Institut für Visualisierung und Interaktive Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit Nr. 3691

Dynamic Acceleration Structures for the Visualization of Time-Dependent Volume Data on the GPU

Hajun Jang

Studiengang:	Informatik
Prüfer:	Prof. Dr. Thomas Ertl
Betreuer:	Dipl.-Inf. Steffen Frey
begonnen am:	10. September 2014
beendet am:	12. März 2015
CR-Klassifikation:	I.3.7

Contents

Chapter 1

Introduction

Volume Rendering includes different techniques for generating images from three-dimensional scalar data. These data are acquired by means of computer tomography(CT), Magnetic resonance imaging(MRI) and numerical simulations.

Volume rendering extensive 3D data is requiring much computing power and doing this efficiently has been a challenge. To build a data structure of which rendering can take advantage is a common solution. Among others these data structures are built with the purpose of empty space skipping and adaptive sampling. Empty space skipping is the main strategy of acceleration in this work. Octree and Kd-tree is constructed and analyzed in reference to construction and rendering performance.

Time-dependent data often come from numerical simulation and show similarities between two successive time instances. These similarities can be exploited to facilitate construction, instead of executing the high workload of a new data structure construction. In Chapter 4 and 5, its method and consequences will be discussed.

Naive volume rendering is conceptually suitable for parallel computing. On the other side, both constructing and traversing hiearchical data structures do not seem to agree with parallel nature of GPUs. Still, it will be shown how to cope with these problems on parallel architecture.

Chapter 2

Related Work

As an acceleration method for volume rendering, Levoy [Lev90a]introduced hierarchical empty space skipping and early ray termination with opacity. Yagel et al. [YS93] introduced first empty space leaping with a pre-scanned emptiness indication coded in distance from current pixel. emptiness-number of current cell in a regular grid tells how many cells in the vicinity is empty and can be skipped. Since a skip is relative to the direction of the ray, cells within a diameter of the specified distance must all be empty.

For space partitioning, the idea of finding appropriate split for a volume is suggested by MacDonald et al. [MB90]. MacDonald et al. proposes surface area heuristics and the candidates of an optimal split, which may lie between the spatial median and median of objects sorted relatively to the axis of division. Horn et al. [HSHH07] introduced kd-restart algorithm and short-stack as well as packetized short-track. hapala et al. [HDW$^+$13] implements a 'state' based traversal with a parent pointer. Foley [FS05] applies CPU-based kd-tree on GPU and uses kd-backtrack as traversal algorithm.

popov et al. [PGSS07] suggests rope traversal, which connects the spatially adjacent nodes. These nodes have high probability to be traversed in nearby order. Packet traversal groups rays starting from neighboring pixels. The expectation is that the rays intersect the subvolumes uniformly, upon which the multiple sampling process can be reduced.

Frame-to-frame adaptation of BVH volume is shown by Wald et al. [WBS07], for a case where there is only a gradual change between consecutive scenes. Lauterbach et al. [LYTM06] introduces a BVH quality measure to determine at which instance of time the degradation of the tree exceeds a predetermined value and an adaptation is not enough. In this case a complete rebuild of BVH is initiated.

For adaptation, Levoy [Lev90b] adapts the number of rays per pixel according to local image complexity. This is different from adaptation interested in this work, but is a measure of acceleration. Freund et al. [FS97] extends the data exploitation by encoding homogenous region and accelerates volume rendering.

Chapter 3

Fundamentals

To make this paper more readable for the general computer science public, some standard concepts and tools will be introduced in this chapter.

3.1 Volume Ray Casting

One can think volume rendering as depicting each pixel of a window. And this pixel accumulates its color collected by the ray that has been sent through the center of the pixel. The rendering equation from Kajiya formulates the spectral radiance coming from a specific position in the direction of the camera. This equation can deal with scenarios where specular and diffuse reflection is involved. It enables inspection of theoretically indefinite depth of secondary rays. The rendering equation is intrinsically incomplete to describe some sophisticated light phenomena like polarization and interference. But these phenomena are rather eccentric and in decent quality scenes where global illumination(refers to scenes with illumination by secondary rays) is the aim, the rendering equation suffices. The concept of secondary ray is not applicable for direct volume rendering since the model to be rendered is not composed of geometry, but is a distribution of scalar data points.

The light transport model generally used in direct volume rendering is emission-absorption model. This model is adopted in this work, too. According to this model, light is emitted from and absorbed by the 'particles'(scalar values) that constitutes the volume data. It is as if the particles or material in the scene are self-luminous. This model assumes that scattering or indirect illumination does not occur. A ray travels through the 3D space starting from the camera. If a ray hits a particle or a material surface, either it gets absorbed or it penetrates the material. The proportion of the absorbed light in relation to the whole is the opacity. Unlike the global illumination model, there is no secondary light pointing into directions other than the initial ray direction. There is a calculative method called alpha-blending by which the ray sums the particles up. It can be shown that alpha blending can be derived from some physical premises. Let the emission and absorption rate c and k, respectively. Emission at a distance d is expected to decrease with a growing d, multiplied by a factor for absorption, k.

$$\frac{\mathrm{d}c(d)}{\mathrm{d}d} = -kc(d) \tag{3.1}$$

Assuming that k is constant all along the distance, integration over d gives:

$$c' = c \cdot \mathrm{e}^{-kd} \tag{3.2}$$

for varying k along the distance:

$$c' = c \cdot \mathrm{e}^{-\int_0^d k(t)\,\mathrm{d}t} \tag{3.3}$$

Bear in mind that this is an emission of a single particle at a specific distance on the ray. Now consider Integrations of rays starting from all possible positions along the ray. These additive fractions of light must be summed up, starting from distance 0 to infinity, since theoretically every one of these have an effect on the resulting pixel.

$$C = \int_0^\infty c(t) \cdot \mathrm{e}^{-\tau(0,t)}\,\mathrm{d}t \tag{3.4}$$

τ is an abbreviation for the integration of absorption coefficient over a specific distance.

$$\tau(d_1, d_2) = \int_{d_1}^{d_2} k(t)\,\mathrm{d}t \tag{3.5}$$

This Integration, also called optical depth, can be approximated by a Riemann sum.

$$\tau(0,t) \approx \tilde{\tau}(0,t) = \sum_{i=0}^{\lfloor t/\Delta t \rfloor} k(i \cdot \Delta t)\Delta t \tag{3.6}$$

Δt is the distance between the current and the next sampling locations. Summation of the exponents equals Multiplication of the exponential expression. Hence:

$$\mathrm{e}^{-\tilde{\tau}(0,t)} = \prod_{i=0}^{\lfloor t/\Delta t \rfloor} \mathrm{e}^{-k(i\cdot\Delta t)\Delta t} \tag{3.7}$$

Now let us define Opacity A and rewrite eq. 3.7

$$A_i = 1 - \mathrm{e}^{-k(i\cdot\Delta t)\Delta t} \tag{3.8}$$

$$\mathrm{e}^{-\tilde{\tau}(0,t)} = \prod_{i=0}^{\lfloor t/d \rfloor} (1 - A_j) \tag{3.9}$$

Which means the successive multiplication of $1 - A_j$ gives the desired light intensity after successive absorption.

Likewise define emission of a segment and multiply that by absorption of the segmentand sum it up over the entire distance of the ray.The multiplicative product is the component that reaches the camera and the global summation yields the resulting pixel.

$$C_i = c(i \cdot \Delta t)\Delta t \tag{3.10}$$

$$\tilde{C} = \sum_{i=0}^{n} C_i \mathrm{e}^{-\tilde{\tau}(0,i-1)} = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 - A_j) \tag{3.11}$$

these derivations(equations) are from [HLSR09].

Since we don't know the alphas and emissions(colors) before collecting the particles we meet, we successively apply multiplication and summation for every alpha and color value we meet. this is called alpha blending and it coincides with the upper equation.

$$C'_i = C'_{i-1} + (1 - A'_{i-1})C_i \tag{3.12}$$

$$A'_i = A'_{i-1} + (1 - A'_{i-1})A_i \tag{3.13}$$

Finally, alpha blending for color and alpha value is derived, which is used universally in graphic applications dealing with partly transparent objects.

3.1.1 Sampling and Transfer Function

As previously mentioned, 3D volume data consists of scalar data and the data points are distributed in regular 3D grid(different than the regular grid in meaning of a partition of a space, in that a volume cell can contain many data points). It is obvious that the exact sample positions do not coincide with the data points. Sample point is somewhere in-between data points. There are methods to sample using those neighboring data points. Nearest neighbor method simply picks the data point nearest to the sample point. Trilinear interpolation is usually well supported from graphics hardware and more widely used than more sophisticated methods like gaussian or cubic spline filter.

Now the scalar value of the sample is acquired. To perform compositing along the ray, a transfer from the scalar value to R,G,B and alpha value is necessary. This is defined as Transfer function, usually in the form of a look up table. In this case it is a one dimension to four dimension function, but a second argument on the left side can be added, which is for example gradient of the underlying scalar data at the sampling point. Widely used with medical computer tomography or magnetic resonance imaging data, the transfer function can be tweaked to realistically match typical material properties (see fig. 3.1). User-interactive transfer functions, for instance controlling only opacity, can be useful to highlight specific material features.

3.2 Spatial Structures

We can use local property of the data to accelerate the sampling process by omitting some executions of sampling, for example. the concept is to create a division by partitioning plane in the volume. The location of the division can be predefined like octree and regular grid, or it can result from discontinuity of data properties. Generally, higher adaptivity of structure to data yields more efficiency for rendering. It is a trade-off between construction time and rendering efficiency. A data structure can be hierarchical or flat. Space partitioning trees are hierarchical, the space partitioning algorithm is executed recursively on each level of the tree. Whereas, the grid is a flat structure offering no hierarchy and is made up of congruent cells. It is possible to conceive a space partitioning plane that is not aligned with an axis. But for reason of compactness and aptness to computation, non-axis aligned volumes are a peculiarity and this work will be confined to the axis-aligned planes. axis-aligned bounding boxes are shortened AABB. Among space partitioning data structures a handful deserve interest up to now. Frequently discussed structures including the ones implemented in this work will be introduced below.

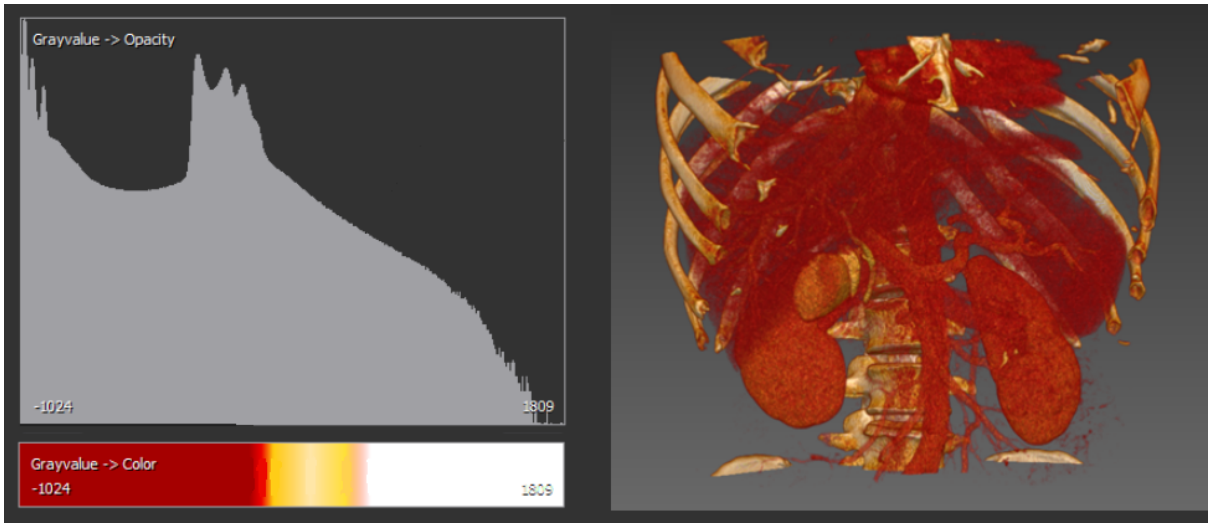

Figure 3.1: Scalar data is classified into Blood, Tissue, Fat, Bone region according to their density coming from data acquisition. Appropriate setting of transfer function in accordance with material helps classification of visible features. image courtesy of Mint Medical GmbH

3.2.1 Regular Grid

3D Volume data has a defined extent. In the Data used in this work, it spans from -1.0 to 1.0 in x, y and z axis. The most simple and perhaps basis for other structures is the regular grid. To be exact, 'regular grid' in this work refers to cartesian grid which has identical dimensions in three axis, whereby the exact definition of regular grid is tessellation of n-dimensional Euclidean space by congruent parallelotopes. The convention seems to be using the term regular grid instead of cartesian grid.

