Interactive Dynamic Ambient Occlusion Using Approximate Voxel Cone Tracing

Master-Thesis von Dennis Basgier
Tag der Einreichung:

1. Gutachten: Prof. Dr. Jan Bender
2. Gutachten: Dipl.-Inform. Sebastian Lipponer

Interactive Dynamic Ambient Occlusion Using Approximate Voxel Cone Tracing

Vorgelegte Master-Thesis von Dennis Basgier

1. Gutachten: Prof. Dr. Jan Bender
2. Gutachten: Dipl.-Inform. Sebastian Lipponer

Tag der Einreichung:

Erklärung zur Master-Thesis

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den May 21, 2015

(Dennis Basgier)

Contents

1 Introduction

Photorealistic renderings, which means computer generated pictures indistinguishable from reality, was and is the primary objective in computer graphics. Whereas offline renderers are capable of producing near perfect images by using physically accurate lighting techniques, interactive application still struggle to catch up [Ritschel et al., 2012]. Most standard real-time solutions use local lighting models like Phong Shading which can only achieve visual plausibility not photorealism. Global illumination models drastically improve the quality of renderings. Unfortunately, they come at high computation costs especially if the scene content is dynamic and approaches millions of primitives. To reduce the overall expenses, the complete global lighting model is divided into several parts, each of which simulates a single phenomena of global illumination. One common effect is an approximate soft shadowing factor called ambient occlusion.

1.1 Motivation

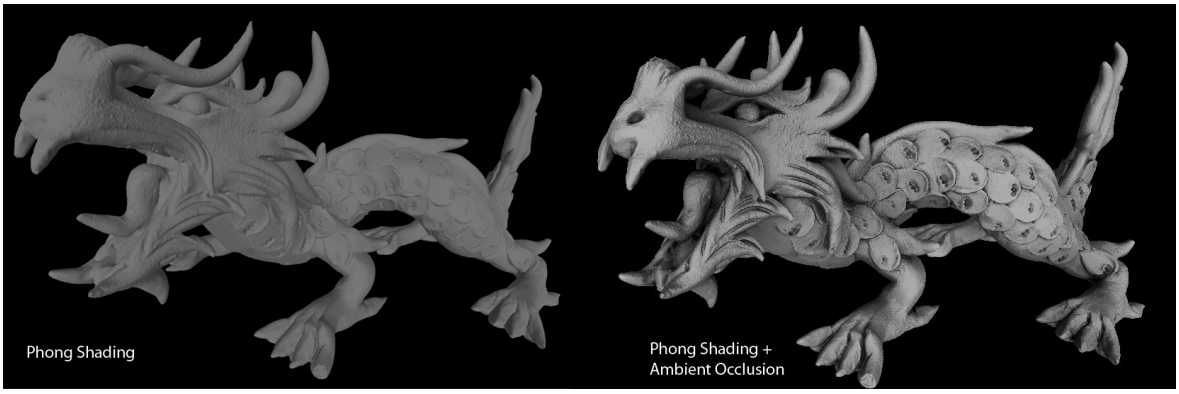


Figure 1.1: Comparison of a rendered mesh using two different types of shading.

Most local illumination models use an ambient light component which specifies a minimum brightness over the whole scene regardless of the spatial relationships between geometric objects. Without shadows, diffuse objects that are lit from many directions look flat and unrealistic (Figure 1.1 left). Especially in dynamic simulations, diffuse lighting models do not provide enough visual queues and details to identify certain effects like small fractions, cracks, wrinkles or deformations. Hence, a more sophisticated and flexible illumination approximation is necessary.

Ambient occlusion (AO) is a relatively fast way to obtain small scale shadows which help to visually indicate curvature and spatial relationships. It is a representation of how much

space of the hemisphere above a surface point is covered by geometry, hence how much light is occluded. The intensity of the shadows can be adjusted to achieve the desired effect. Although it is only an approximative measure, in combination with local illumination models ambient occlusion increases the visual fidelity of the virtual world, which leads to more realistic rendering results (Figure 1.1 right).

1.2 Ambient Occlusion

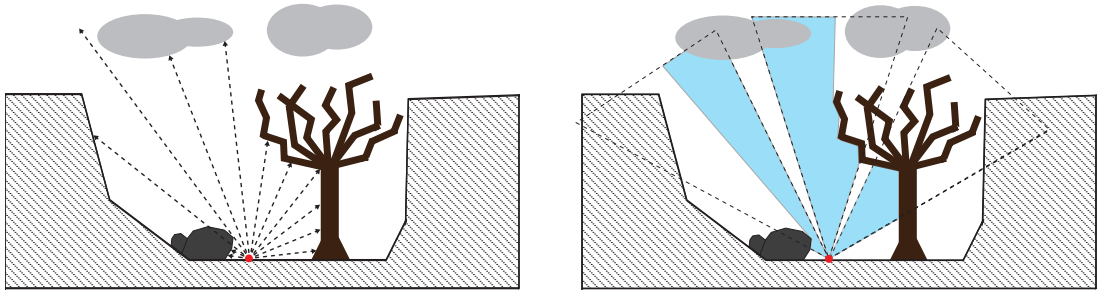


Figure 1.2: Illustration of gathering information for ambient occlusion for a surface point in a simple scene. **Left image:** Using rays to sample the scene. **Right image:** Approximating multiple coherent rays by a few cones.

The ambient occlusion estimate $A(\mathbf{p})$ for a surface point $\mathbf{p}$ is defined as the cosine-weighted fraction of the hemisphere Ω, where incoming light cannot reach the surface. [Zhukov et al., 1998] employed a simple definition which is given by

$$A(\mathbf{p}) = \frac{1}{\pi} \int_{\Omega} V(\mathbf{p}, \omega)(\cos(\omega)) d\omega \tag{1.1}$$

where $V(\mathbf{p}, \omega)$ is the visibility function which is zero if the incoming ray originating at $\mathbf{p}$ in the direction ω intersects the geometry in the scene or one if otherwise. This integral is usually solved by tracing rays in the domain of the hemisphere using Monte Carlo integration [Akenine-Möller et al., 2002]. A high amount of rays is necessary to get a smooth result, which is computationally expensive. Therefore, in this work a discrete definition is used which is given by

$$A(\mathbf{p}, \mathbf{n}) = \frac{1}{N} \sum_{i=1}^{N} V(\mathbf{p}, \omega_i(\mathbf{n}), f(r)) \tag{1.2}$$

Instead of testing a large number of spatially and directionally coherent rays to evaluate the integral, a small number (N) of visibility cones $V(\mathbf{p}, \omega_i(n))$ is used to exploit this coherence (Figure 1.2). Although this approximation is rough, it produces visually plausible results.

The cones originating at $\mathbf{p}$, which direction ω_i is in a hemisphere oriented towards the surface normal $\mathbf{n}$ are evaluated with the help of a technique called approximate voxel cone tracing [Crassin et al., 2011]. Since distant objects usually have lower impact on the shadowing of ambient light, the cone samples are weighted with a falloff function $f(r)$ which recedes with the distance r. The ambient occlusion term can be multiplied with the ambient light component of local illumination models (e.g. Phong-Shading) to generate diffuse small scale shadows (Figure 1.1).

1.3 Related work

To compute ambient occlusion, the hemisphere of any given surface point has to be scanned for other geometric objects which block incoming light. Today's standard rasterization based rendering techniques are not well suited to provide visibility queries between arbitrary points in world-space.

Ray tracing algorithms [Whitted, 1980, Williams, 1983] on the other hand, support arbitrary point-to-point visibility queries and are therefore well suited to scan the hemisphere of a point for occluders. To provide smooth rendering results, typically a large number of rays is necessary, which is computationally expensive. Consequently, ray-tracing techniques were limited to static images for many years. Since then, a lot of acceleration structures, like octrees [Sung, 1991, Crassin et al., 2009], kd-trees [Wald and Havran, 2006] or bounding volume hierarchies [Kay and Kajiya, 1986, Wächter and Keller, 2006] have been developed or improved to accelerate ray tracers, thus interactive ray tracing became available for static scenes [Dmitriev et al., 2004, Reshetov et al., 2005, Overbeck et al., 2007]. However, only a few techniques allow ray tracing in full dynamic scenes [Guntury and Narayanan, 2012, Zirr et al., 2013, Wald, 2004, Ritschel et al., 2012], either by applying complex load balancing schemes on the GPU or by restricting ray tracing only to complex scene section that require it.

[Mittring, 2007] presented screen-space ambient occlusion (SSAO), an alternative approach to gather the visibility information. It allows interactive rendering of ambient occlusion for fully dynamic scenes. The basic idea is to use the depth buffer information to estimate the spatial relationships and occlusions. This simple and fast method quickly became accepted by the industry. Many follow up techniques have been employed to improve the quality of screen-space AO. A brief overview is provided in [Ritschel et al., 2012]. Unfortunately the quality is still inferior to non-local techniques due to systematic errors. On the one hand, these errors are caused by the insufficient 3D information provided by the depth buffer. The problem is that particular cases appear similar in the depth but should actually produce different results. On the other hand the depth buffer is measured from the camera so SSAO is view-dependent and can change under different camera angles.

1.4 Ambient Occlusion using Approximate Voxel Cone Tracing

This work revisits approximate voxel cone tracing, which is a technique introduced by [Crassin et al., 2011]. It supports real-time frame rates in semi dynamic scenes, which means, that geometric objects in the scene are previously classified as static or dynamic. The example scenes used by Crassin et al. usually feature a much larger amount of static than dynamic geometry. The goal of this work is to test and adapt the method for ambient occlusion in full dynamic scenes, without distinguishing between static or dynamic geometry.

Approximate voxel cone tracing exploits the spatial and directional coherence of rays by sending a drastically reduced number of cones instead of single rays into the scene. Each cone gathers the desired visibility information ($V(p, \omega_i(n))$) to compute the final ambient occlusion term by evaluating equation 1.2. The idea is to step along the cone axis and perform lookups in a precomputed hierarchical structure according to the current cone diameter. While sampling the cone the actual values are quadrilinear interpolated to ensure the best possible result and to prevent aliasing.

The approach can be summed up in three main steps:

1. Surface Voxelization
2. Sparse Voxel Octree construction
3. Approximate voxel cone tracing

The foundation of the approach is a voxel representation of the scene. Fast techniques are required to enable the revoxelization of full dynamic scenes in every frame and still support interactive frame rates. To increase performance during the cone tracing process, an appropriate acceleration data structure is necessary. This data structure must allow fast, highly parallel construction, memory efficient storage and fast traversal. The sparse voxel octree (SVO) after [Crassin et al., 2009] can meet the requirements and is also well suited to provide coarser scene approximations in the interior nodes.

Once the SVO is constructed, the cone tracing process is similar to a single ray tracing inside the acceleration structure, with the difference that the cone diameter at each sampling position determines the required level-of-detail.

1.5 Outline

The following chapter provides basic knowledge, techniques and important terms which are fundamental for the further progress of this work.

Chapter 3, 4 and 5 describe the key techniques of approximate voxel cone tracing which are surface voxelization, SVO construction and the tracing process. Each part provides background information about related work and general explanations of the used methods, followed by an insight about the implementation.

The applied concepts are evaluated in the *Results and Performance* chapter using different benchmark tests. Along with the analysis of image quality and computation time, the chapter demonstrates limitations of the approach.

The final chapter summarizes and concludes the thesis and provides possible enhancements of the approach.

2 Fundamentals

Type	Notation	Examples
Scalar	Lowercase letter	a, b, t, u_i, r_{ij}
Vector or point	Lowercase bold letter	$\mathbf{p}, \mathbf{v}, \mathbf{u}$
Angle	Lowercase Greek	α, γ, Θ
Matrix	Capital bold letter	$\mathbf{R}, \mathbf{M}, \mathbf{X}$
Triangle	$\triangle$3Points	$\triangle\mathbf{qpr}$
Geometric primitive	Mathematical calligraphic	$\mathscr{B}, \mathscr{T}, \mathscr{V}$
Cross product	$\times$	$\mathbf{u} \times \mathbf{v}$
Scalar product	$\bullet$	$\mathbf{u} \bullet \mathbf{v}$
Exterior product	$\perp$	$\mathbf{u} \perp \mathbf{v}$

Table 2.1: Summary of mathematical notation

2.1 Spaces and Coordinates

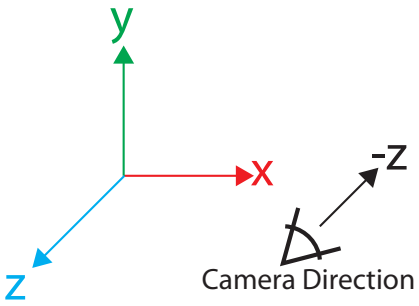


Figure 2.1: Coordinate system convention, including the camera direction in *OpenGL*.

To prevent misunderstandings about the conventions used in this work, this section shortly presents the required coordinate spaces. Since the approach was implemented with the *OpenGL* API, most of the conventions are inherited or adjusted.

World-space: The 3D space where the original geometric objects are located. The world is defined with a reference to an absolute origin. Furthermore the utilized coordinate system is right-handed and the orientation of the axes follows the *OpenGL* convention, as shown in Figure 2.1.

Screen-space: In screen-space the values have been translated into window range. The space is ranging from $(0,0)^T$ to $(I_x, I_y)^T$ where I_i defines the viewport resolution used for rendering. Each coordinate corresponds to a *pixel* on the image plane.

Voxel-space: The voxel-space is the discrete version of the world coordinate system. It ranges from $(0,0,0)^T$ to $(V_x, V_y, V_z)^T$ where V_i defines the voxel grid resolution. Each unit in voxel-space is further referred to as *voxel*.

2.2 2D Exterior Product

The exterior product (also known as outer product) of two vectors: $\mathbf{u} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$ and $\mathbf{v} = \begin{pmatrix} v_1 \\ v_2 \end{pmatrix}$ is denoted as $\mathbf{u} \perp \mathbf{v}$ and defined by:

$$\mathbf{u} \perp \mathbf{v} = u_1 v_2 - u_2 v_1 = |u|\,|v| \sin(\theta)$$

If the vectors are normalized: $|u| = |v| = 1$ then $\mathbf{u} \perp \mathbf{v} = \sin(\theta)$, which is helpful to calculate the angle between two vectors.

Moreover due to the characteristics of the sinus function the sign of the exterior product indicates whether the angle is oriented clockwise or counterclockwise of the direction of one of the vectors.

As a matter of fact, the result can be used to determine on which side one vector is to another:

- $\mathbf{u} \perp \mathbf{v} = 0 \rightarrow$ v and w are collinear
- $\mathbf{u} \perp \mathbf{v} > 0 \rightarrow$ v is located “left” of u
- $\mathbf{u} \perp \mathbf{v} < 0 \rightarrow$ v is located “right” of u

2.3 Edge functions

Several voxelization techniques utilize so called edge functions. [Pineda, 1988] defined them as functions which divides a two dimensional space into three regions. “Right”, “Left” and exactly on the separating line. The direction vector $\mathbf{u}$ of a line is defined by two points:

$$\mathbf{r} = \begin{pmatrix} r_1 \\ r_2 \end{pmatrix} \text{ and } \mathbf{s} = \begin{pmatrix} r_1 + ds_1 \\ r_2 + ds_2 \end{pmatrix}$$

$$\mathbf{u} = \mathbf{s} - \mathbf{r} = \begin{pmatrix} ds_1 \\ ds_2 \end{pmatrix}$$

to determine on which side an arbitrary point $\mathbf{p} = \begin{pmatrix} p_1 \\ p_2 \end{pmatrix}$ is, one can utilize the exterior product of $\mathbf{u}$ and $\mathbf{v} = \mathbf{p} - \mathbf{r} = \begin{pmatrix} p_1 - r_1 \\ p_2 - r_2 \end{pmatrix}$

$$\mathbf{u}\perp\mathbf{v} = ds_1(p_2 - r_2) - ds_2(p_1 - r_1) = \begin{pmatrix} -u_2 \\ u_1 \end{pmatrix} \bullet (\mathbf{p} - \mathbf{r}) = E(\mathbf{p})$$

$E(\mathbf{p})$ is called an edge function: $E(\mathbf{p}) > 0$ indicates that $\mathbf{p}$ is on the left side of the edge and $E(\mathbf{p}) < 0$ indicates the right side. Considering the definition of the normal $\mathbf{n}$ of an edge $\mathbf{u}$.

$$\mathbf{n} = \begin{pmatrix} -u_2 \\ u_1 \end{pmatrix}$$

the edge function can be reformulated to

$$E(p) = \mathbf{n} \bullet \mathbf{p} - \mathbf{n} \bullet \mathbf{r} = \mathbf{n} \bullet \mathbf{p} + \mathbf{c}$$

with $\mathbf{c} = -\mathbf{n} \bullet \mathbf{r}$. The edge function for a point with an offset $E(\mathbf{p} + \mathbf{t})$ becomes

$$E(\mathbf{p} + \mathbf{t}) = \mathbf{n} \bullet (\mathbf{p} + \mathbf{t}) + \mathbf{c} = \mathbf{n} \bullet \mathbf{p} + E(\mathbf{t})$$

2.4 Deferred Shading

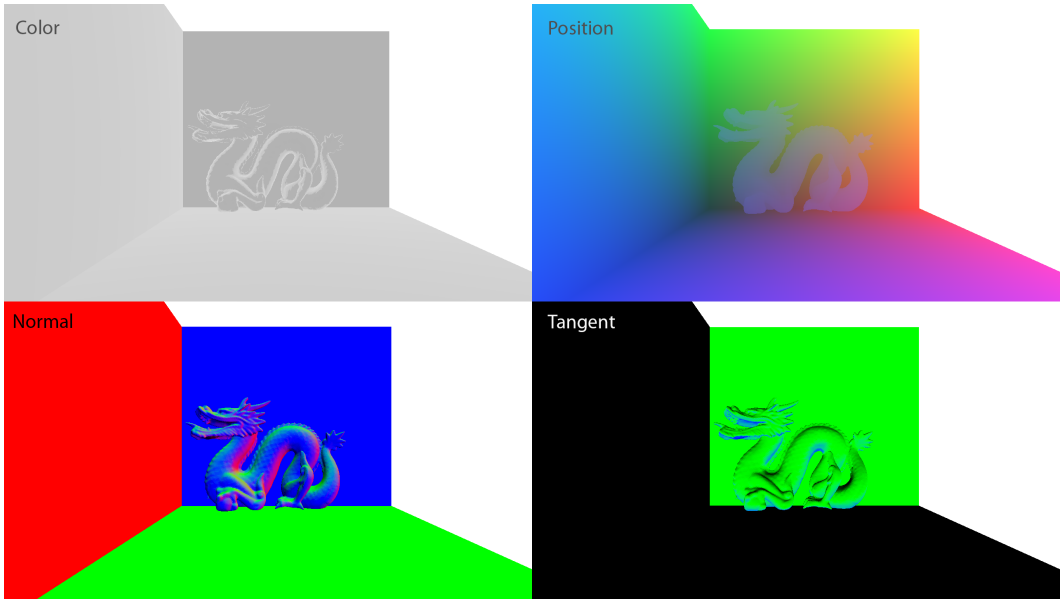


Figure 2.2: Example of a G-buffer consisting of four textures.

Deferred shading [Saito and Takahashi, 1990] represents an alternative approach to the classical forward shading process. Instead of drawing the rendering result directly to the

default frame buffer (e.g. to the screen) the deferred shading pipeline uses two passes to gather the final color values. The first pass computes relevant geometric information, e.g. positions, normals, material properties and stores them in separate 2D textures with the help of *multiple render targets*. This part of the pipeline is called geometry pass and the series of resulting textures is defined as geometric buffer or *G-buffer* (Figure 2.2).

