DYNAMIC UPDATE OF SPARSE VOXEL OCTREE BASED ON MORTON CODE

by

Yucong Pan

A Thesis

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Master of Science

Department of Computer Graphics Technology

West Lafayette Campus, Indiana

May 2021

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Hazar Nicholas Dib
Department of Computer Graphics Technology

Dr. Tim McGraw
Department of Computer Graphics Technology

Dr. Christos Mousas
Department of Computer Graphics Technology

Approved by:
Dr. Nicoletta Adamo-Villani

This thesis is dedicated to my parents, who have been supporting me to pursue my dreams. I am always proud to be your son.

ACKNOWLEDGMENTS

This study is supported by Polytechnic Institute, Purdue University. 3D Models are provided by the Stanford repository. I want to thank specially Professor Tim McGraw and Professor Bedrich Benes, who have motivated me to study in computer graphics. Thanks for the help of my best friends Yuanpei, Yiyun, Zhiquan and Jichun.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

BRDF	Bi-directional Reflectance Distribution Function
DAG	Directed Acyclic Graph
FPS	Frame per second
GI	Global Illumination
RSM	Reflective Shadow Map
SVO	Sparse Voxel Octree
VPL	Virtual Point Light

ABSTRACT

Real-time global illumination has been a very important topic and is widely used in game industry. Previous offline rendering requires a large amount of time to converge and reduce the noise generated in Monte Carlo method. Thus, it cannot be easily adapted in real-time rendering. Using voxels in the field of global illumination has become a popular approach. While a naïve voxel grid occupies huge memory in video card, a data structure called *sparse voxel octree* is often implemented in order to reduce memory cost of voxels and achieve efficient ray casting performance in an interactive frame rate.

However, rendering of voxels can cause block effects due to the nature of voxel. One solution is to increase the resolution of voxel so that one voxel is smaller than a pixel on screen. But this is usually not feasible because higher resolution results in higher memory consumption. Thus, most of the global illumination methods of SVO (sparse voxel octree) only use it in visibility test and radiance storage, rather than render it directly. Previous research has tried to incorporate SVO in ray tracing, radiosity methods and voxel cone tracing, and all achieved real-time frame rates in complex scenes. However, most of them only focus on static scenes and does not consider dynamic updates of SVO and the influence of it on performance.

In this thesis, we will discuss the tradeoff of multiple classic real-time global illumination methods and their implementations using SVO. We will also propose an efficient approach to dynamic update SVO in animated scenes. The deliverables will be implemented in CUDA 11.0 and OpenGL.

CHAPTER 1 PROBLEM AND PURPOSE

Introduction

Global illumination is a very important topic in computer graphics, it can greatly improve the realism of the scene and generate photorealistic images. Currently offline physically based rendering is widely used in movie and animation industry. And real-time global illumination is popular in video game industry. The most popular solution of GI was proposed by Kajiya (1986), also known as the rendering equation. It uses an equation with a very simple form to demonstrate how to simulate the light transport in real world.

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

Kajiya points out that the light of an object has two contributions, which are its emission and reflection. And reflection comes from every other object in the scene. Therefore, it is a recursive equation, and it is mathematically impossible to solve. Because there are two infinities in it. First, light can bounce between different objects for an infinite amount of time. Second the reflection of an object is the result of infinite light transport. Although Kajiya used Monte Carlo path tracing to approximate the result of this equation and got plausible global illumination, the amount of time used in path tracing is not acceptable in real-time applications. This is because path tracing need lots of samples to converge and ray-triangle intersections are slow in complex scenes.

Therefore, another class of geometry representation called voxels is used to discretize the original scene information. Voxel is a unit of graphical information in 3-dimensional space, which is similar to pixel in 2D space. Voxels has several good properties, for example, voxels' complexity is independent from triangle complexity (voxels usually remain the same regardless of the tessellation level of triangle meshes). Ray-voxel intersection is much faster than ray-triangle intersection. And finally, voxel can be easily fit into octree, while BVH-tree or kd-tree (Foley & Sugerman, 2005) using triangles are much more complicated to construct. But voxels also have some drawbacks such as large memory cost and blockiness in rendering.

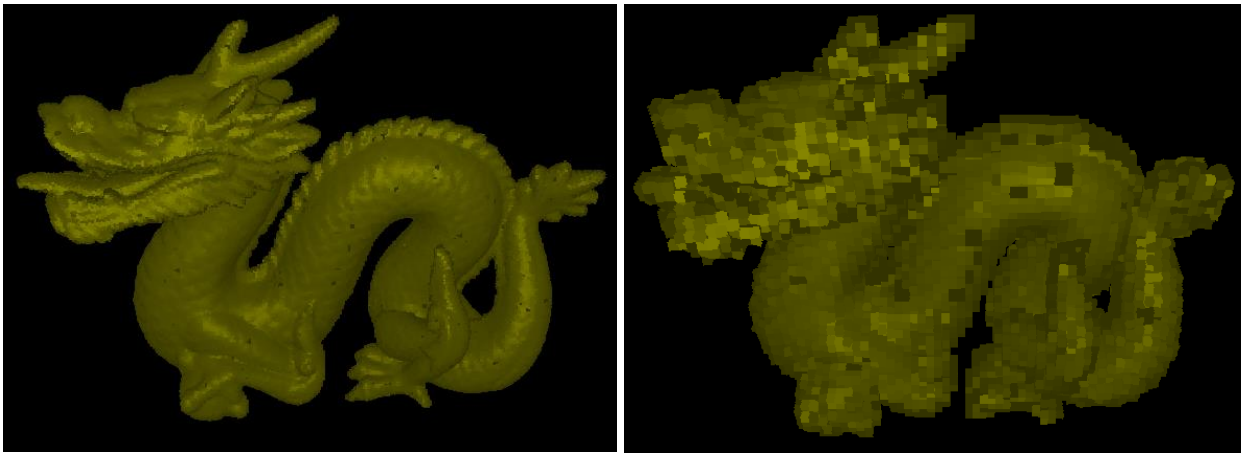

Figure 1. Voxels in different resolutions of Stanford Dragon (Left: 256^3, Right: 64^3)

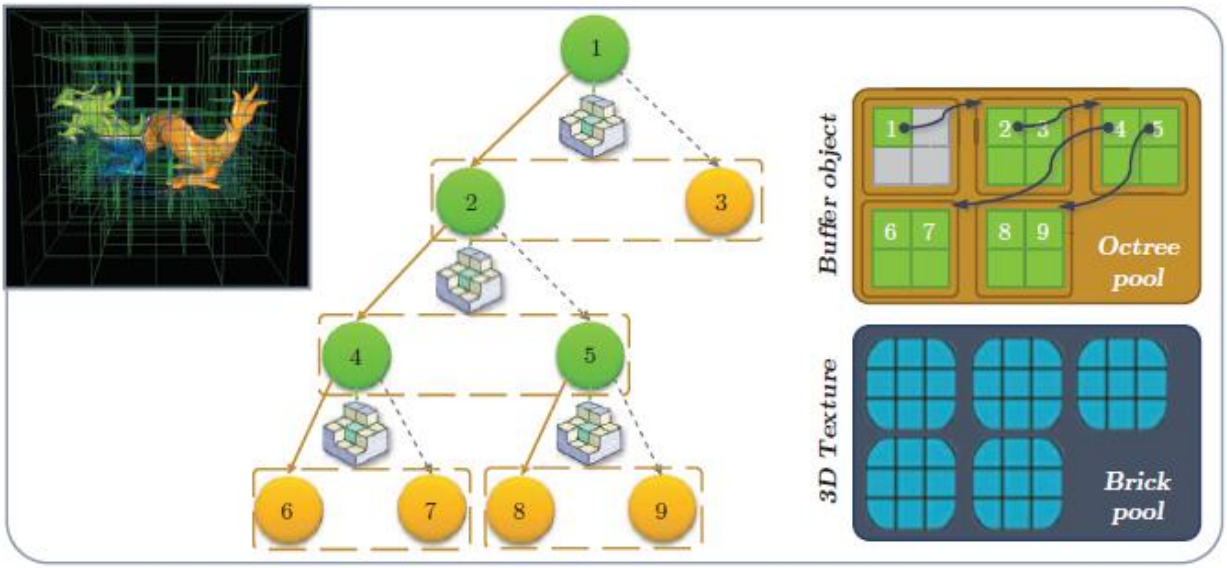

Figure 2. Sparse Voxel Octree structure. Bricks are 2^3 voxel tiles used to store voxels of leaf nodes in texture memory. (Image Courtesy of Crassin & Green, 2012)

The Problem

The problem addressed by this study is that it is very inefficient for traditional global illumination methods to achieve real-time frame rates. In traditional rasterization pipeline, developers always use some tricks to simulate the effect of indirect illumination like ambient or soft shadows. However, although these approaches are plausible to some extent, they lack many important effects in indirect illumination such as color bleeding and caustics. Goral et al., (1984)

came out with the radiosity method to approximate indirect illumination. It divides the scene into lots of tiny patches and treats each patch as a potential light source. Then it calculates the lighting effect between different patches and runs recursively to get global illumination. However, radiosity method is a view independent method, it is a huge waste to calculate for objects that cannot be seen. Also, since this method needs to run recursively, it is difficult to parallel it on modern graphics cards. Kajiya (1986) proposed a method called path tracing. It uses rays to represent light path and sends out one ray per pixel from camera to do backward tracing and gather light accumulation. It uses Monte Carlo method to solve the integration in the rendering equation. Monte Carlo method is great at handling diffuse or glossy reflection, but it is extremely slow to converge. It requires a lot of samples to denoise in final image.

Besides, the methods mentioned above are both very time-consuming, it would be impossible to use them in real-time rendering. The original radiosity and path tracing method usually takes several minutes or even hours to converge an image. There are many optimizations that aims to accelerate the convergence process, for example, bi-directional path tracing (Lafortune & Willems, 1993) combines backward tracing and forward tracing to greatly speed up the convergence in indoor scenes. Also, many data structures can be used to organize the data in the scene, such as kd-tree (Foley & Sugerman, 2005) and BVH-tree. But with these improvements, it is still far from achieving real-time framerate (at least 30 frames per second). Offline rendering approach usually focus on physical accuracy rather than computation time, but real-time rendering needs to guarantee framerate first. Hence the requirement of accuracy could be waived in most situations, sometimes one or two bounces of indirect light is enough. Users often cannot distinguish the details of indirect light in a dynamic scene.

In order to reduce the complexity of light transport simulation, screen-space methods (Shanmugam & Arikan, 2007) and near-filed illumination are commonly used in many real-time applications. These techniques restrict the indirect illumination to geometry that is visible from camera hence greatly increase the efficiency of indirect illumination. However, these methods may lose indirect illumination when light sources that are blocked from camera and generate artifacts when dynamic objects moves in/out the view.

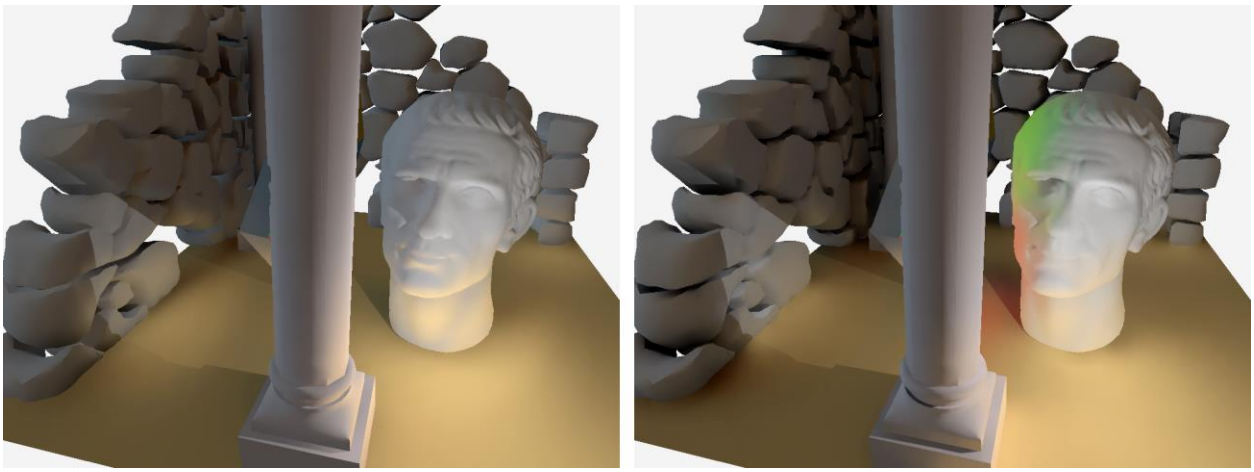

Figure 3. Left: screen space methods. Note that two light sources are hidden behind the column, and screen space method failed to display their reflections. (Image courtesy of Thiedemann et al., 2011)

