Comparing a Clipmap to a Sparse Voxel Octree for Global Illumination

Master's thesis in the Programme Computer Science – Algorithms, Languages and Logic

Eric Arnebäck

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Comparing a Clipmap to a Sparse Voxel Octree for Global Illumination
Eric Arnebäck

Supervisor: Erik Sintorn
Advisor: Johan S. Carlson, Fraunhofer–Chalmers Centre
Examiner: Ulf Assarsson

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: A screenshot of a real-time global illumination approximation achieved using our CVCT implementation.

Typeset in LaTeX
Gothenburg, Sweden 2017

Abstract

Voxel cone tracing is a real-time method that approximates global illumination using a voxel approximation of the original scene. However, a high-resolution voxel approximation, which is necessary for good quality, consumes much memory, and a compact data structure for storing the voxels is necessary. In this thesis, as a primary contribution, we provide a comparison of two such data structures: a Sparse Voxel Octree, and a Clipmap.

We implement these two data structures, and provide detailed descriptions of both with many important implementation details. These descriptions are much more complete than what exists in the current literature, and it is the secondary contribution of this thesis.

In the comparison, we find that the octree performs worse than the clipmap with respect to memory consumption and performance, due to the overhead introduced by the complex octree data structure. However, with respect to visual quality, the octree is the superior choice, since the clipmap does not provide the same voxel resolution everywhere.

Keywords: computer graphics, global illumination, indirect lighting, real-time, octree, clipmap, comparison.

Acknowledgements

This master's thesis was done at Fraunhover-Chalmers Centre in Gothenburg. I would like to thank my boss Johan S. Carlson for letting me explore a topic I find immensely interesting. Many thanks goes to my supervisor Erik Sintorn for being a valuable discussion partner, and helping me greatly improve the quality of this report.

Contents

1

Introduction

In computer graphics, creating photo-realistic visualizations (such as games, 3D animated movies, and scientific visualizations) that are indistinguishable from reality is the ultimate goal. One approach to achieving this goal is to simulate the transport of light quantities in the scene, something often called *Global Illumination*.

But why is photo-realistic graphics important? What purpose does it serve humanity? Simply put, graphics provides us a three-dimensional virtual world. Providing entertaining games is only one possible application of this virtual world. In addition to entertainment, this world can also be used to construct many kinds of useful training simulators. For instance, a surgery simulator allows a practicing surgeon to practice surgery without having to risk lives. The surgeon does this by performing surgery inside a virtual human body. If the insides of the body are drawn using photo-realistic graphics, then it becomes much easier for the surgeon to navigate in the body, and global illumination is one technique that greatly enhances the realism of graphics [1].

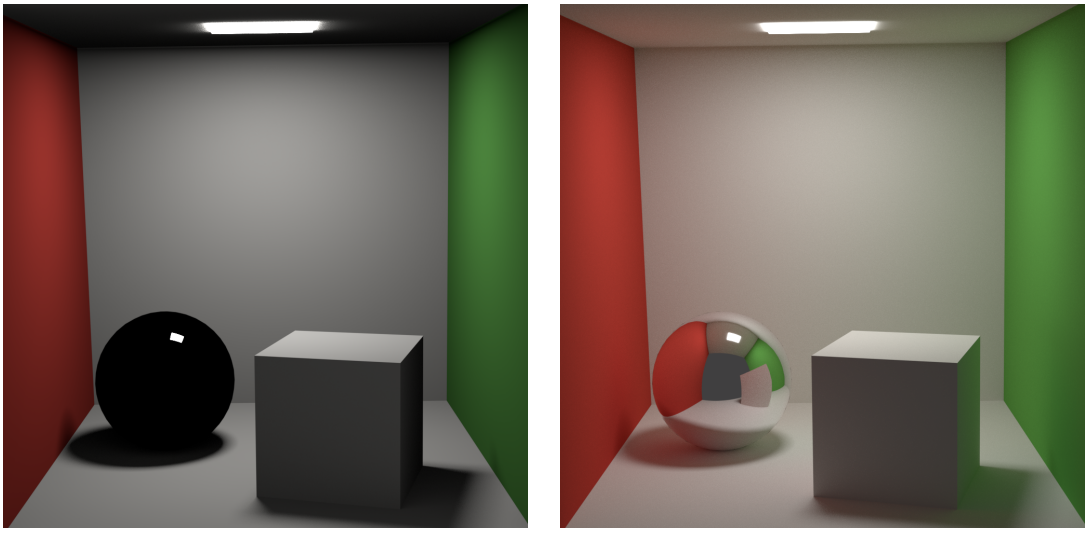
Global illumination is also used in architectural visualization. For an artist designing a building in some modelling software, being able to see whether a room has a visually pleasing placement of light sources is important. Without realistic lighting calculations, it becomes much more difficult to visualize such flaws [2, 3, 4].

Many scientific domains use particle-based methods to model complex phenomena. This is done by simulating some phenomenon with many small spheres called particles, and by rendering those particles the phenomenon can be visualized. For instance, Muzic et al. [37] use particles to visualize biochemical processes. In a study it was shown by Christiaan et al. [5] that global illumination makes scientific particle visualizations easier to interpret, because subtle details in the visualizations become more apparent.

Another application of global illumination is in the virtual prototyping of products. As an example, in the automotive industry, one can use computer graphics to provide a preview of a not-yet-built car to a customer, and for providing a photo-realistic preview of such a car, global illumination is paramount [6].

1.1 Global Illumination

We will first show an example of global illumination. See Figure 1a, which is an image of a virtual room with only *local illumination*, which means that the light source in the roof only affects points that are directly visible from it. A reader with little experience in graphics will probably remark that the image for some reason lacks realism, but unable to find the exact words for what is



(a) Local Illumination **(b)** Global Illumination

Figure 1: Comparing local and global illumination for a single room. The room has a reflective sphere, a white box, red and green walls, and a light source in the roof. Both images were rendered using Blender.

wrong.

The main problem is that the image has no global illumination. The equivalent image with global illumination can be seen in Figure 1b. The difference in realism is obvious. In Figure 1a, the roof of the room is dark. With local illumination, rays of light are emitted from the light source in the roof, and every ray traverses straight forward until it intersects some point of the room, thus illuminating that point. However, none of those rays will hit the roof. With global illumination, those rays are allowed to bounce in the room, and some of those rays will bounce to the roof, thus illuminating it. Another important difference between global and local illumination is the effect of *color bleeding*. It can be seen that the right side of the box is slightly colored green in Figure 1b. This is because rays have bounced off the green wall, and hit the box, thus coloring it slightly green. Finally, in Figure 1b, it can also be observed that the reflective sphere has a reflection of the rest of the room. Reflections is yet another attractive feature that can be implemented if rays are allowed to bounce.

1.2 The Rendering Equation

Global Illumination basically means that the graphics application simulates the transport of light in the scene, as we described in the previous section. In this section, we shall outline one model of performing such a simulation. The model of light transport that is used by a majority of researchers is the famous rendering equation of Kajiya [7]:

$$L_o(\mathbf{p}, \boldsymbol{\omega}) = L_e(\mathbf{p}, \boldsymbol{\omega}) + \int_\Omega f_r(\mathbf{p}, \boldsymbol{\omega}_i, \boldsymbol{\omega}) L_i(\mathbf{p}, \boldsymbol{\omega}_i)(\boldsymbol{\omega}_i \cdot \mathbf{n})\, d\boldsymbol{\omega}_i. \tag{1}$$

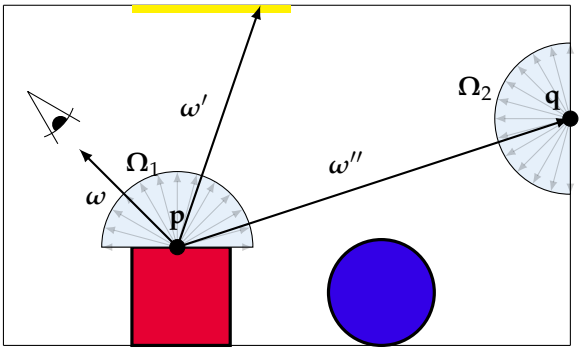


Figure 2: Illustration of Rendering Equation. In order to compute the outgoing radiance from $\mathbf{p}$ to the eye, $L_o(\mathbf{p}, \boldsymbol{\omega})$, we must gather all radiance contributions in the hemisphere $\boldsymbol{\Omega}_1$.

Radiance is a physical quantity that measures the radiant flux per steradian per square meter, and it is commonly used to measure the amount of light energy in light transport calculations [8]. The outgoing radiance from some point $\mathbf{p}$ in the direction $\boldsymbol{\omega}$ is denoted $L_o(\mathbf{p}, \boldsymbol{\omega})$ in Equation 1. Similarly, the incoming radiance from the direction $\boldsymbol{\omega}_i$ to the point $\mathbf{p}$ is denoted $L_i(\mathbf{p}, \boldsymbol{\omega}_i)$.

The domain of integration, Ω, is the positive hemisphere that is oriented by the normal $\mathbf{n}$, which is simply the normal at $\mathbf{p}$. The function f_r is the bi-directional reflectance distribution function (BRDF), which describes how light is reflected at the point $\mathbf{p}$, and it is depends on both the incoming direction $\boldsymbol{\omega}_i$ and the outgoing direction $\boldsymbol{\omega}$. Next, $d\boldsymbol{\omega}_i$ denotes that the hemisphere is integrated over all solid angles of the hemisphere. Finally, the dot product $(\boldsymbol{\omega}_i \cdot \mathbf{n})$ is simply *Lambert's law*, which says that the incoming light energy is proportional to the cosine of the angle between the incoming light direction and the surface normal.

The rendering equation allows us to compute the outgoing radiance $L_o(\mathbf{p}, \boldsymbol{\omega})$ for any point $\mathbf{p}$ and outgoing direction $\boldsymbol{\omega}$. When rendering some scene, we basically want to compute the color of every pixel on the screen. A single pixel will cover some point $\mathbf{p}$ in the scene, and let us denote the direction from that point to the camera $\boldsymbol{\omega}$. Then the color of that pixel is determined by the amount of radiance that reaches it. This amount is $L_o(\mathbf{p}, \boldsymbol{\omega})$, which can simply be computed with Equation 1.

The rendering equation is recursive. In order to calculate the outgoing radiance from $\mathbf{p}$ to the eye (Figure 2), we must evaluate the integral. The integral essentially gathers the incoming radiance contributions from all possible directions in the hemisphere Ω_1. If rays are sent from $\mathbf{p}$ in all the directions, the rays will sooner or later intersect some geometry at some points. From Equation 1 we have that if we compute the outgoing radiance from all these points and add them together weighted with $f_r(\mathbf{p}, \boldsymbol{\omega}_i, \boldsymbol{\omega})(\boldsymbol{\omega}_i \cdot \mathbf{n})$, we obtain $L_o(\mathbf{p}, \boldsymbol{\omega})$.

In the case of the direction $\boldsymbol{\omega}'$ in Figure 2, the ray from $\mathbf{p}$ will directly hit some light source in the roof, and so the radiance contribution $L_i(\mathbf{p}, \boldsymbol{\omega}')$ will simply depend on the light color and strength of the light source.

However, now consider the incoming radiance from the direction $\boldsymbol{\omega}''$. This will be the outgoing radiance from $\mathbf{q}$ in the opposite direction, or simply $L_o(\mathbf{q}, -\boldsymbol{\omega}'')$. But we can now use Equation 1 to find $L_o(\mathbf{q}, -\boldsymbol{\omega}'')$. But this means that we are back where we started: we have to evaluate the

rendering equation again, although this time the domain is $\mathbf{\Omega}_2$. What we just have demonstrated, is that the rendering equation is a recursive equation. Therefore, an analytical solution of the rendering equation is generally unfeasible, and researchers instead approach the problem by finding approximate solutions.

1.3 Problem Statement

Voxel cone tracing is an approach to global illumination that was first described by Crassin et al [9]. They voxelize the scene and put the voxels in an octree, and use this data structure to achieve an approximation to global illumination. In the game "The Tomorrow Children", McLaren et al. use a cascaded grid as an alternative to an octree [10]. Furthermore, NVIDIA has published a global illumination library based on cone tracing, VXGI [11]. It is a based on a clipmap, which is a data structure very similar to a cascaded grid [12].

McLaren et al. propose that a cascaded grid is a better fit for the GPU, but they never perform any benchmarking that proves a cascaded grid superior to an octree. NVIDIA do not to provide any such comparison for their VXGI library either. Performing such a comparison will allow us to see the relative advantages and disadvantages of an octree compared to a clipmap/cascaded grid, and we suspect that valuable insights may be gained by doing so.

Therefore, in order to perform such a comparison, we will first implement voxel cone tracing with an octree as described by Crassin et al [9]. and then implement it using a clipmap, as is done in VXGI [11]. We choose to implement it using the clipmaps of VXGI instead of the cascaded grids of McLaren et al. [10], because we suspect that the former variant will take less time to implement, because it appears less complex. We will dub the first approach SVOCT (Sparse Voxel Octree Cone Tracing) and the second one CVCT (Clipmap Voxel Cone Tracing).

Once both techniques have been implemented, we will perform a comparison of them. The following research question shall be answered by our comparison:

> What advantages does an octree have over a clipmap, and what advantages does a clipmap have over an octree?

The following aspects shall be considered in our comparison:

- Performance for scenes with much dynamic geometry.
- Memory consumption.
- Performance for a quickly moving camera.
- Performance for scenes with dynamic light sources.
- Performance for cone tracing.

- Visual quality of mirror reflections.

Motivations for the above list shall be provided in Section 5. Providing this comparison shall be our primary contribution.

For both techniques, we consider that several important implementation details are missing from their original descriptions. Therefore, as a secondary contribution, we aim to provide a more detailed description of the implementation of both techniques, so that future implementors will be able to implement the techniques with greater ease.

2

Previous Work

In this section, we shall discuss some earlier attempts at achieving real-time global illumination. Only an overview of the different techniques applied for real-time global illumination shall be provided. For a complete survey, we refer the reader to the one of Ritschel et a [2].

2.1 VPL-Based Approaches

One approach to approximating global illumination is the concept of Virtual Point Lights, which Keller [13] use in the Instant Radiosity algorithm. First, photons are traced from the light sources into the scene (Figure 3). At the vertices of the resulting path, we place out Virtual Point Lights (VPLs). For each such VPL, a shadow map is rendered. Finally, those shadow maps are summed up into the final result. The largest drawback with the technique is that rendering the shadow maps is very expensive. Laine et al. [14] devise a scheme that reuses the shadow maps, and as a result, only a few shadow maps need to be rendered every frame. However, their technique also has the drawback that it only handles single-bounce indirect illumination, and it is not possible for dynamic objects to affect other objects with effects such as color bleeding.

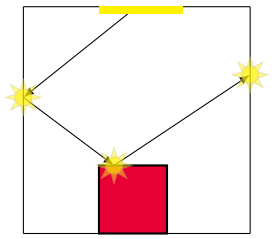


Figure 3: Photons are traced, and VPLs are placed at the path vertices.

Another method is the Imperfect Shadow Maps(ISMs) of Ritschel et al [15]. For each VPL, an ISM is created, which is a low-resolution shadow map renderered from a crude point-based version of the scene. Just like in Instant Radiosity, the ISMs are then summed up to produce the final result.

A different approach to creating the VPLs is Reflective Shadow Maps [16]. The authors render a Reflective Shadow Map(RSM) that contains both depth, world-space coordinates, normals and flux, and then considers every texel in the RSM a VPL. In a separate pass, for every visible point, global illumination is approximated by importance sampling the nearest texels in the RSM.

2.2 Screen-Space-Based Approaches

There are also screen-space-based methods. One example is SSDO, where first the normals and positions of every visible pixel are written to an off-screen buffer, and this buffer is used to compute the indirect illumination for every pixel on the screen. An issue with the technique is

that illumination coming from outside the screen will not be visible when it should be, because the algorithm is only working with screen-space information [17].

Mara et al. invent another screen-space approach based on Deep G-buffers [18]. In common with SSDO, information such as normals and positions are stored for the visible pixels. Additionally, however, the same information is stored for the layer of pixels behind those visible pixels. This two-layer representation is used to achieve an even better approximation to indirect illumination than SSDO. The authors show how it can be used to produce dynamic indirect radiosity, and mirror reflections. But again, the problem with the technique is that it is only based on screen-space data, so there will inevitably be artifacts.

2.3 Light Propagation Volumes

Another attempt at global illumination are the light propagation volumes of Kaplanyan et al. [19]. A low-resolution, three-dimensional grid is used to store the light that every grid cell contains. A reflective shadow map (RSM) is rendered, where every pixel in the RSM is considered a VPL, and the directional distribution of light intensity of every VPL is compactly represented by Spherical Harmonics (SH). Every VPL from the RSM is injected into the grid by simply accumulating the SH-coefficients. Due to the low resolution and the SH-approximation, only low frequency lighting can be represented.

In a second pass, a coarse voxelization of the geometry is injected into a separate grid. Then, using the light grid and the geometry grids, a light propagation scheme is implemented, in order to approximate the bounce of indirect lighting. From every cell, light is propagated to their 6 directly neighboring cells. This propagation pass is repeated a number of passes.

An improvement on the above method is to use a cascaded grid. The cascaded grid builds upon the observation that a high precision light representation is only necessary near the viewer. But for far away areas it becomes less necessary, since these areas occupy much smaller space on the screen. So instead a grid with less resolution far away is used, thus making great savings in storage. Since there are less grid cells, less propagation calculations will have to be performed, and thus the performance is improved as well.

2.4 Voxel-Based Approaches

More closely related to the topic of the thesis are approaches to global illumination that are based on voxels. Put succinctly, they are all based on the idea of approximating the scene by voxels, which are axis-aligned, equally-sized cubes with data associated with them, such as colors or normals. These voxels are used as a coarse approximation of the scene.

Thiedemann et al. [20], in their Voxel-based Global Illumination (VGI), first voxelize the scene using a novel atlas-based voxelization scheme; every object is voxelized by using its UV-map to render the fragments' world-positions to a texture atlas. Every non-empty texel is then inserted into a binary voxel occlusion grid, that encodes the existence of the voxels in the grid. An efficient ray/voxel intersection test can be implemented with this grid, and this test is used to capture

single-bounce indirect illumination as follows: for every light source, a RSM is rendered. The scene is then rendered and for every fragment visible from the camera, several rays are traced, in order to approximate the rendering equation with Monte-Carlo integration. For every ray, the first intersection point is found with the above-mentioned ray/voxel-intersection test. For the intersected voxel, the outgoing radiance must now be calculated, and this is found by simply back-projecting the intersection point onto the RSM (Figure 4).

The attractive part of the above algorithm is that it only requires binary voxels, since they are using the RSM to obtain the radiance, instead of storing the radiance in the voxels. In order to store a 128^3-voxel grid, the authors use a 128x128 2D-texture, where every texel is a 128-bit value, which is only $\sim$0.26 MB big. In contrast to other voxel-based approaches, large 3D-textures are not required. However, a drawback is that the method suffers from noise and flickering artifacts, since it is based on Monto Carlo Integration.

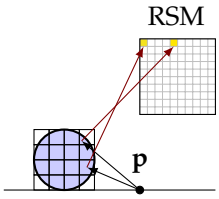


Figure 4: Intersected voxels are back-projected onto the RSM to calculate the radiance.

In his PhD thesis, Crassin developed GigaVoxels. It is a GPU-based rendering pipeline that uses voxels to render detailed scenes in interactive or real-time frame rates [21]. The pipeline uses a compact Sparse Voxel Octree data structure to store the voxels and implements an efficient GPU-based caching and on-demand loading system that allows for rendering of scenes that do not fit in the available GPU memory. More relevant to our thesis, he used his Sparse Voxel Octree data structure to achieve global illumination in interactive to real-time frame rates. Our implementation of this technique shall be described in much detail in section 3.

