deferred shading pass when using tricubic sampling and glossy reflections. Not collected for the CaveCloud scene.

SVO-buffer: memory consumption of the SVO-buffer.

SVO helpers: memory consumption of the SVO helper buffers.

brick pools: memory consumption of the brick pools.

SVO: memory consumption of the whole SVO (sum of **SVO-buffer**, **SVO helpers** and **brick pools**).

G-buffer: memory consumption of the G-buffer.

6 Discussion

In the following, the results of the implementation are evaluated with respect to performance and quality. A summary of the implemented features concludes the chapter.

The profiling results presented in the previous chapter show various correlations between runtimes and framebuffer resolution and between runtimes and chosen SVO depth. As to be expected, the light propagation simulation runtime is strongly dependent on the tree depth while being unaffected by the framebuffer resolution. It is furthermore dominated by the VCT pass. The estimation of a concrete complexity would require more test scenes. By tricubically sampling the reflected radiance in the deferred shading pass instead of trilinearly, a substantial runtime dependency from both, output resolution, as well as SVO depth can be observed. This dependency becomes even more severe when using glossy reflections due to the involved cone tracing for every fragment. The chosen resolution is an especially critical factor for the rendering of surfels which makes up for a large part of the rendering costs in the CaveCloud scene. This costs can vary depending on the observer position in the scene even when the cases appear to be similar (compare **g-pass** of 5.3 and 5.4). It can be assumed that these differences are induced by the order in which the surfels are rendered, leading to more occluded fragments being generated in some cases than in others. The relatively high GPU memory consumption of the approach is in all presented cases heavily dominated by the brick pools. For more complex applications, the memory requirements of the SVO might be too high, even though scalar radiance values are used instead of distributions like in the original approach of [CNS$^+$11].

When comparing the quality of the VCT based approach to high quality renderings, two types of light leaking artifacts become apparent: light leaking at edges and light leaking through walls (see figure 6.1). The edge light leaking is introduced by directly sampling the reflected radiance from the radiance pool instead of performing a final gathering at every fragment as [CNS$^+$11] did. However, per-fragment VCT is very costly as can be seen in the case of glossy reflections. A possible solution could be to evaluate a distribution over incoming radiance in addition to the reflected radiance in the VCT pass. By keeping the radiance distribution monochromatic, it could fit in a three component vector like the NDF. In the deferred shading pass, the NDF from the G-buffer in conjunction with this distribution would then be used to evaluate the monochromatic amount of reflected radiance at each fragment while accounting for

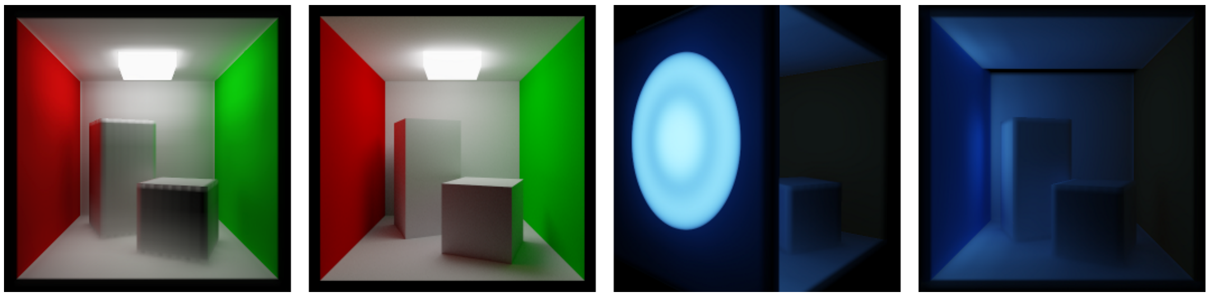


Figure 6.1: *Light leaking aritfacts. CornellBox: left, the VCT result, on the right, the reference rendering. Light leaking along the edges of the cubes can be seen in the left image as well as missing shadows. Direct sampling of the area light by an additional cone as suggested by [Cra12] and [Kas13] could resolve the problem of the missing shadows. CornSnailBox: light from the spotlight leaks through the wall into the box.*