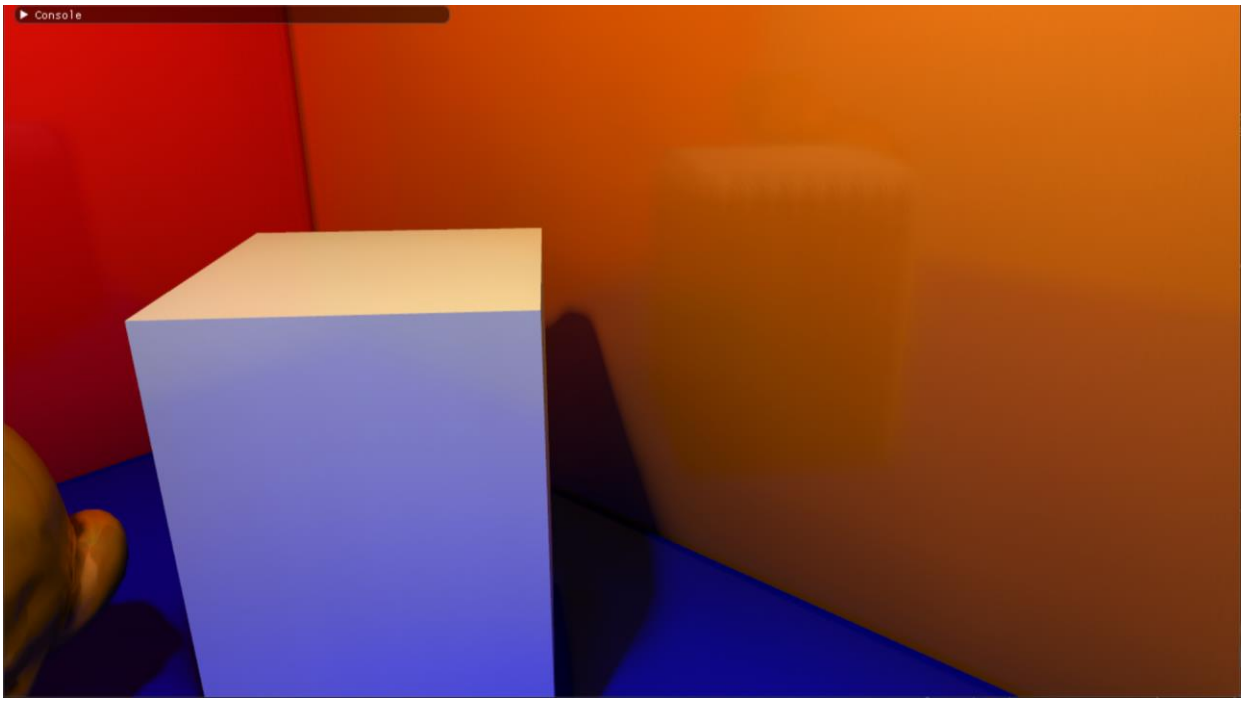

Figure 4. The specular reflection in voxel cone tracing. notice the reflection on the wall shows blockiness due to the nature of voxel.

Significance

As hardware, especially personal GPU, becomes more and more powerful, game developers wish to pursue better graphics effect in games. But traditional rendering pipeline seems to encounter a bottleneck on simulating global illumination. It would be too complex and inconvenient to simulate global illumination using traditional rasterization pipeline. Many developers are trying to migrate the ray tracing method from offline rendering to real-time rendering. Interactive realistic rendering can also benefit other field like medical training or scientific research.

As mentioned before, it is difficult to simulate indirect illumination using traditional rasterization pipeline. Recently, both Nvidia and AMD have published their new generation graphics card, specially designed for real-time ray tracing. Also, Xbox and PlayStation have announced that their next generation consoles will support ray tracing. Many game companies have incorporated indirect illumination in their games. For instance, Control (Burnes, 2019) is a game that supports ray tracing which was released in 2019. It has amazing graphics effects like color bleeding and dynamic reflection on broken glasses. However, it is still impossible to runs it with 4k resolution 60 FPS on almost the most powerful modern graphics card like RTX 3080. The strategy they used was de-noise filtering the results generated by ray tracing, but sometimes players can still see the noise of indirect light clearly in the game. On the other hand, as memory of video cards grows larger, it is possible to fit the entire scene into it as voxels. Kampe et al. successfully store voxels of extremely high resolution ($128K^3$). Fully voxel-based rendering is more feasible with the help of these techniques. The remaining problems mainly focus on dynamic updating of voxels and octrees.

The Purpose

Traditional rendering has some methods to fake single bounce indirect illumination. For example, shadow maps are generated by rendering the scene from the view of light source and use the depth buffer to determine whether each pixel is in shadow or not. Cube maps are often used to simulate reflections or refractions. Also, indirect light maps can be pre-baked in order to add indirect illumination of static objects and light sources. However, these techniques are usually limited to single-bounce indirect light and static scenes, and often requires pre-constructing.

The purpose of this study is to explore the possibility of dynamic sparse voxel octree in global illumination. With the help of efficient octree construction methods, plenty of research (Sun & Agu, 2015) have successfully achieved real-time performance with a static scene. Also, efficient approaches of interactively modifying voxels have been proposed by Careil (2020). Inspired by these researches, we want to develop a feasible algorithm to interactively modify SVO.

Research Questions

The research questions proposed by this study is that:

1. What is an efficient approach to dynamically update sparse voxel octree?
2. How seriously will the updating influence the performance of rendering?
3. How to quickly undo changes to sparse voxel octree.

Deliverables

The deliverables of this study will be a real-time algorithm that can edit sparse voxel octree efficiently. And it will be implemented in OpenGL and CUDA. The method will be based on Morton code to manage voxels.

Definitions

Ray tracing is a method used in computer graphics to simulate light transport in real world. Light is considered as an infinite number of rays. Therefore, light transport can be simulated by using ray-triangle intersection and bi-directional reflectance distribution function (BRDF). Ray tracing includes forward tracing which means rays are sent from light sources, and backward tracing which means rays came from camera. Ray tracing usually use Monte Carlo method to sample diffuse light, hence generate significant noise. Ray tracing methods are usually very slow and need acceleration algorithms.

Voxel cone tracing (Crassin et al., 2011) is a real time ray tracing algorithm. The difference is that, instead of using lots of rays to uniformly sample hemisphere for each hit, it uses multiple cones to represent the area of the hemisphere. Each cone sample on certain level sparse voxel octree depending on the radius of cone. Therefore, it is a noise-free algorithm.

Radiosity method (Goral et al., 1984) is a recursion approach to simulate global illumination. It divides the scene geometry into numerous patches and considers each patch as a potential light source. It computes the direct light in each patch recursively to get indirect light. This method also does not suffer from noise hence is preferred by a lot of research in the field of real-time rendering.

Limitations

The GPU memory in this study is limited to 8GB, so we will have limited size of voxels.

CHAPTER 2 REVIEW OF THE LITERATURE

There are many algorithms trying to approximate global illumination. Kajiya proposed Monte Carlo Path Tracing and try to solve the rendering equation using backward tracing. Also, Photon mapping (Jensen, 1996) and radiosity (Goral et al., 1984) are both powerful methods in global illumination. However, they are designed for offline rendering and cannot be easily adapted in real-time application. The purpose of this section is to discuss offline global illumination algorithms and their voxel-based implementations.

Traditional Methods

The reason why traditional Monte Carlo ray tracing cannot be applied in an interactive framerate is that, Monte Carlo method cannot generate enough samples quickly enough, so that the noise effect in rendering will be quite obvious. The most straight forward solution is to use de-noise filtering to eliminate noise (Bauszat et al., 2011). Various types of filters could be used in this situation, for example, bilateral filter is an edge-preserving filter that can both reduce the noise in image and not cause much loss of details.

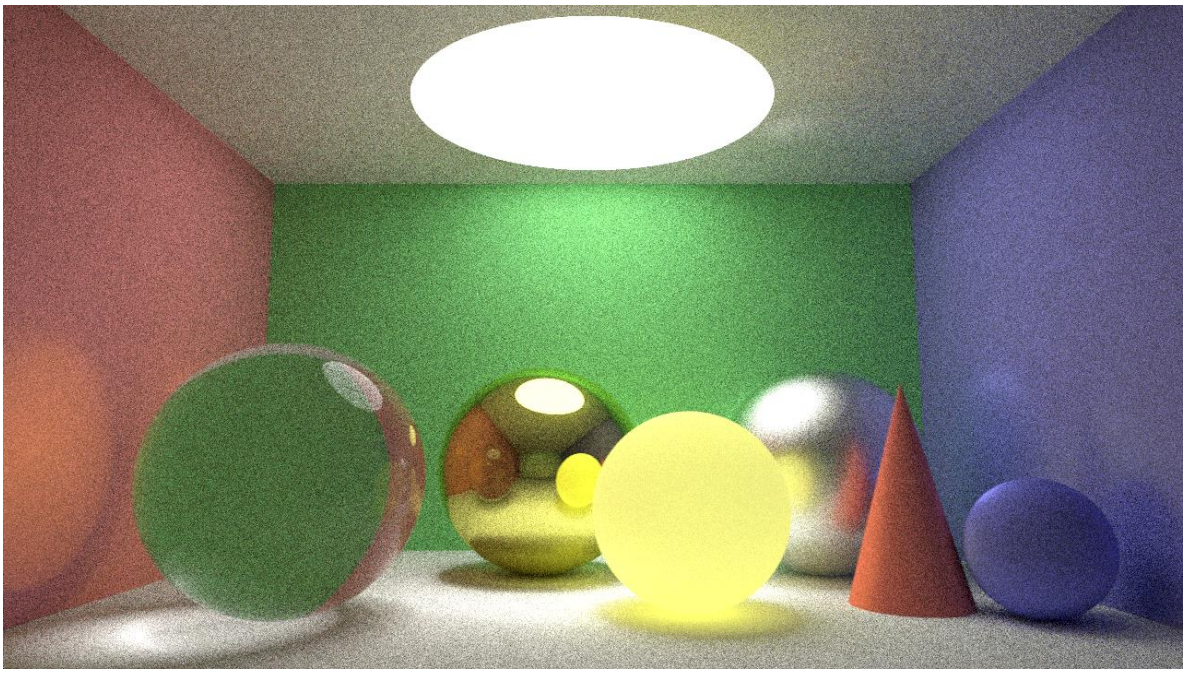

Figure 5. Noise in path tracing due to insufficient sampling (64 samples).

Photon mapping (Jensen, 1996) uses the strategy of shooting and storing photons in a so-called photon map and fetch indirect light by interpolating nearby photons for each pixel. This approach needs giant memory to store photons and has problems dealing with discontinuity in scene. Although Stochastic Progressive Photon Mapping (Hachisuka & Jensen, 2009) greatly increase the efficiency computing distributed ray tracing effects, and also resolve the problem of memory-intensity in original photon mapping, it still cannot achieve real-time framerate and it often suffers from flickering, splotches and artifacts.

Traditional offline algorithms value physical accuracy, but the first few rays captures most of the indirect light information. Therefore, it would be a good idea to sacrifice some extent of accuracy in order to save time. Most real-time algorithms only calculate single bounce or two bounce of indirect illumination. And combines indirect illumination with direct illumination. Instant Radiosity (Keller, 1997) is an extension to radiosity (Goral et al., 1984) method, and it comes up with the concept of Virtual Point Light (VPL), which is used to represent the light information of reflection. In pre-processing, many rays are shot from light sources randomly, once a ray hit a surface, we create a point light there (VPL). Therefore, VPLs can be used to approximate indirect illumination. But if the light sources move frequently, this method could run costly by generating numerous new light sources each frame. Reflective Shadow Maps (Dachsbacher & Stamminger, 2005) has greatly improve the performance of instant radiosity (Keller, 1997). RSM considers each pixel as light source and stores not only depth but also reflection radiance flux and geometric data in shadow map. However, this method does not consider the visibility issues of indirect light, because occlusion information of pixels cannot be queried using shadow map. Updating RSMs in many-light scenes will be time-consuming. But it is an overall very efficient method in real-time rendering of indirect light.

Since diffuse indirect light does not vary frequently, computing diffuse light for each pixel would be a huge waste. Multiresolution methods choose to compute diffuse indirect light in low resolution and upsample the result to full-screen resolution. Hierarchical Image-space Radiosity (Nichols, 2009) uses multiple mipmaps of depth field, direct light and normal based on image-space, hence diffuse indirect light could be computed in low resolution and interpolated to full resolution. This method can provide real-time performance but is restricted to diffuse materials and one-bounce indirect illumination. But the idea of calculating diffuse reflection in lower level of detail is the same among different approaches.

Voxel-based Methods

Some researchers have combined voxels with classic GI methods to achieve interactive frame rates. Voxel-based global illumination (Thiedemann et al., 2011) is a union of voxel-based visibility and near-field illumination. They implemented a novel binary voxelization method which uses a 2D texture atlas to represent 3D voxels. Each texel represent a column in 3D voxel grid and each bit represents whether geometry exists at a specific depth. Thus ray-voxel intersection test can be done very efficiently. However, this kind of representation may be enough for visibility information but is hard to be adapted by voxels that stores colors or normals. Besides, the depth resolution can be limited by texture formats which typically allows 128 bits per texel at most.