Sugihara et al [22]. propose an extension to VGI. In common with VGI, a binary voxel storage scheme is used. However, instead of an ordinary RSM, the technique adopts a novel Layered Reflective Shadow Map (LRSM). Like in Variance Shadow Maps [23], it contains both the depth and the squared depth values, and these two values are filtered in order to compute mipmapped RSMs. As the name implies, the LRSM is layered. First a normal RSM is created, and it is then split into several layers, thus forming the LRSM. The purpose of this splitting is to avoid depth and normal discontinuities.

Another novelty of the method of Sugihara et al. is that they use voxel cone tracing instead of Monte Carlo integration. They perform cone tracing on the binary voxelization, and to take samples in the cone tracing process, they back-project the samples onto the LRSM to calculate the radiance of the samples. Finally, the size of the samples is used to determine the mip-level of the LRSM to sample from.

3

Sparse Voxel Octree Cone Tracing

Sparse Voxel Octree Cone Tracing (SVOCT) was first described by Crassin et al. [9]. Their solution is able to approximate two-bounce indirect lighting, for Lambertian diffuse and glossy BRDFs. By "two-bounce", we mean that the light emitted from the light source is allowed to bounce only two times. In reality, light can bounce many times. However, since the first few bounces often have the greatest visual impact, a two-bounce approximation yields visually pleasing results.

The first step of SVOCT is to voxelize the surface geometry of the scene. This means that the surface triangles of the entire scene are approximated by an N^3-sized grid of equally-sized, axis-aligned cubes, where N is the resolution of the grid. The purpose of this is to provide a coarse approximation of the scene, and it will be easier to approximate global illumination on this representation. A large value of N means smaller voxels, and thus a better approximation to the original scene.

In order to store the voxels, a N^3-sized grid could simply be used, but such a data structure would consume much memory. The voxels will instead be stored in the leaves of an octree data structure. This ensures that no voxel data is stored for parts of the scene that contain no geometry. The octree is illustrated in Figure 6a. The smallest squares are the voxels in the figure.

Next, light is injected into the voxels. This means that every voxel that is visible from the light source will receive radiance from the light source (Figure 6b). First, radiance is injected into the leaf nodes, and then the radiance is filtered into the higher-level nodes of the octree, using a Gaussian weighting kernel. In this manner, the approximate radiance is computed for the higher-level nodes in the octree. In Figure 5a, the original scene can be observed, and in Figure 5b light has been injected into the leaves that are visible from the light source. Further, Figure 5c and Figure 5d illustrates that the radiance in the leaves has been filtered to the higher-level nodes.

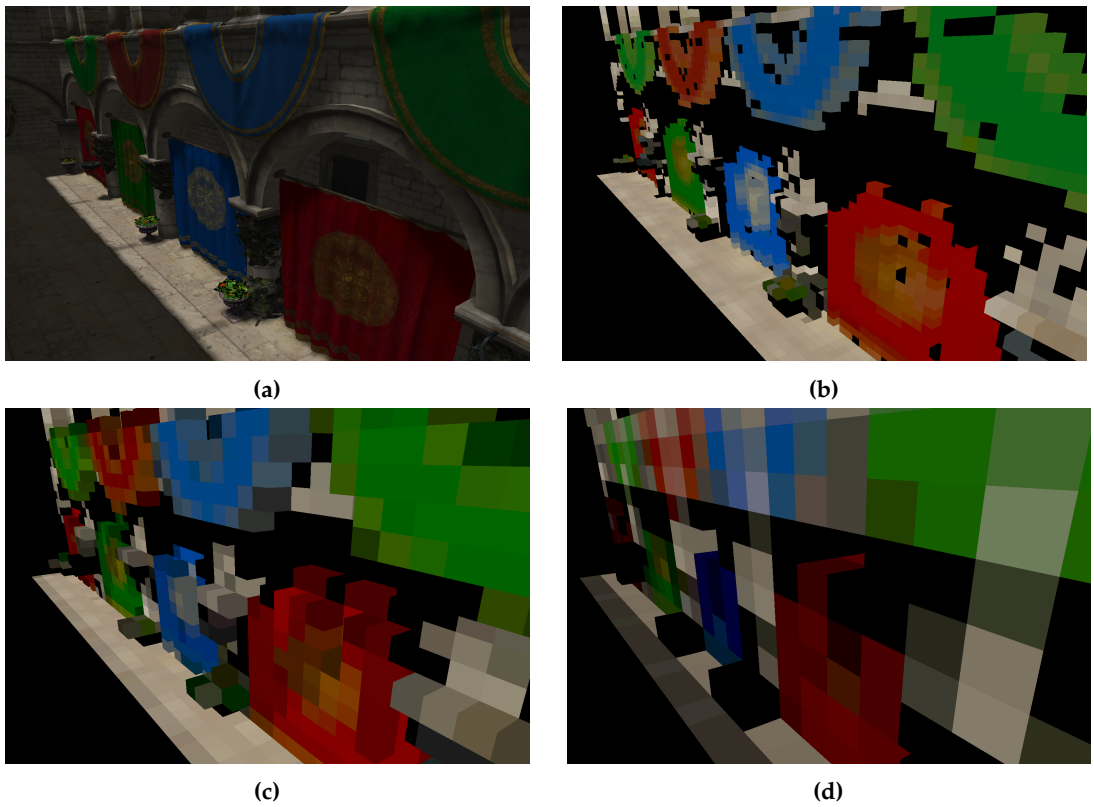


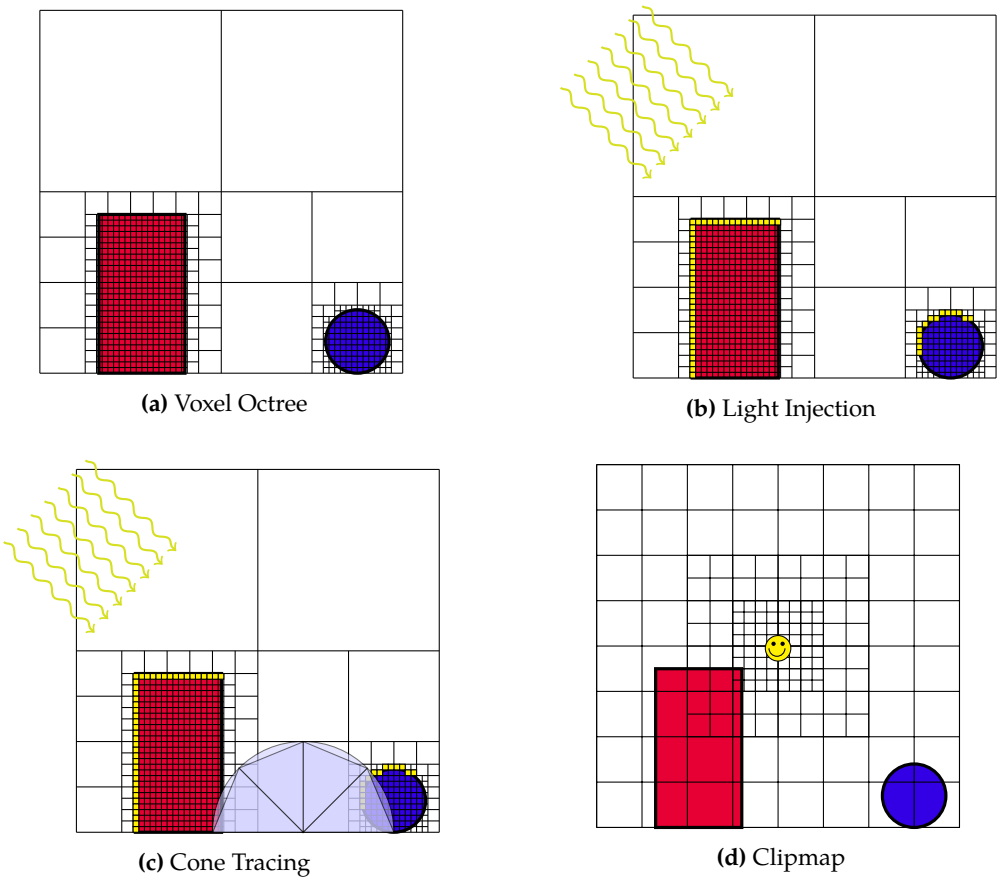
Figure 5: (a) Our test scene. Areas visible from the light source are bright, and the rest is dark (b) Radiance stored at level 1, the leaf-level. (c) Radiance at level 2, the level above the leaf-level. (d) Radiance at level 3.

Finally, the radiance stored in the octree is used to solve Equation 1 for two-bounce indirect lighting. The main difficulty lies in sampling a sufficient number of rays from the domain Ω. It could be solved by casting many rays, and for each ray we compute the radiance at the intersection point, and then the results of the rays are combined with Monte Carlo integration. This was attempted by Thiedemann et al. [20] in VGI, but it suffers from noise and flickering artifacts. Therefore, Crassin et al.[9] instead approximate a large number of rays, with rougly the same direction, with a single cone. The entire hemisphere Ω can be sampled with only a few cones, as is illustrated in Figure 6c. For every cone, they ray march along the cone, taking progressively larger radiance samples from the octree. Since the radiance values in the octree are *prefiltered*, the resulting indirect lighting is smooth, and suffers from no aliasing artifacts.

3.1 Theory

In this section, the theory that is necessary to understand our implementation shall be covered.

3.1.1 Sparse Voxel Octree Data Structure



(a) Voxel Octree

(b) Light Injection

(c) Cone Tracing

(d) Clipmap

Figure 6: A room, where the only geometries are a red box and a blue sphere. (a) The sparse voxel octree of the voxelized room. The leaves of the octree represent the voxels. (b) The octree after light injection. (c) the cone tracing pass. (d) How the room is represented as voxels with a clipmap.

A Sparse Voxel Octree is a compact data structure used to store the voxels that approximate the scene. It is a spatial data structure that recursively subdivides space into octants. This spatial subdivision scheme is illustrated in Figure 7.

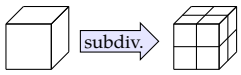


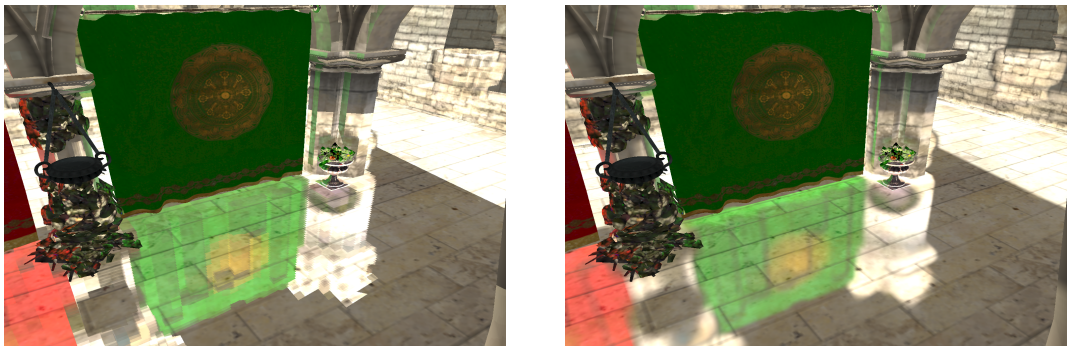
Figure 7: Octree subdivision of space into 8 octants.

The root node of the tree represents the entire scene, and it contains a pointer to its eight children and every such child represents one of the eight octants that the root has been subdivided into. The octants spanned by these children are in turn subdivided into suboctants, and each child has pointers to its suboctant nodes. This subdivision process will continue until the octants are voxel-sized. As a result, the voxels are basically represented by the leaves of the octree. The subdivision process stops preemptively for a node if the node contains no voxel fragments. A quadtree[1] for a simple

[1]To simplify matters, all illustrations will depict 2D voxels, instead of 3D voxels. So quadtrees will be used instead of

scene is illustrated in Figure 6a. As can be observed, if an inner node of the tree contains no voxels, then the subdivision of that node will be stopped before the leaf-level can be reached.

Crassin et al. store the octree with a *node pool* and a *brick pool*. The *node pool* is a long buffer of octree nodes. The node buffer is illustrated in Figure 9. As can be observed, the children of a node are stored consecutively in memory, and every node has a pointer to the first of its eight children. Nodes with no children will simply store a null pointer instead.



(a) **(b)**

Figure 8: (a) Reflection rendered without any interpolation, using no bricks. (b) Reflection rendered using bricks for trilinear interpolation (The reader may zoom into the PDF in order to see more detailed images).

A node will need to contain an approximation of the material colors, normals, and radiance values of the geometry that it contains. Crassin et al. store these three values in *bricks*. The purpose of the brick is to store a 2^3 voxel approximation of the section of the scene that the node contains. Instead of storing a single average value of the containing geometry, average values are stored for each of the eight voxel octants that the node contains.

Bricks are necessary to allow for sampling values with hardware-accelerated trilinear interpolation. If a node could only store a single value instead of a brick, all the samples taken inside that node can only assume a single possible value. If voxel cone tracing is performed on such a node, the result is indirect lighting with a "blocky" appearance, as is illustrated in Figure 8.

The brick is illustrated in Figure 10a, where a two-dimensional 2^2 brick containing the values f, g, j and k is illustrated. In order to sample from the node, interpolation is performed between the 4 nearest values in the brick. For the sample point α in the figure, bilinear interpolation (trilinear interpolation in 3D) would be performed between the values stored at f, g, j and k. This interpolation can be cheaply implemented using hardware-accelerated interpolation, by enabling a magnification filter of `GL_LINEAR` for the brick pool 3D texture(the brick pool will be discussed later) in OpenGL.

octrees in the illustrations.

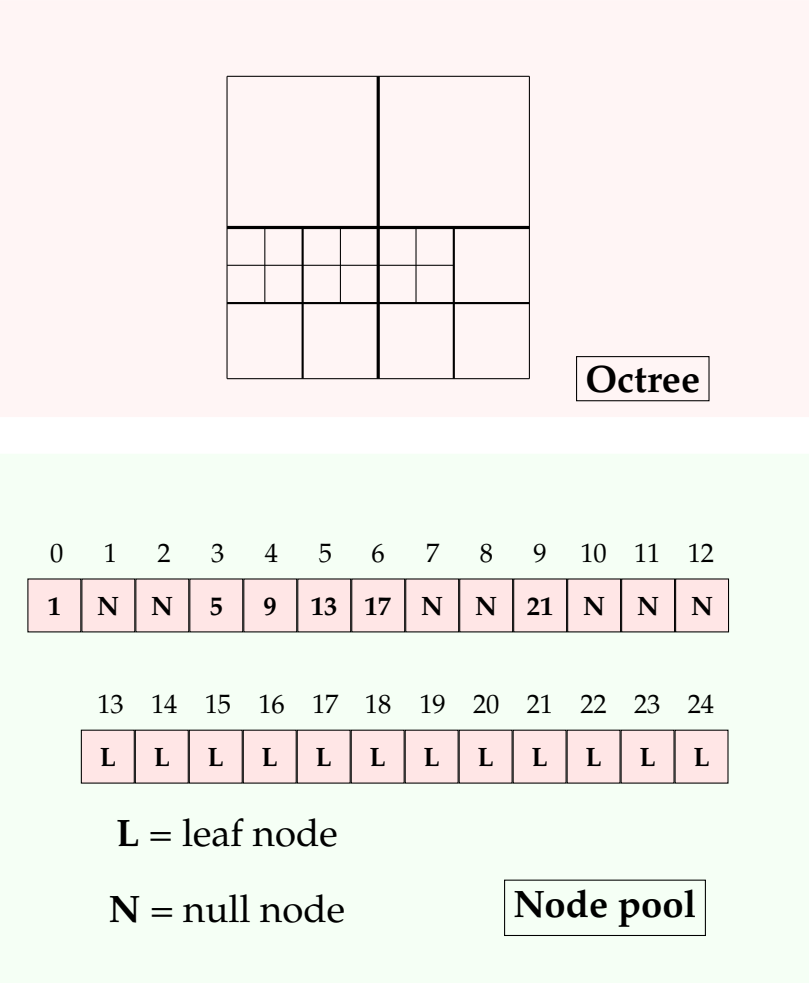


Figure 9: An quadtree and its node pool representation. Nodes $1 - 12$ are inner nodes, while nodes $13 - 24$ are leaf nodes, and 0 is the root-node. The root-node points to the beginning of its four children. First child is the north-west child, the second child is the north-east child, the third child is the south-west child, and the fourth child is the south-east child. For the root-node, since the north-west and north-east child nodes have no children, those two children are null-nodes. The leaf nodes also point to no children.

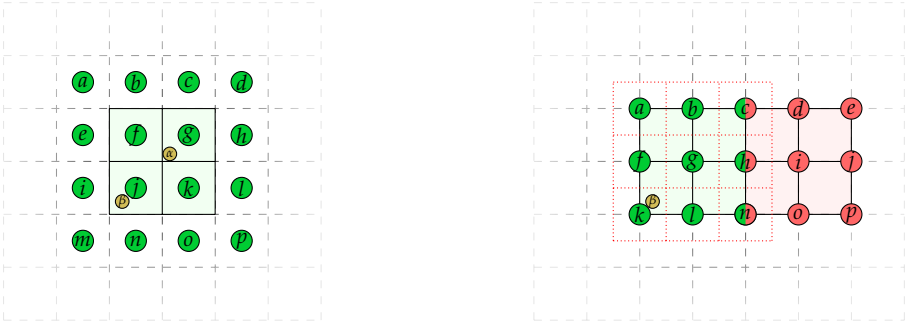


Figure 10: (a) A 4^3 brick. (b) A 3^3 brick

However, in order to sample with bilinear interpolation from points that are near the corners of the brick, the interpolation computation requires access to values that are outside the brick of the current node, and inside the brick of some neighboring node. In order to obtain an interpolated value at the sample point β in the figure, an interpolation between the voxels i, j, m and n is necessary, and so in this case the four values stored in the brick are not enough. Thus, a 2^3 brick does not suffice, and an one-voxel border must be added around the brick, resulting in a 4^3 brick. In Figure 10a, the voxel border is the values a, b, c, d, e, h, i, l, m, n, o and p.

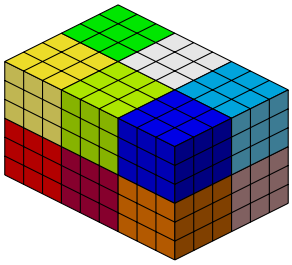


Figure 11: Brick Pool.

The disadvantage of a 4^3 brick this is that much memory will be consumed by the bricks, since every node has a brick. Therefore, Crassin et al. [9] propose a different scheme: in the 4^3 brick, the voxels in the brick are centered on the nodes[2]. That is, the voxels cover the exact same volumes as the nodes. Instead of centering on the node center, Crassin et al. center the voxels on the *node corners*, as is illustrated in Figure 10b. As an example, the voxel a is centered on the corner of the top-left node, and the voxel g is centered on the corners of all the four nodes. In 2D, there are $3^2 = 9$ such corners, in 3D there are $3^3 = 27$ corners, and for every such corner a voxel value is stored. This scheme also allows for hardware-accelerated interpolation; in the figure the sample point β can be obtained by simply interpolating between f, g, k and l.

Clearly, this scheme is more size efficient than the 4^3-scheme, with savings of $4^3 - 3^3 = 37$ voxel values. Unfortunately, the scheme also has redundancy, since voxels at the borders of the brick are duplicated for neighboring bricks. This can be seen in the figure, where c, h and n are duplicated between the green and red bricks.

All these 3^3 bricks are stored in a brick pool. The brick pool is a large 3D texture, and a brick is a 3^3-sized section of this texture (see Figure 11).

[2]On the leaf-level, we imagine that the 8 suboctants are nodes. On the higher levels, the nodes we are referring are simply the child nodes.

