Interactive Dynamic Volume Trees on the GPU

Maik Keller, Nicolas Cuntz, Andreas Kolb

Computer Graphics Group, Institute for Vision and Graphics, University of Siegen
Email: {Maik.Keller,Nicolas.Cuntz,Andreas.Kolb}@uni-siegen.de

Abstract

In many systems hierarchical data structures are used to accelerate the access to spatial data, whereas other applications use a hierarchical volumetric data structure to implicitly represent 3D objects.

We introduce *Dynamic Volume Trees (DVT)*, i.e. an adaptive hierarchical volume data structure which can be modified in real-time. The online capability is achieved by realizing both the hierarchical data structure and the manipulation of the structure solely on the *Graphics Processing Unit (GPU)*. Even though we focus on the representation of highly-detailed volumetric scenes which may be gathered by sensors, e.g. in the fields of robotics and remote sensing applications, DVTs may easily be used to reference different data such as object references. The data is organized in a hierarchical kd-tree-like structure which provides a compact storage of multi-resolution volumes with no redundant memory consumption. Boolean operations are supported, i.e. sub-volumes can be efficiently merged (set union) and removed (set subtraction) with nearly arbitrary resolution. Additionally, a tree optimization is realized in order to improve the performance online. Furthermore, we present two approaches to render the data structure. The power and robustness of DVTs are demonstrated by a multi-resolution volume drawing example.

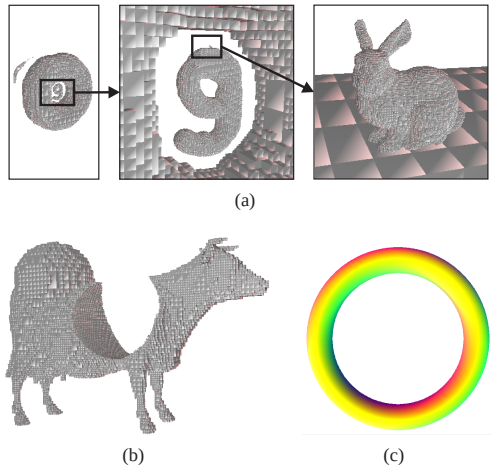


Figure 1: Interactive Dynamic Volume Tree (DVT) containing a multi-resolution drawing entirely managed on the GPU. 1(a) illustrates the tree's high resolution. The objects are drawn interactively in real-time in solid mode (data structure: $2.5M$ nodes, 25 fps with primitives-rendering approach). 1(b) results from boolean operations *merge* and *subtract*. The torus in 1(c) is rendered with the ray casting approach. Its colors are coded with the gradient values ($512x512$,10 fps).

1 Introduction

The use of fast and dynamic data structures is essential in many fields of computer graphics. Algorithms in areas such as ray tracing, collision detection, and volume data processing, for example, require exhaustive memory and computational resources. Thus, this kind of applications organize their data mainly in hierarchical structures for fast and efficient traversals, e.g. for data sampling and manipulation. The development of parallel algorithms becomes increasingly interesting since the trend in todays computing hardware is towards multi-core systems which achieve an enormous speed-up in computation time. This makes the *Graphics Processing Unit (GPU)* with its high computational power of parallelism interesting for handling dynamic data structures. If hierarchy construction methods are parallel, they are scalable on future streaming architectures.

While previous data structures like kd-trees were build by the CPU and finally transferred to the GPU memory for traversal and sampling tasks [6], to-

day's techniques generate even complex data hierarchies directly on the GPU [31]. Thus, a relatively expensive latency for copying data structures is avoided. The topic of GPU-updated sparse and adaptive structures is an area of focus for active research such as ray-tracing [23] and collision detection [7] as well as for representing static objects [3].

In this paper we introduce *Dynamic Volume Trees (DVT)*, a hierarchical data structure in connection with online processing capabilities including the following contributions:

- The main element of DVT is a kd-tree-like hierarchical volume tree which is completely built and managed by the GPU.
- DVT supports GPU-updated operations, i.e. merging and subtraction of sub-volumes with (nearly) arbitrary resolution and at interactive frame-rates.
- Interactive compactification is performed directly on the GPU.
- Regarding the representation of dynamic objects, two rendering approaches are proposed for visualizing DVTs, one is based on a geometry-shader and a "per primitive rendering" for each voxel, whereas the second approach implements a ray casting algorithm.

The development of DVTs aims at the generation of volume models from real sensor data in robotics and remote sensing applications which are still topic of our current research. Therefore, the power and robustness of the data structure is demonstrated by a multi-resolution volume drawing example. It supports simple CSG operations as well as the voxelization of polygon models which are transferred into 3D volumes and then merged into the tree structure within a few milliseconds.

This paper is structured as follows: We start with an overview of the previous work in Sec. 2, followed by a conceptual overview over DVTs (Sec. 3). A detailed description of the DVT-implementation is presented in Sec. 4. Then, in Sec. 5, we present the volume drawing application and the evaluation of the data structure. Finally, Sec. 6 concludes this paper and comments on future work.

