



# Durham E-Theses

---

## *Sparse Volumetric Deformation*

WILLCOCKS, CHRISTOPHER,GEORGE

### How to cite:

---

WILLCOCKS, CHRISTOPHER,GEORGE (2013) *Sparse Volumetric Deformation*, Durham theses, Durham University. Available at Durham E-Theses Online: <http://etheses.dur.ac.uk/8471/>

### Use policy

---

The full-text may be used and/or reproduced, and given to third parties in any format or medium, without prior permission or charge, for personal research or study, educational, or not-for-profit purposes provided that:

- a full bibliographic reference is made to the original source
- a [link](#) is made to the metadata record in Durham E-Theses
- the full-text is not changed in any way

The full-text must not be sold in any format or medium without the formal permission of the copyright holders.

Please consult the [full Durham E-Theses policy](#) for further details.

# Sparse Volumetric Deformation

Chris G. Willcocks

A Thesis presented for the degree of  
Doctor of Philosophy



Department of Computer Science  
University of Durham  
England  
April 2013

# Sparse Volumetric Deformation

Chris G. Willcocks

Submitted for the degree of Doctor of Philosophy

April 2013

## Abstract

Volume rendering is becoming increasingly popular as applications require realistic solid shape representations with seamless texture mapping and accurate filtering. However rendering sparse volumetric data is difficult because of the limited memory and processing capabilities of current hardware. To address these limitations, the volumetric information can be stored at progressive resolutions in the hierarchical branches of a tree structure, and sampled according to the region of interest. This means that only a partial region of the full dataset is processed, and therefore massive volumetric scenes can be rendered efficiently.

The problem with this approach is that it currently only supports static scenes. This is because it is difficult to accurately deform massive amounts of volume elements and reconstruct the scene hierarchy in real-time. Another problem is that deformation operations distort the shape where more than one volume element tries to occupy the same location, and similarly gaps occur where deformation stretches the elements further than one discrete location. It is also challenging to efficiently support sophisticated deformations at hierarchical resolutions, such as character skinning or physically based animation. These types of deformation are expensive and require a control structure (for example a cage or skeleton) that maps to a set of features to accelerate the deformation process. The problems with this technique are that the varying volume hierarchy reflects different feature sizes, and manipulating the features at the original resolution is too expensive; therefore the control structure must also hierarchically capture features according to the varying volumetric resolution.

This thesis investigates the area of deforming and rendering massive amounts of dynamic volumetric content. The proposed approach efficiently deforms hierarchical volume elements without introducing artifacts and supports both ray casting and rasterization renderers. This enables light transport to be modeled both accurately and efficiently with applications in the fields of real-time rendering and computer animation. Sophisticated volumetric deformation, including character animation, is also supported in real-time. This is achieved by automatically generating a control skeleton which is mapped to the varying feature resolution of the volume hierarchy. The output deformations are demonstrated in massive dynamic volumetric scenes.

# Declaration

The work in this thesis is based on research carried out in the Department of Computer Sciences, University of Durham, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

**Copyright © 2013 by Chris G. Willcocks**

“The copyright of this thesis rests with the author. No quotations from it should be published without the author’s prior written consent and information derived from it should be acknowledged”.

# Acknowledgements

- (1) I dedicate this work to God who has shown his amazing and forgiving character to me.
- (2) Thanks to my wife, Anna Willcocks, who has wholeheartedly showered me in continuous love, prayers and support making this work possible.
- (3) I would like to express my thanks to Dr Frederick Li, for supervising me during my Ph.D and helping me with the many revisions.
- (4) I would like to thank my parents for their patience over the past three years and for their encouragement during the challenges.
- (5) I would like to express my gratitude to friends and family members who have helped keep my thinking positive even in challenging times.
- (6) The work described in this thesis was funded by a UK EPSRC grant (Project Number: EP/G009635/1)

# Table of Contents

Abstract .....	2
Declaration .....	3
Acknowledgements.....	3
Table of Contents .....	4
List of Figures .....	8
List of Tables.....	9
Acronyms.....	10
Notations.....	10
<b>Chapter 1 Introduction</b> .....	<b>11</b>
1.1 Background.....	11
The Importance of Volume Rendering.....	11
Out-of-core Hierarchical Volumes.....	11
Problems with Hierarchical Rendering.....	12
Thesis Organization .....	12
1.2 Real-time Animation .....	13
Modern Processes.....	13
Modern Abstractions .....	14
Proposed Workflow.....	15
1.3 Volumetric Deformation in Trees.....	15
Input Sparse Voxel Octrees .....	16
Efficiently Updating Hierarchy .....	16
Efficiently Resampling Voxels.....	17
Efficiently Rendering Voxels.....	18
Output Sparse Voxel Octree.....	19
1.4 Supporting Advanced Deformations.....	20
1.5 Chapter Summary.....	21
Motivation.....	21
Challenges .....	21
Objective .....	21
Problems .....	21
Thesis Contributions.....	22
<b>Chapter 2 Literature Review</b> .....	<b>24</b>
2.1 Overview .....	24
2.2 Geometric Modeling .....	25
Parametric Surfaces .....	25
Implicit Surfaces .....	26

2.3	Modeling Deformation.....	27
	Interpolation Modeling .....	28
	Hierarchical Modeling .....	29
	Physically Based Modeling .....	30
	Advanced Modeling .....	31
2.4	Spatial Partitioning.....	32
	Grid Structures .....	32
	Tree Structures.....	33
	Spatial Hashing.....	35
2.5	Real-time Rendering.....	37
	Rasterization.....	37
	Ray Tracing .....	39
	Out-of-Core Algorithm .....	41
	Program Optimization.....	41
	Parallelism .....	44
2.6	Chapter Summary.....	46
	Conclusion .....	46
	Strategy .....	47
	Further Reading.....	47
<b>Chapter 3</b>	<b>Deformation.....</b>	<b>48</b>
3.1	Abstract .....	48
3.2	Methodology.....	48
	Paging .....	51
	Modeling .....	51
	Selection.....	52
	Deformation .....	55
3.3	Resampling.....	58
	Direct .....	58
	Resize.....	61
	Emit .....	63
3.4	Measurement.....	65
	Accuracy .....	65
	Efficiency .....	72
3.5	Conclusion .....	74
	Decision Tree.....	74
3.6	Chapter Summary.....	75
	Contributions.....	75
	Limitations.....	76

<b>Chapter 4 Rendering</b> .....	77
4.1 Abstract .....	77
4.2 Methodology.....	77
Hierarchy Construction .....	78
Adjusted Indices .....	79
Morton Codes.....	80
Sample Level Preservation .....	81
4.3 Ray Casting .....	82
Ray Traversal .....	82
Space Leaping.....	83
4.4 Rasterization.....	84
Mipmap Strategy.....	84
Voxel Appearance .....	85
4.5 Results .....	85
Accuracy .....	85
Efficiency .....	89
Discussion.....	94
4.6 Chapter Summary.....	96
Contributions.....	96
Limitations.....	96
<b>Chapter 5 Skeletonization</b> .....	98
5.1 Abstract .....	98
5.2 Background.....	99
Chapter Structure.....	100
5.3 Related Work.....	101
Propagation .....	101
Contraction.....	102
Other Approaches .....	102
Proposed Strategy .....	103
5.4 Methodology.....	103
Smoothing .....	104
Merging .....	104
Interpolation .....	106
5.5 Pipeline.....	107
Skeletonization.....	107
Termination .....	107
Preprocessing .....	108
Postprocessing .....	109

Parameters .....	109
Summary .....	109
5.6 Applications.....	110
Refinement.....	110
Segmentation .....	111
Skinning .....	112
Solid Voxelization .....	112
Reconstruction .....	112
5.7 Evaluation.....	113
Performance.....	113
Quality .....	114
Comparison .....	117
5.8 Chapter Summary.....	118
Contributions.....	118
Limitations.....	118
<b>Chapter 6 Conclusion.....</b>	<b>119</b>
6.1 Summary .....	119
6.2 Limitations.....	120
6.3 Contributions.....	121
6.4 Applications.....	122
6.5 Future Work .....	123
<b>References .....</b>	<b>124</b>



# List of Figures

Figure 1: The proposed methods in this thesis uniquely deform and animate massive amounts of sparse volumetric data. This image shows the results rendered in real-time.....	13
Figure 2: Fundamental concepts, processes, and abstractions of modern workflows.....	14
Figure 3: Example of SVO hierarchy. ....	16
Figure 4: Artifacts from resampling the voxel center. From left to right: 1. Original shape, 2. Gaps after rotation, 3. Gaps after scale (enlarge), 4. Distortion artifacts after scale (shrink)....	17
Figure 5: Different depths: hierarchical levels of detail in “Atomontage” (Šileš, 2012) .....	18
Figure 6: Detailed SVO rendering in the GigaVoxels pipeline.(Crassin et al., 2009). ....	19
Figure 7: Frame in the accompanying video showing sparse volumetric deformation at 30fps.	23
Figure 8: Samet’s taxonomy: The interrelationship among different data structures, where deeper levels of the tree are more adaptive with flexibility over the region size and shape. ...	34
Figure 9: Example tree layouts in BFS and DFS memory orderings. ....	35
Figure 10: Pointerless quadtree example (Lewiner, Mello, Peixoto, Pesco, & Lopes, 2010).....	36
Figure 11: Abstraction of the basic memory hierarchy for a modern CPU and GPU.....	43
Figure 12: The hierarchy of hardware maps to the hierarchy of content selection. ....	48
Figure 13: Abstraction of the proposed pipeline stages. ....	50
Figure 14: Architecture of the parallel selection, deformation, and resampling processes.....	56
Figure 15: Transforming Axis-Aligned Bounding Boxes (Arvo, 1990).....	59
Figure 16: Deforming an AABB with just two mapped matrices for each extreme.....	60
Figure 17: The resize method changes the display depth of voxels, which implicitly resizes them. Each voxel has exactly $2n$ samples, giving a fast shape approximation.....	61
Figure 18: The emit method ‘emits’ multiple samples in a cuboid shape, giving a good approximation of the deformed AABB, at the expense of memory. ....	63
Figure 19: Human visual perception of error is highly sensitive to the shape silhouette. ....	65
Figure 20: Pixel coverage for $x$ -axis shear deformation. ....	66
Figure 21: The resize and emit resamplings compared to the unaligned direct rasterization. ..	68
Figure 22: Pixel coverage for $y$ -axis twist deformation. ....	68
Figure 23: Error in the resize and emit resamplings, compared to the direct rasterization.....	70
Figure 24: Pixel coverage for scale transformation in all axes.....	71
Figure 25: Resize and emit methods have none/negligible error in transformation. ....	71
Figure 26: Recursion in the proposed pipeline. ....	72
Figure 27: Number of samples emitted for applying shear to a transparent object. ....	73
Figure 28: Number of samples emitted for applying twist to a transparent object. ....	73
Figure 29: Decision tree for the proposed resampling strategies.....	74
Figure 30: Preserved node hierarchy in parallel hierarchy construction.....	79
Figure 31: Pseudocode for 64-bit Morton Codes. ....	80

Figure 32: Parallel rasterization pipeline. ....	84
Figure 33: The deformations are smooth without gaps or shape distortion. ....	86
Figure 34: Comparison between sparse volumetric deformation (upper) and offline high-resolution ray-traced polygon deformation (lower) for the motorcycle in Figure 33. ....	87
Figure 35: Close-up images showing output ray casting of depths 6, 8, and 10 respectively. Just two depths results in much smoother details, seen in the right image. ....	88
Figure 36: Real-time frame from scene in accompanying video, consisting of massive numbers of objects, each with unique twist, shear, scale, translation, or skinning deformations. ....	89
Figure 37: Timing for 1,000 frames with a massive static scene and a fixed camera. ....	90
Figure 38: Timing for 1,000 frames with a massive static scene and a moving camera. ....	91
Figure 39: Timing for 1,000 frames with a massive animated scene and a moving camera. ....	92
Figure 40: A small region of error occurs behind fast-moving objects (black pixels). ....	97
Figure 41: Two feature approximations for a complex model. ....	98
Figure 42: 'Feature-Varying Skeletonization' operates on a wide variety of input cases. ....	100
Figure 43: The proposed contraction process operating on a model of the human hand. ....	103
Figure 44: High-level pseudocode for the complete pipeline. ....	109
Figure 45: Applications: (a) shows the robust handling of a complex model in two different poses. (b) to (e) demonstrate other applications. ....	110
Figure 46: Application showing automatic refinement of multiple end features. ....	111
Figure 47: Comparing methods for common failure cases. ....	116

## List of Tables

Table 1: Selection criteria during top-down hierarchical traversal. ....	53
Table 2: Varying shear in the $x$ -axis. ....	67
Table 3: Varying twist in the $y$ -axis. ....	69
Table 4: Antagonistic properties between smoothing and merging. ....	105
Table 5: This table shows performance and mesh volume in relation to the feature-size $\omega$ . ....	113

# Acronyms

SVO	Sparse Voxel Octree
SVD	Sparse Volumetric Deformation
FVS	Feature-Varying Skeletonization
GPU	Graphics Processing Unit
LOD	Level of Detail
SIMD	Single Instruction, Multiple Data
AABB	Axis-Aligned Bounding Box
OBB	Oriented Bounding Box
ROI	Region of Interest
BVH	Bounding Volume Hierarchy
BSH	Bounding Sphere Hierarchy
DFS	Depth First Search
BFS	Breadth First Search
DOF	Depth of Field
DDA	Digital Differential Analyzer

# Notations

Type	Examples
Integers	Lowercase: $i, j, k, d, h, r, l$
Scalars	Lowercase: $p, s, t, \omega$
Vectors	Lowercase boldface: $\mathbf{p}, \mathbf{q}, \mathbf{b}, \mathbf{c}, \mathbf{t}, \mathbf{s}, \mathbf{m}, \mathbf{v}$
Components in vector $\mathbf{p}$	Lowercase subscript: $\mathbf{p}_x, \mathbf{p}_y, \mathbf{p}_z, \mathbf{p}_w$ also $\mathbf{p}_1, \mathbf{p}_2, \dots, \mathbf{p}_n$
Matrices	Uppercase boldface: $\mathbf{M}, \mathbf{A}$
Containers	Uppercase boldface: $\mathbf{P}, \mathbf{B}, \mathbf{C}, \mathbf{L}, \mathbf{G}, \mathbf{T}, \mathbf{Z}$
Floor of $x$	$\lfloor x \rfloor$
Ceil of $x$	$\lceil x \rceil$
Maximum component in vector $\mathbf{v}$	$\max(\mathbf{v})$
Maximum componentwise vector	$\max(\mathbf{v}, \mathbf{p}, \dots, \mathbf{q})$
Euclidean distance	$\text{dist}(\mathbf{v}, \mathbf{p})$

# Chapter 1

## Introduction

### 1.1 Background

#### The Importance of Volume Rendering

Real-time rendering has recently enjoyed a rapid increase in geometric detail from the latest developments in graphics hardware including general-purpose GPU computing (Gaster, Kaeli, Howes, Mistry, & Schaa, 2011; Owens et al., 2007; Sanders & Kandrot, 2010) and hardware tessellation (Szirmay-Kalos & Umenhoffer, 2008). Conventional polygon rasterization and point splatting pipelines (Akenine-Möller, Haines, & Hoffman, 2008) are popular in industry for processing dynamic scenes; however they are inefficient for representing and filtering complex geometry with features smaller than the size of an individual pixel (Heitz & Neyret, 2012). This is because the hardware rasterizers are optimized for large polygons that cover tens of pixels (Fatahalian et al., 2009) and because many primitives are required to describe complex shapes.

In contrast, direct volume rendering techniques sample shape information of a continuous 3D scalar field typically given on a discretized grid (Engel, 2006). Volume elements in this grid can represent complex feature-rich geometry, and they can be efficiently tested for intersection, enabling applications in real-time ray tracing on current graphics hardware (S. Laine & Karras, 2011). Furthermore, each volume element can easily store descriptors of the shape attributes, enabling accurate filtering without aliasing (Heitz & Neyret, 2012) without requiring expensive oversampling or inaccurate screen-space blurring. However, the volume images consume large amounts of memory, making them incapable of natively representing large sparse scenes.

#### Out-of-core Hierarchical Volumes

To address this problem, recent work efficiently pages volumetric content into main memory at different geometric resolutions in the hierarchical branches of a tree structure, which can be stored out-of-core and compressed. Branches of this tree can therefore be sampled according to the region of interest, and any underrepresented regions can be fetched from secondary storage (Crassin, 2011). The sparse voxel octree (SVO) is a data structure that stores potentially sparse volumetric shapes, as volume elements (voxels), inside the hierarchical branches or leaf nodes of an octree. This partitions the volume such that each branch stores a maximum of eight volume elements, which can be rendered by sampling along propagating rays or cones to calculate the lighting information. By storing the main part of the octree out-of-core, and only fetching and rendering branches according to the region of interest, the memory requirements and GPU bandwidth can be kept low even for massive, complex and sparse geometric scenes.

## Problems with Hierarchical Rendering

The problem of storing volumetric content in a hierarchical tree structure is that currently only static scenes are supported. This is because the hierarchy implicitly determines the geometric location of the volume elements within the scene, which therefore must be resampled with reconstructed hierarchy for any dynamic changes. Resampling and reconstructing hierarchy is an expensive process which is difficult to achieve in real-time for massive amounts of volumetric data; however with the recent advances of parallel construction of tree structures on graphics hardware (Karras, 2012), the approach is worth investigating.

## Thesis Organization

The objective of this thesis is to support deformation in hierarchical volumes, which are stored out-of-core and retrieved according to the region of interest for real-time rendering of massive scenes. This is challenging because, even with an efficient approach for reconstructing volume hierarchy (Karras, 2012), the resampling and deformation operations are costly to compute for each volume element. Therefore an efficient content selection algorithm is introduced, which simulates deformation hierarchically. Approximations are given for the resampling process in Chapter 3, which is achieved without introducing gaps or distortion artifacts in the deformed shape. Chapter 4 then explores a parallel strategy to construct hierarchy only for the important selected volume elements, which enables many real-time applications. This is followed by the demonstration of scalable rasterization and ray casting pipelines. While the proposed pipeline supports animation in massive scenes, more complex deformations are still too expensive to calculate uniquely for the selected high-resolution content. Following a similar approach to polygon techniques, Chapter 5 proposes generating and using a control skeleton to manipulate the volume elements at a feature level. This means that expensive deformation constraints can be broadly applied to groups of elements (bones in the skeleton) instead of being applied to each individual element. However existing skeletonization methods do not capture features at varying geometric resolutions, or for complex shapes. Therefore an automatic *feature-varying skeletonization* algorithm is presented (Willcocks & Li, 2012), which captures the target feature size, and is directly mapped to the hierarchical volume elements for efficient deformation.

To summarize, this thesis enables real-time support for complex deformations in massive sparse volumetric scenes (Figure 1). This is achieved by approximating the sampling at the new volume element locations after selected elements have been simulated (Chapter 3). Where applicable, hierarchy is reconstructed for the selected visible or influential volume elements before rendering (Chapter 4). Expensive deformations are supported by manipulating a control skeleton instead of the individual volume elements. This structure is automatically generated and mapped at a feature level to groups of hierarchical volume elements (Chapter 5).

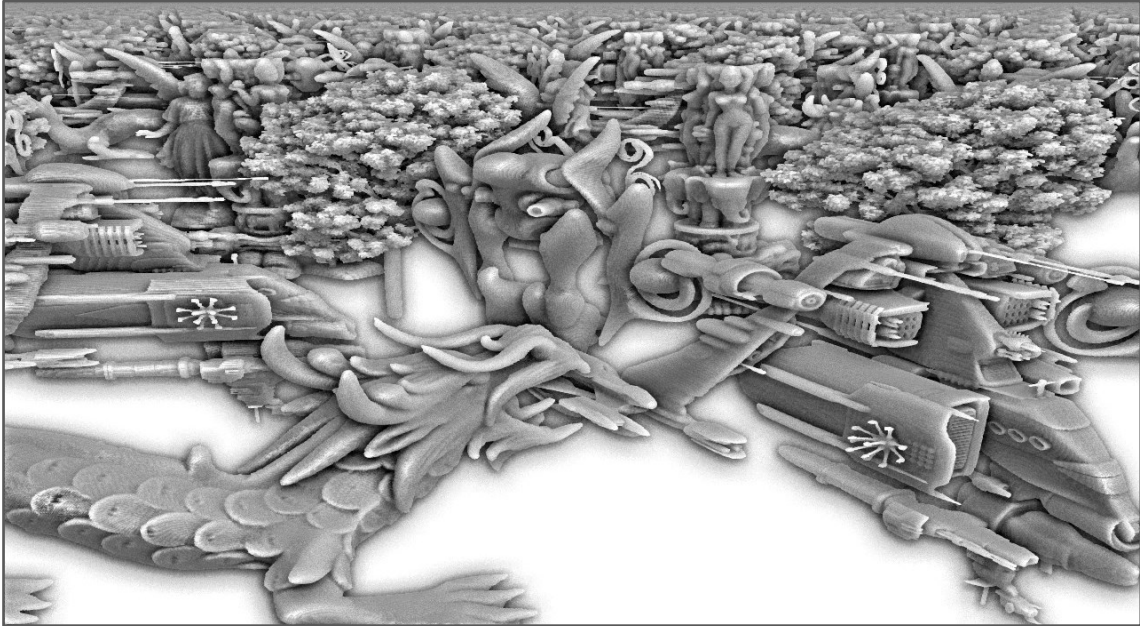


Figure 1: The proposed methods in this thesis uniquely deform and animate massive amounts of sparse volumetric data. This image shows the results rendered in real-time.

## 1.2 Real-time Animation

The purpose of this section is to introduce the fundamental concepts that enable large-scale and appealing computer animations to be calculated in real-time. This generally involves *selecting* important shapes from the scene, *transforming* shapes, and also *deforming* shapes to create new shapes. In particular, the terms *scene transformation* and *object deformation* are used to describe the two main classes of content manipulation. Scene transformation is where entire shapes are transformed in world-space, for example by translation, rotation and scaling. Object deformation describes where features of a shape are locally deformed in object-space to produce a unique effect, such as twist, bend, or shear.

### Modern Processes

In practice, a large animated scene consists of multiple combinations of transformation and deformation operations. In order to manage these operations, modern workflows (Akenine-Möller et al., 2008; Parent, 2012) apply top-down hierarchical matrix multiplications to shapes and shape regions, which are called *compound* or *concatenated* transformations. This strategy means that operations can be prioritized efficiently according to the region of interest, for example by *transforming* each shapes bounding box, and then *selecting* important shapes by frustum culling the transformed bounding boxes. Finally, for any remaining shapes inside the viewing area, more expensive *deformations* such as character skinning can be applied.

## Modern Abstractions

In addition to the top-down hierarchical matrix multiplications, modern workflows introduce further *abstractions* to accelerate real-time animation and provide more intuitive manipulators for the underlying shape information. This section discusses three of the most fundamental modern abstractions: (1) hierarchical organization, (2) instancing and (3) control structures.

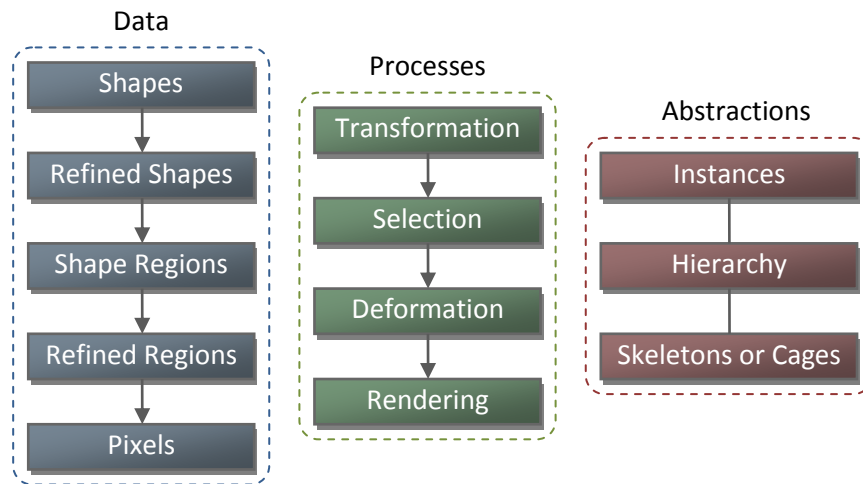


Figure 2: Fundamental concepts, processes, and abstractions of modern workflows.

Figure 2 is a high-level diagram that shows the progression of the underlying shape data (left), as it undergoes the fundamental processes (middle) organized by modern abstractions (right) to achieve real-time animation and rendering or large and interesting scenes. This involves the progressive refinement of scene content to extract important regions for further processing. *Hierarchical* organization involves either storing or referencing shapes in a hierarchical structure, which allows for the *selection* process to efficiently query important information by traversing the hierarchy according to the region of interest. This means that target information can be fetched without expensively processing all the data.

Another important abstraction is *instancing*, which is where the shared attributes, such as geometry or textures, are stored only once, but are referenced and further processed with instanced attributes. In the real-world many shapes are similar and only differ in a few attributes, such as their location, scale, rotation, material, or deformation. *Instances* only store the varying attributes, which means that processing, memory consumption and bandwidth can be greatly reduced. However, even with *instancing* and *hierarchical* organization, complex and realistic deformations are still too expensive to calculate in real-time. To address this problem, modern workflows introduce a simplified geometric abstraction, such as a cage or skeleton. This structure acts on behalf of the shape features and can be manipulated more efficiently

and intuitively. In particular, more expensive constraints, such as kinematic linkages or physical rules, can be applied directly to the simplified structure, and the shape regions can be updated efficiently by a mapping to the structure geometry. Therefore deformation can be modeled as an efficient *interpolation process*, which is more suitable for real-time animation.

## Proposed Workflow

The proposed workflow in Chapter 3 and Chapter 4 utilizes both *instancing* and *hierarchical* abstractions in a fully parallel selection, deformation and rendering pipeline. This operates on high-resolution volumetric shapes by carefully exploiting properties of volumetric content in unity with recent advances in parallelism, discussed more extensively in the literature review. Chapter 5 then introduces an automatic skeletonization algorithm, which generates a mapped *control structure* for intuitive and efficient manipulation of the volumetric shape features.

## 1.3 Volumetric Deformation in Trees

It is important that hierarchy is constructed efficiently for rendering large amounts of dynamic volumetric content. Recently (Karras, 2012) proposes to construct a binary radix tree on data sorted along a space-filling curve to maximize parallelism in the construction of bounding volume hierarchies (BVHs), octrees and  $k$ -d trees. This approach scales linearly to the number of parallel cores and exploits data-locality, meaning that nearby primitives in  $n$ -dimensional space are close together in 1D memory; therefore the cache is well-prepared for fast retrieval.

BVHs construct hierarchy on a set of bounding volumes, such as axis-aligned bounding boxes (AABBs), which contain irregular geometric objects. This requires additional memory to store the extents of the bounding volumes. Similarly  $k$ -d trees are appropriate for storing point primitives, as they partition irregular content by generating hyperplanes that split the space into two parts. The focus of this thesis is on deforming volumes in a continuous 3D scalar field; however in practice, a volumetric field is given on a discretized grid because it is the result of simulation or a measurement (Engel, 2006). The grid itself may be regularly distributed or irregular, although regular distributions lead to more compact representations in memory and fast access to regions, which are extremely important properties for real-time massive scenes. The input is therefore well-suited for a regular tree structure, which organizes the embedding space from which data is drawn into  $N^3$  regions in 3-dimensions (Crassin, Neyret, Lefebvre, & Eisemann, 2009; Lefebvre, Hornus, & Neyret, 2005). Low values for  $N$  consume less memory, but are deeper, where the lowest  $N = 2$  is a special case, in 3D called an *octree* and in 2D a *quadtree*. Choosing a large value of  $N$  is problematic for volumetric deformation, as it means that non-contributing volume elements inside the grid region may get undesirably processed. Therefore the  $2^3$ -tree is a natural choice, as it captures volume regularity with low-memory



consumption. This thesis illustrates concepts in 3-dimensions, therefore an *octree* with  $2^3 = 8$  regions per branch is chosen. A sparse voxel octree (SVO) is the name given to an octree which stores volumetric shapes, as volume elements (voxels), inside the  $2^3$  regions at the branches or leaf nodes of a regular octree organized on the embedding space from which data is drawn.

### Input Sparse Voxel Octrees

Although it is possible to select voxels and maintain scene data within a single input SVO, using multiple input SVOs (aggregated into regions) gives better modeling flexibility. The reasoning is similar to why models are usually stored and paged separately, instead of in a single large scene file: (1) multiple instances of repetitive geometry need not be copied in different regions of a single SVO, (2) multiple SVOs are easier to transform and deform separately, and (3) it's more productive for artists to manage and maintain separate assets for each model.

### Efficiently Updating Hierarchy

With massive amounts of volumetric content stored in separate input SVOs, it is challenging to efficiently deform the voxels in real-time, because any dynamic changes must be resampled with reconstructed hierarchy for the accelerated shape queries required by rendering. The hierarchy of each voxel implicitly determines its location within the scene (Figure 3). This suggests a performance problem with hierarchical deformation, for example transforming the voxel at *a* to location *b* in Figure 3 (lower-right) traditionally requires climbing up the SVO hierarchy to the level *x* containing *a*, and then climbing back down *z* while allocating any new required branches for the new deformed position. This sequential process requires many more operations when compared with the simple per-vertex transformation operations in animated polygon rasterization pipelines.

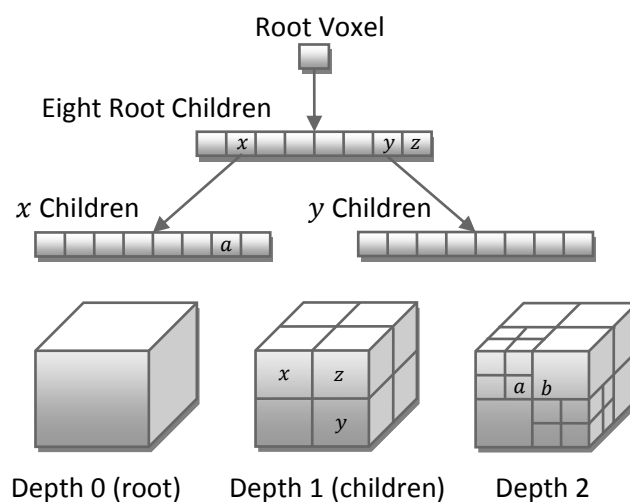


Figure 3: Example of SVO hierarchy.

A more scalable strategy constructs hierarchy in parallel on the GPU (Karras, 2012). This is achieved by calculating the position of each deformed voxel along the z-order curve (which means calculating a 63-bit aligned to 64-bit Morton-code for each voxel by simply interleaving the bits of its 21-bit  $xyz$  components). The locations of the voxels along the curve are then sorted, and any identical adjacent voxels are compacted giving a dense 1D array of the deformed voxels, which are nearby in 3D space. A binary radix tree is constructed on this array, and a parallel prefix sum algorithm is used to identify voxel parents. This in-place hierarchical construction approach is suitable for dynamic volume elements, as it efficiently operates in parallel on the set of deformed voxels, therefore decoupling the selection and deformation process from updating the output SVO allowing data independence.

### Efficiently Resampling Voxels

To enable volume deformation, the new locations of the dynamic voxels must be resampled to the output SVO for rendering. Therefore there are two problems which need to be addressed: (1) deciding how to efficiently transform and deform the input voxels, and (2) deciding how to resample the updated voxel in the output SVO to produce an accurate and efficient rendered image in real-time.

The straightforward approach for these two problems is to transform the voxel bounding-box, and then resample the center of the new box, setting the corresponding voxel in the output SVO. However the rendered result for this approach results in two commonly occurring types of artifact: (1) gaps, and (2) distortion. These are illustrated in Figure 4.

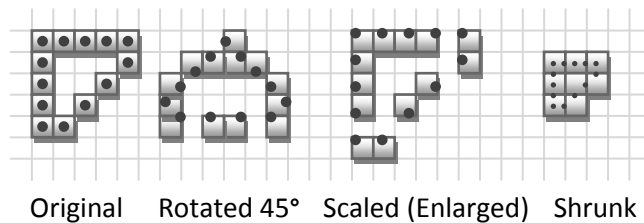
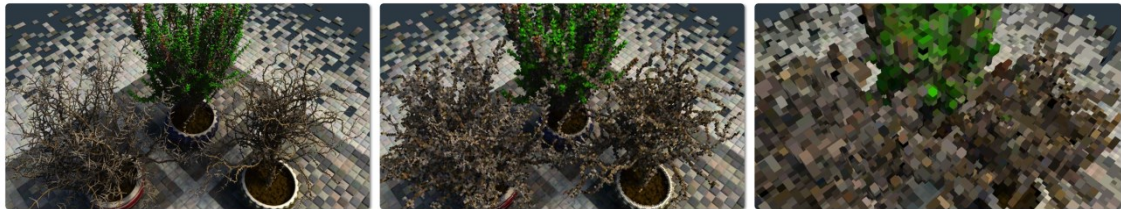


Figure 4: Artifacts from resampling the voxel center. From left to right: 1. Original shape, 2. Gaps after rotation, 3. Gaps after scale (enlarge), 4. Distortion artifacts after scale (shrink).

The 2D illustration in Figure 4 shows a slice of an input SVO (left). Each voxel is transformed and the center is sampled at the same level of hierarchy to produce the output. After a simple rotation transformation, misalignment occurs in the structural regularity (Figure 4 middle-left) creating undesired gaps in the discrete surface. This problem also occurs with scaling (Figure 4 middle-right).

One solution to reduce gaps involves randomly inserting samples inside the transformed voxel. However this is extremely inefficient and does not guarantee that the gaps disappear. Another problem occurs where the transformed voxels are scaled to be smaller than the cell size and multiple voxels occupy the same cell. This causes distortion of the input shape (Figure 4 right). To solve this problem, the size of the output voxels must be changed to shrink the voxels, or in terms of SVO structures, the hierarchical resolution of the output voxels display depth must be increased. Similarly, reducing the resolution of the output voxels implicitly enlarges them, which may therefore be used to close the gap artifacts. The effect of adjusting the rendered level of voxel hierarchy is shown in Figure 5. The left image shows the voxels at the original resolution, then the middle and right images show progressively lower resolutions (higher up the octree hierarchy) which are implicitly larger.



*Figure 5: Different depths: hierarchical levels of detail in “Atomontage” (Sileš, 2012)*

Using this approach to prevent gaps and holes is worth investigating; however a disadvantage is that it causes aliasing as shown on the right image in Figure 5. To reduce the aliasing, the dynamic voxels can be simulated at higher resolution, with improved filtering, however this is more expensive.

The problem of efficiently transforming the input voxels by their mapped data and resampling the output is addressed in detail in Chapter 3. The solution for adjusting the level of hierarchy is determined, as is an approach for more accurately increasing the sample count. These approaches are compared extensively against the unaligned version, which is a rendered result that is not sampled to an SVO (this is achievable by directly rasterizing the dynamic voxels).

### **Efficiently Rendering Voxels**

Chapter 4 explores scalable rasterization and ray casting rendering strategies, and addresses the problem of efficiently reconstructing hierarchy for resamplings of varying display depth. These renderers are both achieved in real-time by the advances in general-purpose computing on graphics hardware. The choice and flexibility of the renderers is important, as it allows more applications: rasterization captures the exact appearance of the deformed shapes, as the updated voxels can be drawn without SVO resampling, whereas the output SVO hierarchy, demonstrated with ray casting, allows for more accurate modeling of light transport.

## Output Sparse Voxel Octree

In the literature, SVOs are intended to accelerate shape queries of volumetric data (S. Laine & Karras, 2011). In order to render an accurate 2D image without aliasing artifacts, geometric information can be sampled from voxels in SVOs according to the Nyquist rate (Crassin, 2011), which is twice the highest frequency of the continuous source signal or 3D data in the scene (Theorem 13 in (Shannon, 1948)). Recently, this has been shown in the GigaVoxels pipeline (Crassin et al., 2009) by sampling a pre-integrated representation of the scene geometry stored in voxels at different hierarchical resolutions. This means that the continuous signal of the 3D scene can be accurately reconstructed for each rendered pixel, without necessarily traversing the volume hierarchy to its full data resolution (this is shown in Figure 6 by three real-time massive detailed scenes that would otherwise be unable to fit into memory). Their approach has been extended further by (Heitz & Neyret, 2012) who additionally account for subpixel occlusion while providing seamless transitions and correct filtering of color, antialiasing, and depth-of-field without requiring expensive oversampling. Therefore, the SVO can be efficiently and accurately used to reconstruct the continuous 3D source signal of shapes for different geometric resolutions, which can be efficiently retrieved from secondary storage according to the region of interest. Furthermore the voxels themselves can store material and color information, which implicitly provides a natural and seamless solution to the texture mapping problem (Benson & Davis, 2002; Watt & Watt, 1992).

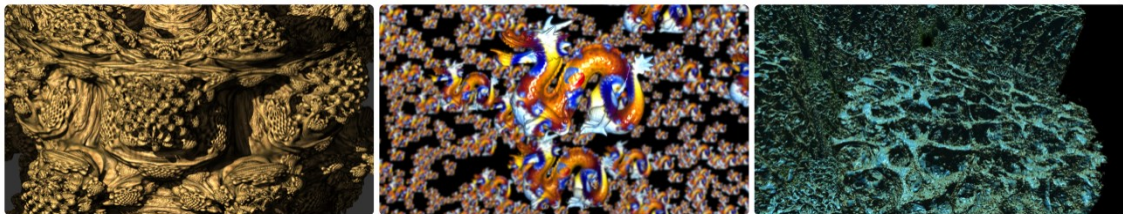


Figure 6: Detailed SVO rendering in the GigaVoxels pipeline.(Crassin et al., 2009).

As the trend for increased detail fidelity continues, so does the strain on application bandwidth and the desire for accurate resolution-independent filtered geometry. The SVO is becoming an increasingly attractive structure for the graphics industry; however without sparse volumetric deformation, volume rendering techniques are severely limited to static or small animated environments. The alternative is to use traditional animated polygon meshes in conjunction with static volume rendering, which can be achieved by either voxelizing a polygon mesh each frame, or by adding a separate polygon layer on top of a static volume renderers. However these inefficient and inconvenient hybrids lose the advantages of volume rendering methods in dynamic content. Furthermore, the real universe is fully *volumetric*, *sparse* and *dynamic*, motivating the graphics industry to use a more realistic and dynamic volumetric data model.

## 1.4 Supporting Advanced Deformations

Modern real-time animation workflows use simplified geometric abstractions to control and manipulate the rendering primitives with respect to features in the input shapes (at Figure 2: *Skeletons<sup>3</sup> or Cages*). This allows for more advanced and expensive deformation operations to be calculated using the simplified structure, such as in character animation, and the respective deformations can be updated efficiently from the structure geometry. There are many types of control structures, but they generally fall into two categories: (1) skeletons, and (2) cages, which try to optimize performance, usability, and accuracy. Examples of control structures include green coordinates (Lipman, Levin, & Cohen-Or, 2008) (cage-based), and eigen-skeletons (Dey, Ranjan, & Wang, 2012) (skeleton-based). Manipulating the respective control structures is much easier, and can be done manually or automatically using techniques such as motion capture, inverse kinematics, key-frame animation, physical simulation, or procedural modeling (Watt & Watt, 1992).

Cage techniques encase the shape with a low resolution mesh or a local coordinate system (for example *free-form deformation*). The idea is that the cage is simple to manipulate with more expensive deformation (which can be nonlinear), and the internal shape is efficiently updated by interpolating between the mapped cage vertices. However cages struggle to capture high-frequency features, such as small details or high genus parts, making them unsuitable for the feature-rich shapes seen in volume rendering. Such high-frequency features can be captured using a hierarchical *skeleton*, which is a thin treelike structure that is located at the shape centerline. Skeletons enforce relative location constraints between the mapped shape features and therefore naturally support hierarchical modeling (such as character animation). While skeletons are smaller and easier to manipulate than cages, they are difficult to generate as the shape centerline is highly sensitive to boundary noise. Also, the resolution of the skeleton needs to reflect the desired size of target features, which may vary in different geometric resolutions, for example with hierarchical volume rendering.

The work in Chapter 5 presents an automatic *feature-varying* skeletonization algorithm which generates a robust skeleton that is mapped to the input model for damaged and noisy inputs. This skeleton can be used directly in the workflow in Figure 2 by using efficient linear blend skinning. With the work by (Kavan, Collins, & O'Sullivan, 2009), high-quality results can be obtained with increased performance compared to the nonlinear alternatives (such as dual quaternion iterative blending) by generating simple virtual blend bones. Manipulating the skeleton therefore allows expensive deformations without greatly impacting the performance.

## 1.5 Chapter Summary

### Motivation

In computer graphics, there is a great motivation to accurately and efficiently model the universe, which is volumetric, sparse, massive, and dynamic. The recent advances in volume rendering take a large step towards this objective, as they are able to render correctly filtered massive scenes consisting of complex and feature-rich shapes in real-time. This is achieved by organizing volumetric content in a tree structure, which is retrieved from secondary storage at varying resolution according to the region of interest. This approach is currently limited to static environments, as it is challenging to make dynamic changes to the volumetric content at interactive frame rates. Existing literature currently addresses this problem by splicing a polygon surface representation with the volumetric content; however this deviates from the realism and accuracy which is to be expected from a purely volumetric model.

### Challenges

Enabling support for dynamic manipulation of the actual volumetric content is challenging to achieve in real-time. This is because of the massive amount of volume elements that need updating each frame, and because of the limitations of current hardware. Furthermore, a successful framework will require careful balance between several topics of computer science, including: (1) computer animation, (2) real-time rendering, (3) hierarchical structures, (4) parallelism, and (5) geometric modeling.

### Objective

The objective of this thesis is to enable real-time support for directly manipulating, deforming, and rendering massive amounts of volumetric content in sparse scenes. This includes designing a simple method for manipulating volumetric content with support for advanced deformations and constraints, for example in character animation.