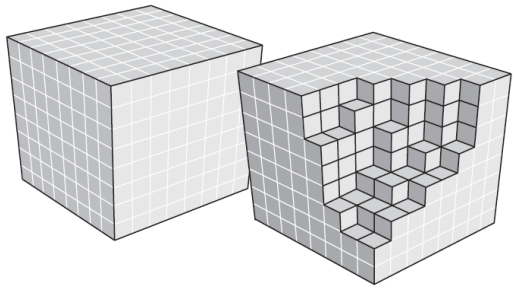

Figure 3.2: regular grid (image from [HLSR09])

3.2.2 Octree

One of the intuitive ideas to divide a geometry is to cut it into 2 congruent parts. Dealing with a 3D cube, one can think of dissecting the volume through 3 planes perpendicular to each of x, y, z axis. this procedure yields 8 sub-volumes, which can be repeated at any each level of subdivision. this recursive property is depicted at figure 3.3.

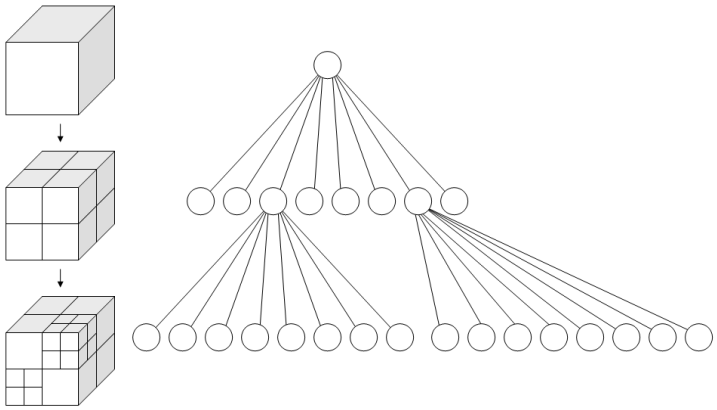

Figure 3.3: Octree, depicting one subdivision at depth 1 and two subdivisions at depth 2. (image from [Man06])

The subdivision can be quit at a predefined depth or if a criteria for termination is met. The termination criteria for volume partitioning is in many cases the number of items in the volume. Another common criteria is the bound of maximum depth. This can greatly reduce construction costs and memory requirements.

Octree and binary space partitioning tree finds a broad application area. N-dimensional data can be mapped into three spatial axis and n-3 -dimensional data point. An example is the color quantization algorithm and nearest neighbor searching algorithm.

3.2.3 k-D Tree

k-D tree is short for k-dimensional tree and belongs to binary space partitioning tree. The difference to octree is that the splitting plane is not fixed but dependent on geometry or data. This raises construction cost significantly, which will in general pay off at the later rendering phase. An apt choice of splitting plane yields a well balanced kd-tree, and this kd-tree approachs theoretic depth of $\log N$.(In the case of ray tracing, N is the number of total triangles or primitives, but for ray casting this formulation does not hold.)

The computation of splitting plane for ray tracing is a well studied issue. Goldsmith et al. proposed an algorithm that computes an optimal splitting plane [GS87]. Before then there has been manual setting of split planes and bounding boxes, but no automatic generation of volume division. Series of developments and improvements on the cost prediction function serves as a Heuristic toward an optimal splitting plane [MB90].

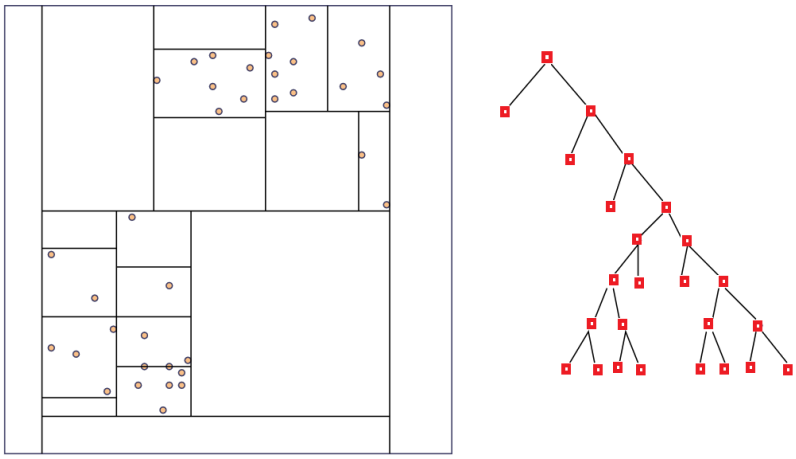

Figure 3.4: A Kd-tree is created from 2D data. The split planes are set in favor of empty space skipping.

The cost prediction function by MacDonald et al. [MB90] is called the Surface Area Heuristic(SAH) and computes the estimates of traversal cost after a volume has been subdivided by a candidate plane (see eq. 3.14). At each Node, greedy strategy to choose the plane with the minimum cost is applied. If the minimum traversal cost is greater than the cost of intersection tests for the original node without splitting, no splitting is performed and the node is set as a leaf. This criteria of termination of tree construction seems more consequent and plausible, if memory consumption is not an issue.

$$
\begin{aligned}
C_{split}(V_l, N_l, V_r, N_r) &= C_{trav} + C_{isec}(P(V_l|V)N_l + P(V_r|V)N_r) \\
\text{where } P(V_l|V) &= \frac{SA(V_l)}{SA(V)} \text{ and } P(V_r|V) = \frac{SA(V_r)}{SA(V)} \\
\text{surface area of a volume } SA(V) &= 2(V_w V_d + V_w V_h + V_d V_h)
\end{aligned}
\tag{3.14}
$$

it is possible to calculate the probability with which a ray that hits a volume also hits any of its sub-volumes. having a voxel V that is partitioned into two voxels VL and VR, the probability of a ray traversing these two sub-voxels can be calculated with $P(V_l|V))$ and $P(V_r|V))$. N_l and N_r is the number of primitives in the volume, C_{trav} is the traversal cost and C_{isec} the cost of a single intersection computation.

There are unrealistic assumptions under which the SAH is proposed. As the summation of area runs over all six areas about a volume, rays are supposedly coming from all directions with equal probability. but if one considers a scene with a volume box lying on a ground, the surface facing to the ground remains intact from rays. Imagine a scene with a cluster of great number of objects, primitives visible from outside occlude the ones in the center. This is not a dissipation of tree if the camera can move into the cluster. Otherwise if those objects can be neglected from vision, occlusion-culling can be applied. Moreover, the original SAH is supposed to compute a perfect tree which optimizes traversal cost, but does not consider construction cost. Simple modifications to reduce computation like lessening cost evaluation points do exist. These downsides might pose no problem for a static scene, but not for a dynamic, time-dependent scenes.

Just like octree (section 3.2.2), termination criteria of kd-tree construction can be a decisive performance factor. Havran et al. regards termination at less or equal triangles to 2 and at depth 16 as optimal performance criteria. But the total number of primitives differ from scene to scene, hence the maximum depth should be scene dependent and cannot be applied universally. For instance, if a volume at depth 16 contains a high number of triangles, it might be profitable to further divide the volume. Of course, this is justified only when the traversal cost does not exceed the cost of intersection tests. correspondingly, Pharr and Humphreys use $8 + 1.3 \log N$ in various scenes as the termination criteria for tree depth [PH04].

3.2.4 Bounding Volume Hierarchy

bounding volume hierarchy(BVH) is another space partitioning hierarchical structure. The bounding volumes do spatially belong in the parent, to conform with hierarchy. The difference is that the union of two children does not necessarily make up the parent volume. And the two children may intersect. Therefore for each node the bounding box must be saved and a splitting plane is not enough. This means more memory consumption. But this cost is outweighed by a compact tree, compared to kd-tree when considering building up same bounding boxes. While traversal, far less intersection steps must be performed (see figure 3.5). For static scenes, manual definition of sub-bouning-volumes is thinkable.

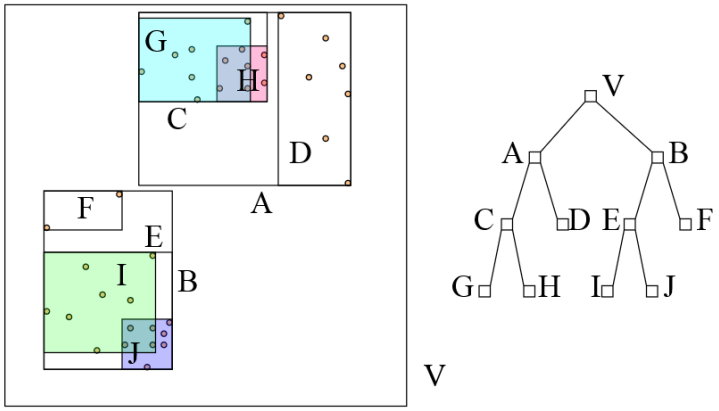

Figure 3.5: A BVH created for empty space skipping, from the same data as fig. 3.4. The tree is more compact then kd-tree in fig. 3.4.

3.3 CUDA

CUDA is short for Compute Unified Device Architecture and is a parallel computing platform and programming model created by NVIDIA. below is a simplified diagram showing the process flow of CUDA.

CUDA source code (.cu extension) written in C or C++ must be compiled with nvcc, Nvidia CUDA Compiler. Unlike GLSL, CUDA source code can contain codes that run on CPU and GPU. Functions and variables are to be marked with qualifiers 'host' and/or 'device', denoting that those are to be run on

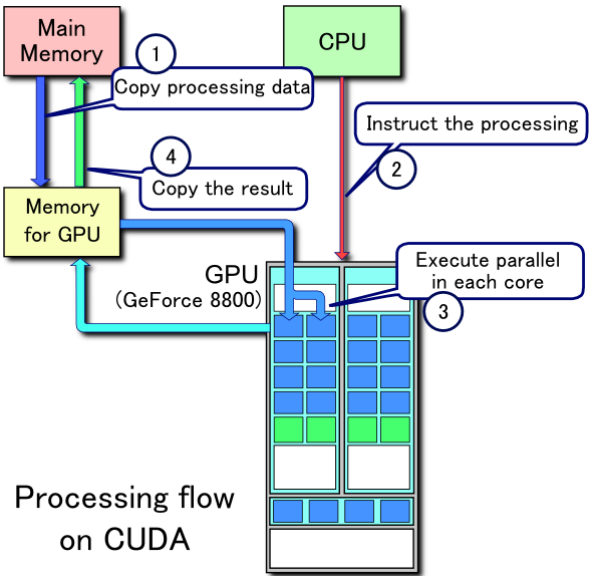

Figure 3.6: 1. Copy data from RAM to GPU memory
2. Instruction from CPU to GPU
3. Parallel execution on GPU
4. Copy results back to GPU image from [Tos]

CPU and/or on GPU, respectively. Nvcc then passes the host code to C compiler like GCC and device code is further compiled and sent to GPU.

There are significant advantages of CUDA over other GPGPU possibilities, which made CUDA so common. among others, memory usage is more advanced in allowing code to access arbitrary GPU memory. additionally CUDA is equipped with shared memory which can be shared among threads in the same block. Shared memory up to 48KB per block provides very fast read and write access. If the result of the computation is not directly to be displayed on graphics output, it has to be copied back to CPU memory. This overhead with the initial data copy before CUDA computation is a performance bottleneck in many cases. To list further negatives of CUDA, it can run only on Nvidia graphic cards and no emulator is provided for alternative GPUs. To save run-time costs, exception handling within device code is excluded. In case of a run-time error, only by running the code with debugger like cuda-gdb or cuda-memcheck it will be possible to spot the thread where the error occurred.

3.3.1 CUDA Thread Hierarchy

The function that runs parallel on GPU is called kernel and must be labeled with function declaration specifier 'global'. Along with the mandatory information of how many threads should be invoked, address to data or user input can be passed as parameters. The threads are organized into blocks, and blocks form grids. The numbers of blocks and grids can be defined with integer or integer vector type dim3. In the code below, 2-dimensional block and grid is created to hierarchically administer threads. The thread IDs are automatically created on local memory for each thread. In the device code, these

are used to identify the thread and guarantee mutual exclusion between threads by avoiding access to identical memory address.

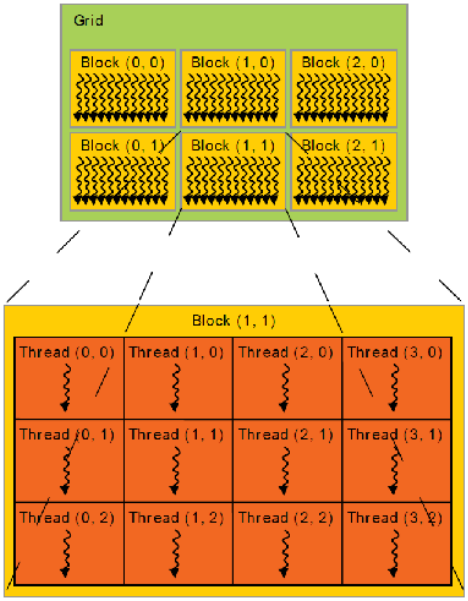

Figure 3.7: kernel invocation with gridDim(3,2) and blockDim(4,3) block and grid can be scalar, 2- or 3-dimensional.