2 Background

Data structures such as kd-trees, grids, or bounding volume hierarchies are especially required in areas of high performance algorithms but the relative effectiveness of these structures varies tremendously depending on the application. It is a general problem how to build or update these data structures and it has been extensively discussed in the field of ray tracing, but it is also applicable to other areas such as collision detection [13, 16] and sorting [14].

Any approaches in the context of dynamic data structures can be classified into three categories. The first category is characterized by reconstructing the entire data structure whenever the scene has changed. Shevtsov et al. [27] rebuild a kd-tree from scratch in every frame, for example. Thus, knowledge of the animation path or other constraints is not required. The primary disadvantage of this approach is that it can be expensive to completely rebuild the data structures especially for large scenes and sophisticated structures.

Secondly, if the majority of the scene remains static, a significant coherence exists among subsequent frames. A complete reconstruction of the data structure is substituted just by updating the parts which have changed. Incremental updates have been proposed for *Bounding Volume Hierarchies* (BVHs) [16], for instance. The major drawback of this approach is that its feasibility significantly depends on the kind of changes which are present in the scene, thus the structure tends to degrade after several updates, especially if the structures' overall topology is not updated.

Finally, the last category contains approaches which are applicable for scenes consisting of a large set of (rather) static objects. The idea is to pre-compute an acceleration structure for the static parts and to separate it from the dynamic geometry. An additional top-level acceleration structure might be necessary which includes coordinate-system transformation nodes in order to recompose the entire scene [20, 28].

One of the first approaches proposing a dynamic data structure for ray tracing applications has been presented by Reinhard et al. [25], which allowed the insertion and the deletion of objects in constant time. The structure is based on hierarchi-

cal grids which entirely ran on the CPU. While some methods were examined in order to avoid the performance limiting reconstruction of such structures [20], the first approaches started to use the GPU for graphics hardware accelerated data structures [1] but were still limited due to architectural constraints. Modern approaches run hierarchical acceleration structures such as kd-trees [6, 11, 12] and BVHs [8]. However, these approaches essentially built their structures on the CPU, and do not support any GPU-based updates. The efficient GPU-based data structures proposed by [2] and [19] still use the CPU as a memory manager instance.

In fact, Purcell et al. [24] were the first to describe an entirely GPU-updated, dynamic sparse data structure. Lefohn proposed *Glift* - a powerful library which offers generic data structures for GPUs [18]. Its dynamic and adaptive structures use the CPU only to generate GPU iterators. Dyken et al. [4] build *histogram pyramids* on the GPU and perform a marching cubes algorithm on top of it. Whereas some approaches started to use hierarchical and dynamic data structures on multi-core CPUs [22, 27, 15]. Very recently, first approaches of manipulating hierarchical structures online on the GPU have been presented. [30] build an octree structure to handle point clouds. Zhou et al. [31] construct a new kd-tree for ray tracing per frame, Lauterbach et al. [17] use a BVH for a similar task.

In contrast to this, our approach is able to represent and manage complete hierarchical volumes on the GPU and introduces further processing capacities, e.g. merging and subtraction.

3 Overview of Concept

Our *Dynamic Volume Tree (DVT)* structure is situated completely on the GPU and it is only updated in those regions where changes have been made. The concept of DVTs involves a representation suited for the GPU and a set of operations which allow to modify the structure in an efficient way. In the following, the DVT is introduced by providing a conceptual overview of the method.

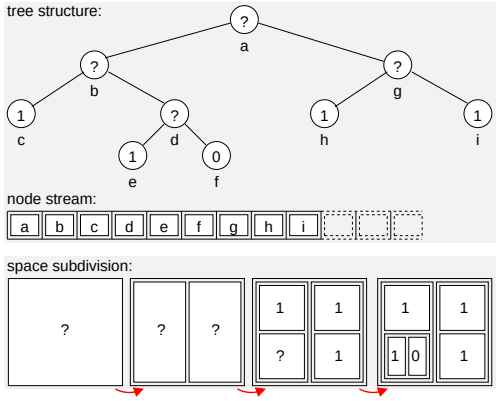


Figure 2: An example (in 2D) consisting of a specific tree and its corresponding spatial representation. The node stream results from a pre-order tree-traversal.

3.1 Tree Topology

The DVT subdivides space similar to a kd-tree. However, the splitting scheme is uniform and axis-aligned: each splitting plane halves the previous block into a pair of two equal-sized sub-blocks. Subdivision is repeatedly performed along the x, y and z-axis, where the depth level of the tree may vary locally. By using a constant bounding box this spatial devision scheme assigns a fixed subspace (voxel) to each node in the binary tree. In principle, this rule can be applied to a spatial subdivision in arbitrary dimensions (see Fig. 2).