### Problems

The problem of *efficiency* lies at the source of hierarchical volumetric deformation. Because the positions of the deformed volume elements are implicitly determined by their location within the rendering tree structure, the tree hierarchy needs to be reconstructed each frame and any dynamic changes need to be resampled accordingly. These operations are expensive and very challenging to achieve in real-time for such huge quantities of data. Furthermore, advanced deformations that apply relative constraints to other parts of the shape are computationally expensive, which is especially problematic considering the extreme density of volume elements and the amount of them required to represent a large scene.

The problem of *accuracy* is inevitably encountered when approximations are made to increase performance. This is especially relevant where the volume elements are resampled to the output rendering tree structure. Increasing the number of samples of the deformed content prevents gaps from occurring in large deformations, but greatly decreases the performance. Alternatively, it was discussed to investigate a strategy that changes the hierarchy of the rendered voxels in order to close gaps and prevent shape distortion. This introduces aliasing, which creates an efficiency problem as it can only be accurately reduced with oversampling.

There is also a problem of *accuracy* and *usability* when manipulating volume elements using a control acceleration structure, such as a skeleton or a cage. Hierarchical volume elements capture complex feature-rich shapes at varying feature resolutions, which must be captured correctly by their associated structure. Manually creating these structures and individually mapping them to each hierarchical resolution would be an extremely labor-intensive task, worsened by the topological complexity found in volumetric shapes. Automatically generating and mapping these structures is also problematic: cages are simple to generate, but fail to capture high-frequency features, whereas accurate skeletons are difficult to define as the shape centerline is highly sensitive to noise found on the shape boundary.

## **Thesis Contributions**

This thesis contributes to several areas of computer graphics united under the common theme of enabling support for real-time sparse volumetric deformation in massive scenes. These contributions are discussed in accordance with the thesis structure:

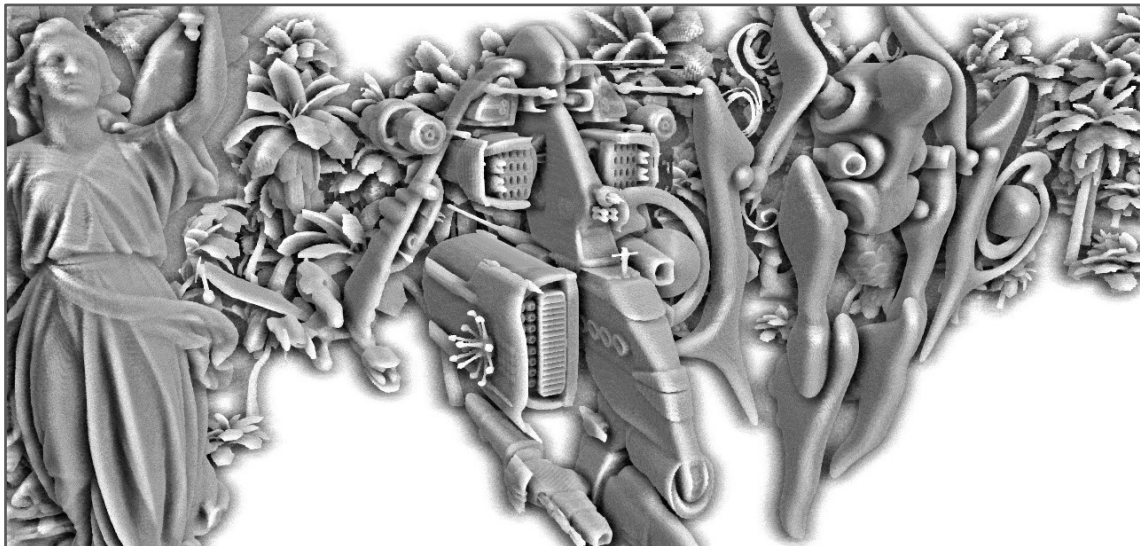
**Chapter 3:** In the field of real-time animation, efficient and high-quality solutions are found for the problem of selecting, deforming, and resampling important volumetric content encoded hierarchically. The selection process operates in a parallel hierarchical algorithm that finds the important shape regions for further simulation. The output can therefore be resampled by determining the correct depth for which to display the output voxels in order to close gaps and prevent shape distortion. This approach means that a fixed number of samples are inserted regardless of the amount of deformation, and is therefore highly efficient and suitable for real-time rendering. Inserting a fixed number of samples of varying rendering resolution inevitably causes aliasing in extreme deformations. In such cases, a secondary method is provided which inserts new samples at the exact locations where gaps in the shape would otherwise occur.

**Chapter 4:** In the field of volume rendering, a parallel framework is presented which addresses the problem of efficiently constructing hierarchy for volumetric elements of varying hierarchy. This means that hierarchy can be constructed for massive sparse scenes at high-resolution,

such as from the deformed and resampled selection in Chapter 3. Additionally, parallelized ray casting and rasterization rendering approaches are implemented, which are scalable and demonstrated in massive animated scenes. These renderers are flexible and share the parallel selection and deformation components. This enables many applications, such hybrid renderers for the real-time modeling of more advanced light transport.

**Chapter 5:** In the field of geometric modeling, the problem of supporting efficient deformation in complex shapes at varying geometric resolution is addressed. A robust and automatic feature-varying skeletonization algorithm is presented, which can operate on complex, noisy, and even damaged inputs. This is particularly useful for non-manifold shapes where traditional homotopic skeletonization methods fail. The algorithm outputs a mapped skeleton structure, and it may process both irregular polygon meshes and regular volumetric content accordingly. To address the efficiency problem of sparse volumetric deformation, the generated skeleton is able to accurately manipulate volumetric content at a feature level instead of per-voxel, and therefore it can efficiently apply expensive constraints, for example in character animation.

In conclusion, manipulating the generated feature-varying skeletons for the model instances (Figure 2) produces appealing animations efficiently. The output can be displayed smoothly in real-time using the proposed resampling and rendering strategies (Figure 7).



*Figure 7: Frame in the accompanying video showing sparse volumetric deformation at 30fps.*



# Chapter 2

## Literature Review

### 2.1 Overview

This chapter discusses a broad range of literature categorized into the four relevant areas for achieving efficient sparse volumetric deformation. These areas are: (1) Geometric Modeling, (2) Modeling Deformation, (3) Spatial Partitioning, and (4) Real-time Rendering. Covering these topics allows for both a wide and thorough analysis of related techniques and data structures that generate *efficient* renderings, measured by the rate of which images are produced, and *accurate* shape representations, measured by the *properties* of the supported geometry, and the *appearance* of the deformed resamplings when compared to the unaligned signals. After this discussion, more literature is presented in the specific areas of the later chapters.

The field of *Geometric Modeling* explores the definition and properties of shape, including both surface and solid modeling techniques, and the method to manipulate and combine operations to express more interesting results. Understanding the properties of shape is therefore important for comparing and analyzing the limitations of different volumetric representations, in particular when targeted at complex models. With an understanding of shape representations, the field of *Modeling Deformation* expands to address how shape is manipulated and deformed in order to create dynamic scenes with appealing animation. This therefore covers both large-scale modeling, such as planetary systems and procedural scenes, alongside small-scale modeling, such as character animation and shape distortion. Managing these deformations in massive scenes is challenging, and requires careful *Spatial Partitioning* such that only relevant or influential information is processed according to the region of interest. The correct choice and usage of a spatial data structure is very important, as can be well-tuned to exploit regularity in volumetric content, and will require efficient construction or update properties in order to be compatible with dynamic scenes. The structure also has a strong relationship with the techniques used to achieve efficient *Real-time Rendering*, which is concerned with producing attractive images that react with little delay from user interaction, allowing for immersion within the dynamic scene.

The method of this thesis is developed in three chapters. Chapter 3 presents the volumetric deformation framework, which details how modeled volumetric shapes can be efficiently and accurately deformed. This therefore relates to literature in the fields of *Geometric Modeling* and *Modeling Deformation*. Chapter 4 extends the method to support efficient rendering of massive scenes by using techniques in the field of *Real-time Rendering*, and through careful

*Spatial Partitioning*. Chapter 5 controls the deformation through an automatically generated control skeleton. This structure is able to efficiently deform volume elements or other shape representations, and relates to fields of *Geometric Modeling* and *Modeling Deformation*.

## 2.2 Geometric Modeling

The word *shape* is used to describe the form or the appearance of an object with respect to its features; therefore shape does not describe the location, scale, or rotation of an object. For example, a square is a rectangular 2D shape with right angle features, whereas a durian fruit has an overall round 3D shape with sharp thorn-like features. In practice, few shapes can be defined with an *explicit* definition (such as a height field) therefore *geometric modeling*, which is concerned with representing shapes (Mortenson, 2006), tends to concentrate on *implicit* or *parametric* definitions. Parametric methods represent the shape surface using a set of piecewise surface patches also called mappings (Ünsalan & Erçil, 2001). This is in contrast to the implicit form, which uses a volumetric definition that naturally captures the interior of an object (Sigg, 2006). Both definitions therefore have different advantages and disadvantages in relation to the primary areas of this thesis, which cover modeling, deformation, and rendering.

### Parametric Surfaces

Representing shape at arbitrary precision is an important goal for computer graphics. The shape surface can be defined by the function:  $f: \Omega \subset \mathbb{R}^2 \rightarrow \mathbb{R}^3$  which maps parametric space  $\Omega$  into 3D space (Sigg, 2006). This means that points in a subset of the 2D space  $(u, v)$  are mapped to 3D space:  $\mathbf{p}(u, v) = (x(u, v), y(u, v), z(u, v))$  creating a surface patch like a single sheet of fabric. Therefore a discrete number of these parametric patches are required in order to cover the entire surface of more complex shapes. The advantages of this mapping are that only a small number of points need to be defined in the parametric domain, which can then be interpolated to form a continuous surface. Popular examples include Bézier surfaces (Farin, 2002) and non-uniform rational B-splines (NURBS) (Piegl & Tiller, 1997), which are compact, easy to construct, and easy to control.

In the past, the graphics pipeline was optimized for rendering a discrete number of triangles (Lindholm, Kligard, & Moreton, 2001) which meant that parametric surfaces had to be converted into a polygon model at discrete levels of detail (F. W. B. Li, 2008). With limited transfer bandwidth, the parametric techniques that dynamically generated geometry on the CPU were generally outperformed by simple methods that swapped pre-computed level of detail meshes (Luebke, 2001). However the latter caused undesirable *popping* artifacts at the transitions between the different resolutions, and consumed valuable video memory for storing the additional levels of detail.

Recently, the graphics pipeline has added support for dynamically emitting more geometry after the input vertices have been processed (Blythe, 2006). This change has enabled new techniques to adaptively tessellate the original geometry, which now implicitly acts as a discrete control mesh, in order to generate smooth surfaces (Loop & Schaefer, 2008). The approach has been enhanced to displace the adaptive surface according to a height field stored on a *parameterized* texture for reproducing fine details without popping artifacts (Szirmay-Kalos & Umenhoffer, 2008). However all of these approaches have shared limitations: (1) the displaced geometry does not support overhanging ledges, (2) only features on the shape surface are captured, and (3) self-intersections are possible as the surface is not guaranteed to be manifold (Sigg, 2006). Advanced deformations may require topology to be split or merged, and therefore a more robust closed-manifold surface definition is required without self-intersections, and with support for solid complex shapes of adaptive geometric resolution. These properties can be achieved with a volumetric definition.

## Implicit Surfaces

Shapes can be defined implicitly with a volumetric definition  $f: \mathbb{R}^3 \rightarrow \mathbb{R}$  that associates a scalar value to points in 3D space. This means that a surface can be described in the implicit form  $f(x, y, z) = 0$  such that points inside the volume are easy to define  $f < 0$  as are points outside the volume  $f > 0$  accordingly. Representing shape using this definition means that complex topological operations, such as merging and splitting, can be achieved by simple operations such as addition and subtraction respectively. Furthermore, complex shapes can be modeled easily by carefully organizing the geometric operations on the volumetric content. Popular examples of these approaches include constructive solid geometry (CSG) (Voelcker & Requicha, 1977), virtual sculpting (Galylean & Hughes, 1991), and procedural generation (Ebert, 2003), all of which can create accurate solid shapes.

Deformation of implicit surfaces can be achieved with two strategies: (1) Proxy Deformation, and (2) Implicit Deformation. *Proxy Deformation* divides the implicit shape into smaller volumes whose geometric deformation can be described by simple transformation. However the deformed outputs are nonplanar which makes efficient intersection tests challenging. *Implicit Deformation* creates a newly sampled scalar field at the deformed location. By discretizing the implicit field as a 3D texture, the approach can be thought of as a *texture space deformation* that distorts the texture mapping implicitly changing the appearance (Engel, 2006). Both proxy and implicit deformation result in a valid visually deformed shape, showing great potential (Esturo, Rössl, & Theisel, 2010), however scalable support for massive scenes has not yet been achieved.

Implicitly defined shapes can be rendered in three ways: (1) Ray tracing, (2) Volume rendering, and (3) Polygonization. *Ray tracing* involves casting rays from the camera and retrieving the closest point of intersection where the isosurface  $f = 0$ , and then generating secondary rays which recursively acquire more lighting information from the scene. *Volume rendering* also casts rays, however samples are accumulated across all isosurfaces to produce a smooth visualization (Engel, 2006). While accurate, these approaches are expensive and therefore the implicit surface is often discretized, whereby the ray traversal process can be accelerated through Spatial Partitioning. *Polygonization*, also known as triangulation, extracts a triangle mesh from the shape surface, where the isosurface  $f = 0$ . This approach was popularized by the marching cubes algorithm (Lorensen & Cline, 1987), followed by improvements to capture sharp features (Ju, Losasso, Schaefer, & Warren, 2002; Schaefer & Warren, 2004), support multiple resolutions (Lengyel, 2010), and even guarantee a manifold shape (Manson & Schaefer, 2010). However accurate isosurface polygonization is too expensive to compute each frame for massive dynamic scenes, and while the polygonized output could be deformed and rasterized easily, it still suffers the limitations of self-intersections, texture parameterization, inaccurate filtering, and aliasing.

In conclusion, implicit surfaces are a more complete representation of shape than popular polygon or parametric approaches; they are rapidly gaining interest in the field of computer graphics. Real-time deformation and ray tracing can be achieved using discretization, but requires careful sampling with spatial partitioning to achieve an accurate result.

## 2.3 Modeling Deformation

Dynamic scenes consist of multiple shapes that transform in location, scale, and rotation. Furthermore, the shapes may change into new shapes in a process called *deformation*. The field of *Modeling Deformation* is concerned with intuitively controlling these dynamic changes to produce appealing new shapes or animations. *Computer Animation* is a large research field which is discussed extensively in the book by (Parent, 2012). It can be generally categorized into two areas: (1) Offline Animation, and (2) Real-time Animation. *Offline Animation* allows artists to manually setup constraints in order to accurately simulate specific visual effects or interactions; this approach is popular in movies (such as *Star Wars* and *Avatar*), computer-generated imagery (CGI such as *Toy Story*, *WALL-E*), cartoons (such as *Mickey Mouse* and *The Lion King*), advertisements, and virtual simulation (such as *hydrological*, *environmental*, *flight*, *medical* and *military* simulations). *Real-time Animation* has crossover with offline animation, in that a pre-prepared database of constraints and animations may be used; however it is more concerned with the cycle of feedback and user interaction with the dynamic images in order to create a sense of immersion within the environment (Akenine-Möller et al., 2008). This will be discussed in more detail later in the section on *Real-time Rendering*; this section is concerned

with *Modeling Deformation*, and therefore the focus is on preparing and manipulating the constraints both intuitively and efficiently to achieve accurate animation.

## Interpolation Modeling

The first animations were achieved by modeling different shapes for each frame and rapidly switching between them. Reconstructing the shape each frame was laborious and unintuitive; therefore the next logical progression was to transform some of the shape attributes, such as its geometry, color, or transparency, across multiple frames to create a smooth transition. However the in-between attributes are unknown, therefore an *interpolation* process is used to estimate new values within the range of the discrete set of known data, by curve-fitting or regression analysis (Monahan, 2011). A popular example of interpolation modeling is *Keyframe Animation* (Burtnyk & Wein, 1971) whereby the animator manually specifies parameters (such as a coordinate of the shape location, the angle of a feature, or a color) at specific animation frames called *keyframes* or *extremes* (Burtnyk & Wein, 1976; Gómez, 1985). The keyframes are then arranged in sequence and the parameters are smoothly interpolated to generate intermediate frames in a process called *in-betweening* or *tweening*. Setting the parameters each keyframe is a laborious task, and may be assisted by offline modeling methods such as by *picking* and *pulling* groups of vertices within some distance of a source vertex (Parent, 2012). However this is impractical and inefficient for modeling feature-rich shapes, and therefore a more a simplified *control structure* is needed to accelerate the modeling process.

*Free-form deformation* (FFD) (Sederberg & Parry, 1986) was the first of a large number of techniques that encase the shape with a simplified *grid* or a *cage*. Deformation is modeled by distorting the local coordinate system, whose vertices are used to map the internal shape. The internal shape is therefore updated by *interpolating* the location of the mapped vertices, which is easier and more efficient than directly manipulating the high-resolution geometry. However this approach is limited by the topology of the control structure; shapes with nearby features or complex topologies (dense, high-genus) require a high-resolution control structure, which introduces a circular problem as the high-resolution control structure itself is difficult and less efficient to manipulate.

*Shape interpolation* is another large research area which looks at merging two parent shapes to create an appealing child with the blended parameters of both parents. This is particularly challenging because it is difficult to find a meaningful mapping which accounts for the varying parameter distributions (such as topology and geometric distribution). This approach is also popular as a two-dimensional postprocessing technique, known as *morphing* for general parameters or *warping* when applied specially to shape location (Alexa, Cohen-Or, & Levin,

2000). While shape interpolation produces limited animation effects, the idea has led to interesting *exploratory modeling* tools that allow users to navigate a virtual design space consisting of all possible shapes within a certain range of parameters (Talton, Gibson, Yang, Hanrahan, & Koltun, 2009). Furthermore, *shape interpolation* continues to be widely used for specific deformations in real-time animation, for example in facial animation and lip-syncing: the exact shape parameters of the jaw, tongue, and surrounding muscle groups are captured according to different facial expressions and may be interpolated to produce the intermediate results (Parke & Waters, 2008). The disadvantage of this approach is that only a limited range of animations are supported between the parameters of the parent shapes.

## Hierarchical Modeling

Hierarchical modeling is a general term which describes how *rules* and *representation* change by relative *ranks*. In the field of simulation, this is associated with progressively changing *efficiency* and *accuracy* of a system in order to target a new class of applications (Berendsen, 2007). However in computer animation, the term is more specifically used to describe the enforcing of relative *constraints* for content organized in a treelike structure (Parent, 2012). The root of the tree generally describes the most important or the largest part of the system; whereas the smallest branches generally represent small objects or fine shape features. This means that deformation can be acquired according to the region of interest (ROI), by traversing the tree and enforcing relative constraints until the desired level of deformation accuracy at each branch has been reached.

Hierarchical models are observed in large-scale examples, such as where a satellite orbits the Earth, which orbits the Sun, which itself orbits the Milky Way, and also in small-scale examples such as the location of a human hand, which is constrained by the joints of the wrist, elbow, and shoulder accordingly. In computer animation, a hierarchical model therefore: (1) assists the manipulation of parameters for groups of objects, (2) means that accuracy can be achieved according to the region of interest, (3) provides an intuitive abstraction for controlling shape features undergoing multiple deformations, and (4) increases efficiency where transformations can be concatenated to reduce operations (Lengyel, 2011).

*Kinematic Linkages* describe motion with respect to other objects, with applications also in the field of Robotics (Craig, 2008). Manipulating linkage chains in a hierarchical model can be achieved with two approaches: (1) Forward Kinematics, and (2) Inverse Kinematics. *Forward Kinematics* is the top-down manipulation of the linkage chain, starting at the root, giving the animator complete control over the direction of motion of each joint (known as a *degree of freedom*). Accurately positioning links at the *end* of the hierarchical chain is inconvenient (Paul, 1981), for example in human character animation, the animator must first manipulate the

torso, then the shoulder, elbow, wrist and finally the fingers in order to reach a *goal* location. *Inverse Kinematics* addresses this problem: the animator specifies the target parameters at the end of the chain (such as the location of the fingers) and a linear system is solved to compute the rotations of all intermediate joints, but this may give undesirable joint configurations.

*Control Skeletons*, also known as rigged skeletons, are a hierarchical linkage chain whose joints are sampled on the smooth central curve of the shape (called the *centerline* or curve skeleton) (Cornea, Silver, & Min, 2007) which itself is located near the medial axis (Biasotti, Marini, Mortara, & Patan, 2003). Segments in the chain are known as *bones* and are mapped to shape features in a process called *skinning* or *rigging*, such that the shape geometry can be efficiently updated by interpolating the weighted influence of its corresponding bone matrices. This means that deformation can be modeled by rotating or scaling the bones, using either forward or inverse kinematics. Because control skeletons map to the shape features, they are much more intuitive, compact, and efficient in comparison to free-form deformation techniques; however they are difficult to generate for arbitrary input shapes as the shape centerline is highly sensitive to noise at the shape boundary. Furthermore, the density for which joints are sampled (the *skeleton resolution*) depends on the application quality requirements, which varies for multi-resolution volumetric content, and therefore a *feature-varying skeletonization* method is needed (Willcocks & Li, 2012).

## **Physically Based Modeling**

Physically-based modeling is where the animator specifies a physical *system* with *rules* influencing the scene geometry, whose motion is simulated across multiple frames. The output animation is therefore constrained by the level of *physical accuracy* of the simulation, which can be loosely defined as a hierarchical model (simulation) with progressive ranks of physical approximation at increased efficiency but with decreased accuracy. Increasing the level of physical accuracy, such as simulating physical interaction between individual human hairs, widens the range of unique motion behaviors, but is more expensive, whereas increasing the level of approximation produces an efficient but more predictable and generic animation. This means that physically based modeling methods support both offline and real-time rendering by adjusting the *physical realism* to reach the desired class of target applications (Berendsen, 2007; Parent, 2012).

Physically based modeling literature is heavily application specific; a unique effect is observed in the real world, and then the level of accuracy of a *system* and its *rules* are approximated to demonstrate the effect at an acceptable degree of realism. The majority of real-time computer graphics literature focuses on a *macroscopic* level, whereby visual effects are represented by *coarse* approximations, such as the Navier-Stokes Equations (or simplifications) for modeling

fluid motion. Popular examples include simulating and rendering effects such as *fire* (Nguyen, Fedkiw, & Jensen, 2002), *smoke* (Fedkiw, Stam, & Jensen, 2001), *clouds* (Dobashi, Kaneda, Yamashita, Okita, & Nishita, 2000), *water* (Enright, Marschner, & Fedkiw, 2002), *wrinkles* (Hadap, Bangerter, Volino, & Magnenat-Thalmann, 1999), *elasticity* (Terzopoulos, Platt, Barr, & Fleischer, 1987), *sand* (Sumner, O'Brien, & Hodgins, 1999), *rain* (Garg & Nayar, 2006), and *cloth* (Choi & Ko, 2002; Vassilev, Spanlang, & Chrysanthou, 2001) at varying degrees of realism. Many of these effects are modeled by applying Newton's laws of motion, including *spring* systems (Georgii & Westermann, 2005; Provot, 1996) and *rigid-body simulation* (Baraff, 1992) to a system of *particles* (Reeves & Blau, 1985; Sims, 1990) or volumetric elements (Y. Chen, Qing-Hong, Kaufman, & Muraki, 1998). The purpose of this thesis is to enable support for hierarchical, sparse, volumetric deformation; therefore a hierarchical physically based model can be natively supported (Berendsen, 2007) whose *simulation accuracy* can be automatically adjusted according to the region of interest.

## Advanced Modeling

Interpolation, hierarchical, and physically based modeling strategies have significant crossover with each other. For example, in real-time or network simulations, a physical system can be updated at discrete time steps, whereby the intermediate results are *interpolated* for efficiency. Another example is where the bones of a *control skeleton* are assigned physical cylinder primitives, of which *rigid-body simulation* is applied to produce unique animations that interact or *collide* with other physical rigid-body primitives in the environment (Zordan, Majkowska, Chiu, & Fast, 2005). More advanced animation behaviors may also be *learned* from training datasets, such as *fitted* motion capture data (C. K. Liu, Hertzmann, & Popović, 2005) in order to produce unique motion styles that would otherwise be laborious to generate. Furthermore, knowledge from other academic disciplines can be combined with the modeling techniques to more accurately describe sophisticated and abstract types of deformation or motion behavior; popular examples include the growth of plants in the field of *botany* (Reffye, Edelin, Françon, Jaeger, & Puech, 1988), the interaction of large crowds or object swarms in the field of *artificial intelligence* (Kennedy, 2006; Sung, Gleicher, & Chenney, 2004), and the movement of characters in the art of *choreography* and the science of *kinesiology* (Y. Li, Wang, & Shum, 2002).

In conclusion, there are many different approaches for manipulating content, organizing constraints, and simulating animation. These are generally measured in terms of human intervention, computational efficiency, and simulation accuracy. The range of applications can be greatly extended using hierarchical models: (1) hierarchical content can be simulated at the target level of physical realism according to the region of interest, and (2) shape features can be intuitively manipulated with relative constraints using a control skeleton.



## 2.4 Spatial Partitioning

Spatial partitioning (or space partitioning) means *dividing space* in order to efficiently query geometric information. This is therefore important for supporting massive scenes by retrieving influential data from secondary storage which contributes to the output simulation, and it also accelerates content intersection tests for culling algorithms, real-time rendering and collision detection (Akenine-Möller et al., 2008; Ericson, 2005). An effective spatial partitioning algorithm will map contiguous regions of one-dimensional memory to the important regions of  $n$ -dimensional space, and can be measured by the: (1) mapping performance, (2) memory usage, (3) construction cost, (4) update costs, and (5) parallelism. These attributes are central to enabling support for sparse volumetric deformation, given the massive amounts of dynamic volumetric content which need to be *selected*, *processed*, and finally *displayed* each frame.

The field of spatial partitioning can be categorized by the type of *spatial data structure* used to organize geometric information in  $n$ -dimensional space (Samet, 1995). Spatial data structures arrange content in: (1) *grid* structures, or (2) *tree* structures; however it is also worth discussing (3) *spatial hashing*, which is a mechanism that increases the functionality of spatial *grid* and *tree* based methods, supporting much larger virtual resolutions than by natively storing the structures in memory, and can also be used to improve parallelism and cache coherency. In the literature, the numerous  $n$ -dimensional data structures (Samet, 2006) are often well-tuned for specific applications, which include: database management systems (e.g. spatial databases, multimedia databases), computer vision, pattern recognition, geographic information systems (GIS), and finite-element analysis. The application of sparse volumetric deformation requires the spatial data structure to update in real-time according to dynamic changes in the scene, therefore the structure must excel in its speed of updating, mapping performance, and level of parallelism. To achieve this *efficiency*, the spatial data structure can benefit from the regularity of the volumetric content.

### Grid Structures

The most common type of grid is the *regular grid*, which is a special case of *structured grid* where all the spatial regions (called *cells* or *bricks*) are *congruent* (of equal size and shape). The tessellation of the regular grid is at a fixed resolution, which allows for geometric content to be accessed efficiently by the overlapping regions; however this introduces the balancing problem of deciding the *grid density*, which is the ratio of objects to regions. Choosing a low-resolution grid with densely distributed objects will impact performance, as each object needs to be *searched* for within each region, whereas choosing a high-resolution grid will consume too much memory (Cohen, Lin, Manocha, & Ponamgi, 1995). The real world contains varying

object distributions as it is has *sparse* regions of empty-space, as well as regions of *densely* packed objects, which means that the regular grid is often both too coarse and too fine.

Supporting varying object distributions can be achieved with *adaptive grids* (Klimaszewski & Sederberg, 1997), *hierarchical clustering* (Cazals, Drettakis, & Puech, 1995), or by *recursive spatial subdivision* until some density criteria for each grid region is satisfied (Jevans & Wyvill, 1988). These approaches use multiple grids stored hierarchically at different resolutions according to content overlapping criteria, which means that sparse regions can be captured without consuming too much memory. However the added complexity of managing multiple grid overlapping criteria is difficult to parallelize, especially when considering *dynamic* volumetric content. More simple overlaying criteria (such as *hierarchical grids* in (Ericson, 2005)) result in deep hierarchical levels consuming lots of memory, which is unsuitable for representing massive sparse scenes. While a *spatial hashing* mechanism can be used to limit this memory consumption, the sparse grid regions cause *cache misses* impacting performance; therefore it is better to use a more consistent *tree* structure to divide the empty-space in order to focus on a tighter spatial region for more coherent *hashing*.

## Tree Structures

Tree data structures are widely used in real-time rendering and collision to hierarchically organize content. This organization is generally achieved by partitioning data according to: (1) its value (labeled *tree-based*), or (2) the embedding space from which the data is drawn (labeled *trie-based*). While both approaches decompose the space into groups, the key difference is that *tree-based* methods are generally more concise but less regular, which is therefore suited for irregular geometry such as polygon meshes, whereas *trie-based* methods are generally suited for *regular* content, such as *voxels* (which represent the value of a *regular grid*). This generalization is shown more clearly in Samet's taxonomy (Figure 8), which shows the interrelationship among the different types of multidimensional data structures (Samet, 2006) using an example set of 2D points. The root of Samet's taxonomy represents no identifiable organization of the content, whereas deeper branches represent the progression from being based on a fixed grid (arrays) to being based on an adaptive grid, whose regions are more flexible in terms of size and shape. The *grid density* for elements in the deep regions is low, but the partitioning procedure is more complex, which is generally more difficult to parallelize. Under this observation, the shallowest structures that retain low *grid density* are *quadtrees* (i.e. octrees in 3D) that are organized on the embedding space.

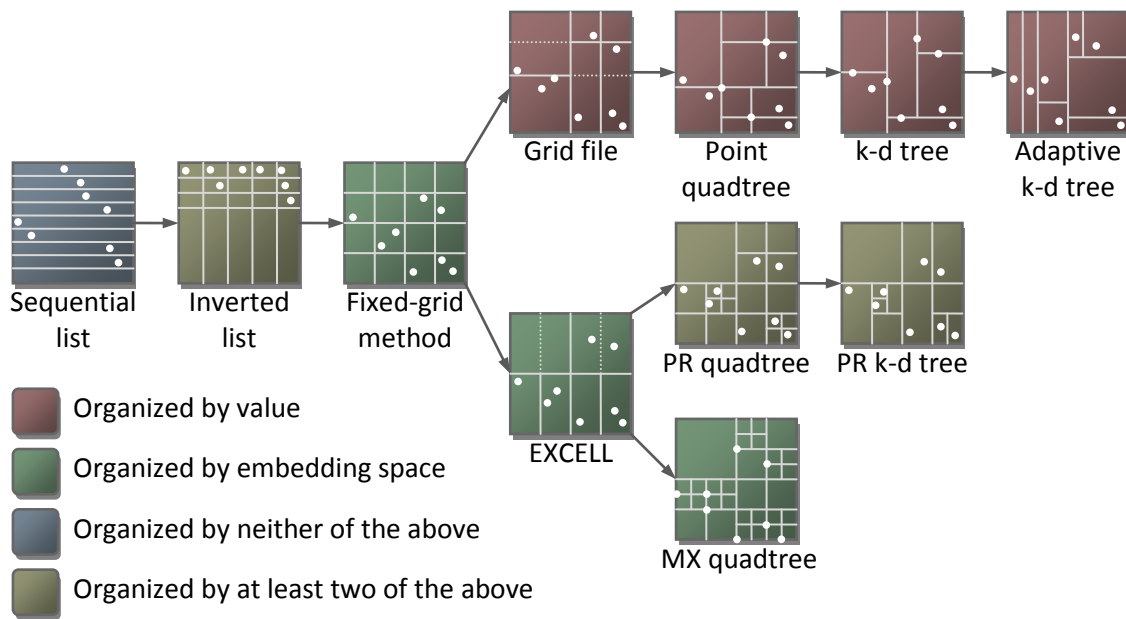


Figure 8: Samet's taxonomy: The interrelationship among different data structures, where deeper levels of the tree are more adaptive with flexibility over the region size and shape.

It is challenging to construct a suitable structure for storing and organizing massive amounts of dynamic volumetric content. Ideally, to utilize the hardware cache, trees should be ordered in memory such that nodes occur linearly in memory during traversal (Ericson, 2005). However this can present challenging design issues, in particular when considering how hierarchical links between nodes are managed. In practice, trees can link data by using either: (1) *pointers*, or (2) hashed access. The remainder of this section examines recent literature which constructs trees on volumetric data using pointers, whereas the next section focuses on data orderings and *spatial hashing* methods.

Generating a tree using pointers is perhaps the simplest method to understand, as pointers provide a natural metaphor describing the links between hierarchical nodes. For example, a tree may be encoded in a *top-down* breadth-first search (BFS) order where each node can store a pointer and an integer value representing the number of its  $n$ -children, which are contiguous in memory. These pointers (links) are represented by arrows in Figure 9, where the left image shows the natural representation of the tree structure, and the right image shows how the nodes are allocated and referenced for top-down traversal in memory. An alternative depth-first search (DFS) memory order, as shown in Figure 9 (lower-right), can be implemented to improve cache utilization as the traversed nodes are slightly closer in memory, however this requires, for example, additional 32-bit pointers and 32-bit positional data per node making the approach often impractical.

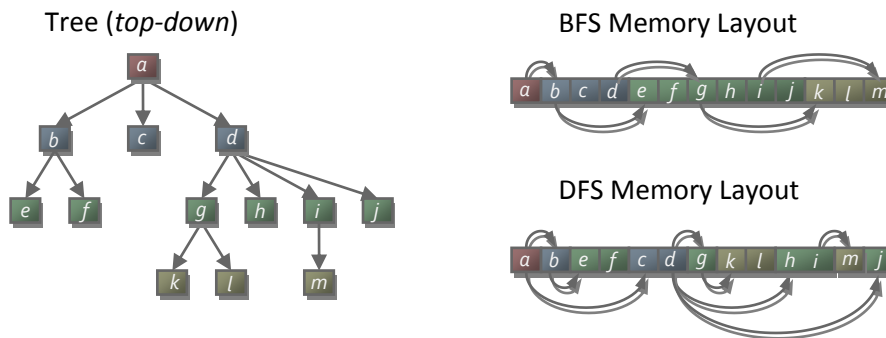


Figure 9: Example tree layouts in BFS and DFS memory orderings.

In the case of *quadtrees* and *octrees*, the BFS memory layout allows the positional data to be reconstructed during traversal by keeping child *voxels* in a fixed order and using an 8-bit flag per branch to determine which of the eight children are present (Benson & Davis, 2002). This popular approach has been efficiently ported to the GPU by representing the contiguous memory as a 2D texture in row-major or column-major order (Lacoste, Boubekeur, Jobard, & Schlick, 2007), where only relevant parts of the hierarchy are transmitted (Lacoste, Perin, & Jobard, 2008). There are many variations on the above layouts, for example one strategy involves preserving all eight children per branch, which increases memory, but simplifies ray traversal without requiring the 8-bit flag per voxel. (S. Laine & Karras, 2011) proposes to use reduced 15-bit pointers, but if the range is exceeded they use an additional *far-bit* that indirectly references another 32-bit pointer. It has also been proposed to use entropy encoding by compressing both the child data bits and colors for further memory reduction (Olick, 2008). To summarize, while pointer-based approaches are simple to understand and easy to implement, the pointers themselves usually consume 16-bits or more, and may lead to poor cache utilization. This has motivated the search for improved data locality and use pointerless methods.

## Spatial Hashing

A *hash function* is used to map a large dataset to a smaller data set, for example a *spatial hash function* maps points in  $n$ -dimensional space to a single 1D value, such as an integer. This is generally a very *efficient* function, for example it can be achieved in 3D by simply interleaving the  $xyz$  bits using a few bitwise operations. The main idea behind *spatial hashing* is to organize the data in a process known as *one-dimensional ordering*, such that nearby data in  $n$ -dimensional space is nearby in 1D memory. Therefore the hardware cache is localized allowing for efficient content retrieval according to the region of interest (this is discussed in more detail in the later section on Program Optimization).

Cache friendly spatial partitioning can be achieved by storing quadtrees and octrees without pointers (Gargantini, 1982); by hashing the index data for direct access to any node (Figure 10). Notable works include (Lefebvre & Hoppe, 2006) who pack sparse data into a dense 3D texture using a small 3D offset table to *jitter* each hash function into a minimal perfect hash function. This results in exactly two memory accesses per query which is ideal for parallel evaluation on GPU hardware. Another recent advance in the field of pointerless octrees exploits the dual of the structure for even faster performance with less memory consumption (Lewiner et al., 2010). Of particular relevance is (Lauterbach, Garland, Sengupta, Luebke, & Manocha, 2009) improved by (Karras, 2012), who assign, sort and compact Morton codes (a type of 1D hash) to make them unique. This step efficiently reorders the voxels such that they are sorted along the z-order curve, preserving data locality as multi-dimensional voxel indices are mapped to one-dimensional contiguous memory. This means that a binary radix-tree can be constructed using the unique keys to generate the octree hierarchy, and that the whole process can run in parallel on the GPU with excellent scalable performance and low memory consumption.

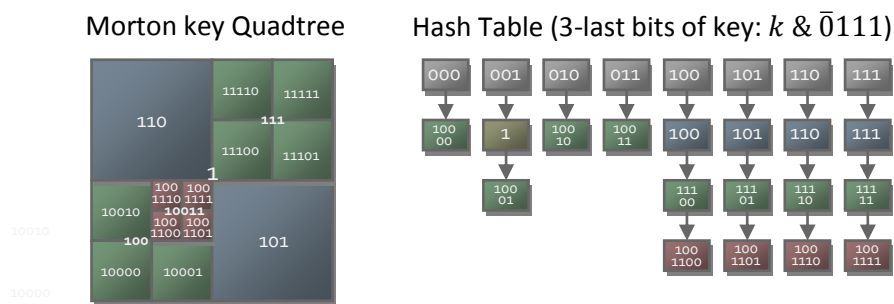


Figure 10: Pointerless quadtree example (Lewiner, Mello, Peixoto, Pesco, & Lopes, 2010).

In addition to low memory consumption and cache coherency, spatial hashing can offer other benefits in the traversal process. While traversal is typically top-down or bottom-up according to the memory encoding, hashed (or pointerless) octrees allow for direct access the octree data simply from a target position and depth value, which is shown in the example for the 2D quadtree case in Figure 10. Hashed octrees have therefore led to the development of more optimized statistical octree searches with far less search time and a low memory footprint (Castro, Lewiner, Lopes, Tavares, & Bordignon, 2008). Another notable advancement was the development of the *kd*-restart and *kd*-backtrack algorithms, which run without a stack (Foley & Sugeran, 2005). This has led to several fixed-size buffer variants, also known as stackless or short stack algorithms, which have been generalized to arbitrary bounding volume hierarchies (BVH) (Samuli Laine, 2010). Also, fixed-buffer techniques have been efficiently implemented with octree ray traversal on the GPU, restarting from the correct level of hierarchy to avoid requiring additional traversal from the root node each time the ray progresses through the scene (S. Laine & Karras, 2011).

In conclusion, advances in parallelization with spatial hashing have demonstrated extremely efficient construction for regular quadtrees and octrees on today's GPUs, which can organize potentially dynamic data in a cache-friendly hierarchical ordering without requiring pointers. This enables fast traversal with applications in collision detection and real-time rendering.

## 2.5 Real-time Rendering

*Real-time* rendering involves the rapid creation of images such that users can dynamically interact with the virtual environment. The rate of which images are produced is measured in frames per second (FPS) or by the time between frames in milliseconds (ms): humans cannot detect individual frames at rates above 72FPS, whereas under 6FPS the immersion within the environment deteriorates (Akenine-Möller et al., 2008). Generally, an application is considered real-time if it can produce images above 15FPS (less than 66ms between frames) however more visually appealing or smooth animation occurs above 30FPS (less than 33ms). Rendering, on the other hand, is concerned with creating an image from a model (Geometric Modeling) and can be measured by the attractiveness, realism, or visual appeal of the generated output. Therefore the relationship between *real-time* and *rendering* implies a balance between the *efficiency* and *quality* of the generated images in order to create an immersive and captivating dynamic virtual environment for the user. The field of *rendering* can be split into two branches: (1) photorealistic rendering, and (2) non-photorealistic rendering (NPR). Photorealistic rendering accurately models *shapes* and *light* as they appear in the real world, then simulates the *physics* of light (*optics*) as it interacts with a scene before *sampling* the light which reaches the camera. Whereas NPR is more concerned with producing *stylized* artistic images, for example to convey an emotion, illustrate an idea, or sell a product. This thesis concentrates on representing massive, detailed and dynamic volumetric scenes as they occur in the real world, and it is therefore more related (but not limited) to photorealistic rendering.

The real-time rendering literature is organized by the two primary *techniques* used for modeling and simulating light within dynamic scenes: (1) rasterization, and (2) ray tracing. However the section is not restricted to discussing light; it also concerns with how shape and deformation can be modeled within these techniques. The later parts of this section expand to more specific literature about: (3) out-of-core rendering, (4) optimization, and (5) parallelism. These topics are important for enabling support for massive dynamic scenes, given the *efficiency* expected of a real-time application, and the constraints of current hardware.

### Rasterization

Graphics hardware has become extremely efficient at rasterizing large numbers of polygons, which may have deformations, tessellation, and parameterized materials. The *graphics rendering pipeline* is commonly associated with the state of the art *rasterization* real-time

rendering method, which operates on a set of rendering *primitives* (Akenine-Möller et al., 2008). The vertices of these primitives are then transformed, and projected to the screen by the camera matrix, according to structures in memory such as vectors, matrices, textures, or arrays of data (such as control skeletons). More primitives may be emitted (Blythe, 2006) and displaced to produce surface variety, which is then followed by a *rasterizer* stage that converts the visible (non-clipped) primitives into a raster image while interpolating per-vertex attributes for any intermediate pixels. The final stage enables lighting information to be calculated for each pixel in the output image, by accessing structures in memory (such as textures, the camera location, and light positions) according to the interpolated per-vertex attributes.