The geometry pass is usually followed by the light pass which accumulates the information of the G-buffer and finally draws the rendering result to the screen. The primary advantage of this procedure is the decoupling of the scene geometry from lighting. This provides the possibilities for a number of techniques such as deferred lighting or numerous effects like motion blur or water refraction [Rosado, 2008, Koonce, 2008].

However, the separation of geometry and lighting leads to some disadvantages: The most relevant limitation for this work is the lack of hardware supported anti-aliasing since interpolated subsamples of geometric information such as position or normals produce incorrect results. Common techniques to overcome this problem are post-process edge-smoothing approaches. [Maule et al., 2012] provides an overview of related methods. Further limitations are difficulties in handling transparent object and organizing of multiple material properties. Solutions for each subject are detailed in [Shishkovtsov, 2005]. Another factor to consider is the increased memory consumption due to the storage of additional textures in the G-Buffer.

2.5 Unrestricted memory access in shaders

Previous shader models (Shader Model version < 5) were strongly limited in compute operations because of the restrictions of write locations. Output procedures were exclusively performed through the ROP (render output unit) which did not allow random access and 3D-addressing. With the release of Shader Model 5 dynamic addressing of arbitrary buffers and textures became available. Consequently, *OpenGL* specification 4.2 introduced standardized image unit access in shaders with the *imageStore* and *imageLoad* operations. Beyond textures (bound in GLSL as *image*), these new features allow dynamic addressing of linear memory regions (*buffer objects*) using "buffer textures" bound to a GLSL *imageBuffer*. With these powerful tools, computation in shaders offers nearly the same flexibility as GPGPU (General purpose GPU) interfaces like *CUDA* or *OpenCL*.

However, direct memory access has one disadvantage: Usually, image data is organized in *sampler* structures and accessed via *texelFetches* to retrieve values on the GPU. In contrast to *imageLoad* a *texelFetch* takes advantage of the texel cache which enables fast access to the desired location due to the optimized memory management. AMD's SDK documentation [AMD, 2012] describe this type of memory access as *FastPath* whereas *imageLoad* operations rely on the slower *CompletePass* which supports advanced operations including atomics and sub-32-bit data transfers.

3 Surface Voxelization

Surface voxelization is the process of converting a geometric object from a mesh representation into a set of voxels that approximates the object's surface. The voxelization generates a discrete volume representation of a geometric object by testing and recording intersections between the object's surface and a voxel grid.

3.1 Related Work

There are two fundamentally different solutions to solve the voxelization problem on the GPU: The first one utilizes the graphics pipeline, especially the rasterization process and the fragments it produces and is therefore often called fragment-parallel approach. The second one still relies on the GPU as a massively parallel computing device (General Purpose GPU) but avoids the hardware rasterization process completely. Instead, each triangle will be voxelized simultaneously, therefore these techniques are called triangle-parallel voxelization.

Voxelization using the Rendering Pipeline. Over the recent two decades many different voxelization techniques have been devised, mainly due to the fast development of new, more powerful GPUs and their potential to be utilized as a highly parallel processing unit.

One of the most relevant approaches is presented by [Fang et al., 2000]. The authors exploit the fixed function rendering pipeline to achieve more or less real-time surface and solid voxelization with a decent resolution of 128^3. The idea is to render the volume slice wise (one slice per voxel in the viewing direction) by restricting the view volume to the current depth span of the slice. After the rasterization process of one slice, the framebuffer is written to a 3D texture at the according depth, so each filled pixel in 2D is equivalent to a filled voxel in 3D. Although the algorithm occasionally misses thin structures, the idea was the basis of many following works which improved performance and robustness of this technique. [Li et al., 2003] reduces the number of rendering passes by automatically skipping empty spaces with a technique identical to depth peeling [Everitt, 2001].

[Dong et al., 2004] proposed a method which reduces the number of voxel misses by considering three grid axes as viewing direction and rendering only triangles which direction of maximum projection corresponds with the current view direction. Additionally, the overall performance was increased or more precisely the amount of slices was reduced by firstly encoding separate bits of the texture to multiple voxels and secondly filling these multiple voxels simultaneously by using multiple render targets, thus treating many slices concurrently in a single rendering pass. To enable rapid storing of the correct bit in the result texture the authors use texel fetches to a predefined texture which provides the appropriate bitmasks and use additive alpha blending to set the required bit values. [Eisemann and

Decoret, 2006] employed a similar approach, using only one rendering direction but potentially enhancing the accuracy of the result by using the actual bounding box of the geometry to locally provide finer slicing.

The results of the previous mentioned techniques are often inaccurate, because of the point-in-triangle test of the conventional hardware accelerated rasterizer (Section 3.3). [Hasselgren et al., 2005] proposed *conservative rasterization*, a method to overcome these problems by temporarily expanding the triangle. Based on that [Zhang et al., 2007] employed a complete algorithm for conservative voxelization.

The introduction of unrestricted memory access in shader ended the era of slice-wise voxelization, because textures can be filled from arbitrary pixel positions. Using these new features and a technique called *dominant axis transformation* (detailed in Section 3.2), [Crassin and Green, 2012] reduced the number of passes to a single pass. Although these more recent approaches produce watertight voxelizations, the performance can suffer when dealing with degenerated or sub-voxel-size triangles.

Voxelization using general purpose GPU. With the possibility of utilizing GPUs as massively parallel computing devices for general purposes, scientists took the opportunity to circumvent the problems of the hardware rasterization based voxelization approaches.

Schwarz and Seidel [Schwarz and Seidel, 2010] proposed a voxelization implemented entirely on the GPU. The algorithm operates triangle-parallel and iterates over the expansion of a triangle, testing every voxel which potentially intersects with it. The employed triangle-box intersection test is based on [Akenine-Möller, 2005] where the authors use edge functions to determine the intersections. In contrast to the rasterization approaches, the triangle-parallel approaches generally suffer from thread imbalances if the size of the primitives vary across the dataset, especially if single triangles are too large.

Following this observation, Pantaleoni [Pantaleoni, 2011] employed a complete voxelization pipeline which consists of two main steps: Firstly, a coarse triangle-parallel rasterizer similar to Schwarz and Seidel, creates a list with all possible intersections. Secondly, after sorting this list a tile-based approach rasterizes the geometry with a fine resolution. This technique splits large triangles according to the initial coarse resolution, hence it ensures a better distribution of workload among the working threads. Furthermore the author employed an a priori classification of the triangles to optimize the load balance of the procedure. Despite being the most advanced technique, the implementation is complex and the focus of Pantaleoni's work is the voxelization of huge and high detailed scenes. Since the focus of this work is on small dynamic scenes it is not appropriate to employ this approach.

Instead a technique is proposed which combines the advantages of the fragment- and triangle- parallel approaches. It is based on [Rauwendaal and Bailey, 2013] in which the authors proposed a heuristic to classify triangles for voxelization. The classifier provides information about whether a fragment- or triangle-parallel technique is more suitable and avoids the need for complex work/load balancing schemes.

The fragment-parallel voxelization part uses Crassin and Green's approach as basis and the triangle-parallel voxelization part is build upon Schwarz and Seidel's technique.

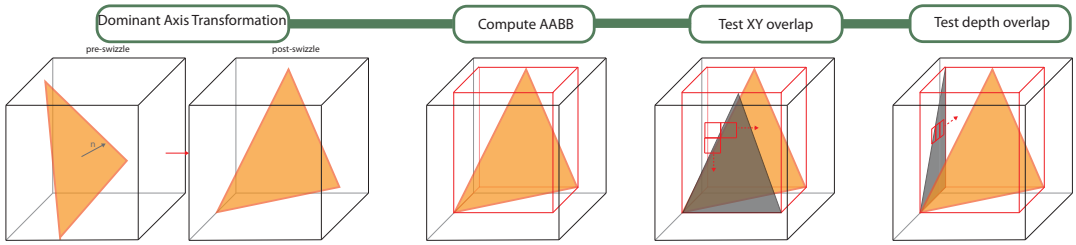


Figure 3.1: An illustration of the crucial steps of the triangle-parallel voxelization pipeline

3.2 Triangle-parallel Voxelization

Pipeline

The concept of voxelizing a given surface is fairly simple: Iterate over the discrete voxel space and check where the primitives intersect the voxels and store these locations in an arbitrary memory structure. The problem with this naive approach is the large overhead due to unnecessary intersection tests. The goal is to minimize the number of tests to increase efficiency.

Calculating the axis aligned bounding box of the triangle and iterate over this reduce set of voxels avoids most of the unnecessary tests. Furthermore, the input geometry is changed based on the dominant normal direction such that the *XY*-plane is always the plane of maximum projection (*dominant axis transformation*, Section 3.2). For efficiency the voxel-triangle intersection test is replaced by three 2D box-triangle overlap tests, one for each major direction (Section 3.2). To reduce the number of tests, the triangle's orthographic *XY*-projection is initially checked for overlaps. If an overlap exists, the minimal depth range (along z-axis) at this particular position is calculated and iterated over to check for the remaining overlaps in the *YZ*- and *ZX*-plane. The depth range is at most three voxel thick due to dominant axis transformation. Moreover, the remaining tests along the z-axis can be skipped entirely if the depth extend is only one voxel.

The surface voxelization can be summarized by these steps (Figure 3.1):

1. Dominant axis transformation into *XY*-plane
2. Compute axis aligned bounding box of the triangle to determine the number of possibly intersecting voxel in *XY*-plane.
3. Iterate over each voxel along the x- and y-axis and test whether the orthographic 2D projection of the triangle overlaps.
4. Calculate the minimal depth range and test the remaining 2D projections (*YZ, ZX*) for overlap.

Dominant Axis Transformation

The dominant axis of a triangle is one of the scene's three main axes which maximizes the projected area (Figure 3.2). The determination of the dominant axis is easy since it is the

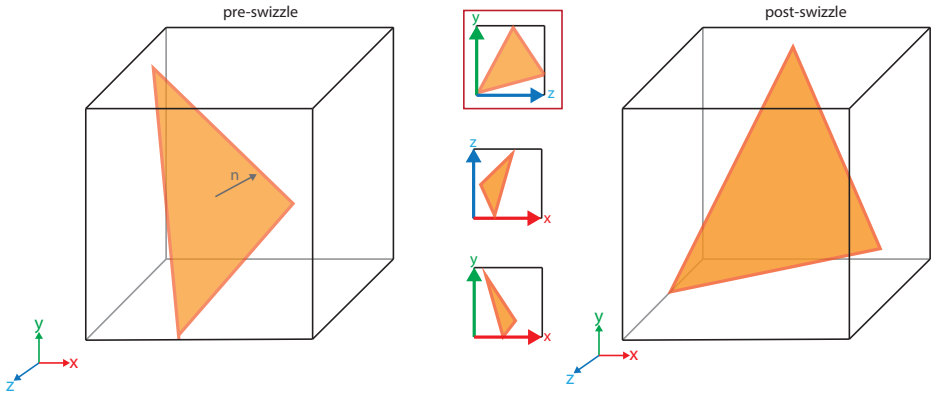


Figure 3.2: An example of the dominant axis transformation. **Left image:** Shows the original triangle. **Middle column:** The orthogonal projection along the three main axes. In this example, the orthographic projection onto the *YZ*-plane maximizes the triangle's area. **Right image**: Corresponding triangle after the transformation

one which provides the maximum value for $|\mathbf{n} \cdot \mathbf{e_i}|$, with $\mathbf{n}$ denoting the triangle's normal and $\mathbf{e_i}$ denoting the unit vectors of the three main axes.

The triangle is projected following the convention, such that the *XY*-plane is always the one of maximal projection. This can be accomplished by a simple hardware-supported vector *swizzle* of the triangle's vertices $\mathbf{v_i}$:

$$\mathbf{v}_{\mathbf{i,xyz}} = \begin{cases} \mathbf{v}_{\mathbf{i,yzx}} & ,if\ x\ is\ dominant \\ \mathbf{v}_{\mathbf{i,zxy}} & ,if\ y\ is\ dominant \\ \mathbf{v}_{\mathbf{i,xyz}} & ,if\ z\ is\ dominant \end{cases}$$

Consequently, when the final voxel location is determined the components must be *unswizzled* before storing.

Voxel-triangle intersection test

Given a set of triangles and a 3D voxel grid, the goal is to find all voxels intersected by a triangle. To determine whether a triangle $\mathscr{T}$ ($\triangle \mathbf{v_0 v_1 v_2}$) and a voxel $\mathscr{B}$ intersect, the 2D orthogonal projections of $\mathscr{T}$ along all three coordinate directions (x, y, z) is tested for overlap with the according planes of the voxel (XY, YZ, ZX). This procedure is very similar to the SAT (separating axis theorem) test [Akenine-Möller et al., 2002], however with the help of edge functions the actual projection of $\mathscr{T}$ is not required. Instead the edge functions of each of the triangle's edges will be evaluated at the "most interior" corner of $\mathscr{B}$. The edge functions will be constructed according to the following convention: The normal of the triangle's edge will always point to the triangle's center. Consequently a positive result of the edge function indicates the location of the corresponding point relative to the edge

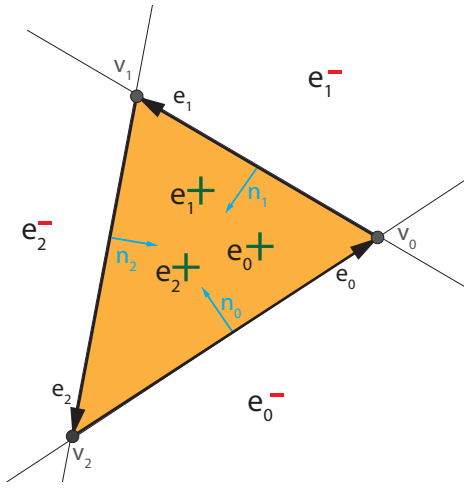


Figure 3.3: An example of the edge functions of a triangle. A single function divides the triangles plane into a positive and negative half-plane. If all three edge functions are non negative for a given point, the point is guaranteed to be inside the triangle.

normal, hence if all three edge functions of a triangle are positive the point is located inside the triangle (Fig.:3.3).

A voxel $\mathscr{B}$ is defined by its minimal corner $\mathbf{p}$ and maximal corner $\mathbf{p}+\mathbf{dp}$. Using the XY-coordinate plane as an example, the "most interior" corner $\mathbf{s}^{\mathbf{xy}}_{\mathbf{e_i}}$ of a voxel with respect to the triangle edges e_0, e_1, e_2 is determined by :

$$\mathbf{s}^{\mathbf{xy}}_{\mathbf{e_i}} = \mathbf{p}^{xy} + \mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_i}}$$

with

$$\mathbf{ds}^{xy}_{e_i} = \left(\left\{ \begin{array}{ll} dp_x, & n^{xy}_{e_i,x} > 0 \\ 0, & n^{xy}_{e_i,x} \leq 0 \end{array} \right\}, \left\{ \begin{array}{ll} dp_y, & n^{xy}_{e_i,y} > 0 \\ 0, & n^{xy}_{e_i,y} \leq 0 \end{array} \right\} \right)^T$$

which is similar to the determination of Haines and Wallace's [Haines and Wallace, 1991] critical point. $\mathbf{n_x}, \mathbf{n_y}$ are the components of the inward facing edge normals which are calculated by:

$$\mathbf{n}^{\mathbf{xy}}_{\mathbf{e_i}} = \left(\begin{array}{c} -e_{i,y} \\ e_{i,x} \end{array} \right) * \left\{ \begin{array}{ll} +1, & n^{\mathscr{T}}_z > 0 \\ -1, & n^{\mathscr{T}}_z \leq 0 \end{array} \right\}$$

where $\mathbf{n}^{\mathscr{T}}$ denotes the triangles normal.

The final overlap test of the voxel plane and the triangle's projection breaks down to the evaluation of the three edge functions (denoted by $E(\mathbf{p})$) at their critical point which is:

$$E_i(\mathbf{p}^{\mathbf{xy}} + \mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_i}}) = \mathbf{n}^{\mathbf{xy}}_{\mathbf{e_i}} \bullet \mathbf{p}^{\mathbf{xy}} + E_i(\mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_i}})$$

$$E(\mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_i}}) = -\mathbf{n}^{\mathbf{xy}}_{\mathbf{e_i}} \bullet \mathbf{v}^{\mathbf{xy}}_{\mathbf{i}} + \mathbf{n}^{xy}_{e_i} \bullet \mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_i}}$$

furthermore, to prevent branching the calculation of $E_i(\mathbf{p}^{\mathbf{xy}} + \mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_i}})$ can be factored out to:

$$E_i(\mathbf{p}^{\mathbf{xy}} + \mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_i}}) = \mathbf{n}^{\mathbf{xy}}_{\mathbf{e_i}} \bullet \mathbf{p}^{\mathbf{xy}} - \mathbf{n}^{\mathbf{xy}}_{\mathbf{e_i}} \bullet \mathbf{v}^{\mathbf{xy}}_{\mathbf{i}} + \max(0, \mathbf{n}^{\mathbf{xy}}_{\mathbf{e_{ix}}} \mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_{ix}}}) + \max(0, \mathbf{n}^{\mathbf{xy}}_{\mathbf{e_{iy}}} \mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_{iy}}})$$

If all three Edge Function result are equal or greater then zero (Figure 3.4), so if

$$\bigwedge_{i=0}^{2} (E_i(\mathbf{p}^{\mathbf{xy}} + \mathbf{ds}^{\mathbf{xy}}_{\mathbf{e_i}}) \geq 0)$$

holds true, voxel $\mathscr{B}$ and triangle $\mathscr{T}$ overlap in this direction. The overlap tests for the remaining two planes (*YX*,*ZX*) is equivalent.

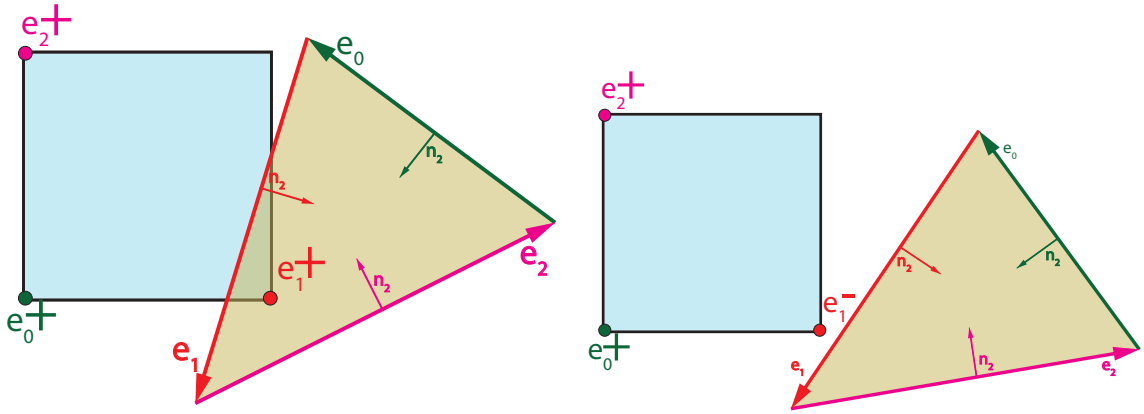


Figure 3.4: Evaluation of the edge function at the critical points. The colored points mark the critical points to the equally colored edge. The sign denotes the result of the edge function. **Left image**: All of the edge function's results are positive, hence the triangle overlaps the box. **Right image**: The edge function of the red edge turns out to be negative, indicating that the triangle does not overlap.