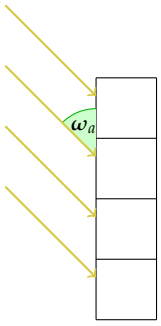


Figure 12: For a directional light source with no bounce, all incoming radiance comes from a single direction.

3.1.2 Light Injection

In this section, it shall be described how radiance is injected into every voxel, in terms of the Rendering Equation. To simplify matters, it will be assumed that the scene has only a single directional light source. Further, the assumption is made that all voxels have the reflective properties of a Lambertian diffuse BRDF. Note that Crasin et al. can also handle voxels with a glossy BRDF. However, we were unable to discover how they deal with a glossy BRDF, so we will only handle a Lambertian diffuse BRDF in the description that follows.

The Lambertian BRDF reflects light equally in all directions, so that the BRDF is $f_r(\mathbf{p}, \boldsymbol{\omega}_i, \boldsymbol{\omega}) = \mathbf{c}$, where $\mathbf{c}$ is the color of the voxel. The outgoing radiance for every voxel is

$$L_o(\mathbf{p}, \boldsymbol{\omega}) = L_e(\mathbf{p}, \boldsymbol{\omega}) + \int_\Omega f_r(\mathbf{p}, \boldsymbol{\omega}_i, \boldsymbol{\omega}) L_i(\mathbf{p}, \boldsymbol{\omega}_i)(\boldsymbol{\omega}_i \cdot \mathbf{n})\, d\boldsymbol{\omega}_i = \int_\Omega \mathbf{c}\, L_i(\mathbf{p}, \boldsymbol{\omega}_i)(\boldsymbol{\omega}_i \cdot \mathbf{n})\, d\boldsymbol{\omega}_i$$

Since it is assumed the voxels emit no light. Another simplification is made: the outgoing radiance emitted from the directional light source is not bounced a single time. Thus, since the light source is directional, all the incoming radiance of a voxel comes from a single direction we denote $\boldsymbol{\omega}_a$, as is illustrated in Figure 12.

The integral in the rendering equation is essentially an infinite sum over infinitesimally small solid angles $d\boldsymbol{\omega}_i$. However, if the incoming radiance comes from a single direction $\boldsymbol{\omega}_a$, then the only non-zero term in this sum will be the term with $\boldsymbol{\omega}_a$. Thus it follows

$$\int_\Omega \mathbf{c}\, L_i(\mathbf{p}, \boldsymbol{\omega}_i)(\boldsymbol{\omega}_i \cdot \mathbf{n})\, d\boldsymbol{\omega}_i = \mathbf{c}\, L_i(\mathbf{p}, \boldsymbol{\omega}_a)(\boldsymbol{\omega}_a \cdot \mathbf{n}) \tag{2}$$

where the value of the incoming incoming radiance $L_i(\mathbf{p}, \boldsymbol{\omega}_a)$ will be determined by the strength and color of the directional light source, and $\mathbf{n}$ is the average voxel normal at the voxel in question.

To conclude, at every voxel visible from the directional light source, its outgoing radiance will be injected, which is calculated by Equation 2. This is illustrated in Figure 6b. Finally, it is simple to also handle point light sources. For this case, ω_a will not be constant, but it will be the vector from the point light position to the voxel position.

3.1.3 Radiance Filtering Kernel

After light injection, the values in the leaf-level bricks will be filtered to the higher-level bricks, thus creating a mipmapped octree. This means that the higher levels will be a lower-resolution approximation of the lower levels. See Figure 13 for an illustration of this.

However, the mipmap computation is complicated by the corner-centered voxels. Every higher-level node will compute its brick voxels from brick voxels of its lower-level children nodes. See Figure 14. In 2D, 9 lower-level brick values are required to compute a single higher-level brick value. For the voxel γ in the figure, these values are indicated with the arrows[3]. Some weighting kernel will be applied to compute the mipmapped value from the 9 values. An obvious suggestion is a box filter:

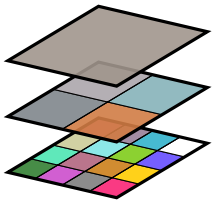


Figure 13: The voxel data is filtered up the voxel hierarchy by averaging. Every layer in the image represents a level in the octree

$$\begin{bmatrix} 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \\ 1/9 & 1/9 & 1/9 \end{bmatrix}$$

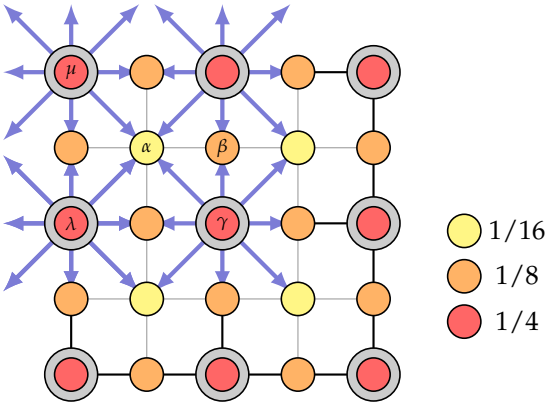


Figure 14: A higher-level node, with its four lower-level children. The brick voxels of the higher-level node is represented by the large circles, and the brick voxels of the lower-level nodes are the small circles (as can be observed some lower-level brick voxels are shared between bricks). The arrows indicate which lower-level brick voxels are used to compute the higher-level brick voxels, and the colors indicate which weights will be used.

[3]Note that the value at the center of the arrows is also included

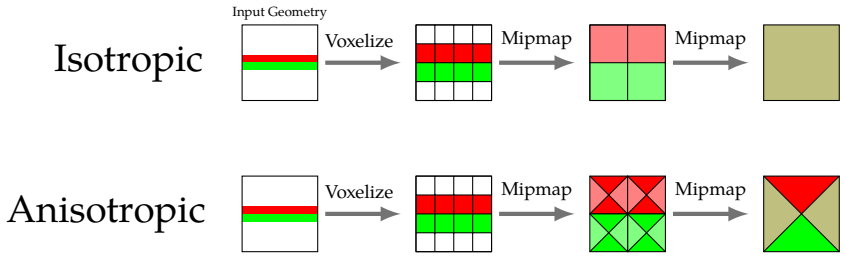


Figure 15: Comparison of mipmapping isotropic voxels and anisotropic voxels.

However, this kernel is biased. Observe that the lower-level brick value α in the figure will be used in the computation of four mipmapped values, since four arrows are pointing to it. Further, β will be used by two, and α will only be used by one. If the same weight value is used for all of the three types, then α will contribute more to the higher-level bricks than β and γ. Thus, a box-filter kernel is biased for the brick scheme of Crassin et al. Instead, they suggest a Gaussian weighting kernel:

$$\begin{bmatrix} 1/16 & 1/8 & 1/16 \\ 1/8 & 1/4 & 1/8 \\ 1/16 & 1/8 & 1/16 \end{bmatrix}$$

with this kernel γ will have the weight $1/4$, β the weight $1/8$, and α the weight $1/16$. Thus, the weight of α is $1/4$ the weight of γ, but this is fair since α contributes to four mipmapped voxels, while γ contributes only to one. Similarly, the weight of β is $1/2$ of γ, since β contributes to two mipmapped voxels.

3.1.4 Anisotropic Voxels

There are two kinds of voxel representations that Crassin et al. discuss: isotropic and anisotropic voxels. Isotropic voxels store a single value per voxel. This representation is denoted isotropic, because no matter what angle such a voxel is viewed from, it will always have the same value. When mipmapping such voxels, the children are multiplied by some kernel, and added together. However, this may have undesired results. As can be observed in Figure 15, if isotropic voxels are mipmapped by averaging (that is, using a box-filter kernel), progressively worse results are obtained. The original geometry is a red and a green wall facing each other, but in the upper levels this becomes a brown voxel. Even with a Gaussian kernel, the end result will be similar.

However, by using an anisotropic voxel representation, the issue can be alleviated. Instead of storing a single value per voxel, one value is stored for each face of the voxel cube, in total six values. These values encode the color of a voxel viewed from a certain direction. See Figure 16. When viewed from the direction ω, the voxel will be yellow, and seen from ω', it will be green. When not directly facing one of the faces, an interpolation will be performed between the faces, as is indicated by the color wheel in the figure. For the direction ω'', the interpolated value will be the average value of yellow and green.

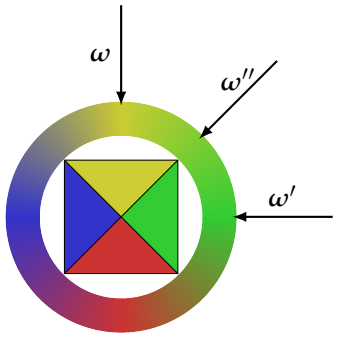


Figure 16: Anisotropic voxel.

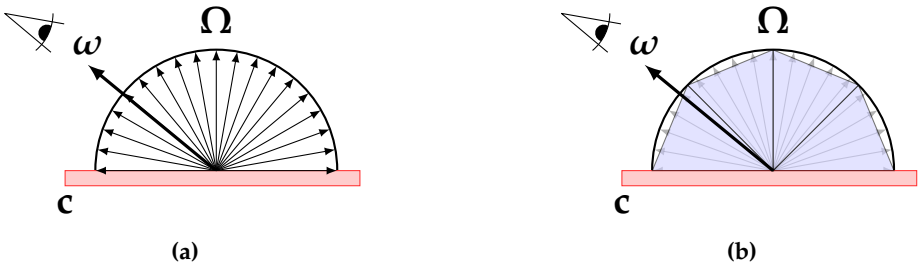


Figure 17: (a) Lambertian Diffuse BRDF. (b) Lambertian diffuse BRDF approximated by cones.

In Figure 15, the result of mipmapping anisotropic voxels can be observed. Now the highest level voxel is no longer brown. Instead, it is red when viewed from above, and green when viewed from below, which is a much better representation of the original red-green wall. The mipmapping algorithm for anisotropic voxels is described in more detail by Crassin et al [9].

3.1.5 Cone Tracing

In this section, it shall be described how the radiance values stored in the octree are used to approximate both a Lambertian diffuse and a glossy specular lighting component with voxel cone tracing.

As mentioned in Section 3.1.2, the Lambertian diffuse BRDF is $f_r(\mathbf{p}, \omega_i, \omega) = \mathbf{c}$. In order to compute the outgoing radiance from a point in the scene with such a BRDF, many rays must be sampled in the hemisphere Ω, which is illustrated in Figure17a. Every ray will intersect some point in the scene, and compute the outgoing radiance at the point of intersection. All the radiance values from the rays will be weighted by the Lambertian diffuse BRDF and the cosine factor(the dot product of Equation 1), and added together.

However, instead of gathering light with many rays, Crassin et al. approximate the hemisphere of rays with cone tracing. That is, the results of a large number of rays with roughly the same direction are approximated with a single cone, and the hemisphere is partitioned into several such cones. This is illustrated in Figure 17b. For every cone, raymarching is performed in order

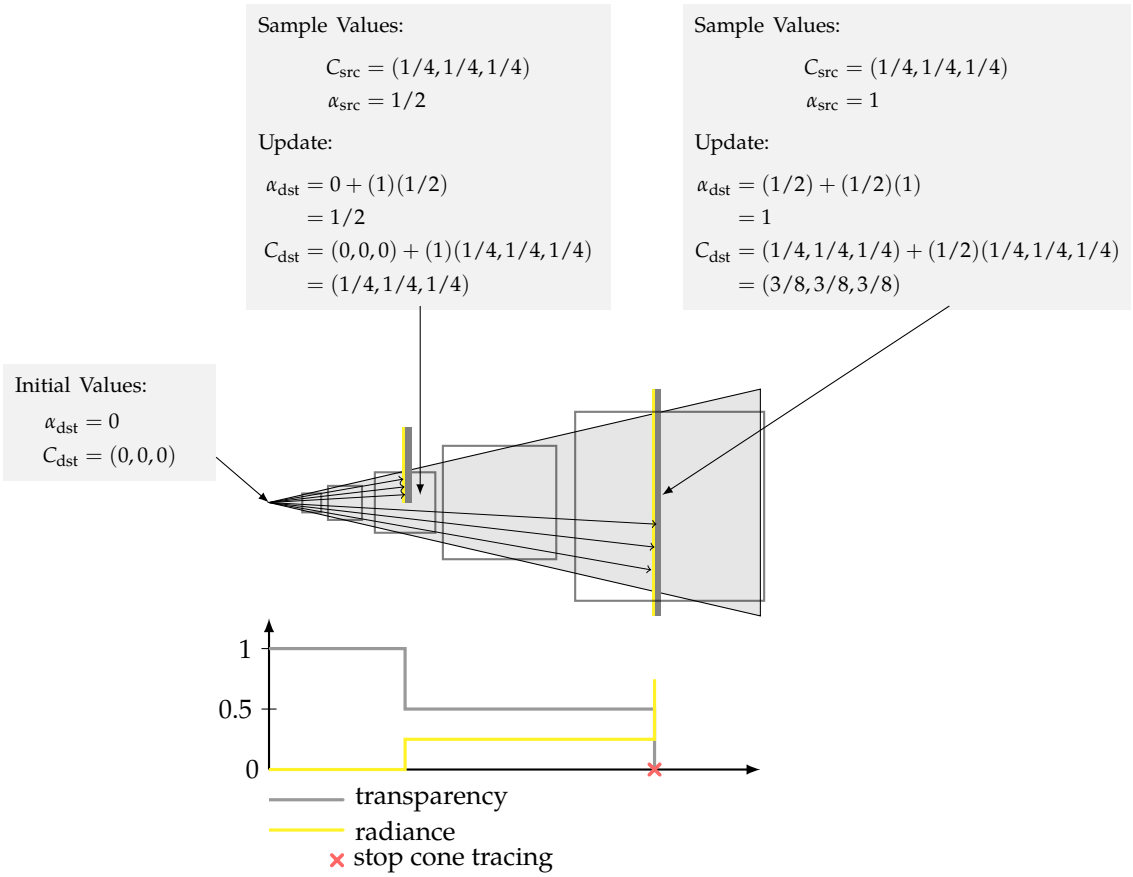


Figure 18: Illustration of how a cone of rays is approximated with cubic volumes. Raymarching is performed along the cone, and radiance is sampled until a transparency value of zero is reached. Observe that transparency $= 1.0 -$ opacity.

to approximate the results of all the rays that the cone contains.

The cone tracing process is illustrated in Figure 18. The cone is approximated by raymarching in the direction of the cone, and taking progressively larger radiance samples from the octree. The radiance samples are accumulated into a value C_{dst}. As can be observed in the figure, some of the rays contained in the cone will be blocked by geometry. In order to approximate this phenomenon, Crassin et al. keep track of an accumulated opacity value α_{dst}. From the start, the accumulated opacity is zero, but the value will be updated for every taken sample with the following update scheme:

$$\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst})C_{src} \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst})\alpha_{src} \end{aligned} \tag{3}$$

This is a scheme that is commonly used when rendering volumetric data sets [24]. The sampled radiance is C_{src}, and the sampled opacity is α_{src}. In Figure 18, the gray text-boxes illustrate how

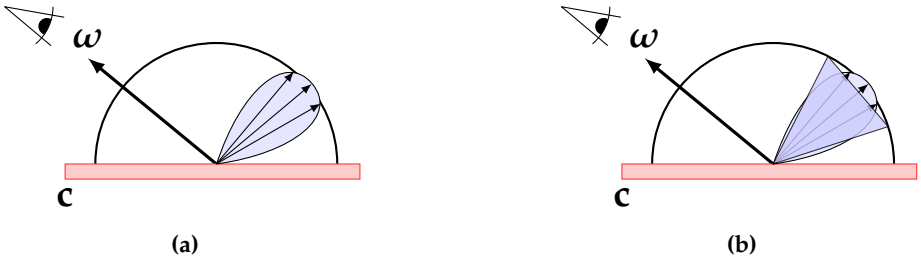


Figure 19: (a) Glossy BRDF. (b) Glossy BRDF approximated by a single cone.

the update scheme is applied for every sample. Inside the third sample, there is some geometry that blocks half the rays. Therefore, the transparency of the sample is $1/2$, and this ensures that the accumulated opacity becomes $\alpha_{\text{dst}} = 1/2$ after applying the update scheme. This means that the next sample will be weighted by $1/2$. But this makes sense, since at the third sample half the rays were blocked. The cone tracing stops once α_{dst} reaches a value of one (full opacity). Thus, in the figure it will stop after the fifth sample.

Finally, cone tracing can also approximate a glossy BRDF. The glossy BRDF is a lobe oriented in the direction of reflection, as is demonstrated in Figure 19a. The rays within the lobe contribute much more to the final result than the remaining rays in the hemisphere. Therefore, a glossy BRDF can be approximated with a single cone in the direction of the lobe, as is shown in Figure 19b. For a rough glossy material, the lobe is large, and it will be approximated with a wide cone. For an almost perfectly specular material, the lobe is very thin, and thus a thin cone will be used.

The cone tracing algorithm implements two-bounce indirect illumination. This is illustrated in Figure 20. First, light is injected into the voxels visible from the light source. The voxels are treated as diffuse reflectors, so the injected light bounces an equal amount in all directions in a hemisphere, as is shown by the gray arrows in the figure. This is the first bounce. Cone tracing is then used to find the incoming radiance at $\mathbf{p}$. Light was diffusely reflected from the voxels, and cone tracing is used to compute the reflected light that reaches $\mathbf{p}$ by sampling the radiance from the octree. In this manner, cone tracing is used to approximate the incoming radiance of $\mathbf{p}$. If the material at $\mathbf{p}$ is a diffuse reflector, then that incoming radiance will diffusely bounce in all directions of the hemisphere, and one of the rays reach the eye. This is the second bounce.

If there is a glossy specular reflector at $\mathbf{p}$, then a cone is traced in the direction of reflection, thus gathering the incoming radiance at $\mathbf{p}$. That incoming radiance will then bounce straight into the eye.

3.2 Implementation

Our implementation of SVOCT shall be described in this section. It is strongly influenced by the original technique of Crassin et al [9], but some parts are slightly different.

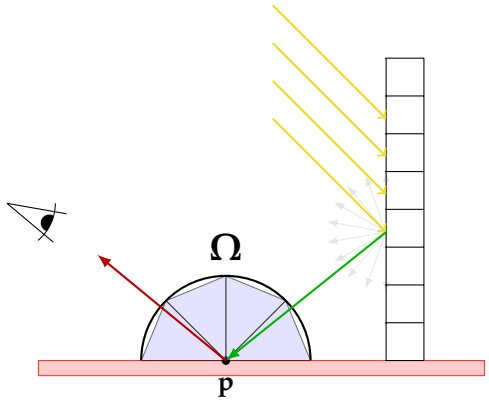
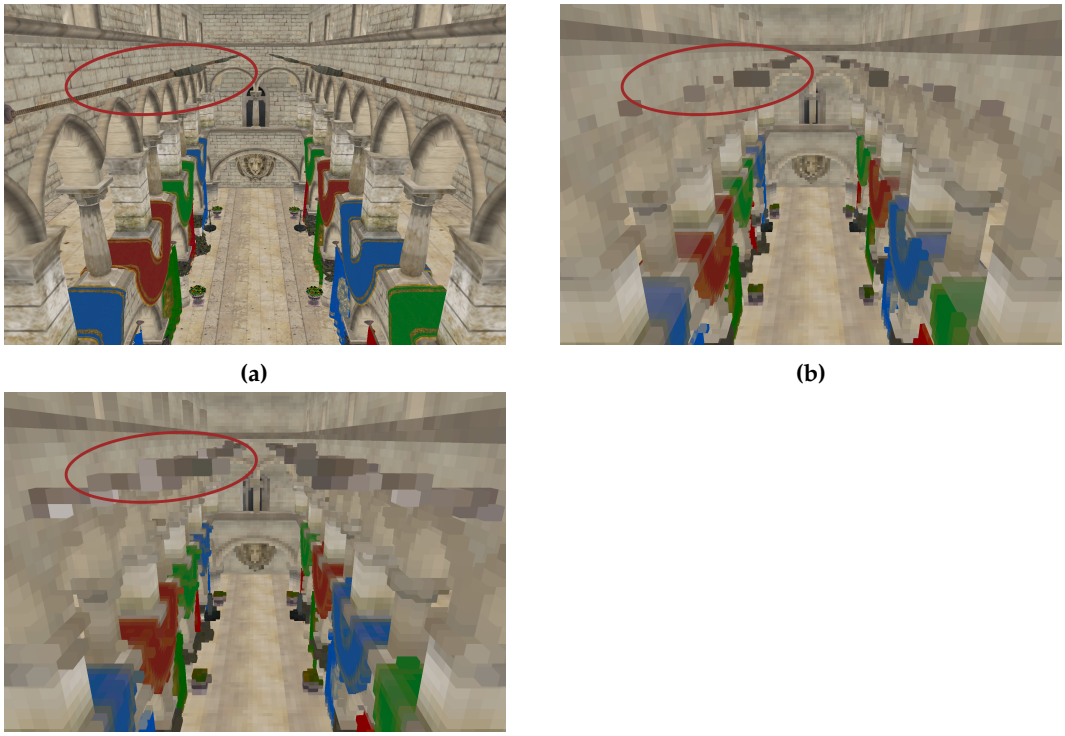


Figure 20: With cone tracing we can implement two-bounce indirect illumination. The green ray is light that has bounced once. The red ray is light that has bounced twice.