While the rasterization approach is optimized for processing large numbers of polygons, it is not designed for modeling the physics of light as observed in the real world. However more recently, real-time rendering research has gained interest in modeling light, where new interactive *global illumination* techniques have been proposed to transport light in scenes, giving an improved approximation of the rendering equation (Kajiya, 1986). It is therefore not surprising that many of these techniques use rasterization (Ritschel, Dachsbacher, Grosch, & Kautz, 2012), in particular by storing advanced lighting information in additional frame buffers and combining the results to compute the output image (based on deferred shading (Saito & Takahashi, 1990)).

An alternative real-time rendering approach, using rasterization, involves storing advanced lighting information inside large numbers of 3D point primitives: these are discussed in the book by (Gross & Pfister, 2007) and the survey (Kobbelt & Botsch, 2004) with topics such as point acquisition, reconstruction, splatting, sample spacing, and hardware acceleration. A recent example of acquiring lighting information with 3D points uses a bounding sphere hierarchy (BSH), which enables parallel *cuts* to be made for calculating indirect light from many different views in the scene (Hollander, Ritschel, Eisemann, & Boubekeur, 2011). Point-based techniques can also be optimized using visibility culling (Koo & Shin, 2005), LOD selection (Pajarola, 2003), screen-space decimation (Rusinkiewicz & Levoy, 2000), and temporal coherency (Guennebaud, Barthe, & Paulin, 2004), whereby expensive rendering and deformation operations only influence the set of influential points. The primary limitation of these strategies is that they are not suitable for *deformation*, as gaps occur between the regularly sampled points. This problem has been addressed for free-form modeling of point sampled geometry by using a hybrid of unstructured point clouds with an implicit moving least squares approximation (Pauly, Keiser, Kobbelt, & Gross, 2003). While suffering a performance penalty making it unsuitable in rendering large scenes, their work is able to dynamically insert or delete samples according to the local sample density in order to preserve topology. A more efficient strategy is given by (Marroquim, Kraus, & Cavalcanti, 2007) who propose an image-

space reconstruction filter. This achieves excellent performance but suffers from aliasing artifacts. In contrast, recent work by (Bautembach, 2011) demonstrated deformation of rasterized point objects simply by stretching the point size based on the number of bone weights  $> 0$ . This is much more efficient than (Pauly et al., 2003), yet it still produces undesired sampling error from excessive point overlapping and, in extreme cases of deformation, undesired holes appear. However the idea of inserting samples and enlarging samples to address such problems is worth investigating.

## Ray Tracing

This section discusses real-time rendering techniques related to *ray tracing*, or techniques that more accurately model the transport of light. Ray tracing based methods have long been the primary rendering strategy for offline animation (see *Modeling Deformation*) however, with the advances of graphics hardware and spatial data structures (see *Spatial Partitioning*), they are now finally possible in *real-time* (Wald et al., 2009). In the real world, photons from lights bounce amongst objects and at each point of intersection they are absorbed, reflected, and refracted in numerous ways. Eventually some of these photons reach the lens of the eye, which are then focused onto photoreceptive cells in the retina (likewise cameras focus them onto photographic film or an image sensor). The rendering equation (Kajiya, 1986) describes all possible paths of light in this process (but it does not capture all physical effects of light, such as fluorescence) and therefore can be used to generate near *photorealistic* images. However it is difficult to compute, as light bounces extremely quickly in a recursive process that requires some approximation. The basic idea of *ray tracing* (in optics) is to cast a large number of rays into the scene to simulate the transport and interaction of light. This can generally be achieved by: (1) casting rays from the light sources, as in the real world (this is very expensive), (2) casting rays from the camera (this is fast, but does not describe all the physical effects of light), (3) casting (or *connecting*) rays from samples (i) in the scene or (ii) on the shape surface, or (4) some combination of these approaches. Instead of randomly shooting rays, *importance sampling* can be used, where important illumination can be found using either bi-directional path tracing (BDPT) (Lafortune & Willems, 1993) or Metropolis Light Transport (MLT) (Veach, 1998).

The most significant advances in light transport and *Global Illumination* (GI) are categorized into the seven strategies proposed by the excellent state of the art (Ritschel et al., 2012). These strategies are briefly discussed and numbered in square brackets [1-4] according to the general light direction given in the previous paragraph. They are: (1) *finite surface element* based, which discretize the shape surfaces into patches and transport light between patches [1, 3ii, and 4] (Goral, Torrance, Greenberg, & Battaile, 1984; Meyer, Eisenacher, Stamminger, & Dachsbacher, 2009; Soler, Hoel, & Rochet, 2010), (2) *Monte Carlo ray tracing*, which casts rays



in random directions (accelerated with BDPT or MLT) from samples in the scene before connecting them to the lights [3i] (Kajiya, 1986), (3) *photon mapping*, which emits photons from light sources and then either gathers them or estimates their density [1] (Hachisuka & Jensen, 2010; Jensen, 1996), (4) *instant radiosity*, which emits photons from light sources that become virtual point lights (VPLs) that can use shadow mapping [1] (Keller, 1997; Ritschel, Eisemann, Ha, Kim, & Seidel, 2011), (5) *many-light-based GI*, which gathers lighting into surface samples from many lights stored in a spatial data structure [3ii] (Walter et al., 2005), *point-based GI*, which gathers lighting and occlusion information into surface samples from both lights and shapes stored in a spatial data structure [3ii] (Maletz & Wang, 2011; Ritschel et al., 2009), (6) *discrete ordinate methods*, which discretize the scene spatially and directionally for transferring light energy between grid regions [3i] (Chandrasekhar, 1960; Kaplanyan & Dachsbacher, 2010), and finally (7) *pre-computed radiance transfer*, which assumes static geometry and pre-computes lighting effects stored on the shape surface for real-time rendering [3ii] (Sloan, Kautz, & Snyder, 2002). The comparison between global illumination methods by (Ritschel et al., 2012) shows that no single rendering strategy outperforms in the measurement criteria that are important for this thesis, for example: (1) performance, (2) quality, (3) dynamic scenes, (4) scalability of geometric complexity, and (5) parallelism. The focus of this thesis is on enabling support for sparse volumetric deformation in massive scenes; the primary goal is not to implement an accurate model of light transport. However it should be observed that many of the interactive global illumination strategies in the literature greatly benefit from hierarchical spatial data structures, such as octrees.

In practice, ray tracing not only consists of a spatial data structure (see *Spatial Partitioning*), but also a ray traversal process. The purpose of the traversal process is to accelerate ray-shape intersection tests, which are used to find where light collides with scene objects. This is usually achieved in a dividing process that searches the scene objects in a treelike data structure. The reader is referred to the large body of literature that focuses on ray traversal in BVH's, grids, octrees, and *k-d* trees; in particular an excellent state of the art is given on the topic of ray tracing animated scenes by (Wald et al., 2009), a doctoral thesis by (Havran, 2000), and an overview on GPU assisted ray traversal by (Thrane, Simonsen, & Ørbæk, 2005). For rendering volumetric content, methods often rely on a simple ray-box intersection check, which became popular following the developments of (Kay & Kajiya, 1986). This was shortly followed by a modification of the traditional digital differential analyzer (DDA or a generalized Bresenham), which allows for quick ray casting in uniform grids by only two comparisons and one addition operation per voxel (Amanatides & Woo, 1987). These two simple intersection approaches formed the foundation for volumetric rendering in hundreds of papers, and are still popular in today's literature (Aila & Laine, 2009).

## Out-of-Core Algorithm

An out-of-core (or external memory) algorithm is able to process data that is otherwise too large to fit into memory at one time. The main idea is to retrieve data from secondary storage only when it is needed, or in anticipation that it will be needed in the near future, while freeing redundant data accordingly. In rendering, this means that only influential regions near the camera are stored in memory; however there is still a performance problem of retrieving the data from secondary storage without generating excessive page faults. With spatial tree data structures, the number of page faults can be reduced by aggregating sections of the tree into pages, such that groups of branches can be retrieved instead of fetching each branch from storage with an independent operation. Similarly, data structures such as the *multiway tree* or B-tree reproduce this effect, where branches are permitted to have multiple values (Bayer & McCreight, 2002; Samet, 2006).

Rendering massive scenes with an out-of-core algorithm requires a careful choice of selection criteria or error metrics, which are used to decide the important scene content (Carmona & Froehlich, 2011). A common trend in the out-of-core selection criteria is to use screen-space metrics based on the pixel size (Dick, Schneider, & Westermann, 2009) reducing the sampling frequency further away from the camera while requesting only *visible* data (Crassin, 2011). However there are other selection criteria based on spectral analysis (Suwelack, Heitz, Unterhinninghofen, & Dillmann, 2010), adaptive cost prediction (Aronov, Herv, Bronnimann, Chang, & Chiang, 2003; Frank & Kaufman, 2009), light gathering area (Christensen, 2008; Kontkanen, Tabellion, & Overbeck, 2011), projected silhouette overlapping (Z. Chen & Chou, 2006), and agglomerative clustering criteria (Walter, Bala, Kulkarni, & Pingali, 2008). In general, these criteria depend on how shapes in the scene are modeled, and also on the type of spatial data structure used. However, hierarchical volumetric content has the advantage of being able to dynamically change its geometric resolution, which is well-suited for screen-space visibility criteria. These approaches can also be constructed across multiple frames (Lei & Xu, 2009) by exploiting temporal coherence (Coorg & Teller, 1996; Crassin et al., 2009), which results in a small amount of visual error that is considered acceptable in order to prevent the transfer delay from interrupting the sense of immersion within the real-time environment.

## Program Optimization

Program *optimization* is the process of modifying a piece of software in order to improve its performance, accuracy, or reduce its resource consumption. In real-time rendering, the focus is on improving the rendering efficiency to produce smoother frame rates, and improving the rendering accuracy to increase the sense of immersion. The methods include: (1) algorithm modification, (2) architecture tuning, and (3) parallelism. In order to apply these techniques, it

is important to identify *where* the bottlenecks are located in the graphics pipeline, by either constructing progressive performance tests or by using tools that *analyze* or *profile* software to display meaningful information at each stage of the application. These tools are in a continually evolving list, where current examples include: CPU architectures (Intel VTune, AMD CodeAnalyst), APIs (OpenGL gDEDebugger, Windows Performance Toolkit), specific programming languages (C++ PVS Studio, Java VisualVM), GPU programming (NVIDIA NSight, NVIDIA PerfKit, AMD GPU PerfStudio), and GPGPU computing (NVIDIA Visual Profiler, AMD APP Profiler, AMD APP Kernel Analyzer). Further discussion on locating bottlenecks within the graphics pipeline is given in the book by (Akenine-Möller et al., 2008).

In real-time rendering, there are many high-level algorithms to optimize a program. Popular examples include: (1) culling techniques, (2) level of detail, (3) texture mipmaps, (4) imposters, (5) instancing, (6) indexing, (7) space leaping, and (8) simulation layers. *Culling* is the selective removal of geometry that does not contribute to the output image, where: (i) *view frustum culling* removes geometry that is outside the camera region, (ii) *backface culling* removes geometry that faces away from the camera, (iii) *portal culling* removes geometry that is outside interior regions clipped by portal doorways, (iv) *contribution* or *detail culling* removes geometry that is too small, distanced, or otherwise insignificant, and (v) *occlusion culling* removes geometry that is hidden behind other objects (Zhang, Manocha, Hudson, & Kenneth E. Hoff, 1997). *Level of detail* (LOD) systems hierarchically model progressive resolutions of shape or deformation, which enables high-contributing objects to be simulated at greater precision than insignificant objects. These different resolutions can either be switched, interpolated, or faded (Giegl & Wimmer, 2007) however methods can also completely change *representation*. For example a polygon LOD system can be used for objects near the camera, which can then transition into a point-based continuous LOD system for distanced objects (Rusinkiewicz & Levoy, 2000). *Texture mipmaps* extend this concept for texture filtering by pre-calculating different hierarchical image resolutions, which may also be compressed (Williams, 1983). An *imposter* replaces scene geometry with a 2D image or animation; this is popularly incorporated into an LOD system as a replacement for distant objects (Decoret, Sillion, Schaufler, & Dorsey, 2001). *Geometry instancing* involves the batching of similar scene objects to avoid duplicate processing of shared attributes (Ashraf & Zhou, 2006). This is alike *indexing*, which uses an index buffer (or an element buffer object) to reuse *identical* attributes repeatedly (Blythe, 2006). *Space leaping* (or empty space leaping) accelerates ray casting by skipping empty regions of space, for example: (i) with a spatial data structure, (ii) by rasterizing simplified bounding primitives to find the ray start location, or (iii) by exploiting temporal coherence (Yagel & Shi, 1993). Lastly, *simulation layers* group objects such that expensive operations, such as lighting or collision, need not be computed between all objects in the scene (Cordier & Magnenat-Thalmann, 2002).

The foundation of all these algorithmic techniques is efficient organization of the underlying data, which itself depends on the target machine architecture. In the section on spatial hashing, it was discussed that data could be organized in memory for cache utilization, with the example of ensuring nearby regions of  $n$ -dimensional space are also nearby in 1D memory. The idea of cache utilization is to use data in a way which benefits the memory hierarchy of the target architecture (Drepper, 2007). Modern computers have many computational cores with cache components organized in hierarchy at different levels (L1, L2, L3), as exemplified in Figure 11. The memory that is close to the cores operates with near zero latency, at about the speed of the core, however this memory is very challenging (expensive) to manufacture in large quantities. If the core needs to fetch the next instruction, or data required for the current instruction, it looks in the progressive cache levels and directly retrieves the data accordingly. However if the data is not found, a *cache miss* occurs meaning that the data has to be retrieved from memory, which can be on the order of 50 to 500 times slower (Ericson, 2005). In contrast to the CPU, the GPU contains multiple streaming multiprocessors (SM) which each contain a large number of cores that enjoy much higher memory bandwidth (Nickolls & Dally, 2010). However data must be transferred by the expansion bus, which is limited to the bus bandwidth or lower memory bandwidth causing a potential bottleneck. This is especially problematic for real-time rendering large quantities of volumetric content, making it necessary to use techniques such as *asynchronous transfers* and exploit *temporal coherency* such that data in the cache levels remain, as it does in the scene, over multiple frames (Cozzi & Riccio, 2012; Crassin, 2011).

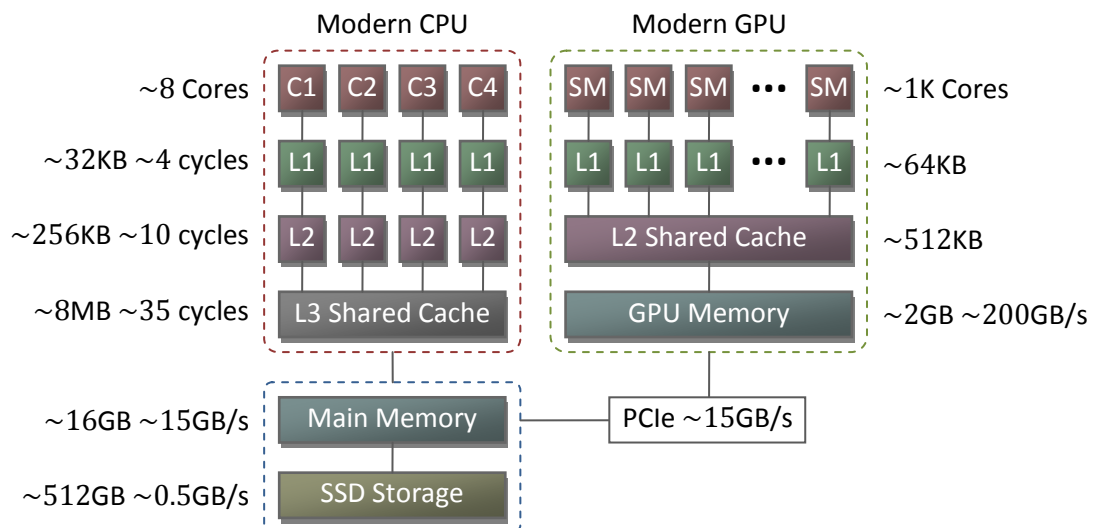


Figure 11: Abstraction of the basic memory hierarchy for a modern CPU and GPU.

The advances in compiler technology only double in program performance about once every 18 years, and cannot currently be relied on for critical optimizations (Scott, 2001). To use the data blocks (cache lines) effectively, intricate and profiled techniques are required to (i) *localize* and (ii) reduce the *size* of the stored data according to the target architecture (Jia, Shaw, & Martonosi, 2012), for example by checking: (1) whether smaller data types can be used, (2) if the fields within a structure can be reordered for better cache alignment, (3) if structures can be split by the data access patterns, (4) if data can be packed into smaller data, (5) if data can be determined more efficiently at runtime (Mikkelsen, 2010), (6) if pointers can be replaced with smaller data, or (7) whether some alternate form of compression is possible, such as reusing parent data in tree structures. For extended and practical examples on these techniques, the reader is referred to the books (Akenine-Möller et al., 2008; Chilimbi, 1999; Drepper, 2007; Engel, 2006; Ericson, 2005).

## Parallelism

Parallel computing involves performing simultaneous calculations to process large problems or multiple separate problems more efficiently. Flynn's taxonomy groups parallel and sequential programs according to whether they have single or multiple *instruction streams* that operate on single or multiple *data streams* (Flynn, 1972). This creates four categories: (1) single instruction single data (SISD), which is an entirely sequential program, (2) single instruction multiple data (SIMD), which describes where multiple processors perform the same instruction on multiple data points, for example *streaming SIMD extensions (SSE)* on modern CPUs, (3) multiple instructions single data (MISD), which has limited applications such as detecting problems in data, and (4) multiple instructions multiple data (MIMD), for example *multithreading*. Flynn's taxonomy is simple to understand, however in practice there is much crossover between the categories. This subsection will look in more detail at the architecture of the GPU, and then discuss how *parallelism* can be well-implemented using modern computer architectures (see Figure 11) for the application of volumetric deformation.

The SIMD idea that a *single instruction* operates on *multiple data* does not extend in practice for many cases, especially where one data element influences another. This is addressed by constraining operations to lists of data (vector processing), for example a single instruction is performed on two input lists, giving a new output list containing the componentwise result (Nuzman, Rosen, & Zaks, 2006). This is in contrast to MIMD, where it is commonplace for instructions to share and influence resources (Hatcher & Quinn, 1991). The heart of the GPU is an SIMD model whose instructions are organized around the graphics pipeline (Harris, 2005), however the SIMD units are partitioned into a very small set of MIMD modules, which are sometimes called MSMID (*multiple SIMD*) or SIMT (single instruction multiple *threads*). This abstraction means that, while some basic forms of GPU and GPGPU programming can support

more general MIMD (Dietz & Young, 2010), the majority of performance gains are from *single instruction* streams operating on vector lists. There are many problems that cannot be efficiently ported to vector lists (*vectorization*) which implies that parallel programming is fundamentally different from writing sequential algorithms. Modern compilers are still at the first stages of vectorization (Yang, Xiang, Kong, & Zhou, 2010), where *algorithmic problems* such as sparse volumetric deformation must be explicitly designed from the beginning to target parallelism, in contrast to most optimization techniques that can be applied later.

Designing algorithms for good SIMD ultimately requires an additional focus on eliminating the *dependencies* between data, which has never been a problem for sequential algorithms. This presents a challenge for shape deformation as surfaces are modeled by multiple adjacent regions, whose deformation typically requires *neighbor access* to reconnect or resample the primitives (in rasterization filling is performed by the hardware rasterizer, where vertices are naturally indexed in triangle connectivity vector lists). Ray tracing cannot efficiently utilize the hardware rasterizer to resample deformed volumetric content, as secondary lighting (bounced rays) require a new rasterized view at each bounce, which is too expensive to compute naively. Therefore some alternative properties of the neighboring data need to be *assumed* in order to maintain surface adjacency, where the method presented in this thesis exploits the *regularity* property between voxels for a wide SIMD implementation. By assuming voxels are distributed regularly, it is possible to determine their extents independently under the new deformation, preserving adjacency with neighbors without accessing their updated data.

Data dependency is also an issue for the dynamic hierarchical spatial data structures required by sparse volumetric deformation (see *Tree Structures*) as tree nodes *depend* on their parent nodes. To support dynamic updates, the existing tree can be reused with partial updates each frame, however this can deteriorate over time creating fragmentation and poor traversal performance (Wald, Ize, & Parker, 2008). The other approach avoids degradation by completely reconstructing the tree each frame, which is more suited for GPU architectures (Lauterbach et al., 2009; Pantaleoni & Luebke, 2010). In real-time rendering, performance is critical, however the majority of parallel methods generate hierarchy sequentially, which limits the parallelism and scalability by the location in the tree (Garanzha, Pantaleoni, & McAllister, 2011; Zhou, Gong, Huang, & Guo, 2011). In contrast, recent work by (Karras, 2012) is the first to present a fully parallel hierarchy generation scheme. This *breaks the dependency* between nodes by assuming some properties about the node orderings: that they are ordered along the z-order space-filling curve, for example with a parallel radix sort on the Morton codes. This has the advantage of improving cache coherency (see *Spatial Hashing* and *Program Optimization*), and also fits the SIMD architecture of modern GPUs, scaling linearly with the number of cores.

In conclusion, this section has united several topics in the previous sections under the theme of *real-time* rendering, with a strong emphasis on techniques and data structures that are well-suited for the target machine architecture. To fully utilize SIMD parallelism, a pattern has been observed to eliminate data dependencies by assuming some other properties of the underlying data. In the case of deformation the data *regularity* can be exploited, whereas in the case of spatial data structures the data *ordering* can be exploited. These properties compliment the goal of regular, cache-friendly, sparse, hierarchical volumetric deformation.

## 2.6 Chapter Summary

This chapter has presented state of the art literature from four large research areas, which are important to enable real-time rendering in massive dynamic volumetric scenes.

### Conclusion

The section on *Geometric Modeling* examined different shape representations, and concluded that a volumetric definition is more flexible and realistic than popular polygon or parametric approaches. This was followed by a section on *Modeling Deformation*, which looked at the many different techniques used to manipulating content, organize constraints, and simulate animation. It was observed that general hierarchical models extend the range of applications as they enable objects to be simulated at different levels of physical realism according to the region of interest, and because they enable shape features to be intuitively manipulated with relative progressive constraints, for example with a control skeleton. The section on *Spatial Partitioning* examined different categories of spatial data structure in order to accelerate real-time rendering, or other applications such as collision detection. In particular spatial hashed data orderings enabled efficient construction of cache-friendly octree structures on the GPU.

The last section examined literature about *Real-time Rendering*, and united several topics in the previous sections with emphasis on the efficiency and accuracy of the output images. The section discussed how light transport is modeled in both rasterization and ray tracing based methods, and it also looked at how to render scenes that are too large to fit into memory (using an out-of-core algorithm). The final topics explored modern computer architectures, emphasizing the importance of utilizing memory hierarchy and designing algorithms to be parallel from the beginning. This concluded with the idea that the dependency between data needs to be eliminated, for example by assuming specific data orderings and regularity for dynamic hierarchical volumetric content.

## Strategy

Efficiently deforming and rendering massive amounts of volumetric content is a *large* problem: (1) large numbers of calculations are required each frame, and (2) there are many parts to the problem. These parts are: (i) content selection, (ii) content deformation, (iii) ensuring voxel adjacency after deformation, (iv) hierarchy construction, (v) rendering content, and (vi) algorithmic optimizations, for example: (a) level of detail, (b) frustum culling, (c) occlusion culling, (d) temporal coherency, (e) instancing, and (f) empty-space leaping.

Modern graphics hardware contains thousands of small efficient cores which are designed for parallel performance. The strategy for this thesis is to break the large problem into lots of *independent* smaller problems that can be solved simultaneously. The smallest element in a volumetric image is a single voxel, where the challenge of this work is to eliminate the dependencies between the voxels, such that they can be processed in wide SIMD hardware. This follows the approach by (Karras, 2012) who demonstrate hierarchy construction in parallel by eliminating the dependencies between the data. In particular, their method assumes and exploits some other data properties, which are inexpensive to enforce using the GPU. In Chapter 1, and in the literature section on *Parallelism*, a strategy was discussed to exploit the property of *regularity* between voxels such that voxels can independently calculate their own deformed region to preserve adjacency with their assumed neighbors. Each voxel can therefore independently fill this region by either emitting new voxels, or by changing its own hierarchy (which implicitly resizes the voxel). In the next chapter both of these approaches are considered, and the results are examined to find the best usage scenario for each solution. Another problem is the selection of volumetric content. Even with SIMD, processing every voxel in a massive scene is too expensive. In the literature, the out-of-core strategy by (Crassin, 2011) effectively exploits the property of *temporal coherence*, by having the GPU request data from the CPU only as it is needed. This approach is much more difficult to extend to dynamic scenes; however the property of temporal coherence can be combined with the idea of *aggregating* sections of multiple octrees (see *Out-of-Core Algorithm*) in preparation for an efficient content selection algorithm.

In conclusion, there are many properties in volumetric data that can be used or enforced to enable parallelism in real-time sparse volumetric deformation and rendering.

## Further Reading

Further reading about the research areas in this chapter are in the books (Akenine-Möller et al., 2008; Engel, 2006; Samet, 2006; Wald et al., 2009). The reader is also referred to (Ericson, 2005) for extensive practical examples of optimized cache-friendly data structures.



# Chapter 3

## Deformation

### 3.1 Abstract

This chapter addresses the problem of deforming massive amounts of volumetric content in real-time on current hardware. The problem is challenging as large numbers of calculations are inevitable, especially in scenes with thousands of dynamic objects, and therefore important calculations need to be prioritized according to the target hardware. In the literature review, many prioritization approaches were discussed, and a strategy was presented to eliminate the dependencies between data for efficient parallel processing. This can be achieved by enforcing a localized cache-friendly data ordering and designing an independent algorithm that utilizes the properties of regularity and temporal coherence, in combination with modern abstractions such as sparse voxel octree *hierarchy* and shape *instancing*.

### 3.2 Methodology

Chapter 1 highlighted the problems when resampling and constructing hierarchy for deformed volumetric content. In the literature, (Crassin, Neyret, Sainz, Green, & Eisemann, 2011) present a real-time voxelization and pre-filtering scheme of animated triangle meshes, however this is not scalable and always requires a high-resolution voxelization process each frame (Crassin, 2011). Instead, this section investigates directly deforming the hierarchical volumetric shape regions as part of a top-down parallel *selection* process. This only processes the contributing shape regions, instead of the entire shapes, at the resolution actually required for rendering. Therefore the *resampling* and *hierarchy construction* processes can be applied more efficiently later, to the selected and deformed regions, instead of the entire high-resolution shapes. This approach also maps well to the hierarchy of hardware, as illustrated in Figure 12:

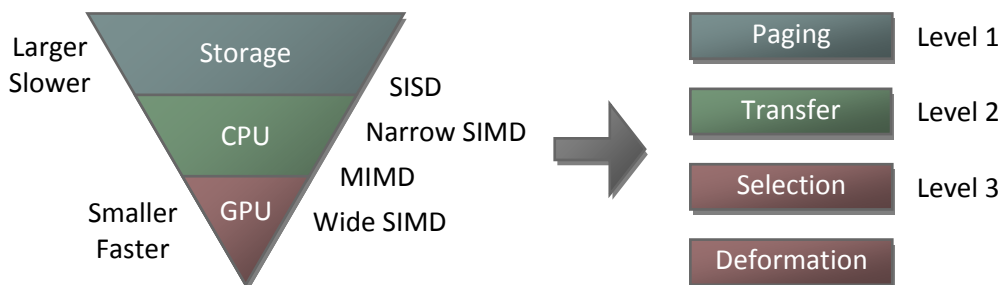


Figure 12: The hierarchy of hardware maps to the hierarchy of content selection.

The idea is to *prioritize* the deformation calculations by only processing important content, and also to *parallelize* deformation such that many elements can be processed each frame. Priority is assigned implicitly with a parallel hierarchical selection process (Figure 12 right) that utilizes the architecture of current hardware (Figure 12 left). The *selection* process requires the shapes to be represented at multiple geometric resolutions, such as with input SVOs: if a branch voxel is unimportant, then its child voxels are also unimportant as the branch spatially contains all of its children. Therefore large shape regions can be inexpensively selected and discarded (Figure 12 level 1, level 2) with progressive refinement (Figure 12 level 3 onwards) until only the important data remains (Figure 2) for *resampling* and *hierarchical construction* processes. In order to maximize parallelism, all of the paged shapes can be processed simultaneously, using *instancing* to avoid heavy data transfers on repeated attributes. In contrast, the dynamic attributes can be organized with other abstractions, such as *control structures* (Figure 2), giving support for more efficient and intuitive shape manipulation.

Parallelizing volumetric deformation is challenging, as dependencies between shape regions need to be eliminated in order to avoid otherwise expensive synchronization. Deformation can cause regions of the shape to stretch, creating gaps that need to be filled with new samples. This is usually addressed by accessing the neighboring shape regions and using a filling algorithm, such as rasterization, however this approach limits the rendering as discussed in the next chapter. Instead, each voxel can independently assume that it shares geometric data with its neighbors, and use this information to ensure visual continuity without accessing their data. In particular, the voxels can either resize themselves by changing their display depth, or emit new voxel samples. This will be discussed in more detail in the later section on *Resampling*.

After selecting important voxels and independently deforming them, the spatial partitioning hierarchy needs to be constructed to enable efficient traversal for applications such as collision and real-time rendering. This is also achieved in parallel, with an approach similar to (Karras, 2012). The voxels are efficiently sorted along the z-order space-filling curve, using a parallel radix sort on the interleaved values of their deformed centers. This improves cache coherence and enables SVO hierarchy to be quickly calculated from a parallel binary radix tree on the sorted data. The output is therefore a single SVO for accelerated real-time rendering, which captures all of the important shape regions from the input shapes that contribute to the scene.

In conclusion, the proposed deformation pipeline introduces a parallel selection process which prioritizes calculations according to the hierarchical volumetric resolution. The voxel data is also independently deformed in parallel, utilizing the wide vector width SIMD architecture of the GPU, without being interrupted by large data transfers. This is shown in the abstraction in Figure 13, which is discussed in more detail in the following sections.

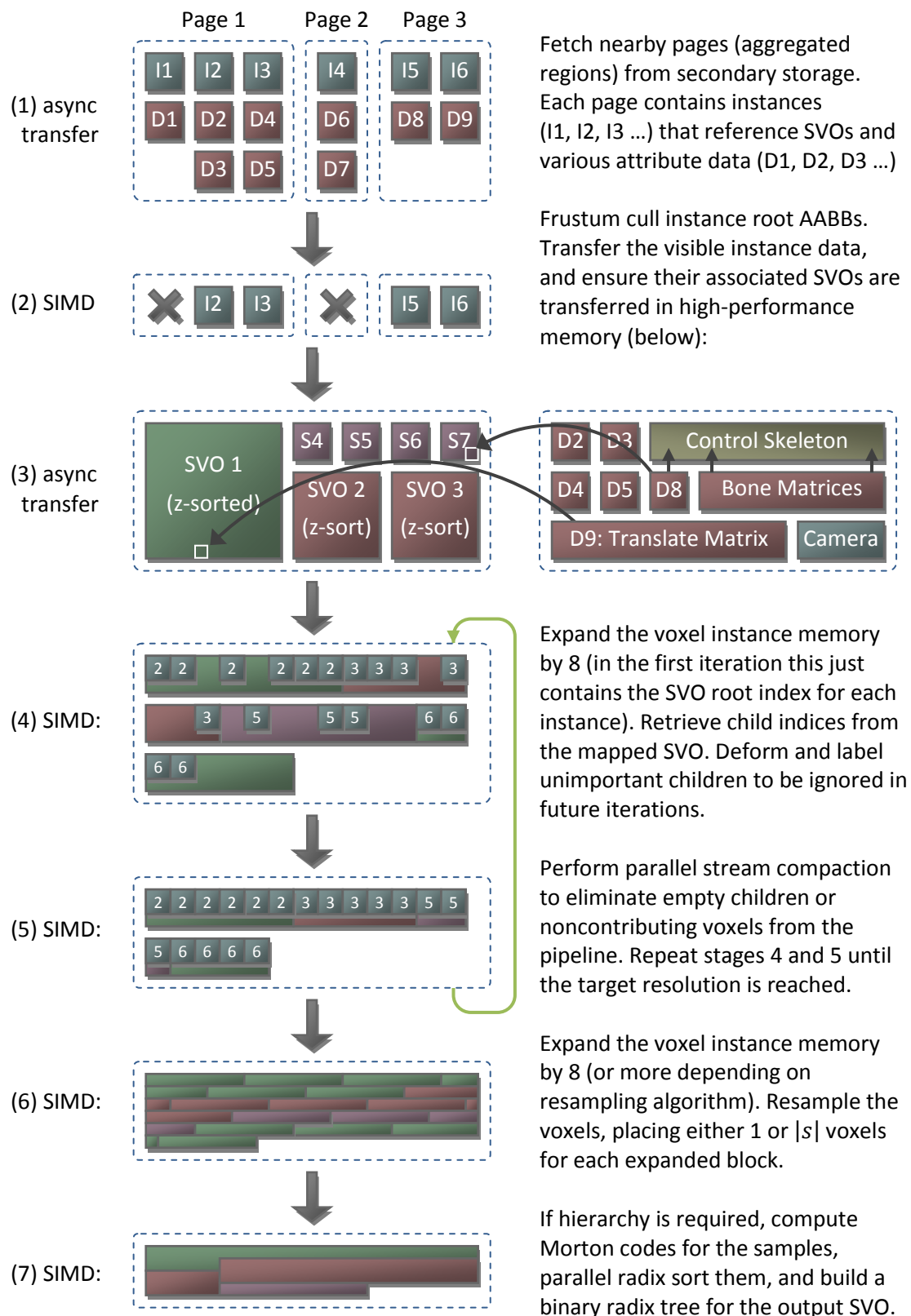


Figure 13: Abstraction of the proposed pipeline stages.

The abstract diagram in Figure 13 shows the complete pipeline of the proposed deformation method. This is organized into seven stages, of which 1 and 3 are asynchronous and occur across multiple frames, and the remaining operates in high-performance wide SIMD. This means that the majority of tasks efficiently utilize parallel hardware (Figure 12). These tasks are: *paging* scene data from secondary storage (stage 1), content *selection* and transfer (stage 2-5), *deformation* (combined with selection in stage 4), *resampling* (stage 6), and finally the task of reconstructing hierarchy for accelerated shape queries for the application of rendering (stage 7, which is discussed in the next chapter). The following sections in this chapter discuss the proposed method in more detail according to these tasks.

## Paging

Paging involves the retrieval of information from secondary storage for use in main memory. The input for the pipeline is a scene format, shown in stage 1, consisting of large aggregated regions of *object instances*, which are small structures that reference a SVO and also store any unshared attribute data, such as unique transformations. This format means that repetitive content, such as blades of grass or piano keys, can be modeled easily and efficiently; the scene can be constructed by the artist placing and transforming object instances (modifying their transformation matrices) and also specifying other attribute data that may be updated in real-time, for example shading parameters. Each instance references a SVO, whose hierarchical voxel data stores shared attributes such as colors, material properties, or a mapping between features in a control skeleton (whose geometry can be stored per-instance). The instances are aggregated into groups called pages, which are retrieved in an asynchronous *Out-of-Core Algorithm* according to the ROI. The amount of aggregation depends on the distribution of data stored in the scene, and therefore requires profiling to find suitable page or region sizes. For example, a large space simulation will benefit from sparser page groupings than an indoor scene. The proposed paging approach retrieves any data which is referenced by these grouped instances (which are discussed in more detail in the later sections) which means that assets are stored separately on disk, and can therefore be maintained easily by artists.

## Modeling

With the *instancing* abstraction discussed in the previous paragraph and also in Figure 2, the scene can be modeled by either creating new instances, or by updating the unshared instance attribute data (D1, D2, D3, ...) each frame. The input shapes are stored on disk as separate SVO files, which represent geometry as volume elements (voxels) in the hierarchical branches and leaf nodes of an octree structure. High-resolution shapes can be created by artists either voxelizing surfaces (see Chapter 5 Applications), or by setting voxels through CSG operations (Voelcker & Requicha, 1977), virtual sculpting (Galyean & Hughes, 1991), or by procedural generation (Ebert, 2003). The voxels contain material attributes and they also store a mapping

to deformation abstractions, such as the index of matrices in a control structure. This mapping is discussed in more detail in the later sections (Figure 14 and Figure 16) where the main idea is that each voxel only needs to be updated by two matrices, one for each of its extreme points, instead of for all eight of its corner vertices. Deformation is therefore modeled by updating the mapped matrices, which are organized with a control structure. The control structure can either be stored as a shared attribute, requiring fewer updates but giving shared animation across instances, or it can be stored as an unshared attribute for each instance, giving unique animations. This instanced modeling approach reduces the amount of transfers, which is important for large amounts of similar animations, such as for grass waving in a field.

## Selection

The selection process is essential for supporting efficient deformation. The purpose is to select volumetric content such that deformation calculations are only applied to contributing regions. Therefore the proposed method deforms shape regions *inside* the selection method (stage 4) instead of popular parallel approaches that first deform the entire shape and then later discard unimportant regions by clipping. This means that less data is processed, which is important for scalability in massive scenes. The selection process spans three levels (i to iii) that map to the hierarchy observed in modern hardware (Figure 12). These levels are: (i) the out-of-core algorithm (stage 1) which retrieves pages from secondary storage (previous section), (ii) the instance root cull and GPU transfer (stages 2-3), which sends data to high-performance memory and quickly discards instances if their root voxels, containing all other voxels, are unimportant, and (iii) parallel selection and deformation (stage 4-5) which efficiently traverses the mapped SVOs and selects the important volumetric content before resampling in stages 6-7. The three selection levels work in unity to achieve two goals: (1) to reduce the amount of voxels for processing, and (2) to reduce the data transfer between memory hierarchies. The method of these levels is discussed in more detail in the following paragraphs.

In stage 2-3, the root voxel for each instance is frustum culled, and any out-of-date SVO or instance attribute data is transferred to high-performance memory. Stages 4-5 then proceed in a top-down hierarchical selection process, which retrieves the important volumetric data from each of the instances in parallel. This is achieved iteratively in two wide SIMD processes: stage 4 traverses to the next level of hierarchy in parallel (by expanding the input vector by 8 and processing the children), and stage 5 then compacts the memory such that only the important voxels are processed in future iterations. In order to prioritize the volumetric content, stage 4 labels the children: (a) to be removed, if the child does not exist in the SVO or if it does not contribute to the output simulation, (b) to be skipped for processing in further iterations, if its projected size is too small, or otherwise (c) for further traversal in the following iterations. These predicates are summarized in Table 1, and are referenced in the following paragraphs.

Stage 5 then performs parallel stream compaction (Billeter, Olsson, & Assarsson, 2009) which removes unwanted elements from the vector by the predicate (a). This is important as the expanded memory would otherwise rapidly become unmanageable by the Stage 4 expansion.

Label Voxels		Predicates
(a)	for removal (compaction)	<i>if the child voxel does not exist in the SVO or if it does not contribute to the simulation</i>
(b)	to be ignored in the future	<i>if its projected size is too small (insignificant)</i>
(c)	for continued traversal	<i>otherwise</i>

Table 1: Selection criteria during top-down hierarchical traversal.

The conditions for predicates (a) and (b) are calculated by deforming the input voxels with two matrices mapped to their extreme points (this is more efficient than deforming their 8 corner vertices, discussed in detail later at Figure 16), and projecting the new  $2^n = 8$  AABB corners to screen-space. The 2D bounding-rectangle is then calculated for the projected voxel, which is used to determine the projected size of the voxel. If the size is smaller than a single pixel, it is classified as too small (predicate *b*). Also, the corners of the 2D bounding rectangle are used to sample a mipmapped hierarchical depth map, retrieved from a rendering (discussed in the next chapter on *Rendering*). If the voxel is not visible (obscured or not in the frustum), it is labeled for removal (predicate *a*) and eliminated during stream compaction in Stage 5. The details for these calculations are described more formally in the following paragraphs.

In order to project the input deformed voxel AABB (see Figure 16) with  $2^n = 8$  corner points  $\mathbf{q} \in \mathbb{R}^4$  where ( $\mathbf{q}_w = 1$ ) to screen-space, its corners  $\mathbf{q}^1, \mathbf{q}^2, \dots, \mathbf{q}^8$  are first transformed by the  $4 \times 4$  camera matrix  $\mathbf{M}$  (the view-projection matrix) in Equation (3.2.1). These 8 transformed corner points  $\mathbf{p}^1, \mathbf{p}^2, \dots, \mathbf{p}^8$  are then mapped back to the real plane with a perspective divide, making them relative to the camera in homogeneous space, where they are transformed from the frustum cube  $[-1,1]$  to the screen region  $[0,1]$  by Equation (3.2.2):

$$\begin{bmatrix} \mathbf{p}^1 \\ \mathbf{p}^2 \\ \vdots \\ \mathbf{p}^8 \end{bmatrix} = \mathbf{M} \begin{bmatrix} \mathbf{q}^1 \\ \mathbf{q}^2 \\ \vdots \\ \mathbf{q}^8 \end{bmatrix} \quad (3.2.1)$$

$$\mathbf{p}' = \begin{bmatrix} 0.5(\mathbf{p}^1/\mathbf{p}_w^1) + 0.5 \\ 0.5(\mathbf{p}^2/\mathbf{p}_w^2) + 0.5 \\ \vdots \\ 0.5(\mathbf{p}^8/\mathbf{p}_w^8) + 0.5 \end{bmatrix} \quad (3.2.2)$$

The 2D bounding rectangle and the voxels minimum depth value can then be calculated from the minimum and maximum extreme points for the 8 corresponding corners  $\mathbf{P}'$  projected to the screen by Equation (3.2.2). The projected size of the voxel (in pixels  $p_{\text{size}}$ ) is calculated from the maximum side of the rectangle, and also the minimum depth  $v_{\text{depth}}$  can be computed from the minimum projected corner, as in the following two equations:

$$p_{\text{size}} = \max \left( \begin{array}{l} \text{screenPixels}_{\text{width}} \cdot (\max(\mathbf{P}')_x - \min(\mathbf{P}')_x) \\ \text{screenPixels}_{\text{height}} \cdot (\max(\mathbf{P}')_y - \min(\mathbf{P}')_y) \end{array} \right) \quad (3.2.3)$$

$$v_{\text{depth}} = \min(\mathbf{P}')_z \quad (3.2.4)$$

$$p_{\text{size}} < 1 \quad (3.2.5)$$

Therefore the predicate (b), for ignoring future traversal, is given where  $p_{\text{size}}$  is less than a single pixel (3.2.5) (as its future child voxels will be too small to contribute).