As mentioned before, we focus on the representation of geometry. Thus, only the leafs of the tree, which are defined as voxels, contain spatial information and we store values to mark the object's interior (0) and exterior (1) as leaf attributes. To improve the accuracy we can store float values within the range of $[0, 1]$ specifying the distance of the voxel to the respective surface of the object (see Sec. 5.1 for rendering aspects). For simplification reasons, in the following we assume simple binary values to be stored in the leafs.

3.2 Tree Manipulation

The modification of the DVT can be seen as an operation changing the structure and the values in the

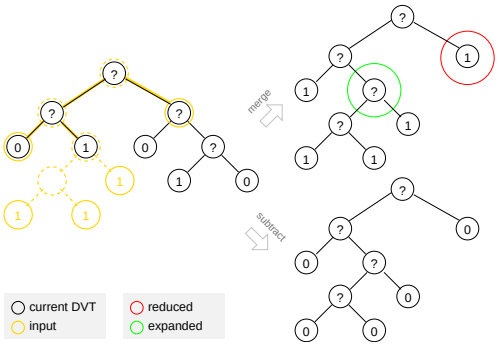


Figure 3: An example showing the result after a merge or subtract operation. The yellow-marked nodes form the input.

tree. Assuming that we have a current tree and an input tree, which may specify an implicit geometry, e.g. a sphere, or a polygonal mesh (see Sec. 4.4 for details on hierarchical geometry rasterization). Now the task is to "insert" the new tree into the current one. We support two boolean operations, i.e.

Merge: The input geometry is merged with the geometry represented in the current tree.

Subtract: The input geometry is subtracted from the geometry represented in the current tree.

Note that processing leafs is sufficient, i.e. both operations can be realized by writing new leafs into the current DVT-structure. Merging and subtracting are realized by reading 1s from the input stream and writing them to the current DVT as 1s and 0s respectively (see Fig. 3.) Furthermore note that the resulting DVT has the same topological structure in both cases, but it may contain redundancies. Then, an optimization pass has been designed which removes redundant subtrees (Sec. 4.3). We do not want to put any restriction on the voxels of the input tree with regard to their locations within the hierarchy which results in different sizes of voxels. This means that the locations in the tree which may be affected by a modification process are scattered arbitrarily. Thus, we face the challenge that, depending on the topology of the current tree, it may be necessary to expand or reduce complete subtrees in order to write the respective input voxels into the tree (see Fig. 3). This is achieved by performing the tree modifications iteratively, namely by adding and removing node levels locally. The iteration terminates, if no modifications are pending and when the node stream of the current DVT is not changing any more. The iteration is split up in three data-parallel passes, the *mark*, the *restructure* and the *remap* pass (see Alg. 3.1). The mark pass selects those nodes within the current DVT which need to be expanded or reduced (see Sec. 4.2.1). The depth of these marked nodes corresponds to the resolution of the respective input voxel. This information needs to be stored together with the input voxels and will later be referred to as the *target depth*. The restructure pass interprets the marked nodes and adds or removes them (see Sec. 4.2.2). Note that both operations handle subtrees despite the fact that one single level is added/removed per iteration. Adding subtrees is realized by marking newly added nodes in the next mark pass; removing subtrees is realized by removing subsequent children during the next restructure pass. Finally, the remap pass ensures a correctly pointered tree structure by resetting all pointers. This involves a two-pass routine using a temporary look-up table (see Sec. 4.2.3). The overall modification process is described in the algorithmic overview given in Alg. 3.1. A more detailed and implementation-driven explanation of the single passes involved in the algorithm, including technical aspects, is given in Sec. 4.

Algorithm 3.1 *(algorithm overview)*

```
1  do
2    // pass 1 (mark):
3    for each input voxel
4      if   appropriate node existent in tree:
5        update value
6      else if   subtree needs to be expanded/reduced:
7        mark node for pass 2
8
9    // pass 2 (restructure):
10   for each marked node n
11     if   n is marked for expansion
12       add children
13     if   (n marked for reduction)
14             or (n has invalid parent)
15       set n's childrens' parent ptrs. invalid
16       remove n
17
18   // pass 3 (remap):
19   for each node n:
20     add node to look−up table
21   for each node n:
22     read pointers from look−up table
23
24 while (∃ marked nodes)
25         or (stream has been altered)
```

4 Implementation

From now on, a GPU specific view of the strategy presented in Sec. 3 is provided, i.e. Alg. 3.1 is explained in detail including all technical aspects which are necessary to realize this idea on the GPU. We have chosen to use a shader API-oriented terminology as our approach fits perfectly to the use of geometry shaders.

4.1 DVT Storage

The tree is stored in three vertex streams in graphics memory. These streams can be propagated through the graphics pipeline in a single render pass. The arrangement of the node information is shown in Fig. 4.

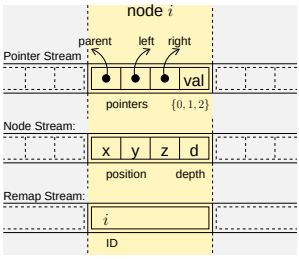


Figure 4: Representation of the tree structure in streams. The cells belonging to one node are marked yellow.