In the listing 3.1 Every one of $4 * 4$ blocks contain $8 * 8$ threads. Total number of threads are $(4 * 4 * 8 * 8 = 1024)$ equal to array size, so assignment of workload for each thread is plausible. cudaMalloc is an analogue to C malloc and allocates specified bytes of global memory. cudaMemCpy is responsible for reads and writes between host and device and within each.

```
__global__ void kernel_func(int* a){
    int thread_index = blockIdx.x *gridDim.y*blockDim.x*blockDim.y
                              +blockIdx.y *blockDim.x*blockDim.y
                              +threadIdx.x *blockDim.y
                              +threadIdx.y;

    foo(a[thread_index]);
}
int main(){
    int a[] = {1, 2, 3, ..., 1024}
    int* dev_a;
    int num_bytes = 1024 * sizeof(int);
    cudaMalloc((void**)&dev_a, num_bytes);
    cudaMemCpy(dev_a, a, num_bytes, cudaMemCpyHostToDevice);
    dim3 gridSize(4, 4);
    dim3 blockSize(8, 8);
    kernel_func<<<gridSize, blockSize>>>(dev_a);
```

```
    cudaMemCpy(a, dev_a, num_bytes, cudaMemCpyDeviceToHost);
}
```

Listing 3.1: A kernel function call in CUDA

In case of GeForce GTX TITAN, it has 14 streaming multiprocessors(SM). Each of these SMs contain 192 cores and a core executes one kernel block at a time. So $192 \times 12 = 2688$ blocks can be worked on simultaneously. In a core, a bundle of 32 threads called a warp are executed parallel in SIMT fashion. Threads in a warp execute the same instruction for each clock. if kernel_func$<<<$ $1024, 48$ $>>>$ is launched, 1024 cores will be active and be running the same code twice, in order to handle 48 threads once with 32 active threads and 16 threads the next. Since a warp runs regardless of whether all of its threads are active, it yields better performance to distribute threads over greater number of blocks. Let the number of total instructions in the kernel code 100 and consider a kernel function call with kernel_func$<<<$ $2048, 24$ $>>>$. 2048 blocks will be invoked and 1warp will do the job, so total number of parallel instructions are 100. Performance is improved compared with the previous case of 48 threads per block giving 200 instruction cycles. Generally it is advantageous to set the number of threads per block a multiple of 32. As seen in the example, for full exploitation of parallel computing power you may want to launch a broader grid than to launch a broader block.

3.3.2 CUDA Memory

Memory on CUDA-capable GPUs are hierarchically organized (see fig. 3.8). Since memory transfers are the most time-consuming jobs in a cuda application and memory copies between host and device are to be reduced to a minimum(while theoretical maximum bandwidth between host and device is 8GB/s on PCIe x16 Gen2, memory Bandwidth on geforce gtx titan is 288.4GB/sec), prompt rule of thumb is to use as much on-chip memory as possible. Still, off-chip memories as texture and constant memory serve its own purpose. Additionally, cuda provides some methods by which it partly overcomes the shortcomings of off-chip memory and alleviates speed drawback.

3.3.2.1 Global Memory

As aforementioned, global memory is slow to access. But in coalesced patterns of memory access, multiple accesses of global memory can be reduced to one. cudaMalloc allocates the start of a memory block at an aligned address. When dealing with 2D array, cudaMalloc2D aligns each row. Access from the threads in a warp coalesce into aligned read that is necessary to cover the memory area required by all threads in the warp. If the accesses are far from adjacent but scattered, suffer from inefficiency since greater proportion of irrelevant memory area is fetched. This inefficiency is worsens when L1 cache with 128 byte lines is used. This holds for devices with compute capability 2.x. Devices with compute capability 3.x use L2 cache with 32 byte lines for global memory access.

Since global memory is available from everywhere within application, threads running parallel shall not write to the same address without synchronization. The precedence is undefined in this case and it leads to undefined value. Global memory allows more accustomed memory handling in c-fashion. Memory address can be accessed by pointers and pointers can be passed to kernel functions. Pointers can be defined by keyword __device__ and can be allocated by cuda function cudaMalloc or cudaMallocPitch.

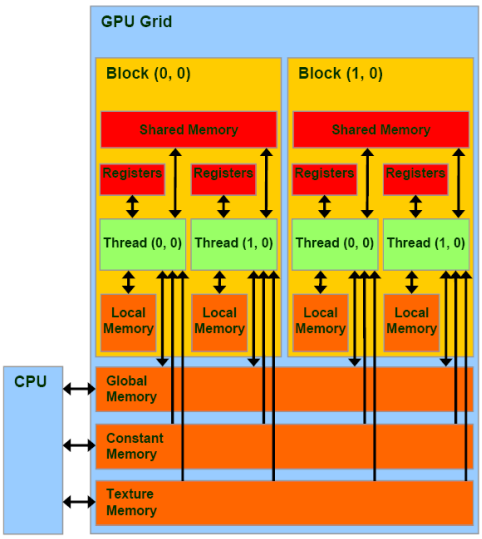

Figure 3.8: Register memory is on chip, can be served from a thread, but cannot be accessed by addressing. Shared memory has block-scope and on chip. Local memory has thread scope but off chip, so very slow and auxiliary to register. Global, Constant and Texture memory is off chip and can be written from cpu. Among these three only to global memory write access is granted from gpu code. Scope of the last three memories run over all threads.

Host and device pointer can not be mixed by any means. Dereferencing a host pointer in device code and vice versa will cause a run-time error.

3.3.2.2 Constant Memory

Among the DRAM-memories off GPU, constant memory and texture memory are specialized for read-only constant values and texture and perform efficient caching. Constant memory performs broadcasting if all threads in the warp accesses the same memory. In this case it is way faster than texture memory and makes the advantage of constant memory. As the name suggests, it best performs when holding constant data and kernel arguments that is used in common over all threads. If a thread accesses differently though, memory read serializes and gets slower than uncached texture memory read. Using constant memory is described as profiting from temporal locality while kernel function is running. Geforce gtx titan has 64KB of constant memory and 8KB of constant memory cache.

3.3.2.3 Texture Memory

In cuda hardware texture processing cluster is assigned per a few streaming multiprocessors and each cluster contain a texture specific unit. Texture unit is composed of texture address units and texture filtering units and share a read-only texture L1 cache. Advantages of texture memory regarding addressing and filtering are performed in these units. Cuda provides various address modes to cope with texture addresses that are out of bound (between 0 and 1). For example if texture parameter addressMode is set to cudaAddressModeClamp, parameter 1.23 will yield 1.0, cudaAddressModeWrap will yield 0.23,

cudaAddressModeMirror will yield 0.77. This computation is performed at the addressing unit. filtering unit computes interpolation of texture data. Linear, bilinear and trilinear hardware interpolation is supported. Computations in texture units run parallel outside of kernel and so is offered at no extra cost for the thread. Texture memory use 2D cache, enabling reading with 2 parameters without array address computation.

To explain the spatial locality of constant memory, it should be mentioned that the texture cache is shared by neighboring streaming multiprocessors. This means that it is more efficient when adjacent blocks belong to adjacent streaming multiprocessors, since those blocks have similar access patterns to texture memory. 3.9 left shows that block 3 and 4 belongs to cores wide apart. Considering probably similar texture access of block 3 and 4, the chance to profit from the other blocks texture read is eliminated because of cache miss. Therefore kernel launch adjusts block layout in reference to cores, in order to better exploit L1 texture cache shared by adjacent blocks. This is laid out in the z-order curve 3.9. Thus can the spatial locality of blocks take advantage of texture memory cache.

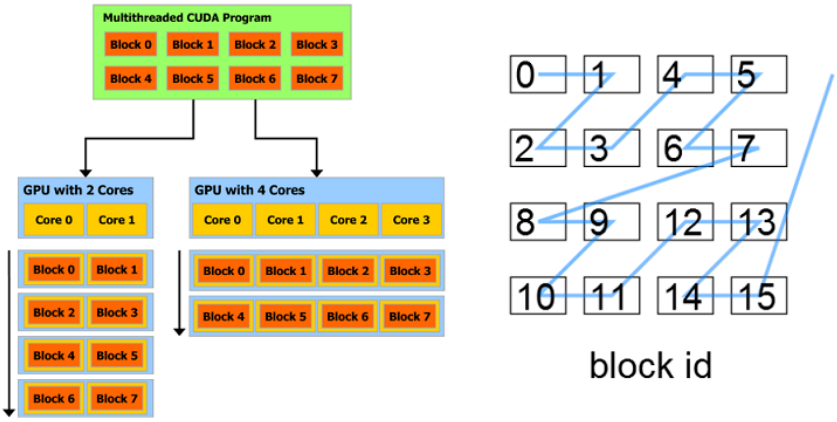

Figure 3.9: left: blocks are dynamically scaled to the numbers of cores. right: blocks are distributed so that the block indexes are in z-curve order and then assigned to each core that stands for the column. block 0,2,8,10 is in core 0, block 1,3,9,11 in core 1 and so on.

3.3.2.4 Shared Memory

Shared memory read and write is up to 10 times faster than global memory since it is located on chip. Shared memory is devised to handle more arbitrary memory access than texture or constant memory, thus reduced the need for caching. Threads may access to different locations at the same time. To cope with this memory access, banks are introduced. Banks are access points for memory partitions with 4 bytes of size. A warp has 32 banks interleaved at 4 bytes distance. Which means bank 0 manages access to 0x000, 0x080, 0x100 and so on, address 0x004, 0x084, 0x104 and so on belong to bank 1. When kernel is launched, each bank serves one memory address pointing to 4 bytes, per clock cycle. So within a block, which is the usage range of shared memory, only 32 (that is the number of banks) accesses can be served in parallel. If more than one shared memory access fall onto the same bank, the access gets undesirably serialized and this coincidence is called bank conflict. To list a few general strategy to avoid bank conflict, do not condense data so closely that they are less than 4 bytes apart. In this case

two or more accesses grab to the exact same bank adress. Second, if the data is bigger than 4 bytes, do not save it in a contiuous shared memory range, but divide them in 4 byte snippets and save them in evenly separated addresses which is accessed by the same bank. Turning array of structures(AoS) into structures of arrays serves the same purpose. If a structure has a size of a multiple of a even number of 4 bytes, chances are high that bank conflicts will occur. In this case, padding the structure in order to allocate it a size of a multiple of a odd number of 4 bytes removes the problem. An example of this fix is to alter a structure of 8 bytes into a structure of 12 bytes by padding 4 bytes. The thinkably worst case of all threads accessing on one bank though, is handled by reading once and broadcasting it to all threads in the warp. In this case read is performed optimally, as fast as when there is no bank conflict. Shared memory access costs as much time as the maximum number of serialized accesses to a bank. In the example of a random access it takes 5 read cycles for the bank number 5.

3.3.2.5 Registers

Registers are another high speed on-chip memory. Physically, registers exist per SM. When a cuda kernel launches, they are dynamically assigned to threads and cannot be accessed by other threads. Locally declared device variables use registers. Register memory cannot be indexed, so local array variables must be stored in local memory. Registers are already a scarce resource though(64 K of regs (16384 32-bit regs) per MP on GTX 200), often resulting in lack.

3.3.2.6 Local Memory

When the register memory is full, local memory in global memory area is used instead. Like register memory, local memory is exclusively assigned to each thread.

For detailed memory specification of the hardware used in this work, see listing 5.1.

Chapter 4

Acceleration Method and its Implementation

The purpose of space partitioning trees can be diverse. For this work, empty space skipping and adaptive sampling are relevant, of which empty space skipping is implemented. This chapter will show the design of construction algorithm in CUDA. Performance is the measure of success of this work, so algorithms implemented will be evaluated and analyzed. Therefore performance factors mentioned in section 3.3 will be considered in implementation.

Principally 3 phases each including one or more kernel calls constitute tree construction and rendering. First phase is setting grid emptiness. In the case of an octree, the size and structure of tree is known prior to construction. Therefore construction can be included in this phase. Second phase is constructing hierarchical structure. This phase is the most performance demanding and thus deserves thorough inspection. Third phase is traversing and rendering, where tree construction should pay off in performance. For time-dependent data, these 3 phases must be executed for each time instance. Alteration to the second phase in order to exploit the properties of time dependency is the central part of this work.

4.1 Grid Emptyness Scan

To reduce the costs of emptiness scan while tree construction a regular grid is laid on the volume and each grid cell is scanned to record its emptiness information of full or empty. Number of data points per axis can be referred to to decide upon the dimension of grid. Tree construction is based on this grid. The construction kernel does not read from texture, but only from grid array. Therefore, granularity of tree-node splits depend on the fineness of the grid. Split planes of kd-tree are set only on grid cell boundary. Thus split planes can have discrete values, and can be saved in an integer or a character.