The predicate (a) for removing occluded voxels from the deformation pipeline follows the popular hierarchical z-buffer visibility approach. This involves sampling the depth buffer (see Chapter 4) according to the projected 2D bounding rectangle, and rejecting a deformed voxel if it is behind all the samples. The hierarchical z-buffer greatly reduces the number of samples required, as large 2D regions can be quickly tested by sampling less pixels at a lower image resolution, which contain the depth information stored in the higher-resolutions. This process is described more formally. Given an  $n \times m$  depth image as input, a hierarchical mipmap is generated on the GPU in a bottom-up SIMD process. This results in  $\lceil \log_2 \max(n, m) \rceil + 1$  images of decreasing resolution  $n' = \frac{n}{2} \times m' = \frac{m}{2}$  until the last  $1 \times 1$  image is generated.

Each new image is half the resolution of the previous, and therefore each pixel overlaps a region of  $2 \times 2$  pixels from the previous iteration. The intensity for each pixel is calculated by the maximum value of this  $2 \times 2$  neighborhood, which means that the low-resolution pixels of the hierarchical mipmap *contain* the largest depths of all their overlapping regions. Therefore, the large projected voxels with a high  $p_{\text{size}}$  can sample fewer pixels at a lower resolution, which is very efficient. By this observation, the predicate (a) for removing occluded voxels can be determined efficiently by sampling from the mipmap level  $l$  which encloses (ceil)  $p_{\text{size}}$ . However it is not possible to simply sample one pixel at the projected 2D rectangle center  $0.5(\max(\mathbf{P}') + \min(\mathbf{P}'))$ , as the rectangle may overlap the boundary of the  $2 \times 2$  partitions. Therefore the depth samples  $\mathbf{Z}$  are retrieved at the four corners of the 2D rectangle (whose extreme points are  $\min(\mathbf{P}')_{xy}$  and  $\max(\mathbf{P}')_{xy}$ ) using the next  $(p_{\text{size}}/2)$  mipmap level  $l$ , where:

$$l = \left\lceil \log_2 \left( \frac{p_{\text{size}}}{2} \right) \right\rceil \quad (3.2.6)$$

In order to use the depth buffer from the previous frame, the spatial-temporal incoherency needs to be considered. This is achieved by adding the deformed voxel size to the difference between the matrix of the previous frame  $\mathbf{M}'$  and the current frame  $\mathbf{M}$  (or simply the 3D Euclidean distance between  $\hat{\mathbf{w}}\mathbf{M}'$  and  $\hat{\mathbf{w}}\mathbf{M}$  where  $\hat{\mathbf{w}} = (0,0,0,1)$ ). This means that the predicate (a) for removing occluded voxels is where the minimum depth of the projected voxel  $v_{\text{depth}}$ , in Equation (3.2.4), is behind all the depths  $\mathbf{Z}$  ( $v_{\text{depth}} > \max(\mathbf{Z})$ ) offset as shown:

$$v_{\text{depth}} > \max(\mathbf{Z}) + \max \left( \begin{array}{l} \max(\mathbf{P}')_x - \min(\mathbf{P}')_x \\ \max(\mathbf{P}')_y - \min(\mathbf{P}')_y \end{array} \right) + \text{dist}(\hat{\mathbf{w}}\mathbf{M}', \hat{\mathbf{w}}\mathbf{M}) \quad (3.2.7)$$

In conclusion, the hierarchical selection process (Figure 13 stages 1-5) operates efficiently in parallel for all important voxels in the scene. The input is set of aggregated instances, which are mapped to SVOs and various attribute data, and the output is the visible set of instance-mapped voxels whose hierarchical resolution is determined by the projected voxel size. It was briefly discussed that deformation occurs *inside* the selection process (Figure 13 stage 4) by transforming the input voxels AABBs according to their mapped matrix data (see *Modeling*). This means that deformation is only applied to contributing volume elements at the resolution of which the data is actually displayed (Table 1) which is scalable.

## Deformation

Volumetric deformation involves changing a volumetric shape into a new volumetric shape. To efficiently parallelize volumetric deformation, the most primitive volumetric elements (voxels) are processed independently through wide SIMD hardware (Figure 13 stage 4-5). During the selection process (previous section) voxels are deformed by arbitrarily transforming the two extreme corners of their AABBs with pre-mapped 4D matrices such as in a control skeleton (Kavan et al., 2009) (this is more efficient than transforming their 8 corners, and is discussed in detail later at Figure 16). The control matrices are concatenated in a hierarchical model (see *Hierarchical Modeling*) with the instance matrix (the model matrix) (Lengyel, 2011). This approach reduces the amount of matrix transformations per voxel, and means that animation can be efficiently modeled by updating the instance and feature-mapped matrices each frame, by changing the high-level attribute data in Figure 13 stage 3 (right) instead of by expensively manipulating each individual voxel (discussed in the previous section on *Modeling*).

In order to efficiently query content, for example in real-time rendering or collision detection applications, a single hierarchical acceleration structure is needed for all of the shapes in the scene, otherwise the shapes need to be iterated for each query, which is not scalable with



large numbers of queries for massive amounts of objects, such as in ray casting. Previously, the advantages were discussed for representing input shapes with multiple separate SVO files (see *Chapter 1* and *Modeling*). Applying deformation to the input shapes voxels will quickly cause their quadrangular faces to become nonplanar, which is extremely difficult and expensive to test for intersection, and results in curved line segments instead of proper edges (Engel, 2006). Therefore, to address this problem, the updated AABB is recalculated for the deformed voxel, which is then resampled to the structure of a single output SVO that encloses the region of the entire scene. Although it is possible to consider resampling the deformed OBB of each voxel, and generating hierarchy for other types of hierarchical structure, the proposed AABB and SVO combination has three main advantages: (1) the axis-aligned test for intersection is much more efficient than the test for oriented boxes, (2) OBBs need an extra matrix multiplication for combining the OBB rotation matrix with the transformation matrix, and (3) OBBs also require extra memory for storing the orientation matrix (Ericson, 2005). Also, the octree structure of the target SVO is well-suited for parallel construction (Karras, 2012) on the resampled points, which is discussed in the next chapter. In summary, the shape queries can be made efficiently with a single SVO traversal, instead of through expensive iteration of the selected shapes.

In order to calculate the deformed AABB for each selected input voxel, the architecture of the selection and deformation *processes* needs to be considered according to the instancing and control structure *abstractions* (see Figure 2). This architecture is presented in Figure 14, which shows these stages of the pipeline (Figure 13 stages 4-6):

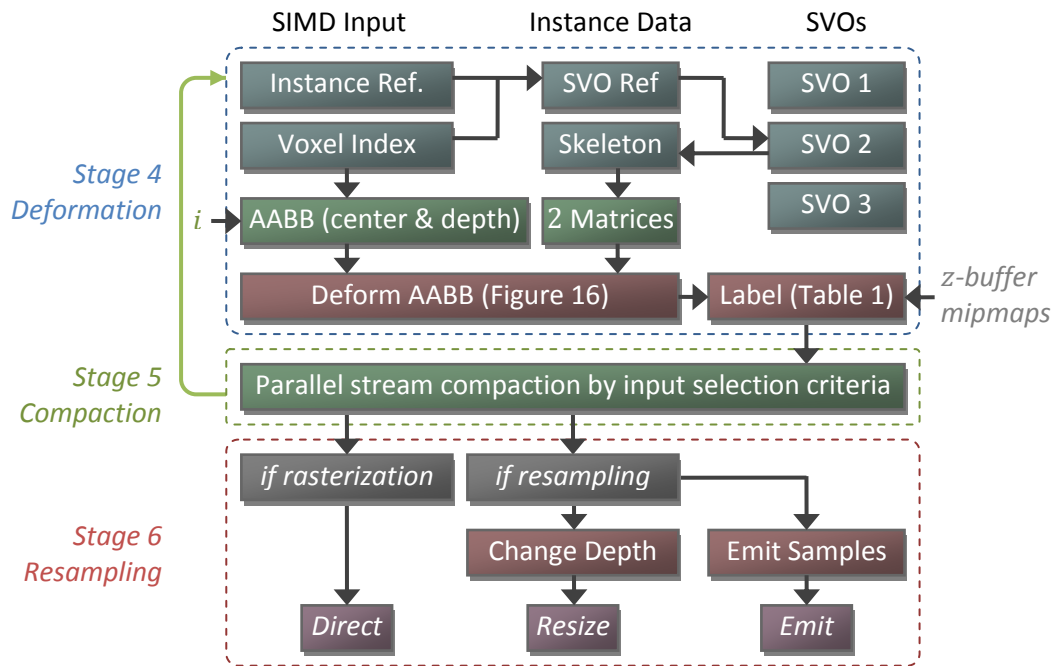


Figure 14: Architecture of the parallel selection, deformation, and resampling processes.

Figure 14 shows how volumetric information is organized by the *instancing* abstraction (top-middle), which references *hierarchical* SVO input shapes (SVO 1, SVO 2, SVO 3, etc., where the example references SVO 2), and retrieves the voxels and their matrices mapped by skeleton control structures, stored as instance attributes. The architecture only requires a small amount of data for each SIMD input element (top-left), and applies the *abstractions* discussed in Figure 2 for fully parallelized selection and deformation *processes*. Therefore the deformed AABB can be efficiently recalculated without expensively storing multiple copies of any shared attributes (this process is discussed in more detail in the next section). The selection process (previous section) also occurs in Figure 13 stage 4, immediately after the deformed AABBs have been calculated: the SIMD input elements are labeled according to Table 1, which are compacted in Figure 13 stage 5, such that future traversal iterations  $i$  only process the contributing shape regions until the target resolution is reached. Therefore only the important volume elements are deformed, as the entire algorithm operates iteratively in parallel for each hierarchical level on progressively refined shape regions (as outlined in Figure 2).

In Chapter 1, it was briefly discussed that deformed voxels (the output after stage 5) can be displayed *directly* using a rasterized rendering approach. This is both *accurate* and *efficient*, as no resampling or hierarchical construction process is required; the selected and deformed voxel primitives can be rasterized using graphics hardware to set their corresponding pixels in the output image accordingly. However this approach is limited as shapes cannot be queried efficiently without some organization of the selected volumetric data, and therefore many applications (such as modeling light transport or detecting collisions) cannot be implemented for large scenes in real-time. To address this problem, *resampling* and *hierarchy construction* stages are introduced (Figure 13 stages 6-7). These allow applications to query shapes in an efficient and scalable real-time traversal process, operating on the structure of a single output SVO as discussed in the previous paragraphs. However, while this has extended applications, it requires more operations and the resampling process impacts quality, when compared to the unaligned *direct* rasterization (Figure 14 *Direct*). Therefore, both approaches are considered (Figure 14: ‘*if rasterization*’ and ‘*if resampling*’) and developed in the later parts of this thesis. This means that more flexible applications can be created, maximizing the requirements and rendering quality, for example by combining high-resolution rasterized direct illumination with lower-resolution SVO hierarchy for accurate output images and efficient and shape queries.

The resampling process aligns the voxels to the structure of an output SVO; otherwise artifacts occur (see Figure 4). In Chapter 1, two resampling solutions were discussed: (1) by resizing the aligned voxels (Figure 14 *Resize*), and (2) by emitting multiple samples (Figure 14 *Emit*). These approaches are developed in the upcoming section on *Resampling*, which is followed by a *Measurement* section that compares the two outputs with the unaligned *direct* rasterization,

and empirically determines the best usage of each scenario. The results of this section are then developed into a practical *Decision Tree* based on the application usage requirements.

### 3.3 Resampling

The resampling approach addresses the problems of: (1) efficiently updating the AABB of each voxel with the mapped deformation data, (2) determining the correct display depth to resize voxels, and (3) determining the location for where gaps in the shape occur. The parts relate to the outputs in Figure 14, which are accordingly named: (1) direct, (2) resize, and (3) emit.

#### Direct

In the previous section (Figure 14 ‘*Deform AABB*’) it was discussed that the hierarchical voxel data can be retrieved from the input SVOs and deformed using matrix data that is mapped to a control skeleton. This section is concerned with the process of efficiently deforming each input voxel AABB by its mapped matrix data, in order to create an updated *AABB* that can either be *directly* rasterized, or serve as input for the *Resize* and *Emit* (Figure 14) resampling processes. This is therefore applied in parallel to individual voxels that are currently *selected* and labeled as important (Figure 13 stage 4-5), such that the output after the hierarchical selection process (after Figure 13 stage 5) contains all of the contributing deformed voxels in the scene.

The input (Figure 14 ‘*Deform AABB*’) is a voxel and some matrices mapped to its corners (this mapping is discussed at the end of this section). The voxel is defined by an  $n = 3$ -dimensional point (its center)  $\mathbf{c}$  and a SVO *depth* value  $d$ , which is the current iteration  $i$  in the hierarchical selection process  $d = i$  (Figure 14 stage 4-5). The input SVO shapes regularly divide space into a maximum of  $2^d$  partitions at each LOD depth  $d$ . This means that each voxel can be represented by an AABB whose side  $s$  at depth  $d$  is definable as  $1/\text{maximum partitions}$  or:

$$s = 2^{-d} \quad (3.3.1)$$

Where  $\mathbf{B}$  is the voxel AABB with center  $\mathbf{c}$  and extents  $\mathbf{t}$  (Equation (3.3.3)). This may alternatively be represented as shown in Equation (3.3.2), or more simply Equation (3.3.4):

$$\mathbf{B} = \begin{bmatrix} \mathbf{b}_{\min} \\ \mathbf{b}_{\max} \end{bmatrix} = \begin{bmatrix} \mathbf{c} - s/2 \\ \mathbf{c} + s/2 \end{bmatrix} \text{ or alternatively } = \begin{bmatrix} \mathbf{c} - 2^{-(d+1)} \\ \mathbf{c} + 2^{-(d+1)} \end{bmatrix} \quad (3.3.2)$$

$$\mathbf{t} = \underbrace{(2^{-(d+1)}, 2^{-(d+1)}, \dots, 2^{-(d+1)})^T}_n \quad (3.3.3)$$

$$\mathbf{B} = \begin{bmatrix} \mathbf{c} - \mathbf{t} \\ \mathbf{c} + \mathbf{t} \end{bmatrix} \quad (3.3.4)$$

Transforming an AABB  $\mathbf{B}$  by some input matrix  $\mathbf{A}$  is traditionally achieved by transforming each of the  $2^n = 8$  AABB corner vertices in 3D, and recalculating the extreme points  $[\mathbf{b}_{\min}, \mathbf{b}_{\max}]$  as shown for  $n$ -dimensions: (min and max do not consider the  $n + 1$  component in this case)

$$\mathbf{B}' = \left[ \min \left( \mathbf{A} \begin{bmatrix} \mathbf{c}_1 \pm \mathbf{t}_1 \\ \mathbf{c}_2 \pm \mathbf{t}_2 \\ \vdots \\ \mathbf{c}_n \pm \mathbf{t}_n \\ 1 \end{bmatrix} \right), \max \left( \mathbf{A} \begin{bmatrix} \mathbf{c}_1 \pm \mathbf{t}_1 \\ \mathbf{c}_2 \pm \mathbf{t}_2 \\ \vdots \\ \mathbf{c}_n \pm \mathbf{t}_n \\ 1 \end{bmatrix} \right) \right] \quad (3.3.5)$$

However to deform the AABB, a unique matrix would need to be constructed and applied for each of the corners, giving  $2^n$  expensive construction and transformation operations. This also means that a mapping would need to be stored to the matrix data (Figure 13 stage 3 right) for the eight vertices of each AABB, which would consume lots of memory. In the literature, (Arvo, 1990) observe that many of these operations are wasteful because they ignore information embodied in the box symmetry, and propose to utilize this symmetry (Figure 15).

---

```

1   TransformAABB(AABB in, Matrix4x4 A, AABB &out) {
2       for (int i = 0; i < 3; i++) {
3           out.min[i] = A[3][i];
4           out.max[i] = A[3][i];
5           for (int j = 0; j < 3; j++) {
6               float a = A[i][j] * in.min[j];
7               float b = A[i][j] * in.max[j];
8               out.min[i] += Min(a,b);
9               out.max[i] += Max(a,b);
10          }
11      }
12  }
```

---

Efficiently transform AABB  $in$  by matrix  $A$  giving a new AABB  $out$ .

---

Figure 15: Transforming Axis-Aligned Bounding Boxes (Arvo, 1990)

Their method is based on the observation that the components of the transformed box need only consider which of the eight vertices produce the minimum or maximum product with the  $i$ th row of the matrix. The products can therefore be formed for each component and the sum of the largest or smallest terms results in the maximum or minimum values accordingly. Translation does not influence these choices and is added (lines 3-4).

AABB transformation (Figure 15) applies one matrix to each voxel. Therefore it is limited to simple transformations that are applied uniformly to all voxels in the shape, such as translation, rotation, and scale operations. Deformation changes the shape *irregularly* and

therefore requires different transformations for the independent voxels. Extending the observation of (Arvo, 1990) it is possible to formulate deformation as two transformations for the two AABB extreme points, which share components with neighboring voxels, and expand the transformed extremes to ensure full enclosure. This approach means that just two matrix mappings, constructions, and transformations are required for each voxel; instead of the original  $2^n = 8$  mappings (comparable to triangles, which typically require three mappings for each of their vertices). The *direct* deformation method is shown in Figure 16:

---

```

1   DeformAABB(AABB in, Matrix4x4 *A, AABB &out) {
2       AABB cur = in;
3       for (int k = 0; k < 2; k++)
4           for (int i = 0; i < 3; i++) {
5               cur.min[i] = A[k][3][i];
6               cur.max[i] = A[k][3][i];
7               for (int j = 0; j < 3; j++) {
8                   float a = A[k][i][j] * in.min[j];
9                   float b = A[k][i][j] * in.max[j];
10                  cur.min[i] += Min(a,b);
11                  cur.max[i] += Max(a,b);
12              }
13              out.min[i] = Min(out.min[i], cur.min[i]);
14              out.max[i] = Max(out.max[i], cur.max[i]);
15          }
16      }

```

---

Efficiently deform AABB *in* by matrices  $A[0]$  and  $A[1]$  mapped to *in.min* and *in.max*

---

Figure 16: Deforming an AABB with just two mapped matrices for each extreme.

The proposed method branches for each of the two extremes (line 3) and then expands the output AABB to enclose the respective transformed extremes (lines 13-14). The output AABB *&out* must be initialized to  $[b_{\min}, b_{\max}] = [\infty, -\infty]$  for correct bounds in lines 13-14. This therefore ensures that the full extents of the deformed AABB are enclosed and axis-aligned, which means that adjacent voxels, sharing extreme components and undergoing the same algorithm, will also have overlapping axis-aligned enclosure and therefore retained adjacency. The updated output AABB *&out* can be *directly* rasterized (Figure 14) or aligned to the output SVO for ray casting with either the *Resize* or *Emit* strategies in the following sections.

Mapping the voxel AABB extremes  $[b_{\min}, b_{\max}]$  to the two matrices (Figure 14 and Figure 13 stage 3) is similar to mapping two vertices with matrices in traditional polygon rasterization, where each vertex (voxel AABB extreme) stores (in the SVO) the index of its associated matrix. The only difference is that the locations of the vertices are not stored as explicit 3D points, instead they  $[b_{\min}, b_{\max}]$  are constructed (Figure 14: 'AABB (center & depth)') from the voxel center and its depth value  $d$  as in Equation (3.3.2).

## Resize

Applications, such as modeling light transport or collision detection, require huge amounts of shape queries to be made each frame. However currently the selected and deformed output voxels (calculated in Figure 16) are unorganized, and therefore require expensive iteration which limits their applications. To address this problem, it was previously discussed that an output SVO could be efficiently constructed (Chapter 4) on the deformed AABBs (Figure 16), meaning that shape queries can be made with efficient SVO traversal and AABB intersection tests. However without resampling, artifacts in the hierarchical structure occur (see Figure 4).

The key idea of the *resize* strategy, discussed in Chapter 1, is to sample the deformed AABBs and set new voxels at the sample locations with different display depths. This implicitly *resizes* them, preventing gap artifacts and shape distortion, while only inserting a fixed number of samples to approximate the new shape. This approach is therefore extremely efficient, as it does not require expensive memory management; the amount of memory required for the output samples is allocated beforehand (in Figure 13 stage 6). The disadvantage is that the fixed number of samples only gives a cube-shaped approximation of the deformed voxel AABB. However the efficiency means that further oversampling can be considered by selecting higher resolution shape data (predicate *b* in Table 1) to achieve better shape approximation, which is discussed in the next section. The resize strategy is shown in the  $n = 2D$  example in Figure 17:

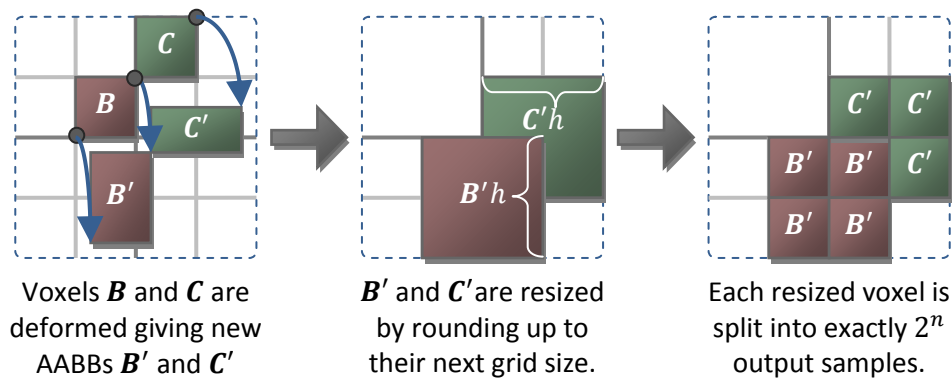


Figure 17: The resize method changes the display depth of voxels, which implicitly resizes them. Each voxel has exactly  $2^n$  samples, giving a fast shape approximation.

Initially (Figure 17 left) the voxel AABBs are deformed (by Figure 16) according to the matrix data (Figure 13 stage 3 right) mapped to each of their two extreme points. The second step (Figure 17 middle) then measures the maximum side of the deformed AABB, and computes the next largest hierarchical level  $h$  which *fits* this side length. This approach addresses the two sampling problems highlighted in Figure 4: large AABBs are enlarged to close *gaps*, or the cell size is reduced for tiny AABBs preventing *distortion*. The new hierarchical level  $h$  is calculated:

$$h = -\lceil \log_2(\max(\mathbf{b}'_{\max} - \mathbf{b}'_{\min})) \rceil \quad (3.3.6)$$

The maximum side  $\max(\mathbf{b}'_{\max} - \mathbf{b}'_{\min})$  of the deformed AABB is used to find the tightest cube-shaped region which encloses the cuboid-shaped AABB (Figure 17 middle). Recall that the side length  $s$  of a voxel is calculated by its hierarchical depth:  $s = 2^{-d}$  (Equation (3.3.1)). This is rearranged to give  $d = -\log_2(s)$ , therefore the *next* largest hierarchical level  $h$  whose region size contains the AABB be computed by  $-\lceil \log_2(\max(\mathbf{b}'_{\max} - \mathbf{b}'_{\min})) \rceil$  (3.3.6).

It is not possible to set a single voxel at hierarchical level  $h$ , which encloses the overlapping SVO regions, because a voxel at level  $h$  may overlap boundaries in the hierarchical structure (Figure 17 middle). This problem can be solved by adding another level of partitioning  $h + 1$ , splitting each voxel into  $2^n$  subvoxels, such that, in cases of overlap, subvoxels fall either side of the structural boundary (Figure 17 right). Therefore the 1D SIMD memory can be enlarged beforehand to  $2^n k$  for  $k$  voxels (Figure 13 stage 6) and the  $2^n$  new sample data is expanded giving contiguous blocks of  $2^n = 8$  indexed elements. The center locations of the new samples  $\mathbf{s}$  at depth  $h + 1$  are aligned to the SVO structure, as shown in the following equations:

$$\mathbf{p} = \frac{(\mathbf{b}'_{\max} + \mathbf{b}'_{\min})}{2} \quad (3.3.7)$$

$$r = 2^{h+1} \quad (3.3.8)$$

$$\mathbf{s} = \frac{1}{r} \begin{bmatrix} \lceil r(\mathbf{p}_1 \pm 0.5/r) \rceil \\ \lceil r(\mathbf{p}_2 \pm 0.5/r) \rceil \\ \lceil r(\mathbf{p}_3 \pm 0.5/r) \rceil \\ \vdots \\ \lceil r(\mathbf{p}_n \pm 0.5/r) \rceil \end{bmatrix} \quad (3.3.9)$$



The locations of the center of the subvoxel samples  $\mathbf{s}$  (Figure 17 right) are calculated from the center of the transformed bounding box  $\mathbf{p}$  in Equation (3.3.7). These sample centers are offset from  $\mathbf{p}$  by a quarter of the region size  $2^{-h}$  at hierarchical level  $h$ , which is  $0.25/2^h$  or more simply  $0.5/r$ . The voxel samples are then aligned to the center of the nearest subregion of  $h$  (Figure 17 right) which is at level  $h + 1$ . Their components  $x$  are aligned by:  $1/r \lceil rx \rceil$ , which scales them to the appropriate region  $r$ , then aligns (floor), and scales back to the nearest center by  $1/r$  (Equation (3.3.9)). This entire process therefore creates  $2^n$  aligned samples with centers  $\mathbf{s}$  at depth  $h + 1$  (Equation (3.3.8)) for the deformed AABB  $\mathbf{B}'$  (Figure 16 &out).

## Emit

The previous resize strategy is based on the principle that a cube-shaped region can enclose the cuboid-shaped region of the deformed voxel ABBB, which means that only a small fixed number of samples are used. The alternative strategy, discussed in Chapter 1, is to *emit* multiple aligned voxels that reflect the cuboid shape of the deformed ABBB. This therefore requires more passes of memory expansion, which is less efficient; however it is expected to achieve better shape approximation. The methods are compared in the next section.

The approach is similar to the resize strategy, apart from three changes which increase the quality of the shape approximation: (1) instead of setting samples from the deformed ABBB center, samples are set between the deformed ABBB extreme points, which better reflects the cuboid shape, (2) voxels are set at higher resolution from the minimum side of the deformed ABBB (not the maximum side), and (3) there is no need to set subvoxels (at level  $h + 1$ ) as the new voxels are set at the same level of the SVO structural boundaries (without overlap). The complete alignment and resampling process is illustrated in the  $n = 2D$  example in Figure 18:

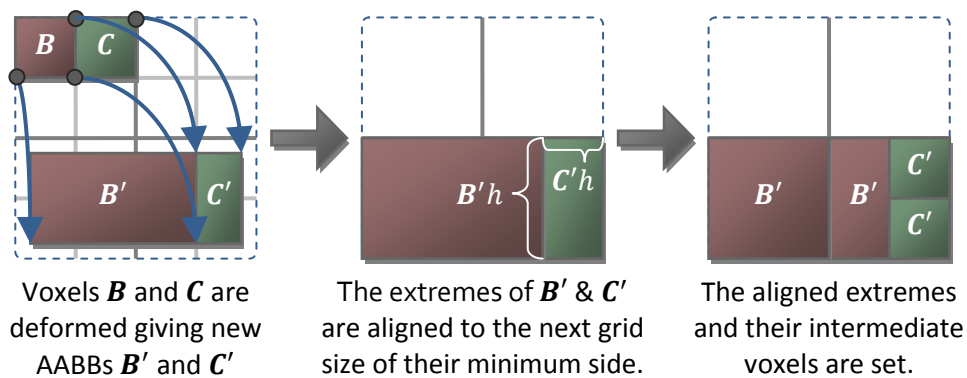


Figure 18: The emit method ‘emits’ multiple samples in a cuboid shape, giving a good approximation of the deformed ABBB, at the expense of memory.

Initially (Figure 18 left) the voxel ABBBs are deformed (by Figure 16) according to the matrix data (Figure 13 stage 3 right) mapped to each of their two extreme points. The second step (Figure 18 middle) then measures the *minimum* side of the deformed ABBB, and computes the next largest hierarchical level  $h$  which *fits* this side length. The two extreme points of each ABBB are then aligned to the corresponding SVO region at level  $h$ , and the intermediate voxels are set. As with the resize strategy, the approach addresses the sampling problems highlighted in Figure 4: gaps are closed where the intermediate voxels are set, and the individual cell size is reduced for tiny ABBBs preventing shape *distortion*.



In (Figure 18 middle) the minimum side  $\min(\mathbf{b}'_{\max} - \mathbf{b}'_{\min})$  of the deformed AABB determines the hierarchical level  $h$  of which to set the output voxels (Figure 18 right). The voxels are set between the aligned extreme points, which are calculated as in the following equations:

$$h = -\lceil \log_2(\min(\mathbf{b}'_{\max} - \mathbf{b}'_{\min})) \rceil \quad (3.3.10)$$

$$r = 2^h \quad (3.3.11)$$

$$\text{Aligned Extremes} = \left( \underbrace{\frac{1}{r} \begin{bmatrix} \lceil r(\mathbf{b}'_{\min_1}) \rceil \\ \lceil r(\mathbf{b}'_{\min_2}) \rceil \\ \lceil r(\mathbf{b}'_{\min_3}) \rceil \\ \vdots \\ \lceil r(\mathbf{b}'_{\min_n}) \rceil \end{bmatrix}}_{\text{Extreme Min}}, \underbrace{\frac{1}{r} \begin{bmatrix} \lceil r(\mathbf{b}'_{\max_1}) \rceil \\ \lceil r(\mathbf{b}'_{\max_2}) \rceil \\ \lceil r(\mathbf{b}'_{\max_3}) \rceil \\ \vdots \\ \lceil r(\mathbf{b}'_{\max_n}) \rceil \end{bmatrix}}_{\text{Extreme Max}} \right)^T \quad (3.3.12)$$

Equation (3.3.10) calculates the hierarchical level of which to set voxels, with similar reasoning to Equation (3.3.6) however it is based on the minimum side of the deformed AABB instead of the maximum, and therefore new samples can overlap with a cuboid shape instead of with a cube shape. Equations (3.3.11) and (3.3.12) align the components of the two AABB extremes  $\mathbf{b}'_{\min}$  and  $\mathbf{b}'_{\max}$ , where all intermediate voxels are set between the two extremes in steps of  $2^{-h}$  (or  $1/r$ ) for each axis. This therefore *emits* multiple voxels in a solid cuboid block between the two aligned extremes. In wide SIMD, it is difficult to determine the amount of memory required beforehand for the multiple emitted samples, and therefore further passes are required to expand and compact the memory for the new samples. Therefore, it is expected that the *resize* strategy is preferable where memory is limited, as it reliably sets exactly  $2^n$  samples per voxel. It is also worth mentioning that the resampled point locations can easily be converted to integer system units, by simply not multiplying the output vectors by  $1/r$  in Equation (3.3.9) and Equation (3.3.12) accordingly, which is useful for accessing the memory.

In conclusion, the *direct* method efficiently deforms the AABB of each input voxel, whose two extremes are mapped to two matrices (Figure 14). The output of this method can be directly rasterized. For efficient hierarchical shape queries, the *resize* and *emit* approaches align and resample the deformed AABB to the structure of a SVO. The methods operate independently using a few operations, and therefore can be computed extremely efficiently with wide SIMD hardware; however they introduce several challenging design decisions which balance memory consumption with the quality of shape approximation. In the next section, the approaches are measured and compared to empirically develop a practical *Decision Tree* for such cases.

### 3.4 Measurement

This section measures the *accuracy* and *efficiency* of the resampled outputs, discussed in the previous section (Figure 14). The purpose of this is to compare and empirically determine the best usage of each scenario, which is developed into a practical *Decision Tree*. This empirical approach is needed as it is difficult to develop a single resampling strategy that balances simulation accuracy with simulation efficiency, as increasing the samples (the *emit* strategy) gives more accurate but less efficient shape approximation in comparison to the fixed sample approach (the *resize* strategy). These methods are also compared with the *directly* deformed output, which can be considered the optimal alignment approach as it is unaligned to the SVO structure and avoids the resampling process; however it does not hierarchically organize the selected voxels, and therefore has limited applications without accelerated shape queries.

#### Accuracy

The accuracy of the resampling reflects the accuracy of the deformed shape, and is therefore an important factor to the level of realism achievable in the virtual environment. In the *resize* and *emit* strategies, the *directly* deformed voxel AABBs are aligned to the SVO structure by *resizing* and *emitting* samples accordingly. The accuracy of these samples can therefore be compared to the original signal, which is the *directly* deformed shape. The sampling accuracy is a metric between this original deformed shape and the new sampling, which is usually taken over the entire population of samples, such as in the application of watermarking where the entire shape is processed. However, in rendering, only a small region of the shape is displayed and also parts of the shape can be considered more influential, as human visual perception is highly sensitive to inconsistencies in the shape silhouette. This is shown by the renderings of both resampling strategies in Figure 19, where the aliasing (sampling error) at the shape silhouette is easy to perceive in the four scenarios. In this section, all images are generated using extremely low-resolution shapes, with the aim to identify sampling error more clearly.

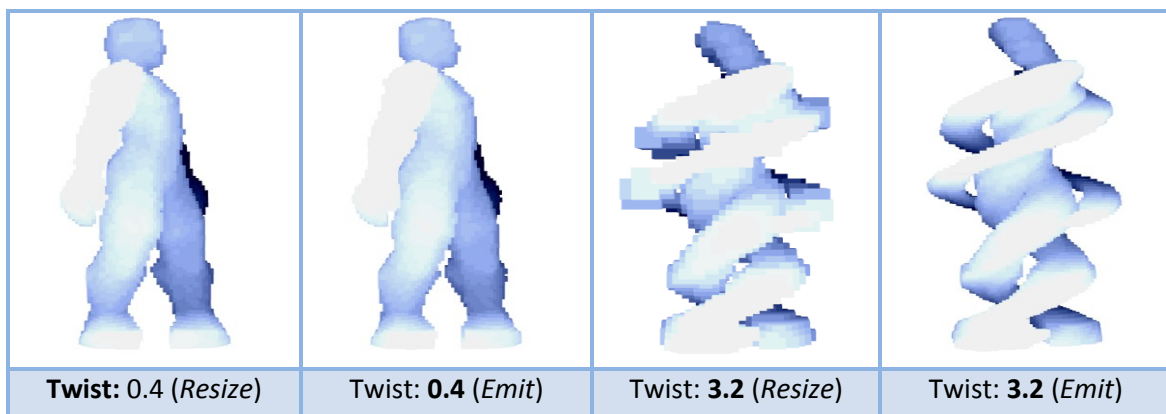


Figure 19: Human visual perception of error is highly sensitive to the shape silhouette.

From this observation, an image-space metric is chosen to better reflect visual perception, instead of using the mean accuracy over the entire sample population. The samples generated from the *resize* and *emit* strategies completely enclose the deformed AABB; therefore the visual error can be measured by rendering the outputs (Figure 14) and comparing the pixel coverage sums between the different methods in a wide variety of different deformation scenarios. The sampling error of the *resize* and *emit* method can then be measured by subtracting their pixel coverage sums from the *direct* rasterization, giving insight into the strategy that best reduces error on the shape silhouette according to the *type* and *amount* of deformation. Although the images in this section show a single human model for familiarity, the measurements generalize to all shapes and scenes (shown later in Chapter 4) as the AABB deformation locally ensures retained adjacency for arbitrary inputs (see Figure 16).

In Figure 20, the pixel coverage is shown for the *resize* and *emit* resampling strategies, as well as for the correct *direct* input shape (blue line). An example shear deformation is applied by varying the shear amount from 0.2 to 6.8 in the *x*-axis (later other axes are also measured), where the low-resolution renderings are shown in Table 2. The resampling error is therefore visible as divergence from the blue line (Figure 20) or as visible aliasing behind the input shape (Table 2). It is observed that the resampling error is greater in the *resize* strategy for large deformations, which is expected as the *resize* resampling has a cube-shaped approximation for the deformed AABB (with large shears, the cube-approximation poorly fits the cuboid-shaped AABB creating causing aliasing). However, it is also observed that the *emit* strategy has error, shown by divergence between the green and blue lines (Figure 20) and by aliasing (Table 2). This error appears to remain regular regardless of the amount of deformation, and is therefore examined more closely by subtracting the resampled results from the *direct* result.

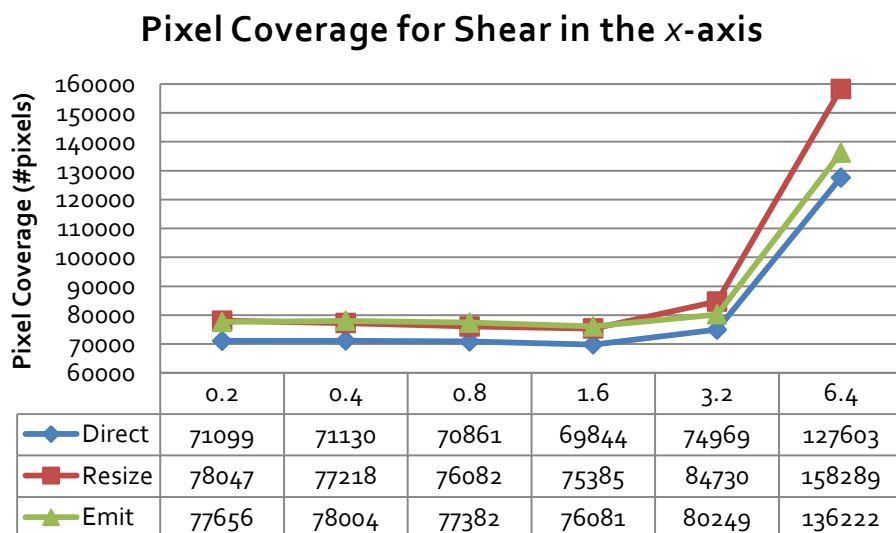











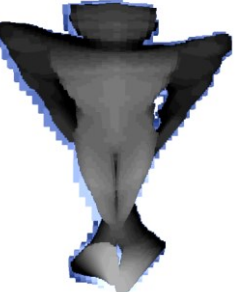







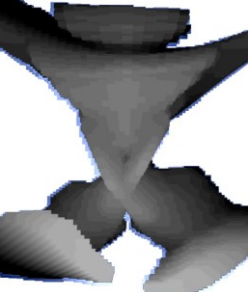



Figure 20: Pixel coverage for *x*-axis shear deformation.

**Table 2: Varying shear in the  $x$ -axis** *This table shows results from the proposed voxel deformation and resampling strategies with extreme amounts of shear in the  $x$ -axis.*

	Shear of <b>0.2</b> in $x$ -axis.	Shear of <b>0.4</b> in $x$ -axis.	Shear of <b>0.8</b> in $x$ -axis.	Shear of <b>1.6</b> in $x$ -axis.	Shear of <b>3.2</b> in $x$ -axis.	Shear of <b>6.4</b> in $x$ -axis.	Shear of <b>12.8</b> in $x$ -axis.
<b>Direct</b> (AABB Deformation) (Calculated in Figure 16)							
<b>Resize</b> (Resampling Method) (Blue Voxels Behind Direct)							
<b>Emit</b> (Resampling Method) (Blue Voxels Behind Direct)							

In Figure 21, the pixel coverage for the *emit* and *resize* resampling strategies is subtracted from the *direct* input, showing the visual sampling error (therefore the height of the red and blue lines represents the amount of aliasing). Interestingly, the *resize* method is slightly higher quality than the *emit* method in small deformations (where the blue line is lower than the red), even though it is expected to be less accurate as has a cube-shaped approximation instead of a cuboid-shaped approximation. This improvement occurs as voxels are split into  $2^n$  subvoxels (Equation (3.3.9)) at 1 higher resolution level  $h + 1$  in Equation (3.3.8) giving better alignment at the SVO structural boundaries. However the resampling error for the *emit* strategy still stays relatively low, even for very large shears such as amount 3.2 or greater (a shear of 12.8 is too large for the fixed rendering area). For further understanding of these amounts, another type of deformation (*twist* in the  $y$ -axis) is measured (Figure 22 and Table 3) and discussed.

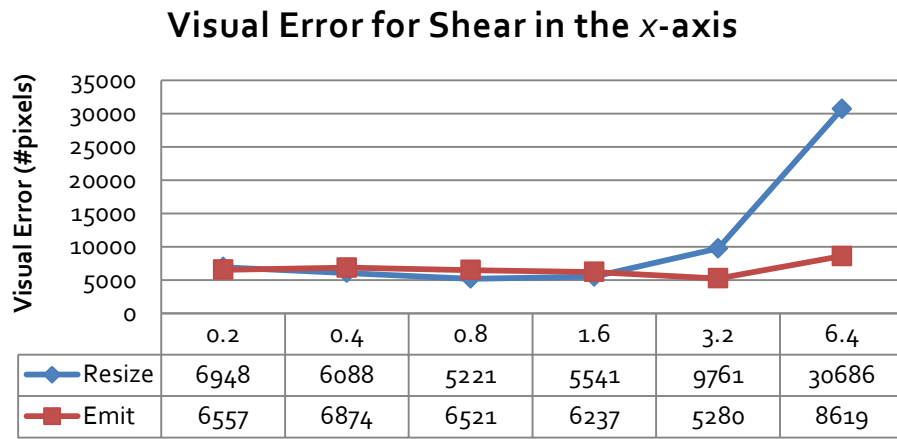


Figure 21: The *resize* and *emit* resamplings compared to the unaligned direct rasterization.