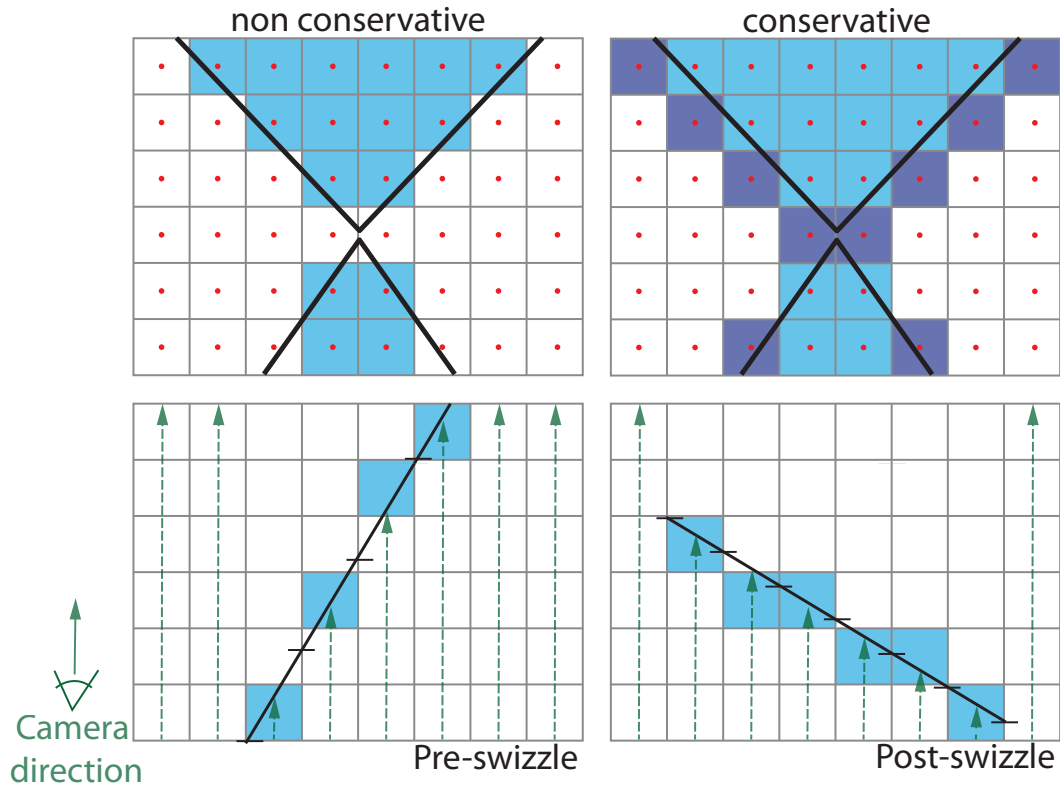


Figure 3.5: Illustration of two different rasterization problems. The black lines indicate the accurate contour of the triangles. The colored boxes are filled pixels after the rasterization process and the red dots represent the pixel centers. **Top row**: A non conservative rasterization versus a conservative rasterization. **Bottom row:** The resulting fragments of a triangle rasterization with an overly oblique camera angle versus the rasterization of the same triangle after dominant axis transformation.

3.3 Fragment-parallel Voxelization

The core of surface voxelization is testing the primitives for intersections with voxels. In the implemented triangle-parallel approach this intersection test is divided into 2D triangle-box overlap tests, one for each major axis. The hardware accelerated rasterizer of the graphics pipeline is specifically designed to compute these overlaps, while providing dynamic parallelism with regard to the triangles and the boxes. Based on this causal relationship, the idea behind fragment-parallel voxelization is to move the required overlap tests to the rasterizer to vastly increase parallelism. However the scan conversion of the rasterizer is too inflexible to produce correct intersections without further adaption: One issue of the standard procedure of the hardware rasterizer is that a fragment will only be generated if the pixel center is covered by the geometry (Figure 3.5, top row). This underestimation can lead to intersection misses which causes holes in the voxelization result. To eliminate this issue an

additional expansion of the triangles is executed as explained in Section 3.3, to ensure that necessary pixel centers are covered.

Aside from underestimation, another hindrance is the determination of the depth value per fragment especially in cases of overly oblique camera angles. The rasterizer produces only one fragment per pixel of an input primitive therefore only one voxel can be filled. As shown in Figure 3.5 (bottom row), this can cause holes in the voxelization since it is possible that one fragment can fill multiple voxel along the depth axis. To solve this problem dominant axis transformation (Section 3.2) is applied, forcing the camera's view axis to be aligned towards the plane of maximum projection.

To simplify terminology, voxels will be further referred to as pixel since the rasterizer performs the scan conversion on the image plane.

Conservative Rasterization

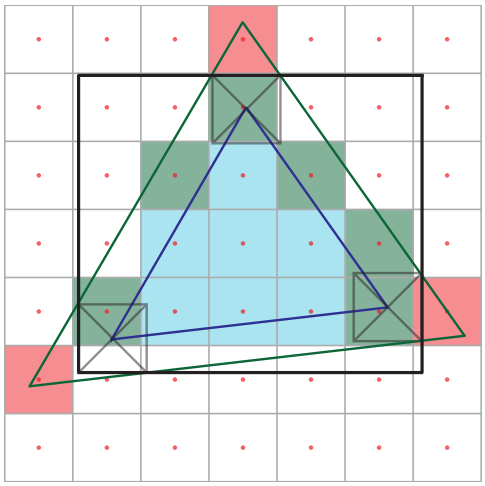


Figure 3.6: An example of conservative rasterization. The boundary of the original triangle (blue) is expanded by half a pixel cell. The green pixels show additional filled cells after conservative rasterization. The red boxes point out the incorrect rasterized geometry (over conservative), which will be clipped by testing them against the conservative axis-aligned bounding box (black rectangle).

A preprocessing step is executed to perform *conservative rasterization* after [Hasselgren et al., 2005] to resolve the issue of intersection misses. The idea is to expand the original triangles until every intersected pixel is covered on its center. Figuratively speaking, the enlargement is obtained by sweeping a pixel sized rectangle along the triangle's edges and computing the extreme values. This is realized before the rasterization process by shifting the triangle's edges $\mathbf{e_0}, \mathbf{e_1}, \mathbf{e_2}$ by the length of a pixel's semi diagonal d in normal direction. The new vertex position are given by the intersection points of the translated edges. Fortunately the intersection points can be calculated without actually determining the new edge equations.

Taking vertex $\mathbf{v_0}$ of the triangle, it's edges $\mathbf{e_0}, \mathbf{e_2}$ and edge normals $\mathbf{n_0}, \mathbf{n_2}$ as an example, the new vertex $\mathbf{p_0}$ can be calculated by

$$\mathbf{p_0} = \mathbf{v_0} + d \cdot (\frac{\mathbf{e_0}}{cos(\alpha)} + \frac{\mathbf{e_2}}{cos(\beta)})$$

With α denoting the angle between $\mathbf{e_0}$ and $\mathbf{n_2}$ and β denoting the angle between $\mathbf{e_2}$ and $\mathbf{n_0}$. With the dot products definition:

$$\mathbf{e} \bullet \mathbf{n} = |\mathbf{e}| \cdot |\mathbf{n}| \cdot \cos(\alpha)$$

the calculation can be simplified to:

$$\mathbf{p_0} = \mathbf{v_0} + d \cdot (\frac{\mathbf{e_0}}{\mathbf{e_0} \bullet \mathbf{n_2}} + \frac{\mathbf{e_2}}{\mathbf{e_2} \bullet \mathbf{n_0}})$$

if the edges and normals are normalized. The other two vertices can be determined equivalently.

The new triangle will generate the necessary fragments for voxelization. Unfortunately, acute angles can cause the creation of additional fragments which do not correspond to the original intersections (Figure 3.6). These overly conservative fragments will be discarded in a postprocessing step by testing them against the conservative axis-aligned bounding box of the triangle which is acquired by expanding the original bounding box by half a pixel length.

3.4 Hybrid Voxelization

To optimize the performance of the voxelization algorithm without complicated work and load balancing schemes, fragment- and triangle-parallel methods are combined in a hybrid voxelization approach after [Rauwendaal and Bailey, 2013]. This method is based on the observation that each technique only performs well if the input meshes exhibit certain characteristics. E.g. the triangle-parallel approach is more efficient when voxelizing meshes which feature an even and regular triangulation. The reason for this is that the naive approach has no mechanism to balance the workload of threads. Consequently, individual threads work alone to voxelize large triangles which slows down the voxelization process. In fragment-parallel voxelization the problem is reversed. Generally they perform well in scenes with a low number of triangles regardless of their size. However, the technique performs poorly in high tessellated meshes with small triangles compared to the voxel size. The reason for this is that sub-voxel-sized triangles produce a large overhead of fragments due to the expansion for conservative voxelization.

In summary, the worst-case scenario for the fragment-parallel voxelization becomes the best-case scenario for triangle-parallel voxelization and vice-versa. Therefore, to support best

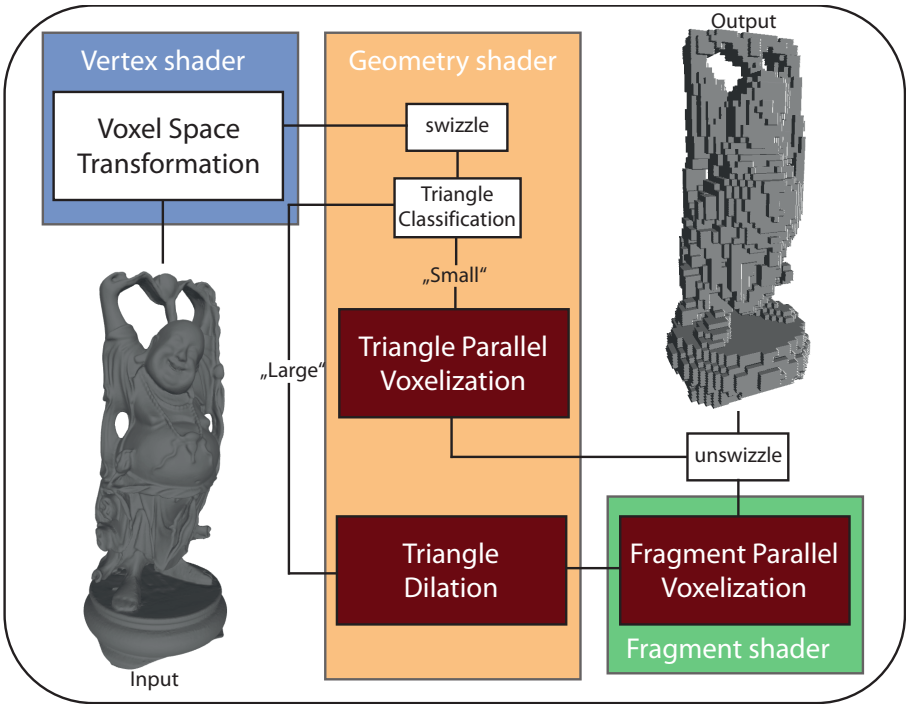


Figure 3.7: An illustration of the complete voxelization pipeline

performance the two techniques will be used in combination. To determine when each approach is appropriate the triangles will be classified in a preprocessing step. In performance tests the simplest classifier provided the best results as the overhead for the classification itself became unnoticeable. The classifier used was the projected 2D surface area in the dominant axis direction of the triangle. This measure provides information about the number of iterations of the triangle based method which is usually the greatest performance loss. Consequently a cutoff value is set, dividing the triangles into triangle-parallel (“large”) and fragment-parallel (“small”) triangles.

3.5 Implementation

The hybrid voxelization pipeline is implemented entirely in the rendering pipeline using a single pass as illustrated in Figure 3.7.

The vertex shader transforms the vertices in voxel space. The geometry shader stage is the core of the approach as it contains the dominant axis transformation (*swizzle*), the triangle classification, the complete triangle-parallel approach and the preprocessing for the fragment-parallel method which is the expansion of the primitives for conservative rasterization. Although the triangle-parallel part is intended to run on general purpose GPUs, the technique can be transferred to the geometry shader since this stage handles all primitives of the scene in parallel.

The fragment-parallel technique is finalized in the fragment shader stage by reverting the dominant axis transformation (*unswizzle*) and discarding all fragments which are located outside of the conservative axis aligned bounding box.

3.5.1 Sparse voxel storage - Voxel fragment list

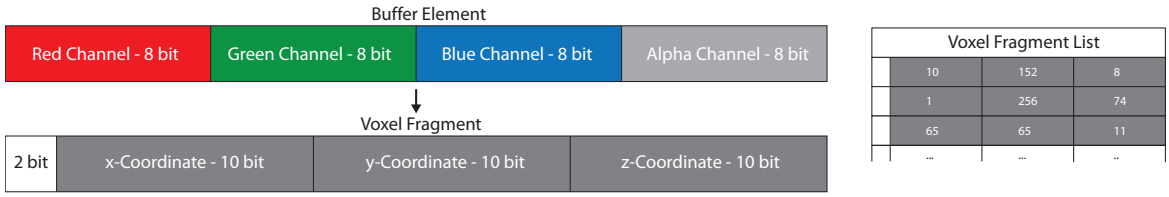


Figure 3.8: Left: The bit repartitioning to store voxel fragments. **Right:** Illustration of the voxel fragment list.

The challenge of exploiting sparsity in the process begins with the storage of the voxelization result. The requirements of this task are: Storing only the positions of the filled voxel without initializing unnecessary memory and converting it to a form which supports the construction of the octree using the massive parallel computing power of the GPU.

Since a dense 3D-texture is no option for storing due to the high memory requirements, a linear buffer is used instead which is further referred to as *voxel fragment list*. Each element of the list is a 32-bit RGBA value with 8 bits per channel. A voxel fragment is an element which contains voxel-space coordinates indicating a filled voxel at this position. To store this information the 4x8 bit RGBA channels are repartitioned to 3x10 bit coordinate channels which allows voxel resolutions up to 1024x1024x1024 to be stored (Figure 3.8). The additional 2 bits are unused.

Each thread during the voxelization process which fills one or multiple voxel, irrespective whether its fragment- or triangle-parallel voxelization, will append the voxel position to the linear buffer. Consequently, multiple threads will access this list concurrently. The next free available space in the buffer is determined by a global atomic counter. Unfortunately, redundant voxel fragments in the list are unavoidable if multiple primitives intersect the same voxel. Nevertheless this representation still saves a large amount of memory compared to 3D-textures.

4 Sparse Voxel Representations

4.1 Related Work

Storing a high resolution voxelization of entire scenes on the GPU is usually a huge problem due to memory limitations. The consequence is often to limit the detail by reducing the voxel resolution which is not a desired solution. General scenes however, consist of large clusters of empty space which can be exploited to save a huge amount of memory. Many different techniques which enable fast storage and also efficient lookups in three dimensional space have been employed.

[Kraus and Ertl, 2002] proposed adaptive texture maps which locally lower the resolution of a 3D section with less detail and rearrange them for compressed storage. The border of the compressed section is duplicated to enable texture interpolation. [Nießner et al., 2013] employed an efficient GPU-accelerated spatial hashing technique. A smart bucketing system allows an even distribution of filled entries across the hash table which allows compact storing of voxel fragments, if the hash table and bucket size are chosen appropriately. Since the lookup in a hashtable is fast (complexity O(n)) the trilinear interpolation can be managed on the fly. Although these techniques allow fast storing and retrieving of values, they do not provide prefiltered data with different level-of-detail.

Octree based representation. The most common way to exploit sparsity is constructing a sparse voxel octree (SVO) which is a hierarchical structure that partitions the three dimensional space by recursively subdividing it into eight equally size octants. The structure becomes compact by stopping the subdivision when an empty space is reached leaving only necessary information in the nodes.

Early advances in this field such as the approach of [Laur and Hanrahan, 1991] often focus on 3D volume rendering. The authors used a precomputed hierarchical octree structure stored on the CPU and provide prefiltered data similar to mipmaps [Williams, 1983] in the interior nodes. The method is slow, as it requires a lot of CPU interactions during the rendering. [Benson and Davis, 2002] as well as [DeBry et al., 2002] coincidentally presented a nearly identical sparse octree representation stored in the texture memory of the GPU. The structure is precomputed on the CPU and transferred to the GPU to enable real-time painting and rendering of a solid 3D-texture on an unparameterized model (a model without 2D texture coordinates). To retrieve trilinear or tricubic interpolated values despite neighboring octree cells being empty, Benson and Davis constructed a virtual uniform grid at the desired resolution around the sample point. Based on this idea many algorithms were developed [Lefebvre, 2005, Bastos and Filho, 2008, Gobbetti et al., 2008, Crassin et al., 2009] which

increased memory efficiency and quality by improving the octree representation on the GPU. There are two drawbacks to these approaches: On the one hand, they often require CPU-GPU interactions which can lead to poor performance. On the other hand, these techniques pay no attention to the construction of the octree since the structure is computed once for static scenes.

In this work the focus lies on real-time rendering of full dynamic and interactive scenes, which requires the reconstruction of the SVO in every frame. [Crassin and Green, 2012] proposed a technique for parallel constructing and storing of the SVO on the GPU. The algorithm does not need to transfer any data between the CPU and GPU, which leads to top performance.

4.2 Pipeline

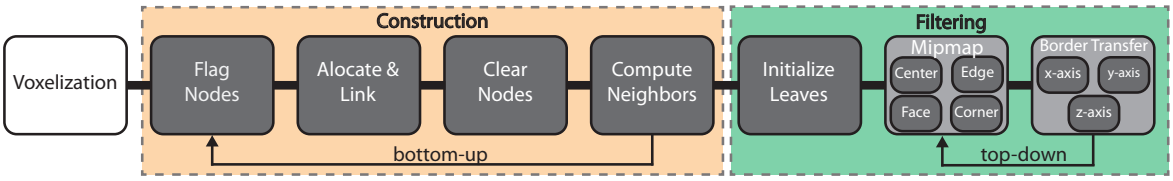


Figure 4.1: Illustration of the complete octree construction pipeline

The scene is voxelized once at the finest resolution of the octree which is 2^d where d denotes the desired tree depth. Although using the finest resolution for the coarser levels will generate a large overhead, it is still cheaper than revoxelizing the scene at each level. The result of the voxelization is stored in a dense linear buffer, which is called voxel fragment list (Section 3.5.1). After the voxelization is complete the sparse voxel octree construction begins. The approach is based on [Crassin and Green, 2012], where the authors used a top-down approach to build the SVO. The basic concept is to start at the root and progressively subdivide each node, one level at a time, until the desired depth is reached or the section of space is empty. The octree structure itself is very compact as it is pointer based only in the top-to-bottom direction not vice versa. To enable fast parallel computation with acceptable work/load balancing, the complete subdivision process per level takes three passes followed by an additional pass to compute the neighbors:

1. Tag nodes
2. Allocate and link children
3. Clear nodes
4. Compute neighbors

After the construction is complete the second part of the pipeline is the filling of the leaves and filtering these values into the interior nodes of the tree. This part is done bottom-up and takes one initial step plus two steps per octree level:

1. Initialize leaf values + Initial border transfer
2. Mipmap
3. Border transfer

4.2.1 Node Subdivision

Tag Nodes

Each element in the voxel fragment list corresponds to a filled voxel in voxel-space and initiates a node subdivision in the current depth. Nodes which are qualified for subdivision will be marked and subdivided in the next pass.