4.1.1 Octree Emptyness Scan

Octrees have predefined box boundary, hence a computation of split plane is not needed and the construction kernel can be skipped. The information each node contains is whether it is empty, full or neither of them. To create an octree with depth d, an array of length $8^d + 8^{d-1} + ... + 8 + 1$ is allocated. First kernel scans the cell boxes and records whether empty or not. The next kernel collects the cell emptiness of the 8 volume boxes of underlying depth. If all sub-volumes uniformly empty or full, its own box information is set as such. This procedure of collecting the sub-volume continues upwards until the root node is reached. Synchronization between the depths is provided between kernel calls.

4.2 KD-Tree Construction

Kd-node computation relies on emptiness grid recorded in the earlier phase. For each node, construction scans through its volume and decides if it is an inner node or a leaf node. If it is an inner node, split plane must be computed. For leaf nodes the type of node, empty or full, is set.

In this implementation, nodes are assigned to threads in one-to-one correspondence. Being hierarchical, node volume can only be defined after computation of its parent node. Construction kernel for one tree depth is launched after construction kernel for its parents' tree depth. In the code below (listing 4.1), kernel function construct_depth iterates through the tree depth and thereby the sequence of construction is guaranteed. Line for line, cudaMalloc, computation kernel and cudaMemcpyToSymbol is executed. cudaMemcpyToSymbol, called with argument cudaMemcpyDeviceToDevice, performs memory copy within device(between global memory and register memory).

```
void construct_kd_tree_initialize(int maxTreeDepth){

   int maxLeafArrayLength = std::pow(2, maxTreeDepth);

   for(int depthLine = 1;
         depthLine < maxLeafArrayLength +1;
         depthLine *=2){

      kd_node* kd_nodes_temp_array = 0;

      int num_bytes = 3* depthLine * sizeof(kd_node);

      cudaMalloc((void**)&kd_nodes_temp_array, num_bytes);

      construct_depth<<<gridSize, blockSize>>>
         (kd_nodes_temp_array, depthLine);

      cudaMemcpyToSymbol( kd_tree, kd_nodes_temp_array, num_bytes,
         (depthLine -1) * sizeof(kd_node), cudaMemcpyDeviceToDevice );

      cudaFree(kd_nodes_temp_array);
   }
}
```

Listing 4.1: construct_kd_tree_initialize

kd-tree is saved in an array of nodes in top-down and left-to-right sequence. Nodes in the same depth are contiguous and listed from left to right (see fig. 4.1). If the algorithm is at depth 3, depthLine is 4 and an temporary array of length 4×3 is created. This array consists of parent line(4 elements) plus children line(4×2 elements). As many threads as the length of parent line is launched, with the temporary array as argument. Each thread is responsible for the parent node and two children nodes. Each thread computes the split plane of parent node and initializes children nodes. After each computation of depthLine, temporary node is copied to the tree, at the right position.

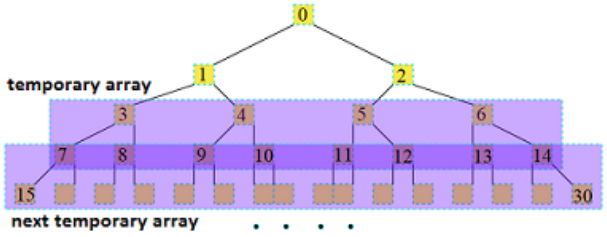

Figure 4.1: The number for a node is the global array index at which the node is recorded. temporary array launches 4 threads writing to 12 nodes. the next kernel launch involves 8 threads writing to 24 nodes.

```
void construct_depth(kd_node* kdn_temp_array, int depthLine){

   kd_node parent;
   parent.init_copy(kd_tree[kd_index]);

   if (BOX_IS_EMPTY==parent.splitDim)||(BOX_IS_FULL==parent.splitDim)
      return;

   compute_maxEmptySlabs(parent, EmptySlabsMin, EmptySlabsMax);

   if emptyOrFull(EmptySlabsMin, EmptySlabsMax){
      parent.splitDim = BOX_IS_EMPTY_OR_FULL;
      return;
   else
      uchar splitPlane =
         chooseSplitPlane(parent, EmptySlabsMin, EmptySlabsMax);
      setChildren(kd_index, splitPlane);
}
```

Listing 4.2: construct_depth

In function construct_depth (listing 4.2), leaf nodes are ruled out from further process and terminate early. For inner nodes, function compute_maxEmptySlabs is called. The function searches the thickest contiguous empty slabs (further refer to as 'slabs') in along all 3 axis and returns its starting and ending positions in EmptySlabsMin, EmptySlabsMax. Using these parameter, function chooseSplitPlane in listing 4.2 actually decides split axis and split plane. All cells being full or empty is a trivial case for which split plane is meaningless. For the rest of the cases, setting the actual split plane using these empty slabs positions is done in a scheme depicted in fig. 4.2. Still one case is not covered where there is no empty slab, but not all cells are full. This example is illustrated in fig. 4.3, on the right.

An aspect regarding the quality of tree can be mentioned, doubting the 'thickest slab' principle. In fig. 4.3 a, the vertical slabs are thicker than the horizontal ones, so the node will set the split plane at the boundary of the vertical slabs. Yet a split plane dividing the horizontal slabs first could result in a tree of better quality, since by geometry it has a higher chance to be hit by a ray. This idea can

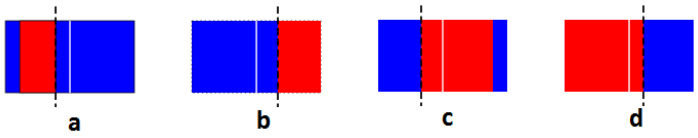

Figure 4.2: 4 possible positions of empty slabs(marked red) Of EmptySlabsMin and EmptySlabsMax, the nearer one to node center(marked white) becomes the split plane(dashed line).
a) EmptySlabsMax is the nearer one.
b) EmptySlabsMin is the nearer one.
EmptySlabsMax is equal to box boundary, hence set right child to BOX_IS_EMPTY.
c) EmptySlabsMin becomes split plane.
d) EmptySlabsMin becomes split plane, set split axis of left child BOX_IS_EMPTY.

be formulated with SAH in sectin 3.2.3. The horizontal slabs have a greater surface area, thus greater chance of traversal. Thus setting this part of sub-tree to a leaf node reduces the cost-function for this node, if the cost-function from SAH can be adopted to volume rendering.

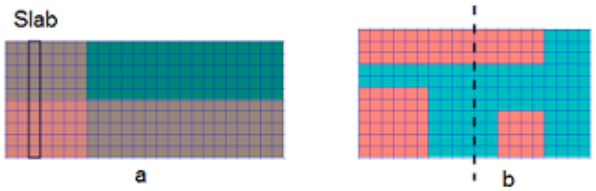

Figure 4.3: In a, sub-tree quality with vertical slabs and horizontal slabs is compared using surface area heuristics from section 3.2.3. In b, when a node has no empty slabs but empty cells, it is devided in the center of its longest axis.

4.2.1 Node Structure

Two possible node structures are implemented. One is to record volume boundary for every each node. The memory costs are high for this method, whereby the advantage is that there is no need to compute the volume while traversing down the tree and while construction. Otherwise if only split axis and split plane is saved, splitting of temporary volume must be applied on every traversal step. While construction, the node for which the split plane is to be computed has to compute its own volume first.

In listing 4.3, when splitDim (split axis) and splitPlane packed into uchar4(splitDim + kdBoxMinDiscrete to uchar4 and splitPlane + kdBoxMaxDiscrete to uchar4), each node takes 8 bytes. splitDim holds split axis for inner nodes and emptyness or fullness information for leaf nodes. splitDim BOX_IS_CELL means the leaf is a grid cell and cannot be divided anymore so is treated as a full node. splitDim BOX_IS_CELL and BOX_IS_FULL together make atomicBoundingBox in listing 4.6.

```
struct kd_node{
   uchar splitDim;
   uchar splitPlane;
```

```
   uchar3 kdBoxMinDiscrete;
   uchar3 kdBoxMaxDiscrete;

   __host__ __device__ void init_node(uint sD, uint sP,
                              uchar3 kdBMinDiscrete,
                              uchar3 kdBMaxDiscrete){
      splitDim = sD;
      splitPlane = sP;
      kdBoxMinDiscrete = kdBMinDiscrete;
      kdBoxMaxDiscrete = kdBMaxDiscrete;
   }

   __host__ __device__ void init_copy(const kd_node& kdN){
                  splitDim = kdN.splitDim;
                  splitPlane = kdN.splitPlane;
                  kdBoxMinDiscrete = kdN.kdBoxMinDiscrete;
                  kdBoxMaxDiscrete = kdN.kdBoxMaxDiscrete;
   }
};
```

Listing 4.3: node structure with box position

```
struct kd_node{
   uchar splitDim;
   uchar splitPlane;

   __host__ __device__ void init_node(uchar sD, uchar sP){
      splitDim = sD;
      splitPlane = sP;
   }

   __host__ __device__ void init_copy(const kd_node& kdN){
                  splitDim = kdN.splitDim;
                  splitPlane = kdN.splitPlane;
   }
};
```

Listing 4.4: compact node structure without box position

Listing 4.4 shows node structure with only split axis and split plane. Summing up to 2bytes per node, significantly saves memory. The drawback is, a box computation must be performed anywhere the application needs the box position. This can be done by using tree index to travel down to the node for which the volume is demanded. At every depth, applying a mask on the tree index will decode if the node belongs to the left or right. Reading the split plane information on each depth, the temporary box is cut to fit to the box of current node.

4.3 Tree Traversal

Rendering kernel, which includes tree traversal is briefly shown in listing 4.5. Rendering is assigns values parallel to each pixel. As many threads as pixels of the application window are created. In render kernel ray for each thread is created and passed to the traverse function. In the traverse function, volume box for leaf node is read or computed according to various traversal method and node structure. Finally, pixel drawing function is called in case of a non-empty leaf volume.

In render_kernel function, the ray is tested if it intersects the volume of the root node. If it passes the test, ray travels and intersects with successive leaf volume boxes. Tree traversal is all about finding the next leaf volume box.

```
render_kernel{
ray_generation(threadidx);
traverse(transfer_parameters, ray, tree_size);
}

traverse{
   box_computation(node);
   drawBox(ray, boxMin, boxMax);
}

drawBox{
   while(inside_box){
      scan_and_alpha_blend(position);
      stride();
   }
}
```

Listing 4.5: simplified render kernel scheme

4.3.1 KD-Restart

As the name suggests, for each inquiry of next volume box kd-restart starts from the root. At every inner node, the algorithm decides to traverse to the left or right node by means of parameterized ray position. The implementation is shown in linsting 4.6. The outer while loop runs over the successive leaf volume boxes. indexRun is the node index, set to 0(root) for every new box computation. The inner while loop traverses down to the next volume box. For a inner node, intersectBoxKdRestart compares the ray parameter position with split plane and returns 1 for left sub-box, 2 for right sub-box. then, drawBox is called if the leaf volume is full. Ray parameter tRun is proceeded to the position where the ray exits the current volume.

```
__device__ void traverse_KD-Restart (const Ray eyeRay, float4 &sum,
                  float tSceneStart, float tSceneEnd, int treeSize){

   const float opacityThreshold = 0.95;
   const float epsilon = 0.0001;
```

```
   float tRun = tSceneStart;

   while (sum.w<opacityThreshold) && (tRun<tSceneEnd-epsilon){

      int indexRun = 0;
      kd_node tempN;
      bool atomicBoundingBox, emptyBoundingBox = false;

      while ( (indexRun < treeSize) &&
            !atomicBoundingBox &&
            !emptyBoundingBox ){

         tempN.init_copy(kd_tree[indexRun]);

         if (BOX_IS_CELL_OR_FULL==tempN.splitDim)
            atomicBoundingBox = true;
         else{
            if (BOX_IS_EMPTY==tempN.splitDim)
               emptyBoundingBox = true;
            else{
               int i = intersectBoxKdRestart(eyeRay, tempN, tRun);
               indexRun = indexRun *2 +i;
            }
         }
      }

      if(!atomicBoundingBox && !emptyBoundingBox)
         tempN.init_copy(kd_tree[indexRun]);

      float3 boxmin, boxmax;
      if(emptyBoundingBox){
         boxDiscreteToContinuous(tempN, boxmin, boxmax);
      }
      else{
         drawBox(eyeRay, sum, boxmin, boxmax);
      }
      proceedRayParam(boxmin, boxmax, tRun);
      /*kdUpTraversal(indexRun, tRun); for kd-backtrack*/
   }
}
```

Listing 4.6: kd-restart

In kd-restart algorithm, frequent execution of intersectBoxKdRestart is distinctive. In this function, ray position is determined through ray parameter. then the ray position is compared with split plane.

This intersection test contains less operations than other traversal algorithms but a better performance cannot be guaranteed since kd-restart invokes far more traversal, thus far more intersection tests.