the coloring by the reflected radiance from the brick pool. Care must be taken to correctly handle the redundant information contained in both, the monochromatic distribution of incoming radiance and the colored scalar of reflected radiance. The distribution could be stored in the second half of the radiance buffer where up to now the geometry backup resides which then would have to be moved to a dedicated buffer. Doing so would also better utilize the capacity of the radiance buffer since the geometry backup usually requires only a small portion of the available memory. However, the wall light leaking appears to be the more critical artifact. It might be intensified by the simplifications introduced compared to the original approach by [CNS$^+$11] which used more expressive lighting information in the mipmap. As can be seen in figure 4.14, the ray based approach captures shadows better which indicates that using more and smaller cones in the VCT pass could reduce the light leaking significantly. Unfortunately, since four cones already exceed the budget most real-time applications might allocate for global illumination, this is not a feasible approach.

The mediocre performance of the VCT pass could be improved by additionally sorting the static base structure of the SVO to improve data coherency for the SVO-buffer and the brick pools. Since albedos and NDFs are usually sampled together, it is probably also beneficial to store them interleaved instead of in separate sections of the geometry pool. Furthermore, it might be feasible to evaluate the reflected radiance in an interleaved fashion where, for example, only two cones are used at every leaf voxel. The result could then be filtered by averaging the radiance of adjacent voxels to virtually increase the amount of utilized cones.

The aliasing induced by moving geometry displays another problem and is not resolved for all situations by the antialiasing scheme. Flat surfaces that are perpendicular aligned to the main axes of the SVO are particularly problematic. By moving along the respective axis, the contribution of such a surface to individual leaf voxel changes abruptly, which in turn leads to frequent changes in the light propagation result. This problem could be resolved by avoiding critical orientations of problematic geometry or by modifying the surface sampling to provide additional gradients in the opacity values where necessary. This artificial opacity could on the down side lead to more self occlusion in the VCT pass and consequently to darker results.

The currently missing overflow handling for the SVO-buffer and helper buffers is also problematic. Overflows due to wrong buffer size estimations can result in permanent corruptions that propagate through the whole SVO structure including the brick pools. Therefore, overflow handling can be considered a mandatory feature for real-world application of the algorithm.

The current implementation of glossy reflections is rather incomplete. One the one side, hard coded smoothness and normal incident reflection settings are used to show off the effect and on the other side the albedo values in the SVO are not adjusted properly to account for the amount of light that is not reflected in a diffuse way and therefore not captured by the simulation. This adjustment could be implemented by simply reducing the albedos appropriately, effectively turning pure metals during the light propagation black. To still account for glossy reflections in a very approximate way, the albedo could be instead modified so that the estimated diffusely reflected light at each leaf voxel also contains the total amount of glossy reflected light.

6.1 Feature set summary

The implementation realizes multiple bounces of ideal diffuse light reflections which are particular beneficial to illuminate environments where direct lighting alone reaches only a small portion of the scene. In addition, a single bounce of glossy reflections is implemented prototypically. Area light sources are naturally handled by the simulation. However, point lights achieve a higher quality since their direct illumination is evaluated analytically and therefore does not rely on the SVO based simulation which is prone to light leaking artifacts.

Ambient occlusion [Fer04] is often employed as affordable approximation of the soft shadowing effects originating from indirect lighting in diffusely reflecting environments (see figure 2.4) or from the light of the sky in outdoor scenarios. These effects are implicitly handled by the presented approach.

The scenes are assumed to remain largely static due to the high costs induced by the SVO construction. Dynamic geometry is added by frame-wise, temporary updates of the static structure.

7 Conclusion

This work presented a SVO based interactive global illumination approach derived from the work of [CNS^{+}11]. To reduce memory consumption and potentially improve the runtime of the algorithm, a less puristic light propagation approach was proposed. The solution focuses primarily on the simulation of diffuse light propagation and handles multiple bounces. A single bounce of glossy reflections can be added at high performance costs. The SVO is directly built from a set of points which represents a regular or irregular sampling of the displayed scene. The approach, therefore, maps well to the visualization of point clouds, which was a primary objective of this thesis. The objective of reproducing the effects of multiple light reflections to improve the realism of interactively rendered scenes was met as well.