Pointer Stream
This stream contains the pointer structure, i.e. the node's parent and children as node IDs, and the node's value. A pointer can be set to -1, meaning that it is non-existent, e.g. the parent in the case of the root or the children in the case of leaf nodes.

Node Stream
This stream provides a node's spatial position and a depth value implying its level in the hierarchy. This stream information can be used for an efficient rendering without always being forced to determine each node's position.

Remap Stream
The third stream is used as a temporary container used for coding the look-up table necessary when remapping pointers (Sec. 4.2.3).

These streams are stored in `gl_Position`, `gl_TexCord[0]` and `gl_TexCord[1]` respectively. During processing, stream information can be transferred to a texture object and back by using the pixel buffer extension. This allows the algorithm to use stream processing (e.g. with the geometry shader) and rendering for scattering operations. This strategy is used to read the pointer stream through texture-fetching. To achieve a fragment-based process, the texture object is double-buffered, thus being accessible for writing operations. From now on we will assume that all streams can be read and written as streams *or* textures, assuming that the data has been transferred appropriately before processing. The nodes of the binary tree are stored in *pre-order*. This ensures that the root node is located at the beginning of the stream and does not change its location when modifying the tree.

4.2 DVT Modification

The *write* operation is the core of the DVT approach. It solves the problem of adding and removing nodes when merging a new input volume to the tree. As mentioned in Sec. 3, depending on the location of input voxels (e.g. leaf of the input DVT) the tree structure must be expanded in regions where the depth is insufficient, whereas it must be reduced at oversampled regions in order to eliminate superfluous nodes. Alg. 3.1 contains the steps necessary to realize modifications in parallel. We now reformulate the approach to the specifics of the GPU, including a workflow which can be realized by using stages of the graphics pipeline.

The workflow consists of a loop in which three passes are used: *mark*, *restructure* and *remap*. Remember that the input consists of a stream of input voxels together with their target depths. The *mark* pass selects a set of tree nodes which are affected by the input by writing special marker values into the pointer stream. The marked stream is then processed in a geometry shader pass: the restructure pass. Finally, the *remap* pass takes care of correct pointers. The three passes are repeated until nothing has to be changed any longer, i.e. if no node is marked and no node is added or removed.

4.2.1 Mark Pass

The idea of the *mark* pass is to select tree nodes which are affected by a write operation. This selection is realized by a scattering approach which is implemented in a vertex shader. In order to simultaneously read from and write to the pointer stream, a double-buffered access is necessary. As only a subset of the stream nodes is changed, the output as well as the input must be initialized to the current state of the pointer stream. The actual marking is done using a scattering approach implemented in a vertex shader: The tree is traversed for each input voxel until the target depth or a leaf node is reached. The traversal follows a path defined by the voxel's spatial location and node locations. There are three cases where nodes are marked when a voxel, i.e. leaf node of the input stream at a given level ("target level"), is processed:

1. If in the current DVT a leaf node at target depth is reached and the value of this leaf node is different from the value of the voxel, then the voxel's value is written into the leaf node of the current DVT.
2. If in the current DVT a leaf node is reached at an insufficient depth, i.e. the DVT needs to be expanded, then the node is marked.
3. If an intermediate node is reached in the current DVT at the target depth, i.e. the tree needs to be reduced, then the node is marked.

In all other cases the current vertex is clipped by moving it out of the output stream and no value is written. Nodes are marked by subtracting a high number from the node's value. Such nodes are later unmarked by adding the same high number to the value.

4.2.2 The Restructure Pass

The *restructure* pass processes the node stream in a geometry shader. Nodes are added and removed where it is necessary. Fig. 5 shows how the removing and adding of nodes works by means of two classic examples. Both operations use a two-pass strategy: turn pointers to -2 and update them in a subsequent pass.

The root of subtrees which have to be removed in the current DVT is indicated by marked intermediate nodes. Remember that the reduction needs to be performed iteratively, starting from the marked node. Since this node remains in the tree as leaf, its children are made invalid, which is indicated by setting their parent pointers to -2. Invalid nodes can be removed safely in the next restructure pass. By removing these nodes in the next pass, their children are transformed into invalid nodes again. Essentially, this realizes the iteration. In the example given on the left hand side of Fig. 5, left, two levels of nodes need to be removed underneath the marked intermediate node. The iteration stops when no change occurs anymore (after step 1d). Nodes are added at marked leaf nodes which imply insufficient depth. To add the children for this node, the geometry program emits two new vertices which, due to the pre-order sequence, are located directly after the current node. The parent-pointers of the two children nodes can be set directly as the parent is exactly the node processed currently in the shader. On the other hand, the newly added nodes do not possess a valid ID yet since the parallel stream processing does not allow to create a globally unique ID sequence. Thus, the parent pointers of the new nodes are temporarily set to -2. The next *remap* takes care of remapping the pointers and updating those pointers set to -2. This is an easy task because the children are located next to the parent node in the stream (at offsets $+1$ and $+2$).