Sun implemented a real-time global illumination method based on VPL and use sparse voxel octree to conduct visibility test. The result shows SVO greatly increase the speed of visibility test, which proves SVO works well in accelerating ray tracing algorithms. But their approach does not have any dynamic objects and the process of sampling from VPLs does not take advantage of SVO as well.

Kaplanyan (2010) used a nested regular voxel grid to produce diffuse indirect illumination. It is low cost and does not have any flickers, but it lacks precisions and is only limited to low-frequency illumination (diffuse).

Voxel cone tracing (Crassin et al., 2011) is another powerful algorithm for real-time global illumination. The core of this method is to build a hierarchical voxel representation of the scene geometry (SVO). First the information of triangles in the scene is voxelized into 3D texture. For static objects in the scene, this process only needs to run once. However, for dynamic objects, their information needs to be frequently updated. Then in the second step, octree is constructed in a bottom-up manner. Finally, direct light information is injected to the leaf nodes of SVO and value is interpolated from child node to its parent for every level of the tree (this is similar to mipmap). After this, several cones are used to trace light information stored in SVO. Typically, 5 or 9 diffuse cones are used to cover the area of hemisphere, and one tiny cone are used to trace glossy or specular reflections. This technique directly make use of SVO to simulate light transport and shows interactive indirect illumination. But one of the drawbacks is that specular reflections looks blocky, and they suffer from light lurking due to the discrete representation of geometry and irradiance. Also, they did not provide an efficient way to handle dynamic objects in the scene.

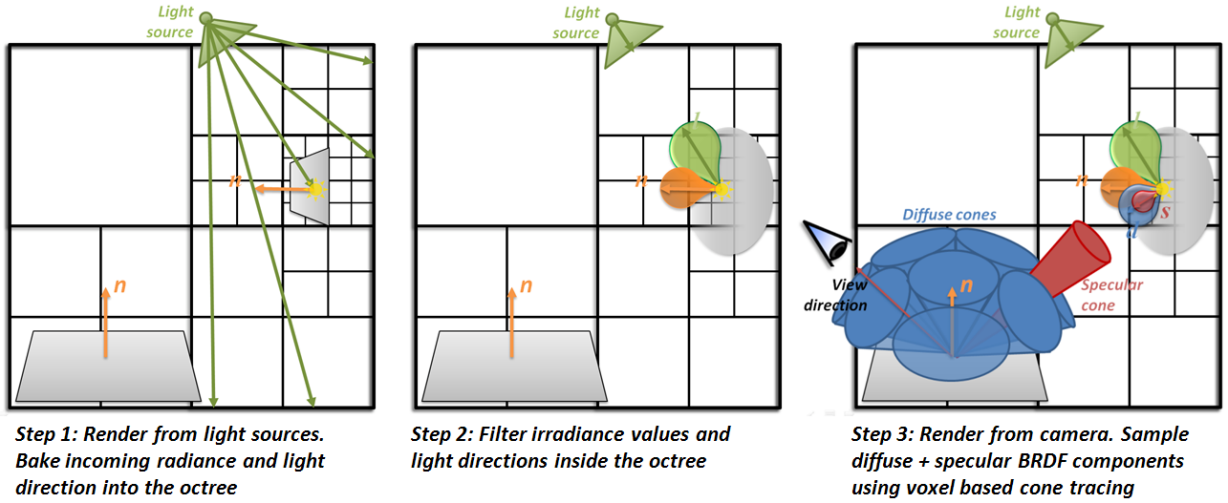

Figure 6. Three steps of voxel cone tracing. (Image courtesy of [Crassin et al., 2011])

Voxelization

Voxels have been popularly used in the context of scientific research, especially with volume data. In computer graphics, voxels are generally stored in 3D textures or 2D texture arrays. Texture is data (usually images) that is stored in cached and read-only memory on device. Reading from texture can be greatly accelerated if reads have certain patterns, i.e., reading in a 2-dimesional manner. However, texture has some drawbacks. It is write-prohibited, and it has strict restrictions on data alignment and format. Although 1D texture does exist, it is not necessary to store 1D arrays in it because caches have already taken care of 1D data.

The process of transforming primitives (typically triangles) of 3D geometry to voxels is called *voxelization*. The similar process in 2D is well known as rasterization. There are two types of rasterization, which are thin rasterization and conservative rasterization. The former one only rasterizes pixels that are close to the line segment while the latter one rasterizes all pixels that overlap the line segment. They are also called 4-separating and 8-separating rasterization.

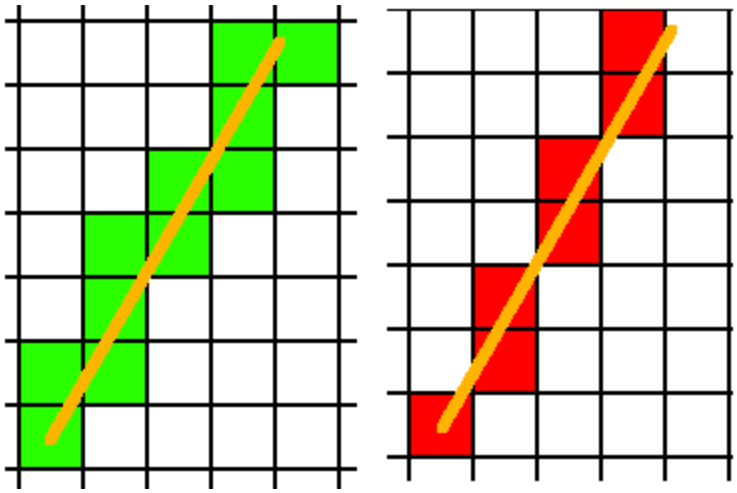

Figure 7 Conservative rasterization (left) and thin rasterization. (Image courtesy of Nathan R.)

Just like rasterization, voxelization can be divided into thin voxelization and conservative voxelization. In this study, we choose conservative voxelization as it is more intuitive and easier to implement. However, our method could be extended to thin voxelization without difficulty.

Voxel-Triangle Intersection

Conservative voxelization means we should voxelize any voxel that overlaps with triangles. Because voxels can be represented by axis-aligned bounding box (AABB), to test whether a voxel intersects with a triangle, is essentially to test whether an AABB intersects with a triangle.

Inspired by Schwarz and Seidel (2010), given a triangle $\boldsymbol{T}$ $(\boldsymbol{v_0}, \boldsymbol{v_1}, \boldsymbol{v_2})$ and an axis-aligned bounding box $\boldsymbol{B}$ $(\boldsymbol{p} + \Delta\boldsymbol{p})$, we found that $\boldsymbol{T}$ overlaps with $\boldsymbol{B}$ if and only if:

a) $\boldsymbol{T}$'s plane overlaps $\boldsymbol{B}$ and

b) For each coordinate, $\boldsymbol{T}$'s 2D projection overlaps $\boldsymbol{B}$' projection

To test whether $\boldsymbol{T}$'s plane overlaps $\boldsymbol{B}$, we can test whether each vertex of $\boldsymbol{B}$ and its opposite vertex are on the different sides of $\boldsymbol{T}$'s plane. If we represent $\boldsymbol{T}$'s normal as $\boldsymbol{n}$, and define a critical point $\boldsymbol{c}$ as

$$c = \left(\begin{cases} \Delta p_x, & n_x > 0 \\ 0, & n_x \le 0 \end{cases}, \begin{cases} \Delta p_y, & n_y > 0 \\ 0, & n_y \le 0 \end{cases}, \begin{cases} \Delta p_z, & n_z > 0 \\ 0, & n_z \le 0 \end{cases}\right)^T$$

And check whether $\boldsymbol{p} + \boldsymbol{c}$ and $\boldsymbol{p} + (\Delta\boldsymbol{p} - \boldsymbol{c})$ are on different sides of the plane.

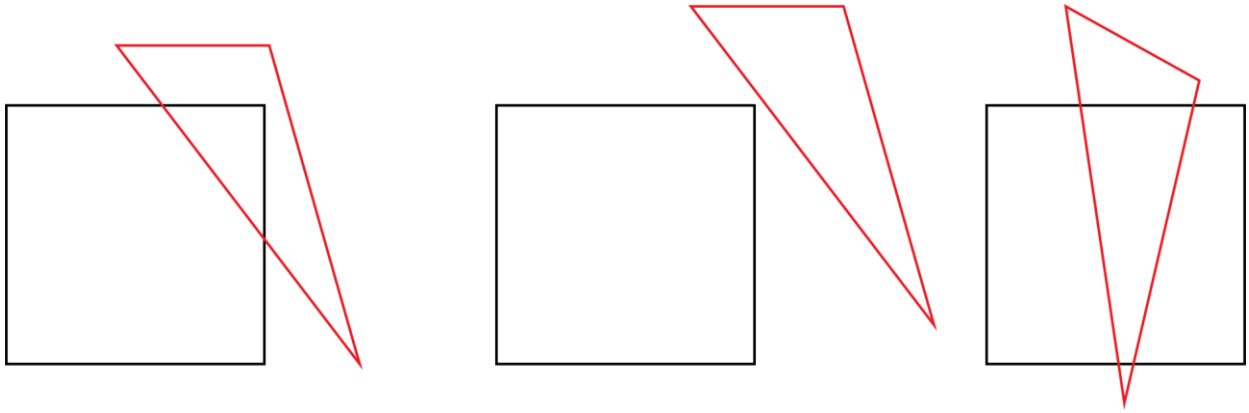

Figure 8 Whether a 2D triangle intersects a square.

To test condition b, we make use of the edge function proposed by Pineda (1988). In a 2-dimensional space, whether a triangle intersects with an axis-aligned square is determined by

a) Test whether the triangle's axis-aligned bounding box (2D) overlaps the square.
b) If yes, test whether any edge of the triangle lies inside the square.

The first problem can be easily resolved because both squares are axis-aligned. Then Pineda's equation can be used to check whether multiple 2D points lie on different sides of a line.

Overall, voxelization is much more complicated and hence more time-consuming than rasterization. But we can make a trade-off between memory and time to greatly lessen the computation during this process. We will discuss how to implement this later.

Morton Order

Voxel is a graphics unit used to represent 3D volumetric data, thus it is obvious that we can store voxel in a 3D data structure. However, practically there are no support for 3D array in most of the programming languages. The most common approach is to use a group of 2D array or just using a long 1-dimensional array instead. Given that the width, height and depth of the whole voxel grid are $\boldsymbol{D}$, any voxel that locates at (x, y, z) can be transformed to a 1D index as

$$x * D * D + y * D + z$$

or

$$z * D * D + y * D + x$$

The former one provides coherency at $\boldsymbol{Z}$ coordinate, while the latter one at $\boldsymbol{X}$ coordinate. However, due to the topological characteristic of *Octree*, we usually read 8 voxels together within one node

of the octree. Although these 8 voxels are adjacent spatially, they are not located coherently on the device memory. Local coherence can significantly reduce the time spent on read/write operations, because every time a read operation is performed, processing unit will fetch the target data as well as other data that locates closely to the target data and stores them in *cache*. So in next read operation, the processing unit will first try to find the target data in cache, and reading from cache is around 100~200 times faster than read from device memory.

Morton order is a linearization of an n-dimensional grid that corresponds to the order in which the leaf nodes of the corresponding 2^n-tree (quadtree, octree) are encountered when performing a post-order depth-first traversal of the tree (Baert et al., 2014). By connecting each point in space in Morton order, the curve has the appearance of the letter Z. Therefore, Morton order is also called Z-order. To encode an n-dimensional coordinate, we can simply interleave the binary code of its coordinate values.