Allocate and link children

The marked nodes are subdivided by allocating memory for the 2x2x2 children and linking them to the parent node. As previously mentioned only the parent has information about their children not vice versa.

Clear nodes

The next step is the memory initialization of the new children which is simply zeroing the newly allocated space.

Compute neighbors

After the new octree level is created, the three neighbors (x,y and z) of each child are computed and stored, as they are required multiple times in following steps of the algorithm.

4.2.2 Filtering

A fast and accurate interpolation of values during the cone tracing process is the foundation for a smooth ambient occlusion computation and a visually plausible rendering result. Since hardware accelerated trilinear interpolation and mipmapping is only available in a dense 3D-texture [Crassin et al., 2011] proposed a brick structure to store the leaf and interior values which allows smooth interpolation. A brick is a 3x3x3 texture space which approximates the scene section of their associated octree node and its surroundings. The center brick cell contains the voxel value at the corresponding location and the border brick cells store the linear interpolated values of its neighbors.

All bricks are stored in a large 3D-texture (*brickpool*) to enable trilinear interpolation inside of each brick. Bricks are stored unordered inside the texture. Only elements of a single octree level are stored at consecutive memory locations due to the level-wise construction of the tree (Figure 4.2). A pointer to the brick's position is stored inside the octree node to retrieve the correct values when required.

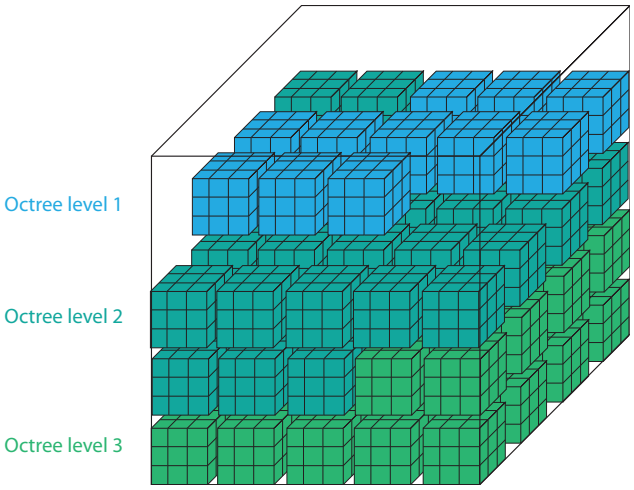


Figure 4.2: Illustration of the brickpool, a 3D-texture with randomly organized bricks.

Once the octree structure has been build, the associated brick has to be filled with the according values. The values inside brick cells represent the ratio of filled to empty space in the corresponding volume, hence the opacity.

Initialize leaf values + initial border transfer

Leaves which are not empty will be filled by setting the brick center to 1.0. Border values inside the brick cells are duplicated to enable linear interpolation inside each individual brick. It is inefficient to compute these values in one pass, since neighboring brick cells perform the same operation. In fact, the same operation is performed up to eight times. So instead, only a partial results is computed for the outer brick cells. The values are finalized by an efficient border transfer scheme (Section 4.2.2).

Mipmap interior nodes

To get the best possible representation of the scene in the coarser interior nodes, the finer levels have to be filtered. The filtering process traverses the octree in bottom-up direction, one level at a time. To achieve better performance the filtering is divided into four separate passes, followed by border transfer. In each pass the data of the eight children is fetched.

Centers: The first pass computes the brick center, by averaging the 27 surrounding values of the finer level. Note that there is a total of 64 surrounding cells, but excluding similar brick cells due to the duplicated border of adjacent bricks only leaves the 27 necessary values (Figure 4.3 left).

Faces: The second pass computes half the value of the center cell of the 6 brick faces, which means 18 neighbors are involved (Figure 4.3 middle).

Edges: The third pass calculates values according to the 12 edge cells utilizing 12 neighboring values per edge.

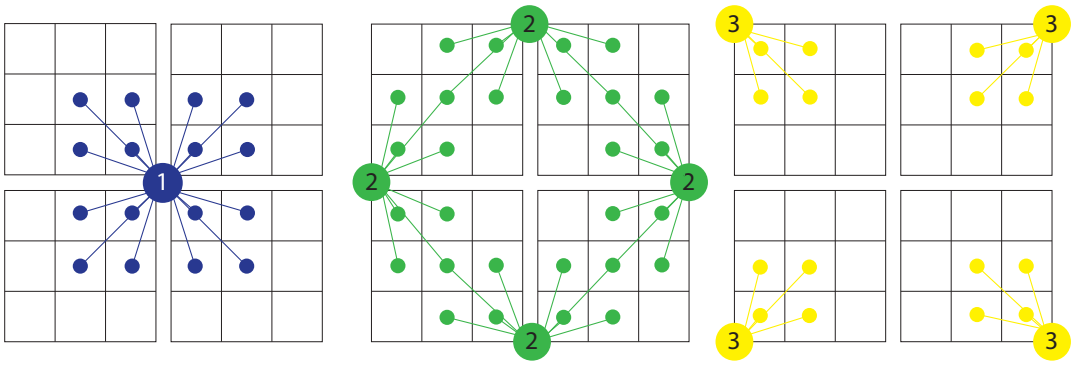


Figure 4.3: Illustration of the three passes of the filtering process for the interior nodes in 2D. The large circles displays the position in the parent brick and the small circles indicate the necessary cells for interpolation.

Corners: Finally, in the last pass the partial result of the brick corners is calculated by averaging the 8 neighboring cells (Figure 4.3 right).

The parent bricks are now in a similar situation as the leaves were before, so the border transfer will finalize these values as well.

Border transfer

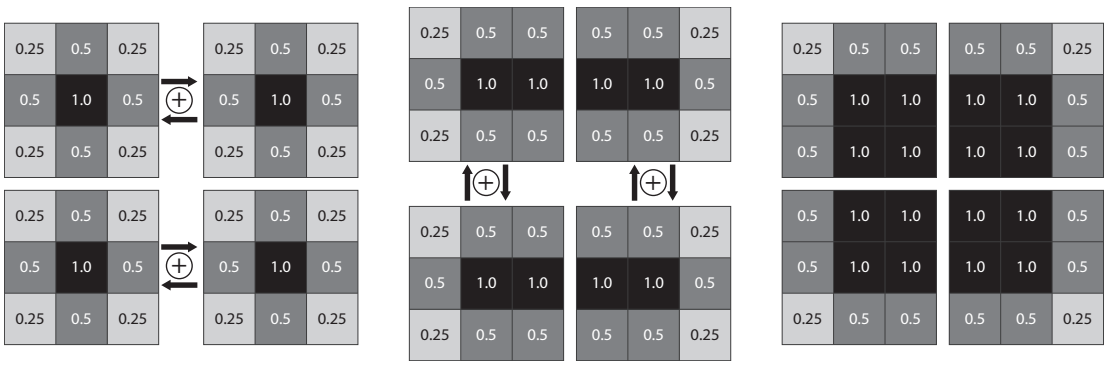


Figure 4.4: Illustration of the border transfer in 2D. The first two images demonstrate the two passes of transfer and the last image shows the according result.

In order to calculate the border of a brick, the partial values of each neighbor are summed up. Bricks consists of 3x3x3 equally sized cubes, so depending on the cube's position within a brick, the number of neighbors can vary between 1 (faces), 2 (edges) and 3 (corners), hence the values before the neighbor transfer are divided by their multiplicity. Since multiple cells are adjacent to each other, a node-parallel (one thread per node) border transfer would lead to thread collisions when storing the accumulated values back to the brick cells. Therefore, the border transfer is divided into three separate passes, one pass per grid direction (Figure

4.4). This method is very efficient since neighbors in one direction are unique so thread collisions are avoided.

Using the x-Axis as an example, each thread computes the values of two faces consisting of nine brick cells. One face belongs to the current node and the other one to the adjacent node in positive x-direction. The neighboring cells are summed up and the result is written back to both of them. It follows that the border brick cells of two neighbors is redundant, but the memory gain of the sparse representation makes this approach still worthwhile.

4.3 Implementation

4.3.1 Octree Storage - Nodepool

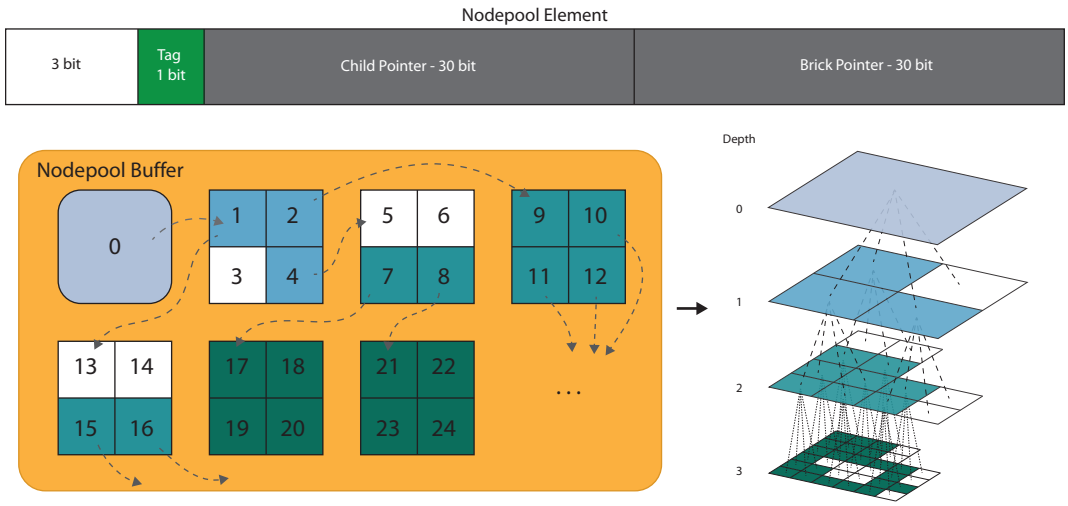


Figure 4.5: Top image: Bit partitioning of a nodepool element. **Left image:** Illustration of the nodepool buffer of a 2D quadtree. **Right image:** The resulting hierarchical representation.

Each pass is implemented in *OpenGL* with the utilization of *compute shaders* to maintain high performance. The advantage of using compute shaders instead of GPGPU APIs is that the required buffers can be initialized in the memory locations which will be later used for rendering, hence additional data transfers to different contexts are avoided. In this implementation octree nodes are stored in a linear buffer on the GPU which is called nodepool.

The elements of the nodepool allow storing of 64 bits information. The 64 bits are repartitioned as follows:

- 1 bit: Node tag, indicating whether the node is further subdivided or not. E.g. for empty regions and leaf nodes the bit is set to zero. The tag is also used to mark children during the subdivision process.

- 30 bits: Pointer to the 2x2x2 child tile if the node is interior and not empty. In fact, the pointer is merely the index to another element in the nodepool buffer
- 30 bits: Pointer to an 3x3x3 brick or empty if the according volume is empty.

Branches of empty nodes are terminated, to save memory. Also, each parent node only saves one address to locate its children instead of eight. This is possible because children of the same parent node are stored in consecutive memory areas (Figure 4.5 left). Furthermore, since the subdivision of nodes is handled level-wise all nodes are automatically sorted level-wise.

4.3.2 Thread scheduling

To enable the best possible performance in the subdivision and filtering process each pass utilizes a specific amount of threads.

Tag Nodes: The first pass, which is the node tagging process launches one thread for each element in the voxel fragment list. The threads fetch the position of their voxel fragment and traverse the octree until they arrive at a leaf node with the current maximum depth. The tag bit of this node is toggled to indicate a non empty node. Every single thread does exactly the same work in the process which leads to optimal work balancing. The only drawback is the fact that multiple fragments can hold the same voxel position which leads to numerous threads marking the same node. These unnecessary threads cannot be avoided since there are no alternative sparse storage techniques aside from the voxel fragment list which support this algorithm.

Allocate and link children: The allocation process launches one thread for each node at the current level. This pass uses the circumstance that octree nodes of one level are stored at consecutive memory locations by using an level offset which allows the threads to fetch their data directly without the traversal of the whole tree. Once the data is fetched each thread's node which was previously marked for subdivision, allocates space in the nodepool for 2x2x2 children. The next free address in the nodepool buffer is calculated by using an global atomic counter since multiple threads will try to allocate their children concurrently. To link the new children to the parent, the value of the atomic counter will be written back to the parent node. Afterwards the atomic counter is incremented. Although each thread performs the same task, the work balance is unfortunately still not optimal, because the time it takes to increment a global atomic counter in a massive parallel environment is dependent on the number of concurrently accessing threads. After this pass is finished the value of the atomic counter is stored since it is level offset of the next level.

Clear nodes: This pass is node-parallel which means one thread per newly allocated node is launched. The number of nodes is acquired by using the level offset before and after the allocation process. Once computed, the threads can directly access the memory location by

utilizing the last level offset. Each thread will initialize the memory location by zeroing the child pointer to prevent any conflicts with accidentally residual data.

Compute Neighbors: Of all the above passes this step is the worst in terms of work balancing. It is impossible to utilize the level offsets since the data in the nodes do not provide any information about their location. The only possibility to compute a node's location is by traversing the tree from the top to the desired node. To guarantee that all nodes will be reached in the process, 2^d threads have to be launched where d denotes the current depth of the octree. Each thread will traverse the octree to the maximum depth but since empty branches are terminated a high amount of threads will stop at a lower level. The other threads, now holding the location of their octree node, will traverse the octree again to an adjacent position. Finally the nodepool index of the neighboring node will be stored in a linear buffer (*neighbor buffer*).

Initialize leaf values: This pass launches one thread for every leaf node at the maximum level of the octree. Each thread fetches the data of their corresponding node and checks the tag bit. If the tag bit is toggled the corresponding brick will be filled with the initial values, otherwise the thread is terminated. After this procedure the leaves contain a partial result which will be completed by an initial border transfer.

Mipmap: After the leaf values are finalized by border transfer the filtering process for the interior nodes begins. The progress is done from bottom to top in $d-1$ steps for an octree of depth d. An octree traversal is not necessary since all the required nodes are automatically sorted per level inside the nodepool. One thread per node in the parent level is launched to mipmap their eight children. Each thread fetches the children's bricks from the brickpool to calculate the required cell values. To provide a better distribution of work the process is split into four passes (centers, corners, edges and faces).

Border transfer: Similar to the mipmapping step this pass requires one thread for every parent node at the current level. The neighbor buffer allows fast access to each required neighbor, which makes the transfer procedure very efficient.

5 Approximate Voxel Cone Tracing

5.1 Related Work

Ambient occlusion is a term which defines the amount of obstructed light by other screen geometry at any given surface point. In world-space methods this value is computed by casting rays from the point and record intersections with the scene. To produce a smooth rendering result typically a large number of rays is necessary, which leads to high computation times. While the first ray tracing algorithms [Whitted, 1980, Williams, 1983] were computationally expensive, researchers consecutively improved the methods to increase performance.

Visibility Rays in a typical scene are not organized randomly. Usually there is a high amount of spatially and directionally coherent rays which can be grouped together to drastically reduce the number of necessary rays. State of the art ray tracers exploit this coherence to reduce computation costs.

[Heckbert and Hanrahan, 1984] presented a technique called beam tracing to perform exact area sampling without explicitly using rays at all. Instead they use one large pyramidal beam originating at the eye which represents a volume of perspective parallel rays. The beam is recursively split at the scene's geometry boundaries until the resulting list of beams represents the visible surface. Although beam tracing has been considered slow because a sorting of the primitives is necessary to split the beams in the right order, [Overbeck et al., 2007] demonstrated that beam tracers are still considerable for scenes with large triangles sizes, when supported by the right acceleration structure. Unfortunately, beam tracers perform poorly when confronted with small triangle sizes since fine contours force a large number of beam splits. A similar idea was proposed by [Amanatides, 1984] in which the authors used cones, originating from the camera to every pixel of the image plane. The aperture angle is chosen so that the cone radius matches the size of a pixel at the distance of the virtual image plane. Each cone is tested for scene intersections and the fraction of the blocked cone volume is computed, so the combination of all cones is an approximation of the surface. This approach is an elegant merge of the benefits of beam tracing and the simplicity of ray tracing. However, the calculation of an exact intersection between a cone and an object is rather complex, which makes the technique slow for high resolutions.

[Wald et al., 2001] employed a highly optimized implementation of a SIMD (Single instruction multiple data) ray tracer. The most important performance improvement of this approach is the parallel tracing of packets of rays. They start by computing a BSP-tree (Binary Space Partitioning) of the scene and bundle four rays together to traverse the tree in parallel. The caching is optimized to provide coherent memory accesses during traversal.

There are two drawbacks to this approach: Firstly, Wald et al. did not detail the construction of the BSP-tree and results were only provided for static scenes. Secondly, although tracing rays in parallel, packet tracing still performs all traversal steps as a single-ray ray tracer. [Dmitriev et al., 2004] improved the SIMD ray tracer by parallelizing in a larger scale. They create multiple shafts and determine if all the rays inside the shaft's bounds hit the same node in a precomputed kd-tree or miss the scene completely, hence terminating them early. If they do hit the same node, all rays can be accurately tested in parallel. They use four bounding rays to test the shafts against the nodes of the tree. [Reshetov et al., 2005] proofed that testing only the four bounding rays can lead to errors and therefore extended the approach by testing a whole frustum instead, hence the name frustum culling. This basic concept was also extended to grids [Wald et al., 2006] and bounding volume hierarchies [Wald et al., 2007]. These approaches are optimized in finding a high amount parallel rays but still require the usual ray sampling similar to single-ray tracing for each packet. Also the amount of parallelism is strongly depending on the scene.

Instead of explicitly searching for parallel rays, approximate voxel cone tracing [Crassin et al., 2011] assumes a certain amount of parallelism. The approach is similar to [Amanatides, 1984] as it uses cones to approximate a bundle of rays but instead of accurately intersecting the cones with the geometry it uses a precomputed approximation of the scene section at each step of the sampling process. Although this simplification is not as accurate as single ray casting techniques, it requires only 5 to 6 large cones to produces visually plausible results instead of millions of rays which tremendously improves performance.

5.2 Cone Tracing

Approximate voxel cone tracing after [Crassin et al., 2011] was implemented to gather the required opacity values for ambient occlusion. Every rendered fragment launches a number of cones arranged in the hemisphere around the surface normal. Each cone is described by the origin, an opening angle and a direction. The origin of the cones is located at the current surface point of the geometric object in world-space, translated by an offset in normal direction to avoid self occlusion. From there on the cone tracing process involves stepping along the cone axis while accumulating the opacity values. Each step requires a traversal of the sparse voxel octree to fetch the queried value. The cone diameter at the current sampling step determines the required level-of-detail, hence the octree depth to obtain the best approximation from. Given the diameter d the level is calculated by

$$l = \log_2(\frac{1}{d})$$

If the value lies between two octree levels, quadrilinear interpolation is provided by including the coarser resolution. The sampled values of both levels are linearly interpolated using the fraction of l as weight.