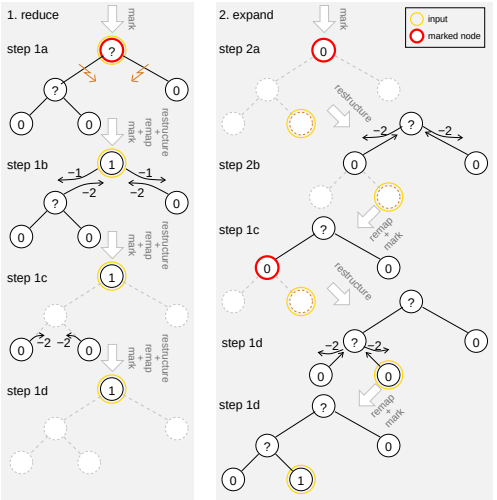


Figure 5: Reduction and expansion of DVT.

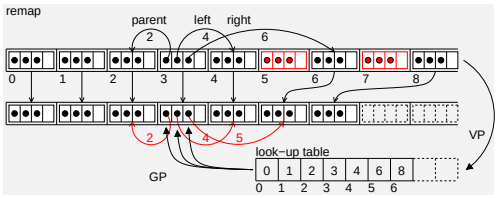


Figure 6: The remap operation. Two nodes (marked red) are removed.

4.2.3 The Remap Pass

Each time the size of the node stream is altered, the pointer structure needs to be updated accordingly. The problem is that moving nodes to other stream locations implies that all pointers to these nodes must be redirected (see Fig. 6) which can be done in parallel using the following scattering-approach. Assume that each node possesses an ID corresponding to its location in the stream. One node moves from an old ID to a new ID when the stream is modified:

1. Render new node IDs as vertices into the remap stream. The output location of each vertex is determined by the old ID.
2. Process the new stream and perform the following operation: update the pointers (parent, left, right) for each node by reading the remap stream.

4.3 Redundancy Optimization

After a write operation on the tree there are cases which can occur where two children or even a whole subtree hold the same value (compare with Fig. 3). These cases can be collapsed without changing the represented volume. Accordingly, the *collapse* operation solves the problem of joining two leaves (at same depth) with equal value. The operator is implemented in a geometry pass which detects superfluous successors. The parent node is set to the value of the successors and the successors themselves are removed. In order to speed-up the shader, a window of four adjacent nodes is processed by using the adjacency capability of OpenGL in connection with line strips. This avoids texture fetch and ensures best caching behaviour.

After *collapse*, a pointer remap is performed according to Sec. 4.2.3.

4.4 DVT generation

In Sec. 3 we assumed that our input geometry is already given as DVT. Even though the rasterization of 3D objects is a research field by its own, we make some remarks on how to generate DVTs for objects.

Voxelization of Polygon Meshes
A simple, yet potentially exhaustive technique is to rasterize an object on a a certain tree level and apply the optimization technique from Sec. 4.3 (see also [5]). Potentially, this requires a huge amount of data and time since the DVT has to be fully instantiated at the predefined level. We use an algorithm similar to [29] which uses clipping planes and front- and back-face rendering to generate slices which are rendered to a 3D texture. The result is a binary solid voxel model, the quality of which is depending on the volume resolution of the 3D texture.

Hierarchical Rasterization
A more effective approach directly works in a hierarchical manner. The hierarchical rasterization provided in Alg. 4.1 creates a temporary node stream containing a hierarchical description of the object to be written into the global structure. The stream is constructed in a loop where each iteration adds one depth level. Thus, the strategy can be seen as a refinement of the hierarchy from coarse (only the root node) to a specific target depth which is given as input parameter. A special marker is used to identify nodes which do not need to be refined again, so-called *finished* nodes. It has to be noted that this algorithm heavily depends on the inside/outside/boundary detection for a given geometry in order to be rasterized hierarchically. For implicit geometries, this functionality can easily be realized using the inherent distance measure, whereas polygonal meshes require more sophisticated techniques.

Algorithm 4.1 *(hierarchical rasterization)*

```
1 initialize a stream with root node of value 1
2 for each depth d ∈ {0, ..., d_target}
3   for each non-finished stream node n
4     if  n is outside the object:
5       discard n
6     else if  n is inside the object, or d = d_target:
7       mark n as finished
```

```
8       else if  n crosses the object's surface
9           subdivide n by adding 2 children to stream
10  process the new stream according to Sec. 3.2
```

Sub-Voxel Accuracy
As mentioned in Sec. 3.1, we may also store float values which specify the distance of the voxel center to the exact object's surface. Computing this distance depends on the type of geometry, e.g. for implicit geometries it is rather straight forward. To keep the memory consumption low, this is applied only to voxels close to the surface, i.e. within a narrow band. This leads to a representation of the object as a signed distance field and allows rendering of the surface at sub-voxel accuracy as shown in Sec. 5.1.

5 Results and Analysis

In this section we demonstrate the application of the DVT by presenting a multi-resolution volume drawing example and a performance analysis. For visualization purpose we propose two rendering approaches.