Nevertheless, despite the introduced simplifications, the algorithm remains expensive with regard to performance and memory consumption. Furthermore, the complexity introduced by the SVO leads to a much higher implementation effort compared to techniques that work on dense grids. For completely dynamic GI, LPVs and VCT on nested dense grids, therefore, potentially provide a better cost-benefit ratio of development effort and GI performance at the moment. However, if dynamic GI with consistent quality over a large scale is required, the use of a sparse structure can probably not be avoided. Besides, SVO-based scene representations provide additional possible applications beyond full GI, for example, for depth-of-field effects, antialiasing, soft shadows, ambient occlusion, volumetric effects, and even collision detection or sound propagation simulations, making the expensive maintenance of the SVO more worthwhile [Cra12].

Bibliography

[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

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

[Bur12] Brent Burley. Physically-based shading at disney. `https://disney-animation.s3.amazonaws.com/library/s2012_pbs_disney_brdf_notes_v2.pdf`, 2012.

[CBo] The cornell box. `http://www.graphics.cornell.edu/online/box/`.

[CNS$^+$11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing, sep 2011.

[Cra11] Cyril Crassin. Gigavoxels: A voxel-based rendering pipeline for efficient exploration of large and detailed scenes, July 2011. English and web-optimized version.

[Cra12] Cyril Crassin. In *SIGGRAPH 2012 Course : Beyond Programmable Shading*. ACM SIGGRAPH, 2012.

[CUD] Cuda parallel computing platform. `http://www.nvidia.com/object/cuda_home_new.html`.

[D3Da] Hardware support for direct3d 11 formats. `https://msdn.microsoft.com/en-us/library/windows/desktop/ff471325(v=vs.85).aspx`.

[D3Db] Resource limits (direct3d 11). `https://msdn.microsoft.com/en-us/library/windows/desktop/ff819065(v=vs.85).aspx`.

[DBB06] P. Dutre, P. Bekaert, and K. Bala. *Advanced Global Illumination, Second Edition*. Ak Peters

Series. Taylor & Francis, 2006.

[DL06] William Donnelly and Andrew Lauritzen. Variance shadow maps. In *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*, I3D '06, pages 161–165, New York, NY, USA, 2006. ACM.

[Epp10] David Eppstein. The z planefilling curve. `http://commons.wikimedia.org/wiki/File:Z-curve.svg`, 2010.

[Fer04] Randima Fernando. *GPU Gems: Programming Techniques, Tips and Tricks for Real-Time Graphics*. Pearson Higher Education, 2004.

[Fou92] Alain Fournier. Normal distribution functions and multiple surfaces. In *Graphics Interface '92 Workshop on Local Illumination*, pages 45–52, Vancouver, BC, Canada, 1992.

[ICG86] David S. Immel, Michael F. Cohen, and Donald P. Greenberg. A radiosity method for non-diffuse environments. In *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '86, pages 133–142, New York, NY, USA, 1986. ACM.

[Jen96] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the Eurographics Workshop on Rendering Techniques '96*, pages 21–30, London, UK, UK, 1996. Springer-Verlag.

[JMLH01] Henrik Wann Jensen, Stephen R. Marschner, Marc Levoy, and Pat Hanrahan. A practical model for subsurface light transport. In *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '01, pages 511–518, New York, NY, USA, 2001. ACM.

[Kaj86] James T. Kajiya. The rendering equation. In *Computer Graphics*, pages 143–150, 1986.

[Kar13] Brian Karis. Real shading in unreal engine 4. `http://blog.selfshadow.com/publications/s2013-shading-course/karis/s2013_pbs_epic_notes_v2.pdf`, 2013.

[Kas13] Nikolas Kasyan. Playing with real-time shadows. `http://www.crytek.com/cryengine/presentations&page=1`, 2013.

[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*, I3D '10, pages 99–107, New York, NY, USA, 2010. ACM.