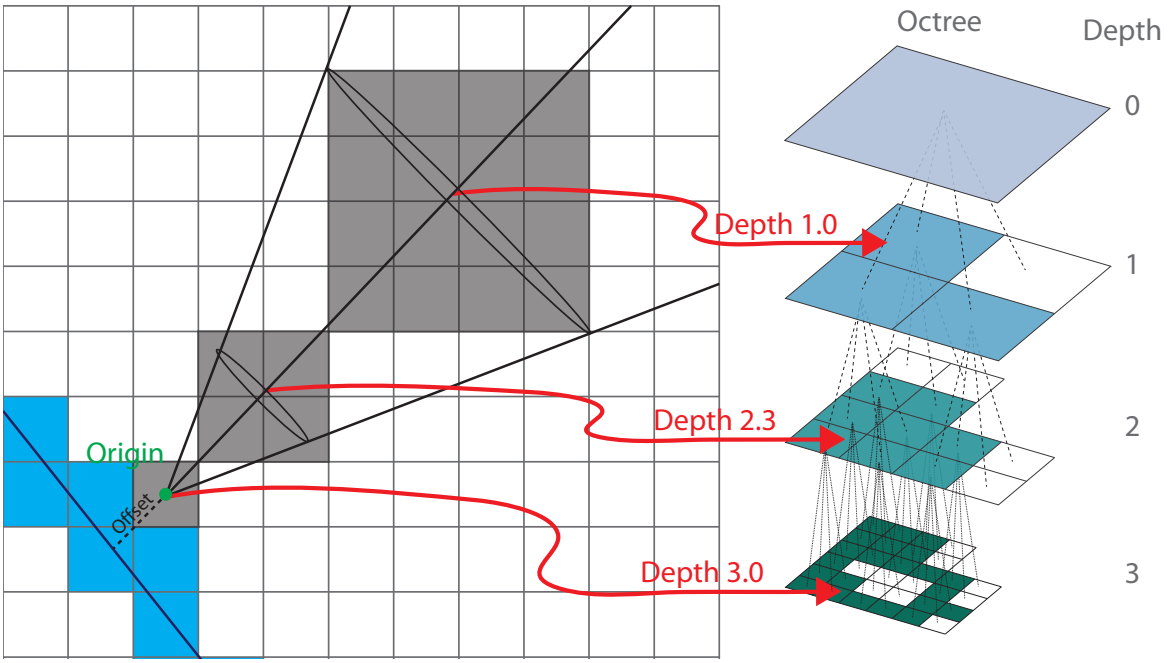


Figure 5.1: Illustration of the cone tracing process

The opacity α is updated by the classical volumetric front-to-back accumulation:

$$\alpha = \alpha + (1 - \alpha) \cdot \alpha'$$

where α' denotes the current queried value. To simulate the decreasing influence of distant objects the opacity α' in each sample step is weighted with a function $f(r)$ which decays with the distance r. For this work $f(r) = \frac{1}{1+\lambda r}$ was chosen, where λ denotes a predefined value which determines the speed of the distance falloff. The distance to the next sampling point is 1.5 times the length of the cell diagonal at the current level-of-detail (Figure 5.1). This distance is sufficient large to avoid redundant sampling due to overlapping cells in different octree levels. If a sampling point corresponds to an empty cell, the sparse section will be skipped and the location is translated to the next filled space. The cone tracing process is terminated if either the opacity value equals one (total occlusion) or a predefined maximal distance is reached.

Like previously mentioned, to gather the occlusion of the hemisphere around the surface point only a few large cones are necessary. Figure 5.2 presents three example cone setups, each providing different sample locations in the hemisphere. The final ambient occlusion for the fragment is calculated by averaging the contribution of all cones. Furthermore the term can be scaled to achieve the desired intensity of the effect.

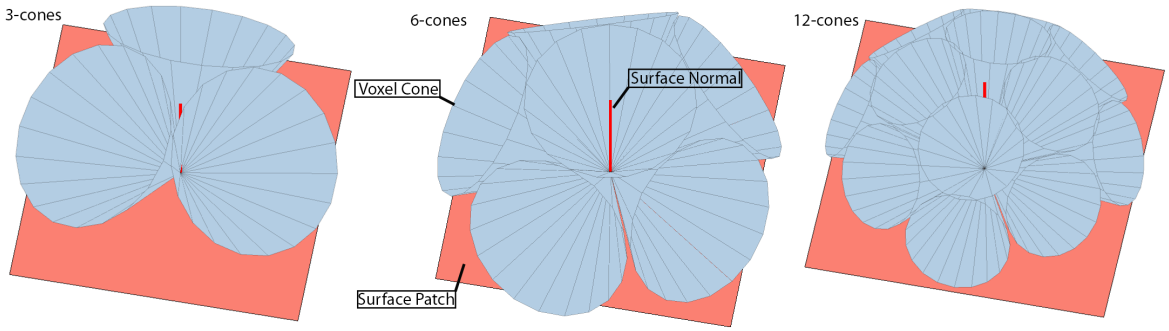


Figure 5.2: Ambient occlusion for a small surface patch with three example cone setups.

5.3 Implementation

The entire ambient occlusion calculation is performed in the fragment shader. Deferred rendering (Section 2.4) is used for efficiency, avoiding the evaluation for hidden geometry. Four different properties are rendered to the G-Buffer in the geometry pass: World position, normal, tangent and bitangent. For the light pass, a scene aligned quad is rendered and the AO computation is executed on each pixel. The data in the G-Buffer contains the information about the origin and orientation of the hemisphere. Consequently, the cone setup can be accurately placed and aligned to the surface patch.

5.3.1 Trilinear interpolation in bricks

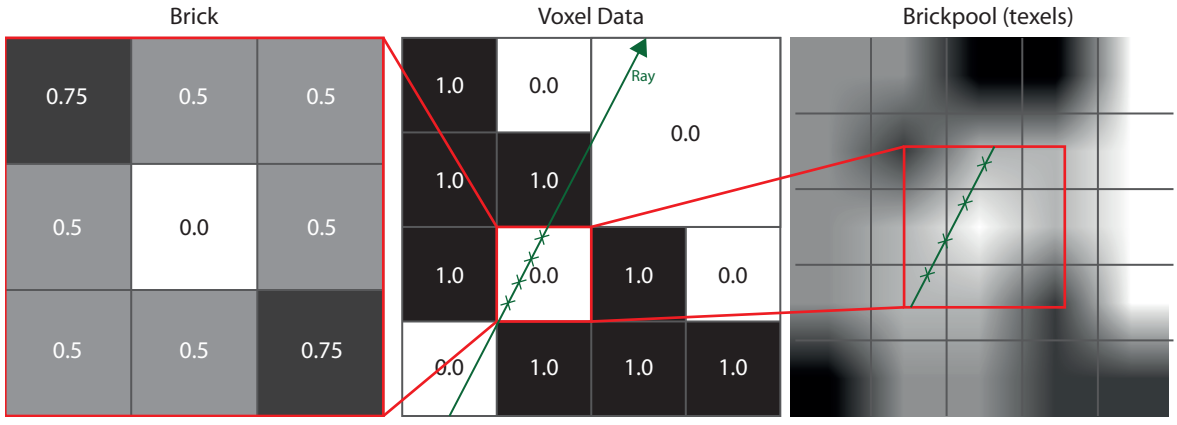


Figure 5.3: Illustration of the brick storage scheme. The initial voxel data (**middle**) with the interpolated adjacent values is stored in a 3x3x3 brick (**left**). Bricks are randomly ordered in a 3D-texture to enable hardware interpolation (**right**).

The cone tracing process relies on a precise scene approximation stored in the leaves and interior nodes. Instead of a single value, a brick scheme (similar to [Christensen and Batali, 2004]) was chosen to represent the scene section with an adequate quality. The bricks consist of 26 border cells and one center cell. The center stores the actual voxel data whereas the border contains linear interpolated values of the corresponding voxel neighbors. This scheme guarantees correctly interpolated values inside octree nodes. The bricks are stored in a dense 3D texture (brickpool) which allows the usage of hardware trilinear interpolation. The coordinate has to be chosen carefully to retrieve the queried values: Adjacent texels are not necessarily spatially associated since bricks are randomly ordered inside the brickpool (Section 4.2.2). The corners of the original voxel corresponds to the centers of the outer brick cells, as shown in Figure5.3.

This technique enables continuous transitions between filled octree cells, unfortunately empty spaces are collapsed in the SVO which leads to missing bricks and therefore discontinuities (details in Section 6.3.1).

5.3.2 Traversing the Octree

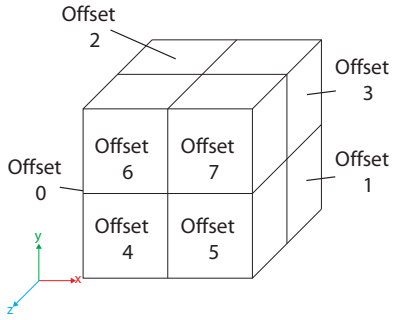


Figure 5.4: Illustration of the children's offsets in the nodepool buffer corresponding to their location.

During the cone tracing process every sample step requires a traversal of the octree to the appropriate level-of-detail. Usually a recursive octree descent is applied, but the packed octree-structure allows a more efficient iterative approach:

The inputs for the algorithm are the location in world-space coordinates and the desired octree depth. The traversal begins at the root and sequentially descends the tree, one level at a time, until the appropriate level-of-detail is reached or a branch is preemptively terminated, due to collapsed empty spaces. The descent is very fast, since the coordinates of a point directly correspond to the node's address. Let $x \in [0,1]^3$ be the location of the point defined in the local coordinate system of the voxel volume's bounding box. The eight children of a node are stored at consecutive memory locations in the nodepool buffer, with the offset scheme shown in Figure 5.4. The offset to the child node containing x can be calculated by $\lfloor x * 2 \rfloor$. The child is fetched at $c + \lfloor x * 2 \rfloor$, where c denotes the nodepool offset of the parent

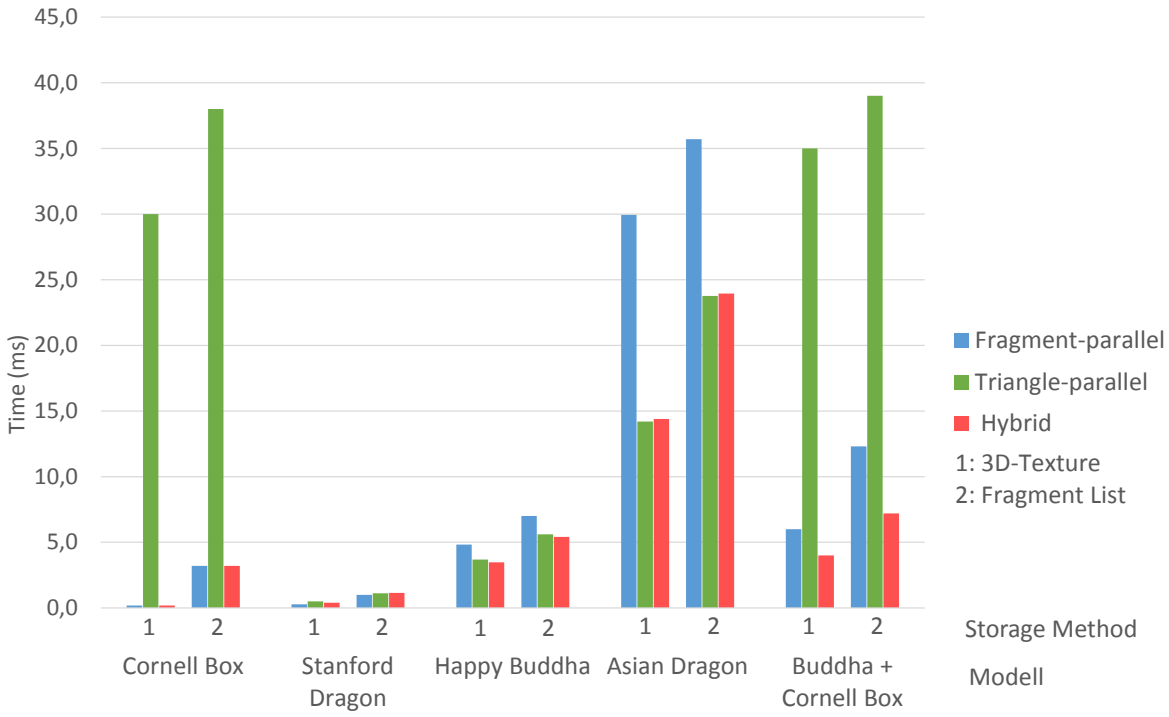
node. The descent is continued by updating x with fract($\lfloor x * 2 \rfloor$). The function *fract* returns the fractional of a floating point value.

Cones send from the same pixel usually follow the same pointers to the desired location, thus the hardware texture cache is optimized to enable fast access to these memory locations. Since addresses are repeatedly fetched from the nodepool the algorithm greatly benefits from *texelfetches* in cached texture memory (Section 2.5).

6 Performance and Results

All results and tests were performed on an Intel Core i5-4670K @3.40Ghz system with an AMD Radeon R9 280x video card.

6.1 Voxelization



	Cornell Box	Stanford Dragon	Happy Buddha	Asian Dragon	Combined
Number of Triangles	36	~80 k	~1.1 mio.	~7.2 mio.	~1.1 mio.

Figure 6.1: Computation time for binary voxelization with a 512x512x512 voxel grid. The timings were separately recorded for full and sparse storage: The full storage is performed in a single-component dense 3D-texture. The sparse storage utilizes the voxel fragment list.

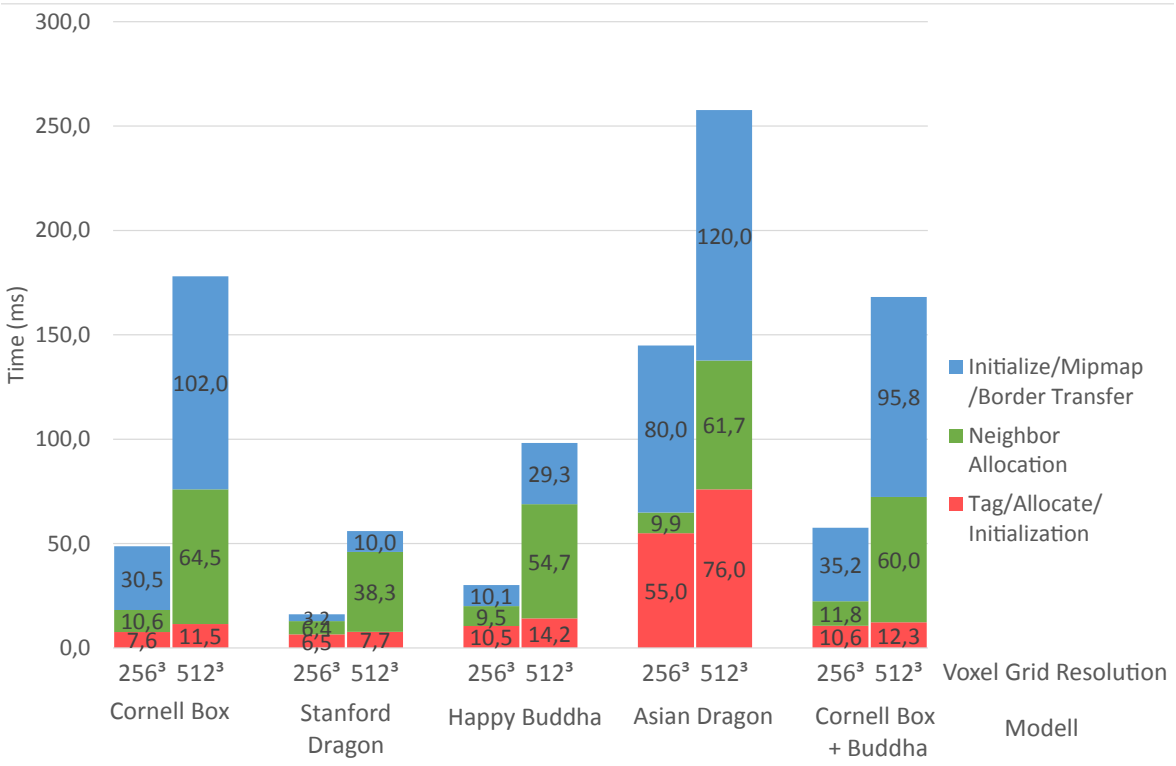
All three voxelization techniques were tested against several models using a voxel grid resolutions of 512x512x512. The bechmark scenes intentionally include best- and worst-case tests for the fragment-parallel and the triangle-parallel technique: As a pathological worst-case scenario for the triangle-parallel voxelization the Cornell Box scene was tested, which

Figure 6.2: Binary voxelization of the XYZ RGB Asian Dragon with two different voxel grid resolutions

solely consists of 36 large triangles, some of them spanning across the whole scene. For triangle-parallel voxelization, these conditions cause single threads to iterate over 512x512 voxel and perform the triangle-box overlap test, which leads to poor performance. In contrast, a dense mesh like the XYZ RGB Asian Dragon (Figure 6.2) with it's huge amount of small, evenly tesselated triangles causes the fragment-parallel voxelization to produce a large amount of unnecessary fragments due to the enlargement of triangles for conservative voxelization. To simulate a mixed scenario a combined scene was constructed, which includes the Cornell Box scene and the Happy Buddha statue. This scene features a large distribution of triangle sizes.

The prime examples confirmed the strengths and weaknesses for each approach as shown in Figure 6.1. The triangle-parallel approach is nearly twice as fast as the fragment-parallel technique in voxelizing the Asian Dragon. The hybrid voxelization performs better than both of it's competitors in the combined scene and the Buddha statue which consist of a great amount of triangles with a large distribution of size. The remaining examples which are the Cornell Box and the Stanford Dragon feature a rather small amount of large triangles. This leads to top speed of the fragment-parallel voxelization, due to the insignificant overhead from triangle enlargement. Although it is is not the best choice in all scenes, the time difference between the hybrid voxelization and the best performer was always within 3%, which indicates the low overhead of this approach.

Regarding the storage technique, inserting data in the voxel fragment list is slower than in a dense texture, because of the involvement of global atomic counters. Their incrementation directly impacts the performance if many threads attempt to access it concurrently. Unfortunately, there is no way to increase the performance or influence the behavior of atomic counters, since the entire voxelization algorithm is implemented in the rendering pipeline.



	Depth/Resolution	Cornell Box	Stanford Dragon	Happy Buddha	Asian Dragon	Combined
# voxel fragments	9 / 256^3	425 k	~ 254 k	~ 2.1 mio.	~ 9.2 mio.	~ 2 mio.
	10 / 512^3	~ 1.68 mio.	~ 505 k	~3.46 mio.	~11.4 mio.	~ 3.8 mio.
# initialized leaves	9 / 256^3	~ 813 k	~ 80 k	~ 263 k	~ 184 k	~ 877 k
	10 / 512^3	~ 3.06mio	~ 315k	~1.05mio	~743k	~ 3.5 mio.
Sparsity (percentage)	9 / 256^3	95.1%	99.5%	98.6%	98.9%	94.8%
	10 / 512^3	97.7%	99.7%	99.1%	99.3%	97.4%

Figure 6.3: Computation time for octree construction. The table shows crucial data values, which greatly influence performance.

6.2 Octree Construction

Figure 6.3 illustrates the results along with important data which impacts the performance. Aside from the actual computation, the timings include the reinitialization of the associated buffers.

As a matter of fact, resetting buffers has great impact on the performance of each step:

- The construction step (tag/allocate/initialize) initializes the nodepool buffer. Since each voxel fragment has the potential to create a leaf, the size of the buffer is conservatively estimated to have enough space for the worst case. Consequently, models with a high amount voxel fragments consume most time by reinitializing the buffer.
- The neighbor allocation pass utilizes the information from the previous step to allocate the neighbor buffer. The number of existing nodes in the nodepool determines the size, thus the clearing costs.
- Similarly, the mipmapping procedure can accurately initialize the brickpool. However, since one brick holds 27 values for each node, the resetting of a large 3D-texture is still costly.