5.1 Rendering

The following visualization approaches are examples of how DVTs can be traversed in order to render the information of the leaf nodes at interactive frame-rates. Due to the hierarchical structure, both approaches implicitly implement an empty-space skipping similar to [26].

Primitive Rendering
This rendering approach uses the graphics hardware's geometry shader. The node stream is processed by a geometry program which renders a geometric primitive, e.g. a cube, for each voxel which is marked as the object's interior (see Fig. 1(a) and 1(b)). The voxel's spatial representation can easily be calculated by identifying its position in the DVT, i.e. by taking the depth value of the node stream into account. The geometry shader's processing speed is affected by the increasing number of nodes which can be reduced by the redundancy optimization. The approach skips the empty space as the geometric primitives adapt exactly their spatial regions corresponding to the voxel size. Its complexity scales with the number of nodes n which are to process with $O(n)$ rather than with the number of pixels of the output image.

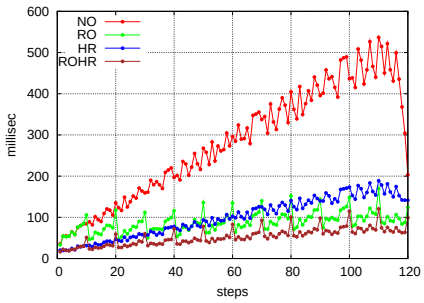
Ray Casting
This approach implements a ray casting algorithm similar to [9] as a fragment program. The DVT is traversed for each pixel of the target image. All the voxels along a viewing ray are traced for further evaluation. If a voxel does not fit the criterion for the object's interior, it is skipped and the next voxel along the ray is processed. The first hit of a voxel of the object's interior represents its surface. Its complexity with $O(p)$ depends on the number of pixels p of the resulting image which initiate the DVT-traversals along the viewing rays.

In Sec. 4.4 we refer to the storage of float values. This feature can be utilized in order to improve the visual appearance of the tree's rendering. The information can be applied to the calculation of shading algorithms, e.g. the computation of gradients at sub-voxel level visibly improves the phong shading (see Fig. 1(c)).

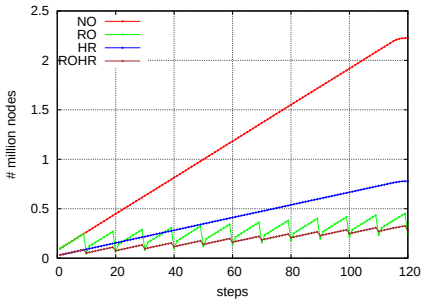
5.2 Volume Drawing

We demonstrate an interactive volume drawing application that stores the drawing in a DVT which is entirely maintained and updated on the GPU (see Fig. 1(a)). The application is well suited to show the performance of the hierarchical data structure. The data can be *drawn* into the DVT with a nearly arbitrary effective draw-resolution that is only constrained by the hardware's floating point precision. Its size is bounded by the GPU memory. The brush mode $b_{\text{mode}} \in \{\text{surface}, \text{solid}\}$, the brush size b_{size} as well as the brush's current voxel size v_{size} are adjustable by the user. v_{size} defines the target depth of the tree where the current drawing is written to. The application catches drawing events at 60fps and processes the data immediately. Thus, the application provides an interactive feedback while drawing in 3D space. Additionally, the application provides an interface for the voxelization of polygonal meshes which can be processed by the DVT.

We maintain highly interactive rates during the drawing process. In particular, the frame rates depend on the number of nodes which are currently to be processed as well as the fill-level of the DVT.



(a)



(b)

Figure 7: Performace-comparison of different strategies using the write and collapse operations (see Sec. 5.3 for legend's abbreviations). A torus is drawn in 120 steps (x-axis). In 7(a) the time for writing the new data into the DVT is measured in milliseconds (y-axis). 7(b) displays the total number of nodes (fill-level) during the drawing.

With regard to the boolean operations *merge* and *subtract*, DVTs can not only add new data to the structure but provide a feature to carve data out. Please see the accompanying video for live demos.

5.3 Performance Analysis

The volume drawing application presented in the previous section has been used to analyze the performance of DVTs. The application is tested on an Intel Dual Core 2.67GHz CPU with a NVIDIA GeForce GTX 280 (1024MB) graphics card. The upper bound for the number of nodes is set to $n_{\max} = 2^{21}$. The test set-up automatically draws a torus with $b_{\text{size}} = 0.04$, $v_{\text{size}} = \frac{1}{512}$ and $b_{\text{mode}} =$ solid in four different ways:

1. Drawing with no redundancy optimization and no hierarchical rasterization (*NO*),
2. drawing with redundancy optimization enabled and no hierarchical rasterization (*RO*),
3. drawing with hierarchical rasterization enabled (*HR*),
4. and drawing with both redundancy optimization and hierarchical rasterization enabled (*ROHR*).