(a) **(b)**

(c)

Figure 21: (a) The scene to be voxelized. (b) The scene voxelized normally. (c) The scene voxelized with MSAA enabled (The reader may zoom into the PDF in order to see more detailed images).

3.2.1 Voxelization

Our voxelization method of choice is the method described by Crassin and Green [25]. It was chosen because it is simple, and there was not any time to evaluate other approaches. We summarize the method here: first, we define an orthographic projection matrix that entirely contains the scene, and define a view matrix so that the camera looks at the scene along the z-axis. Next, we disable all framebuffer operations, such as depth write, depth testing, and color writes. Finally, we set the 2D viewport resolution to equal the voxel resolution of the grid.

Afterward, we render the entire scene using the projection and view matrices just defined. With these matrices and with the depth testing disabled, all geometry in the scene will be rendered, and nothing will be clipped. The rendered geometry will generate fragments, and every fragment will fall into some grid-cell in the voxel grid. We denote these fragments as *voxel fragments*, and they are appended to a buffer called the *fragment list*. We will later use this fragment list as input to the algorithm that creates the Sparse Voxel Octree.

Listing 1: Voxel Fragment Data

```
struct FragmentData {
  uint color;
  uint position;
  uint normal;
};
```

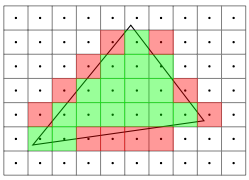


Figure 22: Voxelization of a triangle. Green fragments are voxelized, while red fragments are not, because they do not cover the fragment center.

The data stored with a voxel fragment is shown in Listing 1, in GLSL syntax. The `color` and `normal` are simply encoded with a standard `RGBA8` format. As for `position`, we use a compact format where 10 bits each are assigned to the x, y and z components. By doing so, we are able to store `position` in a 32-bit `uint`. Before we store values into this format, all three channels are normalized into a range of $[0, 1]$. With this format, 10 bits each are afforded each of the three coordinate components, and storing the position with maximal resolution is important. Because observe that for a voxel resolution of 512^3, at least 9 bits of precision would be necessary, and Crassin et al. [9] use this resolution to produce their results.

However, there are several issues with the above approach, and we will describe how to solve them. The first issue is that a fragment is only generated if the fragment center is covered. Thus, even though a triangle may intersect a cell in the voxel grid, it will not necessarily generate a voxel fragment, as is illustrated in Figure 22. As a result, the voxelized version of the geometry may not accurately represent the actual geometry. Crassin and Green suggest the solution of *conservative voxelization*, which means enlarging the triangle by outwardly shifting the edges of the triangle, in order to ensure that all the centers of the necessary fragments are covered. Conservative voxelization can be implemented by moving the vertices of a triangle so that the triangle edges are pushed outwardly. This algorithm can be realized with a geometry shader.

For Nvidia GPUs of the Maxwell generation and beyond an even easier implementation exists: simply use the extension `NV_conservative_raster` [26, 27]. If enabled, this extension makes sure that fragments covered by geometry will *always* be generated, regardless of whether the fragment center is actually covered.

A simple alternative to conservative voxelization was suggested by Takeshige [28]: when doing the voxelization, enable MSAA. This means that every fragment sample will have a number of subsamples. A possible subsample pattern is illustrated in Figure 23. It is enough that only one of these is covered, for the fragment to be generated. By doing this, the probability that a necessary voxel fragment is not generated is much decreased. In our implementation, we adopted this approach, instead of conservative voxelization, because it was easy to implement, and gave sufficient results for our purposes. In Figure 21b, we can see Crytek Sponza voxelized normally, and observe that the ringed in poles have not been fully voxelized. By enabling MSAA, the probability that such small geometry is not fully voxelized is much decreased, as can be observed in Figure 21c.

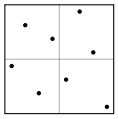


Figure 23: A possible MSAA subsample pattern .

Now let us solve the second issue: if the major axis of a triangle is not the z-axis, then not many fragments will be generated, as is illustrated in Figure 24a. If the triangle in the figure is projected onto the y-axis, then more fragments are generated than if it is projected on the z-axis. So a flat floor that is entirely facing the y-axis would not produce any voxel fragments at all, since the camera used for voxelization is entirely facing the z-axis. The solution to the problem is simple: first, find the major axis. The major axis of the triangle is the one that evaluates to the maximum value for $|\mathbf{n} \bullet \mathbf{v}_{\{x,y,z\}}|$, where $\mathbf{n}$ is the triangle normal, and $\mathbf{v}_{\{x,y,z\}}$ are the three coordinates axes. Then, if the major axis is the z-axis, $\mathbf{v}_z$, we do nothing. However, if the major axis is the y-axis, $\mathbf{v}_y$, we need to make sure that we generate the same number of fragments as when the triangle is projected onto the y-axis. Thus, we just swizzle the y- and z-values in the vertex coordinates of the triangle (like `vc.xyz = vc.xzy` in GLSL syntax), so that the maximum number of fragments will be generated in voxelization, as is illustrated in Figure 24b. Finally, if the major axis is the x-axis, $\mathbf{v}_x$, we instead swizzle the x- and z-values. We can implement this algorithm with a geometry shader, since this is the only stage in the pipeline where we can read and modify the three vertices of a triangle.

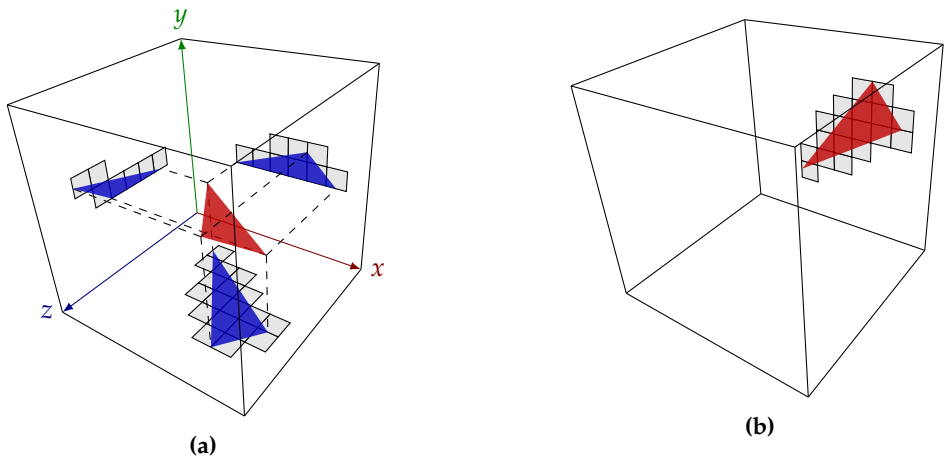


Figure 24: (a) Illustration of the fragments a triangle covers on every axis. As can be observed, because the triangle mostly faces the y-axis, it generates lots of fragments on this axis. (b) By swizzling the y- and z-values, we can project the triangle onto the z-axis, and generate the maximal number of fragments

Listing 2: Octree node code

```
struct OctNode {
  uint nodePtr;
  uint brickPtr;

  uint xPosNeighbor;
  uint yPosNeighbor;
  uint zPosNeighbor;
  uint xNegNeighbor;
  uint yNegNeighbor;
  uint zNegNeighbor;

  uint color;
  uint normal;

  uint centerPos;
};
```

3.2.2 Sparse Voxel Octree Implementation

Data Structure Details

Our octree is stored in GPU memory using a node pool and brick pool, as we described in Section 3.1.1. The data contents of every octree node is shown in Listing 2, in GLSL syntax. The pointer `nodePtr` points to the eight children of the node, and `brickPtr` points to the brick of the node. Every node also has pointers to its six direct neighbors. These neighbour pointers will be necessary in our light injection step, as is discussed in Section 3.2.3

In `color` and `normal`, we store the average color and normal of the voxel fragments that fall into the node. Note that Crassin et al. stored both of these two in bricks, but we have chosen to store a single value in the nodes to save memory. We made the choice to only store the radiance in bricks. Furthermore, we use isotropic instead of anisotropic voxels, and as a result $3^3 = 27$ radiance values are stored per brick. Finally, `centerPos` stores the coordinate of the center point of the node. Having this value stored with the node simplifies our octree construction algorithm.

Note that anisotropic voxels are important when cone tracing wide cones, since only wide cones will sample the uppermost levels in the octree. Therefore, anisotropic voxels will serve to increase the quality of the indirect diffuse lighting approximation. Since we are using isotropic voxels, the quality of our indirect diffuse lighting approximation will be worse, compared to the results of Crassin et al.

The octree construction algorithm used by Crassin et al. [9] will now be outlined. For every generated voxel fragment, the tree is traversed and subdivided on demand until the leaf that contains the voxel fragment is reached. Mutexes are used to ensure that several threads do not subdivide the same node. If a thread can not subdivide due to such a mutex, the thread is interrupted, and put into a list of interrupted threads. The alternative to interruption is a busy-wait loop for that thread, but Crassin et al. found this to be too expensive. Threads from the list are re-executed, until all threads have reached their corresponding leaf.

However, we did not implement the above method, and instead adopted the algorithm described by Crassin and Green [25], because it was easier to implement. We remark that in the thesis of Gomes [29] it is described in much detail how to implement this algorithm, and our implementation draws heavy inspiration from this.

The construction algorithm starts with an octree consisting only of a root node, and then builds the octree one level at a time. For every level, we perform a number of passes, which are described in the following sections

Pass: Flag Nodes

We launch a compute shader thread for every voxel fragment in the voxel fragment list. In the thread, we use the voxel fragment position to traverse the octree until we reach a node with no children. If we have reached the leaf level, we simply flag the node as a leaf node. But if it is an inner node, we must mark it for subdivision, so that it can be subdivided in a later pass. All flagged nodes will be on the same level, and we call this the *active level*.

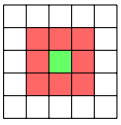


Figure 25: The Moore neighbourhood of the green node are all the red nodes. That is, all of its direct as well as diagonal neighbours.

Once we have flagged a node, we must make sure that all the nodes in the Moore Neighborhood of the node exists, for reasons that will be explained in Section 3.2.3. The Moore neighbourhood is illustrated in Figure 25. Let `centerPos` be the center position of the node, and `vs` the node size at the active level. In order to make sure that the neighbor in the positive x-direction exist, we will traverse the octree until we find the node at the same level, at the position `centerPos + (vs,0,0)`, and flag this node for subdivision. The same is done for

all the remaining nodes in the Moore neighborhood.

Pass: Find Neighbors

For later usage, the nodes on the active level will store pointers to its 6 direct neighbors. Thus, we launch a thread for the nodes on the active level, and find the neighbors. To, for instance, find the x-positive neighbor of some node, we will traverse down the octree until we find the node at the position `centerPos + (vs,0,0)` in the active level. If a node that satisfies this is found, this node is assigned as the x-positive neighbor. Else, it is set to null.

Pass: Subdivision

Next, we launch a thread for every node in the active level. If the node has been marked as a leaf node, we simply assign the leaf a new brick from the brick pool. To ensure that the node is assigned a unique brick, an atomic counter. However, if the node has been marked for subdivision, we assign a brick, and we also store the address to the eight children of the node, thus subdividing the node. The eight next free addresses in the node pool will store the children of the node. We use an atomic counter to keep track of the next free address in the pool. Thanks to the atomic counter, it is not possible that a different node uses the same addresses.

We will repeat the passes "Flag Nodes", "Find Neighbors", and "Subdivision" until we have reached the leaf nodes. Once those have been reached, the octree construction is done, and we will next assign values to the nodes.

Pass: Leaf Value Assignment

After the octree has been constructed, the leaves must be filled with color and normal information. We launch a thread for every voxel fragment, and in every thread we traverse the octree until we find the leaf node that contains the voxel fragment. In the leaf node we want to store the average normal and average material color values of the voxel fragments that it contains. An easy approach to achieving this is adding all voxel fragment contributions into a final sum, and then dividing the sum by the total number of voxel fragments that fell into the node. The alpha channel is used to keep track of the total number of voxel fragments.

However, if colors and normals are stored using the format `RGBA8`, then only 8 bits of precision are reserved for every channel, and hence the risk of overflow is high. To overcome this, we instead compute a *moving average* formula, as was suggested by Crassin and Green [25]. This means every time a voxel fragment falls into a leaf node, we compute the average value of this fragment, and all previous fragments, by updating the cumulative average value with the formula

$$C_{i+1} = \frac{iC_i + x_{i+1}}{i+1}$$

Where C_i is the cumulative average value before updating, and i is the voxel fragment count

before updating. After applying the formula, we must store C_{i+1} in the leaf node with an atomic compare-and-swap operation `atomicCompSwap()`. Further, the alpha channel is used to store i. We use the above average formula for computing the average normal and the average material color. More details about this can be read in the text by Crassin and Green [25].

Dynamic Geometry

It is not difficult to extend the octree construction algorithm so that it handles dynamic geometry. We will voxelize the static geometry only once, and put the nodes of that geometry in the *static section* of the node pool. Directly after the static section we will store the nodes of the dynamic geometry, and this the *dynamic section*. To handle dynamic geometry, at the beginning of every frame we will perform this update algorithm:

1. Clear the dynamic section, by setting everything to null pointers or zero. Also, all pointers to the dynamic section from the static section will be set to null-pointers. Moreover, all neighbor-pointers pointing to nodes from the dynamic section will be set to null-pointers.

2. We voxelize the dynamic geometry and put the voxel fragments in a fragment list.

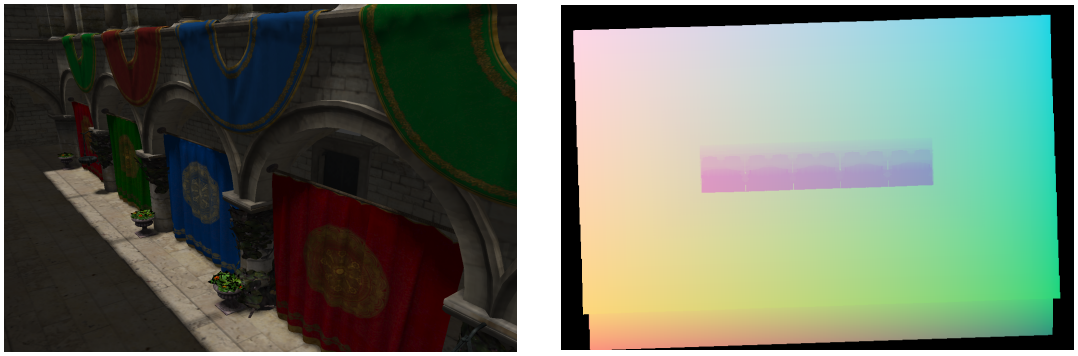
3. Run the passes "Flag Nodes", "Find Neighbors" and "Subdivision", on that fragment list. If we need to subdivide some inner node, then the node pointer of that node will point to some area in the dynamic section. Another complication is that if we reach some leaf node, we must not touch it if it is already in the static section. These leaf nodes have normal and color data from the static geometry of the scene, and if we touched this data we would have to voxelize the static geometry again for correct results, and we only want to voxelize the static geometry once. Thus, only if a leaf node is not used in the static section, will we assign a brick to it, and put it in the dynamic section.

   Observe that since all nodes of the dynamic geometry are stored in the dynamic section, it is trivial to clear the dynamic geometry at the beginning of the frame: we simply remove everything in the dynamic section.

4. Finally, we run the pass "Leaf Value Assignment", in order to fill the leaves with normal and color information. Again, it is paramount that we do not modify the leaves of the static section when doing this.

3.2.3 Light Injection

After we have constructed the octree, we will inject radiance into the bricks of the leaf nodes. For every directional light source, we render a reflective shadow map [16]. These are very similar to ordinary shadow maps [30], except that for every texel in the RSM, we store the world-space position of the texel. In Figure 26, a visualization of an RSM for some directional light source is provided. Every non-empty texel in the RSM represents a position that will be hit by the outgoing radiance of the light source. Thus, for every such texel, we inject radiance into the brick of the leaf node that contains that world-space position. A compute shader thread is launched for every



(a) (b)

Figure 26: (a) A light source illuminates areas in a scene. (b) Visualization of the RSM of the light source. The (x, y, z)-coordinates stored at every texel are visualized as RGB-values.

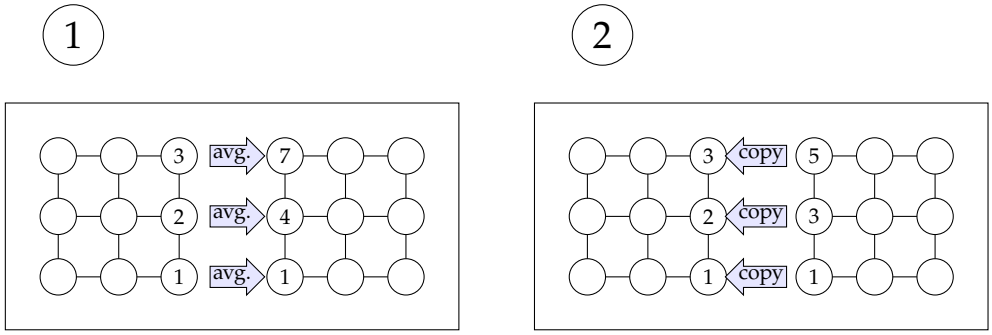


Figure 27: The neighbor transfer algorithm on the x-axis.

texel, and the thread traverses the octree until the leaf that contains that world-space position is found. We then use Equation 2 to compute the radiance contribution to a leaf voxel from a single texel in the RSM, using the average normal and color vectors stored in the leaf voxel. This contribution is then stored in the brick of the leaf-node.

However, since some voxels are shared between the bricks we need to ensure that these shared voxels have the same values. Observe in Figure 10b that the voxel h will be shared by the green and the red brick. However, after the light injection pass, the value of h in the green brick will be the radiance value of the left section of the voxel (the voxels are marked with red dots) while the value of h in the red brick will be the radiance value of the right section of the voxel. This is due to our corner-centered brick-scheme. However, we want h to have the same value in both the bricks, or the interpolation will not be correct. More precisely, we want h to be the average radiance value of the two different values of h in the two bricks.