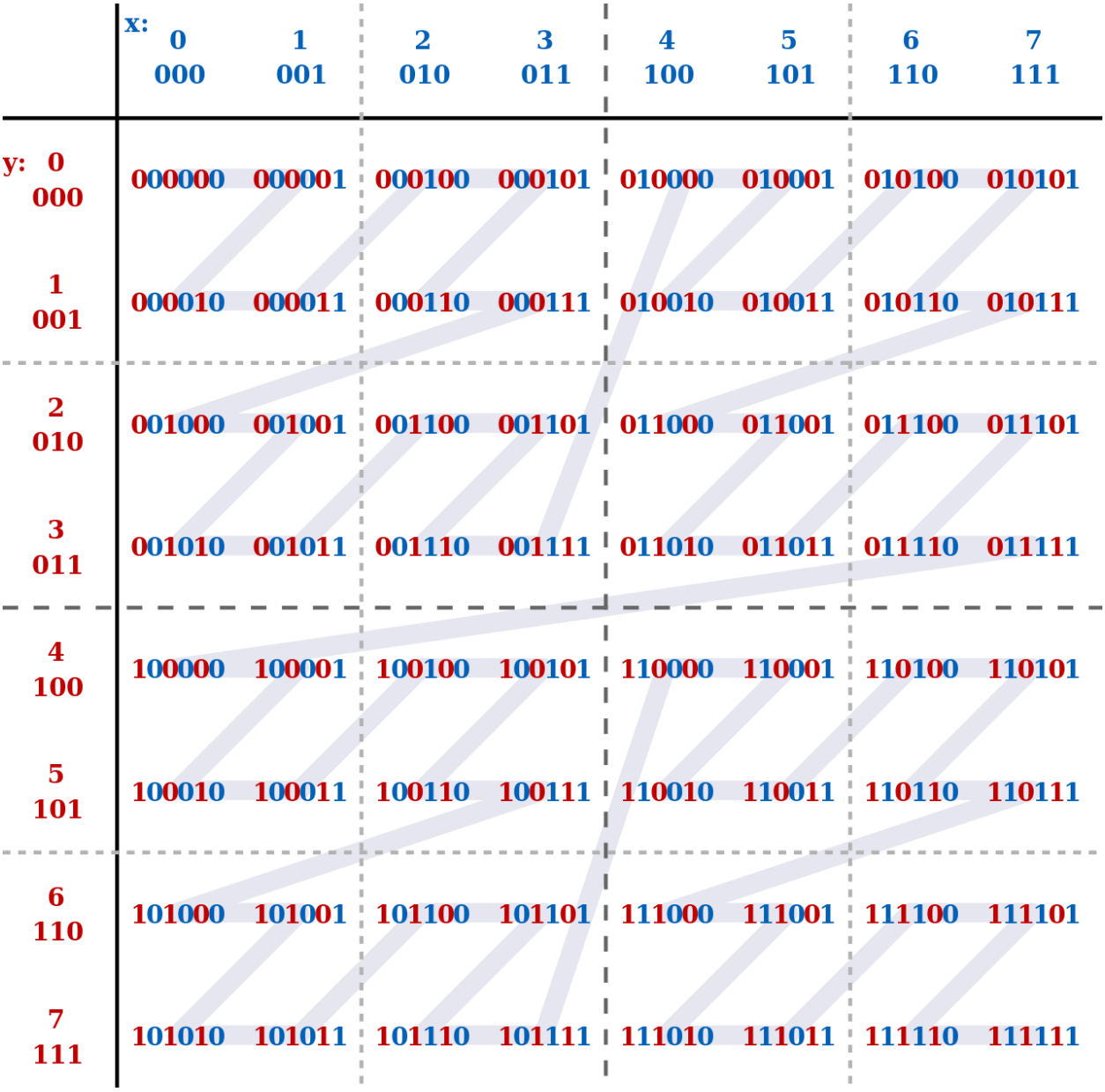

Figure 9 the Morton code and the z-curve (Image Courtesy of David, E.)

Such property of Morton order grants us local coherence while we read/write to voxels on device memory. Besides, Morton order is also very helpful in octree construction, we can easily know the corresponding Morton code of each depth of a 2^n-tree.

$$Code_{d-1} = \frac{Code_d}{2^n}$$

Where d equals to the depth of the 2^n-tree.

Octree Construction

Octree construction is already a complex task, and it is even more challenging on GPU. While each thread has no information about other thread's work, to prevent write collision and race condition becomes the most significant problem. In *Gigavoxels*, Crassin and Green (2012) provides a solution to construct the octree in a top-down manner. First, they allocate a chunk of memory on device which they use to store nodes of octree (this is called *node pool*). Then they open one thread per voxel to do depth-first search on the octree. When one thread finds there should be a child node in the octree where there isn't one, the thread will mark the parent node as *marked*. Second, they open one thread per node on current depth to search if there are any node that is *marked*. If so, they will allocate memory of 8 nodes' size from the *node pool* and let the parent node points to the index of the first nodes in 8 children. This process will be conducted iteratively until it reaches the bottom level (leaf node). Finally, voxel's value will be assigned to leaf nodes, then mipmapped from bottom to top.

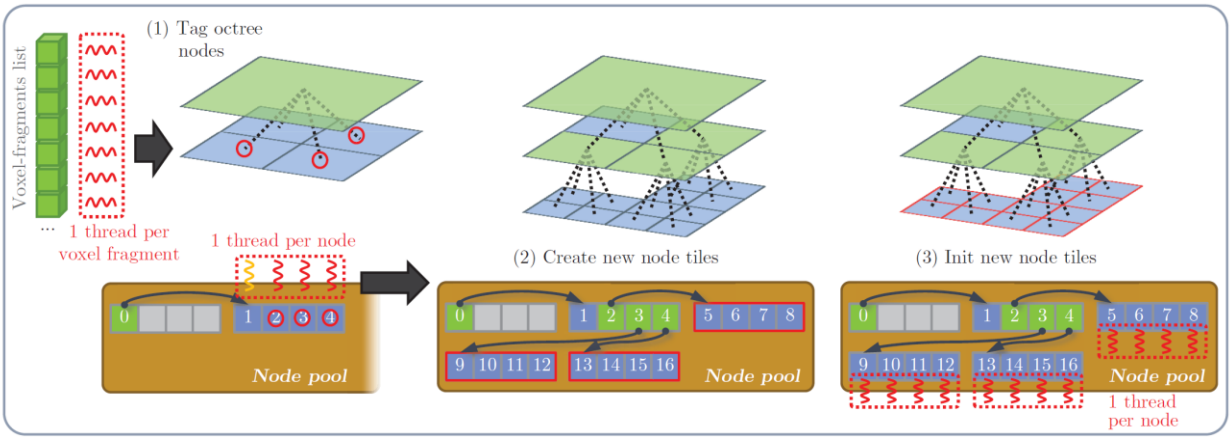

Figure 10 node pool and allocation of new child nodes (quadtree).

In implementation, this approach requires multiple kernel calls per iteration and large amount of write collisions exist in low depth of iteration. Also, the depth-first search needs to restart from the root in each iteration, which causes the same path being traversed repeatedly.

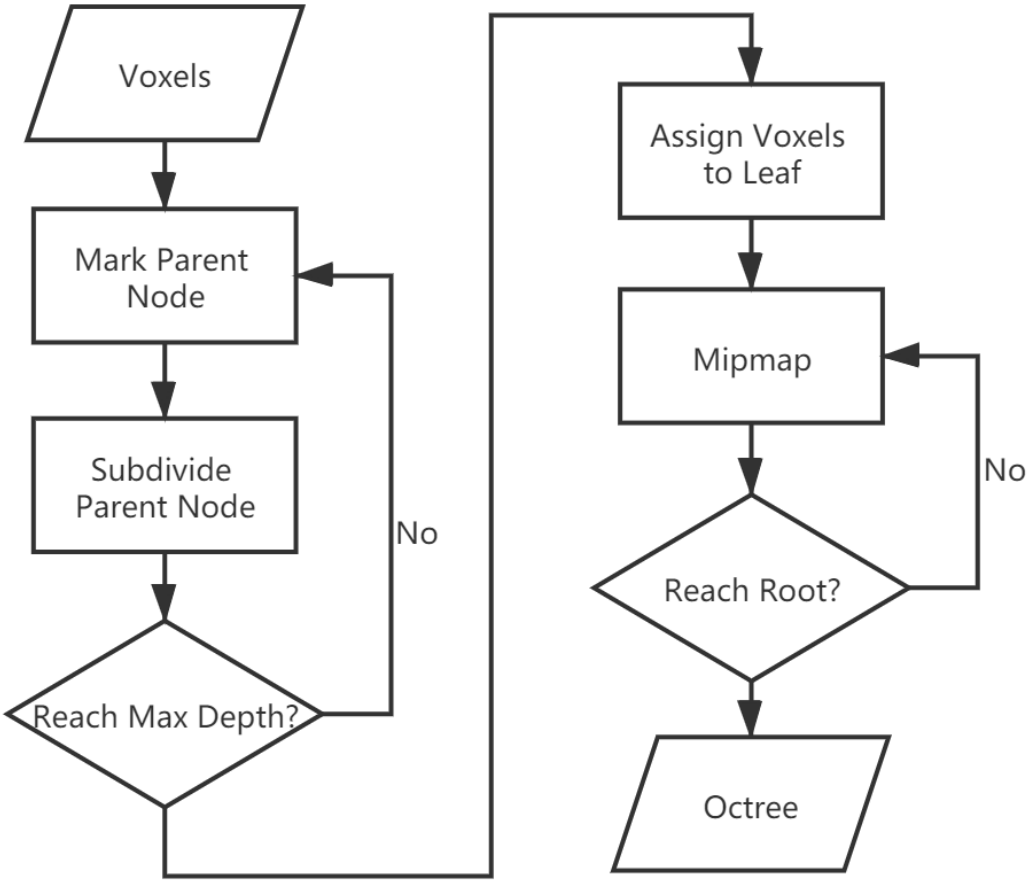

Figure 11 octree construction in top-down manner.

Baert et al. (2014) proposed a novel approach for octree construction which build the octree from bottom to top. With the help of Morton code, it is easy to split the whole voxel grid into multiple small areas and build the octree separately. In the paper they use this property for out-of-core construction of octree. They come up with a method which is extremely memory-saving (total space equals $8 * d * sizeof(node)$, where d is the maximum depth of the octree) and theoretically they could build octree of any number of voxels.

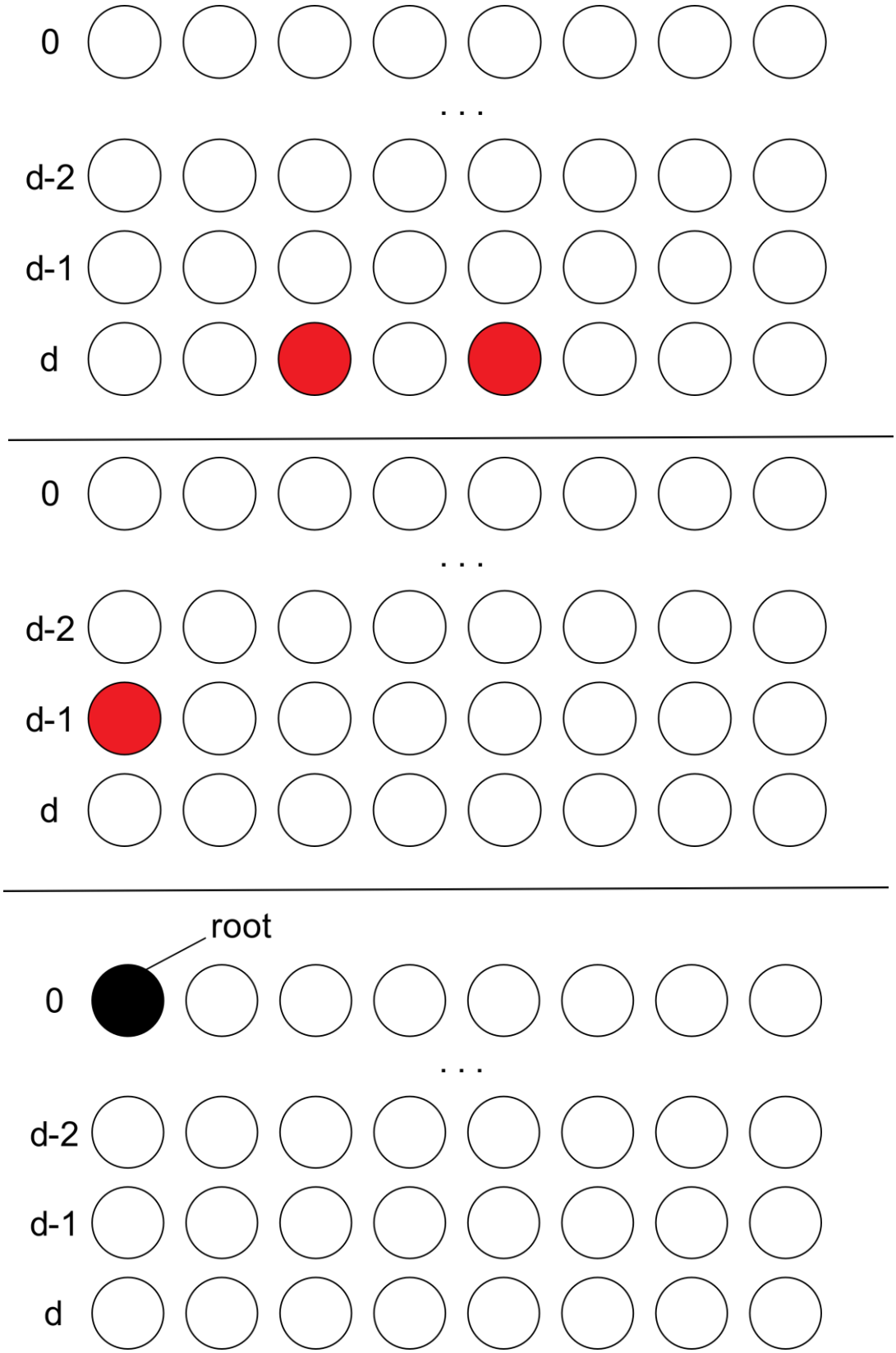

Figure 12 input Morton-sorted voxels, if there are any non-empty voxels in 8 consecutive voxels, they can form a new node in lower level. Repeat this process until the root is constructed.