The construction of the torus is done by moving the brush 120 steps in a circular shape with a step-width of 3 degrees (see Fig. 1(c)). The results of the measurements of DVT modifications, i.e. performing write and collapse operations, are displayed in Fig. 7(a) and 7(b). The best performance is achieved by enabling the redundancy optimization as well as the hierarchical rasterization which performs the collapse operation every 10th step as indicated by the respective peaks in the time measurement (see RO and ROHR in Fig. 7(a)). The effect of the redundancy optimization is also illustrated by the decreasing number of nodes after each iteration (see RO and ROHR in Fig. 7(b)). The continuous ups and downs of each plot in Fig. 7(a) are explained easily: the tree does not need to be expanded in certain regions during each write operation as previous write operations may have expanded the tree already at the specific target depth. This results in a better performance and thus in a better time measurement.

The scalability of our highly parallel DVT implementation is examined by disabling processing units of the graphics hardware. Similar to [31] we use the NVStrap-driver in RivaTuner [21] in order to reduce the number of processors of a Geforce 8800 GTX since the NVStrap can not be applied to G90 chipsets of the latest NVIDIA GPUs. The running time of the write operation is scalable to a great extend. However, its scalability is sublinear due to the constant overhead in API management (see Table 1).

5.4 Discussion

Alternative Implementation Technique
Our current DVT-version is implemented using *Shader Model 4 (SM4)*-capabilities of the graphics hardware as described in the previous sections,

#procs	16	32	48	64	80	96	112	128	speed-up
171.751 nodes	474ms	254ms	185ms	149ms	131ms	120ms	113ms	107ms	4.43
1.232.589 nodes	4086ms	2100ms	1444ms	1120ms	938ms	816ms	773ms	669ms	6.11

Table 1: Scalability of the DVT's write operation on a GeForce 8800 GTX graphics card. The last column shows the speed-up of the use from 16 to 128 processors while inserting a certain number of nodes into the DVT.

which are based on the execution of shader programs. We have chosen to take the shader-based approach because its stream processing capabilities (e.g. with the geometry shader) are extensively used in our algorithms. In addition to this, we do not use any shared memory functionality and thus we do not require the latest graphics hardware.

We compared the stream processing capabilities between SM4 and CUDA for performance reasons. The stream expansion and compaction algorithms [10] have been implemented for DVTs and compared to the geometry program usage with its transform feedback feature. In contrast to Dyken et al. [4], who have noticed a slightly better performance of their OpenGL implementation over the CUDA approach, our results show no significant difference.

Comparison
In contrast to former spatial data structures the DVT focuses on a new multi-resolution spatial volume representation. This difference makes it rather difficult to do a direct comparison with well known acceleration structures mostly used in the area of ray tracing without integrating these data structures into the same application. However, Zhou et al. [31] construct a kd-tree on graphics hardware for ray tracing as well as photon mapping for dynamic scenes. The kd-tree is build from scratch for every frame, similar to Lauterbach et al. [17] who rebuild their BVH for each frame. In spite of the similarities of GPU managed spatial data structures, we point out that our approach merges and subtracts complete sub-volumes at interactive frame-rates.

Similar to our kd-tree-like structure, Zhou et al. [30] build an octree-structure in real-time on the GPU to handle point clouds. New points can be inserted into the hierarchy and the respective object surface is reconstructed. Basic boolean operations are also supported by computing implicit functions for the surface. However, the DVTs can handle data in a nearly arbitrary resolution, i.e. target depth, and the values of the data structure do not have to be sorted before processing.

In general, a quantitative comparison of the data structures is difficult due to the different fields of applications.

6 Conclusion

With this paper we have presented a new adaptive hierarchical volume data structure, namely the *Dynamic Volume Tree (DVT)*, which runs entirely on the GPU and can be modified interactively. The kd-tree-like hierarchical structure is completely built and managed by the GPU and supports boolean operations for the merging and subtraction of sub-volumes with nearly arbitrary resolution at interactive frame-rates. We have also presented two rendering approaches. The tree is integrated into a volume drawing application for multi-resolution drawing in real-time.

There are several directions for future investigation. A sophisticated solution for an hierarchical rasterization of polygonal meshes is still pending as the rasterization of an object on a predefined level of detail requires a huge amount of data for the DVT's fully instantiation. We also intend to implement a bricking support for a dynamic allocation and the management of several DVTs in a single application. Future potential is seen for the CUDA implementation of DVTs since the complexity of the program structure decreases significantly. Upcoming architectures such as Larrabee with its high number of cores should exploit the tree's parallel scalability and may be worth investigating further. With regard to applications, we are working on the use of DVTs for the online generation of volume models from real sensor data, e.g. in robotics and remote sensing applications. The application of DVTs for real-time GPU ray tracing could also be an interesting topic for further investigation.

Acknowledgments

This work is partially funded by grant V3DDS001 from the German Federal Ministry of Education and Research (BMBF).

References

[1] N. Carr, J. Hall, and J. Hart. The ray engine, 2002.