4.3.2 KD-Backtrack

Another traversal method with reduced traversal steps is the kd-backtrack. After handling a leaf node, it travels the tree upwards in search of the nearest parent that includes the ray position. If such parent is found, the algorithm can continue the down-travel searching the next leaf node. This procedure can be embedded into kd-restart algorithm (position highlighted in listing 4.6). In kdUpTraversal function (see listing 4.7), function tRunInside is called to decide if ray position is inside the current parent node. Up-traversal is continued until this condition is met or root node is reached.

```
__device__ void kdUpTraversal (int& indexRun, float tRun){

   kd_node kdParent;
   moveToParent(indexRun);
   bool tRunPosInsideParent =false;

   while ((indexRun > 0) && !tRunPosInsideParent) {

      kdParent.init_copy(kd_tree_new[indexRun]);

      if tRunInside(tRun, kdParent)
         tRunPosInsideParent = true;
      else
         moveToParent(indexRun);
   }
}
```

Listing 4.7: kdUpTraversal

Number of tRunInside function call and subsequent down-travel continued in listing 4.6 will decide the performance of kd-backtrack.

Using kd-node structure that does not contain the volume box information in listing 4.3, code to find the volume box of the parent must be implemented additionally to kdUpTraversal. This will be explained with fig. 4.4. The ray has intersected node C and handled its volume box. Through kdUpTraversal node B will be retrieved. The volume box must be fit to this Node. But the information needed to expand the volume box is not recorded in node B, so further inquiry upwards must be made. Information to expand box C to box B is recorded in node A.

4.3.3 Kd-tree Stack Traversal

Stack traversal for kd-tree distinguishes from kd-restart only in where the intersection method is called. For kd-tree the stack needs to record only node indexes, that is one uchar. kdLRintersection intersects left and right boxes with the eyeRay and returns code according to whether the ray intersects none of them, only the left node, only the right node, left node first then the right node, or right node first and then

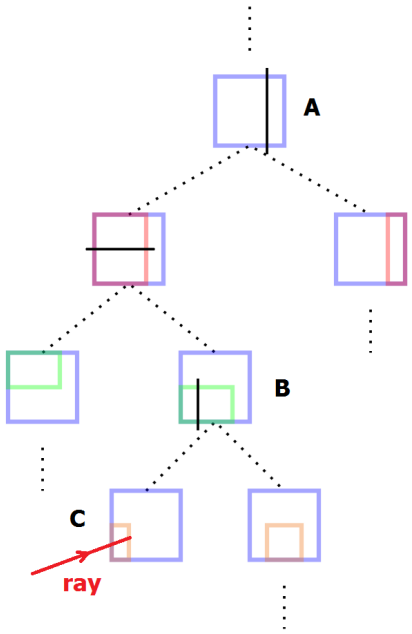

Figure 4.4: shows a box expansion to retrieve the box of the parent node in kdUpTraversal algorithm, a part of kd-backtrack. After ray handled the box C, the parent box is B, but the box information of B is in node A.

the left node. Listing 4.8 shows the stack operations. The intersection method here, kdLRintersection performs 2 box-ray intersections.

```
if(!atomicBoundingBox && !emptyBoundingBox){

   int LRi = kdLRintersection(eyeRay, lNode, rNode);
   if ((ONLY_L == LRi) || (R_THEN_L == LRi)){
      StackPtr->push(LNindex);
      if (R_THEN_L == LRi)
         myStackPtr->push(RNindex);
   }
   if ((ONLY_R == LRi) || (L_THEN_R == LRi)){
      StackPtr->push(RNindex);
      if (L_THEN_R == LRi)
         StackPtr->push(LNindex);
   }
}
```

Listing 4.8: kd-tree stack travesal

4.3.4 Octree Restart Traversal

Restart method used in section 4.3.1 can be used with an alteration to the function which computes the next array index. computeSubboxAndIndex in listing 4.9 adjusts the volume to the next sub-box and returns the sub-box index, according to ray parameter tRun. Unlike intersection test intersectBoxKdRestart for kd-tree restart traversal (listing 4.6), computeSubboxAndIndex has to compare ray position with 3 axis.

```
while ((sum.w < opacityThreshold) && (tRun < tSceneEnd -epsilon)){

   bool leafFound = false;
   int i = 0;

   while( i < octreeDepth && !leaffound ){
      index[i] = computeSubboxAndIndex(boxMin, boxMax, tRun);
      if(BOX_IS_FULL == readOctree(index)){
         drawBox(boxMin, boxMax, sum, tRun);
         leafFound = true;
      }
      if(BOX_IS_Empty == readOctree(index)){
         leafFound = true;
         proceedRayParam(boxMin, boxMax, tRun);
      }
      i++;
   }
}
```

Listing 4.9: octree restart traversal

4.3.5 Octree Stack Traversal

Stack for an octree item works as follows: A stack item saves a space code that codes the current volume box. It is an integer with the i-th digit indicating the sub-box index of depth i. Stack traversal pops an item from the stack and if the item is not a leaf, computes which sub-boxes intersect with the ray, sorts the sub-boxes according to the sequence of being hit by the ray, pushes the sub-boxes onto the stack. Temporary array 'subBoxes' in listing 4.10 for saving the sub-box index has length 4.[1] in computeSubOctBoxes (bottom function in listing 4.10), makeSubBoxAndTest is executed and subBoxes must then be sorted in reference to eyeRay. 5 comparisons suffice for sorting 4 elements. The operations in makeSubBoxAndTest require 8 intersection tests beside sub-box generation. compareAndSwap is relatively costly, too.

```
void octStackTraverse(stack* stackPtr){
```

[1]It can be proved that the maximum number of intersecting sub-box is 4. If there is n axis-cutting plane, every additional plane enables an additional intersection. Thereby maximum number of intersecting sub-volume is n+1.

```
    stackItem tempItem;
    stackItem subBoxes[4];

    while (!(myStackPtr->stackEmpty())){

       myStackPtr->pop(tempItem);

       if (BOX_IS_FULL == readOctree(tempItem))
          drawBox();
       else if (BOX_IS_EMPTY == readOctree(tempItem))
          proceedRayParam();
       else {
          computeSubOctBoxes(tempItem, subBoxes);
          stackPtr->push(subBoxes);
       }
    }
 }

 void computeSubOctBoxes(stackItem tempItem, stackItem* subBoxes, ray eyeRay)

    for(int i = 0; i < 8; i++){
       if(makeSubBoxAndTest(i), eyeRay)
          queueSubBox(tempItem, subBoxes, i);
    }

    compareAndSwap(eyeRay, subBoxes, 0,1);
    compareAndSwap(eyeRay, subBoxes, 2,3);

    compareAndSwap(eyeRay, subBoxes, 0,2);
    compareAndSwap(eyeRay, subBoxes, 1,3);

    compareAndSwap(eyeRay, subBoxes, 1,2);
 }
```

Listing 4.10: octree stack traverse

4.4 KD-Tree Adaptation

Time-dependent data sets show limited change between successive time instances. If each time instance is newly constructed, successive trees will show similarity also in their structure. The adaptation method exploits the information of previous time instance through recycling the tree. A node in the old tree is read by the new tree node with the same tree index. Split axis and split plane of the old node is used as a base with which a new split plane is computed.[2] fig. 4.5 shows an old sub-tree a and the corresponding

[2]The algorithm for split computation, compute_maxEmptySlabs from listing 4.2 is replaced by compute_newSplit.

new sub-tree b. Fig 4.5 a + b shows their relative location in data grid, overlapped. Node a1 and b1 are

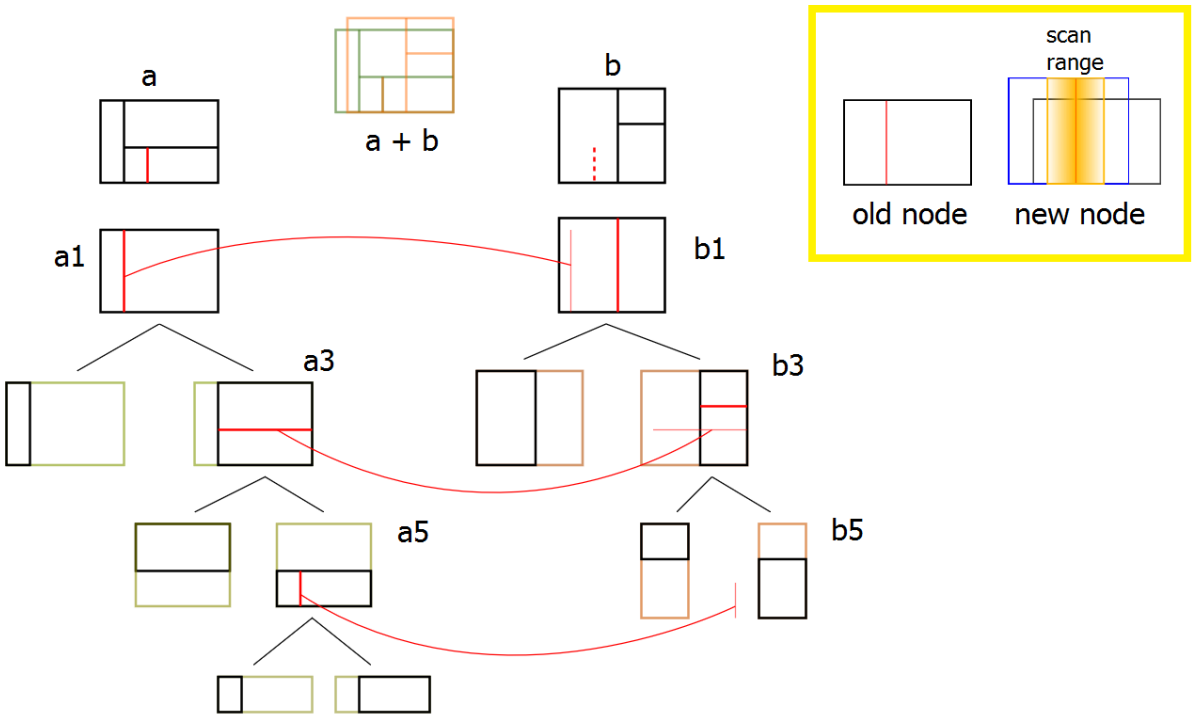

Figure 4.5: a is the old tree, b is the new tree that is being adapted to the old. a + b shows relative position in the grid, overlapped.

reasonably similar, but their sub-trees are less similar. In nodes b1 and b3, a new split plane is found. Whereby in b5, old split plane is positioned outside of the box. In this case, the guess is that the new volume is too far drifted from the old one. Thus the algorithm gives up scanning and sets split axis to BOX_IS_FULL.

The algorithm compute_newSplit, scans only a defined range about the old split plane(see fig.4.5 top right). The volume might have a proper split plane outside of this range, but the algorithm spares effort of searching beyond the scan range.

Not to mention, Search for a new split must consider the context of child nodes. But when both children of old node are inner nodes, scanning is meaningless since the old split plane is not produced by scanning. In this case the old split plane is simply copied, in the expectation of sub-trees still making use of the old tree. The number of nodes for the new tree only drops, because there are inner nodes in the old tree out of which a leaf node is produced.

In this algorithm, If old split axis value holds X_AXIS, Y_AXIS, or Z_AXIS, the new split axis does not change to another one of these 3 values, but either keep the value or change to BOX_IS_EMPTY or BOX_IS_FULL. This precaution is to lessen the number of excessive scanning, because changing the

axis will require a complete scan along an axis, for which there is no information where to start the scan from.

4.4.1 Local Rebuild

As explained in section 4.4, with only Adaptation a degradation of tree for each following time instance is unavoidable. Adapted trees are more coarse, but the existing nodes are correct and optimal in itself. The idea for improvement is to locally re-build the subtrees of these nodes in the same manner as the initial construction.

Conditions to initiate a local re-build is that the split computation in adaptation mode yields a full node that is 'too big' and not really full(explanation for inexactly setting a node as full: in section 4.4). after computing the split plane, the transition to local re-build mode must be marked in node information. otherwise the algorithm computes for the child node falsely in adapt-mode, where the probability of having the old split plane in the own volume is low.

4.5 Adaptive Scan

adaptive scan implemented in this work is simple, it runs without a tree specially generated for adaptive sampling. This stride-skipping algorithm keeps track of old scan values from previous stride, and if the new scan is identical with the old one, ray stride is doubled. This doubling continues up to a predefined maximum. When a newly scanned value is different from the old one, the scan is discarded, the ray retreated to the old position and ray stride reduced in half. When alpha-blending the scanned value while ray stride is greater than 1, alpha-blending must be repeated in a loop. Stride-skipping is used together with empty space skipping.

4.6 Data Dependencies

Flower data (see chapter 5 for data information) is densely packed with 1024 data points per axis. So the grid for emptiness scan is adjusted as to contain 8 data points per axis per cell, yielding 128 cells per axis. This standard of 8 data points is maintained across data sets. Flower data additionally has an overall clear distinction between data-filled area and empty area. Thus the advantages of a deeper tree construction will pay off. An octree of depth 7 has equivalent granularity to the underlying grid of 128 cells per axis. The lambda data also shows a high spatial variation. In order to ensure rendering performance, kd-tree for lambda data is also built with a maximum depth to match the granularity of the grid. For time-dependent data it will show that the tree quality(tree depth) can be traded off against construction time, as long as rendering performance is within acceptable range of interactivity. The ray step size must be also adapted to data specification.