However, they did not address how to construct octree with multi-threading on GPU based on Morton code. We will mention later how Morton code fit nicely with SIMD (single instruction multiple device).

Dynamic Update

In computer graphics applications, geometry is usually divided into dynamic part and static part. In global illumination, effects like ambient occlusion and indirect illumination that is static are usually baked once and dynamic global illumination need to be rendered every frame. Same in octree construction, the static nodes need only to be voxelized and constructed once, while the nodes which belong to dynamic objects need to be revoxelized and reconstructed in runtime.

The simplest method is to use two octrees, one for static objects and one for dynamic objects. However, this may cause huge performance loss in rendering, because averagely ray-casting two octrees will spend twice more time than ray-casting one octree.

Therefore, the only solution is to use only one octree, but here raises two problems. First, how to edit or add new nodes to the octree while maintain the original data untouched? Second, how to safely delete new nodes in every frame? Crassin et al. (2011) believe dynamic nodes need to be inserted at the end of the octree buffer, but they did not address how to implement it. Careil et al. (2020) implement a novel algorithm to dynamic edit a svDAG (sparse voxel directed acyclic graph), which is similar to our approach. While creating new nodes of DAG, they search whether there are any existing nodes in the old DAG. If yes, they copy the old node and merge it with new node, if not they just simply create a new node instead. After updating, a new root of DAG will be generated, and the old root remains in the original data structure. This permits quick undoing to editing.

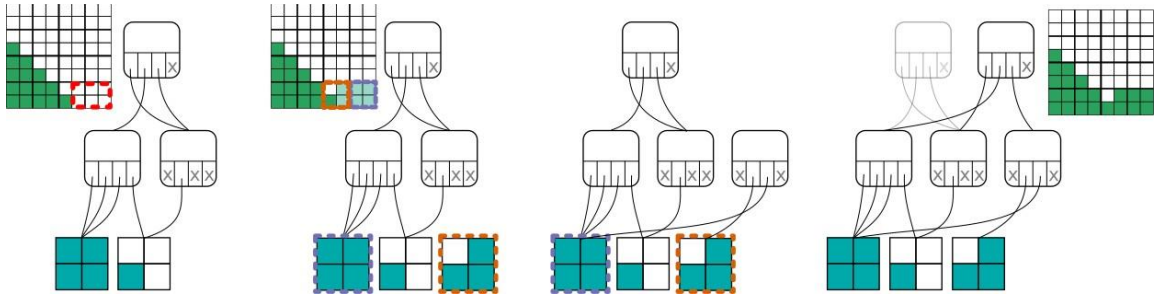

Figure 13 Overview of the updating process. Red region is the area to be updated. (Image Courtesy of Careil et al. (2020))

Besides full reconstruction of dynamic octree, there are also researches about updating octree incrementally (Kim et al., 2018, Lefohn et al., 2006). Recently, Hoetzlein (2016) supports dynamic topological updates of octree on GPU by memory pooling. This is often applied in 3D volumetric painting applications, in which the dynamic and static parts cannot be separated. Kim described a method to dynamically update octree with streaming between CPU and GPU. In their study, the octree structure is stored on CPU and mapped to GPU textures. Instead of updating the entire tree from CPU to GPU in every frame, they designed a method to update only part of the octree that has been modified. Also, they split the process into multiple frames in order to maintain consistent frame rates in rendering.

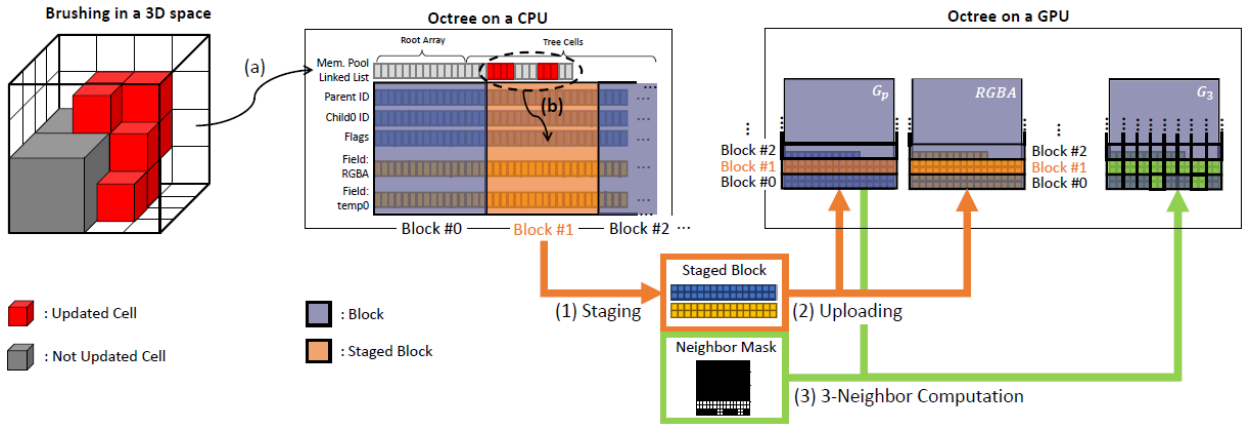

Figure 14 The workflow of dynamically updating octree from CPU to GPU. (Image Courtesy of Kim et al., 2018)

In addition, Campbell et al. (2003) have discussed the performance of different space-filling curves including Morton order in octree-based algorithms. In which they mentioned Hilbert code would also be a good choice in octree loading. We are looking forward to exploring it in the future.

Zeng et al. (2013) proposed a real-time 3D reconstruction method based on octree. They implemented an algorithm called *NodeSplit* to add new nodes to their octree structures. While at the same time, they designed an algorithm which is a reversed procedure of *NodeSplit* to remove nodes of moving objects from the octree. We believe deleting nodes explicitly is less efficient compared to our approach, which deletes all dynamic nodes in constant time. In fact, in their analysis, they have noticed that the node removal operation will slow down the performance.

However, they designed a novel octree structure which divides the entire octree into different layers depending on the depth of each level. This layered structure can accelerate octree traversal by organizing the top few levels of the octree. Octree operations can start from the branch layer instead.

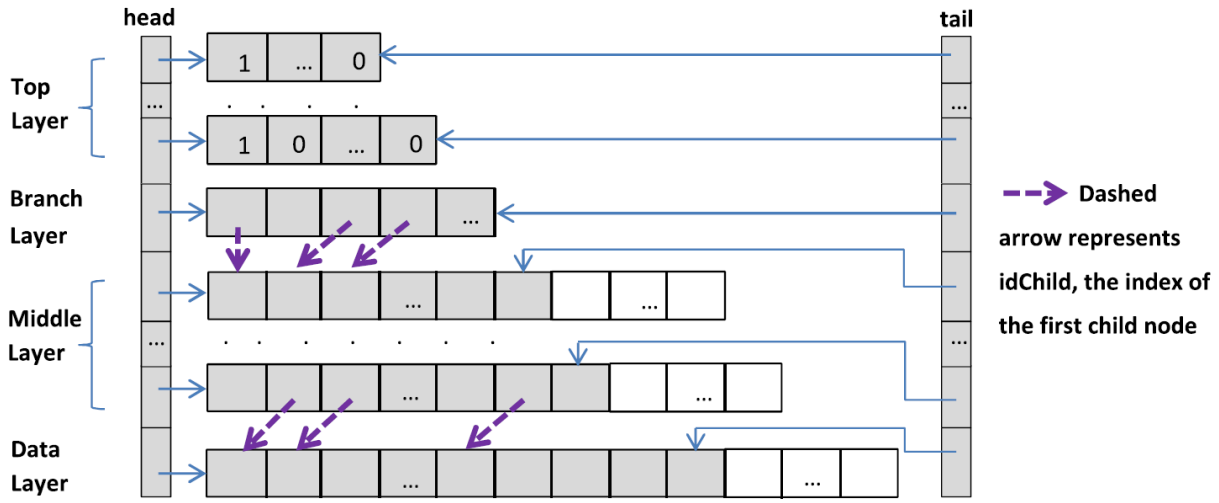

Figure 15 Layered structure of the octree. (Image Courtesy of Zeng et al., 2013)

Summary

Supporting of dynamic light sources and objects requires efficient algorithm of voxelization construction and octree generation. Crassin proposed a simple approach of voxelization. Instead of rendering the scene from 3 different axis directions, they use geometry shader to project each triangle along the dominant axis of its normal, which means the axis that maximizes the fragments of the triangle. Therefore, voxelization can be done by a single draw call. This could be an alternate voxelization method to our voxelization method.

Also, atomic operations are used to avoid race conditions in voxelization, but it will easily ruin the efficiency of GPU parallel computing. We believe that atomic operations are not necessary because in most conditions different threads are writing the same information due to the continuity of mesh.

In Crassin's implementation (Crassin & Green, 2012), octree is built in a top-down manner, and there is a sperate pass mipmapping node value from bottom to up. This process requires more than 20 passes to GPU and is redundant to our demands. There are several other efficient octree

construction and ray casting methods (Laine & Karras, 2010, Baert et al., 2014, Schwarz & Seidel, 2010) that worth trying.

The algorithm of updating voxel nodes proposed by Careil (2020) is very interesting. They generate new nodes in every frame for dynamic objects and derived a new root node which contains the new nodes. While the static node remains the same. In this way, they provide the ability to edit and undo changes to voxels and limit the size of memory that needs to be modified in every frame. Our approach will base on the methods mentioned above.

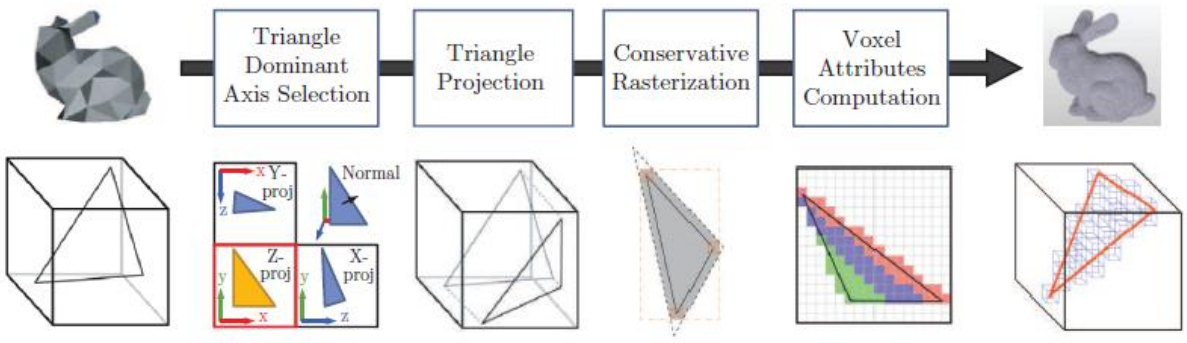

Figure 16. Single pass voxelization. (Image courtesy of [Crassin & Green, 2012])

A lot of research has proved that SVO shows great potential in acceleration to different global illumination algorithms. While majority of research were focusing on efficient voxelization and octree construction, few have explored the dynamic updating of octree. The remaining problem is to find a robust and efficient way of modifying SVO to support dynamic objects and dynamic light sources.