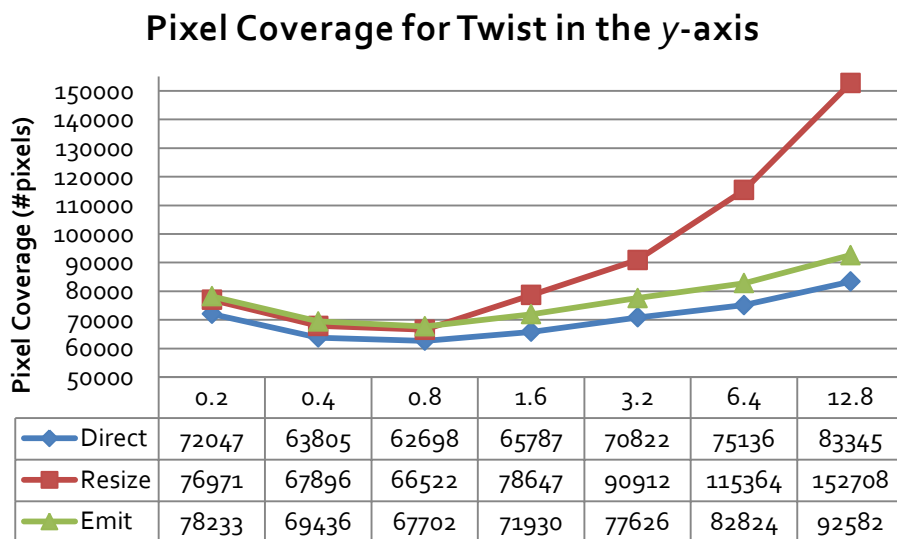


Figure 22: Pixel coverage for  $y$ -axis twist deformation.

**Table 3: Varying twist in the y-axis**

*This table shows results from the proposed voxel deformation and resampling strategies with extreme amounts of twist in the y-axis.*






















	Twist of <b>0.2</b> in y-axis.	Twist of <b>0.4</b> in y-axis.	Twist of <b>0.8</b> in y-axis.	Twist of <b>1.6</b> in y-axis.	Twist of <b>3.2</b> in y-axis.	Twist of <b>6.4</b> in y-axis.	Twist of <b>12.8</b> in y-axis.
<b>Direct</b> (AABB Deformation) (Calculated in Figure 16)							
<b>Resize</b> (Resampling Method) (Blue Voxels Behind Direct)							
<b>Emit</b> (Resampling Method) (Blue Voxels Behind Direct)							

Figure 22 and Table 3 show extremes of twist applied in the  $y$ -axis from 0.2 to 12.8, where the resampling error can be seen as divergence from the *direct* rasterization (the blue line in Figure 22) and similarly by aliasing behind the input deformed shape in Table 3. Unlike the shear example, the pixel coverage does not consistently increase with the amount of deformation. This is mainly because the  $y$ -axis twist deformation causes large parts of the model to become occluded. By itself, pixel coverage is therefore not very meaningful however there is still a trend where *resize* strategy deviates in quality from the *emit* strategy, which follows more closely to the *direct*. To see this more clearly, the pixel coverage for *resize* and *emit* resampling strategies are subtracted from the correct direct rasterization, as shown in Figure 23:

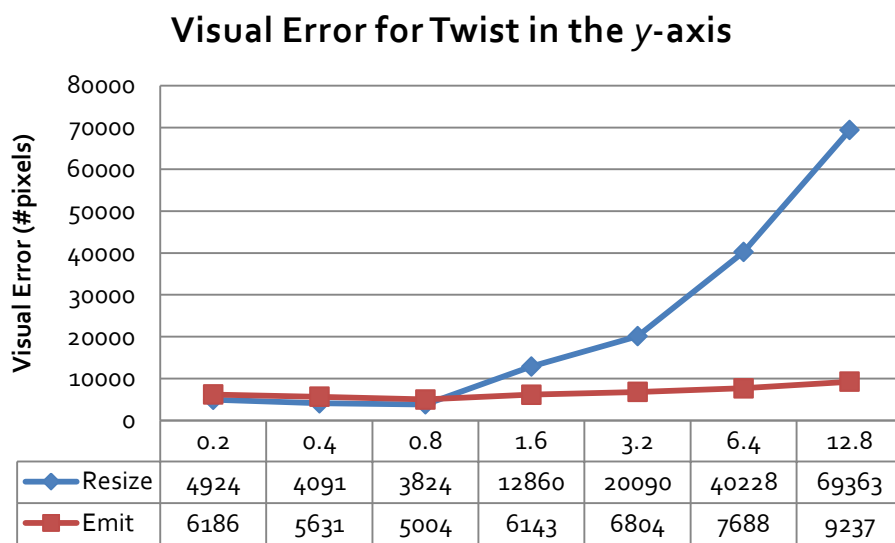


Figure 23: Error in the *resize* and *emit* resamplings, compared to the *direct* rasterization.

In Figure 23, the silhouette error in the *emit* resampling (red line) stays low with all extremes of twist. However the *resize* method degrades in quality earlier than with shear (the quality degraded in the shear at about 1.0 while the twist degrades at about 0.8). The error becomes increasingly severe for large twists, meaning that the *resize* method is unsuitable for large deformations. However, for real-time applications where stable performance is required, the *resize* strategy can still be used exclusively with oversampling. This is achieved by *selecting* voxels smaller than the pixel size: choosing a small value (such as 0.1) in the traversal condition (Equation (3.2.5)). This implicitly reduces the cube-shaped sample size and respective error at the shape boundary. Finally, it is also worth measuring the resampling accuracy for *scale* transformation. Scale in all axes is an interesting transformation as visually nothing changes for the rendered output, except the projected size of the model (Figure 24 shows this increase in pixel coverage, where the model is enlarged by the scale amount). This is replicated in the results, and therefore no comparison table is shown as all of the images appear visually similar.

### Pixel Coverage for Scale in all axes

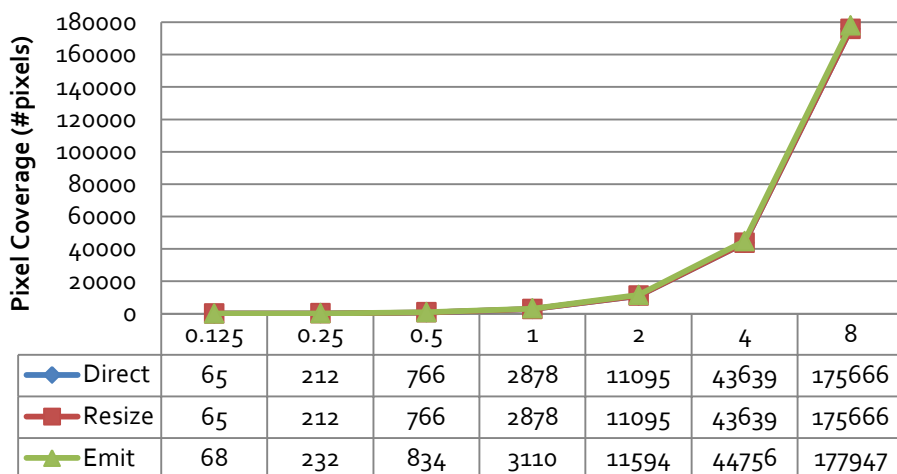


Figure 24: Pixel coverage for scale transformation in all axes.

As with the previous measurements, the error for the *resize* and *emit* resamplings is compared with the *direct* rasterization by subtracting their pixel coverage accordingly (Figure 25). The *resize* resampling (blue line at value 0) has the same silhouette as the unaligned rasterization, as the projected subvoxel size at hierarchical level  $h + 1$  (Equation (3.3.8)), is smaller than the size of an individual pixel. The *emit* resampling has a very small amount of error caused by misalignment of the voxels at the current SVO hierarchical level  $h$ . This suggests that a single *level* of further oversampling can remove the error, choosing level  $h + 1$  (Equation (3.3.11)) or by adjusting the traversal condition (Equation (3.2.5)) to half the pixel size:  $p_{size} < 0.5$ .

### Visual Error for Scale in all axes

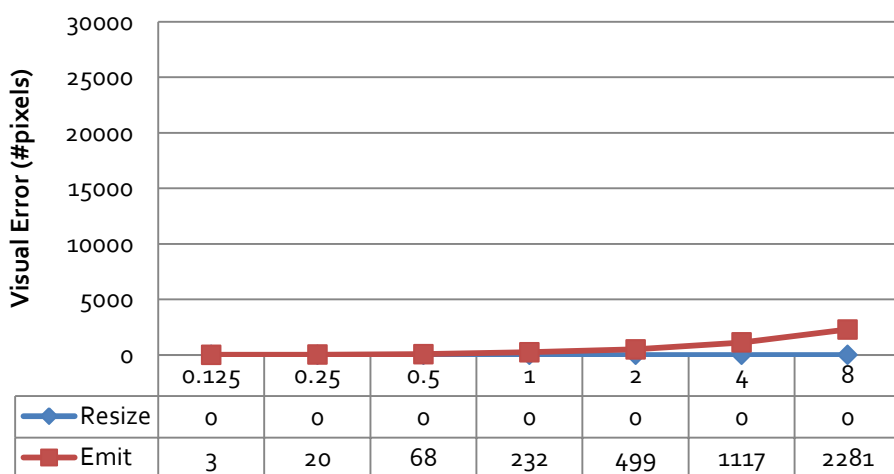


Figure 25: Resize and emit methods have none/negligible error in transformation.



## Efficiency

The efficiency of the resampling strategies (Figure 14) is primarily sensitive to the number of elements in the SIMD input vector. This depends on the hierarchical selection and deformation process (Figure 13 stage 4-5) which itself depends on the rendering resolution (Equations (3.2.5) and (3.2.7)). Therefore, to understand which resampling method is suitable for different circumstances, the relationship between components in the entire pipeline is considered:

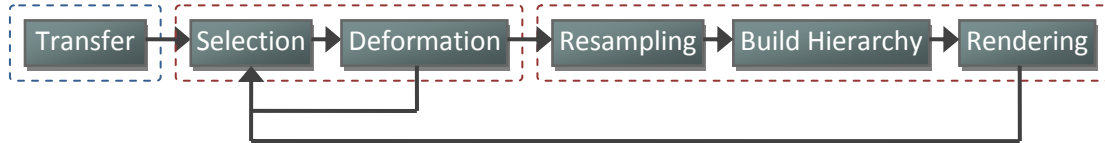


Figure 26: Recursion in the proposed pipeline.

Figure 26 gives an overview of the recursion in the proposed method (Figure 12 and Figure 13). While the resampling process itself consists of only a few operations, it can impact other parts of the pipeline, such as the hierarchy construction and rendering. The input vector for the resampling consists of approximately one voxel for each pixel in the screen (Equation (3.2.5)) where the hierarchical selection and deformation process (Figure 13 stages 4-5) utilize occlusion culling in Equation (3.2.7). However hierarchical depth buffer occlusion culling (see section 3.2 titled *Selection*) cannot be used accurately by transparent objects. This means that, in such worst-case scenarios, all voxels from the entire object need to be resampled.

To understand the extent of this worst-case scenario, the total number of samples emitted for each method is counted without hierarchical occlusion culling in the selection stage. This is measured for different types and amounts of deformation, as in the previous section. The test model is the same as in Table 2 and Table 3, which has been voxelized at a low resolution in order to see error in individual voxels, which means that the measurement stays consistent as all voxels are larger than a single pixel (instead of some voxels being discarded where they're too small by Equation (3.2.5)). The total number of output samples  $|s|$  is counted for each input voxel in the SIMD vector after Figure 13 stage 6. This is shown in Figure 27 by applying a shear deformation of amount 0.2 up to 12.8 in the  $x$ -axis. The *direct* rasterization is not adding any samples; therefore the number 33,720 is equivalent to the total number of voxels at the highest resolution in the model voxelization. This remains consistent across all deformations, as no voxels are discarded during the hierarchical traversal (explained previously). Similarly, the *resize* method emits exactly  $2^n = 8$  times as many samples (Equation (3.3.9)) regardless of the amount or shape of the deformation; a cube-shaped approximation is used in all scenarios where the display depth is changed to prevent distortion and gap artifacts.

### Worst-case resampling for shear in the x-axis

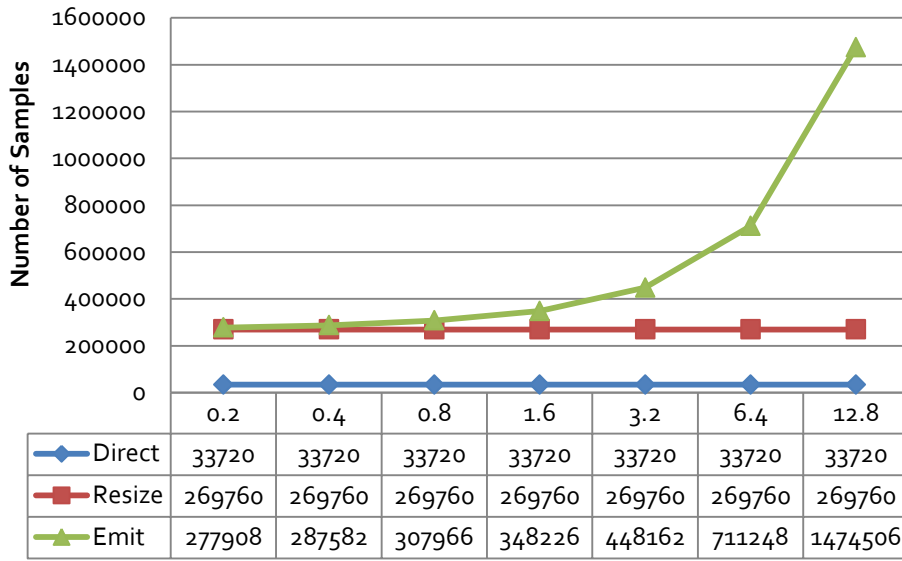


Figure 27: Number of samples emitted for applying shear to a transparent object.

Figure 27 shows the *emit* method (green line) generates a number of samples approximately proportional to the *amount* of shear deformation. This is expected as the cuboid shape of the deformed voxel AABBs stretch with larger shears, and therefore more samples need to be set to fill the cuboid-shaped region (Equation (3.3.12)). However, this appears to be less sensitive to the *type* of deformation, as shown by the similar results with twist in the y-axis in Figure 28. In contrast, the *resize* method (red line) always efficiently has  $2^n = 8$  samples per input.

### Worst-case resampling for twist in the y-axis

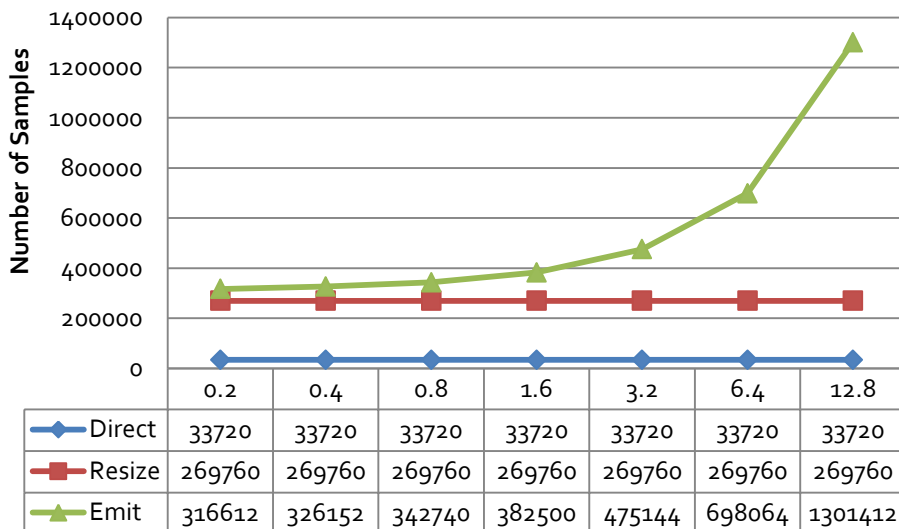


Figure 28: Number of samples emitted for applying twist to a transparent object.

### 3.5 Conclusion

In the previous section, the accuracy and efficiency of the resampling methods were compared for different types and amounts of deformation for a low-resolution input shape (the human shape was chosen for its familiarity, where the appearance of sampling error can be seen more clearly). However the results apply for all other shapes and scenes as shown in Chapter 4. This section formulates the main observations from the experiments into a practical decision tree.

The first observation was that the human visual system is particularly sensitive to the shape silhouette (Figure 19) and that increased sample density is required for large deformations, for example with the *emit* strategy or by oversampling in the *resize* strategy. However, while the *emit* strategy produces good accuracy, the dynamic sample increase is sensitive to the amount of deformation, which can impact other parts of the pipeline (Figure 26, Figure 27, and Figure 28). Fortunately, it was observed that the *resize* strategy gives a fixed sample count for all types and amounts of deformation (Figure 27 and Figure 28) and that it can be oversampled easily according to a size criteria in Equation (3.2.5) giving a robust increase in accuracy. This can act as a resolution parameter for dynamically adjusting the balance of simulation accuracy and efficiency according to the target frame budget, for real-time rendering. This approach can therefore be used in high-performance applications instead of the *emit* strategy, which would otherwise require additional memory expansion and compaction iterations to allocate space for the dynamic sample increase (Equation (3.3.12)). In special cases where performance and memory are not heavily restricted, such as in offline rendering, the emit strategy can be used to guarantee a uniform level of accuracy (Figure 21 and Figure 23).

#### Decision Tree

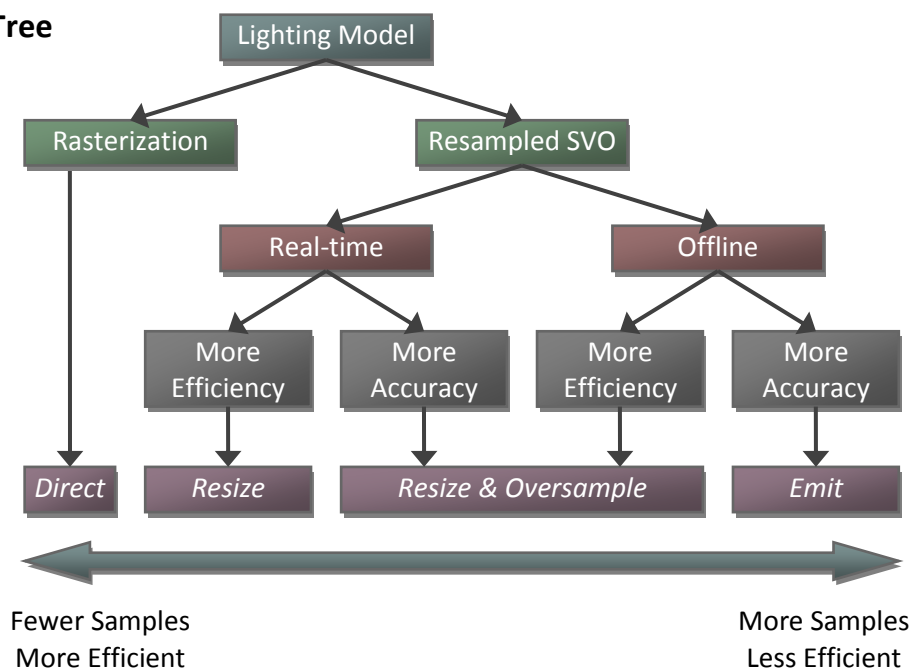


Figure 29: Decision tree for the proposed resampling strategies.

Figure 29 summarizes the observations from the previous experiments in a practical decision tree, which presents the relationship between simulation accuracy and efficiency in terms of the target sample density. The most important decision is whether or not resampling in a SVO structure is needed. For rasterization-based lighting models, the *direct* output of the selection and deformation process (the unaligned deformed AABBs at the output of stage 4 in Figure 13 and Figure 14 calculated in Figure 16) can be immediately transformed by the camera matrix and rasterized. This is therefore more efficient and accurate than resampling; however there is no hierarchical structure, which means that the accurate modeling of light transport is more expensive, such as secondary light bounces in global illumination (Ritschel et al., 2012). The resampled SVO accelerates shape queries within the massive scene, with applications in both real-time and offline rendering. Real-time applications favor efficiency, where a smooth frame rate is required to create a sense of immersion within the virtual environment. This can be controlled by varying the selected voxel size in Equation (3.2.5) which acts as a resolution parameter in combination with the *resize* strategy, giving consistent efficiency regardless of the type or amount of deformation. In contrast, the *emit* strategy is preferable in offline applications, where a consistent level of accuracy is favored above the rendering performance.

In conclusion, the decision tree provides an intuitive approach for varying the accuracy and efficiency in order to favor either real-time or offline rendering applications. Furthermore, the approaches are able to fine-tune the resolution parameter, which maximizes the benefit in either circumstance, regardless of the capabilities of the target machine.

## 3.6 Chapter Summary

This chapter has presented a pipeline for efficiently selecting, deforming, and resampling large amounts of volumetric content (Figure 13). The pipeline is constructed to map to the parallel processing capabilities of current hardware (Figure 12) and supports optimization techniques such as geometry instancing and occlusion culling to reduce memory with repetitive content and increase performance accordingly. The resampling approach is flexible, with support for rasterization-based or ray casting based renderers that may target both offline and real-time applications. This is controlled by the sample density, which is determined by the resampling method and the resolution parameter (Equation (3.2.5)) meaning that processing can be allocated to stabilize *efficiency* while giving the maximum accuracy (real-time), or to stabilize *accuracy* with the maximum efficiency (offline). These decisions were developed empirically and presented in a practical decision tree.

### Contributions

This chapter has presented a fully parallelized hierarchical content selection method, which prioritizes voxels according to their projected size, and discards occluded voxels. The unity of

these two criteria means that approximately one voxel is retrieved for each pixel, with the exception of transparent objects, which is insensitive to the shape complexity. The primary contribution of this approach is that parallel deformation is supported *inside* the selection process (Figure 13 stage 4) and therefore massive animated scenes can be displayed without wasting deformation calculations on unimportant regions of the scene. This is in contrast to traditional parallel approaches, which expensively animate entire objects at a high geometric resolution, and then later apply less effective culling techniques. These traditional parallel approaches also suffer from popping artifacts, when switching between the different levels of detail, whereas the proposed parallel approach smoothly transitions between geometric resolutions as the shape is partitioned and simulated hierarchically according to its projection and occlusion criteria. Also, in contrast to GPU parametric tessellation, the proposed pipeline supports overhanging ledges.

The later parts of this chapter have focused on efficient resampling strategies which contribute to the problem of gaps and shape distortion when realigning the deformed shapes to a SVO. Two strategies were proposed which *resize* or *emit* samples in order to maximize *efficiency* or *accuracy* accordingly. These two strategies were compared to the unaligned *direct* deformed output (which is not resampled) and their properties observed for different types and amounts of deformation. This was concluded with a practical decision tree that presents how to utilize the available processing, regardless of the restrictions imposed by the target machine, in order to benefit the target application in both real-time or offline rendering.

## Limitations

The focus of this work is on *efficiently* modeling deformation, and has focused on sampling the localized axis-aligned output of the deformed selected voxels. An improvement would be to consider sampling the oriented output of the deformed voxels; however this would require expensive matrix storage and transformation operations (Ericson, 2005). It would also be worth considering the importance of sample distribution according to other heuristics: in the *Measurement* section, it was observed that human visual perception is particularly sensitive to the shape silhouette. Therefore it would be worth investigating storing an importance metric in a 2D buffer, which highlights such areas with post-processing of the previous frame, in order to influence Equation (3.2.5) to allow for further oversampling in temporal coherent regions.

The current occlusion culling approach uses the popular hierarchical depth buffer strategy, which does not correctly handle transparent or translucent objects. In such cases, many voxels per-pixel are processed, which is not scalable in massive transparent scenes with high depth-complexity. It would be worthwhile investigating an occlusion culling strategy that supports transparent objects, considering other visibility feedback information from the renderer.

# Chapter 4

## Rendering

### 4.1 Abstract

This chapter investigates the problem of rendering massive dynamic volumetric scenes. This is challenging because huge amounts of shape queries are required to model light transport each frame. However, in the previous chapter, a hierarchical selection and deformation pipeline was presented (Figure 13), which retrieves, deforms, and resamples (where relevant) the important shape regions. This therefore reduces the scene content to a smaller set of contributing shape regions (Figure 2), which are more manageable and can be hierarchically organized in real-time. In this chapter, a method is presented to organize the selected, deformed and resampled volumetric elements (Figure 13 stage 7) such that huge numbers of shape queries can now be made each frame for massive dynamic volumetric scenes. The later parts of this chapter then demonstrate the sparse volumetric deformation pipeline with efficient rasterization and ray casting based renderers, which greatly extends the range of applications.

### 4.2 Methodology

Photorealistic rendering seeks to accurately model *shapes* and *light* as they occur in the real world, and to accurately simulate the physics of light (*optics*) as it interacts with a scene before sampling light which reaches the camera. In the real world, photons bounce amongst objects and at each point of intersection they are absorbed, reflected, and refracted in numerous ways. In massive dynamic scenes, modeling such paths of light transport inevitably requires a lot of queries and therefore, for real-time rendering applications, the shape information needs to be organized to prevent iterating unimportant regions (see *Spatial Partitioning*). However, it is also possible to consider non-photorealistic rendering (NPR) for example by only considering direct light from light sources and approximating other effects of light, such as screen space ambient occlusion (SSAO) and environment (reflection) mapping. While not as accurate, these approaches can be rasterized *directly* without resampling or organizing the selected content, and are therefore expected to be more efficient with applications in stylized rendering and performance-critical simulation.

In this chapter, an efficient method is developed for hierarchically organizing the resampled output, by generating hierarchy on the GPU, such that shape queries can be accelerated with an efficient real-time traversal process. This is demonstrated in a modern real-time ray casting implementation. The second part of this chapter explores an efficient rasterization-based renderer, which is able to efficiently display direct illumination without requiring a hierarchical

structure. The unity of these two approaches means that a wide range of applications are available in both photorealistic and non-photorealistic rendering. This therefore demonstrates the flexibility and applications of the sparse volumetric deformation pipeline.

## Hierarchy Construction

This section addresses the problem of efficiently constructing hierarchy for the resampled output after deformation. In the literature review, it was discussed that reusing the existing tree hierarchy causes expensive deterioration and fragmentation in dynamic scenes. It was suggested that such degradation is avoidable by reconstructing the tree each frame, which is more suitable for GPU architectures. The majority of parallel methods generate hierarchy sequentially; however this limits parallelism and scalability according to the location in the tree. In contrast, the recent work by (Karras, 2012) presents a method that is able to maximize the amount of parallelism and achieve a consistent acceleration. Their approach first chooses the order for which nodes appear in the tree, and then they generate the internal branches with respect to this order. Nodes are therefore sorted along the  $z$ -order space-filling curve, which ensures data localization, and then the internal branches are ordered in a specific way which enables their children to be calculated without depending on earlier results. This means that all of the work can be done completely in parallel, which can give a large performance improvement over traditional top-down approaches.

The challenge with parallelized hierarchical construction is that it operates on a uniform leaf level of hierarchy. However in the parallelized selection and deformation pipeline (Figure 13) the new samples are of varying hierarchy levels, for example in Equations (3.3.6) and (3.3.10). In order to retain their level information, the nodes are adjusted to a uniform depth and their hierarchical level  $h$  is preserved throughout the hierarchical construction process. Internal branches whose depth is greater than this level can then be flagged to be ignored in traversal.

Figure 30 shows this approach, which retains the hierarchical level of the samples throughout the parallel hierarchical construction algorithm. The main idea is to adjust the grid indices of the resampled voxel data in order to simulate them being at the same (leaf) level of hierarchy (Figure 30 stage 1-2). Therefore a fully parallelized hierarchical construction approach can be used (Figure 30 stages 3-5) (Karras, 2012) with some modifications (Figure 30 stage 4) in order to preserve the sample level stored in the value of the branches. This guarantees stable worst-case construction performance and maximizes the parallelism, which is suitable for the sparse volumetric deformation pipeline. The upcoming sections discuss the details of this approach for each component in Figure 30.

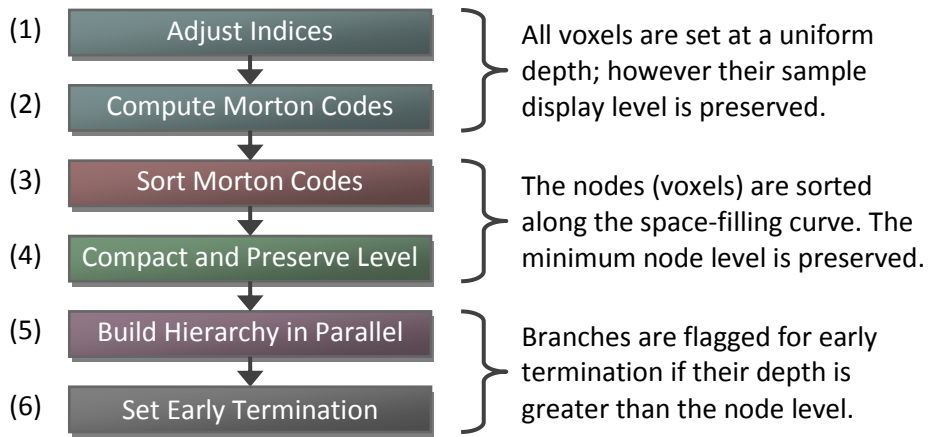


Figure 30: Preserved node hierarchy in parallel hierarchy construction.

### Adjusted Indices

The hierarchical SVO structure regularly partitions space, where each hierarchical level may be considered a sparse 3D regular grid. The corresponding cells for each sample location, at level  $h$ , may therefore be calculated according to the location within the SVO. Therefore, the scene geometry needs to be transformed to be located within the SVO bounds; within the unit cube  $[0,1]$ . According to this assumption, the leaf cell location  $\mathbf{m}$  is calculated by multiplying each sample position  $\mathbf{s}$  by the total number of regions  $r = 2^h$  in each axis at the desired level  $h$ , as in the following equation:

$$\mathbf{m} = \begin{bmatrix} \lfloor s_x r \rfloor \\ \lfloor s_y r \rfloor \\ \lfloor s_z r \rfloor \end{bmatrix} \quad (4.2.1)$$

This can be can also be achieved by not multiplying the samples by  $1/r$  in Equation (3.3.9) and Equation (3.3.12) accordingly, however, in both cases the samples outside the unit cube must be constrained to be within the SVO bounds, therefore each component of  $\mathbf{m}$  is clamped to the range  $[0, r - 1]$ .

In order to set all samples at a uniform level,  $r$  can be set where  $r = 2^i$  and where  $i$  is the target SVO hierarchical level (discussed in the next section); however the original sample level is stored as a temporary *value* for each sample. This hierarchical level value is later preserved throughout the entire parallelized hierarchical construction process (Figure 30) and then used to determine the minimum traversal depth (an early-termination condition) for the generated internal branches.



## Morton Codes

Morton codes are a 1D value which describes the location along an  $n$ -dimensional space-filling curve. Therefore sorting the 3D sample locations according to their Morton codes ensures that the 3D data becomes localized in 1D memory. Morton codes in 3D can be efficiently calculated for the integer indices  $\mathbf{m}$  by interleaving the three components  $\mathbf{m}_x, \mathbf{m}_y, \mathbf{m}_z$  to produce a single unsigned integer Morton code  $k$ . This is shown in Figure 31:

```
1 // Dilate bits along the 64-bit unsigned integer
2 ulong Dilate(ulong x) {
3     x = (x | (x << 32)) & 0x7fff00000000ffff;
4     x = (x | (x << 16)) & 0x00ff0000ff0000ff;
5     x = (x | (x << 8)) & 0x700f00f00f00f00f;
6     x = (x | (x << 4)) & 0x30c30c30c30c30c3;
7     x = (x | (x << 2)) & 0x1249249249249249;
8     return x;
9 }
10
11 // Contract bits along the 64-bit unsigned integer
12 ulong Contract(ulong x) {
13     x = (x >> 2) & 0x1249249249249249;
14     x = (x | (x >> 2)) & 0x30c30c30c30c30c3;
15     x = (x | (x >> 4)) & 0x700f00f00f00f00f;
16     x = (x | (x >> 8)) & 0x00ff0000ff0000ff;
17     x = (x | (x >> 16)) & 0x7fff00000000ffff;
18     x = (x | (x >> 32)) & 0x00000000ffffffff;
19     return x;
20 }
21
22 // Encode a 21-bit xyz to its 64-bit morton code
23 ulong Encode(ulong x, ulong y, ulong z) {
24     return Dilate(x) | (Dilate(y) << 1) | (Dilate(z) << 2);
25 }
26
27 // Decode a 64-bit code to its 21-bit index xyz
28 void Decode(ulong code, ulong &x, ulong &y, ulong &z) {
29     x = Contract(code >> 2);
30     y = Contract(code >> 1);
31     z = Contract(code >> 0);
32 }
```

Encode and decode a 3D location to a 64-bit 1D morton code along the space-filling curve.

Figure 31: Pseudocode for 64-bit Morton Codes.

In Figure 31, the function *Encode* (line 23) interleaves the three components of a grid location into the 1D location along the  $z$ -order curve, and similarly *Decode* (line 28) de-interleaves the 1D value to generate the respective three components accordingly. For example, a 32-bit 3D Morton code can be calculated by interleaving three 10-bit integers, which therefore supports a maximum 3D resolution of  $(2^{10})^3 = 1,024^3$ . In practice, this resolution is too small for many

applications; however a much larger resolution can be supported with a 64-bit Morton code, with the interleaved value of three 21-bit numbers giving a maximum resolution  $i = 21$ , resulting in  $(2^{21})^3 = 2,097,152^3$ . This is calculated in Figure 31 by the shift-based algorithm shown for 32-bit integers in (Raman & Wise, 2008; Stocco & Schrack, 1995) but dilated along the 64-bit word size (lines 3-7), and similarly they are compacted for the *Decode* operation along the 64-bit word size (lines 13-18).

The samples can therefore be efficiently sorted along the  $z$ -order space-filling curve (Figure 30 stage 3) using a parallel radix sort (Merrill & Grimshaw, 2010; Satish, Harris, & Garland, 2009) on the Morton code key values  $k$  (calculated by *Encode* in line 23 of Figure 31 for each sample of its uniform adjusted index  $m$ ). The original display level  $h$  for each sample (Equation (3.3.6) +1 for *resize* and Equation (3.3.10) for *emit*) is stored in the value for each sample, and preserved throughout this sorting process and the following components in the hierarchical construction pipeline.

### Sample Level Preservation

In the previous sections, the samples were localized at a uniform hierarchical level by sorting them along the  $z$ -order space-filling curve (Figure 30 stages 1-3). This means that identical samples, sharing the same Morton code (within the same 3D cell) can be efficiently removed using parallel stream compaction to ensure that every leaf node is unique (Figure 30 stage 4). Therefore, a binary radix tree can be efficiently constructed on the leaf nodes in parallel with the approach by (Karras, 2012) (Figure 30 stage 5). Finally, the newly generated branches are updated with the preserved values  $h$ , retaining the varying hierarchy of the input samples (Figure 30 stage 6). This section discusses the method of stages 4-6 in more detail.

Parallel stream compaction is an important parallel primitive used to remove undesired elements in sparse data (Billeter et al., 2009; Merrill & Grimshaw, 2010). Samples within the same leaf region are flagged for removal by comparing each pair of adjacent (sorted) Morton codes (Karras, 2012). However it is additionally required to preserve the varying levels of the input samples  $h$ . This is achieved by a parallel compaction by key, with a minimum binary operator on the  $h$  value (Figure 30 stage 4).

With unique Morton codes at uniform level  $i$ , and the minimum preserved  $h$  values, an octree can be built efficiently (Figure 30 stage 5). Initially, a parallel binary radix tree is constructed and the octree nodes are allocated with a parallel prefix sum. The parents of the octree nodes are then found by looking at the immediate ancestors of each radix tree node, as described in (Karras, 2012). The correct  $h$  value is set for the internal branches by the minimum  $h$  value of the nodes children, as in Equation (4.2.2):

$$h = \min(h_{\text{children}}) \quad (4.2.2)$$

Therefore each internal branch of depth  $d$  can be flagged for early termination where it is undesired (Figure 30 stage 6). This is where it is too deep to represent the minimum sample level of its containing leaves, or more formally by Equation (4.2.3) where:

$$d \geq h \quad (4.2.3)$$

In conclusion, the pipeline in Figure 30 constructs a SVO with varying sample hierarchy by preserving the sample levels  $h$  through a uniform parallelized octree construction process on the resampled volume elements. The internal branches are then flagged for early termination, where their undesired children are ignored for continued traversal, allowing for efficient shape queries or ray casting applications.

### 4.3 Ray Casting

Ray casting based rendering simulates light transport by firing many rays within the scene, and testing for intersection with the shapes. With massive scenes, this is expensive; therefore a hierarchical structure is used to accelerate the shape queries. The previous section discussed how a SVO structure was efficiently constructed for important shape samples, and therefore rays can be cast into this structure in parallel by traversing the SVO hierarchy. The purpose of this section is to demonstrate efficient real-time ray casting with parallel ray traversal of the SVO structure, showing the applications of the sparse volumetric deformation pipeline.

#### Ray Traversal

Ray traversal involves finding the point of intersection between the shapes in the scene and the input rays. A ray  $\mathbf{R}$  contains a starting position  $\mathbf{s}$  and a normalized direction vector  $\mathbf{v}$ . The common notation defines a point  $\mathbf{p}(t)$  along the ray by an interval  $t$ , where  $t \geq 0$  and:

$$\mathbf{p}(t) = \mathbf{s} + t\mathbf{v} \quad (4.3.1)$$

Calculating the intersection with the voxels in the SVO (constructed in the previous section) is equivalent to testing for intersection with their AABBs  $[\mathbf{b}_{\min}, \mathbf{b}_{\max}]$ . This process is optimized in (Kay & Kajiya, 1986) to give the start and end intervals of intersection  $t_{\text{start}}$  and  $t_{\text{end}}$ :

$$\begin{aligned} t_{\text{near}} &= \max\left(\min\left(\frac{1}{\mathbf{v}} \cdot (\mathbf{b}_{\min} - \mathbf{s}), \frac{1}{\mathbf{v}} \cdot (\mathbf{b}_{\max} - \mathbf{s})\right)\right) \\ t_{\text{far}} &= \min\left(\max\left(\frac{1}{\mathbf{v}} \cdot (\mathbf{b}_{\min} - \mathbf{s}), \frac{1}{\mathbf{v}} \cdot (\mathbf{b}_{\max} - \mathbf{s})\right)\right) \end{aligned} \quad (4.3.2)$$

Where  $t_{\text{near}}$  represents where the ray enters the AABB and likewise  $t_{\text{far}}$  represents where the ray exits the AABB. To traverse the SVO, the rays (for example located at each pixel) are initially tested for intersection with the root voxel at the unit cube. If intersection occurs ( $t_{\text{far}} > t_{\text{near}}$ ) the ray is updated to the front of the SVO, as in the following equation:

$$\mathbf{s} = \mathbf{s} + t_{\text{near}}\mathbf{v} \quad (4.3.3)$$

The traditional method then steps along the ray using a digital differential analyzer (DDA) line generating algorithm (Chang, 2001; Fujimoto, Tanaka, & Iwata, 1988) however this requires the SVO leaf size to be known in advance. Furthermore, assuming the SVO is stored in the GPU texture memory, a huge number of texture fetch operations are required with this approach as the SVO hierarchy must be traversed from the root at each small step along the line.

### Space Leaping

A more efficient strategy exploits the sparse properties of the encoded SVO by skipping over empty space (Crassin et al., 2009; S. Laine & Karras, 2011) which outperforms the DDA method and does not require the leaf size to be known in advance. The hierarchy is initially traversed from the root; however, when an empty node is found, the ray position is updated to a new point just after the current  $t_{\text{far}}$ , to exit the current AABB. This means that the empty space is *leaped* according to the size of the current AABB, which is large in the sparse regions. In practice, this approach greatly accelerates the traversal process.

To find  $t_{\text{far}}$ , it is already known that the ray position is inside the current voxel AABB (the root node in the first iteration), therefore  $t_{\text{far}}$  is updated according to the extremes of the current child (Equation (4.3.2)). After finding  $t_{\text{far}}$ , the ray is *pushed* into the next cell by moving it a tiny amount  $\varepsilon$  in the axis-aligned direction of the ray direction (the sign of  $\mathbf{v}$ ), shown as follows:

$$\mathbf{s} = \mathbf{s} + t_{\text{far}}\mathbf{v} + \varepsilon \cdot \text{sign}(\mathbf{v}) \quad (4.3.4)$$

The process is then repeated recursively, starting from the root again, while the ray is inside the SVO (the unit cube) or until a sample is found according to the internal branch level (where Equation (4.2.3) is true). This process can be further optimized by the exploiting of integer coordinates, and by climbing back up the hierarchy instead of starting each time from the root. However the texture cache is well prepared in both cases with small performance gain to be found. More details on integer coordinate optimizations are given by (S. Laine & Karras, 2011).