Thus, after the light injection pass, all the voxels except for g and i (that is, all the voxels on the borders) will have the wrong values. We need an algorithm that transfers the values of the green brick to the red brick in Figure 10b, so that we can correct the value of h. We call this our *Neighbour Transfer Algorithm*. In order to correct the voxels on the x-axis, we will use the algorithm shown in Figure 27; we launch a thread for every leaf-node, and use the right neighbor pointer to access the shared voxels in the neighboring brick, compute the correct average value, and then store

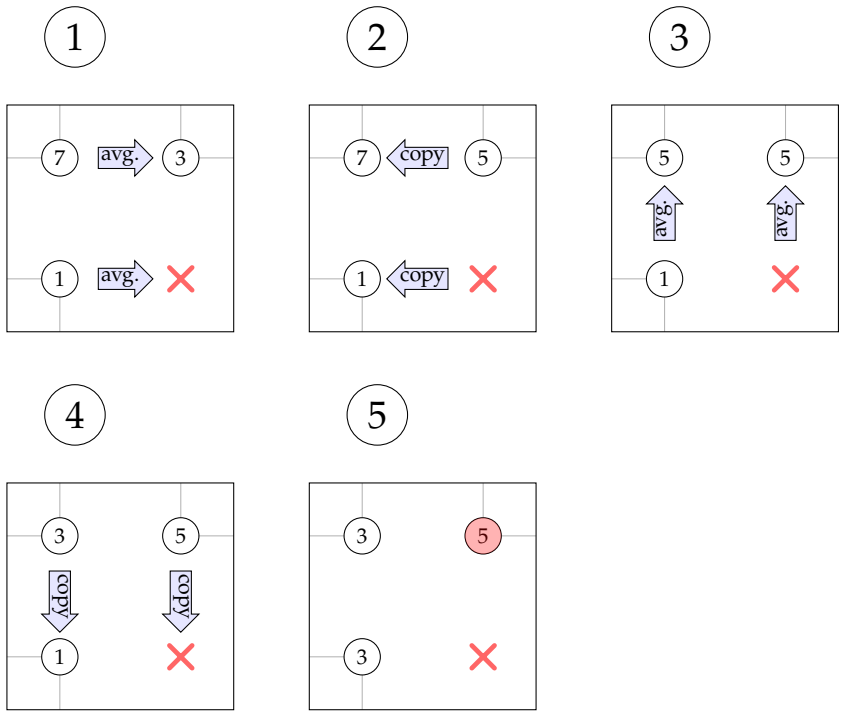


Figure 28: A case for which the described neighbor transfer algorithm fails. Observe that the south-east node is missing, and because of this the algorithm yields incorrect end-results.

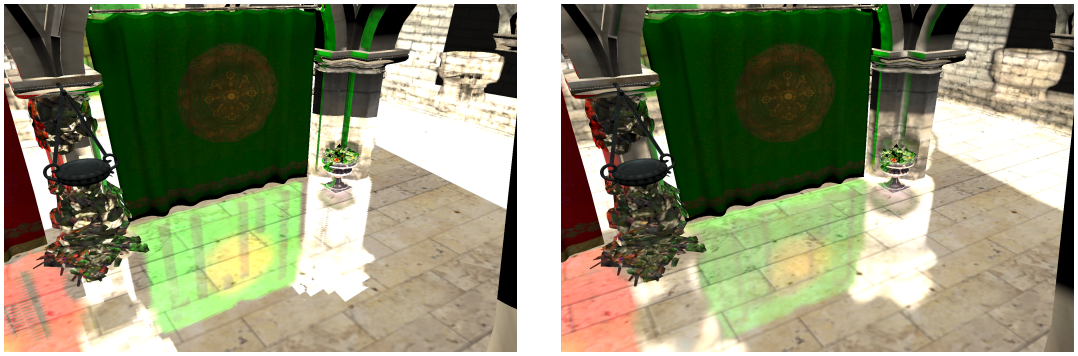
that average value in the neighboring brick. Then, in the next pass we again launch a thread for every leaf node, and now use the left neighbor pointer to transfer the computed average value to the neighbor. With that, the voxels f, h and j in Figure 10b will have their correct values. In order to fix the remaining voxels, we will perform the same algorithm in Figure 27 again, but this time on the y-axis. After that, the values of the shared voxels will be consistent.

The above algorithm was described by Crassin et al. [9] However, their description accidentally omitted a crucial detail, and if this detail is not taken into account, the above algorithm will not yield correct results in all cases. Consider the case in Figure 28. In this case, the south east node does not exist, and as a result, the final voxel values are not consistent across the bricks, even after running the neighbor transfer algorithm. All voxel values should be equal in the final step, but the voxel value in the north east node does not equal the other two.

If the voxels that are shared across the bricks are not consistent, visible artifacts will result, since the trilinear interpolation within the bricks will not be correct. In Figure 29a, these artifacts are illustrated for glossy reflections.

The solution to the problem is to make sure that for every opaque node, all the nodes in its Moore neighborhood must exist (Figure 25), where opaque nodes are the ones that actually contain voxel fragments. If some opaque node lacks a neighbor in its 8 node Moore neighborhood[4], then we create that neighbor node, and set its brick voxel values to fully transparent black. If we do this,

[4]In 3D there are 26 nodes in the Moore neighborhood



(a) **(b)**

Figure 29: (a) Artifacts in glossy reflections due to inconsistent brick voxels at boundaries. (b) After fixing the brick voxels at boundaries, the artifacts go away (the reader may zoom into the PDF in order to see more detailed images).

then the neighbor transfer algorithm will yield the desired end result. The results of this can be observed in Figure 29b.

To summarize, by running the algorithms described in this section, the leaf level bricks will be filled with average radiance values. This is illustrated in Figure 5a, where the original scene can be observed, and in Figure 5b, where light has been injected into the leaves that are visible from the light source.

3.2.4 Mipmap Algorithm

Next, the radiance values must be filtered up the hierarchy, creating a mipmapped octree. Like Crassin et al., we adopt a Gaussian kernel for filtering the radiance values. We will describe an algorithm for computing the mipmapped bricks. Assume that level $L-1$ is the children nodes of level L, and that the bricks of $L-1$ has already been mipmapped. Now we describe an algorithm for computing the mipmapped bricks of level L, using the bricks from $L-1$ as input.

When computing the brick voxel values of some node in level L, the bricks in only the children nodes of that node will not be enough. In Figure 14, brick values from the two left node neighbors will also be used in order to compute the mipmapped voxel λ. Thus, one approach to computing the mipmapped values is to launch one thread for every higher-level node, and use the necessary values on the lower level to compute the mipmapped values. For cases like λ, we will use the neighbor pointers to fetch the necessary values from the neighboring bricks. However, for some nodes these neighbors will not exist, meaning that some parts of the calculation can be skipped, but for other nodes they will exist. But this results in the threads having uneven workloads, which is bad for performance. Another issue is that some computations will be redundantly performed several times. Observe in Figure 14 that the level $L-1$ voxels μ and its south, east and south east neighbors will be multiplied by their weights and added together, in order to compute the level L mipmapped value at μ. However, since μ is repeated in the north, west and north-west neighboring nodes, the same calculations will also be performed for those bricks as well. However, that means that this calculation is redundantly performed four times. In the 3D it

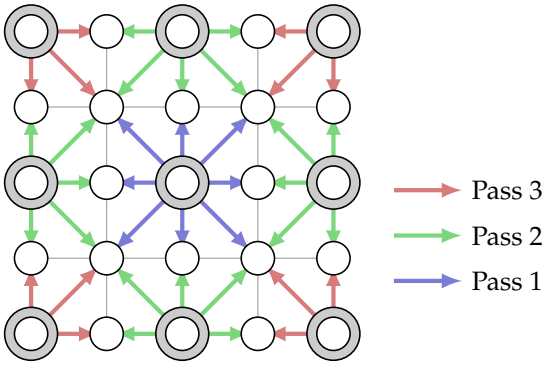


Figure 30: The voxels used in the calculation of the partial mipmap results are indicated by the arrows. Separate colors indicate separate passes

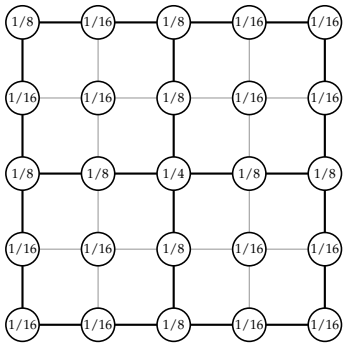


Figure 31: The modified weights of our mipmapping algorithm.

is even worse: a calculation can be redundantly performed up to 8 times.

Thus, we need a more efficient algorithm for computing the mipmapped values. Crassin et al. devised a multi-pass algorithm for this: in Pass 1, a thread is launched for every node in the current level, and the center brick voxel is mipmapped, as is illustrated in Figure 30. In this case, a full mipmapped value can be computed, without having to read any values from the neighbors. However, in Pass 2, we will compute partial results for the side voxels, as is indicated by the green arrows. The voxels indicated by the arrows are weighted and then added together, thus obtaining our partial result. The result is partial, because we have not added the values from the neighboring nodes. In Pass 3, we simply compute the partial results for the corners.

Once all partial results have been computed, the full results will be computed. Note that the level L voxel λ in Figure 14 has a shared voxel in the left neighbor node at level L, and this shared voxel contains one half of the partial result, and λ contains the other half. Thus, we will simply use the Neighbor Transfer Algorithm described in Section 3.2.3 to calculate the complete result. The only difference is that in this case we will not be computing the average, but will be adding the values together.

However, the kernel weights must be modified in order to achieve the correct results. Note in level $L-1$ that λ (Figure 14) and also its north and south neighbors are repeated in the bricks of the

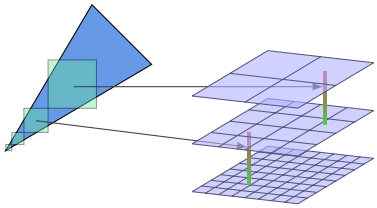


Figure 32: The layers represent levels in the octree. To take cubical samples, one sample each is taken from the two levels that are closest in size to the sample from the octree. Linear interpolation is then performed between the two samples

left node neighbors. But then, we must halve their weights, or the kernel will not be normalized. For the corners we must divide the weight by 4, and the weights of the inner voxels will be left untouched. Our modified weights are shown in Figure 31.

The results of our mipmapping implementation can be seen in Figure 5c and Figure 5d, where the resulting mipmapped radiance values are visualized.

3.2.5 Cone Tracing Implementation

We will implement the cone tracing step as is illustrated in Figure 18. That is, we raymarch along the cone, and take progressively larger, cubical samples. At every such cube, we take a radiance sample from the octree. If the sample is about as big as a voxel, we simply take the sample from the voxel leaf that is closest to that sample. If it is twice the size, we take the sample from the level above the leaves. However, in reality, the size of the sample is something between two levels of the octree. Thus, we take two samples from the two levels which the sample size is between and linearly interpolate between them, as is illustrated in Figure 32. The radiance samples are combined into a single radiance value with Equation (3), as was discussed in Section 3.1.5.

Choice of Cones

In order to approximate a Lambertian diffuse BRDF, we must find cones that approximate a hemisphere (Figure 17b). We choose the *cone direction* of the first cone as $\boldsymbol{\omega}_1 = (0, 1, 0)$, where the cone direction is the vector from the cone apex to the base center. In order to cover the rest of the hemisphere, we will find five other cones that surround the first cone. We let the first of these five cone directions be $\boldsymbol{\omega}_2(0.0, 0.5, 0.866025)$. The angle between $\boldsymbol{\omega}_2$ and the y-axis is $\frac{\pi}{3}$, and some experiments showed that this angle results in a good hemisphere coverage. We now calculate $\boldsymbol{\omega}_3$ by rotating $\boldsymbol{\omega}_2$ $\frac{2\pi}{5}$ radians around the y-axis, we obtain $\boldsymbol{\omega}_4$ by rotating $\boldsymbol{\omega}_2$ $\frac{2 \cdot 2\pi}{5}$, and so on until $\boldsymbol{\omega}_6$. In the end, we obtain the cone directions

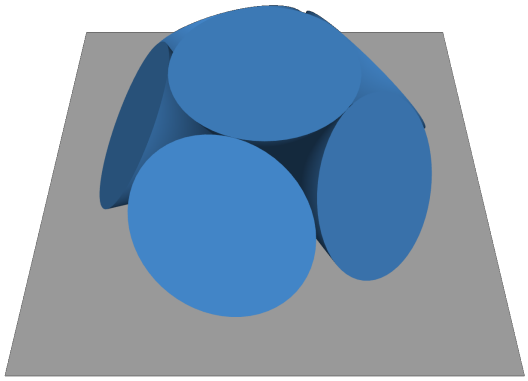


Figure 33: The cones we use to approximate a hemisphere.

$$
\begin{aligned}
\omega_1 &= (0, 1, 0), \\
\omega_2 &= (0, 0.5, 0.866025), \\
\omega_3 &= (0.823639, 0.5, 0.267617), \\
\omega_4 &= (0.509037, 0.5, -0.700629), \\
\omega_5 &= (-0.509037, 0.5, -0.700629), \\
\omega_6 &= (-0.823639, 0.5, 0.267617)
\end{aligned}
$$

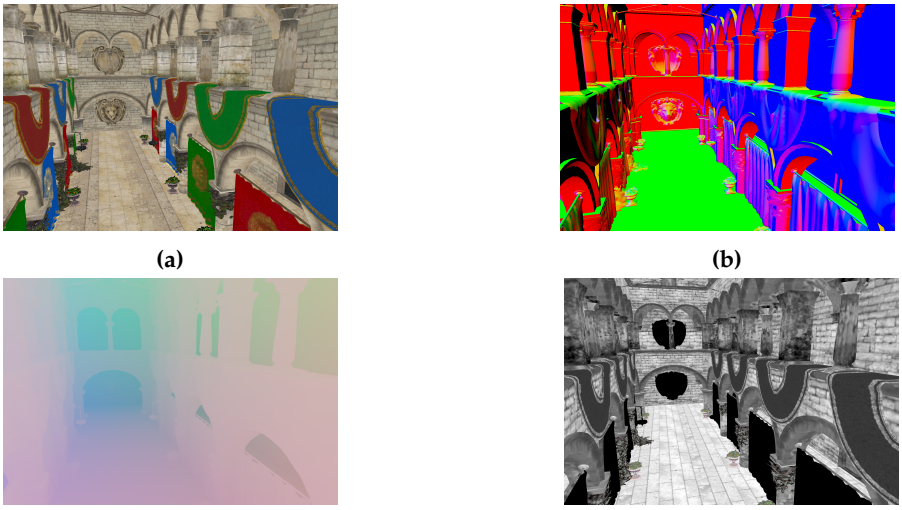
The cones are shown in Figure 33.

Crassin et al. used approximately 5 cones and achieved good results, so we considered that our 6 cones would be a sufficient number. Also, the cones we use are the same cones used by Yeung[31], and he achieved good results using them. For these reasons, we did not spend much time with experimenting with increasing and decreasing the number of cones, and with testing different sets of cones. Instead, we simply used the 6 cones we described above.

We will perform the cone tracing algorithm for each of the 6 cones, and thus obtain a radiance value. We let the weight of the cone number i be $\omega_i \bullet (0, 1, 0)$. By weighting the radiance of each cone by its weight, we compute the sum of the cone radiance values to obtain the final radiance value. The weights ensure that Lambert's cosine law is taken into account.

Deferred Shading

In a naïve implementation, we would render every object in the scene using its material shader, and perform cone tracing for every generated fragment in the fragment shader. We would doing so obtain indirect diffuse and indirect glossy values, and we could use these values in the lighting equations of the material. However, unless we render all geometry front-to-back, this approach would mean that we would redundantly perform cone tracing for fragments that are hidden from view.



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

Figure 34: The G-buffer. (a) Albedo color. (b) Normal. (c) Position. (d) specular color

To solve this issue, we will adopt a deferred shading approach instead. We first render the geometry. The fragment shader will not be doing any shading, but will instead output the albedo color, specular color, geometry normal, and position to a G-buffer (Figure 34). We then execute the cone tracing operation for every pixel on the screen, using the data in the G-buffer as input to the cone tracing algorithm. We use the position to determine the apex of the cones, and we use the normal to rotate the cones so that the cone direction of the center cone in Figure 33 is aligned with the normal. We do this rotation because the hemisphere should be oriented by the normal

4

Clipmap Voxel Cone Tracing

In Clipmap Voxel Cone Tracing(CVCT), the voxels are stored in a clipmap data structure instead of an octree [12]. This means that areas near the camera are represented with a number of high-resolution voxel grids that cover small areas, while distant areas use a number of low-resolution voxel grids that cover a larger area. With such a representation, handling dynamic geometry is much faster, since the clipmap is much easier to update than an octree. However, the data structure also has the major disadvantage that for distant areas the global illumination approximation will be worse. The clipmap representation of a the scene of Figure 35a can be seen in Figure 35b.

A mipmapping mechanism is naturally integrated into the data structure: in order to, for instance, take radiance samples from large cubic volumes(which is needed when cone tracing wide cones), radiance is simply sampled from the low-resolution voxel grids.

Regarding light injection, a RSM could certainly be used to inject outgoing radiance for this technique as well. However, for our CVCT implementation we adopt a voxelization-based light injection method: after a voxel has been created through voxelization, the voxel position is compared with a shadow map rendered from the light source. If it is not in shadow, light is injected; otherwise, nothing is done.

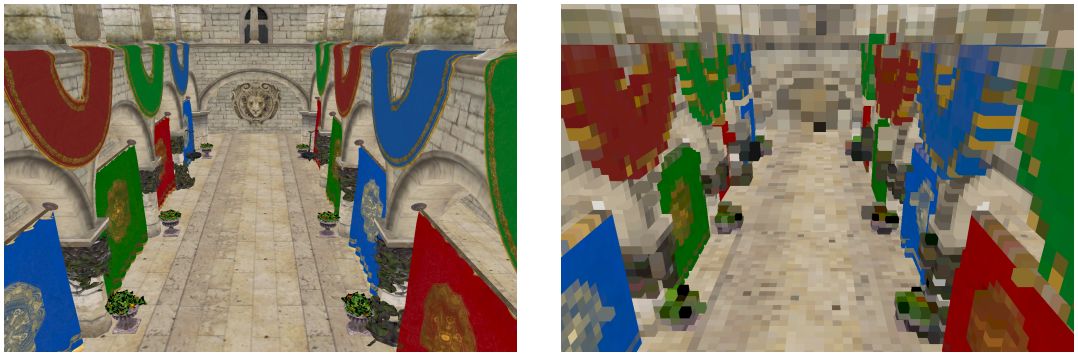
The cone tracing algorithm with CVCT is almost identical to the one of SVOCT, except for one small complication: one must be careful to not sample from voxels that are not stored in the clipmap. Since the high-resolution grids cover only near sections of the scene, one must not sample from these grids when performing cone tracing for distant sections.

4.1 Theory

VXGI is an implementation of CVCT provided by NVIDIA, and in the work of Panteleev [12] the details of VXGI are described. Our implementation draws heavy inspiration from VXGI, and we shall now describe the necessary theory.

4.1.1 Clipmap Data Structure

The clipmap data structure was first introduced by Tanner et al. [32], and they used it as an approach to avoiding storing a large texture in memory. The clipmap is a grid that stores high-resolution voxels near the camera, but lower resolution voxels far away from the camera, as is



(a) **(b)**

Figure 35: (a) The original scene. (b) The scene represented with a clipmap. As can be observed, lower-resolution voxels are used farther away from the camera.

illustrated in Figure 6d. The idea is that since areas that are far away from the camera generally occupy few pixels on the screen, high resolution voxels are not necessary for those areas.

This data structure is implemented with several grids that move with the camera, as is illustrated with a simple example in Figure 36. These grids shall be explained. In order for voxel cone tracing to be implemented using the same algorithm as was used for SVOCT, it must be ensured that every level L voxel, should have 8 voxels in level $L-1$ that together cover its entire volume. This means that every voxel in level L will have 8 children, just like in SVOCT. This is fulfilled in two cases: **(1)** Grid level L and level $L-1$ have the same side-length, but grid level $L-1$ has twice the resolution of grid level L. This is true for level 4 and 3 in the figure. **(2)** The grids have equal resolution, but grid level L has twice the side-length of grid level $L-1$. This is true for level 2 and 1 in the figure.