Chapter 5

Results

All results are produced on Geforce GTX TITAN. below are hardware specifications in listing 5.1).

```
Major revision number:         3
Minor revision number:         5
Total global memory:           6441730048
Total shared memory per block: 49152
Total registers per block:     65536
Warp size:                     32
Maximum memory pitch:          2147483647
Maximum threads per block:     1024
Maximum dimension 0 of block:  1024
Maximum dimension 1 of block:  1024
Maximum dimension 2 of block:  64
Maximum dimension 0 of grid:   2147483647
Maximum dimension 1 of grid:   65535
Maximum dimension 2 of grid:   65535
Clock rate:                    875500
Total constant memory:         65536
Texture alignment:             512
Concurrent copy and execution: Yes
Number of multiprocessors:     14
Kernel execution timeout:      Yes
```

Listing 5.1: GTX TITAN memory specification

There are some parameters for volume rendering to mention. For ray step that is added onto ray parameter, data_range/(5×data_points_per_axis) is selected. All data rendered are cubic except for Rayleigh-Taylor, for which performance data is not analyzed in this work. Early ray termination with opacity 0.95 is applied.

Specifications and acknowledgements of Provided data-sets for this work:

Flower:
A microCT scan of a dried flower (leucadendron rubrum)
Data resolution: $1024 \times 1024 \times 1024$
Time steps: 1

Acquired using a Faro Focus 3D laser range scanner by the Visualization and MultiMedia Lab (VMML) at two locations of the University of Zürich (UZH Nord and Irchel campuses) and one of ETH Zürich (Zentrum campus)

Lambda2:
Data resolution: $529 \times 529 \times 529$
Time steps: 80
Temporal development of the vortex cascade, visualized with the λ_2 criterion

Supernova:
Data resolution: $432 \times 432 \times 432$
Time steps: 50
Dr. John Blondin at North Carolina State University through US Department of Energy's SciDAC Institute for Ultrascale Visuaization

Rayleigh-Taylor instability dataset:
Data resolution: $128 \times 128 \times 256$
Time steps: 25
Verena Krupp (Simulation Techniques Scientific Computing, University of Siegen)

5.1 Static Data

As expected in section 4.1.1, octree build-up takes insignificant time(4.6 ms scan time for flower data set). Non-hierarchical grid scan, as needed in prior to kd-tree construction, takes less than this and is assumably negligible. Stack traversal on octree takes 198ms compared to 225.5 ms naive rendering, so the performance gain is marginal. Otree-restart algorithm takes 153.3 ms for rendering, showing the advantage of space skipping.

Kd-tree construction for tree depth 24, the tree depicted in figure 5.1 right, takes 9947 ms. High construction cost is due to the fine grid. But for static data this is not an issue and can be supported by a render performance of 18 ms. Aiming a faster construction though, render performance degrades to 25.4 ms for a kd-tree with depth 18.

For kd-tree traversal, kd-backtrack method shows slight degradation in performance over kd-restart method. Traversing on stack severely worsens rendering performance, from 25 ms for kd-restart to 64 ms. If not otherwise noted, only results with kd-restart method is presented from now on with tree depth 24, at 64 cells per axis, using lambda data.

5.2 Kd-Tree Adaptation

Cost for a complete new construction of kd-tree is high at 487 to 622 ms throughout the data-set. The number of grid cells that has to be read is decisive. compute_maxEmptySlabs function in section 4.2 scans volume size $x \times y \times z$ cells when the box is empty or full, that is only the best case. Worst case is to scan the volume in every axis, performing $3 \times x \times y \times z$ reads. When a box has no empty slab,

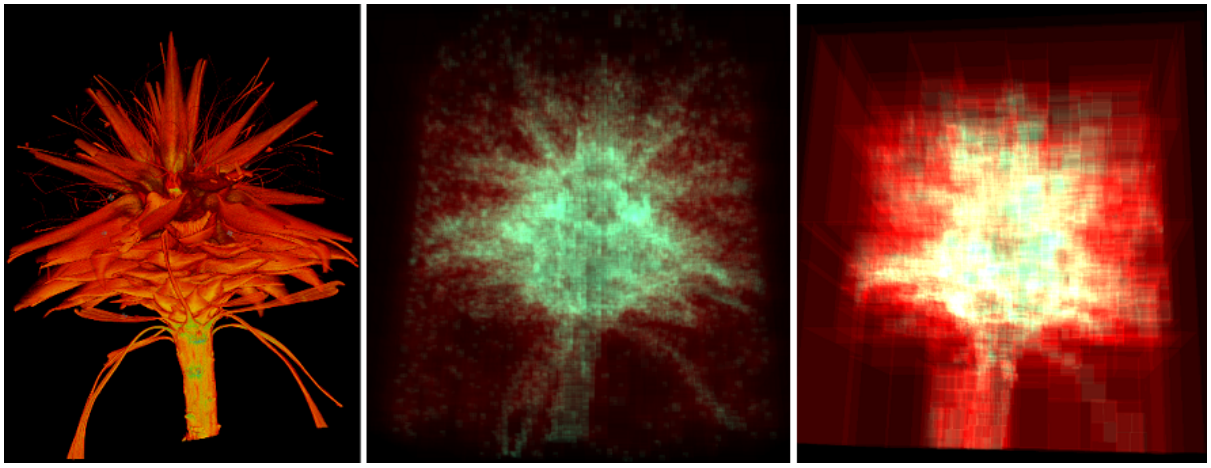

Figure 5.1: On the left is the redering of flower data, in the center is the octree with depth 7 for it(the pixels blend with red color component when intersecting an empty box, with blue-green component when intersecting a full box). On the right is the Kd-tree with depth 24.

but one or more empty cell, the outer loop will run through 3 axis in vain. The average cost is maybe $(3 \times x \times y \times z)/2$ for finding empty slabs while scannig along the second axis, but this is dependent to data. When objects are scattered and well distributed like lambda data (especially true for later time instances), it will approach $(3 \times x \times y \times z)$. When objects are forming handful of dense clusters, $x \times y \times z$ or less scans are enough to decide the split plane (supernova data in figure 5.2 is an extreme example). The scan algorithm can be altered to scan only in the apparently longer axis. Smaller slabs are more probable to be empty, yet it is shown in fig. 4.3 that this might not always build a better tree.

So box adaptation uses the old tree information (old node, since the information in the same node index is taken) and tries to decide the split plane with the least scans. This method is only valid for datasets whos difference between time steps is not great as it is with lambda data. In fig.5.3, change in the data between consecutive time instances (between 25 and 26, between 60 and 61) are barely discernible. Between greater time steps(between 25 and 60), the change is is apparent.

Performance of tree construction with adaptation algorithm is shown in fig. 5.4. Construction takes almost constant time as the algorithm steps through time instance. This is because at deeper depth, many nodes are assigned no volume to scan. As explained in section 4.4, the tree can be trimmed at earlier depth by setting inner nodes as full. This leaves no workload for the kernels responsible for the nodes with greater depth. The nodes at the top of tree cannot fasely be set to a full node though. This accounts for the constancy of adaptation algorithm. Visible in fig. 5.4, new construction cost increases for later time sequences. The lambda data-set gradually expands in space. (see fig. 5.3) As time instance increases the data gets more dispersed and the nodes get bigger. It leads to more cells to scan.

The Tree stepping through time steps using adaptation algorithm degrades quickly. The degradation during 4 time sequences is shown in fig. 5.5. This degradation of tree is caused by setting a node as a full node.

In fig. 5.6, render performance of adaptation is presented. The blue bar shows rendering performance

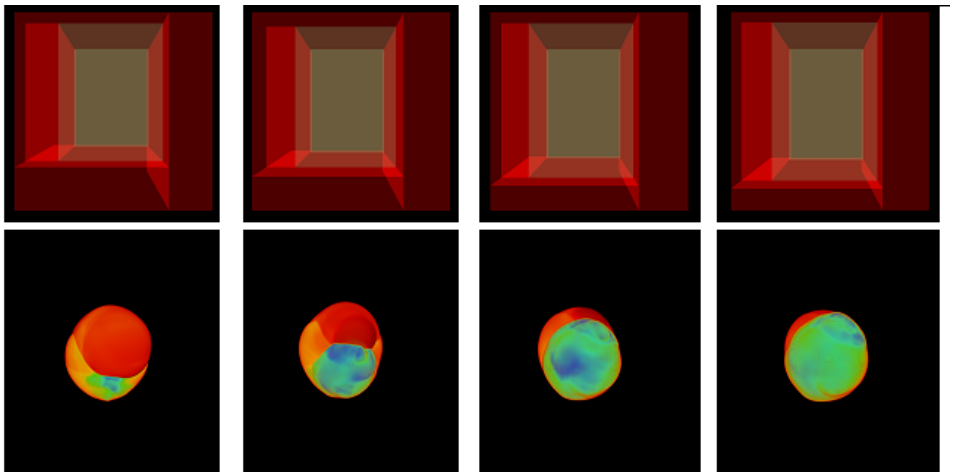

Figure 5.2: Supernova data from time instance 2 - 5. Top row shows kd-tree. Supernova data remains a chunk and therefore sequential adjustments of only one single box is needed, so adaptation without local re-build suffices. for building one box, tree depth of 5 is enough. around 10ms of adaptation time(because tree depth only 5) and up to 50% saving in rendering.

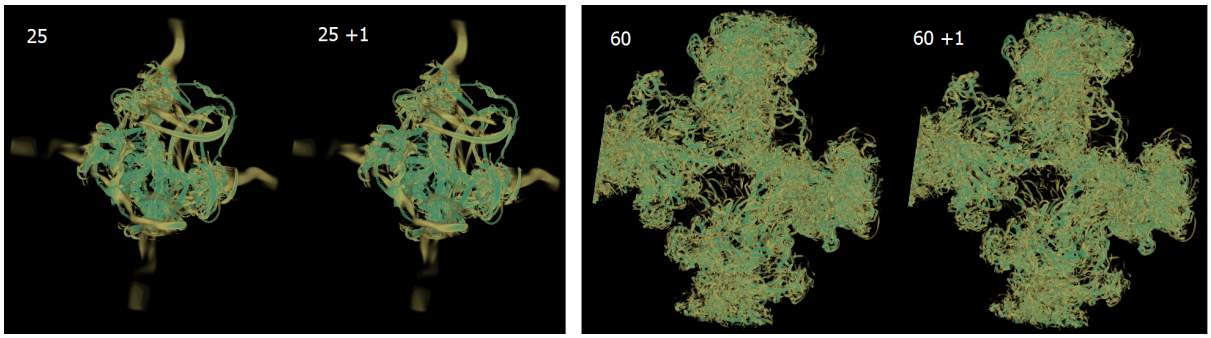

Figure 5.3: lambda data at time instances 25-26 and 60-61. Data-set is only gradually changing.

with for each step newly constructed tree, which is the fastest. The orange bar, render time with adapted tree, grows constantly as the tree degrades. (see fig. 5.5) The gray bar adds construction time and render time together. The time needed to construct and render approaches to naive rendering time, without data structure.

Fig. how often the pixels have sampled the texture data. Each pixel has a different sampling frequency along the ray. In this mode of rendering, Sampling frequency is mapped to monochrome color density of each pixel. The color density is additionally proportional to number of boxes the ray intersects, in order to mimic the traversal cost. In this way, time for render kernel is expected to be proportional to the most dense pixel of each time instance. The top row shows the sampling frequency with degrading tree, the bottom row the frequency with new trees for every time instance. The pixels are getting brighter in the top row, which accords with performance degradation in fig. 5.6 right. (Render time for adaptation is

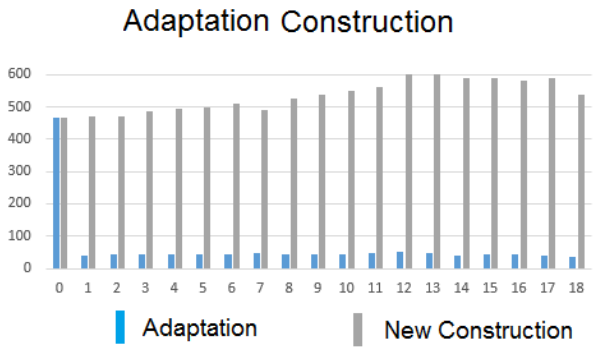

Figure 5.4: Performance of kd-tree construction using adaptation in ms. Tree for lambda data is initially constructed at time instance 25, stepping up to time instance 43.