Apart from the initialization costs, the computation time is mainly influenced by the number of voxel fragments in the fragment list and by the amount of filled leaves and interior nodes, hence the sparsity of the voxelization. As expected, all models feature a huge amount of empty cells, which makes the computation of the SVO worthwhile, in terms of memory consumption. A voxel fragment list stores the voxel data, avoiding the initialization of a dense 3D-texture. Unfortunately this approach allows redundant storage of voxel fragments. Duplicated elements occur if multiple primitives intersect the same voxel. Therefore high resolution models with small triangles like the Asian Dragon or the Buddha statue cause a large amount of overhead. Especially in the construction phase (Tag/Allocation/Initialization) this overhead directly correlates with the computation time, since the tagging process operates fragment-parallel. Hence, the time consumption for coarse geometry, like the Stanford Dragon is insignificant, whereas the tagging of the Asian-Dragon is unnecessary expensive. Noticeable in this context is the Cornell Box scene which computation time is unexpectedly high, despite its low triangle count. The reason for this is the large area these few triangles cover, resulting in a lot of filled leaves. The time consumption is raised, due to an increased amount of threads concurrently accessing the atomic counter for node allocation.

As expected, the neighbor allocation is an expensive task. A huge amount of threads is necessary, each traversing the tree completely to retrieve the location of every potential child. If existent, possible neighbors are allocated by another three tree traversals to the adjacent locations. Unfortunately, the top-down pointer based structure as it is, does not allow an alternative approach. Nevertheless, since border transfers are performed multiple times during the filtering, it is still worthwhile to store the neighbors instead of computing them on the fly. Once the neighbor buffers are completed, the mipmapping procedure is efficient aside from the high initialization costs.

Considering the overall timings, revoxelizing and reconstructing the SVO in every frame and still support interactive frame rates is barely manageable for a voxel grid resolution of 512^3 (octree depth 9). E.g., the Stanford Dragon takes about 70 ms on average from voxelization to complete filtering, without considering the rendering and application overhead. It is worth mentioning, that despite the sparse storage the memory consumption is still significant to store the necessary structures. For the Combined Scene roughly 1GB of video memory is allocated for an octree depth of 9.

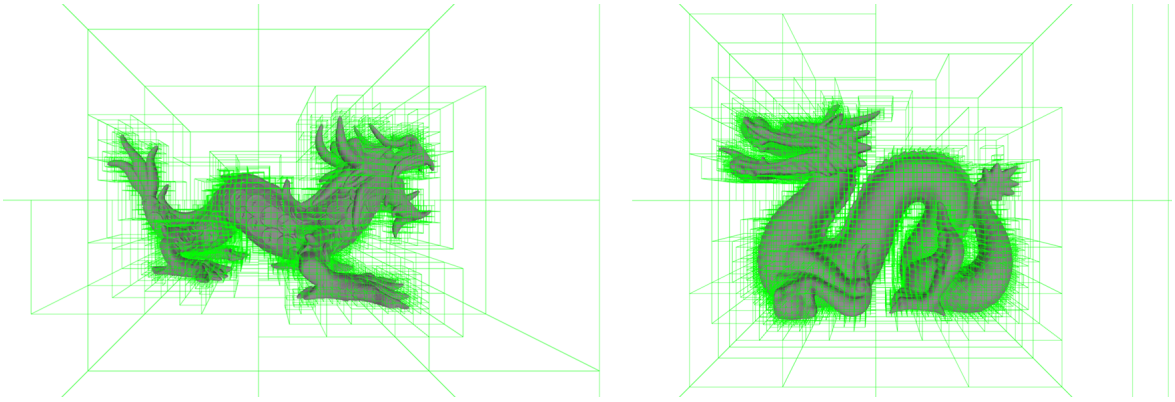


Figure 6.4: Visualization of the sparse voxel octree for the Asian Dragon and Stanford Dragon. Empty cells are collapsed.

6.3 Ambient occlusion using voxel cone tracing

6.3.1 Image quality

The overall image quality is strongly dependent on the variety of settings this approach offers, which are:

- Octree configuration: Depth, voxel grid resolution, sparsity
- Cone constellation: Direction, number of cones and aperture angles.
- Tracing Distance: Maximal Distance, occlusion decay, origins

The following sections will point out the problems which can arise with each setting.

Octree configuration

Figure 6.5: Ambient Occlusion results for the Asian Dragon with two different voxel grid resolutions.

The sparse voxel octree is used for a fast computation of the occlusion value. Its depth and associated voxel grid resolution is therefore the determining factor for the quality of the overall results. The requirement for finer voxels however is depending on the richness of detail of the geometric object itself and the scale of the scene. As long as the geometry is oversampled by voxelization no details are lost. I.e. a simple model which does not feature fine small scale curvature (e.g. Stanford Dragon) is quickly saturated by a specific voxel size, hence a higher resolution is unnecessary. However if the geometry is undersampled which means the voxel are considerably larger than the details, the approximation loses accuracy. Consequently, a large scene with very fine details requires a tremendous amount of voxel to contain all information. In regard of memory limits and computation time a maximal resolution of 512x512x512 is supported. Fortunately this resolution is sufficient for most of the tested scenes. Figure **6.5** compares two different voxel grid resolutions for the dense and high detailed Asian Dragon model. Whereas the coarse resolution barely provides enough detail to calculate ambient occlusion for the large head area. The fine resolution results put emphasis even on fine details, like the small wrinkles on the dragon's scales.

In dynamic scenes, a coarser voxel resolution can cause jittering artifacts when the voxelization and the SVO is updated. This effect occurs, because small scale movements can not be continuously represented by the discrete voxelization. Instead, changes appear in discrete steps, more precisely, when the mesh has moved a greater distance than the world-space size of a voxel. These irregularities cause fluctuating ambient occlusion values between frames, which results in flickering of single pixels. The SVO is especially prone to voxelization updates since small changes can cause significant consequences by collapsing or creating new branches. A higher voxelization precision minimizes these errors.

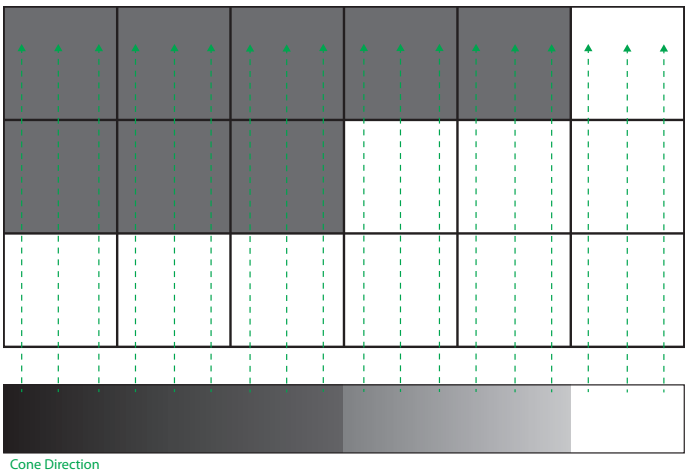


Figure 6.6: A voxelization causing artifacts.The green arrows illustrate the coherent cone directions and the bottom section represents the accumulated opacity.

Although the sparse storage of data is the key to enable high resolution voxelization, it is also the cause of systematic errors (Figure 6.7). These quad-like artifacts occur around sharp edges and small details, which is a consequence of the brick storage scheme. In general they appear where filled and sparse voxel cells are adjacent to another. The reason for this is that bricks enable trilinear interpolation between voxel cells. Since they are only existent in non-empty octree nodes, the transition between adjacent cones which cross the border between filled and sparse space is discontinuous. The effect becomes more significant if the directions of multiple neighboring cones are coherent and completely aligned with this border (Figure 6.6). To handle the problem the coherent alignment of adjacent cones must be avoided. A solution is presented in the following section.

Cone constellation

The calculation of the final occlusion value breaks down to the accumulation of the opacity values during the cone tracing process, therefore the initial constellation of the cones has great impact on the rendering result. The constellation of the cones is defined by the number of cones, their direction and their aperture angles. The three settings together define the

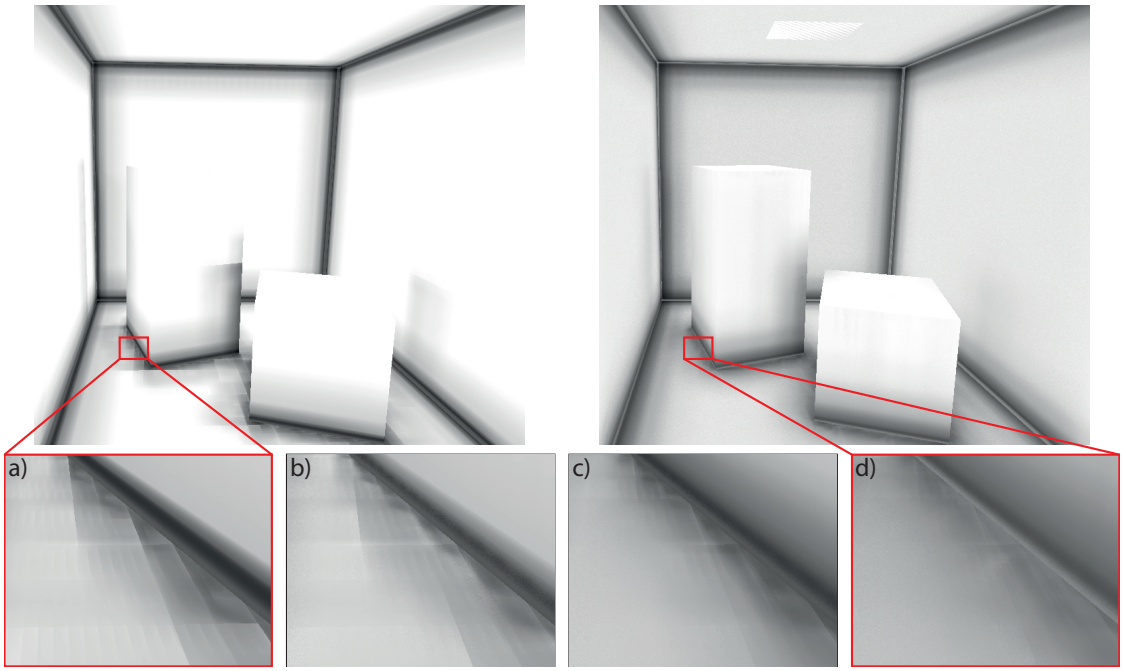


Figure 6.7: Left image: Ambient occlusion in the Cornell Box scene rendered with the 5-cones. (**a**) Quad-like artifacts caused by adjacent filled and empty voxel cells. (**b**) Blur by jittering of the cone directions. (**c**) Reduction by overlapping cones**.** (**d**) Amplify by using more cones. **Right image:** Scene after the combined reduction techniques sampled with eight cones.

covered volume of the hemisphere around the current surface point. The number of cones determines their individual contribution to the total occlusion amount. A higher number results in a better coverage of the volume and therefore to more precise results, but also leads to expensive computation. Associated with the amount of cones is their appropriate aperture angle which determines the cone radius whilst sampling along the cone axis during the tracing process. Usually, redundancy of sampled areas due to overlapping cones has to be avoided, hence a higher number of cones leads to smaller opening angles.

The last setting is the cone direction, the goal is to provide directions which are evenly distributed in the hemisphere around the surface point. For quality evaluation two different cone constellations were tested. The first one consisting of five cones and the second one is a more accurate approximation composed of eight cones. The predefined cone constellations are aligned towards the surface normal, tangent and bi-tangent. Adjacent pixels with similar properties launch the same cones, only with different origins. This leads to the previously mentioned systematic errors around sharp edges (Figure 6.7a). These artifacts can be smoothed out by slightly modifying the predefined constellations:

Local jittering of cones: To completely avoid coherent cones in adjacent pixels, the predefined cone directions are altered by pseudo random noise.

The utilize noise function transforms a 2D coordinate into random numbers. To guarantee consistency between frames, the function is evaluated with the current texture coordinate of a scene aligned quad. The result is a constant blur in strongly occluded areas, which smoothens out erroneous regions (Figure 6.7b). Since the dislocation is very small, the precision loss is minor.

Overlapping cones: In some cases it is beneficial if areas are sampled multiple times from different directions. Especially the affected transition between sparse and filled cells becomes continuous if neighboring cones sample the same space at different positions (Figure 6.7c). To achieve this, the opening angle of the cones is increased until they start to overlap. Furthermore, the smoothness gain of this solution scales with the number of cones, since more regions are overlapping (Figure 6.7c). Although the increased opening angles cause inaccuracies by redundant sampling, the loss in precision is negligible since ambient occlusion is merely a soft shadow approximation and visually plausibility has priority.

Tracing Distance

The distance settings determines which geometry in the scene is able to influence the occlusion on a given surface point and how. The length of the cones represent the radius of the sampling hemisphere. Changing it does not have impact on the quality directly, but it is a source for errors and artifacts if not chosen carefully. The cone is sampled in single steps, which means that longer cones require more sample points. Longer traveling distances lead to larger cone radii resulting in coarse mipmap look-ups, where details (i.e. thin walls) are filtered out. Consequently, it is unlikely that long wide cones will ever reach total occlusion the further they travel. This problem can lead to light leaking artifacts (6.8b), due to the utilization of surface voxelization which discretizes only the contours of geometric objects not their interior. Fortunately, to achieve plausible ambient occlusion long tracing distances are unnecessary, since distant objects have less to no impact on the occlusion, due to occlusion decay.

Another source for errors is determination of the cone origins. To prevent banding artifacts caused by self occlusion (Figure 6.8a), a sufficiently large offset in normal direction is applied to the origins. 1.5 times the length of the voxel diagonal (finest resolution) is sufficient to avoid early cone termination by self occlusion. However, this simple scheme is problematic in small concave areas as they can be accidentally be skipped (Figure 6.8c). Also, moving the origin of all cones is equivalent to moving the original sampling hemisphere away from the actual surface. The consequence is light leaking through very close surfaces. As a matter of fact, it is difficult to find a general solution with the given cone tracing technique. These problems require alternative techniques to fall back when needed, Section **7** provides possible improvements.

6.3.1.1 Comparison

To evaluate the image quality the results of two different cone setups were compared to a reference image computed by Mitsuba renderer [Jakob, 2010] (Figure 6.9). 1000 samples

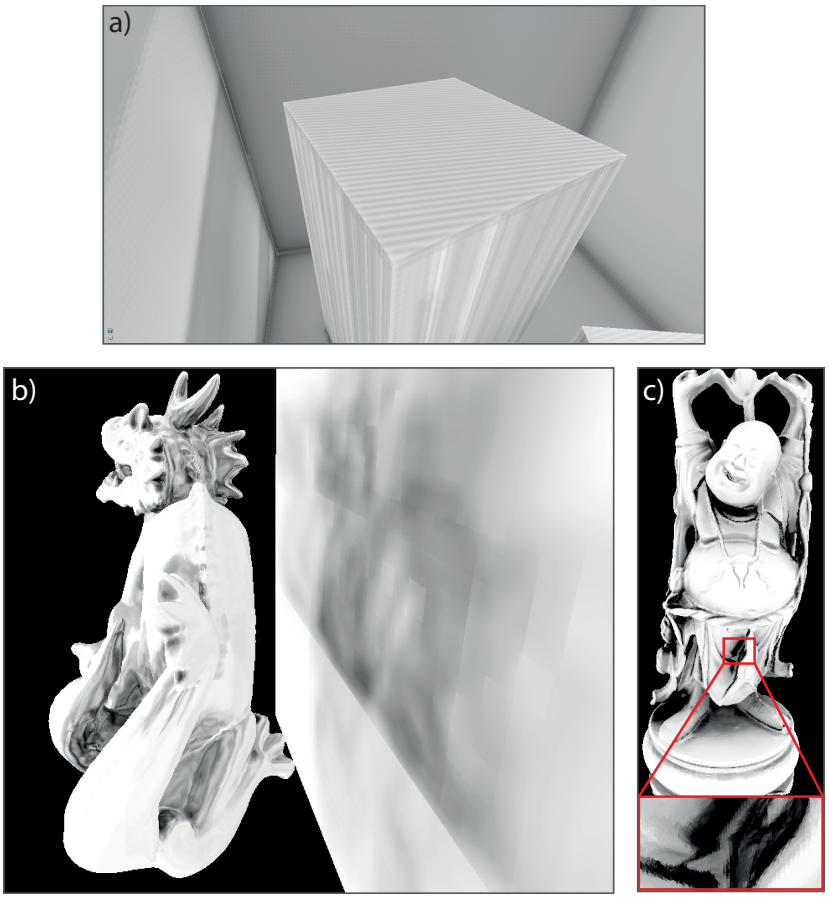


Figure 6.8: (**a)** Banding artifacts caused by self occlusion. (**b**) Silhouette artifacts caused by tracing long cones with a surface voxelization. (**c**) Light leakage artifacts in concave areas.

per ray were used in the ray tracing application, which took about 3 minutes to render a single image on the CPU. The cones/rays were traced for 0.3 of the length of the models diagonal. The 5-cone constellation feature an opening angle of 90 degrees and the 8-cone constellation 60 degrees.

Both cone setups produce visually plausible results compared to the reference but still suffer from the above mentioned artifacts: The 5-cone setup exhibits quad-like artifacts near the edges of the walls and boxes. These artifacts are smoothed out in the 8-cone constellation, but instead banding artifacts from self occlusion are visible at both of the boxes.

Apart from these small errors, the image quality is still satisfying considering the low rendering time of 19ms and 27ms (details in 6.3.2) for five and eight cones with an octree depth of 9.

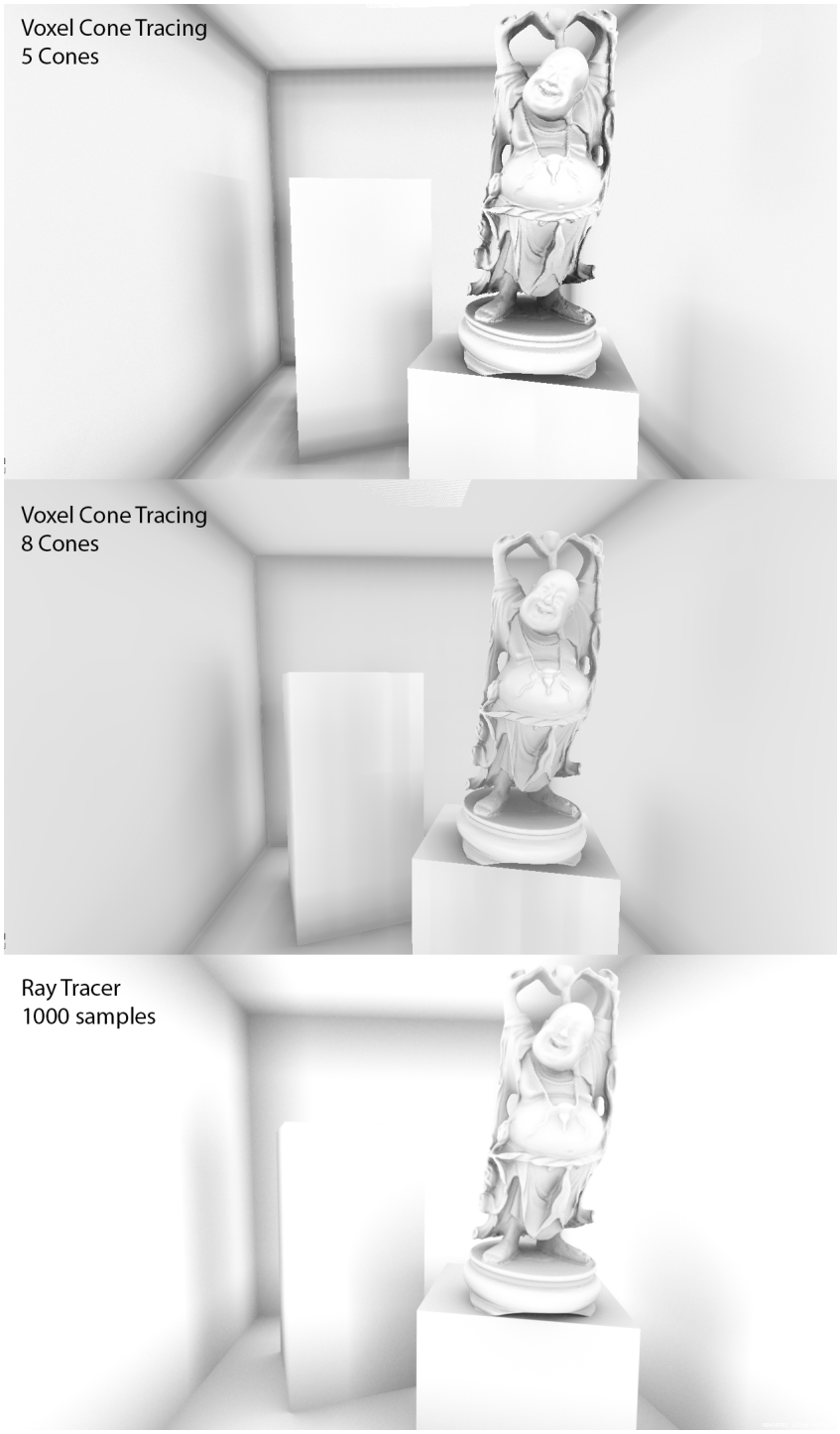


Figure 6.9: Comparison of the voxel cone tracing method with two different cone setups and an offline ray tracer.