In Figure 36, it is shown how a clipmap of 5 levels can be built. The level-1 grid has a resolution of 4^3. The level-2 grid has twice the side-length of the level-1 grid, and the same pattern is repeated for the level-3 grid.

Let us assume in our example that the level-3 grid is large enough to cover the entire world. If more grid levels are desired, their resolution must be the half of the previous level, while maintaining the side-length of the previous level, because increasing the side-length would just create redundant voxels. Thus, grid level-4 has half the resolution of grid level-3, and grid level-5 is half the resolution of grid level-4.

Since the camera is in the center of the grids, they must be updated when the camera move. To be more specific, a grid must be updated when the camera has been moved by at least the side-length of one voxel for that grid. An easy approach for performing this update is to simply revoxelize the parts of the scene that the grid contains every time an update is necessary. However, then all the geometry within the grid would have to be revoxelized every time the camera moves.

Panteleev [12] describes how this can be avoided by adopting a toroidal addressing system. A grid is stored in a 3D texture. Toroidal addressing will ensure that every point in world space maps to some texture address, such that every address is in the range $[0, 1]$. We will now explain

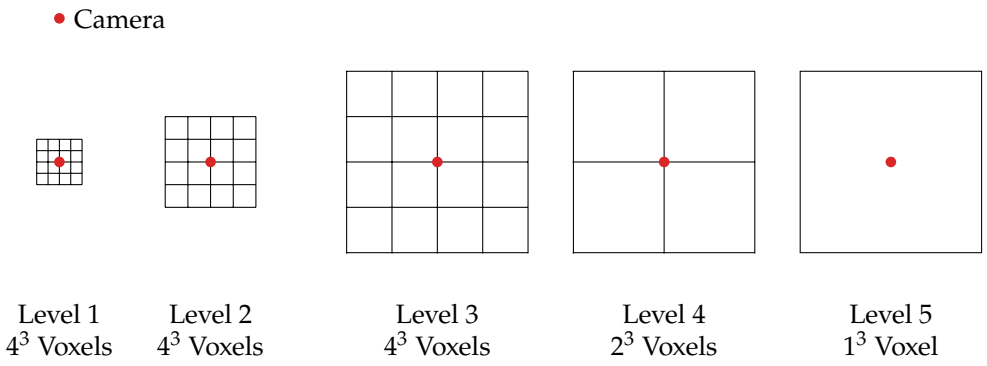


Figure 36: The clipmap is implemented by a combination of several grids.

this addressing scheme on the x-axis only, for some grid of side-length l. Then a point with position x in worldspace will be assigned the texture coordinate

$$t(x) = \text{frac}(x/l).$$

Where the function $\text{frac}(\cdot)$ yields the fractional part of its argument. After a voxel has been created through voxelization, the output texture address of that voxel is simply determined with the function $t(\cdot)$.

The advantage of toroidal addressing is that it makes it easy to update the clipmap. In Figure 37a, a grid at its initial position is illustrated. In Figure 37b, the camera has moved in the positive x and y directions. Then, for the grid to be centered around the camera, the areas A, B and C must be voxelized and added to the grid. Note that the areas E, D and F were once part of the grid, but are no longer. Observe that the toroidal addresses within the area C are equal to those of E, and the same applies for A and F, and then also for B and D is this true. However, this means that if the toroidal addresses of C are written to, then the old values from E will be overwritten. Thus, it is completely trivial to update the grid. Whenever the camera is moved, the new section is voxelized, and the voxels are simply written to the grid using toroidal addressing. Thanks to the toroidal addressing scheme, the old addresses will be overwritten and reused.

The clipmap also makes it easy to handle dynamic objects. If some object changes or moves, an Axis-aligned bounding box (AABB) that contains the object in both the current frame and the previous frame is first calculated. Then, all the geometry contained within this AABB is simply revoxelized. This is illustrated in Figure 38.

Finally, it is not necessary to implement a mipmapping algorithm for CVCT. In order to take large samples during cone tracing, the higher levels are simply sampled; to take small samples, the lower levels are sampled.

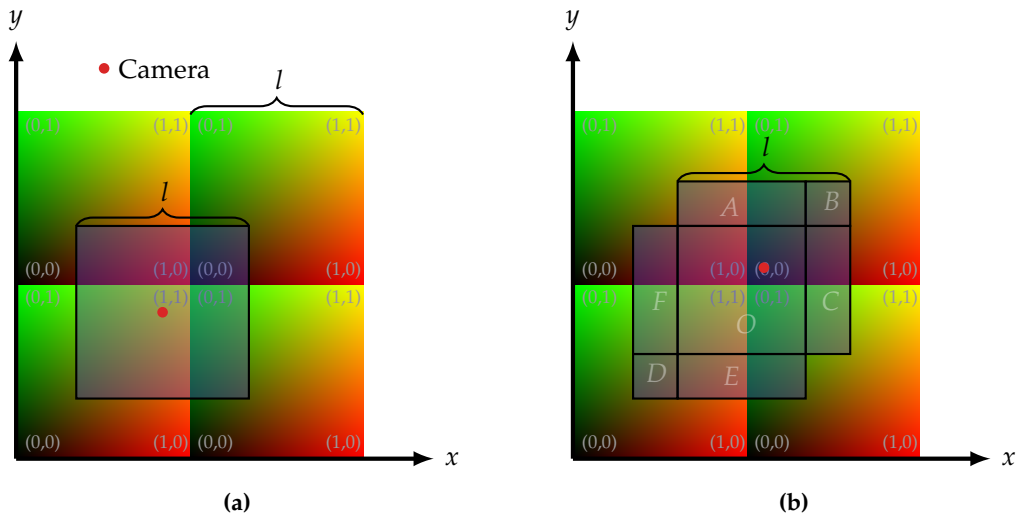


Figure 37: (a) A grid before moving the camera. (b) The grid after moving the camera. The areas A, B and C have to be voxelized, while the areas E, D and F have to be thrown away. Note that the side-length of the grid is l. The toroidal addresses are illustrated with colors, and the gray coordinate labels. The red circle is the grid center.

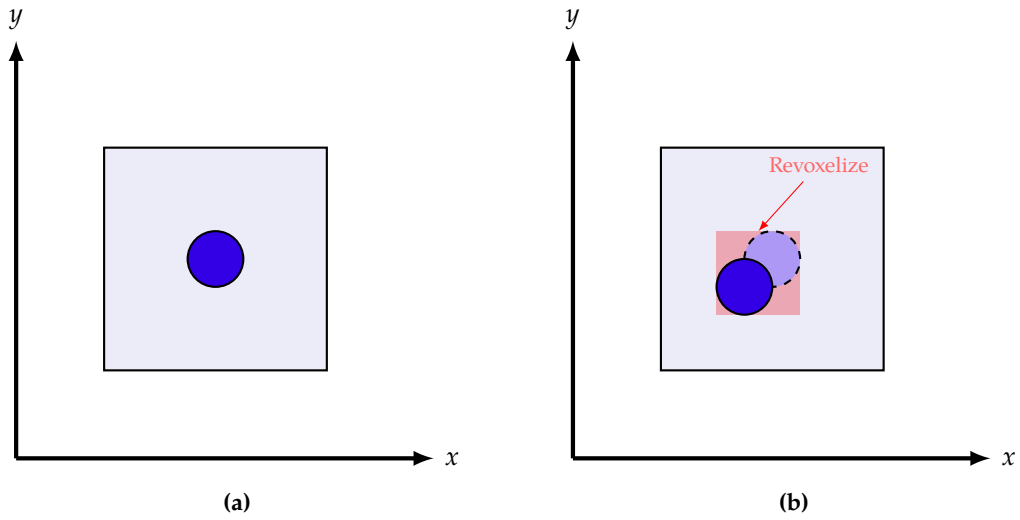


Figure 38: (a) A grid with a sphere object. (b) In the next frame, the sphere is moved in the negative x and y-directions, and thus the AABB encompassed by the sphere in both frames must be revoxelized.

4.1.2 Light Injection

Panteleev [12] suggests several approaches to injecting outgoing radiance in a clipmap, and we shall briefly describe them all.

In the simplest option, we first render the material color, normal and world-space position to an RSM. Then, a compute shader thread is launched for every opaque voxel, and the thread checks whether the voxel center is within shadow. If it is not in shadow, the voxel center is back projected onto the RSM, and the material color and normal in the RSM are used to compute the outgoing radiance of the voxel. However, if the RSM resolution is high, then it is likely that there are more than one RSM texel associated with every voxel, so testing for only the voxel center may not be enough. As a result, there may be undersampling, and thus aliasing issues may occur.

An alternative approach which yields better quality is gathering all RSM texels which belong to the voxel. This could be implemented by first projecting the voxel cube onto the RSM. A two-dimensional AABB is found from the projected voxel cube, and all RSM texels within this AABB are fetched, in order to compute the total outgoing radiance of the voxel. However, this approach has the disadvantage that it will result in many texel fetches per voxel.

There is also an alternative, third approach which potentially has better performance than the previous: the RSM texels are scatted into the voxels with atomic operations. A compute shader thread is launched for every RSM texel, and the thread adds the incoming radiance of that texel into the voxel with the same world position as the texel. The addition must be performed as an atomic add, since one voxel may be covered by several texels. Note that this is the same approach that was used for light injection in SVOCT. This approach avoids having lots of texture fetches for every voxel, since every RSM texel is fetched exactly once. However, if the shadow map resolution is much higher than the clipmap grid resolution, then there will be many texels per voxel, and thus many atomic collisions. This can potentially become a performance issue.

A fourth, voxelization-based approach is to first render a regular shadow map for the light source (containing only z-values), and afterward all objects within the light source frustum are voxelized. During this voxelization, light injection is also performed: if the voxel is not in shadow, then the incomming radiance from the light source is computed and added to the voxel. When voxelizing an object, the normal, material color and world space position are all easily accessible, so there is no need to render an expensive and memory-consuming RSM, a significant advantage. The disadvantage is, however, that all objects within the light frustum must be revoxelized every time the light source moves. This is not necessary with the scattering-based approach.

4.1.3 Sparse Diffuse Cone Tracing

For approximating a diffuse BRDF, several cones must be traced for every pixel, which can be too expensive. Therefore, Panteleev suggests instead doing *sparse diffuse cone tracing*, where cone tracing is only performed for every N:th pixel on the screen, where N is often some integer in the range 4...16. Doing cone tracing for only a subset of the pixels should yield acceptable results, since diffuse lighting is usually low-frequency, and so performing cone tracing for every pixel can be redundant.

Level	Resolution	Side-Length
1	128^3	0.5
2	128^3	1.0
3	128^3	2.0
4	64^3	2.0
5	32^3	2.0

Table 1: The grid settings used for implementation. The side-length refers to the side-length of the grid at the specified level.

These sparse pixel samples are also rotated to produce a rotated grid pattern, as in Rotated Grid Supersampling. This helps alleviate antialiasing for nearly vertical and horizontal edges, which are usually the ones that produce the worst antialiasing artifacts [8]. For obtaining the values of the remaining pixels, a bilateral filter is used for interpolation.

This optimization is not only possible for CVCT, but it can also be performed for SVOCT as well.

4.1.4 MSAA-Based Opacity Voxelization

Panteleev describe a quite sophisticated method for computing the opacity of voxels. They rasterize the geometry with MSAA enabled, thus obtaining subsamples for the z-axis. Then they reproject those samples onto the x- and y-axes. Finally, they thicken the results. By now counting the number of samples on each axis, they are able to obtain opacity values for each axis, and by using an anisotropic voxel representation, they can store these values in the corresponding faces.

4.2 Implementation

In this section, we shall discuss the details of our implementation of CVCT. The grids shown in Figure 36 was but a toy example. The settings of the grid levels in our real clipmap can be seen in Table 1. Note that we have placed the scene in the smallest possible axis-aligned cube that contains it, and then rescaled the scene so that the cube length is exactly 1.0.

This means that for a grid where the side-length is 1.0, the grid covers the entire scene if the camera is in the center of the cube. Further, observe that the side-length is 2.0 for the last three levels. Those grids are then always able to cover the entire scene, even if the camera is in one of the corners of the cube. Finally, we remark that the level 1 grid has a voxel resolution of 256^3 (since $0.5/128 = 1.0/256$). Albeit only within grid level 1 are voxels of this resolution available. Outside this level, only lower-resolution voxels are available, and so the quality will degrade with the distance, as is illustrated in Figure 35b.

In order to provide the voxels for the clipmap, we need to render the geometry at five different

voxel resolutions, since every one of the five grids in the clipmap has a different resolution. We can voxelize geometry with a single drawcall by using the geometry shader instancing feature and the viewport array feature. We use geometry shader instancing to rasterize every triangle five times. However, each one of the five triangles uses a different viewport from the viewport array. The viewport is chosen by assigning a value to `gl_ViewportIndex` in the geometry shader. Assuming we are using the grids in Table 1, the first triangle uses a viewport that corresponds to a voxel resolution of 256^3, the second to a viewport with a voxel resolution of 128^3, and so on for the remaining three triangles. Note that we do not need an intermediate fragment list like in SVOCT, but instead we simply voxelize the geometry and add it to the clipmap with the moving average formula we used for SVOCT in Section 3.2.2. When the camera is moved, the grids are cheaply updated using toroidal addressing, as discussed in Section 4.1.1.

For light injection, we use the fourth, voxelization-based approach described in Section 4.1.2, because it seemed the most promising out of the four alternatives.

For every one of the five levels, we have one grid of opacity values, and one of outgoing radiance values. The opacity grid is a 3D texture with color format `RG8` (the second channel is used for the moving average formula of Section 3.2.2), and the radiance grid is also a 3D texture, and has a color format `RGBA8`. The grid of outgoing radiance values only needs to be updated when some light source has changed position, orientation or intensity. We need to update both grids when the vertices of some dynamic geometry are modified, or when some geometry simply moves.

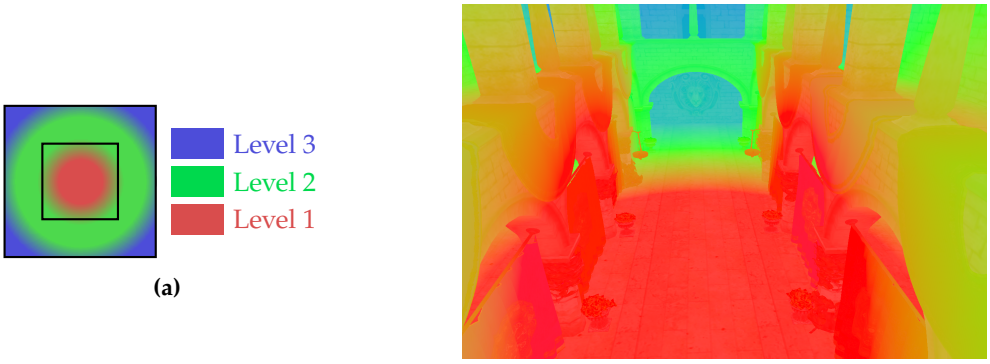
We were unable to implement the MSAA-Based Opacity Voxelization scheme described in Section 4.1.4, because its description was too concise, and several implementation details were missing. Instead, we use isotropic voxels, and set the opacity of a voxel to 1 if it contains at least one triangle, and to 0 otherwise.

4.2.1 Minimum Level Selection

The cone tracing algorithm we use is almost identical to the one of SVOCT, that we described in Section 3.1.5. We did not implement the Sparse Diffuse Cone Tracing described in Section 4.1.3, due to time constraints.

The smallest possible voxels are the ones stored in grid level 1, and the largest voxels are stored in level 5. When taking radiance samples in the cone tracing process, we will sample from the two levels that are closest to our sample cube, and interpolate between them. However, there is one complication: if, for instance, we are tracing cones for positions that are outside the grid of level 1, we must not take samples from level 1, since that level stores no data for those positions. Level 1 is the smallest, so usually the first few samples are taken from this level, but for CVCT we must make sure that we do not sample from this level for the distant areas. We call the *minimum level* the smallest possible level we will sample from when cone tracing, and for positions outside the grid of level 1, the minimum level must be at least two.

In Figure 39a, our method of choosing the minimum level is illustrated. As can be observed, it is based on the Euclidean distance from the camera (which is also the grid center). This method was first proposed by McLaren [10]. With this scheme, we are wasting a bit of data at the corners



(a) **(b)**

Figure 39: (a) How we choose the minimum level for the first two grid levels. (b) Our grid selection scheme illustrated for a real scene

of the grid, but it is necessary that we use an Euclidean distance, otherwise there would be more resolution in some directions than others. Furthermore, we are smoothly transitioning from one level to the other, by linearly interpolating based on the Euclidean distance. If we did not interpolate, the transition from one level to the other would be jarringly sudden, and there would be visible artifacts. In Figure 39b, our grid selection method is illustrated for a real scene.

5

Results

In this section, we will first present the results of our implementation of the techniques described in Section 3 and Section 4. Then we shall perform a comparison between the two techniques. All comparisons shall be performed on the card NVIDIA GTX 1070.

5.1 SVOCT

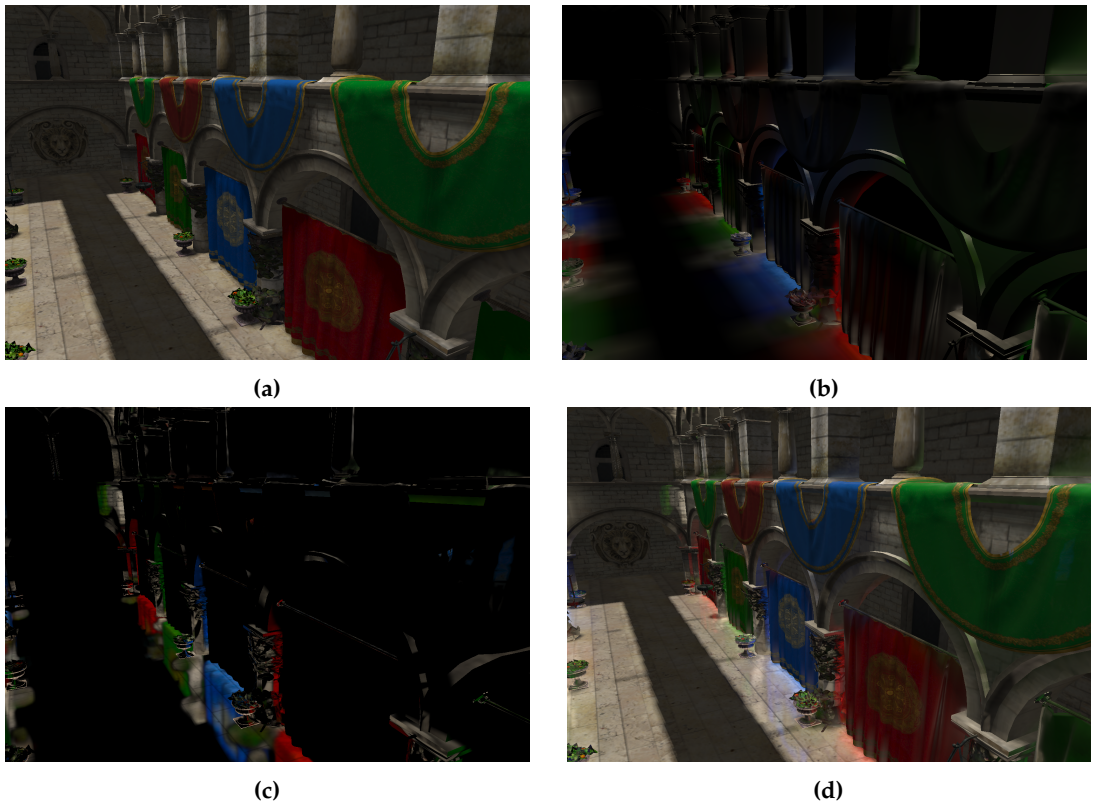


Figure 40: SVOCT results. (a) Direct. (b) Indirect diffuse. (c) Indirect specular. (d) All the three combined.