Ray tracing continues this process by generating and traversing secondary rays at the point of intersection in accordance with the shape information to capture more advanced lighting. Global illumination can be captured by simulating light transport according to the properties of the rendering equation (Kajiya, 1986; Ritschel et al., 2012). However in practice, the main contribution of light which reaches the retina or camera is from direct illumination, which can be displayed without resampling by using a rasterization pipeline (Figure 14). Rasterization can also be combined with a low-resolution generated SVO to capture the main effects of indirect lighting with the high-resolution rasterized direct lighting (Crassin et al., 2011).

## 4.4 Rasterization

This section extends the selection and deformation pipeline (Figure 13 stages 4-5) with an efficient rasterization approach that does not require resampling or a hierarchical construction procedure (Figure 13 stages 6-7). The straightforward strategy for rasterization is to gather the selected voxels and emit and rasterize a set of box primitives. However most of these boxes are approximately the size of a pixel (see Equation (3.2.5)) which gives very poor rasterization performance and requires expensive accumulation.

### Mipmap Strategy

Instead, the proposed approach extends the hierarchical occlusion culling strategy (Equation (3.2.7)) but only after the target voxels are selected (after Figure 13 stage 5) with the goal of setting a hierarchical framebuffer instead of sampling from it. Therefore leaf voxels, which are close to the camera occupying more than one pixel, are set at a lower resolution mipmap, whose overlapping region covers their high-resolution pixels. These mipmaps are efficiently merged to produce the final image. This strategy is outlined in the diagram in Figure 32:

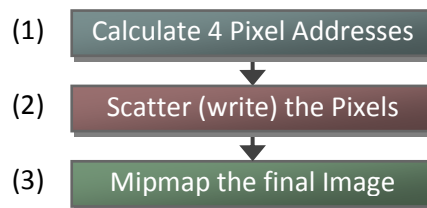


Figure 32: Parallel rasterization pipeline.

In the selection and deformation pipeline (Figure 13 stages 4-5) the 2D rectangle of the projected deformed voxels was calculated and used to sample four pixels  $Z$  in the hierarchical z-buffer at level  $l$  (Equation (3.2.6)). These four pixels cover the overlapping region of the deformed voxels in the final output image, and therefore they can be used for rendering. In order to achieve this in parallel, the four pixel indices  $Z$  calculated in (Equation (3.2.6)) are used as a map (Figure 32 stage 1) for a scattering operation (Figure 32 stage 2) that writes the

output pixels if they are the minimum (nearest) value in the depth-buffer. The final image is then calculated in a top-down parallel mipmap operation, which sets the high-resolution  $2 \times 2$  pixel regions, by the value (if it exists) of their parent pixel. Therefore the highest resolution mipmap captures all of the information, and can be displayed as the rendered output image. It is worth mentioning, for optimization, that the scattering (Figure 32 stages 1 and 2) can occur *inside* Figure 13 stage 4, instead of as a postprocess, which means that a temporary buffer is not required to store the currently selected shape information.

## Voxel Appearance

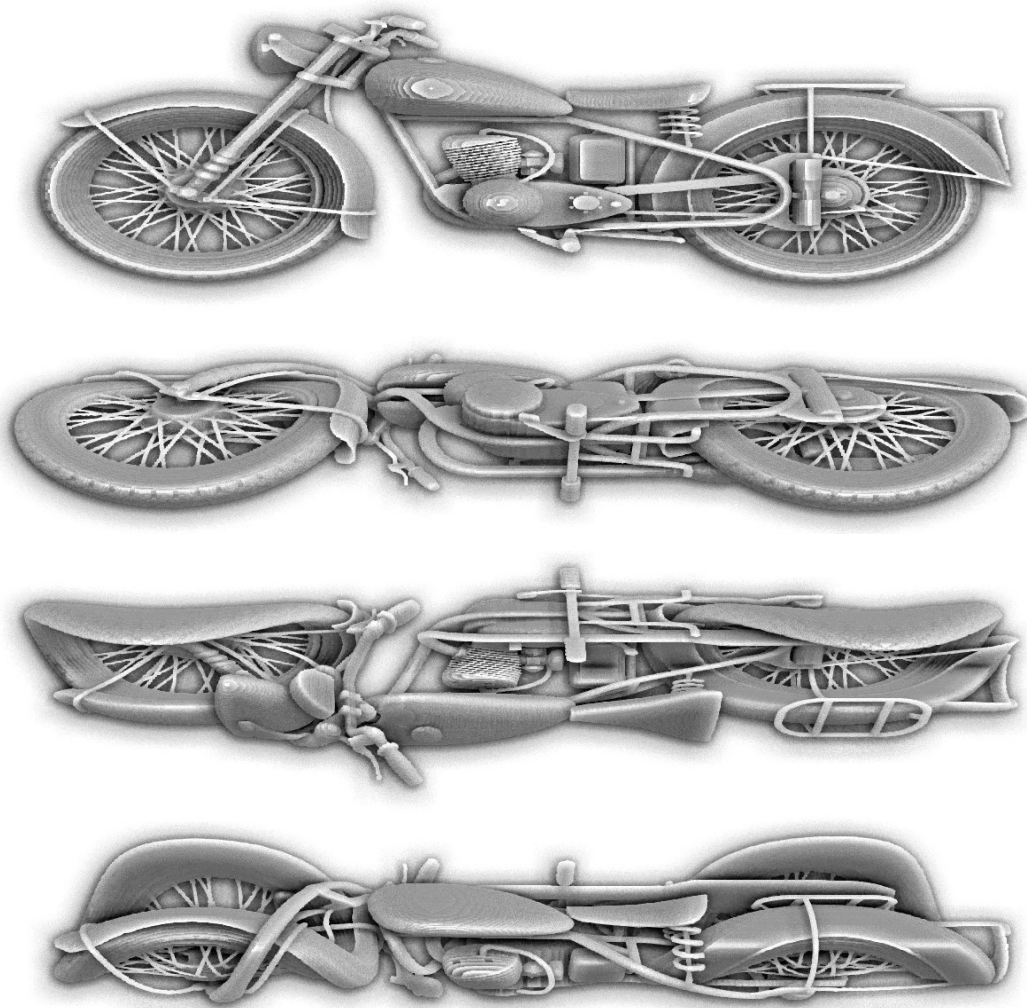
The disadvantage of this approach is that the projected 2D rectangle may poorly approximate the shape of the deformed AABB after projection (Equation (3.2.2)) and therefore, in such cases where the projected size  $p_{\text{size}} \gg 1$  (Equation (3.2.3)), the voxel primitives can be accumulated and emitted traditionally. However, in practice, this is not necessary if the input models are voxelized at sufficient resolution. This is because their equivalent lower-resolution mipmap provides an *acceptable* representation of the shape; the equivalent unfiltered rasterized AABB also has an undesirable appearance (S. Laine & Karras, 2011). Instead, post-processing effects, such as blurring and fading, can be applied to any low-resolution mipmapped voxels to hide their coarse shape, giving an attractive depth-of-field effect. Ideally, more accurate dynamic filtering is needed; however this remains an area for future work.

## 4.5 Results

This section discusses both the *efficiency* and *accuracy* of the sparse volumetric deformation pipeline and the rendering approaches. The quality and performance are examined for the generated images, and the work is compared with related previous approaches.

### Accuracy

The accuracy of the generated images is mainly sensitive to the resolution of which voxels are selected for deformation and rendering (Equation (3.2.5)). This depends on the resolution of the input models and the screen resolution. Therefore, it is important to voxelize the input models at high-resolution to ensure that their selected regions are represented at sufficient detail. By selecting and simulating deformation for approximately one voxel for each pixel (Equation (3.2.5)), the generated deformations and rendered images are aesthetically valid and attractive. This is demonstrated in Figure 33, which shows four frames from a sequence of extreme twist deformations for a complex motorcycle model, rendered in real-time at 23ms per frame in the accompanying video. In particular, the artifacts highlighted in Chapter 1 have been addressed, where gaps or shape distortion occur (Figure 4), which can be seen by the smoothness of the deformed tires and twisted bike spokes in the lower image of Figure 33.



*Figure 33: The deformations are smooth without gaps or shape distortion.*

In the implementation, FXAA (Lottes, 2009) and a naïve SSAO post-process have been added to reduce aliasing on the shape silhouette and highlight the high-frequency details accordingly. However, without more accurate filtering such as in (Heitz & Neyret, 2012; S. Laine & Karras, 2011), the shape regions appear undesirably thick and have banding artifacts (top image of the motorcycle in Figure 33). In order to see the thick regions more clearly, another extreme twist deformation is also applied to the original highly detailed polygon mesh, consisting of 148,524 triangles, which has been ray traced offline to generate the lower image in Figure 34. The same deformation is also applied to the motorcycle using the proposed pipeline (top image in Figure 34) for visual comparison. It is observed that the deformed polygon mesh has intersections and undesirable tearing artifacts according to the irregular distribution of the mesh geometry. In contrast, the regular distribution of the volumetric deformation gives smoother and more accurate deformation, as seen by the spokes in the lower right of both images. Therefore, both strategies have advantages and limitations, however better filtering is an area for future work.



Figure 34: Comparison between sparse volumetric deformation (upper) and offline high-resolution ray-traced polygon deformation (lower) for the motorcycle in Figure 33.

Comparing the rasterization and ray casting renderers is equivalent to comparing the results between the *direct*, *resize*, and *emit* resampling strategies. These are discussed extensively in the previous chapter on *Measurement*. In practice, it is observed that the image quality of the two rendering approaches remains consistent with the amount of sampling (Figure 29), as the accuracy can be increased proportionally in all cases by altering Equation (3.2.5) to select higher resolution shape information for simulation. The effect of selecting different resolutions is presented more clearly in Figure 35, which shows the ray casting of the Thai Statue rendered at the equivalent selected depths of 6, 8, and 10 (with resolutions of  $64^3$ ,  $256^3$ , and  $1024^3$  accordingly). The generated images highlight the importance of ensuring that the input shapes are voxelized at sufficient resolution, as just two *depths* can capture much smoother and finer details, as shown by the right image. Therefore oversampling, by tuning Equation (3.2.5), can be used to simulate deformation at finer resolution to give more accurate results, however it processes more data through the pipeline (Figure 13), which is less efficient. This means that Equation (3.2.5) can be tuned according to the capabilities of the target machine.

In conclusion, the generated images are smooth and attractive as approximately one voxel has been selected for each pixel for simulation and rendering. In all cases, the deformations do not have undesired gap or distortion artifacts, and can be calculated efficiently, however more accurate filtering is needed to preserve the appearance of fine features or thin shape regions.



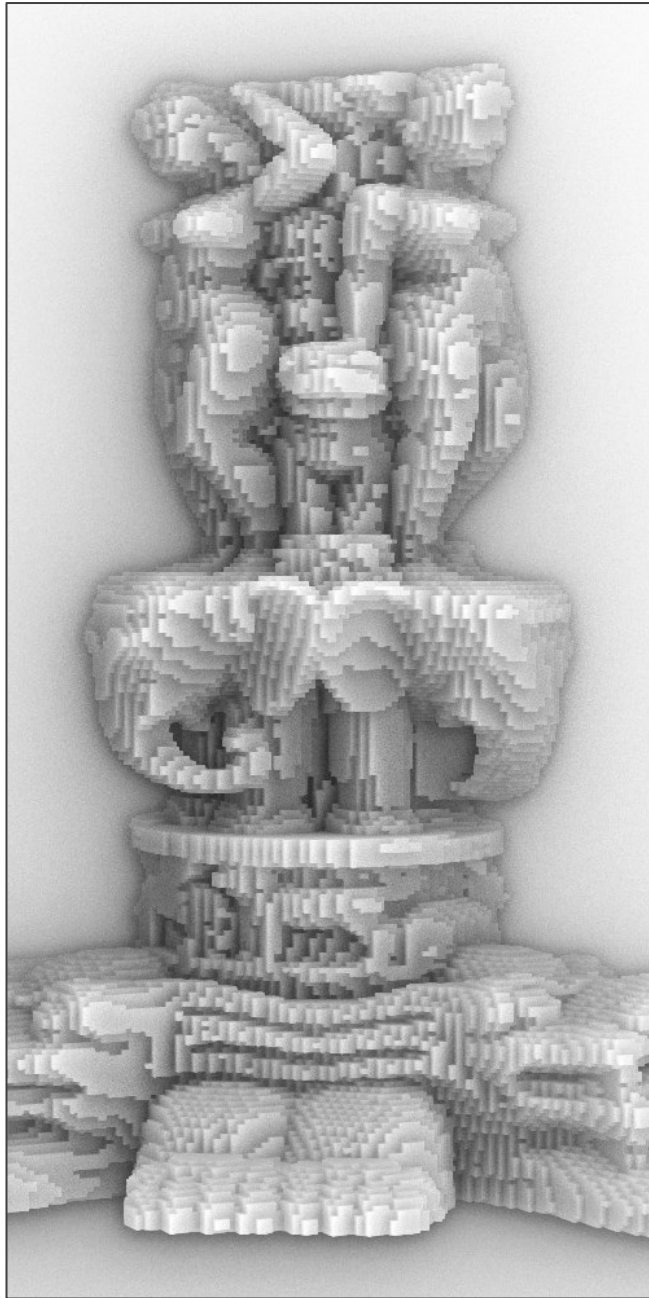
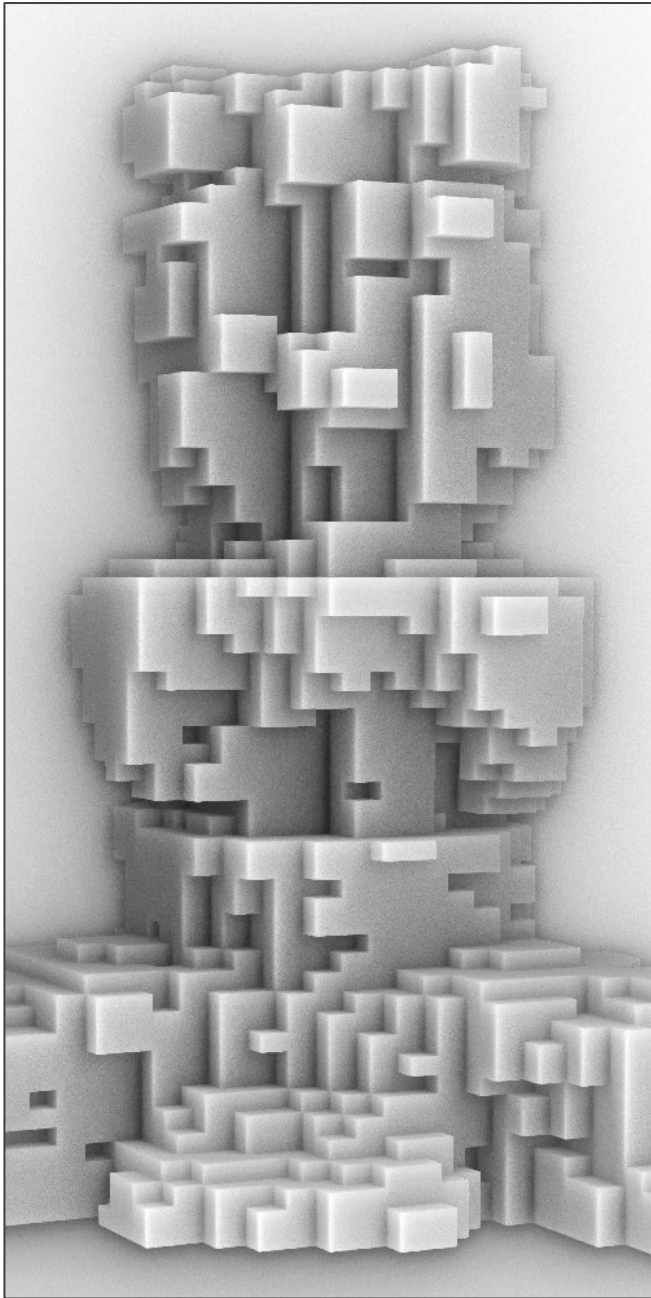
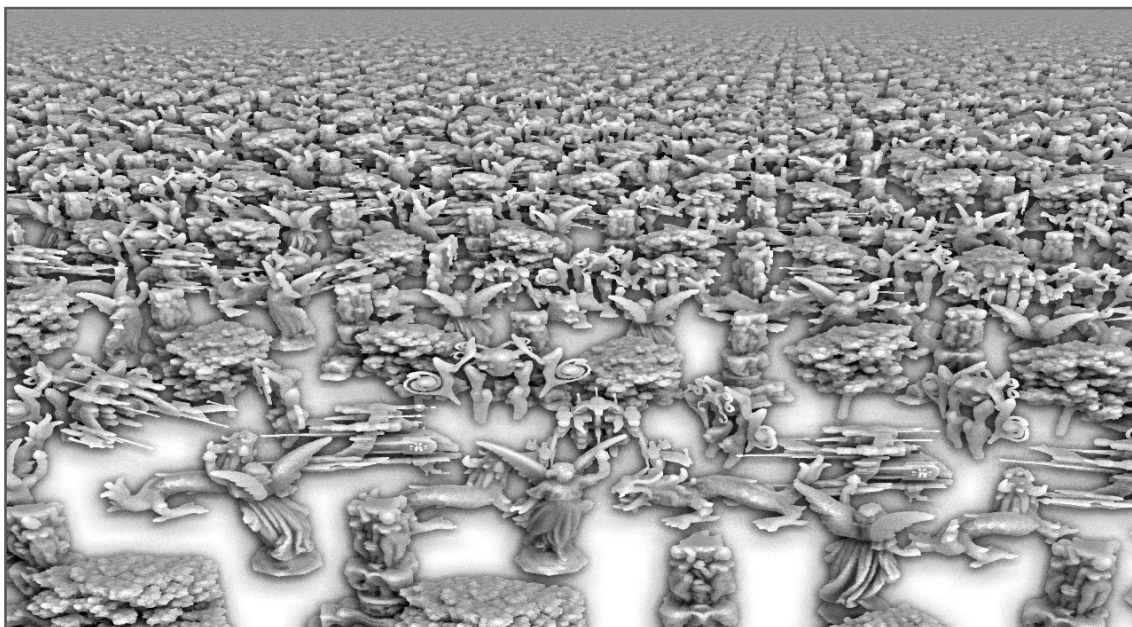


Figure 35: Close-up images showing output ray casting of depths 6, 8, and 10 respectively. Just two depths results in much smoother details, seen in the right image.

## Efficiency

The selection and deformation pipeline (Figure 13), resampling strategies (Figure 14), hierarchy construction method (Figure 30), and rasterization approach (Figure 32) are all designed for wide SIMD parallelism. This has been programmed for a GTX 260 with 111.9 GB/s memory bandwidth using CUDA C and OpenGL. This section discusses the memory usage and the rate of which images are generated using the sparse volumetric deformation pipeline.

The performance of the deformation and rendering pipeline is measured for the massive scene shown in Figure 36 (also Figure 1), which consists of one million uniquely animated instances of six high-resolution models rendered in real-time. The models are complex with overhanging ledges and internal volumetric features and they natively require 1.4GB of memory, which is consumed by their high-resolution SVOs. The expansion iterations (Figure 13 stages 4-5) can therefore exceed the target GPU memory, however if the GPU limit is reached the future expansions are prematurely terminated for that frame such that the currently selected information can be displayed. This also highlights the importance of instancing with volumetric modeling, as representing all of these models at their original resolution within a single SVO would exceed even secondary storage.



*Figure 36: Real-time frame from scene in accompanying video, consisting of massive numbers of objects, each with unique twist, shear, scale, translation, or skinning deformations.*

In order to understand the performance of the proposed pipeline, the timing measurements are separated according to the components in Figure 13: in particular the timing is measured for gathering, deformation and labeling (stage 4), expansion (also in stage 4), parallel stream compaction (stage 5), and rendering. These measurements use a GPU timer (*cudaEventRecord*) which allows asynchronous parts of the pipeline to continue uninterrupted. In collecting the results, it was found that the timings are sensitive to the camera location within the scene; and therefore the scene is simulated in different camera scenarios for static and animated objects accordingly: (1) with static objects and a fixed camera location, (2) with static objects and a moving camera, and (3) by animated objects with a moving camera.

Figure 37 shows 1,000 frames of the scene in Figure 36 using static objects and a fixed camera location similar to that in the image, which is useful because it shows a clear and consistent separation of the timings for each of the main parts of the method. The most expensive part appears to be stage 4 (Figure 13), which is represented by blue and red colors (Figure 37) that consume on average 55% (blue) and 22% (red) of the total frame time. This bottleneck is caused by the large amount of gathering operations required to retrieve the volumetric information as there are no deformation calculations. Although there is no advanced caching system which reuses information from the previous frame (other than the depth buffer) such as by (Crassin, 2011) and therefore further optimizations may be possible in future work. The stream compaction algorithm (green 17% of the total frame) implements the approach by (Billeter et al., 2009), which is a GPU memory bandwidth-bound operation on the 64-bit keys that contain the instance reference and the voxel index (see the SIMD input of Figure 14).

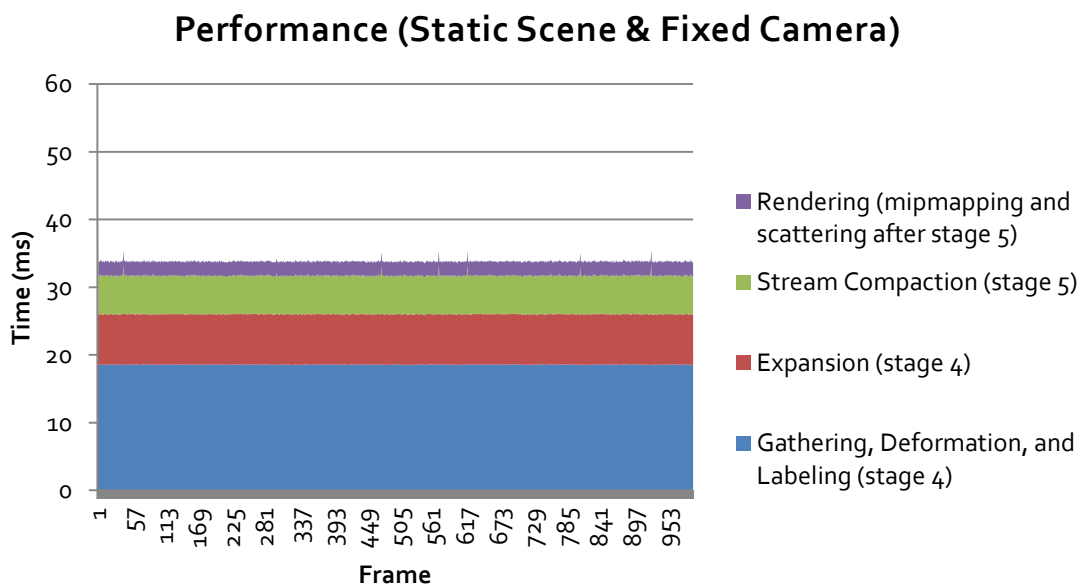


Figure 37: Timing for 1,000 frames with a massive static scene and a fixed camera.

The rendering (purple) in Figure 37 takes approximately 6% of the frame time to generate the mipmaps and map the CUDA device memory to OpenGL to display the output image. However it is worth mentioning that parts of the renderer scattering operation have been implemented as an optimization inside the gathering and deformation stage (Figure 13 stage 4 and Figure 37 blue) discussed in the *Mipmap Strategy* section. In summary, the Figure 37 data shows that the method averages a memory gathering-bound 33.78ms per frame (approximately 30FPS) giving real-time performance for a massive static scene with a fixed camera.

Figure 38 shows the timing for 1,000 frames as a human navigates the scene in Figure 36 with static objects using a first-person camera controller. In the first 100 frames, the camera is facing a single shape that is isolated from the majority of the scene content. This is reflected by the high-performance (small *millisecond* values) in the graph where the shape is rendered at approximately 7.6ms per frame. In Figure 13 stages 4-5, the first iteration just contains the transformed shape SVO root voxels, where the majority of the 1,000,000 shape instances are labeled for removal (Table 1) as they are not within the viewing frustum. With the root voxels removed from the pipeline by stream compaction (Figure 13 stage 5) their children are no longer processed in the following iterations giving very fast frame rates. After the first 100 frames, the camera turns towards the scene content until about frame 200, where between 1,844 and 39,409 objects are visible on the camera path from frames 200 to 1,000, shown in the image in Figure 36. However, only a relatively small portion of the scene content is processed as the projection criteria (Table 1) prevents high-frequency distant shape geometry from being processed if their projected size is too small.

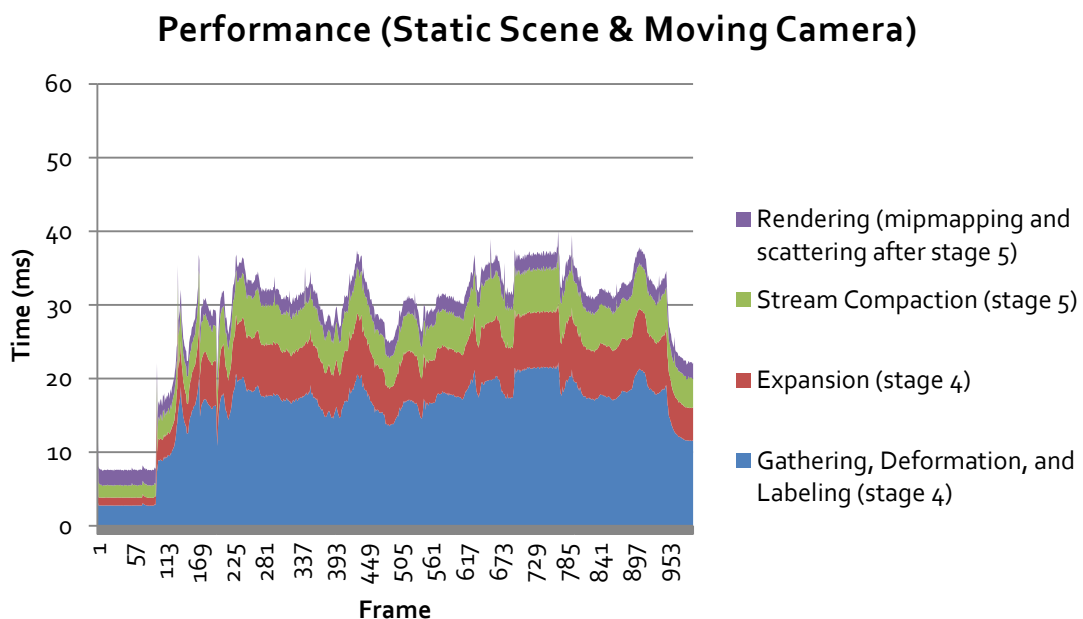


Figure 38: Timing for 1,000 frames with a massive static scene and a moving camera.

In the remaining frames 200 to 1,000 (Figure 38) the camera moves between and above the objects, where the compaction by the occlusion, frustum culling, and projection criteria (Table 1) remove noncontributing voxels and their associated children from the pipeline accordingly. This is shown by the fluctuation in the results with a worst-case of 40ms per frame (25FPS).

Figure 39 shows the animated scene in Figure 36 where the objects are assigned unique twist, shear, scale, translation, and walking deformations with a moving camera on a similar path of the measurements in Figure 38. The static scene and dynamic scene share similar performance (Figure 38 compared to Figure 39) however there is more fluctuation where fast-moving parts of the animated shapes suffer incoherency with the previous frames depth-buffer. This means that more shape information needs to be processed as the occlusion criteria fails to capture the incoherent regions (discussed later in the section on *Limitations*). However the longest frame is 52.9ms (18.9FPS) which is adequate considering the scene and animation complexity.

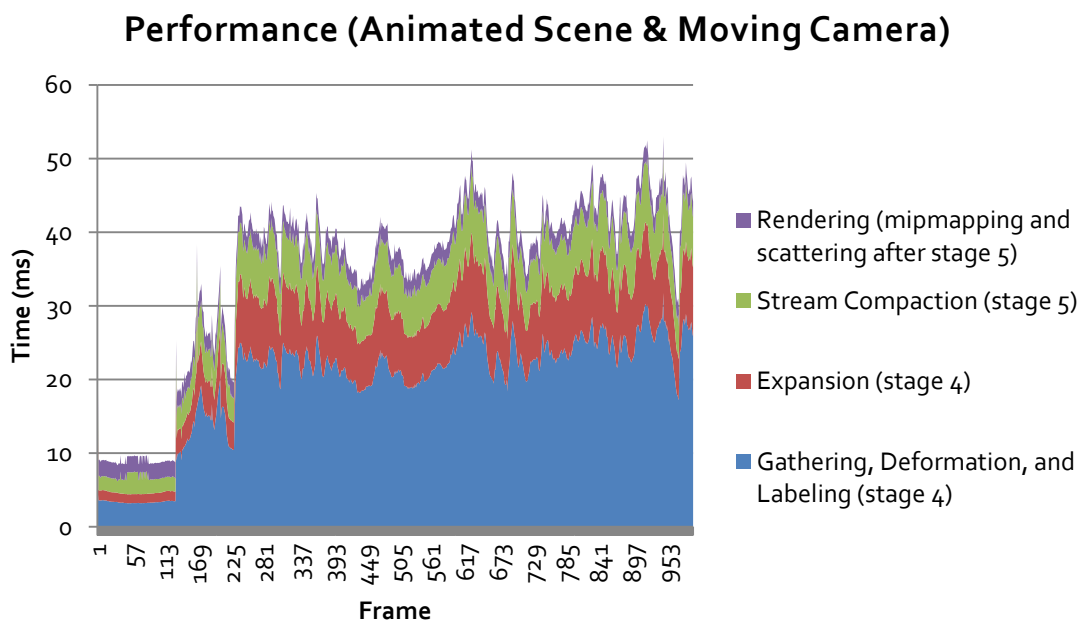


Figure 39: Timing for 1,000 frames with a massive animated scene and a moving camera.

In summary, the main observation is the harmony between the selection criteria in Equations (3.2.5) and (3.2.7). These criteria work together to efficiently reduce the amount of voxels for simulation, as the depth complexity is high where objects are densely packed (Figure 36) giving ideal conditions for the hierarchical occlusion culling (Equation (3.2.7)). On the other hand, if large numbers of objects are visible with low depth-complexity, the camera is generally further away from the objects, which gives ideal conditions for the projection criteria (Equation (3.2.5)). In the real world, scenes with high depth-complexity and massive numbers of visible

objects are rare. This means that the rate of which images are generated is mainly sensitive to the resolution of the selected information. The resolution (Equation (3.2.5)) can be desirably tuned  $p_{\text{size}} < \delta$  to reach the target frame rate or rendering accuracy. Lowering  $\delta < 1$  traverses the shape hierarchy to capture features smaller than a single pixel, whereas increasing  $\delta > 1$  terminates the traversal earlier and therefore less information is processed in the pipeline. This also means that fewer depth samples are required for the hierarchical occlusion culling.

The hierarchical construction (Figure 13 stages 6-7) and ray casting renderer is more expensive than rasterization, as it requires more computation (Figure 30) and processes more samples (Figure 29). It was observed that the same quality for the scene in Figure 36 can be achieved at 90ms; however in practice such a high-quality SVO is unnecessary for indirect illumination (Crassin et al., 2011). Investigating hybrid ray-traced and rasterized rendering remains an interesting area for future work. For example it may be possible to branch the pipeline at Figure 13 stage 5 after less iterations, in order to generate a lower resolution SVO efficiently, yet still use rasterization for direct illumination. In such an approach it may even be beneficial to remove the occlusion culling criteria from the first selection iterations of SVO construction. This would enable indirect light effects to be accurately modeled from hidden shape regions.

This thesis is accompanied by a set of videos that demonstrate the real-time results in Figure 1, Figure 7, Figure 33, Figure 36, and Figure 40. The video for Figure 7 and Figure 40 shows a typical scene consisting of several different models with unique deformations. The video for Figure 1 and Figure 36 shows an extreme example of a massive scene, which makes heavy use of instancing to process huge numbers of high-resolution objects. The remaining videos show extreme deformations on complex models such as in Figure 33. In summary, these videos demonstrate the robustness and scalability of the sparse volumetric deformation pipeline and its ability to render attractive and immersive images in real-time.

In conclusion, the efficiency of the rendered images mainly depends on the resolution in which shape information is selected for further processing (Equation (3.2.5)), and the pipeline itself is mainly sensitive to the memory bandwidth-bound gathering operations (discussed in Figure 37 and Figure 38) instead of the deformation calculations. However it is important to profile and optimize the entire workflow, as just a few hierarchical levels can have a large impact on the appearance of the generated images (Figure 35). Adjusting the resolution  $\delta$  gives users control over the amount of information selected and processed in the pipeline. This is highly desirable as it means that smooth frame rates can be guaranteed according to the capabilities of the target machine, allowing for immersion within the virtual environment.

## Discussion

This section compares the proposed sparse volumetric deformation and rendering approaches with state of the art methods from the main categories of shape representation. These are discussed in the paragraphs below: (1) polygon meshes with hardware tessellation, (2) out-of-core sparse voxel octrees, (3) large-scale point clouds, and (4) hybrid shape representation.

The graphics rendering pipeline is tuned for rasterizing polygon meshes (Lindholm et al., 2001). The addition of the tessellation stage enables surface representations to be sampled efficiently to produce more detailed meshes; therefore smooth surfaces can be represented more accurately near the viewing area, and similarly high-frequency details can be displaced with a heightmap texture. The limitations of this approach are that the rasterization and shading processes are optimized for large triangles causing redundant shading of micropolygons (Fatahalian et al., 2010). Furthermore, the underlying parametric surfaces do not support overhanging ledges, and may cause undesirable non-manifold self-intersections (Sigg, 2006), limiting the modeling of advanced deformations where topology may split or merge. Meshes with complex topology and internal features are challenging to accurately simplify in parallel for massive scenes; artists currently assist the generation of billboards for medium view ranges, and imposters for distant objects (Décoret, Durand, Sillion, & Dorsey, 2003), which is laborious, hinders animation, and causes undesirable popping artifacts between the detail transitions. In contrast, the proposed approach seamlessly selects and simulates deformation for important volumetric shape regions prior to rendering. This enables smooth transitions between the geometric resolutions, which are automatically created in the SVO hierarchy. The input shapes are not limited to surface representations, and easily support complex topology such as the tree model in Figure 1.

Sparse voxel octrees are rapidly gaining interest with advances in compact representations (S. Laine & Karras, 2011) and filtering (Heitz & Neyret, 2012). In particular, the work by (Crassin et al., 2009) build a GPU caching and on-demand loading mechanism that requires only a small subset of the total SVO to be kept in GPU memory, minimizing the amount of data streaming. The advantage of this technique is that massive scenes of highly complex shapes can be efficiently rendered with alias-free pre-integrated voxel cone tracing. The main disadvantages are that the on-demand loading scheme does not support deformation, and that to support animation the pre-filtering needs to be computed in a bottom-up process. Therefore a high-resolution voxelization is always needed, regardless of resolution actually required for rendering (Crassin et al., 2011), which is not scalable for large numbers of animated objects. In comparison, the proposed approach operates in a parallel top-down process on instanced SVO shapes, which simulates deformation only for the important volumetric content. This has two main advantages: (1) massive scenes need not be modeled with the constraints of a single SVO

structure, which means instancing can be used to efficiently reduce memory consumption in repetitive content. (2) Deformation is scalable, as it is only simulated for important shape regions at the resolution of which data is displayed. However, unlike (Crassin et al., 2009), transparent objects are inefficient with the current selection criteria, and therefore remain an area for future work, perhaps by considering other visibility feedback data from the renderer.

Point clouds are a flexible representation that can represent the surfaces of detailed shapes. Their main advantages are that they can be organized for efficient shape queries (Rusinkiewicz & Levoy, 2000) for example with bounding sphere hierarchy (BSH), and they can also be rasterized efficiently with the graphics pipeline. However efficiently modeling accurate deformation with point clouds is more challenging, as their irregular local sample spacing needs to be considered otherwise artifacts occur. In the literature (Pauly et al., 2003) address this problem for free-form deformation with moving least-squares projection, however this is not currently scalable for massive scenes of animated objects in real-time. In contrast (Marroquim et al., 2007) investigate an image space reconstruction technique, which is more efficient, but is unanimated and suffers from aliasing. Recently (Bautembach, 2011) propose using a regular sample spacing organized in an octree structure, and stretching the size of the primitives to account for deformation; however their approach suffers from gaps and distortion artifacts in extreme deformations, and the output is unaligned meaning that hierarchy cannot be easily reconstructed for more advanced rendering. In comparison, the proposed approach supports efficient and scalable deformation for important volumetric shape regions. These are aligned to a SVO structure for efficient hierarchical construction, without suffering from gap or distortion artifacts.

It is also worth considering hybrid shape representations. Recently, there have been several efficient real-time global illumination strategies that combine a high-resolution rendering for direct illumination, with automatically generated low-resolution hierarchical structures, such as BSHs or SVOs, for modeling indirect light transport (Crassin et al., 2011; Hollander et al., 2011). The disadvantage of these approaches is that the tree updates are performed in a parallel bottom-up order, to calculate the internal branches after animation, which means that a high-resolution shape is always needed. This greatly limits the scalability in massive scenes, for example when considering the visibility of distant objects. Furthermore, the limitations of the previously discussed shape representations still apply for the high-resolution rendering. In comparison, the proposed approach operates with a purely volumetric representation and simulates deformation in a top-down hierarchical selection algorithm. Therefore the hierarchy can be constructed in parallel only for the important content, when the desired selection iteration is reached. This strategy is more scalable, and yet it still allows for flexibility with the rendering approach.



## 4.6 Chapter Summary

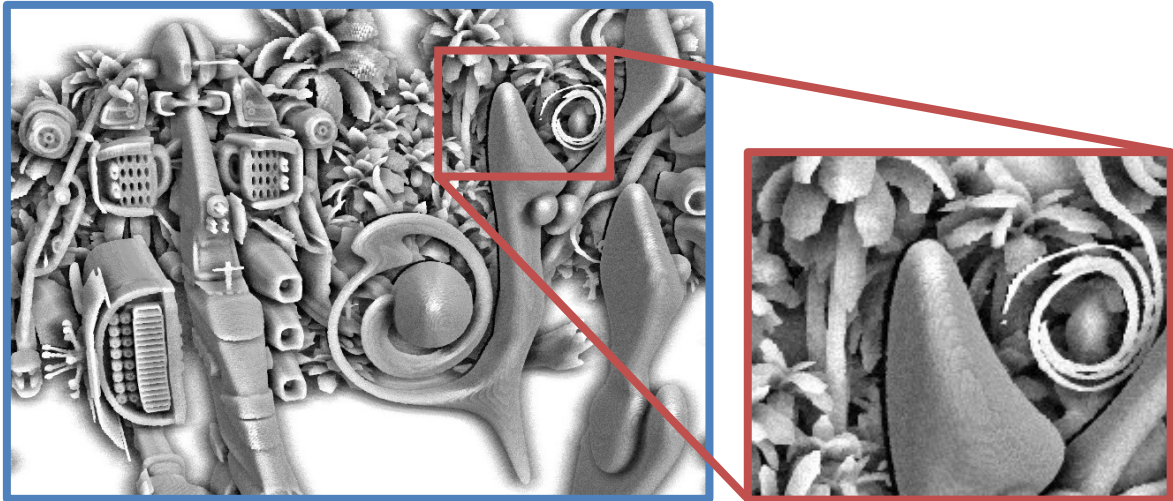
This chapter has presented a method for constructing SVO hierarchy in parallel (Figure 30) for the important deformed volumetric content (Figure 13). This means that shape queries can be made efficiently for massive amounts of animated objects, enabling real-time applications that would otherwise be restricted to small scenes. Furthermore, an efficient parallel rasterization approach has been presented (Figure 32). This is able to quickly display high-resolution content without requiring resampling or high-resolution hierarchy. The proposed rasterization and ray casting renderers share the same selection and deformation components, and therefore more sophisticated and flexible rendering strategies can be implemented efficiently.

### Contributions

This chapter has concentrated on efficiently rendering massive scenes of animated volumetric content. The first contribution addresses the problem of efficiently constructing hierarchy for large amounts of voxels, resampled at varying hierarchical levels. This enables support for the hierarchical construction of sparse scenes, for example where important content has smaller and denser sample distributions than distant shape regions. This is achieved by extending a fully parallel hierarchical construction pipeline, whereby leaf nodes are initially set at a uniform sample level. The sample spacing is retained throughout the parallel construction process, and the newly generated internal branches are flagged for termination where they are undesired. This approach maximizes the parallelism and preserves the varying sample hierarchy, which means that efficient ray casting with space leaping can be supported. The second contribution addresses the problem of efficiently rasterizing volumetric shapes at high-resolution, without requiring resampling or hierarchical construction. This gives flexibility in real-time rendering applications, for example where rasterization can be combined with a SVO generated more efficiently at lower resolutions for modeling light transport. The straightforward rasterization approach expensively accumulates and emits large numbers of primitives, whereas the proposed method utilizes a hierarchical mipmap strategy that enjoys the benefit of wide parallelism. This is therefore more scalable and compatible with SVO ray casting.

### Limitations

In the selection and deformation pipeline (Figure 13), occlusion culling is currently performed by sampling a hierarchical depth buffer of the previous frame (Equation (3.2.7)). The problem with this approach is that a small amount of error occurs, where no voxels are rendered at the incoherent region between frames, shown by the black region behind the shape in Figure 40. However it may be possible to mask this error with a motion blur effect, using (McGuire, Hennessy, Bukowski, & Osman, 2012). Investigating this effect is left for future work.



*Figure 40: A small region of error occurs behind fast-moving objects (black pixels).*

Existing out-of-core static SVO approaches (Crassin et al., 2009) are able to accurately page sections of a single master input SVO according to visibility information from the renderer. In contrast, the proposed approach selects information from multiple SVO instances, which gives greater modeling flexibility and does not consume additional memory for repetitive content. This also allows for more parallelism in the selection pipeline (Figure 13) especially in the initial iterations of the selection process. Unfortunately this approach is sensitive to the aggregation of the paged instance data, in comparison to streaming regions of a single SVO. In practice, the advantages outweigh the restrictions imposed by modeling within the constraints of the SVO structure; however enabling more parallelism for a gigantic unique single input SVO requires splitting the input into smaller SVOs. Ideally an automatic algorithm is needed to split and aggregate sections of large or small models, although investigating such advanced paging requires careful profiling of the target hardware, which remains an area for future work.