CHAPTER 3 METHODOLOGY

Flow Chart

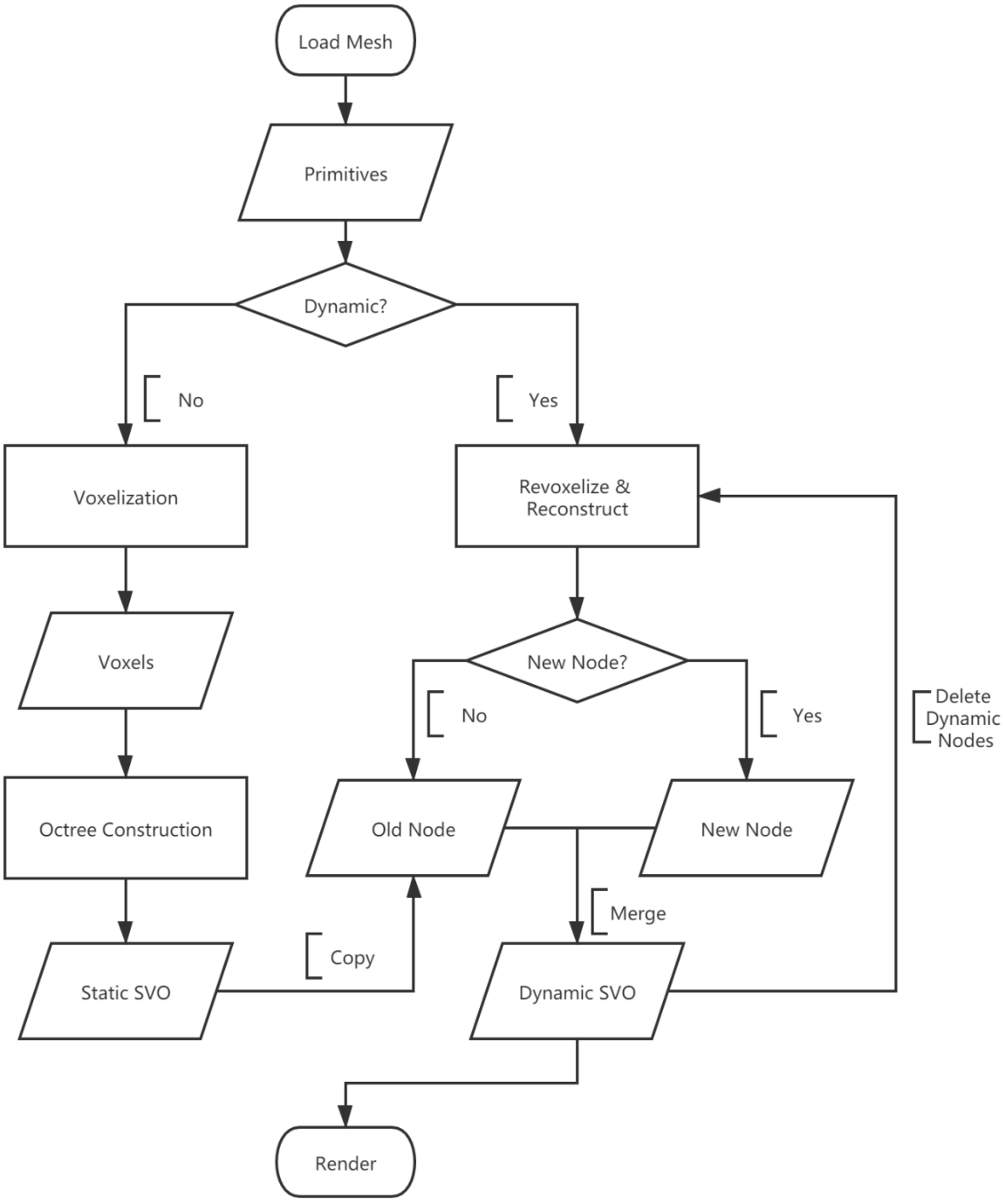

Figure 17. The flow chart of steps used in this study.

Research Methods

The type of research in this study is developmental research. This study will develop and validate a real-time algorithm of Modifying sparse voxel octree.

The variables in this research are the performance and robustness of the algorithm, which will be calculated by *framerates per second* (FPS) or computation time per frame in millisecond. Additionally, we need to test whether the algorithm would change static nodes by mistake or cause error in rendering in various situations, especially when two objects overlap.

Data Collection

The performance of the methods could be collected by counters that is used in the code. For instance, frame per second is a good variable to evaluate the computing speed of the algorithm. Usually, a high FPS results in high performance. As for the robustness of the algorithm, we will test under extreme conditions and prove it in theory.

Data Analysis

Comparing efficiency of different algorithms is simple. Quantitative data like FPS and running time per frame will be recorded, and different scenes will be used to test the robustness of this algorithm.

CHAPTER 4 IMPLEMENTATION

System Overview

In this study, our algorithm is implemented using CUDA and modern OpenGL. CUDA is designed for high performance GPU-accelerated applications. With CUDA, we will be able to use C/C++ style programming language on GPU, which provides more flexibility on coding compared to compute shader in graphics API. OpenGL is only used for the purpose of rendering, which is ray-casting in this case.

The GPU used in this study is RTX 3080, one of the most powerful consumer GPUs, which has a memory capacity of 10 Gigabytes. The CPU used in this study is AMD Ryzen 7 3700x with 16 Gigabytes RAM. However, since voxelization, octree construction and updating are running on GPU, CPU's performance has little impact on our methods.

Voxel

In this study, voxels are used to represent geometry information, in which color and normal are two most important data in rendering. Note that the position of voxels do not need to be explicitly defined, it is because the position of a voxel can be calculated using its coordinate $\boldsymbol{coord}$ and the axis-aligned bounding box ($\boldsymbol{p}, \boldsymbol{p} + \boldsymbol{\Delta p}$)of its root node.

$$pos = \frac{coord}{dim} * \Delta p + p$$

Where $\boldsymbol{pos}$ is the position of the voxel and $\boldsymbol{dim}$ is the size of voxels in current level ($\mathrm{dim} = 2^n$).

To store the normal information, we could use 3 floats where each coordinate (x, y, z) occupies one float. Similarly, we could use 3 floats to store color information, where each channel (r, g, b) use one float. Therefore, totally we will need 6 floats to store one voxel, which equals 24 bytes in memory. However, we should know that time cost of read and write operations to memory increases with data size linearly. Since we are dealing with a 3-dimensional data structure, it is most likely that we need read/write millions of voxels per kernel call (function that runs on device in CUDA). With 24 bytes in one voxel, not only it will greatly slow down the performance of our method, but it will also limit the maximus size/depth of our octree structure. The less size of the voxel is, the more voxels we can store in our device memory.

A common approach is to compress the data into smaller data type. For example, one of the most popular format of color information in computer graphics is RGBA8, which means 8 bytes for each channel of color (red, green, blue and alpha). Hence each channel will have a value from 0 to 255. Then we are able to store color in an *unsigned integer*. Normal can also be stored in one unsigned integer in the same way. However, since normal itself is a signed value, we need to use 1 bit for the sign, which means normal only have a precision of 7 bits. The method below can be easily implemented using bitwise operation, for example,

given that

$$color = (r, g, b, a)$$

The compressed value is

```
(unsigned int(a) & 0x000000FF) << 24U | (unsigned int(b) & 0x000000FF) << 16U
   | (unsigned int(g) & 0x000000FF) << 8U | (unsigned int(r) & 0x000000FF)
```

Now we only need 8 bytes for each voxel, theoretically we have gained a 300% speed up. It should be noted that normal only requires 24 bits in total (8 bits per coordinate), with 8 bits left in an unsigned integer. Normally we do not need alpha information in color if transparent objects are not involved, this gives us extra 8 bits. In fact, we are only using 6 bytes per voxel.

So why not use an unsigned integer (4 bytes) and a short integer (2 bytes) to represent one voxel? Unfortunately, the operation system will automatically align the data in memory by 4 bytes. Unless we reduce the size of one voxel to 4 bytes, it will not make any difference whether we are using 6 bytes per voxel or 8 bytes per voxel. Then is it possible to store color and normal both in a single unsigned integer?

There is another more compact format for color called RGB565, which means 5 bytes for red and blue channel and 6 bytes for green channel (human eyes are more sensitive to green). Then we have 16 bits left for normal. We could use 5 bits for each coordinate, but regarding normal is a signed value, 4 bits' precision is not accurate enough for rendering. However, remind that normal is a normalized value, its length will always be exactly 1.

$$length(n) = n.x^2 + n.y^2 + n.z^2 = 1.0$$

Which means we only need to store 2 coordinates of the normal and use one bit to store the sign of the last coordinate, we should be able to calculate the last coordinate in runtime. In this way, both color and normal can lie in one unsigned integer. As for the extra one bit for last coordinate, it could be stored in the *pointer* value in node which we will discuss in next section.

Octree and Node

Our octree structure consists of nodes of different depths, in which the node of lowest depth is called *root* and the node of highest depth or with no children is called *leaf*. Each node other than the root node has its *parent node* and each node other than leaf nodes has 8 *child nodes*.

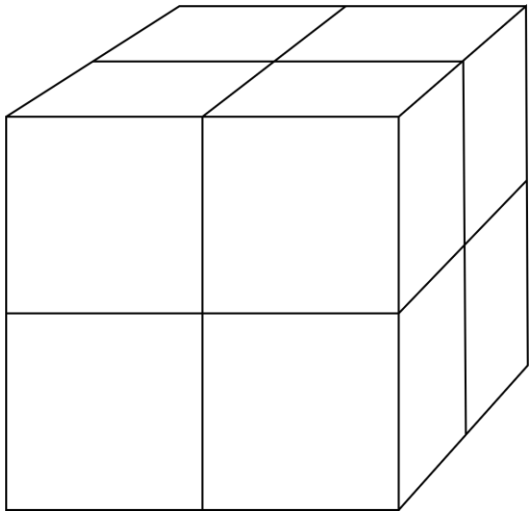

Figure 18 the large cube is the parent node, and 8 small cubes are the children of the parent node.

Since rendering of sparse voxel octree is not the major topic of this study, we choose the simplest form of octree nodes, where each node only stores a pointer to its first children. Additionally, we always put 8 child nodes in consecutive memory. Thus, we can fetch each one of them once we know the index of the first child node. Because we are using Morton code, we will also know each child node's position regarding its parent.

Figure 19 How parent node and child nodes are connected in a 1D array. Yellow square represents parent node.

Other than a pointer to its children, each node also has a voxel. Nodes in the bottom level of octree represent voxels with a one-to-one ratio, while nodes in lower levels have voxels that contain average values of their children's voxels. As mentioned in last section, we could compress voxels into 4 bytes. But that may cause precision loss and glitches in rendering, but we are glad to test it in future experiments. For now, we use a voxel of 8 bytes, and that makes the size of our node structure 12 bytes totally. Note that we use a bitmask of 8 bits totally to indicate whether each children node is an empty node (node that does not contain geometry information). This permits quick child node check in ray-casting.

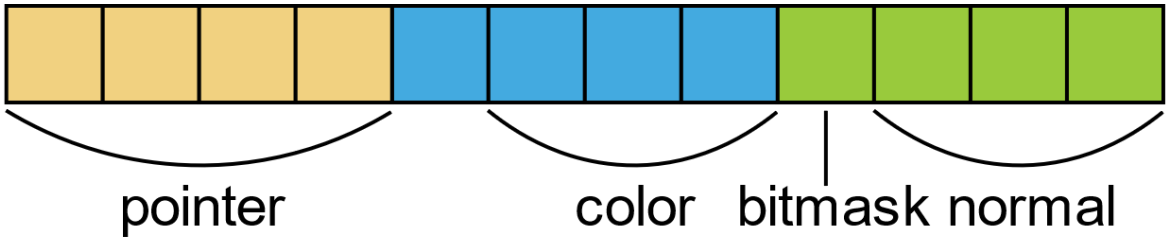

Figure 20 data alignment of our node structure, each square represents one byte in memory. Note that we have one empty byte which could be used for other purpose (i.e., alpha channel).

Voxelization

Our voxelization method is very straight-forward. We launch one thread per triangle and calculate the triangle's axis-aligned bounding box. Then we can find the start voxel and the end voxel based on the minimum corner and maximum corner of the triangle's AABB. Finally, we iterate each voxel in the AABB for voxel-triangle intersection test, if result returns yes, we store the corresponding color and normal information (interpolated using barycentric coordinates) into that voxel. Here is the pseudocode of this process.

```
minAABB, maxAABB = GetAABB(Triangle)
startVoxel = GetVoxel(minAABB)
endVoxel = GetVoxel(maxAABB)
for( i = startVoxel.x; i <= endVoxel.x; i++)
     for( j = startVoxel.y; i <= endVoxel.y; j++)
          for( k = startVoxel.z; k <= endVoxel.z; k++)
```

```
if intersect(Triangle, Voxel(i,j,k))
     arrayIdx = EncodeMorton(i,j,k)
     voxelList[arrayIdx] = Voxel(color, normal)
```

As mentioned in last chapter, the process of voxelization, specifically voxel-triangle intersection test, can be accelerated by precompute and repeatedly used some intermediate value. For example, the normal of each triangle is the same among each intersection test, but by precomputing and storing it with each triangle, we can save time for one cross product and one normalize operation for each test.

Nevertheless, it is possible one voxel may be shared by multiple triangles. This often happens on the edges of triangles. In multi-thread programs, this problem is called write collisions or race conditions, where multiple threads try to read/write to the same area in memory. This could cause serious problem if not handled. For example, if one thread needs to add one to a variable *sum* in device memory, it requires 3 steps. First, fetch the value of *sum* in device memory; Second, add one the *sum* in local memory; Third, copy the new value back to device memory. But if two threads *A* and *B* are trying to perform addition to *sum* at the same time, it may result in unexpected behavior.