Images generated with our implementation of SVOCT can be seen in Figure 40. In the figure, the cone-traced indirect specular and indirect diffuse lighting can be observed, and the appearance of this indirect lighting added to the direct lighting can also be seen. Note that the indirect lighting

has been exaggerated in the figure, so that it is more apparent. The indirect specular lighting allows us to render reflections, and the indirect diffuse lighting results in color bleeding effects, which can be seen on the pillars in the image.

The voxel resolution is 256^3, the image resolution is 1497x1014. There are three directional light sources in the test scene. We let a camera fly through the scene, capturing a frame time for every frame, and we obtain an average frame time of 18.66*ms* for this fly-through. If we make the three light sources dynamic, so that the light-filtering step must be performed at the beginning of every frame, our frame time is 91.49*ms*. There are no dynamic objects for this scene, because there was not enough time to implement light filtering for dynamic geometry. But we have written code for updating the octree with dynamic geometry, and the update performance for the octree shall be compared with the clipmap in our comparison. More detailed performance numbers shall be provided in the comparison of Section 5.3. Unfortunately, both the above times exceed our frame budget, so our implementation needs further optimization before it can be used in real-time scenarios.

Finally, the octree node pool and brick pool in total takes up 278 MB of GPU memory.

5.2 CVCT

Images of our implementation of CVCT can be seen in Figure 41. We are using the settings specified in Table 1, achieving a average frame time of 5.44*ms* for a camera fly-through, for a window resolution of 1497x1014. If two dynamic sphere objects that move back and forth are added, then the frame time becomes 5.89*ms*. If we furthermore make the three light sources be dynamic, the frame time is 10.05*ms*. This time is within our frame budget, so our implementation suitable for real-time scenarios. More detailed performance numbers shall be provided in the comparison of Section 5.3.

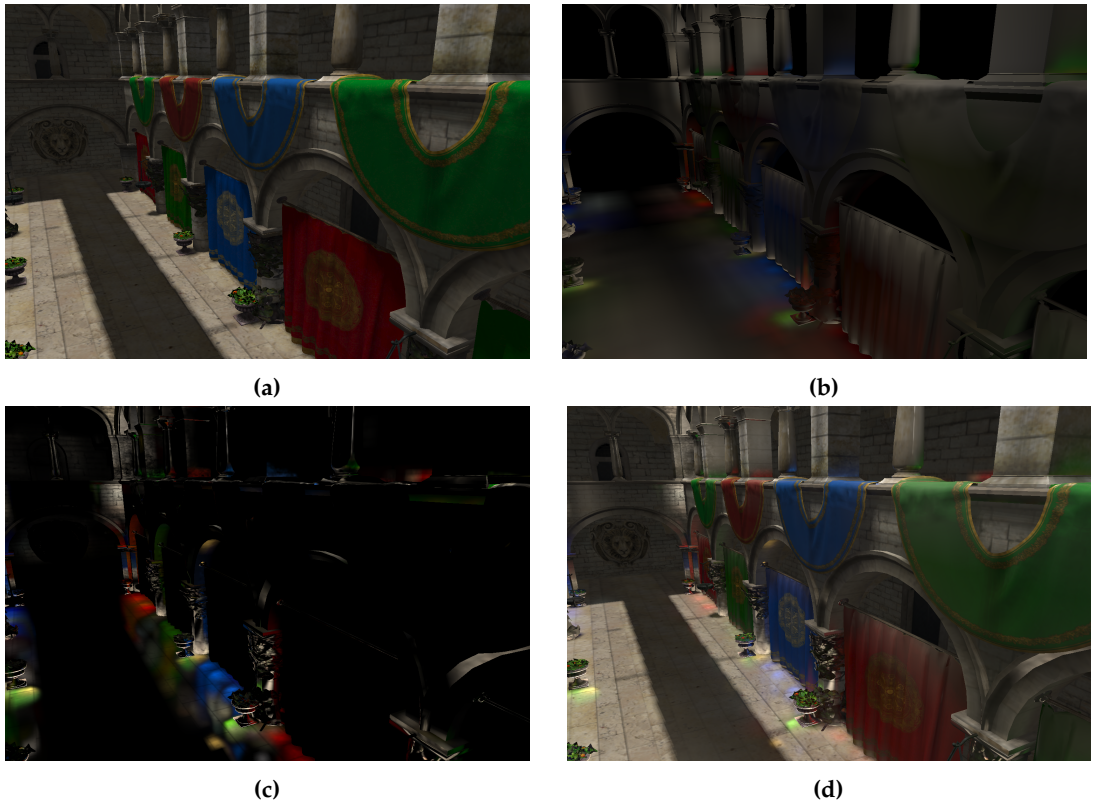
Unfortunately, an issue with the technique is that the indirect lighting changes as the camera is moved. This is to be expected, since camera movement causes distant areas to use lower resolution voxels for their indirect lighting approximation. The artifact is unfortunately relatively noticeable in our implementation.

It appears that lighting changes is an intrinsic part of the technique. Two other implementations of voxel cone tracing using clipmaps is the VXGI library provided by Nvidia [11], and an open source implementation provided by Kröker [33]. We have tried out both implementations, and they also suffer from the problem of indirect lighting changing with camera movement.

5.3 Comparison

In this section, we shall perform a comparison between CVCT and SVOCT.

Both techniques will be compared in a number of *test cases*. In each test case, certain aspects of both techniques will be examined and compared, such as how well they handle scenes with many



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

Figure 41: CVCT results. (a) Direct. (b) Indirect diffuse. (c) Indirect specular. (d) All the three combined.

dynamic objects. First, we will in detail describe the common setting of all test cases.

All test cases were performed in the Crytek Sponza scene [34]. The scene is unmodified, except for that we have centered it on the origin, since this simplified several implementation details.

We call an octree with resolution 256^3 a 256^3-octree. We call the clipmap specified in Table 1 a 256^3-clipmap. This is because the lowest-level grid(level 1 grid) has the resolution 256^3 (since $0.5/128 = 1/256$. Recall that the longest side-length of the AABB that exactly contains the scene has the value 1). Thus, a 256^3-clipmap has the same resolution as the leaf-voxels of a 256^3-octree, within the lowest-level grid. Similarly, a clipmap with the same side-lengths as in Table 1, but grid resolutions, 64^3, 64^3, 64^3, 32^3, 16^3 is denoted a 128^3-clipmap.

In some of our test cases, a 256^3-octree is compared to an 256^3-clipmap, and a 128^3-octree is compared to a 128^3-clipmap, and so on. It may be argued that comparing these is slightly unfair, since the quality of the clipmap degrades with the distance to the camera. However, this is taken into account in the test case of Section 5.3.6.

We use the OpenGL extension `ARB_timer_query` to obtain accurate timing information of code executed on the graphics card, and use this data to do performance measurements. All performance measurements are given in milliseconds per frame.

For many applications, such as games, a framerate of 60FPS is desirable. This means that only about 16.6 ms of computation time is available per frame. We call this our *frame budget*, and if some computation of either SVOCT or CVCT occupies a significant portion of this budget, we call this computation expensive.

5.3.1 Dynamic Geometry

In this test case, the difference in performance for a scene with many dynamic objects is compared. A number of spheres are placed in our test scene, and these spheres constantly move back and fourth, so that updating their voxel representations will be necessary every single frame. For each of the two methods, we will perform five tests with a different number of spheres, in order to examine how well they scale as the amount of dynamic geometry is increased. In our five tests we will use: 1, 9, 27, 64, and 125 spheres.

With CVCT, the voxel fragments can simply be added to the grids of the clipmap with an atomic add operation. However, in SVOCT, the octree must first be updated before the voxel fragments can even be written to the leaves. We are particularly interested in how much overhead updating the octree introduces. Therefore, we will in our tests write only to the lowest level. For SVOCT, we update the octree, and then write only to the leaves; with CVCT, we simply write to the grid with the lowest level. For the remaining levels we do nothing. By only writing to the lowest level, the cost from writing to the remaining levels is not included, and so it becomes easier to examine how much impact rebuilding the octree exactly has.

The octree has a resolution of 256^3, and the lowest-level clipmap grid also has the resolution 256^3, for the purpose of making the test fair.

Frame Time(ms)		
SVOCT	CVCT	Num. Spheres
4.22	0.14	1
4.38	0.85	9
4.83	3.12	27
6.83	8.04	64
22.44	15.36	125

(a)

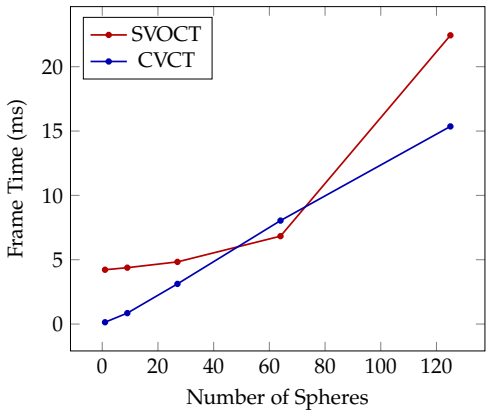


Figure 42: Test case results for dynamic geometry. (a) Test data. (b) Plot of test data.

The results of the test case can be seen in Figure 42. We can see that even for only one sphere, the overhead of updating the octree is significant. There is an initial overhead of about 4.2 ms, due to the octree update.

For CVCT, the frame time increases more than for SVOCT as more spheres are added. This is because in our implementation of CVCT, we make no distinction between dynamic and static geometry. In every frame, for every dynamic sphere, an AABB that contains the dynamic sphere in the previous and current frame is defined, and everything within this AABB is revoxelized. Thus, if some static geometry of the test scene intersects the sphere, then also that geometry will also be revoxelized. In SVOCT, we made sure not to ever overwrite static geometry with dynamic geometry.

For CVCT, a scene with 125 dynamic spheres has a frame time of 15.3 ms. Unfortunately, this is almost the entire frame budget. Modern games tend to have much dynamic geometry, so these results indicate that both techniques may not be very suitable for approximating global illumination for such games, without further optimizations or approximations.

5.3.2 Memory Consumption

This test case compares the differences in memory consumption of the two methods, for our test scene. For reasons unknown to us, the graphics drivers of the hardware would not allow us to allocate enough memory for an octree of resolution 1024^3 or higher. Therefore, measurements were only performed for the resolutions 64^3, 128^3, 256^3, 512^3. If these data points are plotted with a logarithmic scale, then it can be seen that the points follow a straight line. Therefore, we can extrapolate to approximate the data points for the remaining resolutions. The results can be seen in Figure 43.

Our implementation of SVOCT surpasses CVCT only at a large resolution of 8192^3. However, an octree of size $155296MB = 151GB$ is not practical, as our graphics hardware only has $8GB$ of memory. Thus, for the voxel resolutions that are usable in practice, CVCT appears to be superior. However, note that we did not implement an important size-saving optimization which will be

Memory Consum.(MB)		
SVOCT	CVCT	Res.
11.3	0.58	64^3
58.4	4.71	128^3
287	37.69	256^3
1320	301.5	512^3
6468	2412	1024^3
31693	19296	2048^3
155296	154368	4096^3
760953	1234944	8192^3
3728673	9879552	16384^3

(a)

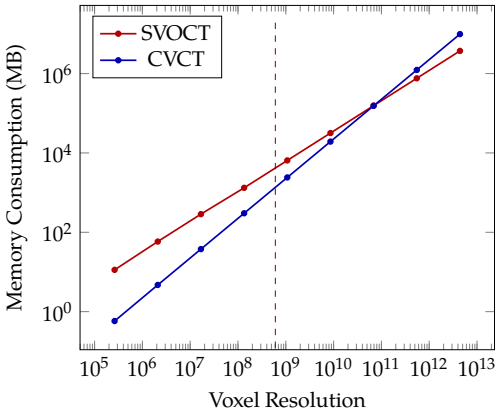


Figure 43: Test case results for memory consumption. (a) Test data. Values with gray background are extrapolated values (b) Plot of test data. All data points on the right side of the dashed-line are extrapolated data points.

discussed in Section 5.4, and this may also have made our results for SVOCT worse.

5.3.3 Clipmap Moving Camera

An issue with the clipmap is that it must be updated as the camera moves. The faster the camera moves, the more geometry must be voxelized every frame. This may cause noticeable performance issues if the technique were used in a fast-paced game. Therefore, it was examined how costly the clipmap update is for a fast-moving camera. In the test case, the camera is moving alongside an ellipse that covers the entire test scene. There is only static geometry in the test scene, and the test was performed for five different camera velocities: 448m/s, 1121m/s, 2242m/s, 3363m/s, and 4485m/s. The unit of measurements for Crytek Sponza is not included with the scene, but we have assumed that the unit is meters. Regarding the size of the scene, we state that the smallest possible AABB that contains the scene has the side-length 3720 meters.

The results of the test case can be seen in Figure 44. As can be observed, the overhead introduced from the voxelization is not very expensive. Thanks to the toroidal addressing scheme, only a small part of the scene has to be voxelized and added when the camera moves. A camera that moves at velocity 4485m/s is very fast, but even so the cost is only 1.2 ms per frame. This is only 7% of the frame budget. Finally, with the octree this cost will occupy 0% of the frame budget, since the octree already contains the entire voxelized scene.

5.3.4 Dynamic Light Source

Both the methods we have implemented use quite different approaches to injecting radiance into the data structure. With SVOCT the radiance is first scattered into the octree leaves from the RSM of the light source, and then the radiance is filtered up the hierarchy with a Gaussian kernel.

Frame Time(ms)	
CVCT	Camera Vel.(m/s)
0.85	448
0.99	1121
1.12	2242
1.17	3363
1.21	4485

(a)

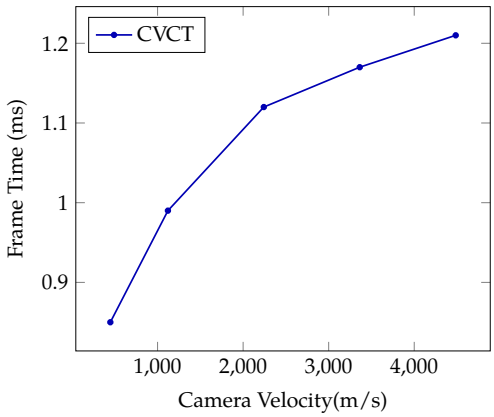
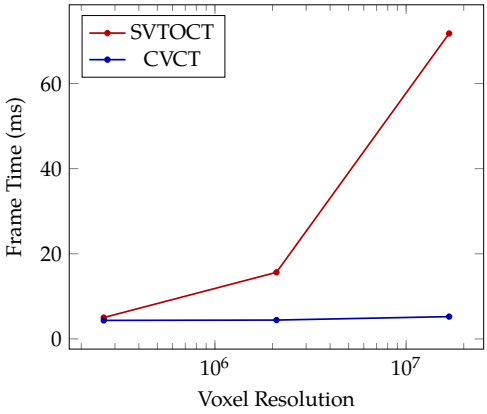


Figure 44: Test case results for moving camera. (a) Test data. (b) Plot of test data.

Frame Time(ms)		
SVOCT	CVCT	Res.
5.02	4.35	64^3
15.64	4.43	128^3
71.75	5.24	256^3

(a)



(b)

Figure 45: Test case results for dynamic light source. (a) Test data. (b) Plot of test data.

On the other hand, in CVCT the geometry is voxelized, and a shadow map is used to determine whether radiance should be injected into the voxel.

In a scene with very dynamic light sources that changes every frame, the above light injection methods would have to be repeated every frame. It is desirable that both methods are fast, and we will in this test case render a scene with one dynamic light source that constantly changes position, and compare the performance for both methods. With SVOCT, the higher the resolution, the more levels there are in the octree, which means more mipmapping computations. Because the radiance is mipmapped up the octree one layer at a time. For that reason, the comparison shall be done for three different resolutions, so that we can see how much the number of levels affect the performance. Those resolutions are 64^3, 128^3, and 256^3.

The results of the test case can be seen in Figure 45. CVCT scales very well as the resolution is increased. A resolution of 256^3 has a frame time of 5.24 ms, which is about 30% of the frame budget. This is quite a large portion of the budget, so the current implementation may need more optimization if it is to be used in a game with a framerate of 60FPS.

Frame Time(ms)		
SVOCT	CVCT	Res.
4.16	0.96	64^3
7.11	1.96	128^3
13.19	3.36	256^3
135.85	6.05	512^3

(a)

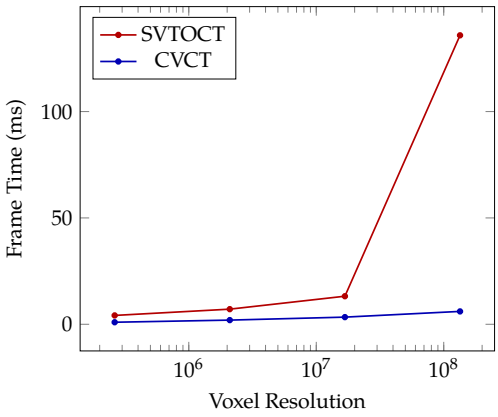


Figure 46: Test case results for cone tracing. (a) Test data. (b) Plot of test data.

For SVOCT, the performance is worse than CVCT. A frame time of 71.75 ms is more than four times the frame budget. We were unable to implement an important optimization, which may have contributed to these results, as we will discuss in Section 5.4.

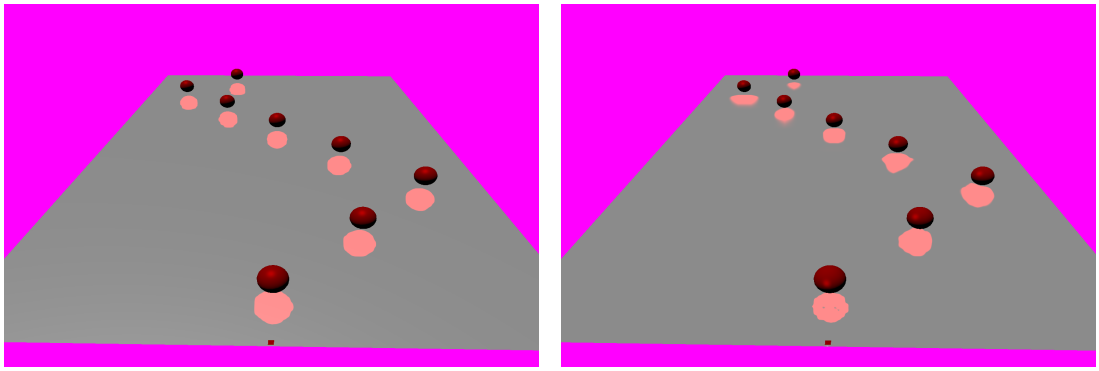
5.3.5 Cone Tracing Performance

The performance of the cone tracing algorithm will be compared for both techniques. In the case of SVOCT, the octree must be traversed before a sample can be taken, and this introduces a certain of amount of overhead to the cone tracing. In the test case of this section, a very slim cone is traced for a glossy specular lighting component, and no cones at all for the diffuse component. Since the cone is slim, only very small samples will be taken, and so the octree must be traversed to the leaves for most samples. Thus, by cone tracing a very thin cone, we can see how expensive the cone tracing is in the worst case. Also, we make sure that no cone tracing is performed outside the lowest level of the clipmap. It would otherwise be unfair, since outside this lowest level the voxels are larger, and thus the step length in the cone tracing will also become larger. This means that the cone tracing would terminate earlier than SVOCT, which is unfair.