[Kel97] Alexander Keller. Instant radiosity. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 49–56, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co.

[Lam] Lambda expressions (c sharp programming guide).
`https://msdn.microsoft.com/en-us/library/bb397687.aspx`.

[Lam92] Johann Heinrich Lambert. *Lamberts Photometrie*.
`https://archive.org/details/lambertsphotome00lambgoog`, 1760, 1892.

[Lau08] Andrew T. Lauritzen. *Rendering Antialiased Shadows using Warped Variance Shadow Maps*.
`http://www.punkuser.net/lvsm/lvsm.pdf`, 2008.

[LdR14] Sebastien Lagarde and Charles de Rousiers. Moving frostbite to pbr.
`http://blog.selfshadow.com/publications/s2014-shading-course/frostbite/s2014_pbs_frostbite_slides.pdf`, 2014.

[LSK+07] Samuli Laine, Hannu Saransaari, Janne Kontkanen, Jaakko Lehtinen, and Timo Aila. Incremental instant radiosity for real-time indirect illumination. In *Proceedings of Eurographics Symposium on Rendering 2007*, pages 277–286. Eurographics Association, 2007.

[Max95] Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

[McL14] James McLaren. Cascaded voxel cone tracing in the tomorrow children.
`http://fumufumu.q-games.com/archives/2014_09.php`, 2014.

[Mor66] G.M. Morton. *A Computer Oriented Geodetic Data Base and a New Technique in File Sequencing*. International Business Machines Company, 1966.

[Nic65] Fred E. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Appl. Opt.*, 4(7):767–775, Jul 1965.

[OGL] Glsl atomicadd.
`https://www.opengl.org/sdk/docs/man/html/atomicAdd.xhtml`.

[Ope] Opencl the open standard for parallel programming of heterogeneous systems.
`https://www.khronos.org/opencl/`.

[PD84] Thomas Porter and Tom Duff. Compositing digital images. In *Proceedings of the 11th*

Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '84, pages 253–259, New York, NY, USA, 1984. ACM.

[PH10] Matt Pharr and Greg Humphreys. *Physically Based Rendering, Second Edition: From Theory To Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2010.

[PJW12] Reinhold Preiner, Stefan Jeschke, and Michael Wimmer. Auto splats: Dynamic point cloud visualization on the gpu. In H. Childs and T. Kuhlen, editors, *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization*, pages 139–148. Eurographics Association 2012, May 2012.

[PW10] Reinhold Preiner and Michael Wimmer. Real-time global illumination for point cloud scenes. *Computer Graphics Geometry*, 12(1):2–16, 2010.

[RGK$^+$08] T. Ritschel, T. Grosch, M. H. Kim, H.-P. Seidel, C. Dachsbacher, and J. Kautz. Imperfect shadow maps for efficient computation of indirect illumination. *ACM Trans. Graph.*, 27(5):129:1–129:8, December 2008.

[RSC87] William T. Reeves, David H. Salesin, and Robert L. Cook. Rendering antialiased shadows with depth maps. *SIGGRAPH Comput. Graph.*, 21(4):283–291, August 1987.

[Sha] Sharpdx.
`http://sharpdx.org/`.

[Spe93] Stephen Spencer. Radiosity overview.
`http://www.siggraph.org/education/materials/HyperGraph/radiosity/overview_1.htm`, 1993.

[ST90] Takafumi Saito and Tokiichiro Takahashi. Comprehensible rendering of 3-d shapes. In *Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '90, pages 197–206, New York, NY, USA, 1990. ACM.

[TFCRS11] William Thompson, Roland Fleming, Sarah Creem-Regehr, and Jeanine Kelly Stefanucci. *Visual Perception from a Computer Graphics Perspective*. A. K. Peters, Ltd., Natick, MA, USA, 1st edition, 2011.

[Tok04] Michael Toksvig. Mipmapping normal maps.
`http://www.nvidia.com/object/mipmapping_normal_maps.html`, 2004.

[Wil78] Lance Williams. Casting curved shadows on curved surfaces. *SIGGRAPH Comput. Graph.*, 12(3):270–274, August 1978.