Furthermore, the current implementation does not consider filtering, and therefore the output images suffer from aliasing and an overall thick appearance. Future work needs to consider more accurate voxel representations, compression, and filtering; in the literature, work by (S. Laine & Karras, 2011) achieve approximately 5-bytes per voxel (1 byte for color, 2 bytes for the normal, and the remainder for geometry), whereas a much higher quality representation is achieved at about 15-20 bytes per voxel, featuring multi-scale geometry and view-dependent filtered RGB colors (Heitz & Neyret, 2012). Ideally, such methods need to be compared with more efficient 2D post-process filters, such as applying depth-of-field or blurring to reduce the aliasing of nearby voxels (S. Laine & Karras, 2011; Síleš, 2012)). It would be worthwhile looking at these screen-space techniques in more detail, in combination with more recent antialiasing (Jimenez, Echevarria, Sousa, & Gutierrez, 2012; Jimenez et al., 2011) comparing accurate filtering representations with highly-compressed approximations and post-processes.

# Chapter 5

## Skeletonization

### 5.1 Abstract

In the previous chapters, volumetric deformation and rendering strategies were proposed. This chapter looks at automatically acquiring hierarchical control skeletons for use in the sparse volumetric deformation pipeline, or for use in traditional polygon renderers. The chapter is based on the recently published work ‘*Feature-Varying Skeletonization*’ (Willcocks & Li, 2012). Hierarchical deformation plays an important role in computer animation for achieving accurate simulation in both organic and mechanical objects (Parent, 2012). In order to achieve efficient hierarchical deformation in complex or detailed models, a simplified control structure is used such as a cage or a skeleton. This work focuses on automatically generating skeletons that are mapped to the original model, which naturally captures hierarchy similar to the skeletons in nature. However, existing skeletonization algorithms strive to produce a single centered result, which is homotopic and insensitive to surface noise; this traditional approach may not well-capture the main parts of complex models, and may even produce poor results for animation. Instead, the proposed approach approximates the topology through a target feature size  $\omega$ , where undesired features smaller than  $\omega$  are smoothed, and features larger than  $\omega$  are retained into bones, as demonstrated in Figure 41. This relaxed feature-varying strategy gives robust and meaningful results without requiring additional parameter tuning, even for noisy, damaged, complex, or high genus models.

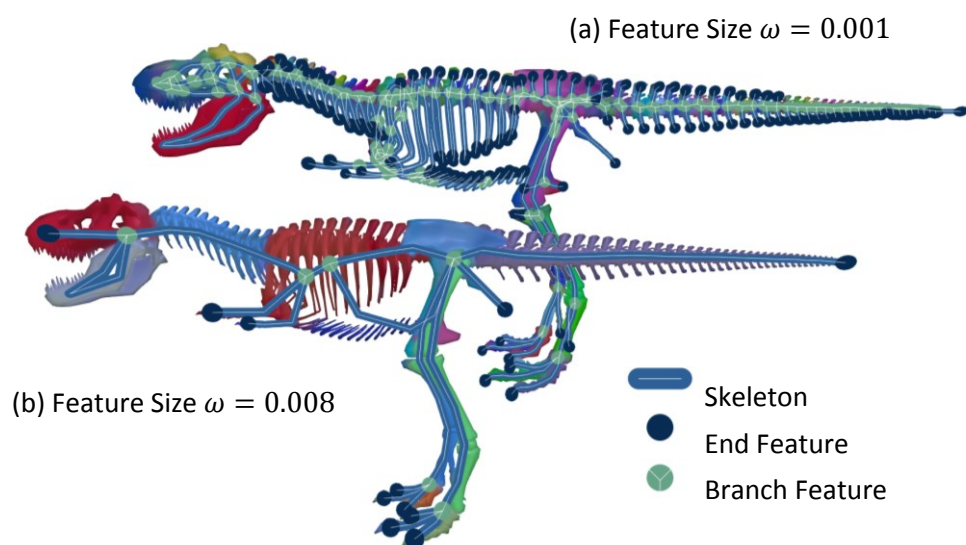


Figure 41: Two feature approximations for a complex model.

## 5.2 Background

Skeleton generation is a large research field (Biasotti, Attali, et al., 2008; Cornea et al., 2007) with multiple applications in animation, modeling, shape retrieval and segmentation. Many of these applications require a skeleton to describe only the main shape features, for example Figure 41b is more suitable for simple deformation than the homotopic and centered skeleton in Figure 41a. It can be more meaningful for a skeleton to describe features by a target size  $\omega$ , especially with complex models, such as vegetation or high genus parts.

Skeleton detail can be controlled by pulling nearby features of size  $\omega$  together; however doing this introduces topological error where nearby model regions are undesirably joined. Instead, the proposed strategy contracts the model into a skeleton, which causes the nearby model regions to naturally separate. Therefore features of size  $\omega$  can be merged during contraction itself, without close regions being undesirably joined. This new approach also means that the output skeleton can better describe complex models with multiple parts and non-manifold connectivity, as the topology is combined from relevant features of size  $\omega$  instead of from the potentially damaged input connectivity.

The existing skeleton generation methods are often unintuitive, with multiple parameters to address issues such as: ‘how are features pruned?’, ‘how to connect topological parts?’, and ‘how smooth is the skeleton?’ Instead the feature-varying method unites all these questions under a single target parameter  $\omega = \textit{feature abstraction size}$ , which allows users to control a contraction process that separates vertices directly into bones during the contraction process itself. This is an improvement over previous methods, as topological and spacing errors are avoided where the output skeleton is traditionally decimated into bones. The elegant handling of these challenging issues by  $\omega$  ultimately results in a more robust and simple algorithm, with less failure cases, as shown by the skeletons generated for different shapes in Figure 42. In summary, the main contributions of the proposed approach are listed as follows:

- (1) Unique topological abstraction (Figure 41).
- (2) Intuitive control over target feature size  $\omega$ .
- (3) Only two input parameters ( $\omega$  and a constant  $\delta$ ).
- (4) Direct contraction into a thin zero-volume skeleton.
- (5) Robust on extremely damaged inputs (Figure 42).
- (6) Precision over the removal of surface noise with  $\omega$ .
- (7) Efficient generation, where the main model features can be captured nearly instantly.

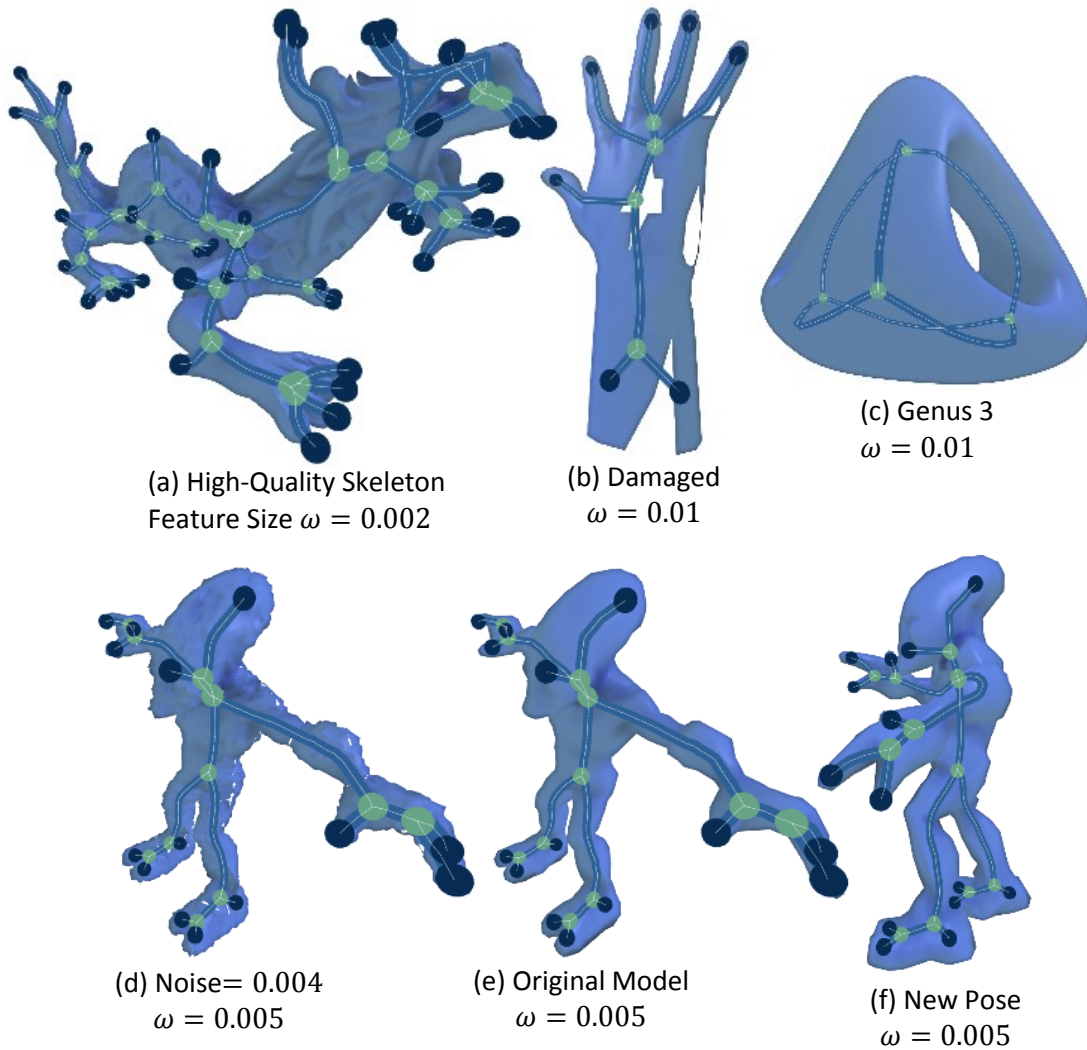


Figure 42: 'Feature-Varying Skeletonization' operates on a wide variety of input cases.

## Chapter Structure

The upcoming sections in this chapter follow with a discussion about *Related Work* in order to examine the problems and strategies encountered with existing methods. This leads into the development of a new strategy. The *Methodology* section breaks down the strategy outlined in the literature review, by providing a more detailed discussion on the core components of the skeletonization algorithm. This is followed by several *Applications* to demonstrate the flexibility and novelty in the proposed approach. The approach is then evaluated by measuring the performance and quality of the generated output, and by comparing it with existing state of the art methods. Finally, a conclusion is given which discusses the limitations of the algorithm, and highlights areas for future work.

## 5.3 Related Work

This section splits the literature according to the two general directions in which skeletons can be generated. They can either be: (1) grown by *propagating* through the input and connecting shape samples, or they can be (2) produced by *contracting* or thinning the entire input directly into a 1D curve. In both cases, the input can be either a surface (e.g. a polygon mesh) or a volume (e.g. a voxel image or a SVO), where this classification helps highlight related issues in topology, noise sensitivity, input requirements and parameter tuning.

### Propagation

Skeletons can be generated using a propagation strategy which reconstructs or grows connectivity from sets of surface features, or from critical points located inside the model. Approaches include searching for and connecting cutting-planes (Tagliasacchi, Zhang, & Cohen-Or, 2009), critical-features (Cornea, Silver, Yuan, & Balasubramanian, 2005; Ning et al., 2010), reeb-graphs (critical points on level sets) (Aujay, Hetroy, Lazarus, & Depraz, 2007), geodesic features (Oda et al., 2006), and internal vector-field features (Hassouna & Farag, 2009; Pantuwong & Sugimoto, 2010). From this selection, the methods which connect critical points inside the model (Cornea et al., 2005; Hassouna & Farag, 2009; Pantuwong & Sugimoto, 2010) require solid voxelization, which merges small details and changes the output topology according to the input resolution. Such an approximation can be useful as it may simplify undesired topology in complex models, which can be controlled by adjusting the voxel resolution, however this is often considered a limitation (Brunner & Brunnett, 2005) because nearby mesh parts are undesirably joined introducing topological error. On the other hand, the propagation methods which connect features on the surface (Aujay et al., 2007; Ning et al., 2010; Oda et al., 2006; Tagliasacchi et al., 2009) do not require voxelization, but such a surface selection offers little control over the spacing or size of the output bones.

An exception is the discrete scale axis (Miklos, Giesen, & Pauly, 2010), although only a medial representation, it gives meaningful control over feature abstraction by growing a spherical approximation of the shape boundary, which uses sampling (Boissonnat & Oudot, 2005) to process voxel images, polygonal meshes, and level sets. Additional methods which allow control over the shape approximation include adaptive feature preserving surface extraction (Ho, Wu, Chen, Chuang, & Ouhyoung, 2005) and multi-resolution structure preserving representations (Marinov & Kobbelt, 2005), however, neither these nor the discrete scale axis approximation can be used as a preprocess to improve the feature selection of propagation methods, because the search functions (cutting-planes, critical-points, reeb-graphs, etc) are designed to be insensitive to the shape boundary.

## Contraction

Skeletons can also be produced by reducing the entire input surface or volume into a thin curve. Methods are therefore often homotopic to the input geometry (Au et al., 2008; Y.-S. Wang & Lee, 2008), but sensitive to the original connectivity. This can be addressed by applying remeshing as a preprocess (Cao et al., 2010), but as with previous propagation methods, topological error occurs where there are nearby separate features (Arcelli, di Baja, & Serino, 2011; Brunner & Brunnett, 2005; L. Liu et al., 2010), and it introduces further parameter tuning to choose the sample or voxel resolution. Thinning methods are often simple (L. Liu et al., 2010) allowing for parallel implementation, however they produce pixilation artifacts and are sensitive to the object rotation (Brunner & Brunnett, 2005; L. Liu et al., 2010). Recently (Arcelli et al., 2011) use a weighted distance metric which is less sensitive to rotation, however these methods require a properly connected watertight surface for solid voxelization.

The contraction method by (Dey & Sun, 2006) defines a skeleton as a subset of the medial axis, which is similar to (L. Liu et al., 2010) who use a medial quality criteria, and guarantees centeredness. Both use a form of erosion to remove undesired features from surface noise, yet (Dey & Sun, 2006) relies on a geodesic metric which limits it to a connected boundary, and (L. Liu et al., 2010) requires voxelization. Geometric contraction (Au et al., 2008; Cao et al., 2010) does not require solid voxelization, and operates by smoothing (Desbrun, Meyer, Schroder, & Barr, 1999) applied with different weights over multiple contraction iterations. This approach is both homotopic and insensitive to surface noise, however the smoothing process requires users to manually set the contraction weights, which produces non-thin skeletons that require further decimation and postprocessing, and the method offers little control over the size of the output features or skeleton topology. While the proposed feature-varying skeletonization method is a contraction strategy, it is able to bypass many of the limitations in the existing literature by selecting nearby features during contraction itself. This is discussed in more detail in the later sections.

## Other Approaches

Instead of *contraction* or *propagation*, skeletons can be embedded from training data (Baran & Popovic, 2007), modified manually, sketched, or generated from training poses (Hasler, Thormählen, Rosenhahn, & Seidel, 2010; He, Xiao, & Seah, 2009) to give improved results. For example reeb-graphs (Biasotti, Giorgi, Spagnuolo, & Falcidieno, 2008) can be constructed on a harmonic function of poses to provide pose-invariance (He et al., 2009). However these methods are too expensive in terms of computation and data collection as they require additional poses, user-intervention, or a training database, to generate outputs.

## Proposed Strategy

In the literature, there is a circular problem where using *nearby feature approximation*, for example a form of remeshing, is required to control the output topology, noise insensitivity, and feature size. But nearby feature approximation itself introduces topological error by joining nearby separate features, and it requires additional parameters to control the voxel resolution or sample spacing. This work looks at a window of opportunity to perform the feature approximation inside a contraction method, instead of as a remeshing preprocess. More specifically, nearby features may be pulled apart first through smoothing (Desbrun et al., 1999), before being merged into the corresponding skeleton bone. This strategy avoids topological error, while still providing noise insensitive results without requiring additional parameters or a fully connected boundary.

## 5.4 Methodology

Given an input triangle mesh or voxelized model, this section introduces a novel algorithm that updates the triangle vertices (or voxel centers) in an iterative contraction process. This moves the vertices towards the shape centerline, and separates them into groups called *bones*. The input mesh connectivity (or adjacent voxel neighbors) is maintained through the contraction process (Figure 43 middle); therefore the output bones are already connected as a skeleton (Figure 43 right). This combined approach means that, unlike previous methods, additional parameters are not required to control the skeleton smoothness or connectivity, resulting in a more robust and simple algorithm.

Figure 43 shows the iterative contraction process on a model of the human hand. The first five images show frames where the input shape (left) is contracted (middle) towards the shape centerline (middle-right). The image on the far right highlights topological branches and end features in the generated output.

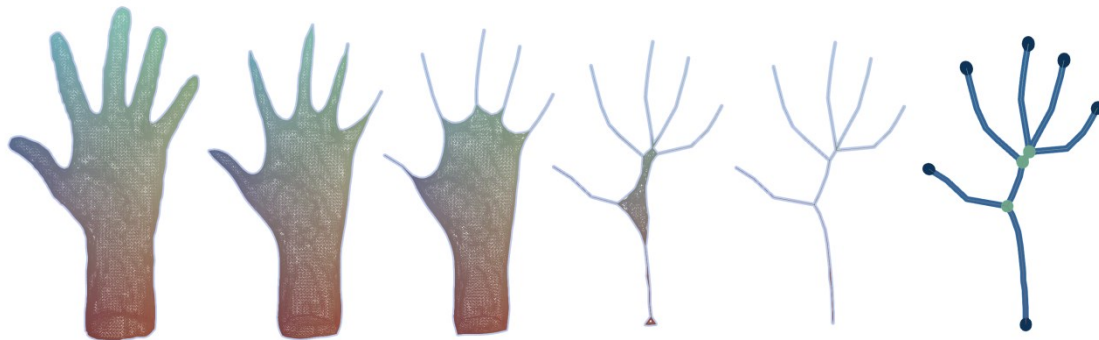


Figure 43: The proposed contraction process operating on a model of the human hand.



The previous literature section concluded with a contraction strategy that initially pulls nearby features apart by smoothing them, and then merges them together into bones. The merge operation is based on Euclidean distance, which allows for accurate topological control as with other surface feature approximation methods (Boissonnat & Oudot, 2005; Miklos et al., 2010). But because merging is performed inside the contraction process, it only influences features which are already pulled apart from smoothing. This means that topological error can be avoided where nearby features are undesirably joined, while retaining the advantages of having accurate control over feature size, noise insensitivity, and low parameter requirements.

It is not possible to smooth the entire mesh for several iterations, and then later switch to a Euclidean merge operation, because some parts of the mesh may contract before others. Therefore the smoothing and merging processes are interpolated based on a metric which determines how much the local mesh region has contracted. To clarify between smoothing and merging the terms *local* and *nearby* are used, where ‘locally adjacent’ vertices are those that share an edge in the original connectivity (also called *one-ring neighbors*, or adjacent voxels) and ‘nearby vertices’ are those close in Euclidean distance.

*Assumption:* For simplicity of explanation, it is assumed that the input is a uniformly sampled mesh such that the length of any edge is  $< \omega$ , for example by recursively subdividing any triangles whose edges are greater than  $\omega$ , replacing them with the medial triangle and three respective corner triangles. However in the *preprocessing* section, more inputs are discussed.

## Smoothing

The umbrella operator is the simplest form of smoothing. It operates by updating each vertex  $v$  to the average of its locally adjacent one-ring neighbors  $\mathbf{L} = \{v^i\}$

$$s = \frac{1}{|\mathbf{L}|} \sum_{i \in \mathbf{L}} v^i \quad (5.4.1)$$

This operator requires uniform sampling; otherwise it produces sliding and distortion artifacts. These can be removed by introducing cotangent weights as with the discrete Laplace operator (Desbrun et al., 1999). However both of these approaches only operate locally, which makes them unsuitable to pull nearby parts of the mesh together into a skeleton, such as at topological branches. Therefore a different domain is needed.

## Merging

To create a tight one-dimensional skeleton, nearby parts of the mesh can be pulled together into the same position by using the fixed-radius nearest neighbors of the contracting shape

regions. The merged positions  $\mathbf{m}$  share a similar definition to smoothed positions  $\mathbf{s}$ , but using a different domain: Let  $\mathbf{G} = \{\mathbf{v}^j\}$  be the set of any nearby neighbors from the input mesh within Euclidean distance  $\omega$  of  $\mathbf{v}$ . Therefore each vertex  $\mathbf{v}$  is updated by averaging, where:

$$\mathbf{m} = \frac{1}{|\mathbf{G}|} \sum_{j \in \mathbf{L}} \mathbf{v}^j \quad (5.4.2)$$

**Advantage 1:** Merging causes vertices to pull together tightly into groups (these become *bone groups*) because it is influenced by any nearby part of the mesh. This is in contrast to the discrete Laplace operator which undesirably pulls branches apart as a side effect of smoothing.

**Advantage 2:** Additionally, merging stops contracting when groups are formed, because the spacing between vertices outside the group becomes greater than  $\omega$ . This prevents the oversimplification of end features in the output skeleton, which is commonly seen in traditional smoothing operations. Also the averaging does not have smooth weights, which means that it stops contracting immediately when groups are spaced  $> \omega$ .

**Advantage 3:** The parameter  $\omega$  determines the distance in which features are pulled together, which implicitly reflects the size of the output features.

**Disadvantage:** Merging pulls vertices together too quickly, causing groups to form on the surface of the mesh instead of only on the skeleton. This means that contraction may terminate before thick regions of the model are able to form a skeleton, because the groups on the surface may have spacing greater than  $\omega$ . In order to eliminate this disadvantage, the antagonistic relationship between the smooth and merge operations is exploited in regard for the skeletonization problem. These properties are outlined in Table 4:

Smoothing	Merging
Pulls branches apart	Pulls branches together
Smoothens features	Retains end features
Separates vertex groups	Creates vertex groups

Table 4: Antagonistic properties between smoothing and merging.

It is observed that smoothing can be used to separate vertex groups that have prematurely formed on the surface of the mesh. This means that it can be used in combination with the merge operation in order to ensure the continued contraction of the mesh. However this also

introduces the new problem of deciding how to balance the combination of the two operations to give continuous contraction into a tight skeleton.

## Interpolation

The influence of the merge operation must dominate the smooth operation at locally thin regions of the contracting mesh, otherwise the smooth operation will pull apart topological branches giving a non-thin skeleton. In harmony to this, the smooth operation can dominate the merge operation at thick regions of the contracting mesh in order to eliminate surface noise and continue the contraction process. Therefore the *contraction proportion* needs to be determined in order to distinguish between the thick regions of the contracting mesh that require smoothing, and the thin regions of the contracting mesh that require merging. The output skeleton must be zero-volume with a corresponding total surface area  $t = 0$ , whereas the contracting mesh will have a total surface area  $t > 0$ , where:

$$t = \sum_{i \in \mathbb{N}} a^i \quad \text{where } a^i = \text{area of one-ring-neighborhood.} \quad (5.4.3)$$

Respectively, on examining the one-ring-area of each contracting vertex  $a$  with its original one-ring-area  $\rho$ , it is observed that  $a/\rho = 0$  where the vertex  $v$  is locally *thin* and therefore part of the skeleton. Similarly, it is observed that  $0 < a/\rho \leq 1$  for all parts of the mesh which have not yet formed a skeleton, e.g. the *thick* parts of the mesh. Therefore  $a/\rho$  can be used to decide *how* to interpolate between the smooth and merge operations.

For each contraction iteration, let  $s$  and  $m$  be the new vertex positions for the smooth and merge operations respectively. The following are the four required prerequisites defined for the target interpolation between the two operations:

1.  $v = m$  where  $a/\rho = 0$
2.  $v = m + \beta(s - m)$  for  $0 < \beta \leq 1$  where  $a/\rho > 0$

This means that: (1) the merge operation must dominate contraction where the vertices have formed a skeleton, and (2) there must be some influence of the smooth operation where the vertices have not yet formed a skeleton. In its most simple form, a linear interpolation can be used to satisfy these conditions and perform skeletonization:

$$v = m + \frac{a}{\rho}(s - m) \quad (5.4.4)$$

The linear interpolation in Equation (5.4.4) means that ‘smoothing’ dominates ‘merging’ more strongly in the initial contraction iterations, which results in the algorithm being highly insensitive to noise. More formally, it satisfies:

3.  $\mathbf{v} = \mathbf{m} + \beta(\mathbf{s} - \mathbf{m})$  where  $\beta$  is large where noisy.
4.  $\mathbf{v} = \mathbf{m} + \beta(\mathbf{s} - \mathbf{m})$  where  $\beta$  is small where there are features.

That is  $\beta$ , or the *thickness* of the contracted region, is large where there is noise, and  $\beta$  is small where there are features: (3) high frequency surface features are smoothed whereas (4) the low frequency shape features are captured and separated into bone groups by the merge operation. It is possible to use more sophisticated forms of interpolation between  $\mathbf{m}$  and  $\mathbf{s}$  which satisfy the four interpolation prerequisites, but these only influence the convergence properties of the contraction process. The other forms of interpolation would be unable to control the size of features or noise selection, which is instead controlled by varying  $\omega$  to influence the groups formed by the merge operation.

## 5.5 Pipeline

### Skeletonization

Skeletonization proceeds as an iterative contraction process which repeatedly applies Equation (5.4.4) to all vertices in the mesh. An automatic termination condition is required to stop contraction, which is challenging because the one-ring areas of vertices are sensitive to small fluctuations during smoothing, and the topology of the input mesh. These small fluctuations are caused by tension in the non-thin topological branches, where the smooth operation tries to pull branches apart just before they are tightened into a group by the merge operation. Therefore contraction cannot be terminated simply when the total surface area (Equation (5.4.4)) of the contracted mesh is equal to 0.

### Termination

To address this problem an elegant property of merging is exploited. Vertices stop moving when they converge on their respective bone groups, as the spacing between the groups becomes greater than  $\omega$ . Let  $\mathbf{p}$  and  $\mathbf{v}$  be the position of each vertex before and after applying Equation (5.4.4) at each iteration. The contraction process can be therefore terminated when the Euclidean distance  $\text{dist}(\mathbf{p}, \mathbf{v})$  becomes insignificant for all the vertices:

$$\sum_{i \in \mathbb{N}} \text{dist}(\mathbf{p}^i, \mathbf{v}^i) < \delta \quad (5.5.1)$$

This condition is based on changes in the vertex positions directly. Also  $\delta$  is not sensitive to the topology or surface area of the input mesh, as with an area-based termination condition, because the merge operation eventually dominates the contraction process. However  $\delta$  is sensitive to the size of the input model, and therefore the mesh needs to be transformed to fit within the unit cube  $[0,1]$ . This also allows  $\omega$  to capture similar-sized features for different model proportions, making the range of target feature sizes:  $\omega = 0 \dots 1$

## Preprocessing

The input mesh vertices are transformed such that the maximum side of the model's bounding box is 1. This means that a constant value for  $\delta$  can be used to support all sizes of model. In the implementation, it was found that  $\delta = 0.05$  stops extremely intuitively, and that the contraction process stops without delay after the skeleton is visually zero-volume. Initially it was assumed that the input mesh is uniformly sampled. However applying uniform sampling to a polygon mesh can cause simplification of features or other topological information. However it is observed that the merge operation requires nearby neighbors within distance  $\omega$  and that it does not require regularity between the vertices or edge lengths. In the sparse volumetric deformation pipeline, the input SVOs are already uniformly sampled. Recall that  $\omega$  is specified to reflect the target feature size, therefore the smallest value for  $\omega$  can be automatically determined based on the resolution of the volume image. More specifically, for an input SVO of depth  $d$ , it is suggested to choose  $\omega = 2^{-d}$  to capture features equivalent to the size of each grid region. However the automatic selection of  $\omega$  is not mandatory as the user or application may manually vary  $\omega$  in order to control the output level of feature abstraction, as demonstrated in Figure 41.

Unfortunately, adjacent voxels (belonging to any of the 26 possible neighbors in 3D) include shape regions for nearby topological parts. In the literature, this was investigated by (Brunner & Brunnett, 2005) who examine the input primitives surface normal to disconnect topology in separate regions. This extends well for assuming uniform sampling in voxelized polygon meshes in order to calculate the one-ring-neighbors (Equation (5.4.1)), however in practice most SVOs are of sufficiently high resolution where this is not required (for example the shapes in Figure 1). The method operates only on nearby or locally adjacent data, which means there is no need for a fully-connected input. For example, multiple separate models can be placed in the same scene and the algorithm can simultaneously generate the corresponding skeletons in a single contraction process. This is one of the main advantages of contraction strategies.

## Postprocessing

The output of the contraction process is visually a thin skeleton, as all parts of the mesh have been merged into groups with zero area, and have kept their original edge connectivity. Therefore a connectivity-surgery procedure is not required to convert the mesh into a skeleton (Au et al., 2008). Instead, given that the spacing between each of the output vertex groups is less than  $\omega$ , a simple weld operation can be performed for any vertices within distance  $\omega$ , which collapses and removes duplicate edges from the mesh connectivity. This operation causes nothing to change visually: it simply removes duplicate *merged* vertices and edges in the output skeleton. It is important to keep track of any weld operations to create a skeleton-mesh mapping between the new bones and the original vertices, for extended applications.

## Parameters

The method only has two parameters:  $\omega$  is a value between 0 and 1 which corresponds to the size of features to be merged, generating unique and meaningful topology (Figure 41 and Figure 42). And  $\delta$ , which is a constant, determines when to automatically terminate contraction. In the implementation, it was observed that setting  $\delta = 0.05$  is effective. That is, the value 0.05 is sufficiently small for the postprocessing weld operation to remove duplicate vertices and corresponding connectivity from the output skeleton.

## Summary

The complete pipeline can be summarized in the following high-level pseudocode (Figure 44). This shows the simplicity of the method, with preprocessing (lines 2-3), iterative contraction (lines 5-10), termination (line 4) and postprocessing stages (line 11) discussed previously.

---

```
1  Skeletonize (Mesh mesh, float omega = 0.01f, Skeleton &out) {
2      mesh.FitToBox(1);
3      mesh.Tessellate(omega);
4      while (mesh.NotMovedBy(0.05)) {
5          foreach (Vertex v in mesh) {
6              Vertex s = Smooth(v);
7              Vertex m = Merge(v, mesh, omega);
8              v = m + (v.GetArea() / v.OriginalArea) * (s - m);
9          }
10     }
11     out = mesh.Weld(omega);
12 }
```

---

Generate a skeleton &out given an input mesh and target feature size parameter omega.

---

Figure 44: High-level pseudocode for the complete pipeline.

## 5.6 Applications

There are many applications given the simplicity of the proposed method, and its high output yield. This section describes a small selection of them, including: (1) refinement, (2) automatic segmentation, (3) automatic skinning, (4) solid voxelization, and (5) reconstruction.

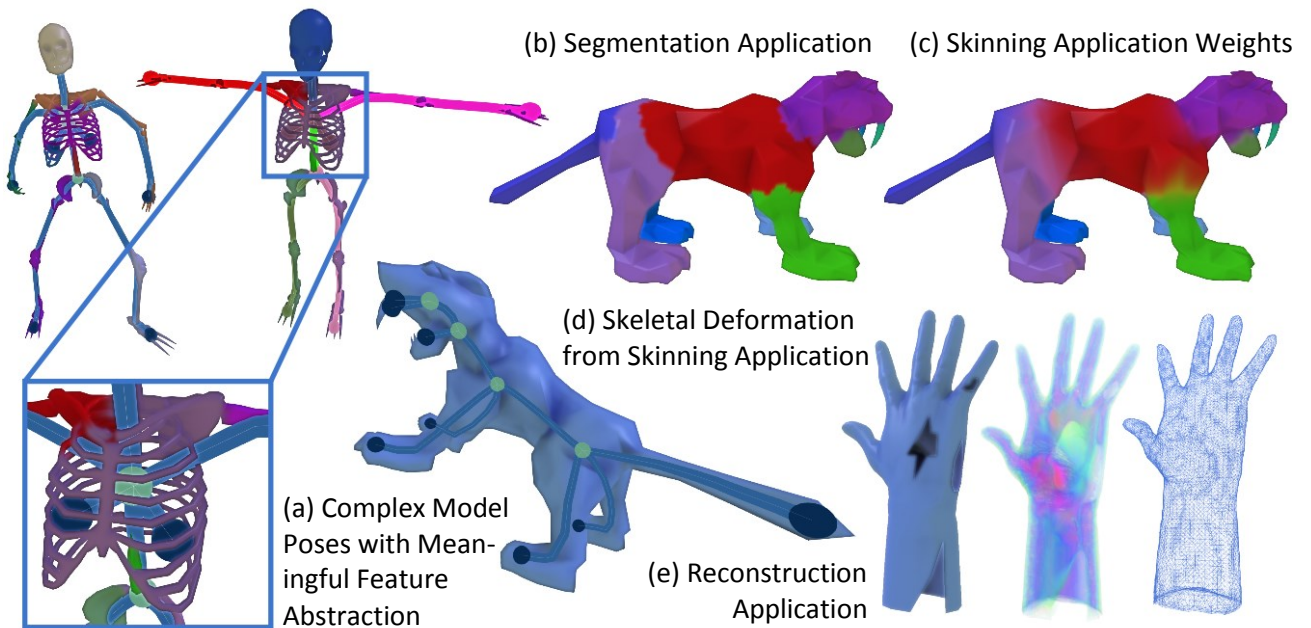


Figure 45: Applications: (a) shows the robust handling of a complex model in two different poses. (b) to (e) demonstrate other applications.

### Refinement

The low parameter requirements enable a unique and interactive refinement application that reprocesses specified skeleton segments with new feature sizes, and automatically combines the output with the original skeleton. This is shown in Figure 46. Such an application is useful in modeling or animation, where an artist can have precise control over the detail at different parts of the skeleton, and make fast incremental changes without re-skeletonizing the entire model. The procedure is:

1. Select bones in the skeleton to refine.
2. For each selected bone, mark its vertices on the input mesh from the skeleton-mesh mapping.
  - (i) Reprocess the full algorithm at a different resolution  $\neq \omega$  using only marked vertices as input.

- (ii) Merge the old skeleton with the new output.

Implementation is mostly straightforward, with the exception of merging the two output skeletons. This is challenging because a cut was made in the original mesh which contains undesirable sharp features which are retained during contraction (Figure 46a). To address this problem, vertices which are locally adjacent to the initial selection in Step 2 are highlighted. These highlighted vertices get reprocessed as with Step 2i, which means there is a shared highlighted region for both skeleton outputs. The new highlighted skeleton bones can be discarded, and the connectivity of any bones which are adjacent to the highlighted regions (from both skeletons) is joined to produce the final output (Figure 46b).

Refinement can therefore be used as a means to fine-tune and make small incremental changes to the output skeleton. It is also possible to use an automatic selection criteria in Step 1 to generate different results, such as selection based on: (a) end-bones (with only 1 adjacent edge), (b) the angle between bone edges, or (c) where bones leave the exterior of the input mesh surface; however a detailed evaluation of the performance and quality of such results is beyond the scope of this work.

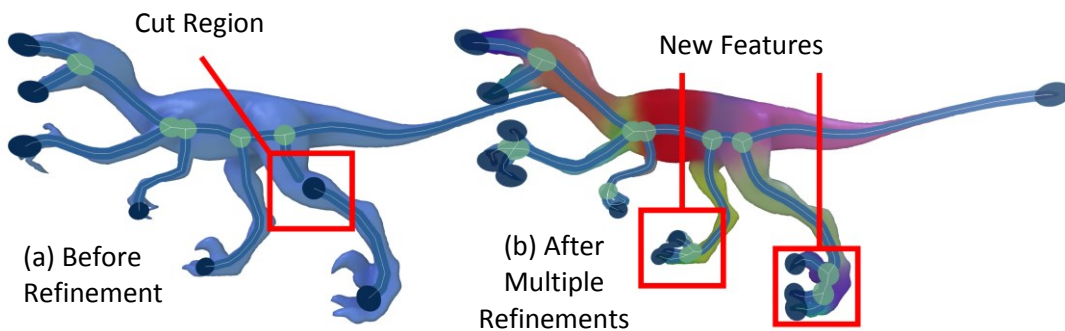


Figure 46: Application showing automatic refinement of multiple end features.

## Segmentation

An automatic segmentation application can be created by setting the skeleton bones with colors, which are mapped onto the input mesh (Figure 45b). The bones directly correspond to the target feature size  $\omega$  and therefore produce unique and meaningful segmentations in complex models (see coloring in Figure 41b & Figure 45a). The output skeleton is an undirected graph of vertices and edges: label vertices with 1 adjacent edge as *end features*. Also identify *branches* as any vertices with 3 or more adjacent edges (Figure 41). These two sets can be colored to produce a feature extraction algorithm. If every *end* and *branch* feature is colored randomly, segmentation is a simple recursive function that colors vertices which are not yet



visited by their adjacent (parent) color. Similarly *hierarchy* can be extracted for the skeleton by using Dijkstra's algorithm, where end features are used as seeds (source vertices) for the input.

## Skinning

Skinned skeletons are the industry standard for efficient hierarchical animation, and are easily extendable to the GPU for real-time deformation (Lindholm et al., 2001; R. Y. Wang, Pulli, & Popovic, 2007). The accompanying video shows smooth animations produced by automatic skinning: the skeleton-mesh mapping is used to transfer the segmentation assignments as hard skinning weights, which are later smoothed using their locally adjacent neighbors to produce softer weights. The output deformation from the soft weights is suitable for organic animation (Figure 45b→d).

## Solid Voxelization

The proposed method does not require a fully-connected boundary, and operates on damaged meshes (Figure 41 & Figure 42b) with control over the feature size through  $\omega$ . This makes it suitable for an intuitive solid voxelization application. At each contraction iteration, the input triangles  $T$  are voxelized with the updated vertex positions  $v \in T$  (after Equation (5.4.4)) to produce a solid voxelization of the input mesh (Figure 45e). Initially a 3D grid is allocated with voxels of size  $\omega$ . The rasterization process sets the corresponding corner voxels of  $T$ , and it sets the midpoint voxels from the medial triangle of  $T$ . Rasterization is then recursively called on the medial triangle and three corner triangles, while the distance between any of the corner voxels is greater than  $\sqrt{3}$ , that is the direct neighbors in all directions in 3D voxel space. The output solid voxelization has applications in parameterization, visualization, and skinning. It is also used as input for the thinning method shown in Figure 47e.

## Reconstruction

Solid voxelization (previous) can be produced from meshes with large holes even missing entire sides, which is appropriate for a reconstruction application. To reconstruct a damaged mesh, the boundary voxels are first labeled from the solid voxelization. Then, the voxelization process is repeated for the input surface, which is used to unlabel boundary voxels, leaving a new set of unlabeled boundary voxels at the positions of missing data in the original model. Surface meshing is applied to the union of both boundary sets giving new connectivity without holes. The input, voxelization, and output are shown in Figure 45e.

## 5.7 Evaluation

In this section, the performance and quality are evaluated for the feature-varying method. This is followed with a visual comparison and discussion with other approaches from state of the art skeletonization literature.

### Performance

The merge operation requires computation of nearest neighbors within a fixed radius  $\omega$ . These are expensive to compute, requiring an acceleration structure such as a tree or a grid, to produce acceptable performance for inputs over a few thousand triangles. Results in this paper are based on an optimized implementation of the C++ ANN library (Mount & Arya, 2010) which uses a combination of kd-trees and box-decomposition trees to perform the fixed-radius search procedure. The simplicity of this method allows for the computation of the updated vertex positions to be done in parallel, as only nearby vertex information is required from the fixed-radius acceleration structure.

Model	# triangles	$\omega = 0.025$		$\omega = 0.01$	
		volume	time	volume	time
<b>Alien</b>	2,618	871	0.7s	15,662	4.6s
<b>Armadillo</b>	69k	4,988	3.3s	52,839	22.5s
<b>Dragon</b>	3.6mil	5,535	6.1s	58,565	24.0s
<b>Horse</b>	10k	3,224	2.0s	43,860	10.2s
<b>Raptor</b>	16k	367	0.5s	4,586	1.5s
<b>Sabretooth</b>	1,156	1,286	0.9s	17,437	3.9s

Table 5: This table shows performance and mesh volume in relation to the feature-size  $\omega$ .

Table 5 shows the performance of the proposed method (preprocessing, iterative contraction, and postprocessing) using a 2.66 GHz processor with 8 processing cores and 4GB RAM.

The proposed method generates high-quality skeletons of feature size  $\omega = 0.01$  within a few seconds, and it can nearly instantly produce simple skeletons of feature size  $\omega = 0.025$  which only capture the primary model topology. Comparatively, the dragon takes greater than 10 minutes to process with previous geometric contraction (Au et al., 2008), whereas the current implementation uses only 530mb memory to store the dragon's vertices and acceleration structure, which is less than an unoptimized  $1024^3$  voxelization containing undesired aliasing. It is observed that larger values of  $\omega$  result in faster contraction convergence; however the method is reasonably insensitive to the number of triangles as the merge operation pulls larger

features together in fewer iterations. Performance has a strong correlation to the *volume* of the mesh and a weak correlation to the number of triangles. This is seen in Table 5 where the *volume* columns show the number of voxels of size  $\omega$  that are used to construct the solid representation of the model (this is calculated as discussed in the applications section: *Solid Voxelization*). The raptor model has a small volume but a large triangle count, and is shown to contract much faster than the sabretooth, which has a large volume but a small triangle count.

## Quality

The output skeleton is discussed according to the generally-accepted properties (Cornea et al., 2007) that an ideal skeleton should have.