6.3.2 Performance

This section is divided into two parts. The first part covers the performance results of the cone tracing process in separate. The second part provides an evaluation of the approach in full dynamic scenes, by benchmarking the whole pipeline from voxelization to the final rendering.

Voxel Cone Tracing

Measuring the performance of the cone tracing process is a difficult task due to its variety of settings. The options were limited to three cone constellation: 3-cones, 5-cones and 8-cones. The opening angles and maximal distances were chosen appropriately with the focus on maximizing image quality. The results also cover timings for different octree depths, including 7, 8 and 9 levels with their associated voxel grid resolutions of 128^3, 256^3 and 512^3.

The test scene is a combination of the Cornell Box scene and the Happy Buddha Statue (Figure 6.9). The scene was chosen because of its features which are:

- Full coverage of the voxelization volume
- A realistic ratio of filled to sparse sections
- Large triangles describing the outer walls of a room (36 triangles) and dense detail in the center ($\sim$ 1 mio. triangles), representing a typical interior scenery

A resolution of 1280x720 is used for rendering. The cone tracing process is applied to each visible fragment, therefore the absolute screen coverage of the geometry is crucial as it determines the total amount of cones send to the scene. To push the limit, the scene was captured with full screen coverage, effectively resulting in close to 1 mio. fragments.

The final results are presented in Figure 6.10. The deferred shading pass was included to the measurements as it is directly involved in the rendering process. The performance of this rendering pass is solely dependent on the input geometry, more precisely on the number of triangles and amount of generated fragments. Since the rendered geometry is constant in each testing scenario, the timing is also consistent. To potentially save performance an deferred rendering was tested which captures the geometry information in half screen resolution and uses texture interpolation to retrieve values in full screen resolution during the light pass. Unfortunately, the performance gain was minor compared to the quality loss, so the technique was not further evaluated.

The voxel cone tracing pass greatly benefits from the precomputed sparse voxel octree. Increasing the octree depth only leads to minor performance losses since the traversal is very efficient. However, the performance drop is more prominent when a higher number of cones with smaller opening angles is used. The reason for this is the increased amount of lookups

in deeper octree levels. Consequently, the three and five cone constellations perform better than the eight cone setup in higher voxel resolutions.

Aside from the voxel resolution, the quantity of cones is also a crucial performance factor. Raising the amount of cones directly influences the number of tracing operations, hence increases computation time.

In summary, including the application overhead, the rendering of the static benchmark scene with a 5-cone constellation results in an average of 60 fps for 128^3, 52 fps for 256^3 and 45 fps for 512^3 voxels.

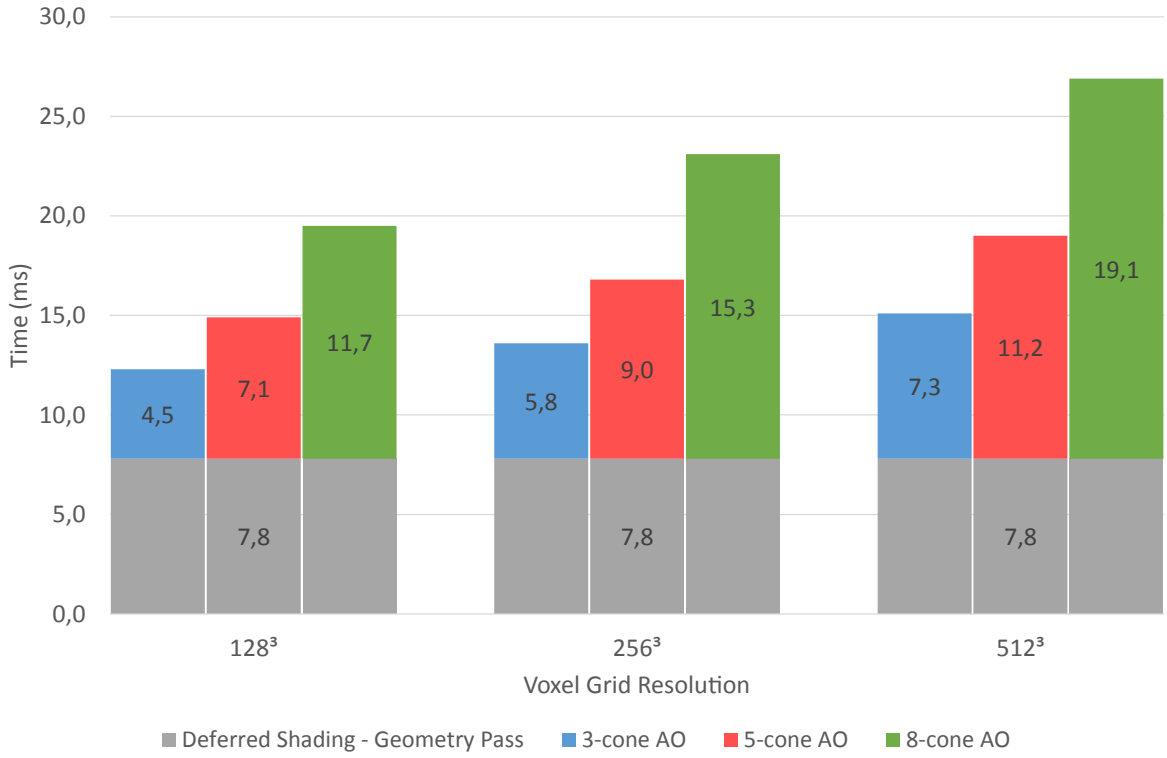


Figure 6.10: Performance measurements for three different voxel grid resolutions and three different cone constellations, including the deferred shading pass.

6.3.2.1 Full dynamic Scenes

As previously mentioned the cone tracing process is very efficient since it relies on precomputed structures. For dynamic scenes these structures have to maintained or reconstructed periodically. Updating an existent sparse voxel octree is a complex task, as it includes managing and sorting of time stamps over multiple frames. This task becomes more expensive as the amount of dynamic elements increases. Since the primary objective is illuminating full dynamic scenes this strategy is more expensive than revoxelizing and reconstructing the

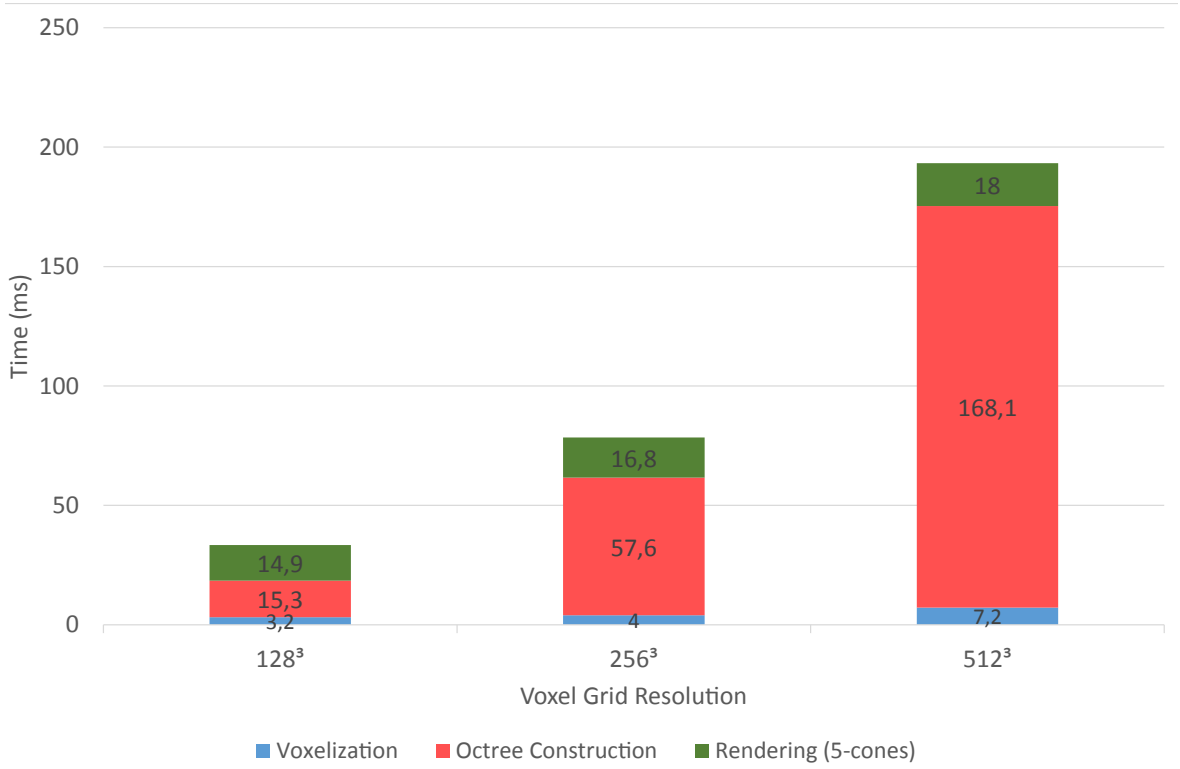


Figure 6.11: Comparison of the timings for each pipeline step.

whole scene when an update is due. Unfortunately, with increasing voxel grid resolution the computation demands of the octree reconstruction becomes prominent (Figure 6.11). Its disproportionate behavior turns out to be the determining factor for the final performance. Rendering the test scene with 5-cones while revoxelizing and reconstructing the sparse voxel octree in every frame limits the average frame rate to 11FPS, 6FPS and 3FPS for the voxel resolutions 128^3, 256^3 and 512^3.

Optimization

Fortunately, executing the rebuilding process in every frame is unnecessary since dynamic changes rarely occur at such frame rates. This fact can be exploited to increase performance. Instead of revoxelizing and reconstructing the SVO frame-wise, an update frequency is determined which is independent of the rendering loop. The rate of updates corresponds to the precision requirements of the dynamic simulation. Although this strategy effectively increases the average frame rate, the reconstruction time steps take significantly longer than the remaining ones (Figure 6.12 top). The outcome are strongly fluctuating FPS.

As a solution a pipeline strategy was applied to evenly distribute the workload: The octree construction process was developed in regard of optimized thread balancing, therefore it consists of various low computation steps (E.g. 54 steps for an octree of depth 8). These

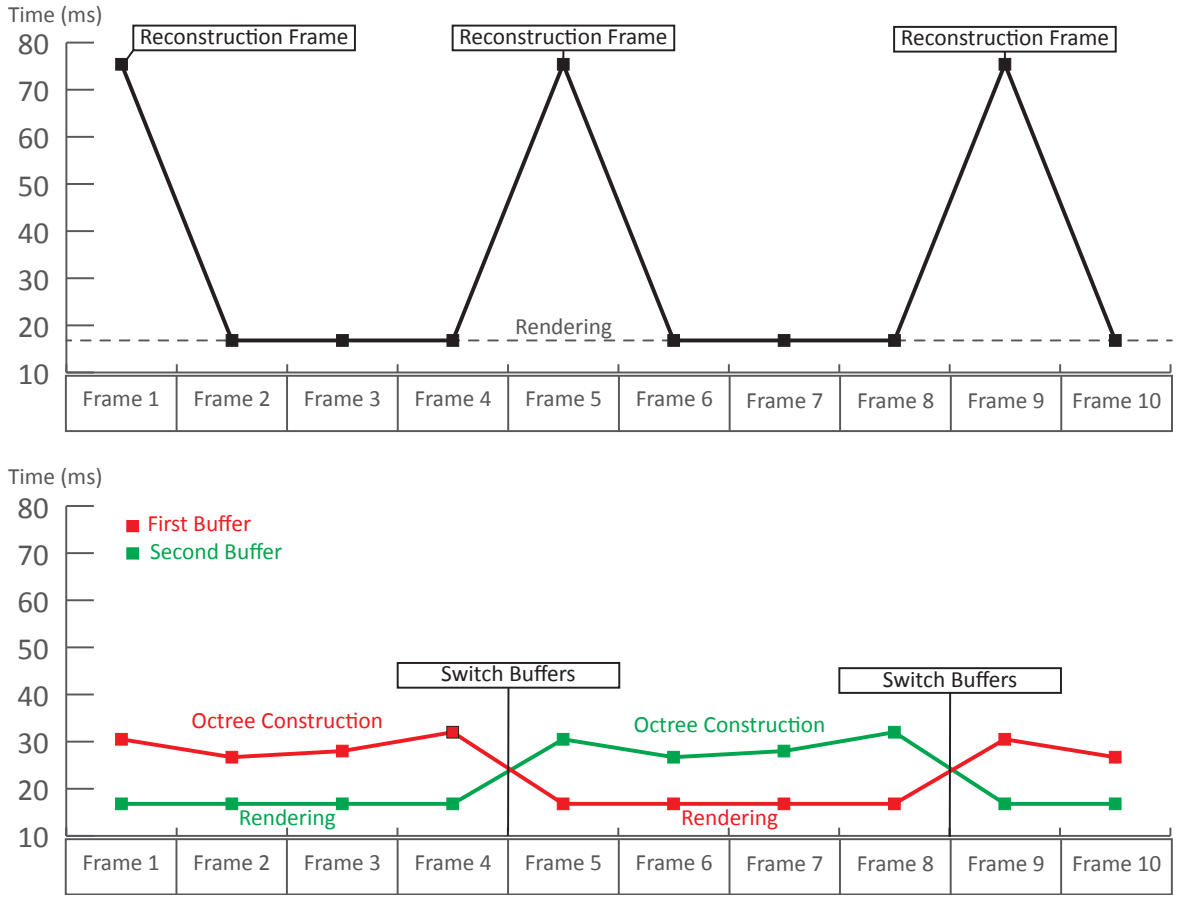


Figure 6.12: Comparison of simple frame skipping and double octree buffering.

passes are bundled to packets with similar computation time and distributed over multiple frames (Figure 6.12 bottom). As soon as the new SVO is ready the data is transferred to the rendering pipeline and a new construction process is initiated. To guarantee consistent the octree data for the rendering is updated via double buffering. By using this method the average frame rate of the complete process increased from 6 FPS to 24 FPS for a 256^3 voxel resolution.

The drawback of this technique is the increased memory consumption, since double buffering requires redundant storage of the data structures. Also, regarding the strong dependence on the input data, it is difficult to find a general scheme to evenly distribute the workload of the construction process.

7 Conclusion and Future Work

This chapter recapitalizes the work of this thesis by summarizing the applied concepts and techniques and concludes the approach under consideration of the result chapter. The final section proposes possible improvements and enhancements.

7.1 Summary

Ambient occlusion is a small scale shadowing term which enhances the visual quality by increasing the perception of shape and depth and is therefore a highly desired optical feature in interactive rendering applications. Non-local solutions are capable of computing high-quality AO. Unfortunately they are usually too expensive to provide real-time frame rates in dynamic scenes. This work revisited approximate voxel cone tracing, a non-local approach proposed by [Crassin et al., 2011]. The main idea of this technique is to replace millions of spatially and directionally coherent rays by a few large cones, thus dramatically reducing rendering time. The cone tracing process heavily relies on a precomputed sparse voxel octree to fetch data from. The prerequisite for the octree construction is an accurate surface voxelization of the scene.

Three different voxelization approaches were implemented and evaluated: The first method relied on the highly optimized hardware rasterizer and is therefore called fragment-parallel voxelization. The second voxelization technique completely avoids the rasterizer. Instead, the algorithm tests each triangle for voxel intersections, hence the name triangle-parallel voxelization. Both approaches have their individual drawbacks depending on the geometric properties of the input data. Therefore, as a third approach, a hybrid pipeline was introduced which combines the advantages of both techniques. Following the observation that a voxelization of a typical scene mainly consists of empty cells, filled voxel were stored in a linear buffer (voxel fragment list) instead of a dense 3D-texture to save memory.

After the voxelization a sparse voxel octree structure after [Crassin et al., 2009] was implemented. The structure is memory efficient and provides prefiltered data in the interior nodes. To maintain an accurate scene representation despite sparsity, a brick scheme was applied for leaves and interior nodes. The construction process was divided into multiple passes to enable highly parallel computation.

Finally, to compute ambient occlusion the cone tracing approach after [Crassin et al., 2011] was implemented. The technique uses a predefined cone setup to integrate the opacity values in the hemisphere around each surface point. Each cone is sampled along the axis to gather the opacity values stored in the SVO. The cone diameter at each sampling step is used to determine the required level-of-detail, hence the octree level to fetch the scene approximation from.

7.2 Conclusion

The tests showed that approximate voxel cone tracing is able to produce high quality ambient occlusion. The results were comparable to offline physically based ray tracers. However, further analysis exposed weaknesses of the approach. On the one hand the empty regions in the SVO led to quad-like artifacts due to discontinuities between filled nodes. On the other hand the discrete voxel approximation of the geometry cause light leaking artifacts in concave areas. Also, in the original work of [Crassin et al., 2011] the cone constellation and opening angles was not a subject of discussion, although it can be the cause of self-occlusion and silhouette artifacts. Solutions to prevent this artifacts like local cone jittering or overlapping of the cones were able to reduce the errors but not completely eradicate them.

The performance of the cone tracing process was sufficient to support interactive frame rates in high detailed static scenes, since the gathering process relies on precomputed acceleration data structures. Unfortunately in dynamic scenes these structures have to be updated periodically. The timing results revealed that the reconstruction of the scene approximations is the determining performance factor. The original work used a fragment-parallel technique to voxelize the scene in real-time [Crassin and Green, 2012]. In this work the same approach was implemented and combined with a triangle-based method which together formed a hybrid voxelization pipeline. The benchmark tests have shown that the technique performs better in scenes with arbitrary geometric characteristics. However, even with this optimization it was barely manageable to maintain interactive frame rates. The reason for this was the reconstruction of the sparse voxel octree. The application overhead, due to the reinitialization of the data structure was immense. Moreover, the main drawback of the construction was the strong dependency on the input geometry. The results of the performance checks revealed that the process is strongly influenced by the amount of empty spaces in the scene as well as the acuteness of the triangulation of the surfaces. Both of these characteristics had great impact on the amount of threads which were needed in each construction pass. Higher voxel resolutions amplified the performance loss.