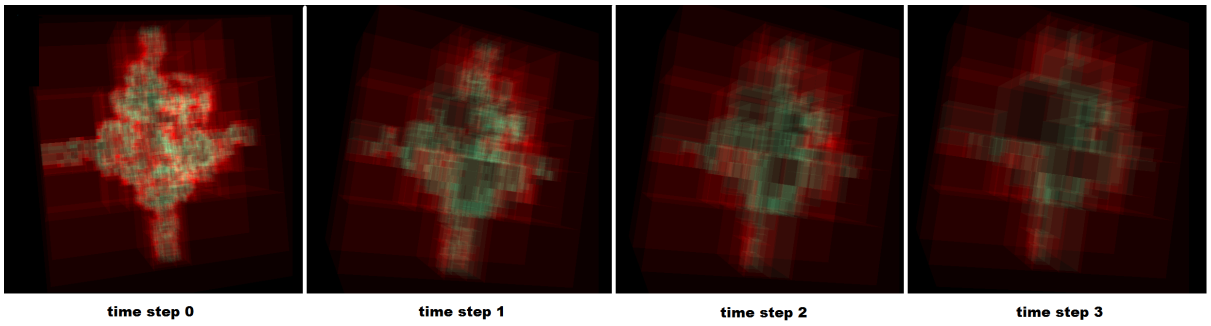

Figure 5.5: kd-tree is initially built for time instance 0. For each following time instances degradation is clearly visible.

increasing) The bottom row shows no apparent difference in brightness between time instances, which can be also approved in fig. 5.6 right.(Render time for new construction is nearly constant)

5.3 kd-tree local rebuild

local re-build method from section 4.4.1 yields good results. In fig. 5.8, a comparison with adaptation method is shown. although additional local re-build means more computation, the amount of time added is marginal. (shown in fig.5.8 left) On the other side the advantages while rendering is unmatched. (see fig.5.8 left) This rendering performance is mostly the same as the performance with new trees.

These results of local re-build are well supported by the quality of trees shown in fig. 5.9. each tree is qualitatively not distinguishable from newly built tree for that time instance.

The reason why local re-build can be executed so fast lies in the size of volume a node in local re-build mode has to process. As discussed in section 5.2, the cost of a full scan of a volume reaches from

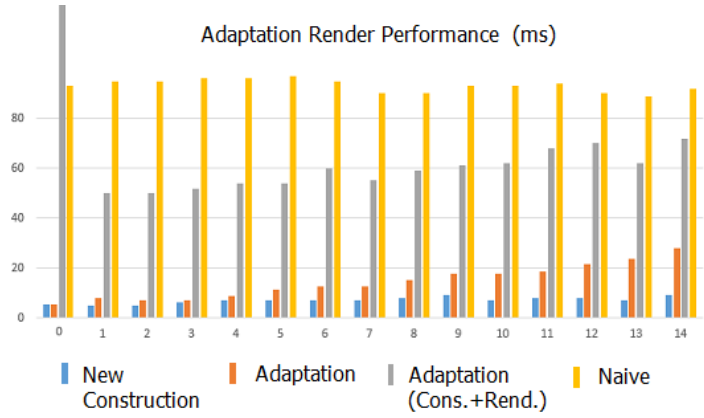

Figure 5.6: Rendering time for lambda data, from time instance 25 on. with different trees. Only the gray bar shows construction time and rendering time added up, the other 3 refer only to render time. The orange bar value for rendering with adapted tree is included in the gray bar value. For adapted tree, the rendering gets slower and gets almost as slow as the naive rendering.

$x \times y \times z$ to $3 \times x \times y \times z$. In the first time step after a new tree is constructed, the nodes for wich local re-build is called lies deeper in the tree, meaning that their volume is small. So the x,y,z values are small, resulting in a much smaller cost. If in that way a tree with an equivalent quality to the newly created follows, every next step needs to peform only a little amount of additional computation.

So far local rebuild greatly improves only adaptation. greater tree depth is in favor of rendering performance and at the cost of construction performance. But the time for construction and rendering together is still not interactive for some time instances. In that case the tree depth dropped. lower tree quality can be backed by enhancement in construction performance. Fig. 5.10 show two sequences of progression, starting from time instance 25 at the left, and at the right side starting from time instance 60. For this tree, tree depth 14 is used instead of 24. the sequence at the left side drops instantly and construction + rendering can be done in almost interactive time. While for a later time instance 60, it needs far more steps to be dropped under the naive rendering time.

Initial construction of time instance 25 yields 3237 nodes. This number slightly increases for a few number of time instances, but on the whole decreases to 2325 nodes at time instance 60, however with visibly greater box sizes. The reason for the bigger box sizes might be the dissipation of data in as time passes, thereby widening the boxes without empty slabs, which raises the call to local re-build function. In any case, it sees that bigger box sizes have greater effect on performance of local re-build than the total number of nodes in the tree.

The advantages of adaptation and local re-build method is shown until now, but there are some data-sets with properties that pose a problem to this method. When an object divides, local re-build is able to keep track of it and to render two separated masses correctly and also to use the space between divided objects to skip correctly. On the other side, when an object emerges in an previously empty space(for such an occurence see fig. 5.11), adaptation and local re-build method is unable to correctly render this change. The space containing the new object will be skipped as an empty space.

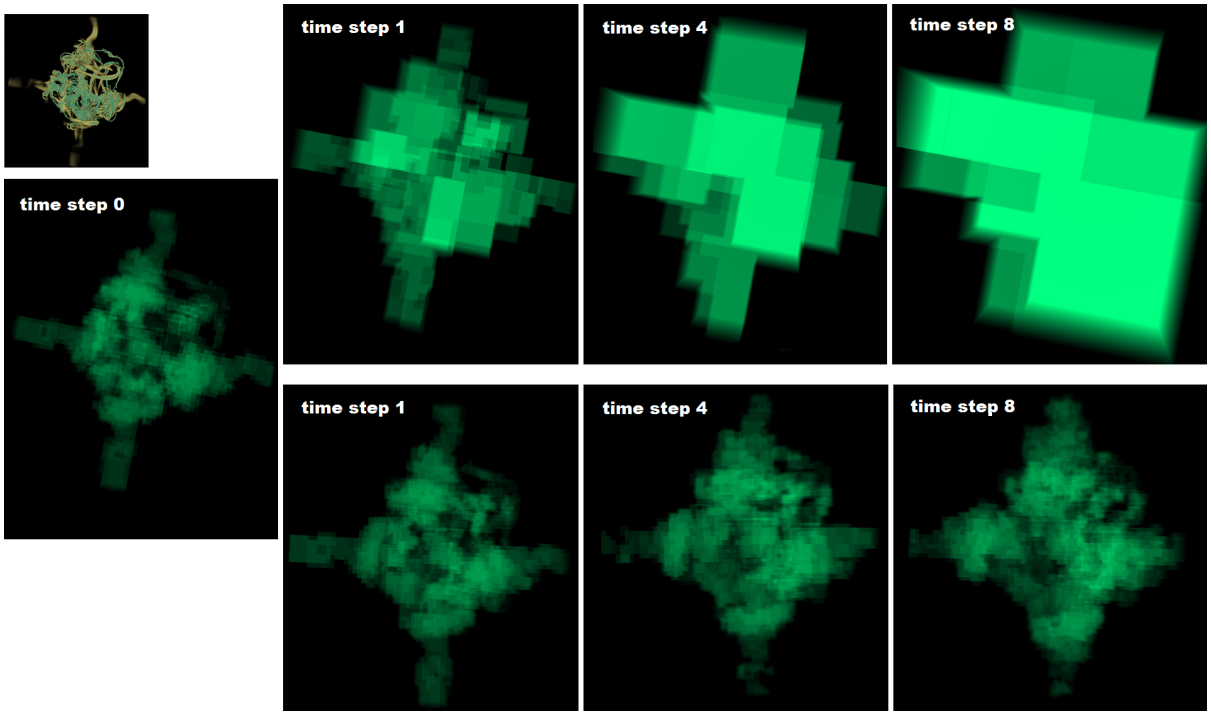

Figure 5.7: lambda data is shown for time instance 25, 26, 29, 33. Starting from time instance 25 and stepping through the time steps. In the upper row using adapted tree, boxes are expanded and the ray has more distance to scan. In the bottom row, new trees for each time instance is used. The more often a ray executes sampling, the brighter is the resulting pixel in these images. Thus these images intend to show the frequency of scan.

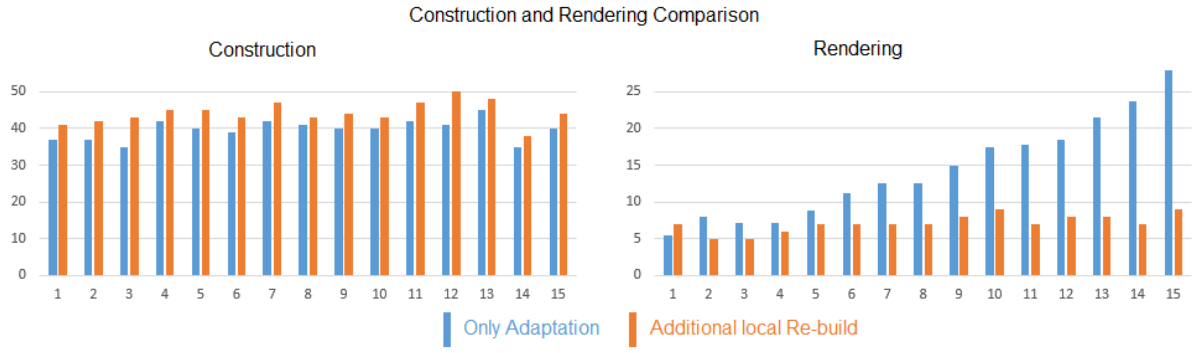

Figure 5.8: local re-build is compared with adaptation method. tree launched at time instance 25, for lambda data.

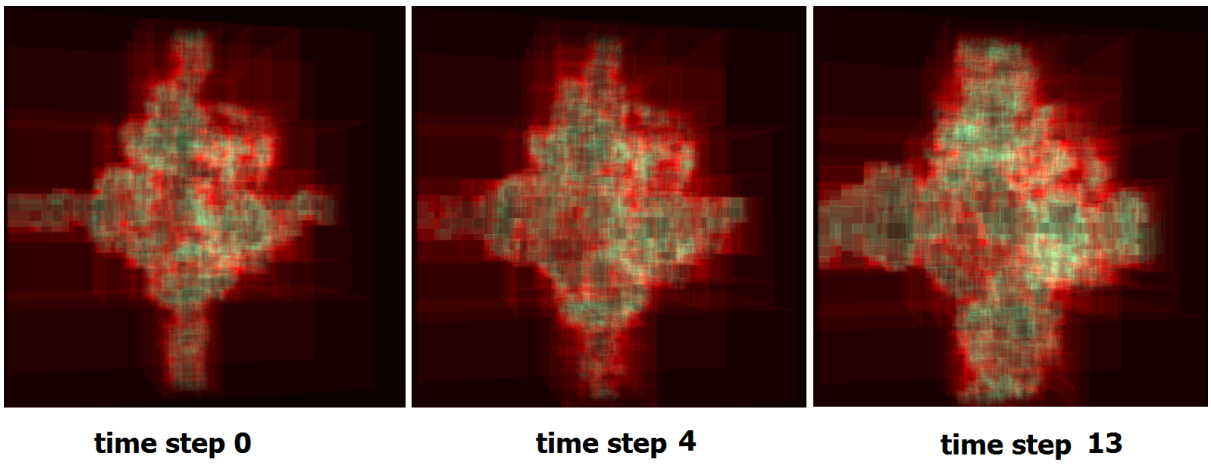

Figure 5.9: Kd-trees built with local re-build method. No degradation in tree quality is visible over 14 time instances. The rendering results approve this in fig. 5.8. The tree at specific time instance shown here has a quality equivalent to the one newly constructed at that time instance, but cannot be the identical tree.

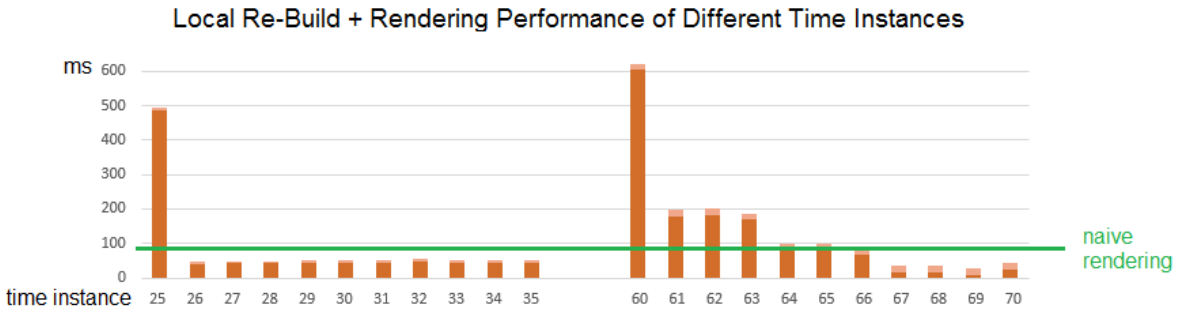

Figure 5.10: Two sequences of local re-build performance, construction plus rendering. drop of kernel duration between 63-64 and 66-67 is data specific. There is less change in the data compared to other time intervals. Note that tree depth 14 is used for data sequence starting at time instance 60.