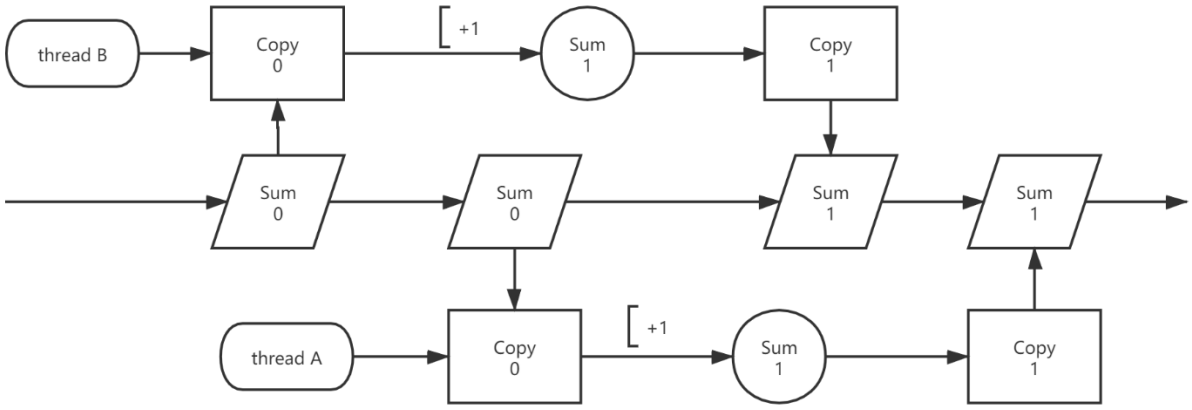

Figure 21 here is an example where thread A and B are trying to add one to the same variable in device memory and return a false result. Circle means local variable in each thread.

Atomic operations are created to prevent race conditions, it guarantees multiple writes to the same variable will be performed in chronological order, but it will have a huge impact on

performance. Considering that normal and color information are usually changing smoothly on the same object, we may assume that even a voxel is written by two adjacent triangles, they are given the same color and normal. Therefore, it is safe to abandon atomic operations in exchange of high performance.

Octree Construction

Our octree construction method is an extension to Baert's (2014) method. In previous chapter, we introduced how to implement out-of-core octree construction using only a small buffer. Now we need to combine it with multi-threading on GPU. Since we are using Morton code, every group of 8 consecutive nodes of Morton code i ($i \; mod \; 8 = 0 \sim 7$) are guaranteed to have the same parent node. Therefore, we can launch one thread per 8 nodes.

The input of the algorithm should be two buffers with the size equals to the total amount of voxels in current depth (8^n). One buffer should contain voxels (color and normal) and another buffer contains the corresponding pointer for each voxel in the first buffer. Note that these two buffers consist of all potential voxels' information, empty and valid.

First, we scan 8 voxels per thread to find whether there are nonempty voxels among them. If yes, we accumulate the color and normal value, then create a parent voxel based on the average value of all nonempty voxels. At the same time, we create 8 nodes using the voxels and pointers in input buffers and store them in a pre-allocated node buffer. Finally, we store the parent voxel and its pointer (the index of the first node of aforementioned 8 nodes in node buffer). By repeating this process for each depth in octree, we achieve the octree construction from bottom to top. Below is the pseudocode of our octree construction (one iteration), we can switch the pointers of *voxelList* and *nextVoxelList* after each iteration.

```
Input voxelList, pointerList
Input threadIdx
counter = 0
acc_color, acc_normal = 0
for i = threadIdx * 8, i < threadIdx * 8 + 8, i++
     if !empty(voxelList[i])
          acc_color += voxelList[i].color
```

```
            acc_normal += voxelList[i].normal
            counter++
if counter > 0
      index = AtomicAdd(nodeCount, 8)
      nextVoxelList[threadIdx] = Voxel(acc_color/8, acc_normal/8)
      nextPointerList[threadIdx] = index
      for i = 0, i < 8, i++
            node[index + i] = Node(voxelList[threadIdx * 8 + i],
            pointerList[threadIdx * 8 + i])
```

The octree's capacity depends on the maximum depth. In the worst case, the octree needs to contain all possible nodes in each level. Given an octree with maximum n levels, the total amount of nodes is

$$1 + 8 + 8^2 + 8^3 + \cdots + 8^n = \frac{8^{n+1} - 1}{8 - 1}$$

However, due to the sparseness of our voxel data, the number of nodes generated is far less than this. The size of the *nodeList* should be determined by the geometry in practice.

Dynamic Update

The dynamic objects in the scene should be updated separately in runtime. After octree construction, we should have a complete octree structure of all static objects in the scene. The structure is called *static octree*, in contrast to *dynamic octree*. In order to quickly undo the updating, the static octree should remain unchanged in the process of updating. Therefore, we only modify part of the *nodeList* at the end of all static nodes.

Figure 22 dynamic nodes should be updated at the end of static nodes (empty part in the list). The benefit of this is that when we want to delete dynamic nodes, we can simply clear all the data after the last static nodes in *nodeList*.

The method of updating is pretty similar to construction, while the major difference is that every time we need to create a node, we need to do depth-first searching (this is easy using Morton code) in the static octree to find whether there is an existing node. If yes, we need to copy static node and merge it with the new node we created. Thus, the dynamic octree can be connected with the static octree.

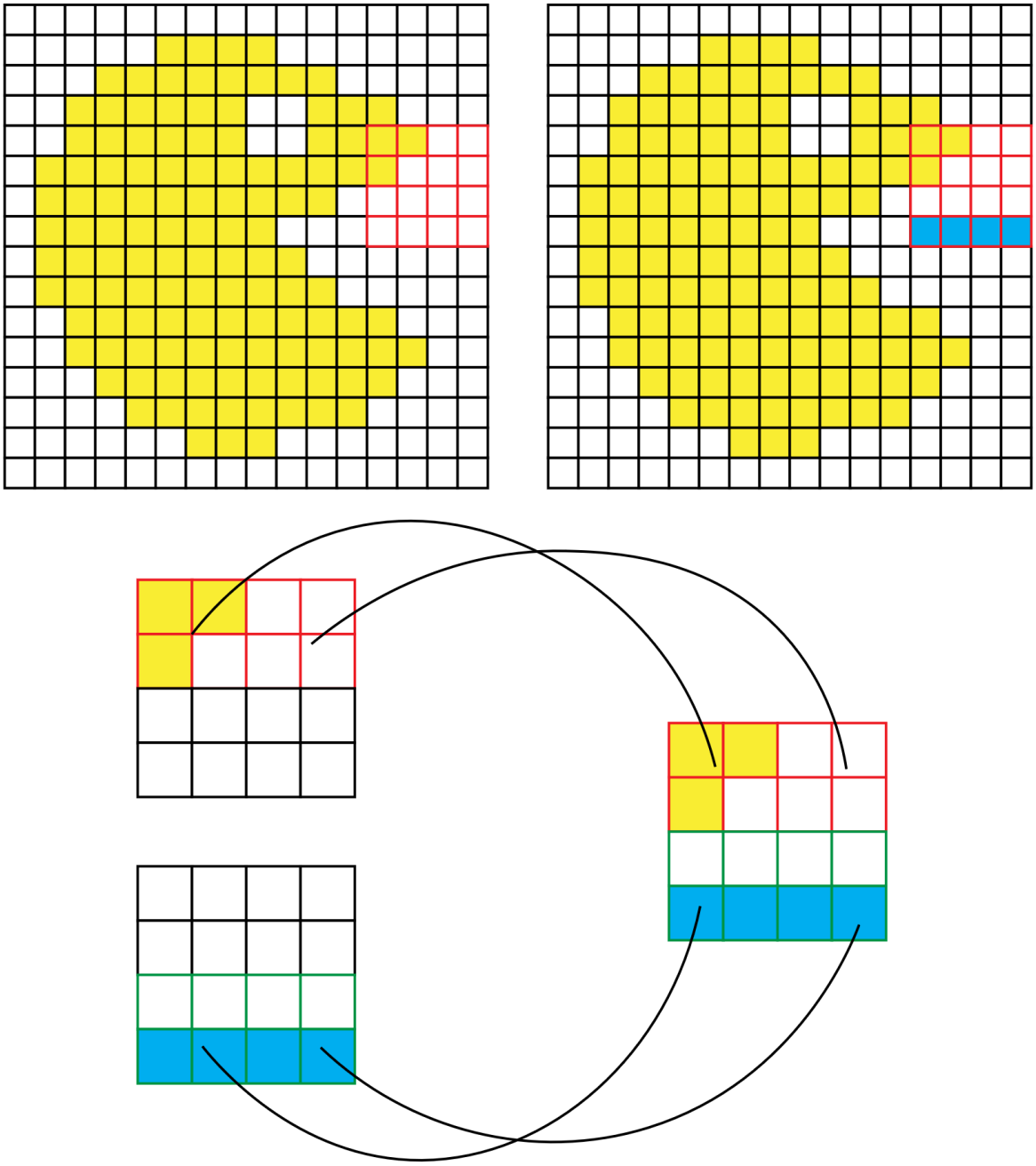

Figure 23 shows how to merge a static node and a dynamic node while updating.

After updating, we will use the new root generated instead of the static root for rendering. When octree need to be updated again or the dynamic object is destroyed, we can switch back to the static root immediately.

Rendering

Although rendering is beyond the focus of this study, we still need a ray-casting algorithm to demonstrate our results. The standard method to render a voxel grid is ray-marching. For each pixel on the screen, a ray is casted out to move step by step in the voxel grid and accumulate occlusion information along the ray direction. The time complexity of this method is apparently $O(n)$, where n equals the size of voxels in the grid. Instead, using octree will help us reduce the time complexity to $O(\log n)$.

Ray-casting can be easily implemented based on our data structure. We start traversing from the root node. First, we need to find all nonempty (this can be obtained using the bitmask in parent node) child nodes that are hit by the ray. Second, sort them in an ascending order of t, where t is the distance from the ray origin and the hit point. Finally, start the traversal again in child nodes. Repeat the above process until we find the first node in the maximum depth. Because we are always tracing the nearest node over ray origin, the result is guaranteed to be correct in rendering.

CHAPTER 5 RESULTS

In this chapter, we present the results of our sparse voxel octree algorithm, as well as images rendered with using our data structure. Our test platform is RTX 3080 8GB. The dimension of our voxel grid is 512^3, which corresponds to a sparse voxel octree of 9 depths. The meshes we are using include *Stanford bunny*, *Stanford dragon*, *happy Buddha* and *Crytek Sponza*. The resolution in the ray-casting algorithm is 1280×720. The following table shows the details in each scene, the performance of our voxelization, octree construction and octree updating methods.

Table 1: Detailed amount of triangle, voxel and node in the test scene of our algorithm. The number of voxel or node of the dynamic mesh may vary in runtime. The voxel dimension is 512^3, and the actual size of voxel is depending on the AABB of the static mesh.

	Triangle	Voxel	Node
Stanford Bunny	4968	35300 ± 300	16500 ± 300
Stanford Dragon	871414	4880000 ± 30000	643000 ± 3000
Happy Buddha & Cube	110782	1261258	843576
Crytek Sponza	262205	7355495	5579960

We designed two scenes to experiment on our dynamic updating algorithm. In the first scene, the static objects include the *Happy Buddha* and a cube, while the dynamic object in the scene is the *Stanford Dragon*. The dynamic mesh is rotating orbiting its center. We also tried to make two meshes overlap as much as possible to test the robustness of our algorithm. In the other scene, the static object is the *Crytek Sponza* and the dynamic object is *Stanford Bunny*. The dynamic mesh is moving uniformly along a straight line.

Table 2: timings of critical processes of our method measured in milliseconds. The voxelization is divided into two phases: preprocess and voxelization.

	Buddha and Dragon (ms)	Crytek Sponza (ms)
Voxelization	0.32/0.82	0.25/401.26
Octree Construction (bottom-top)	20.33	20.80
Octree Construction (top-bottom)	15.88	21.18
Dynamic Voxelization	1.29/1.07	0.17/0.20
Dynamic Update	22.23	20.90
Ray-casting	10.04	20.12

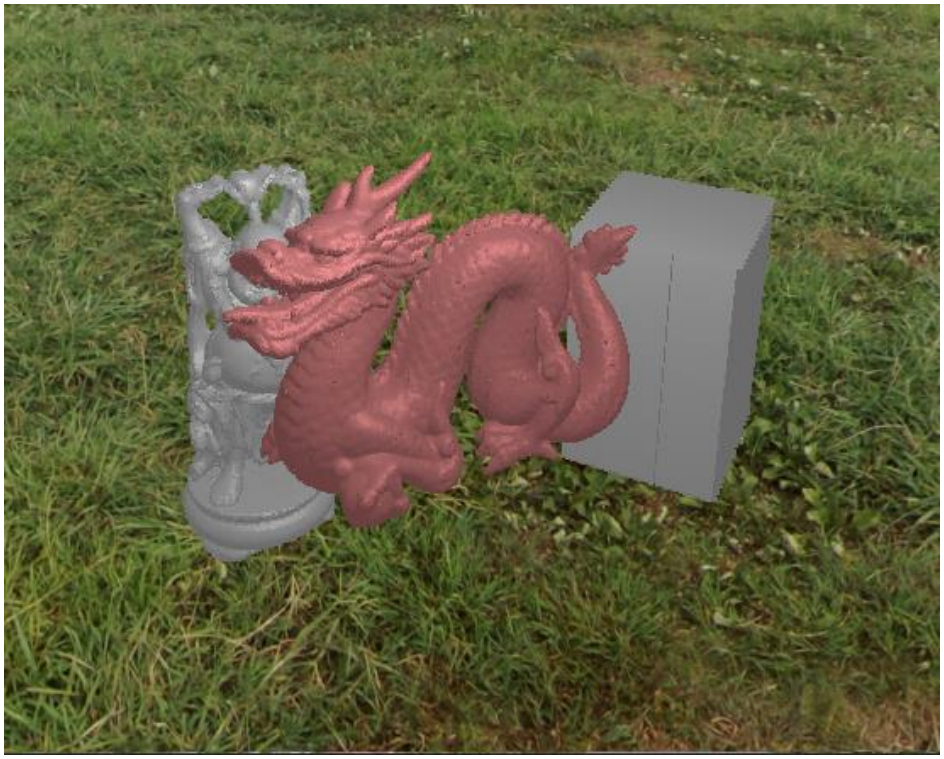

Figure 24 Rendering of the test scene. The white mesh is static while the red mesh is dynamic. The dynamic mesh is revoxelized and updated in octree in every frame. Note that how our method merges static nodes with dynamic nodes when two objects overlap.