Another factor is that the used SVO structure is not well suited for the applied filtering procedure since cell neighbors are required in every step. Unfortunately the octree data structure only stores child pointers to save memory. The position of a child is defined relative to its parent which means the tree must be traversed from the root to acquire the world-space position of a child. This circumstance makes the determination of neighboring nodes in the octree a computationally expensive procedure. Although Crassin et al. utilized precomputed neighbors for filtering, the neighbor search was not further detailed in the original work. However, with basic knowledge about the scene the computation of the SVO can be be divided into smaller packets and distributed over several frames, which tripled the average FPS in benchmark tests. The drawback of this pipeline strategy is a loss in simulation precision since dynamic changes are updated with a lower frequency and higher memory requirements due to double buffering.

In summary, although the image quality is sufficient, the limitations of the required preprocessing stage are overwhelming which makes the implemented technique not well suited for arbitrary dynamic scenes.

7.3 Future work

7.3.1 Voxelization

For the voxelization of dynamic scenes the proposed techniques prioritize high performance and low memory requirements over quality which led to errors in the final results. The implemented surface voxelizations are sufficient to compute ambient occlusion but can cause light leaking artifacts by accidentally stepping over thin walls of geometric objects. To prevent these errors the techniques can be enhanced to support solid voxelization: [Schwarz and Seidel, 2010] and [Zhang et al., 2007] proposed extensions for triangle-parallel as well as fragment-parallel approaches. An important limitation of solid voxelization is the requirement for closed, watertight meshes to properly determine the inside and outside of a geometric object.

To reduce self occlusion artifacts a view-dependent voxelization scheme can be applied similar to [Gobbetti and Marton, 2005]. The result are anisotropic voxel which store opacity values for each major direction. During the cone tracing process this additional data provides information about the orientation of the original faces which can be used to exclude voxel, depending on the cone direction.

Another problem of any voxelization technique is memory efficient storage. The voxel fragment list is a valid technique for that matter but as the results have shown, suffers from redundant storage of voxel fragments. Future implementation can benefit from AMDs active research in the field of partially resident textures [Sellers et al., 2012]. The idea of these sparse textures is to separate the physical texture storage from the GPU texture address space, similar to classical caching. Consequently, sparsity can be exploited by setting up sufficiently large virtual GPU space, which empty regions are not backed by physical data. Although this technique is promising for sparse storage, the current implementation in OpenGL (*AMD_sparse_texture*) allows only addressing of 64kb tiles (minimum) in physical memory. For a typical 32-bit 3D-texture 64kb corresponds to 64x64x64 elements which is too coarse to sufficiently describe empty spaces in a voxelization. Hopefully, future releases enable a finer tile granularity.

Very recently Nvidia released a new GPU architecture [NVIDIA, 2014] which supports hardware accelerated conservative rasterization (OpenGL extension *NV_conservative_raster*). With this feature the manual expansion of the triangles in the geometry shader becomes unnecessary. Hence, no additional fragments are created which dramatically increases performance of fragment-parallel voxelization.

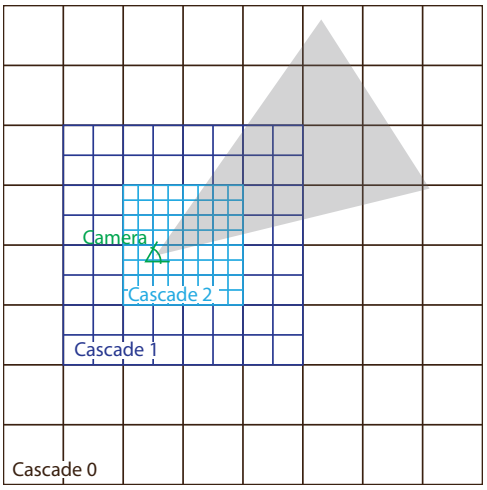


Figure 7.1: Illustration of a voxel texture cascade.

7.3.2 Sparse Voxel Representation

As the results have shown the implemented sparse voxel octree is not well suited for frequent reconstruction. Also the performance of an update strategy implemented in [Crassin et al., 2011] is strongly dependent on the amount of dynamic parts of the scene as the authors described in their results. Therefore, [McLaren and Q-Games, 2015] very recently proposed an alternative acceleration structure which is well suited for voxel cone tracing, called *voxel texture cascade*. In contrast to sparse voxel octrees which utilizes the sparsity of the scene, the idea behind voxel texture cascades is to save memory and computation time by limiting the level-of-detail of the geometry depending on the camera's focal point. The voxel data is stored in multiple overlapping 3D textures (low resolution) with each level being the same resolution, but the dimensions of the area they cover doubles with each successive level (Figure 7.1). Each cascade can be updated separately depending on the movements of the geometric object and the camera. The consequential accuracy loss is barely visible, as long as the finest cascade is updated more frequently. This structure is nearly independent of the input data and very flexible in regard to texture sizes and levels which makes it a potential solution for arbitrary dynamic scenes.

7.3.3 Voxel Cone tracing

The ambient occlusion quality in the benchmark scene was satisfying, with the exception of regions with small scale geometric details. On the one hand, a reason for this is the limitation of detail due to the voxel approximation, on the other hand the offset applied to the cone origins causes small concave areas to be skipped. A solution for this problem is complementing the world-space ambient occlusion term with a screen-space calculation. To prevent overlapping of both ambient occlusion terms the sampling area for SSAO should not

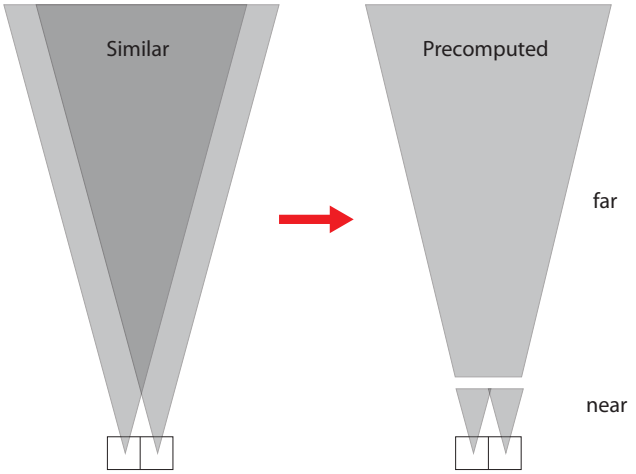


Figure 7.2: The visibility cones of two adjacent pixel become increasingly similar the further they travel.

exceed the world-space dimension of a voxel. As a matter of fact, the small sampling area reduces the systematic errors of SSAO and therefore produces good results in crucial regions.

Regarding performance [McLaren and Q-Games, 2015] proposed an optimization based on the observation that the traversed data of two parallel cones from adjacent pixel becomes increasingly similar while moving further along the cone axis (Figure 7.2). Consequently, this data can be traced once and stored in a 3D-texture. To make use of this data, the tracing cones are split up in a *near* and *far* part. The near part is computed by actually tracing the cone, whearas the far part uses the precomputed data. The ratio of the near to far distance can be adjusted to provide the best balance between quality and speed.

7.3.4 Other Applications

In this work, approximate voxel cone tracing was used to compute ambient occlusion. Aside from gathering opacity values, the technique has the potential to compute other visual phenomena. As a matter of fact [Oliver, 2012] demonstrated a full global illumination model based on voxel cone tracing which supports opaque and translucent materials as well as glossy surfaces. The procedure is very similar to the computation of ambient occlusion, only that the acceleration data structure needs to store additional values in the leaves and interior nodes: Aside from the color albedo and normals, the incoming radiance is injected by rasterizing the scene from the light's perspective and storing the resulting fragment position as photons in the octree cells, along with their angle of incidence. Once the photons of each light source are gathered, the cone tracing procedure as explained in Section 5 is executed, with the difference that the view angle is included in the sampling process to correctly calculate the incoming radiance.

Bibliography

[Akenine-Möller, 2005] Akenine-Möller, T. (2005). Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA. ACM. 11

[Akenine-Möller et al., 2002] Akenine-Möller, T., Möller, T., and Haines, E. (2002). *Real-Time Rendering*. A. K. Peters, Ltd., Natick, MA, USA, 2nd edition. 2, 13

[Amanatides, 1984] Amanatides, J. (1984). Ray tracing with cones. *SIGGRAPH Comput. Graph.*, 18(3):129–135. 29, 30

[AMD, 2012] AMD (2012). Amd accelerated parallel processing opencl programming guide. 9

[Bastos and Filho, 2008] Bastos, T. and Filho, W. C. (2008). Gpu-accelerated adaptively sampled distance fields. In *Shape Modeling International*, pages 171–178. IEEE. 21

[Benson and Davis, 2002] Benson, D. and Davis, J. (2002). Octree textures. *ACM Trans. Graph.*, 21(3):785–790. 21

[Christensen and Batali, 2004] Christensen, P. H. and Batali, D. (2004). An irradiance atlas for global illumination in complex production scenes. In *Proceedings of the Fifteenth Eurographics Conference on Rendering Techniques*, EGSR'04, pages 133–141, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 33

[Crassin and Green, 2012] Crassin, C. and Green, S. (2012). *Octree-Based Sparse Voxelization Using The GPU Hardware Rasterizer*. CRC Press, Patrick Cozzi and Christophe Riccio. 11, 22, 51

[Crassin et al., 2009] Crassin, C., Neyret, F., Lefebvre, S., and Eisemann, E. (2009). Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, I3D '09, pages 15–22, New York, NY, USA. ACM. 3, 4, 21, 50

[Crassin et al., 2011] Crassin, C., Neyret, F., Sainz, M., Green, S., and Eisemann, E. (2011). Interactive indirect illumination using voxel-based cone tracing: An insight. In *ACM SIGGRAPH 2011 Talks*, SIGGRAPH '11, pages 20:1–20:1, New York, NY, USA. ACM. 3, 4, 23, 30, 50, 51, 53

[DeBry et al., 2002] DeBry, D., Gibbs, J., Petty, D. D., and Robins, N. (2002). Painting and rendering textures on unparameterized models. *ACM Trans. Graph.*, 21(3):763–768. 21

[Dmitriev et al., 2004] Dmitriev, K., Seidel, H.-P., and Havran, V. (2004). Faster ray tracing with SIMD shaft culling. Technical Report MPI-I-2004-4-006, Max-Planck Institut für Informatik (Sarrebruck, DE), Saarbrücken. 3, 30

[Dong et al., 2004] Dong, Z., Chen, W., Bao, H., Zhang, H., and Peng, Q. (2004). Real-time voxelization for complex polygonal models. In *Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on*, pages 43–50. 10

[Eisemann and Decoret, 2006] Eisemann, E. and Decoret, X. (2006). Fast scene voxelization and applications. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 71–78. ACM SIGGRAPH. Best paper award - reprised at SIGGRAPH. 11

[Everitt, 2001] Everitt, C. (2001). Interactive order-independent transparency. 10

[Fang et al., 2000] Fang, S., Fang, S., Chen, H., and Chen, H. (2000). Hardware accelerated voxelization. *Computers and Graphics*, 24(3):433–442. 10

[Gobbetti and Marton, 2005] Gobbetti, E. and Marton, F. (2005). Far voxels: A multiresolution framework for interactive rendering of huge complex 3d models on commodity graphics platforms. *ACM Trans. Graph.*, 24(3):878–885. 52

[Gobbetti et al., 2008] Gobbetti, E., Marton, F., and Guitián, J. A. I. (2008). A single-pass gpu ray casting framework for interactive out-of-core rendering of massive volumetric datasets. *The Visual Computer*, 24(7-9):797–806. 21

[Guntury and Narayanan, 2012] Guntury, S. and Narayanan, P. (2012). Raytracing dynamic scenes on the gpu using grids. *Visualization and Computer Graphics, IEEE Transactions on*, 18(1):5–16. 3

[Haines and Wallace, 1991] Haines, E. and Wallace, J. (1991). Shaft culling for efficient ray-traced radiosity. 14

[Hasselgren et al., 2005] Hasselgren, J., Akenine-Mö ller, T., and Ohlsson, L. (2005). *Conservative Rasterization*, pages 677–690. GPU Gems 2. Addison-Wesley Professional. 11, 17

[Heckbert and Hanrahan, 1984] Heckbert, P. S. and Hanrahan, P. (1984). Beam tracing polygonal objects. In *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '84, pages 119–127, New York, NY, USA. ACM. 29

[Jakob, 2010] Jakob, W. (2010). Mitsuba renderer. http://www.mitsuba-renderer.org. 43

[Kay and Kajiya, 1986] Kay, T. L. and Kajiya, J. T. (1986). Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20(4):269–278. 3

[Koonce, 2008] Koonce, R. (2008). Deferred shading in tabula rasa. In Nguyen, H., editor, *GPU Gems 3*, pages 429–457. Addison-Wesley. 9

[Kraus and Ertl, 2002] Kraus, M. and Ertl, T. (2002). Adaptive texture maps. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 7–15, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 21

[Laur and Hanrahan, 1991] Laur, D. and Hanrahan, P. (1991). Hierarchical splatting: A progressive refinement algorithm for volume rendering. *SIGGRAPH Comput. Graph.*, 25(4):285–288. 21

[Lefebvre, 2005] Lefebvre, Hornus, N. (2005). *GPU Gems 2, Chapter 42: Octree textures on the gpu*. Addison-Wesley Professional. 21

[Li et al., 2003] Li, W., Fan, Z., Wei, X., and Kaufman, A. (2003). Gpu-based flow simulation with complex boundaries. Technical report. 10

[Maule et al., 2012] Maule, M., Comba, J., Torchelsen, R., and Bastos, R. (2012). Transparency and anti-aliasing techniques for real-time rendering. In *Graphics, Patterns and Images Tutorials (SIBGRAPI-T), 2012 25th SIBGRAPI Conference on*, pages 50–59. 9

[McLaren and Q-Games, 2015] McLaren, J. and Q-Games (2015). The technology of the tomorrow children. GDC 2015 talk. 53, 54

[Mittring, 2007] Mittring, M. (2007). Finding next gen: Cryengine 2. In *ACM SIGGRAPH 2007 Courses*, SIGGRAPH '07, pages 97–121, New York, NY, USA. ACM. 3

[Nießner et al., 2013] Nießner, M., Zollhöfer, M., Izadi, S., and Stamminger, M. (2013). Real-time 3d reconstruction at scale using voxel hashing. *ACM Trans. Graph.*, 32(6):169:1–169:11. 21

[NVIDIA, 2014] NVIDIA (2014). Cuda toolkit documentation v7.0. 52

[Oliver, 2012] Oliver, P. (2012). Unreal engine 4 elemental. In *ACM SIGGRAPH 2012 Computer Animation Festival*, SIGGRAPH '12, pages 86–86, New York, NY, USA. ACM. 54

[Overbeck et al., 2007] Overbeck, R., Ramamoorthi, R., and Mark, W. R. (2007). A real-time beam tracer with application to exact soft shadows. In *Proceedings of the 18th Eurographics Conference on Rendering Techniques*, EGSR'07, pages 85–98, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 3, 29

[Pantaleoni, 2011] Pantaleoni, J. (2011). Voxelpipe: A programmable pipeline for 3d voxelization. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, HPG '11, pages 99–106, New York, NY, USA. ACM. 11

[Pineda, 1988] Pineda, J. (1988). A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph.*, 22(4):17–20. 7

[Rauwendaal and Bailey, 2013] Rauwendaal, R. and Bailey, M. (2013). Hybrid computational voxelization using the graphics pipeline. *Journal of Computer Graphics Techniques (JCGT)*, 2(1):15–37. 11, 18

[Reshetov et al., 2005] Reshetov, A., Soupikov, A., and Hurley, J. (2005). Multi-level ray tracing algorithm. *ACM Trans. Graph.*, 24(3):1176–1185. 3, 30

[Ritschel et al., 2012] Ritschel, T., Dachsbacher, C., Grosch, T., and Kautz, J. (2012). The state of the art in interactive global illumination. *Comput. Graph. Forum*, 31(1):160–188. 1, 3

[Rosado, 2008] Rosado, G. (2008). Motion blur as a post-processing effect. In Nguyen, H., editor, *GPU Gems 3*, pages 575–581. Addison-Wesley. 9

[Saito and Takahashi, 1990] Saito, T. and Takahashi, T. (1990). An ambient light illumination model. *SIGGRAPH Comput. Graph.*, 24(4):197–206. 8

[Schwarz and Seidel, 2010] Schwarz, M. and Seidel, H.-P. (2010). Fast parallel surface and solid voxelization on gpus. *ACM Trans. Graph.*, 29(6):179:1–179:10. 11, 52

[Sellers et al., 2012] Sellers, G., Obert, J., and van Waveren, J. (2012). Virtual texturing in software and hardware. Siggraph 2012 conference. 52

[Shishkovtsov, 2005] Shishkovtsov, O. (2005). Deferred shading in s.t.a.l.k.e.r. In Pharr, M., editor, *GPU Gems 2*, pages 143–166. Addison-Wesley. 9

[Sung, 1991] Sung (1991). A dda octree traversal algorithm for ray tracing. *Eurographics*, pages 73–85. 3

[Wächter and Keller, 2006] Wächter, C. and Keller, A. (2006). Instant ray tracing: The bounding interval hierarchy. In *Proceedings of the 17th Eurographics Conference on Rendering Techniques*, EGSR '06, pages 139–149, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 3

[Wald, 2004] Wald, I. (2004). *Realtime Ray Tracing and Interactive Global Illumination*. PhD thesis, Computer Graphics Group, Saarland University. 3

[Wald et al., 2007] Wald, I., Boulos, S., and Shirley, P. (2007). Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Trans. Graph.*, 26(1). 30

[Wald and Havran, 2006] Wald, I. and Havran, V. (2006). On building fast kd-trees for ray tracing, and on doing that in o(n log n). In *IN PROCEEDINGS OF THE 2006 IEEE SYMPOSIUM ON INTERACTIVE RAY TRACING*, pages 61–70. 3

[Wald et al., 2006] Wald, I., Ize, T., Kensler, A., Knoll, A., and Parker, S. G. (2006). Ray tracing animated scenes using coherent grid traversal. *ACM Trans. Graph.*, 25(3):485–493. 30

[Wald et al., 2001] Wald, I., Slusallek, P., Benthin, C., and 0004, M. W. (2001). Interactive rendering with coherent ray tracing. *Comput. Graph. Forum*, 20(3):153–165. 29

[Whitted, 1980] Whitted, T. (1980). An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349. 3, 29

[Williams, 1983] Williams, L. (1983). Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3):1–11. 3, 21, 29

[Zhang et al., 2007] Zhang, L., Chen, W., Ebert, D. S., and Peng, Q. (2007). Conservative voxelization. *Vis. Comput.*, 23(9):783–792. 11, 52

[Zhukov et al., 1998] Zhukov, S., Inoes, A., and Kronin, G. (1998). An Ambient Light Illumination Model. In Drettakis, G. and Max, N., editors, *Rendering Techniques '98*, Eurographics, pages 45–56. Springer-Verlag Wien New York. 2

[Zirr et al., 2013] Zirr, T., Rehfeld, H., and Dachsbacher, C. (2013). Object-order ray tracing for fully dynamic scenes. In Engel, W., editor, *GPU Pro 5*. A K Peters/CRC Press, Boca Raton, FL, USA. 3

List of Figures