In the cone tracing algorithm illustrated in Figure 18, the cone tracing process stops once the opacity value reaches a value of one. Unfortunately, due to small bugs in our implementation, the voxelizations of the test scene in CVCT and SVOCT are slightly different, so stopping once transparency reaches one may be unfair. For this reason, the cone tracing process is only stopped once the cone has been raymarched a certain length. This means that the exact same amount of raymarching will be performed for all pixels on the screen.

The results of the test case can be seen in Figure 46. As can be observed, the cost of traversing the octree is non-trivial; for a resolution of 256^3, SVOCT takes 13.2 ms and CVCT takes only 3.4 ms. With CVCT, no expensive octree traversal is necessary, and so it is faster.

For a resolution of 512^3 for SVOCT, the frame time is 135 ms, which is ten times the time for the resolution 256^3. We suspect that this result is due to bugs in our code.



(a) (b)

Figure 47: (a) Ground truth reflection. (b) Reflection achieved with CVCT. The tiny red square marks the center of the clipmap.

For a resolution of 512^3 for CVCT, the cone tracing takes 6.0 ms, which is 36% of the rendering budget, a significant cost. However, this can cost possibly be reduced by an empty space skipping scheme, which is discussed in Section 5.4.

5.3.6 Reflection Comparison

A major drawback of CVCT is that the quality of the indirect lighting decreases with distance, since distant areas are only covered by low-resolution grids. This is a drawback that SVOCT does not have.

In this test case, a number of spheres, and a reflective plane will be rendered. A very thin cone will be raymarched to achieve a mirror reflections in the plane, and the reflection we achieve using SVOCT will be considered the "ground truth". We will render the same reflections using CVCT, and compare the achieved reflections to the "ground truth" reflections. The resolution for both data structures is 256^3.

The results of the test case can be seen in Figure 47. The reflections of all the spheres should be identical, as in Figure 47a. However, with CVCT, the quality of the reflections degrade with distance. This is a major drawback of CVCT.

5.4 Limitations

There were certain features of both techniques that could not be implemented due to time constraints, and the lack of these features may have affected the above comparisons. These features will now briefly be outlined.

Crassin [21] remarks that for almost homogeneous regions, using bricks is excessive. If all the voxels in the 3^3 brick are almost equal, then instead storing it as a single voxel saves much

memory(Figure 48). This optimization might have improved our result for SVOCT in Section 5.3.2. Furthermore, if a region is represented using a single voxel instead of a brick, then there is no need to raymarch across this entire region in the cone tracing algorithm. Instead, the result of raymarching this region can be directly computed, and the region is then skipped. Implementing this optimization might have improved the results of Section 5.3.5.

With every octree node we store 11 32-bit integers(Listing 2). However, a more compact node storage format is likely possible. For instance, Yeung [31] stores only 7 32-bit integers in his implementation of SVOCT. If we had better optimized for size, better numbers would have been obtained for SVOCT in Section 5.3.2.

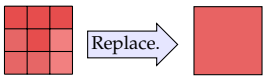


Figure 48: A nearly homogeneous brick can be replaced with a single voxel.

Unfortunately, little time was spent on optimizing our implementations of SVOCT and CVCT. In particular for SVOCT, quite a few compute shaders must be executed to update the octree structure, and more time should have been spent on optimizing these shaders. One important optimization that might yield great gains is storing the octree node pool as a structure of arrays. In our implementation, the node pool is stored as an array of `OctNode` structs(Listing 2). However, in certain cases storing every field of that struct in a separate array is more cache-friendly, and therefore possibly more performant.

A problem with our mipmapping algorithm described in Section 3.2.4 is that the filtering step is executed for every single node in the tree. However, this is redundant, since only nodes visible from the light source will contain non-zero values. Therefore, Crassin et al [9] use the RSM to find the nodes for which filtering is actually necessary. In order to filter level L, a compute shader thread is launched per texel in the RSM, and the octree is traversed until a node that contains the position of the texel in level L is found, and then the filtering is done for that node. This will yield correct results, but much redundant work is still performed, since duplicate work is done for threads that end up in the same node. There is a especially a lot of duplicated work for the higher levels. Crassin et al. [9] describe an heuristic that finds and terminates threads with such duplicated work, and we refer to their text for a detailed description.

6

Discussion and Conclusion

6.1 Discussion

In theory, the performance of both techniques should be nearly independent of the complexity of the scene. This is what Crassin et al. claimed in their original paper [9]. This comes from the fact that the tessellation level of some geometry has little impact on how many voxels that geometry covers, as is illustrated in Figure 49. As a result, the cone tracing process will not become more expensive just because the geometry is more complex, because the number of voxels is still the same. However, our test case of Section 5.3.1 reveals this to be false for dynamic geometry; as the amount of dynamic geometry increases, the cost of updating both structures grows.

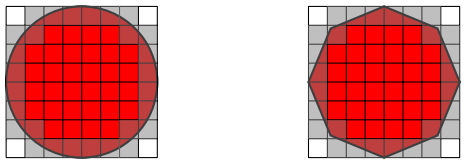


Figure 49: Surface voxelizations of two spheres, one with high tessellation and the other with low tessellation. The darkened grid-cells are part of the voxelizations. As can be observed, both versions cover the same number of voxels.

We found that the octree adds a substantial amount of overhead. In Figure 42, it was shown that there is an initial overhead to updating the structure. It is of course natural that such an overhead exists, because the octree is much more complex than the clipmap, and therefore such an overhead will always exist, no matter how much optimization is performed. The overhead also makes it slower to query information from the structure, and it therefore makes the cone tracing slightly slower, as was demonstrated in Figure 46.

The main reason for using the octree is saving memory. Unfortunately, as can be seen in Figure 43, our octree implementation is smaller than a clipmap only at impractical voxel resolutions. We suspect that the octree of Crassin et al. [9] is much more compact than ours, and that they aggressively simplify homogeneous bricks into single voxels. Additionally, a more compact octree structure means less bandwidth requirements, and this possibly makes the traversal of their octree more performant.

However, the clipmap also suffers from the major disadvantage that the quality of the indirect lighting decreases with the distance to the camera. This may not, however, be visible in all scenes.

For instance, in a scene with lots of obstacles that blocks the line-of-sight, so that distant areas are much less visible, the degradation of quality may not be as apparent. However, in scenes that are very open, like an open grass field, there will be few obstacles to block the view, and the artifact would be very noticeable in such scenes. So the technique may be less suited for such situations.

If we compare Section 4 and Section 3, we realize that SVOCT much more complex than CVCT, and involves much more implementation effort. For this reason, we advise readers interested in implementing voxel cone tracing to first implement CVCT before even considering SVOCT, since CVCT is much simpler, and if it provides good enough quality, then much time would be saved.

6.2 Future Work

We would like to extend our implementation of SVOCT by adding support for anisotropic voxels. Also, as discussed in Section 3.1.2, in our light injection method we assumed that all light-injected voxels have the reflective properties of a Lambertian diffuse BRDF. But this assumption does not hold if light is injected into a voxel that represents a surface with glossy reflective properties, and so it is not physically correct. For future work, we want to explore how to handle this case.

Our current implementation only supports two bounce indirect illumination. However, if we somehow after injection let the injected radiance bounce one time, we could increase the bounce-count to three. The details of how to implement this are not entirely clear to us yet, but we would like to explore it.

In Table 46, it can be seen that cone tracing a thin cone is quite expensive for high resolutions. Thin cones are necessary to achieve mirror-like glossy reflections. It may be more cheap to instead use a screen-space approach, like the one of Mara et al. [18](Section 2.2), for glossy reflections, and using cone tracing only for indirect diffuse lighting. Of course, this would have the artifacts associated with all screen-space approaches, but we think it would be interesting to explore how much performance could be gained.

There are several different light injection approaches that still need to be explored. We were only able to explore one of the four approaches of CVCT in Section 4.1.2. For SVOCT, we suspect that it is possible to implement a light injection approach that is similar to the one we used for CVCT. That is, for every light source, we store the information that is necessary to calculate the radiance, such as material color and geometry normal, in an RSM. To then inject light into a voxel, we check if that voxel is in shadow or not, from the perspective of the light source of the RSM. If it is in light, the radiance of the voxel is calculated using the information in the RSM. With such an approach, it is not necessary to store the geometry normals and material colors in the nodes, and much memory would be saved. Note that we are unsure whether this approach is actually feasible, because an implementation has not been attempted.

6.3 Conclusion

We have in this thesis implemented two different data structures for voxel cone tracing: an octree, and a clipmap. As can be seen from our description in Section 3, maintaining an octree on the GPU involves quite a few steps. Quite a few compute shaders must be executed in order to the build hierarchy, inject radiance, and then mipmap that radiance up the hierarchy. For usage in real-time scenarious, much optimization work still remains to be done on these compute shaders. Nonetheless, our version implements a majority of the features described by Crassin et al [9]. We also consider that our description of the technique is much more complete than what exists in the current literature. To our knowledge, we are the first to provide a detailed description of the mipmapping algorithm of Section 3.2.4. We are also the first to discuss how to implement a working neighbour transfer algorithm, that avoids the issue illustrated in Figure 28.

In constrast to the octree, the clipmap is a much more simple data structure, and required much less implementation effort. The performance of our implementation allows it to be used in real-time scenarious. Of course, the main drawback of the technique is that the quality of the indirect lighting degrades with the distance to the camera, and that the indirect lighting changes as the camera moves, but these are instrintic parts of the technique that we could do little about. There exists very little literature on using a clipmap for voxel cone tracing, and we consider that our description of the technique should make it easier for people to implement it in the future.

To conclude, we shall answer our original research question, which was posed in Section 1.3: we discovered that the complex octree data structure incurs much overhead that causes it to perform worse than the clipmap, both with respect to memory consumption and performance. However, with respect to visual quality the clipmap is inferior to the octree, because the clipmap does not provide the same voxel resolution everywhere in the scene.

6.4 Ethical Considerations

Advancements in the field of global illumination will ultimately lead to increased realism in video games. A possible effect of this is that, for the people playing the games, the worlds in the games become so similar to reality that the line between reality and game becomes blurred. If then the virtual world offered by the game gives greater satisfaction than the real world, the result may be that the player of the game falls into deep video game addiction.

Unfortunately, to our knowledge, there have not yet been any studies on whether better graphics can lead to video game addiction, but let us for the sake of argument assume that it does. If it does lead to video game addiction, then it will lead to a whole host of negative effects. One such effect is decreased academic performance. The results of a study performed by Ip et al. suggests that the frequency of playing video games relates negatively to the academic performance of students [35].

Another negative effect of video game addiction is health issues. Playing video games means sitting still for prolonged periods of time, which may cause obesity and blood clots. Indeed, in a case report Chang et al. argue that prolonged video gaming can be a risk factor that could possibly cause deep vein thrombosis [36], a form of blood clot that can be deadly.

Improvements in the field of global illumination techniques may result in an increase of realism in video games, which may in turn result in video game addiction. In this manner, global illumination may have negative effects for humanity. However, global illumination also has a potential to do great good for humanity. As was mentioned in the beginning of Section 1, global illumination makes scientific particle simulations easier to interpret [5]. Particle simulations can for instance be used to visualize biochemical processes [37]. Without global illumination, interpreting the results of these visualizations will become harder, and in worst case, we may even miss important details, causing us to miss potential scientific breakthroughs.

Our conclusion of the above is that advancing the field will have both negative and positive results. We do not think that the individual researcher who is trying to invent new global illumination techniques is directly responsible for any of those negative effects. This is because the negative effects of video game addiction is caused by global illumination in only a very indirect manner.

References

[1] P. Kán. "High-Quality Real-Time Global Illumination in Augmented Reality". PhD thesis. Institute of Software Technology and Interactive Systems.

[2] T. Ritschel et al. "The State of the Art in Interactive Global Illumination". In: *Comput. Graph. Forum* 31.1 (Feb. 2012), pp. 160–188. URL: http://dx.doi.org/10.1111/j.1467-8659.2012.02093.x.

[3] R. Kuchar and T. Schairer. "State-of-the-art Rendering Techniques in Real-time Architectural Visualization". In: *ACM SIGGRAPH 2007 Posters*. SIGGRAPH '07. New York, NY, USA: ACM, 2007. URL: http://doi.acm.org/10.1145/1280720.1280854.

[4] N. Denut. "Optimal photon behaviour in virtual environment: A Case study using 3ds Max for architectural visualization". B.S. Thesis. Helsinki Metropolia University of Applied Sciences, 2014.

[5] C. P. Gribble and S. G. Parker. "Enhancing Interactive Particle Visualization with Advanced Shading Models". In: *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*. APGV '06. New York, NY, USA: ACM, 2006, pp. 111–118. URL: http://doi.acm.org/10.1145/1140491.1140514.

[6] M. Rump et al. "Photo-realistic Rendering of Metallic Car Paint from Image-Based Measurements". In: *Computer Graphics Forum* 27.2 (Apr. 2008). Ed. by R. Scopigno and E. Gröller, pp. 527–536.

[7] J. T. Kajiya. "The Rendering Equation". In: *Proceedings of the 13th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '86. New York, NY, USA: ACM, 1986, pp. 143–150. URL: http://doi.acm.org/10.1145/15922.15902.

[8] T. Akenine-Möller et al. *Real-Time Rendering, Third Edition*. CRC Press, 2008.

[9] C. Crassin et al. "Interactive Indirect Illumination Using Voxel-based Cone Tracing: An Insight". In: *ACM SIGGRAPH 2011 Talks*. SIGGRAPH '11. New York, NY, USA: ACM, 2011, 20:1–20:1. URL: http://doi.acm.org/10.1145/2037826.2037853.

[10] J. McLaren and T. Yang. "The Tomorrow Children: Lighting and Mining with Voxels". In: *ACM SIGGRAPH 2015 Talks*. SIGGRAPH '15. New York, NY, USA: ACM, 2015, 67:1–67:1. URL: http://doi.acm.org/10.1145/2775280.2792546.

[11] Nvidia. *VXGI*. 2016. URL: https://developer.nvidia.com/vxgi.

[12] A. Panteleev. *Practical Real-Time Voxel-Based Global Illumination for Current GPUs*. 2014. URL: http://on-demand.gputechconf.com/gtc/2014/presentations/S4552-rt-voxel-based-global-illumination-gpus.pdf.

[13] A. Keller. "Instant Radiosity". In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 49–56. URL: http://dx.doi.org/10.1145/258734.258769.

[14] S. Laine et al. "Incremental Instant Radiosity for Real-Time Indirect Illumination". In: *Proceedings of Eurographics Symposium on Rendering 2007*. Eurographics Association, 2007, pp. 277–286.

[15] T. Ritschel et al. "Imperfect Shadow Maps for Efficient Computation of Indirect Illumination". In: *ACM SIGGRAPH Asia 2008 Papers*. SIGGRAPH Asia '08. New York, NY, USA: ACM, 2008, 129:1–129:8. URL: http://doi.acm.org/10.1145/1457515.1409082.

[16] C. Dachsbacher and M. Stamminger. "Reflective Shadow Maps". In: *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games*. I3D '05. New York, NY, USA: ACM, 2005, pp. 203–231. URL: http://doi.acm.org/10.1145/1053427.1053460.

[17] T. Ritschel et al. "Approximating Dynamic Global Illumination in Image Space". In: *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*. I3D '09. New York, NY, USA: ACM, 2009, pp. 75–82. URL: http://doi.acm.org/10.1145/1507149.1507161.

[18] M. Mara et al. "Deep G-buffers for Stable Global Illumination Approximation". In: *Proceedings of High Performance Graphics*. HPG '16. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2016, pp. 87–98. URL: http://dx.doi.org/10.2312/hpg.20161195.

[19] A. Kaplanyan and C. Dachsbacher. "Cascaded Light Propagation Volumes for Real-time Indirect Illumination". In: *Proceedings of the 2010 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*. I3D '10. New York, NY, USA: ACM, 2010, pp. 99–107. URL: http://doi.acm.org/10.1145/1730804.1730821.

[20] S. Thiedemann et al. "Voxel-based Global Illumination". In: *Symposium on Interactive 3D Graphics and Games*. I3D '11. New York, NY, USA: ACM, 2011, pp. 103–110. URL: http://doi.acm.org/10.1145/1944745.1944763.

[21] C. Crassin. "GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes". English and web-optimized version. PhD thesis. UNIVERSITE DE GRENOBLE, July 2011. URL: http://maverick.inria.fr/Publications/2011/Cra11.

[22] M. Sugihara et al. "Layered Reflective Shadow Maps for Voxel-based Indirect Illumination". In: *Proceedings of High Performance Graphics*. HPG '14. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2014, pp. 117–125. URL: http://dl.acm.org/citation.cfm?id=2980009.2980022.

[23] W. Donnelly and A. Lauritzen. "Variance Shadow Maps". In: *Proceedings of the 2006 Symposium on Interactive 3D Graphics and Games*. I3D '06. New York, NY, USA: ACM, 2006, pp. 161–165. URL: http://doi.acm.org/10.1145/1111411.1111440.

[24] M. Levoy. "Efficient Ray Tracing of Volume Data". In: *ACM Trans. Graph.* 9.3 (July 1990), pp. 245–261. URL: http://doi.acm.org/10.1145/78964.78965.

[25] C. Crassin and S. Green. "Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer". In: *OpenGL Insights*. Ed. by P. Cozzi and C. Riccio. OpenGL, OpenGL Es, and Webgl Community Experiences. Taylor & Francis, 2012. Chap. 22.

[26] Nvidia. *NV_conservative_raster*. 2017. URL: https://developer.nvidia.com/sites/default/files/akamai/opengl/specs/GL_NV_conservative_raster.txt (visited on 04/06/2017).

[27] M. Kilgard. *Nvidia OpenGL in 2016*. 2017. URL: http://on-demand.gputechconf.com/siggraph/2016/presentation/sig1609-kilgard-jeffrey-keil-nvidia-opengl-in-2016.pdf (visited on 04/21/2017).

[28] M. Takeshige. *The Basics of GPU Voxelization*. Accessed: 2016-03-05. 2015. URL: https://developer.nvidia.com/content/basics-gpu-voxelization.

[29] D. J. T. Gomes. "Voxel Based Real-Time Global Illumination Techniques". MA thesis. Universidade do Minho, 2015.

[30] L. Williams. "Casting Curved Shadows on Curved Surfaces". In: *Proceedings of the 5th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '78. New York, NY, USA: ACM, 1978, pp. 270–274. URL: http://doi.acm.org/10.1145/800248.807402.

[31] S. Yeung. *Implementing Voxel Cone Tracing*. 2017. URL: http://simonstechblog.blogspot.se/2013/01/implementing-voxel-cone-tracing.html (visited on 04/18/2017).

[32] C. C. Tanner et al. "The Clipmap: A Virtual Mipmap". In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH '98. New York, NY, USA: ACM, 1998, pp. 151–158. URL: http://doi.acm.org/10.1145/280814.280855.

[33] W. Kröker. *Real-Time Indirect Illumination with Voxel Cone Tracing*. 2017. URL: https://github.com/compix/VoxelConeTracingGI (visited on 04/27/2017).

[34] M. McGuire. *Computer Graphics Archive*. http://graphics.cs.williams.edu/data. Aug. 2011. URL: http://graphics.cs.williams.edu/data.

[35] A. W. Barry Ip Gabriel Jacobs. "Gaming Frequency and Academic Performance". In: *Australasian Journal of Educational Technology* 24.4 (Aug. 2008), pp. 355–373.

[36] H.-C. L. Chang et al. "Extensive deep vein thrombosis following prolonged gaming ('gamer's thrombosis'): a case report". In: *Journal of Medical Case Reports* 7.1 (2013), p. 235. URL: http://dx.doi.org/10.1186/1752-1947-7-235.

[37] M. L. Muzic et al. "Illustrative Visualization of Molecular Reactions using Omniscient Intelligence and Passive Agents". In: *Computer Graphics Forum* 33.3 (June 2014). Article first published online: 12 JUL 2014, pp. 141–150. URL: https://www.cg.tuwien.ac.at/research/publications/2014/lemuzic-2014-ivm/.