Figure 24 continued

Figure 24 continued

Figure 24 continued

CHAPTER 6 CONCOLUSION

Discussion

We have presented an alternate method to construct and update sparse voxel octree in real time on GPU. Our method utilizes Morton code to organize voxels, which results in easy octree construction and traversal. We preprocess triangle information to accelerate the voxel-triangle intersection test in voxelization. In dynamic updating, our method achieves fast undo and modifying SVO by copying and merging new nodes with static nodes. Our algorithm is fully automatic and supports complex geometry.

Similar to other voxel-based methods, the time complexity of our method is independent from scene's geometry complexity. Instead, it seems to be relative to the area of the geometry. Because we are launching one thread per triangle, the more triangles the mesh has, the faster our algorithm performs. We should avoid large triangles in practice as much as possible. It is usually a good idea to subdivide the large triangles before voxelization.

Our method shows strong robustness in updating even when two objects overlap. Although we only show translation and rotation in our experiments, it would work for any animated objects. Because animations will only influent the input geometry information and will have no direct effect on our voxel-based algorithm.

Notice that our method of octree construction is slower in some cases than Crassin's method (top-down). This is because in our octree construction and updating method, we need to traverse all the nodes whether they are empty or nonempty, while Crassin's method only need to traverse all valid voxels. However, our approach shows better performance in more complex scenes. In development, we always find that most area in the octree remains empty. It would be a good idea to use some global variable to constrain our method, so that we will not open thread for areas that are totally empty. For example, we can use two variables to record the smallest and the largest Morton code encountered, and only construct the octree based on nodes between these two values. We can use atomic exchange operation to keep updating them, we would like to explore how effectively it can simplify our method.

Our algorithm of octree updating is running in constant time among different scenes. This may be beneficial in complex scenes with numerous dynamic objects. While Crassin's method is

faster when fewer voxels need to be updated. Because the octrees that are generated by different construction methods are identical. In real application, we can switch between two methods adaptively to achieve the best performance.

Future Works

One of the most important implementations of our method is dynamic global illumination. It is well known that static global illumination can be approximated by using pre-baked textures. However, interactive global illumination is a very challenging topic in computer graphics. SVO has been proved to be helpful to some real-time global illumination methods. However, voxels are essentially tiny axis-aligned cubes. Therefore, pre-filtering of voxel data may be needed for anti-aliasing in global illumination.

Another important goal is to incorporate our algorithm into graphics APIs. In this study, our method is implemented in CUDA because its flexibility and convenience in GPU programming. However, in graphics APIs, memory management and GPU programming would be more challenging. But it is feasible considering most graphics APIs have supported GPGPU. Besides, we can take advantage of the fast rendering pipeline of graphics APIs to improve the performance of our voxelization method.

In our experiment, the dynamic part of the objects and the octree are revoxlized and reconstructed in every frame. In real application, this level of accuracy is usually less important. It is common to distribute the update of indirect illumination temporally into multiple frames for better performance. Likewise, the update of octree can be separated into multiple frames because the octree is constructed level by level.

Inspired by Zeng et al. (2013), we find that nodes in the top few levels of the octree are always occupied regardless of the scene's arrangement. We wonder whether we can use the similar approach to accelerate our octree operations. Also, it is interesting to test different space filling curves other than Morton Order to analyze their performance in octree updating.

Finally, in graphics applications, some dynamic objects do not keep moving in the whole time. In other word, how can we choose the objects selectively, so that only the objects that moved in current frame will be updated. One idea is to create multiple layers of dynamic octrees. This needs further experiment in the future.

REFERENCE

Baert, J., Lagae, A., & Dutré, P. (2014). Out-of-Core Construction of Sparse Voxel Octrees. *Computer Graphics Forum*, *33*(6), 220–227. https://doi.org/10.1111/cgf.12345

Bauszat, P., Eisemann, M., & Magnor, M. (2011). Guided Image Filtering for Interactive High-quality Global Illumination. *Computer Graphics Forum*, *30*(4), 1361–1368. Oxford, UK: Blackwell Publishing Ltd. Retrieved July 25, 2020 from https://doi.org/10.1111/j.1467-8659.2011.01996.x

Burnes, A. (2019, August 27). *Control: Multiple Stunning Ray-Traced Effects Raise The Bar For Game Graphics*. Artificial Intelligence Computing Leadership from NVIDIA. Retrieved July 12, 2020 from https://www.nvidia.com/en-us/geforce/news/control-rtx-ray-tracing-dlss-out-now/.

Campbell, P. M., Devine, K. D., Flaherty, J. E., Gervasio, L. G., & Teresco, J. D. (2003). Dynamic octree load balancing using space-filling curves. *Williams College Department of Computer Science, Tech. Rep. CS-03-01*.

Careil, V., Billeter, M., & Eisemann, E. (2020). Interactively Modifying Compressed Sparse Voxel Representations. *Computer Graphics Forum*, *39*(2), 111–119. https://doi.org/10.1111/cgf.13916

Crassin, C., & Green, S. (2012). Octree-Based Sparse Voxelization Using the GPU Hardware Rasterizer. *OpenGL Insights*, 303–320. https://doi.org/10.1201/b12288-26

Crassin, C., Neyret, F., Sainz, M., Green, S., & Eisemann, E. (2011, September). Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum*, *30*(7), 1921–1930. Oxford, UK: Blackwell Publishing Ltd. Retrieved July 7, 2020 from https://doi.org/10.1111/j.1467-8659.2011.02063.x

Crassin, C., Neyret, F., Lefebvre, S., & Eisemann, E. (2009). Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering. In E. Haines, M. McGuire, D.G. Aliaga, & M.M. Oliveira (Co-chairs), Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games - I3D '09, 15-22. Boston, MA Feb 27-March 1, 2009. New York, NY: ACM. Retrieved July 12, 2020 from https://doi.org/10.1145/1507149.1507152

Dachsbacher, C., & Stamminger, M. (2005). Reflective shadow maps. In A. Lastra & M. Olano (Co-chairs), D. Luebke & H. Pfister (Program Charis). *Proceedings of the 2005 Symposium on Interactive 3D Graphics and Games - SI3D '05*. District of Columbia, Washington, April 3-6, 2005. New York, NY: ACM. Retrieved July 25, 2020 from https://doi.org/10.1145/1053427.1053460

Foley, T., & Sugerman, J. (2005). KD-tree acceleration structures for a GPU raytracer. In M. Harris & D. Luebke (Co-chairs), *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware - HWWS '05*. Los Angeles, CA. July 30-31, 2005. New York, NY: ACM. Retrieved 8 July, 2020 from https://doi.org/10.1145/1071866.1071869

Goral, C. M., Torrance, K. E., Greenberg, D. P., & Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces. In D. A. Luther, R. M. Mueller, R. A. Weinberg & R. A. Ellis (Charimans), H. Christiansen (Editor). *Proceedings of the 11th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '84*, 213–222. New York, NY: ACM. Retrieved July 5, 2020 from https://doi.org/10.1145/800031.808601

Hachisuka, T., & Jensen, H. W. (2009). Stochastic progressive photon mapping. In M. Inakage (Co-chairs). *ACM SIGGRAPH Asia 2009 Papers on - SIGGRAPH Asia '09*. Yokohama, Japan, December 16-19, 2009. New York, NY: ACM. https://doi.org/10.1145/1661412.1618487

Horn, D. R., Sugerman, J., Houston, M., & Hanrahan, P. (2007). Interactive k-d tree GPU raytracing. In B. Gooch & P. Sloan (General Chairs), J. Cohen, G. Turk & B. Watson (Program Chairs). *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games - I3D '07*, 167–174. Washington, Seattle, April 30 – May 2, 2007. New York, NY: ACM. Retrieved July 28, 2020 from https://doi.org/10.1145/1230100.1230129

Hoetzlein, R. K. (2016). GVDB: Raytracing sparse voxel database structures on the GPU. In *Proceedings of High Performance Graphics* (pp. 109-117). http://dx.doi.org/10.2312/hpg.20161197

Jensen, H. W. (1996). Global Illumination using Photon Maps. In: Pueyo X., Schröder P. (eds). *Eurographics Rendering Techniques '96*, 21–30. Eurographics. Springer, Vienna. Retrieved July 5, 2020 from https://doi.org/10.1007/978-3-7091-7484-5_3

Kajiya, J. T. (1986, August). The rendering equation. In D.C. Evans & R.J. Athay, *ACM SIGGRAPH Computer Graphics*, *20*(4), 143–150. New York, NY: ACM. Retrieved July 12, 2020 from https://doi.org/10.1145/15886.15902

Kaplanyan, A., & Dachsbacher, C. (2010). Cascaded light propagation volumes for real-time indirect illumination. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D 10*. https://doi.org/10.1145/1730804.1730821

Keller, A. (1997). Instant radiosity. In G. S. Owen, T. Whitted & B. Mones-Hattal (Chairman). *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques - SIGGRAPH '97*. ACM Press/Addison-Wesley Publishing Co. https://doi.org/10.1145/258734.258769

Kämpe, V., Sintorn, E., & Assarsson, U. (2013). High resolution sparse voxel DAGs. *ACM Transactions on Graphics*, *32*(4), 1. New York, NY: ACM. https://doi.org/10.1145/2461912.2462024

Kim, Y., Kim, B., & Kim, Y. J. (2018). Dynamic deep octree for high-resolution volumetric painting in virtual reality. *Computer Graphics Forum*, *37*(7), 179–190. https://doi.org/10.1111/cgf.13558

Lafortune, E. P., & Willems, Y. D. (1993). Bi-directional path tracing. *Proceedings of Third International Conference on Computational Graphics and Visualization Techniques (Compugraphics '93)*, 145–153. Alvor, Portugal, December, 1993. Retrieved July 26, 2020 from http://graphics.cs.kuleuven.be/publications/BDPT/

Laine, S., & Karras, T. (2010). Efficient sparse voxel octrees. *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D 10*. https://doi.org/10.1145/1730804.1730814

Lefohn, A. E., Sengupta, S., Kniss, J., Strzodka, R., & Owens, J. D. (2006). Glift. *ACM Transactions on Graphics*, *25*(1), 60–99. https://doi.org/10.1145/1122501.1122505

Nichols, G., Shopf, J., & Wyman, C. (2009). Hierarchical Image-Space Radiosity for Interactive Global Illumination. *Computer Graphics Forum*, *28*(4), 1141–1149. Oxford, UK: Blackwell Publishing Ltd. Retrieved July 19, 2020 from https://doi.org/10.1111/j.1467-8659.2009.01491.x

Pineda, J. (1988, June). A parallel algorithm for polygon rasterization. In *Proceedings of the 15th annual conference on Computer graphics and interactive techniques* (pp. 17-20).

Schwarz, M., & Seidel, H.-P. (2010). Fast parallel surface and solid voxelization on GPUs. *ACM SIGGRAPH Asia 2010 Papers on - SIGGRAPH ASIA '10*. https://doi.org/10.1145/1882262.1866201

Shanmugam, P., & Arikan, O. (2007). Hardware accelerated ambient occlusion techniques on GPUs. *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games - I3D '07*. https://doi.org/10.1145/1230100.1230113

Sun, C., & Agu, E. (2015). Many-Lights Real Time Global Illumination Using Sparse Voxel Octree. *Advances in Visual Computing Lecture Notes in Computer Science*, 150–159. https://doi.org/10.1007/978-3-319-27863-6_14

Thiedemann, S., Henrich, N., Grosch, T., & Müller, S. (2011). Voxel-based global illumination. *Symposium on Interactive 3D Graphics and Games on - I3D '11*. https://doi.org/10.1145/1944745.1944763

Zeng, M., Zhao, F., Zheng, J., & Liu, X. (2013). Octree-based fusion for realtime 3D reconstruction. *Graphical Models*, *75*(3), 126–136. https://doi.org/10.1016/j.gmod.2012.09.002