[2] G. Coombe, M. Harris, and A. Lastra. Radiosity on graphics hardware. In *Proc. Graphics Interface*, pages 161–168, 2004.

[3] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *Proc. Symp. on Interactive 3D Graphics and Games (I3D)*, 2009.

[4] C. Dyken, G. Ziegler, C. Theobalt, and H.-P. Seidel. High-speed marching cubes using histopyramids. *Comput. Graph. Forum*, 27(8):2028–2039, 2008.

[5] S. Fang and D. Liao. Fast csg voxelization by frame buffer pixel mapping. In *Proc. IEEE Symp. Volume visualization*, pages 43–48, 2000.

[6] T. Foley and J. Sugerman. Kd-tree acceleration structures for a GPU raytracer. In *Proc. Graphics Hardware*, pages 15–22, 2005.

[7] A. Gress, M. Guthe, and R. Klein. GPU-based collision detection for deformable parameterized surfaces. *Computer Graphics Forum*, 25(3):497–506, 2006.

[8] J. Günther, S. Popov, H.-P. Seidel, and P. Slusallek. Realtime ray tracing on GPU with BVH-based packet traversal. In *Proc. Interactive Ray Tracing*, pages 113–118, 2007.

[9] M. Hadwiger, C. Sigg, H. Scharsach, K. Bühler, and M. Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum*, 24(3):303–312, 2005.

[10] M. Harris, S. Sengupta, and J. Owens. Parallel prefix sum (scan) with CUDA. In *GPU Gems 3*. Addison Wesley, 2007.

[11] D. Horn, J. Sugerman, M. Houston, and P. Hanrahan. Interactive k-d tree GPU raytracing. In *Proc. I3D*, pages 167–174, 2007.

[12] E. Hubo, T. Mertens, T. Haber, and P. Bekaert. The quantized kd-tree: Efficient ray tracing of compressed point clouds. *Symp. on Interactive Ray Tracing*, 0:105–113, 2006.

[13] D. James and D. Pai. BD-Tree: Output-sensitive collision detection for reduced deformable models. *ACM Trans. on Graphics*, 23(3), 2004.

[14] D. Knuth. *The art of computer programming, volume 2 (3rd ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing, 1997.

[15] A. Lagae and P. Dutré. Compact, fast and robust grids for ray tracing. In *SIGGRAPH '08: ACM SIGGRAPH 2008 talks*, pages 1–1. ACM, 2008.

[16] T. Larsson and T. Akenine-Möller. A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics*, 30(3):451–460, 2006.

[17] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha. Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2):375–384, 2009.

[18] A. Lefohn. *Glift: generic data structures for graphics hardware*. PhD thesis, Davis, CA, USA, 2006.

[19] A. Lefohn, J. Kniss, C. Hansen, and R. Whitaker. A streaming narrow-band algorithm: interactive computation and visualization of level sets. In *ACM SIGGRAPH Courses Notes*, page 243, 2005.

[20] J. Lext and T. Akenine-Möller. Towards rapid reconstruction for animated ray tracing. In *Proc. EUROGRAPHICS, Short Papers*, 2001.

[21] A. Nicolaychuk, 2009. www.guru3d.com/rivatuner/.

[22] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Experiences with streaming construction of SAH KD-trees. In *Proc. Interactive Ray Tracing*, pages 89–94, 2006.

[23] S. Popov, J. Günther, H.-P. Seidel, and P. Slusallek. Stackless kd-tree traversal for high performance GPU ray tracing. *Computer Graphics Forum*, 26(3):415–424, 2007.

[24] T. Purcell, C. Donner, . Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proc. Graphics Hardware*, pages 41–50, 2003.

[25] E. Reinhard, B. Smits, and C. Hansen. Dynamic acceleration structures for interactive ray tracing. In *Proc. EG Rendering Workshop*, pages 299–306, 2000.

[26] C. Rezk-Salama and A. Kolb. A vertex program for efficient box-plane intersection. In *Proc. Vision, Modeling and Visualization*, pages 115–122, 2005.

[27] M. Shevtsov, A. Soupikov, and A. Kapustin. Highly parallel fast kd-tree construction for interactive ray tracing of dynamic scenes. *Comput. Graph. Forum*, 26(3):395–404, 2007.

[28] I. Wald, C. Benthin, and P. Slusallek. Distributed interactive ray tracing of dynamic scenes. In *Proc. IEEE Symp. on Parallel and Large-Data Visualization and Graphics (PVG)*, 2003.

[29] R. Westermann and T. Ertl. Efficiently using graphics hardware in volume rendering applications. In *Proc. SIGGRAPH '98*, pages 169–177, 1998.

[30] K. Zhou, M. Gong, X. Huang, and B. Guo. Highly parallel surface reconstruction. Research Report MSR-TR-2008-53, Microsoft Research, 2008.

[31] K. Zhou, Q. Hou, R. Wang, and B. Guo. Real-time KD-tree construction on graphics hardware. *ACM Trans. Graph.*, 27(5):1–11, 2008.