5.4 Adaptive Scan

Adaptive scan applied for lambda data results in about the same performance like the non-adaptive scan. This is caused by high variation in the data, thus letting the ray stride grow does not mean an advantage. In fig. right, ray leaping through successive empty scan positions easily skips a thin data object, resulting in a artifact contrasted by neighboring ray that did not skip this object. compared with the center image rendered with non-adaptive scan, artifacts are apparent.

As fig. 5.2 shows, the supernova data remains a chunk and when the ray once enters there is no

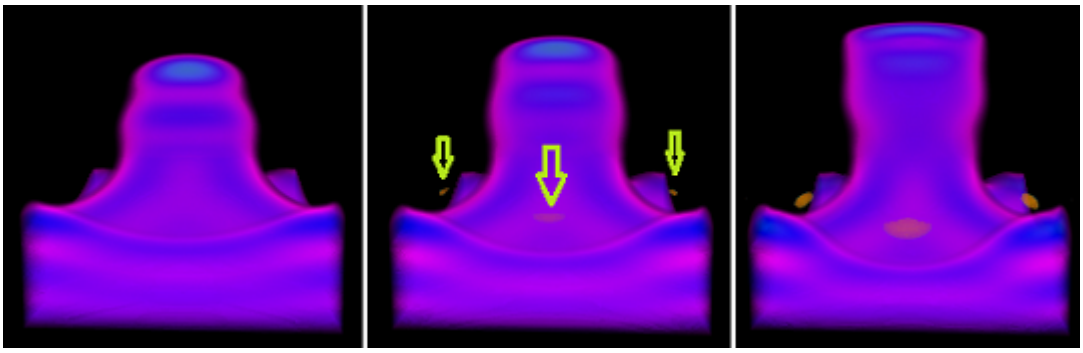

Figure 5.11: 3 time squences of Rayleigh-Taylor data shows that an object unconnected to the previously existing bulk has emerged. Adaptation and local re-build method can neither merge an existing box with a volume including this object nor detect and construct a new volume box for this object.

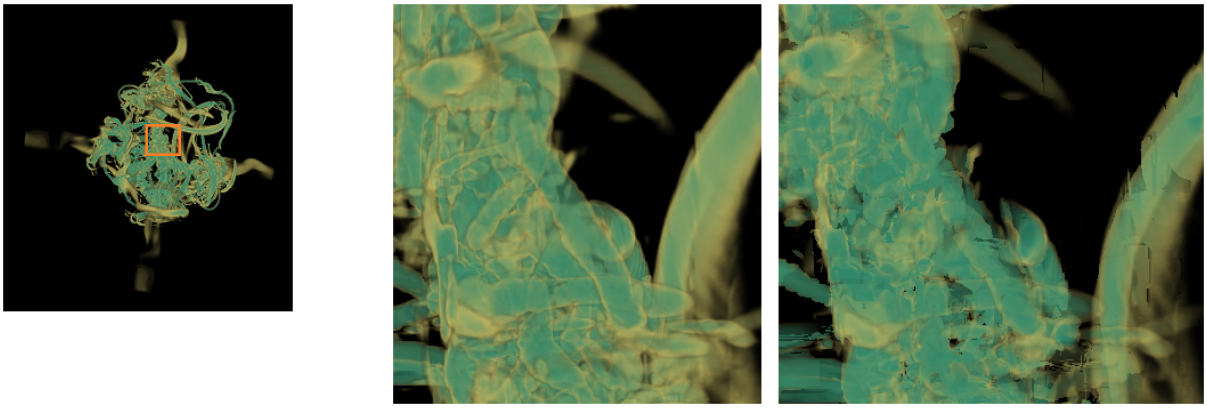

Figure 5.12: left is lambda data-set at time instance 25, with close up area marked. in the center is the normal sampled image, in the right sampled with adaptive scan.

empty space between objects. Therefore adaptive scanning is expected to work without artifacts seen in fig. 5.12. It indeed yields an improvement from 17 to 6 ms with favorable transfer function setting. Rendered image is shown in fig. 5.13, left. In fig. 5.13 right, transfer scale is raised 16% to give more distiction. Density is set set only at 10% of the left image, which allowes the ray to proceed farther into the mass without the opacity getting saturated and thereby terminating too early to look into the mass.

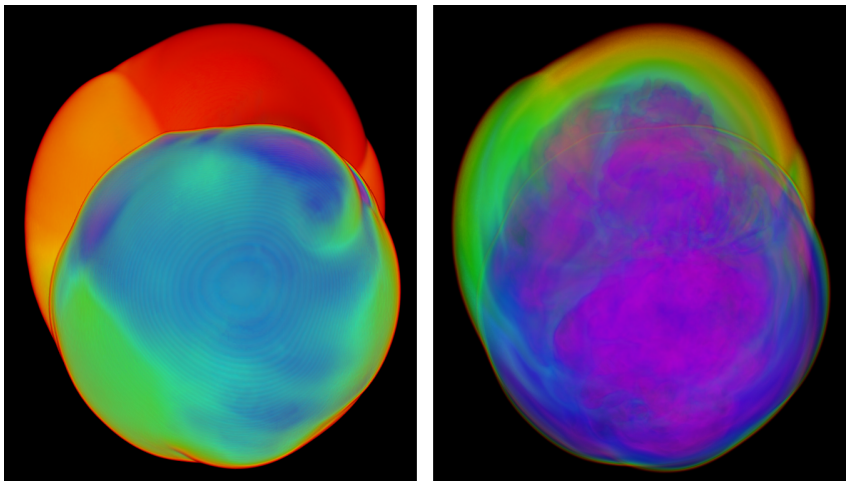

Figure 5.13: Two adaptive scans for supernova at the same time instance, with identical transfer function but different transfer function parameters. In the left, adaptive scan yields performance improvement(6 ms) over non-adaptive scan(17 ms). Homogenous areas are visible, from which adaptive scanning can take advantage. In the right, adaptive scan yields no performance improvement due to high variation. In both cases, adaptive and non-adaptive scan yields no visible difference in rendered image.

Chapter 6

Conclusion

Octrees perform fast construction, but its traversal requires higher cost than kd-tree. This comes mainly from more complicated intersection test while traversal. If Octree or kd-tree, traversal using stack is extremely slow although it is implemented on shared memory.

The main focus of this work, Kd-tree construction for time-dependent data, has greatly profited from local rebuild method. This method adds a marginally low additional cost to adaptation method(see fig. 5.8). Nonetheless the result is a tree of quality equal to a new construction tree(see fig. 5.9. For the most cases it was possible to get construction + render performance in interactive range, even for a high quality tree like the one in lambda, earlier time sequence(see fig. 5.10 left).

User may decide where the focus is, faster rendering or faster overall performance. Dynamic load-balancing of construction and rendering while stepping through time instances could be thinkable, For different time instances may require different tree quality for optimal performance.

A negative point is that initial construction cost is high(see fig. 5.10). This could amongst others come from iteration over tree depth. To reduce kernel-call iteration, standard non-parallel top-down method could be mixed with parallel method.

Adaptive scan can lead to degradation in render quality, so data must be examined for suitability. Applied on supernova data it did show a performance enhancement (see fig. 5.13).

For future works, Packet traversal and global illumination can be mentioned. Packet traversal is aiming a better performance, whereby global illumination is to render a scene with a different lighting model.

Having a tree for empty space skipping, adaptive skipping can be applied on the full nodes. Full nodes are leaf nodes and subtree can be attached easily.

Packet traversal groups rays in a packet. Sampling and computation of volume-local summand to the final pixel can be substituted by a single representative ray. Moreover, the computation of ray traversal uniformity can be reduced for some rays in the packet. This could be done by bundling the packet in a way that the peripherial rays completely surround the center rays. If the rays at the periphery agree on traversal, rays in the center are to follow the same path.

For more photorealism, abandon the emission absorption model in 3.1 and go for global illumination model. In ideal global illumination, for every data sampling, the light falling on this data position must be computed. Correct method to compute this would be to scan along the shadow ray heading toward the light source. This is correct in the sense of not only opacity, but also in the sense of color of the arriving light. As the primary ray travels forth, this tracking of shadows rays must be performed at multiple positions along the ray, which will burden the application enormously. Instead, as a heuristic, summing only the scan distances along the ray can be implemented similarly to 5.7. Change of light color along

the shadow ray is not considered in this heuristic though. Navigation in space with nebulae-surrounded stars can be a good use-scenario of this technique.

Bibliography

[FS97] FREUND, Jason; SLOAN, Kenneth: Accelerated Volume Rendering Using Homogeneous Region Encoding. In: *Proceedings of the 8th Conference on Visualization '97*. Los Alamitos, CA, USA : IEEE Computer Society Press, 1997 (VIS '97). – ISBN 1–58113–011–2, 191–ff.

[FS05] FOLEY, Tim; SUGERMAN, Jeremy: KD-tree Acceleration Structures for a GPU Raytracer. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*. New York, NY, USA : ACM, 2005 (HWWS '05). – ISBN 1–59593–086–8, 15–22

[GS87] GOLDSMITH, J.; SALMON, J.: Automatic Creation of Object Hierarchies for Ray Tracing, 1987. – ISSN 0272–1716, S. 14–20

[HDW+13] HAPALA, Michal; DAVIDOVIČ, Tomáš; WALD, Ingo; HAVRAN, Vlastimil ; SLUSALLEK, Philipp: Efficient Stack-less BVH Traversal for Ray Tracing. In: *Proceedings of the 27th Spring Conference on Computer Graphics*. New York, NY, USA : ACM, 2013 (SCCG '11). – ISBN 978–1–4503–1978–2, 7–12

[HLSR09] HADWIGER, Markus; LJUNG, Patric; SALAMA, Christof ; ROPINSKI, Timo: *Advanced Illumination Techniques for GPU-Based Volume Raycasting*. `http://scivis.itn.liu.se/publications/2009/RHRL09/siggraph09-coursenotes.pdf`. Version: 2009

[HSHH07] HORN, Daniel R.; SUGERMAN, Jeremy; HOUSTON, Mike ; HANRAHAN, Pat: Interactive K-d Tree GPU Raytracing. In: *Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*. New York, NY, USA : ACM, 2007 (I3D '07). – ISBN 978–1–59593–628–8, 167–174

[KW03] KR"UGER, Jens; WESTERMANN, R"udiger: Acceleration techniques for GPU-based volume rendering. In: *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, 2003, S. 38

[Lev90a] LEVOY, Marc: Efficient Ray Tracing of Volume Data. In: *ACM Trans. Graph.* Bd. 9. New York, NY, USA : ACM, Juli 1990. – ISSN 0730–0301, 245–261

[Lev90b] LEVOY, Marc: Volume rendering by adaptive refinement. In: *The Visual Computer* Bd. 6, Springer-Verlag, 1990. – ISSN 0178–2789, 2-7

[LYTM06] LAUTERBACH, C.; YOON, S.-E.; TUFT, D. ; MANOCHA, D.: RT-DEFORM: Interactive Ray Tracing of Dynamic Scenes using BVHs. In: *Interactive Ray Tracing 2006, IEEE Symposium on*, 2006, S. 39–46

[Man06] MANSKE, Magnus: *Octree*. `http://de.wikipedia.org/wiki/Octree#mediaviewer/File:Octree2.png`. Version: Apr 2006

[MB90] MACDONALD, J.David; BOOTH, KelloggS.: Heuristics for ray tracing using space subdivision. In: *The Visual Computer* Bd. 6, Springer-Verlag, 1990. – ISSN 0178–2789, 153-166

[PGSS07] POPOV, S.; G"UNTHER, J.; SEIDEL, H.-P. ; SLUSALLEK, P.: Stackless KD-Tree Traversal for High Performance GPU Ray Tracing. In: *Computer Graphics Forum*, 2007 (26), S. 415 – 424

[PH04] PHARR, Matt; HUMPHREYS, Greg: *Physically based rendering: from theory to implementation*. Amsterdam; Heidelberg [u.a.] : Elsevier, Morgan Kaufmann, 2004. – 131–134 S. `http://www.sciencedirect.com/science/book/9780125531801`. – ISBN 0–12–553180–X

[Tos] TOSAKA: *CUDA*$_p$*rocessing*$_f$*low*.*http://en.wikipedia.org/wiki/File:CUDA_processing_flow_(En).PNG*

WALD, Ingo; BOULOS, Solomon ; SHIRLEY, Peter: Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies. In: *ACM Trans. Graph.* Bd. 26. New York, NY, USA : ACM, Januar 2007. – ISSN 0730–0301

YAGEL, R.; SHI, Z.: Accelerating volume animation by space-leaping. In: *Visualization, 1993. Visualization '93, Proceedings., IEEE Conference on*, 1993, S. 62–69

List of Figures

Erklärung

Hiermit versichere ich, diese Arbeit selbständig verfasst und nur die angegebenen Quellen benutzt zu haben.

_____

(Hajun Jang)