*Thinness:* Skeletons are zero volume with an area sum of 0, therefore the *thinness error* is measured according to the amount in which the area has reduced from its original value. The error should always be zero, where:

$$\text{Error \%} = 100 \sum_{i \in \mathbb{N}} a^i / \sum_{i \in \mathbb{N}} \rho^i \quad (5.7.1)$$

More specifically, the sum of one-ring neighborhood areas, after applying Equation (5.4.4), is divided by the sum of the original areas. The method continuously contracts while the area is  $> 0$ , and terminates when all contraction movement stops (Equation (5.5.1)). Therefore it always produces 0% error in Equation (5.7.1), and a completely thin 1D skeleton.

*Topology:* Contraction interpolates between the merge and smooth operations. The merge operation pulls together nearby features of size  $\omega$ , which does not guarantee there being the same number of connected components, or at least one loop for every tunnel or cavity in the input mesh, therefore it is not homotopic. Instead the generated topology relates more meaningfully to the target feature size. This can be seen in Figure 41b & Figure 45a, where the skeletons better represent the general shape of the input model. This is more useful for animation applications, and in creating different levels of skeleton detail for efficient real-time rendering (Lindholm et al., 2001) or on-demand download (F. W. B. Li, Lau, Kilis, & Li, 2011) as required by online gaming systems. In contrast, homotopic skeletonization may cause disconnection artifacts where the mesh is not watertight (Figure 47b middle), however the merge operation better approximates the shape according to nearby features (Figure 47a middle), and can even preserve internal loops (Figure 47a top) according to the desired target feature abstraction. This gives unique topology which is highly suitable for applications including deformation and shape comparison.

*Centeredness:* Changing the feature abstraction and topology through  $\omega$  creates skeletons of varying centeredness within the model. This is clearly shown in Figure 41b and Figure 45a, where the skeletons leave the mesh boundary in order to better capture the specified feature size. However an attractive property of the method is observed where both the *homotopy* and *centeredness* consistently improve with smaller values of  $\omega$ , as the merge operation can only bridge topology in features which are smaller than  $\omega$  (e.g. surface *noise*).

*Noise Invariance:* The smoothing operation leads the initial contraction iterations (Equation (5.4.4)) which removes small features or surface noise, before the merge operation is able to capture them. This process is controlled by a target feature size  $\omega$  where features of size  $< \omega$  are removed and features of size  $> \omega$  are retained. This is demonstrated in Figure 42d and e, where noise was introduced by randomly displacing vertices by 0.004, and using a value of  $\omega = 0.005$  for the algorithm.

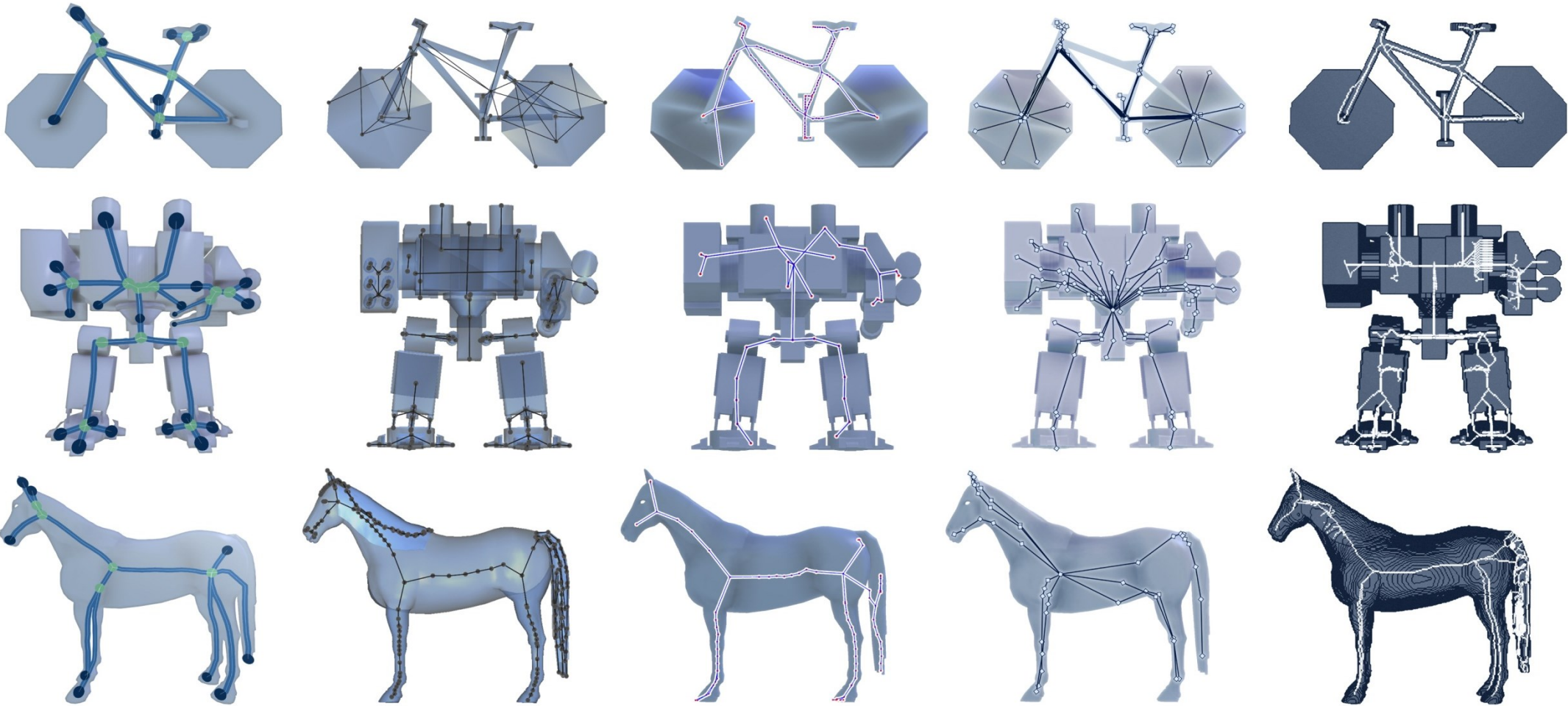
*Isometric, Rotational and Pose Invariance:* The output skeleton is invariant under isometric transformation, as it is not dependent on the object orientation, a user-specified root point, or a height function. Contraction conforms to nearby features through the merge operation without requiring user interaction. The contraction process in Equation (5.4.4) is also invariant under global rotations (rotational invariance), and locally invariant under local rotations, and therefore pose-invariant, as shown by the identical topology of the two skeleton poses in Figure 42e and Figure 42f.

*Mesh Prerequisites:* Skeletonization methods can introduce input restrictions such as requiring genus zero topology, or a properly connected, manifold closed mesh. The proposed method introduces no such restrictions beyond a simple preprocess to satisfy the domain of the merge operation. The contraction process uses nearby vertex positions and one-ring areas, making it extremely robust and capable of handling severely damaged models (Figure 42b), with holes (Figure 42b & d), high genus (Figure 42c & Figure 47a→Bike), nearby parallel regions (Figure 47a→Bike), surface noise (Figure 42d), and even consisting of multiple separate disconnected parts (Figure 41 & Figure 47a→Mech).

*Resolution Invariance and Parallelization:* The proposed method operates at varying resolution according to the target feature size  $\omega$ . It is not limited to voxelized inputs, with low-resolution artifacts as seen in thinning algorithms (Figure 47e). Also the method is suitable for parallelism, given the small dependencies of the domains in Equations (5.4.1) and (5.4.2).

### Figure 47: Comparing methods for common failure cases

The models in this figure have been carefully selected to be difficult to skeletonize. The bike has a low triangle count with nearby parallel regions (the wheels). The mech is also difficult to skeletonize as it consists of multiple disconnected components. Similarly, the horse contains disconnected and self-intersecting components.



(a) Feature-Varying Skeletonization  
Feature Size  $\omega = 0.002$   
(The Proposed Method)

(b) Geometric Contraction  
(Au, Tai, Chu, Cohen-Or, & Lee, 2008)

(c) Sampled Geometric Contraction  
(Cao, Tagliasacchi, Olson, Zhang, & Su, 2010)

(d) Propagation from Critical Points  
(Ning, Li, Zhang, & Wang, 2010)

(e) Thinning from Voxelization  
(L. Liu, Chambers, Letscher, & Ju, 2010)

## Comparison

The focus of this comparison is with unassisted methods, which excludes height functions: harmonic and reeb-graphs, database-training, pose-learning, and root specification; however contraction and propagation strategies are discussed. In Figure 47 the method is compared using mechanical and organic models of a Bike, Mech, and Horse. These three models cover a large range of shape properties including: poor connectivity, depth-variation, topological loops, non-manifold parts, nearby disconnected vertices, sharp angular features, flat regions, sparseness, and dense geometric detail.

*Contraction:* Figure 47b shows a popular contraction method (Au et al., 2008) which produces homotopic skeletons. Their method is highly sensitive to the connectivity (Figure 47b middle), and vertex distribution (Figure 47b top), resulting in poor abstractions for many models (Figure 47b bottom). The work by (Cao et al., 2010) improves the result by introducing remeshing as a preprocess (Figure 47c), however this introduces error with unpredictable feature abstraction (Figure 47c middle). Figure 47e shows a thinning method (L. Liu et al., 2010) using the *Solid Voxelization* application as input. Thinning can be performed in parallel and produces good curve-skeletons on the Bike (Figure 47e), but requires post-processing to segment the output into bones (Brunner & Brunnett, 2005) and the output from voxelization is often sensitive to the surface boundary (Figure 47e middle & bottom).

*Propagation:* A state of the art propagation method based on critical points was implemented (Ning et al., 2010) but, their result was improved (Figure 47d) by ensuring proper connectivity with uniform sampling, which is similarly achieved by (Hassouna & Farag, 2009) requiring solid-voxelization as input. The methods require manually setting multiple parameters to control the surface or volume samples, and to control the conditions for connecting the propagating fronts in topological loops. Also (Ning et al., 2010) is limited in high genus inputs (Figure 47d top).

In summary, the proposed method is classified as a contraction method, however in contrast to previous contraction and propagation methods, it has a simple implementation which only requires two parameters:  $\omega$  and a constant  $\delta$ , to produce efficient thin skeletons with meaningful topology. Skeletons can be produced for high genus models without requiring proper connectivity or a manifold closed mesh. Furthermore, the results predictably retain features of size  $\omega$ , to produce useful variation in topology, centeredness and control over the level of feature abstraction.

## 5.8 Chapter Summary

This chapter has presented a unique skeletonization method that creates meaningful topology and feature abstraction according to a target feature size  $\omega$ . Not only does  $\omega$  control the efficiency of the algorithm, where large values allow for interactive skeletonization on huge inputs, but also it gives control over the degree of homotopy and centeredness of the output.

### Contributions

Feature-varying skeletonization addresses the problem of controlling the target feature size in automatically generated skeletons. This is important for real-time animation, which only needs to simulate deformation for the main model features, whose size is greater than  $\omega$ . In the case of volume images, sparse voxel octrees, or uniformly sampled models, the value for  $\omega$  can be suggested automatically by the sample spacing. Therefore, in such cases, the method can run without requiring any input parameters. In the case of polygon-meshes, the algorithm can operate on severely damaged, disconnected models, which sets it apart from the majority of the existing literature. The algorithm is also highly insensitive to noise, as features smaller than  $\omega$  are smoothed, whereas features larger than  $\omega$  are retained by a novel merge operation. The contraction process is controlled by simple and efficient interpolation which guarantees the resulting skeleton to be zero-volume, where vertices are automatically separated into groups that represent the output bones. These maintain correspondence with original model for many applications, including skinning and segmentation.

### Limitations

The merge operation ensures that vertex groups are spaced  $> \omega$ , but in rare cases the vertices may progressively merge with denser regions, resulting in sparser bone positioning where fine features join with larger features. Ideally every bone should be spaced with a distance of  $\omega$  for robustness and regularity, which could be achieved by introducing a length constraint into the iterative contraction process. Also, the termination condition can be improved by being local, instead of global (for all vertices). This would prevent processing vertices that have already formed parts of the skeleton, and require no further contraction or processing. Furthermore, the method is neither homotopic nor centered (Figure 41b & Figure 45a) because the skeleton features are specified according to the target size  $\omega$ . Future approaches could analyze the input in order to determine different values of  $\omega$  for each input vertex. The aim would be to generate a single centered, unique, and homotopic result. However such analysis is challenging because of surface noise, which may require at least one parameter. Also this strategy will not always generate useful results in cases such as Figure 41a.

# Chapter 6

## Conclusion

### 6.1 Summary

This thesis has conducted research in the area of deforming volumetric shapes in real-time. Efficient and high-quality solutions have been found for the problems of selecting, deforming, resampling, reconstructing hierarchy, and rendering dynamic volumetric content. In addition, an automatic skeletonization algorithm has been presented, which generates feature-varying and mapped control skeletons for intuitively and efficiently manipulating the underlying shape regions. This thesis therefore takes a step forward to the goal of interactively modeling shapes as observed in the real world, which are volumetric, sparse, and dynamic. This enables new applications in real-time computer animation and in realistic large-scale simulations.

In the existing literature, massive real-time animated scenes are often modeled with multiple polygon meshes, of which different geometric detail representations are switched in memory for each shape, which are further tessellated and displaced on graphics hardware according to the region of interest. In order to apply deformation efficiently in parallel, the nearby meshes are selected, and then their vertices are transformed using the vertex shading capabilities of graphics hardware. This is then followed by discarding unimportant shape regions by clipping them according to the viewing frustum: this approach therefore processes many undesirable regions of the input shapes, such as those occluded by other parts of the model. Furthermore, the tessellated surface details do not support overhanging ledges and the irregular distribution of the mesh geometry often leads to intersections and undesirable tearing, as seen in Figure 34 (lower). In contrast, the proposed pipeline only applies deformation to contributing shape regions within a parallelized selection process (Figure 13) that operates on purely volumetric shapes. This means that calculations are only applied to the important shape regions, which are partitioned and simulated hierarchically, giving smooth transitions between geometric resolutions according to the projection and occlusion criteria (Table 1). By tuning these criteria according to the capabilities of the target machine, the resolution of the displayed shape regions can be controlled, which fulfills the objective of balancing the simulation accuracy and efficiency, highlighted in Chapter 1. Furthermore the input volumetric shapes support complex internal features and overhanging ledges, such as the tree in Figure 1, without the limitations and intricacies encountered when modeling with parametric surfaces.



The volumetric shapes are automatically mapped to a generated skeleton structure that allows for easy and intuitive manipulation of the underlying shape regions, and allows for expensive constraints to be enforced in real-time on the simplified structure geometry. In the literature, skeletonization methods strive to be centered and homotopic; however it is observed that the generated skeletons may not capture a meaningful approximation of the shape features for complex topologies or damaged inputs. Also, unintuitive parameters are often introduced to address issues such as surface noise or how topological parts are connected, which hinders the modeling process. In contrast, a feature-varying strategy is introduced that unites these issues under a single parameter, which gives control over the output topology and the target size of the captured features. This means that skeletons can be generated robustly and efficiently according to the feature size parameter, allowing for usable skeletons to be generated for complex, damaged, or noisy models. Therefore the objective of usability has been addressed, as the labor-intensive workflow of creating and mapping control skeleton structures has been streamlined. Furthermore, the volumetric shape regions can be updated efficiently by their mapped skeleton geometry, which helps enable real-time animation in massive scenes.

## 6.2 Limitations

This research has not investigated filtering the deformed volumetric shapes, and therefore the generated images appear undesirably thick, soft and dull in comparison to other approaches (Figure 34). It is challenging to efficiently calculate pre-filtering in dynamic pipelines, as this is typically calculated in a bottom-up process, regardless of the resolution accurately required for rendering (Crassin, 2011). Also, it requires more memory consumption to store accurate pre-filtering subpixel information (Heitz & Neyret, 2012; S. Laine & Karras, 2011) and therefore it may be worth investigating other techniques to improve the appearance of the voxels, such as depth-of-field and postprocessing effects (S. Laine & Karras, 2011; Síleš, 2012).

In order to determine which voxels are visible during the selection process, a hierarchical z-buffer approach is used; however the limitations of this approach are that it does not consider visibility data from transparent objects, and that the visible regions are out-of-date by a single frame (Figure 40). The advantage of the hierarchical z-buffer is that the visibility information is already organized, and can therefore be sampled efficiently during the selection algorithm with a small number of cache-coherent texture fetch operations. Therefore, to support more sophisticated visibility feedback, more memory is needed to store the visibility of transparent objects, and also additional organization of the feedback data may be required to ensure that the information can be efficiently retrieved during the selection process. Novel visibility criteria are also particularly challenging to design, as they need to consider the temporality of dynamic shapes, which may be occluded by different parts of the model according to its animation.

## 6.3 Contributions

This thesis contributes to the field of real-time computer animation by providing efficient and high-quality solutions to the problems of selecting, deforming, resampling, and manipulating massive amounts of volumetric shapes in sparse scenes. These problems are important as the shapes observed in the real-world are volumetric, and many modern applications require more accurate simulation with a robust closed-manifold surface definition without self-intersections, and with support for complex solid shapes. The selection and deformation problems are two fundamental challenges faced with animated volumetric shapes. This is because the volumetric shapes store huge amounts of geometric information, which requires careful organization to accelerate shape queries for real-time applications; however animation may destroy the pre-calculated data organization. Instead, the proposed pipeline (Figure 13) makes use of *hierarchy* applied to *instanced* input shapes, and then simulates deformation inside a parallel selection process without wasting deformation calculations on unimportant parts of the scene. In any applications where further content organization is needed, hierarchy is efficiently constructed in parallel for the much smaller set of contributing regions at the required resolution.

Deforming the volume elements of input shapes causes their quadrangular faces to become nonplanar, which is extremely difficult and expensive to test for intersection. This problem is addressed with efficient resampling strategies that generate samples for the deformed voxels at the grid locations in a sparse voxel octree structure. The corresponding voxels can therefore be set at the new sample locations when the SVO hierarchy is constructed, which is designed to prevent any gap or distortion artifacts from occurring (Figure 4). It is observed that adding more samples is more accurate, but less efficient, and therefore two solutions are presented to target different applications: (1) by generating a fixed number of samples for consistent *efficiency*, and (2) by generating a varying number of samples for consistent *accuracy*. These solutions are measured by comparing them with the unaligned input, which is then developed into a practical Decision Tree. After this resampling process, the SVO hierarchy is constructed in parallel for the contributing regions, as discussed in the previous paragraph, which means that the deformed shapes can be efficiently tested for intersection in real-time applications.

It is challenging to create appealing animation for massive scenes with lots of detailed shapes, as complex and realistic deformations can require constraints that are otherwise too expensive to calculate for all the selected shape regions in real-time. In the literature, this problem is addressed by using a control structure abstraction, which acts on behalf of the shape features, and can be manipulated more efficiently and intuitively. Therefore, expensive constraints, such as kinematic linkages or physical rules, can be applied directly to the simplified structure, and the shape regions can be updated efficiently by a mapping to the structure geometry. This

means that deformation can be modeled with an efficient interpolation process, which is more suitable for real-time animation. However, creating and mapping these structures to the input shapes is a problematic and laborious task: cage-based structures are simple to generate, but they fail to capture high-frequency features unless the cage resolution is increased, which impacts the efficiency. In contrast, accurate skeleton structures are good at capturing shape features, but they are difficult to define as the shape centerline is highly sensitive to the shape topology, and also to noise found on the shape boundary. This problem is addressed with an automatic feature-varying skeletonization algorithm (Willcocks & Li, 2012) that contributes to the field of geometric modeling by generating meaningful skeleton abstractions for complex, damaged, or noisy inputs. This is achieved by using a single tunable parameter, which controls the output topology and the target size of the captured features, allowing users to rapidly set the desired approximation of the shape features in order to generate efficient animations.

## 6.4 Applications

This research has focused on the real-time animation of large amounts of volumetric shapes, which is an important step towards the goal of simulating shapes as they occur in nature. The results demonstrate the potential of volumetric shapes in the real-time graphics industry: in particular in large-scale virtual simulation, such as environmental, flight, medical, military, or crowd simulations. In such cases, simulating accurate animation for large amounts of objects is important; however, achieving this efficiently in parallel often requires artist-specified levels of simulation, such as manually refined level of detail meshes or imposter geometry. In contrast, the proposed method partitions and simulates shape regions hierarchically, giving smooth transitions between the simulation levels according to the projection and occlusion criteria. This approach is less noticeable allowing for more immersion within the virtual environment.

Volumetric shapes are also an interesting proposal for the computer games industry. However the popular development workflows focus on polygon-based production pipelines and tool chains. Recently, volumetric shapes are being used in parts of these pipelines, such as in volumetric sculpting and terrain modeling, whereby volumetric shape information can either be baked to displacement textures and tessellated using graphics hardware, or converted to polygons. While this work proposes a purely volumetric workflow, there still needs to be a large amount of effort to address issues such as filtering and designing production tools from the ground up to use volumetric primitives in physical effects encountered during video games, which includes simulating collisions, fluids, cloth, and various natural phenomena.

In addition to virtual simulations and video games, this research has applications in geometric modeling. The proposed feature-varying skeletonization algorithm addresses the challenge of generating meaningful control skeleton structures, which describe important features of the

underlying shape. This therefore enables artists to rapidly generate a skeleton structure for their input shapes, by simply tuning a single parameter to capture the desired size of feature abstraction, which improves productivity in comparison to the manual process of specifying and mapping bones to the input shape. The generated feature-varying skeletons can either be used directly in real-time animation, or they can act as a descriptor for features and properties of the input shape. Additionally, the skeletons can be further fine-tuned, either manually or as an assisted process (Figure 46), according to the specific requirements of the artist. This would therefore be a useful addition to increase usability in existing modeling tools.

## 6.5 Future Work

The objective of this thesis is to enable real-time support for directly manipulating, deforming, and rendering massive amounts of volumetric content in sparse scenes. In the future, it would be worth comparing different types of volumetric storage formats, filtering representations, and postprocessing effects that improve the appearance of volumetric shapes. In particular, it would be beneficial to understand how volumetric shapes can be encoded with small memory consumption while achieving realistic shape representation, and also to understand which types of filtering approach can be supported with dynamic pipelines such as in this thesis. This would allow volumetric shapes to be used more widely in industries that require more realistic real-time shape representations and will lead to more modeling and rendering applications.

It would also be worth extending the sparse volumetric deformation approach to handle more sophisticated visibility feedback information from the renderer, including transparent objects and lighting information from different views in the scene. This would allow for more accurate images to be generated with indirect lighting, and for more efficient images to be generated with transparent shapes; currently all of the transparent shape regions need to be simulated, which is not scalable in massive scenes.

Finally, in the future, it would be exciting to develop volumetric production pipelines and tool chains which are designed from the beginning to target large-scale volumetric animation and rendering. The current real-time 3D graphics industry is heavily influenced by polygon-specific workflows that only use volumetric shapes in a small part of the modeling process. These shapes are often converted back into polygon meshes for rendering, which greatly hinders productivity and adds undesirable complexity for developers and artists. It would therefore be beneficial to develop and compare new volumetric tool chains that consider using volumetric shapes as the basic primitive for simulating collisions, fluids, cloth, wrinkles, fire, clouds, rain, and various other effects and natural phenomena observed in the real-world.

## References

- Aila, T., & Laine, S. (2009). *Understanding the efficiency of ray traversal on GPUs*. Paper presented at the Proceedings of the Conference on High Performance Graphics 2009, New Orleans, Louisiana.
- Akenine-Möller, T., Haines, E., & Hoffman, N. (2008). *Real-Time Rendering* (Third Edition ed.): A. K. Peters, Ltd.
- Alexa, M., Cohen-Or, D., & Levin, D. (2000). *As-rigid-as-possible shape interpolation*. Paper presented at the Proceedings of the 27th annual conference on Computer graphics and interactive techniques.
- Amanatides, J., & Woo, A. (1987). A fast voxel traversal algorithm for ray tracing. *Eurographics*, 3-10.
- Arcelli, C., di Baja, G. S., & Serino, L. (2011). Distance-Driven Skeletonization in Voxel Images. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(4), 709-720. doi: 10.1109/tpami.2010.140
- Aronov, B., Herv, Bronnimann, Chang, A. Y., & Chiang, Y.-J. (2003). *Cost-driven octree construction schemes: an experimental study*. Paper presented at the Proceedings of the nineteenth annual symposium on Computational geometry, San Diego, California, USA.
- Arvo, J. (1990). Transforming axis-aligned bounding boxes. In S. G. Andrew (Ed.), *Graphics gems* (pp. 548-550): Academic Press Professional, Inc.
- Ashraf, G., & Zhou, J. (2006). *Hardware accelerated skin deformation for animated crowds*. Paper presented at the Proceedings of the 13th International conference on Multimedia Modeling - Volume Part II, Singapore.
- Au, O. K.-C., Tai, C.-L., Chu, H.-K., Cohen-Or, D., & Lee, T.-Y. (2008). Skeleton extraction by mesh contraction. *ACM Trans. Graph.*, 27(3), 1-10. doi: 10.1145/1360612.1360643
- Aujay, G., Hetroy, F., Lazarus, F., & Depraz, C. (2007). *Harmonic skeleton for realistic character animation*. Paper presented at the Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation, San Diego, California.
- Baraff, D. (1992). *Rigid body simulation*. Paper presented at the SIGGRAPH 95 Course Note 34. ACM SIGGRAPH.
- Baran, I., & Popovic, J. (2007). Automatic rigging and animation of 3D characters *SIGGRAPH '07: ACM SIGGRAPH 2007 papers* (pp. 72). San Diego, California: ACM.
- Bautembach, D. (2011). *Animated Sparse Voxel Octrees*. University of Hamburg, Hamburg.
- Bayer, R., & McCreight, E. (2002). Organization and maintenance of large ordered indexes. In B. Manfred & D. Ernst (Eds.), *Software pioneers* (pp. 245-262): Springer-Verlag New York, Inc.

- Benson, D., & Davis, J. (2002). Octree textures. *ACM Trans. Graph.*, 21(3), 785-790. doi: 10.1145/566654.566652
- Berendsen, H. J. C. (2007). *Simulating the Physical World: Hierarchical Modeling from Quantum Mechanics to Fluid Dynamics*: Cambridge University Press.
- Biasotti, S., Attali, D., Boissonnat, J.-D., Edelsbrunner, H., Elber, G., Mortara, M., . . . Veltkamp, R. (2008). Skeletal Structures, *Shape Analysis and Structuring*, pp. 145-183.
- Biasotti, S., Giorgi, D., Spagnuolo, M., & Falcidieno, B. (2008). Reeb graphs for shape analysis and applications. *Theoretical Computer Science*, 392(1-3), 5-22.
- Biasotti, S., Marini, S., Mortara, M., & Patan, G. (2003). *An overview on properties and efficacy of topological skeletons in Shape Modelling*. Paper presented at the Proceedings of the Shape Modeling International 2003. <http://dl.acm.org/citation.cfm?id=829510.830292>
- Billeter, M., Olsson, O., & Assarsson, U. (2009). *Efficient stream compaction on wide SIMD many-core architectures*. Paper presented at the Proceedings of the Conference on High Performance Graphics 2009, New Orleans, Louisiana.
- Blythe, D. (2006). *The Direct3D 10 system*. Paper presented at the ACM SIGGRAPH 2006 Papers, Boston, Massachusetts.
- Boissonnat, J.-D., & Oudot, S. (2005). Provably good sampling and meshing of surfaces. *Graphical Models*, 67(5), 405 - 451.
- Brunner, D., & Brunnett, G. (2005). An extended concept of voxel neighborhoods for correct thinning in mesh segmentation *Proceedings of the 21st spring conference on Computer graphics - SCCG '05* (pp. 119). Budmerice, Slovakia.
- Burtnyk, N., & Wein, M. (1971). Computer-Generated Key-Frame Animation. *Journal of the SMPTE*, 80(3), 149-153. doi: 10.5594/j07698
- Burtnyk, N., & Wein, M. (1976). Interactive skeleton techniques for enhancing motion dynamics in key frame animation. *Commun. ACM*, 19(10), 564-569. doi: 10.1145/360349.360357
- Cao, J., Tagliasacchi, A., Olson, M., Zhang, H., & Su, Z. (2010). Point Cloud Skeletons via Laplacian Based Contraction *Proceedings of the 2010 Shape Modeling International Conference* (pp. 187-197). Washington, DC, USA: IEEE Computer Society.
- Carmona, R., & Froehlich, B. (2011). Error-controlled real-time cut updates for multi-resolution volume rendering. *Computers & Graphics-Uk*, 35(4), 931-944. doi: DOI 10.1016/j.cag.2011.01.007
- Castro, R., Lewiner, T., Lopes, H., Tavares, G., & Bordignon, A. (2008). Statistical optimization of octree searches. *Computer Graphics Forum*, 27(6), 1557-1566. doi: 10.1111/j.1467-8659.2007.01104.x
- Cazals, F., Drettakis, G., & Puech, C. (1995). *Filtering, Clustering and Hierarchy Construction: a New Solution for Ray-Tracing Complex Scenes*. Paper presented at the Computer Graphics Forum.
- Chandrasekhar, S. (1960). *Radiative Transfer*: Dover Publ.

- Chang, A. Y. (2001). A Survey of Geometric Data Structures for Ray Tracing (pp. 91). Brooklyn, New York: Department of Computer and Information Science Polytechnic University.
- Chen, Y., Qing-Hong, Z., Kaufman, A., & Muraki, S. (1998, 8-10 Jun 1998). *Physically-based animation of volumetric objects*. Paper presented at the Computer Animation 98. Proceedings.
- Chen, Z., & Chou, H.-L. (2006). *New Efficient Octree Construction from Multiple Object Silhouettes with Construction Quality Control*. Paper presented at the Proceedings of the 18th International Conference on Pattern Recognition - Volume 01.
- Chilimbi, T. M. (1999). *Cache-conscious data structures: design and implementation*: University of Wisconsin--Madison.
- Choi, K.-J., & Ko, H.-S. (2002). Stable but responsive cloth. *ACM Trans. Graph.*, 21(3), 604-611. doi: 10.1145/566654.566624
- Christensen, P. H. (2008). Point-Based Approximate Color Bleeding: Pixar Animation Studios.
- Cohen, J. D., Lin, M. C., Manocha, D., & Ponamgi, M. (1995). *I-COLLIDE: an interactive and exact collision detection system for large-scale environments*. Paper presented at the Proceedings of the 1995 symposium on Interactive 3D graphics, Monterey, California, USA.
- Coorg, S., & Teller, S. (1996). *Temporally coherent conservative visibility (extended abstract)*. Paper presented at the Proceedings of the twelfth annual symposium on Computational geometry, Philadelphia, Pennsylvania, United States.
- Cordier, F., & Magnenat-Thalmann, N. (2002). Real-time Animation of Dressed Virtual Humans. *Computer Graphics Forum*, 21(3), 327-335. doi: 10.1111/1467-8659.t01-1-00592
- Cornea, N. D., Silver, D., & Min, P. (2007). Curve-Skeleton Properties, Applications, and Algorithms. *IEEE Transactions on Visualization and Computer Graphics*, 13(3), 530-548.
- Cornea, N. D., Silver, D., Yuan, X., & Balasubramanian, R. (2005). Computing hierarchical curve-skeletons of 3D objects. *The Visual Computer*, 21(11), 945-955.
- Cozzi, P., & Riccio, C. (2012). *OpenGL Insights*: CRC Press.
- Craig, J. J. (2008). *Introduction To Robotics: Mechanics And Control* (Third Edition ed.): Pearson Education.
- Crassin, C. (2011). *GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes*. Universite de Grenoble. Retrieved from <http://maverick.inria.fr/Publications/2011/Cra11>
- Crassin, C., Neyret, F., Lefebvre, S., & Eisemann, E. (2009). *GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering*. Paper presented at the Proceedings of the 2009 symposium on Interactive 3D graphics and games, Boston, Massachusetts.
- Crassin, C., Neyret, F., Sainz, M., Green, S., & Eisemann, E. (2011). Interactive Indirect Illumination Using Voxel Cone Tracing. *Computer Graphics Forum*, 30(7), 1921-1930. doi: 10.1111/j.1467-8659.2011.02063.x

- Décoret, X., Durand, F., Sillion, F. X., & Dorsey, J. (2003). *Billboard clouds for extreme model simplification*. Paper presented at the ACM SIGGRAPH 2003 Papers, San Diego, California.
- Decoret, X., Sillion, F., Schaufler, G., & Dorsey, J. (2001). *Multi-layered impostors for accelerated rendering*. Paper presented at the Computer Graphics Forum.
- Desbrun, M., Meyer, M., Schroder, P., & Barr, A. H. (1999). Implicit fairing of irregular meshes using diffusion and curvature flow *Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (pp. 317-324). New York, NY, USA: ACM Press/Addison-Wesley Publishing Co.
- Dey, T. K., Ranjan, P., & Wang, Y. (2012). Eigen deformation of 3D models. *The Visual Computer*, 28(6-8), 585-595. doi: 10.1007/s00371-012-0705-0
- Dey, T. K., & Sun, J. (2006). Defining and computing curve-skeletons with medial geodesic function *Proceedings of the fourth Eurographics symposium on Geometry processing* (pp. 143-152). Cagliari, Sardinia, Italy: Eurographics Association.
- Dick, C., Schneider, J., & Westermann, R. (2009). Efficient Geometry Compression for GPU-based Decoding in Realtime Terrain Rendering. *Computer Graphics Forum*, 28(1), 67-83. doi: DOI 10.1111/j.1467-8659.2008.01298.x
- Dietz, H. G., & Young, B. D. (2010). *MIMD interpretation on a GPU*. Paper presented at the Proceedings of the 22nd international conference on Languages and Compilers for Parallel Computing, Newark, DE.
- Dobashi, Y., Kaneda, K., Yamashita, H., Okita, T., & Nishita, T. (2000). *A simple, efficient method for realistic animation of clouds*. Paper presented at the Proceedings of the 27th annual conference on Computer graphics and interactive techniques.
- Drepper, U. (2007). *What every programmer should know about memory*. Red Hat, Inc.
- Ebert, D. S. (2003). *Texturing and Modeling: A Procedural Approach*: Elsevier Science.
- Engel, K. (2006). *Real-Time Volume Graphics*: A K Peters, Ltd.
- Enright, D., Marschner, S., & Fedkiw, R. (2002). Animation and rendering of complex water surfaces. *ACM Trans. Graph.*, 21(3), 736-744. doi: 10.1145/566654.566645
- Ericson, C. (2005). *Real-Time Collision Detection* (Vol. 1): Morgan Kaufmann.
- Esturo, J. M., Rössl, C., & Theisel, H. (2010). *Continuous Deformations of Implicit Surfaces*. Paper presented at the Proceedings of the Vision, Modeling, and Visualization Workshop. <http://dblp.uni-trier.de/db/conf/vmv/vmv2010.html#EsturoRT10>
- Farin, G. E. (2002). *Curves and Surfaces for CAD: A Practical Guide*: Elsevier Science.
- Fatahalian, K., Boulos, S., Hegarty, J., Akeley, K., Mark, W. R., Moreton, H., & Hanrahan, P. (2010). *Reducing shading on GPUs using quad-fragment merging*. Paper presented at the ACM SIGGRAPH 2010 papers, Los Angeles, California.
- Fatahalian, K., Luong, E., Boulos, S., Akeley, K., Mark, W. R., & Hanrahan, P. (2009). *Data-parallel rasterization of micropolygons with defocus and motion blur*. Paper presented



at the Proceedings of the Conference on High Performance Graphics 2009, New Orleans, Louisiana.

- Fedkiw, R., Stam, J., & Jensen, H. W. (2001). *Visual simulation of smoke*. Paper presented at the Proceedings of the 28th annual conference on Computer graphics and interactive techniques.
- Flynn, M. (1972). Some Computer Organizations and Their Effectiveness. *Computers, IEEE Transactions on, C-21*(9), 948-960. doi: 10.1109/tc.1972.5009071
- Foley, T., & Sugerman, J. (2005). *KD-tree acceleration structures for a GPU raytracer*. Paper presented at the Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Los Angeles, California.
- Frank, S., & Kaufman, A. (2009). Out-of-Core and Dynamic Programming for Data Distribution on a Volume Visualization Cluster. *Computer Graphics Forum, 28*(1), 141-153. doi: 10.1111/j.1467-8659.2008.01307.x
- Fujimoto, A., Tanaka, T., & Iwata, K. (1988). ARTS: accelerated ray-tracing system *Tutorial: computer graphics; image synthesis* (pp. 148-159): Computer Science Press, Inc.
- Galyean, T. A., & Hughes, J. F. (1991). Sculpting: an interactive volumetric modeling technique. *SIGGRAPH Comput. Graph., 25*(4), 267-274. doi: 10.1145/127719.122747
- Garanzha, K., Pantaleoni, J., & McAllister, D. (2011). *Simpler and faster HLBVH with work queues*. Paper presented at the Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics, Vancouver, British Columbia, Canada.
- Garg, K., & Nayar, S. K. (2006). *Photorealistic rendering of rain streaks*. Paper presented at the ACM SIGGRAPH 2006 Papers, Boston, Massachusetts.
- Gargantini, I. (1982). Linear octrees for fast processing of three-dimensional objects. *Computer Graphics and Image Processing, 20*(4), 365 - 374. doi: 10.1016/0146-664X(82)90058-2
- Gaster, B., Kaeli, D. R., Howes, L., Mistry, P., & Schaa, D. (2011). *Heterogeneous Computing With OpenCL*: Elsevier Science.
- Georgii, J., & Westermann, R. (2005). Mass-spring systems on the GPU. *Simulation Modelling Practice and Theory, 13*(8), 693-702. doi: <http://dx.doi.org/10.1016/j.simpat.2005.08.004>
- Giegl, M., & Wimmer, M. (2007). Unpopping: Solving the Image-Space Blend Problem for Smooth Discrete LOD Transitions. *Computer Graphics Forum, 26*(1), 46-49.
- Gómez, J. E. (1985). TWIXT: A 3D animation system. *Computers & Graphics, 9*(3), 291-298. doi: [http://dx.doi.org/10.1016/0097-8493\(85\)90056-1](http://dx.doi.org/10.1016/0097-8493(85)90056-1)
- Goral, C. M., Torrance, K. E., Greenberg, D. P., & Battaile, B. (1984). Modeling the interaction of light between diffuse surfaces. *SIGGRAPH Comput. Graph., 18*(3), 213-222. doi: 10.1145/964965.808601
- Gross, M., & Pfister, H. (2007). *Point-Based Graphics*: Morgan Kaufmann.
- Guennebaud, G., Barthe, L., & Paulin, M. (2004). Deferred Splatting. *Computer Graphics Forum, 23*(3), 653-660. doi: 10.1111/j.1467-8659.2004.00797.x

- Hachisuka, T., & Jensen, H. W. (2010). *Parallel progressive photon mapping on GPUs*. Paper presented at the ACM SIGGRAPH ASIA 2010 Sketches, Seoul, Republic of Korea.
- Hadap, S., Bangerter, E., Volino, P., & Magnenat-Thalmann, N. (1999). *Animating wrinkles on clothes*. Paper presented at the Proceedings of the conference on Visualization '99: celebrating ten years, San Francisco, California, USA.
- Harris, M. (2005). *Mapping computational concepts to GPUs*. Paper presented at the ACM SIGGRAPH 2005 Courses, Los Angeles, California.
- Hasler, N., Thormählen, T., Rosenhahn, B., & Seidel, H.-P. (2010). Learning skeletons for shape and pose *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games - I3D '10* (pp. 23). Washington, D.C.
- Hassouna, M. S., & Farag, A. A. (2009). Variational Curve Skeletons Using Gradient Vector Flow. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 31(12), 2257-2274.
- Hatcher, P. J., & Quinn, M. J. (1991). *Data-Parallel Programming on Mimd Computers*: Mit Press.
- Havran, V. (2000). *Heuristic Ray Shooting Algorithms*. Doctoral Thesis, Czech Technical University, Prague. Retrieved from <http://www.cgg.cvut.cz/~havran/phdthesis.html>
- He, Y., Xiao, X., & Seah, H.-S. (2009). Harmonic 1-form based skeleton extraction from examples. *Graphical Models*, 71(2), 49-62.
- Heitz, E., & Neyret, F. (2012). *Representing Appearance and Pre-filtering Subpixel Data in Sparse Voxel Octrees*. Paper presented at the High Performance Graphics.
- Ho, C.-C., Wu, F.-C., Chen, B.-Y., Chuang, Y.-Y., & Ouhyoung, M. (2005). Cubical Marching Squares: Adaptive Feature Preserving Surface Extraction from Volume Data. *Computer Graphics Forum, (Proceedings Eurographics 2005)*, 24(3), 537-545.
- Hollander, M., Ritschel, T., Eisemann, E., & Boubekur, T. (2011). ManyLoDs: Parallel Many-View Level-of-Detail Selection for Real-Time Global Illumination. *Computer Graphics Forum*, 30(4), 1233-1240. doi: 10.1111/j.1467-8659.2011.01982.x
- Jensen, H. W. (1996). *Global illumination using photon maps*. Paper presented at the Proceedings of the eurographics workshop on Rendering techniques, Porto, Portugal.
- Jevans, D., & Wyvill, B. (1988). *Adaptive voxel subdivision for ray tracing*. Retrieved from <http://hdl.handle.net/1880/18015>
- Jia, W., Shaw, K. A., & Martonosi, M. (2012). *Characterizing and improving the use of demand-fetched caches in GPUs*. Paper presented at the Proceedings of the 26th ACM international conference on Supercomputing, San Servolo Island, Venice, Italy.
- Jimenez, J., Echevarria, J. I., Sousa, T., & Gutierrez, D. (2012). SMAA: Enhanced Morphological Antialiasing. *Computer Graphics Forum (Proc. EUROGRAPHICS 2012)*, 31(2).
- Jimenez, J., Gutierrez, D., Yang, J., Reshetov, A., Demoreuille, P., Berghoff, T., . . . Sousa, T. (2011). Filtering Approaches for Real-Time Anti-Aliasing *ACM SIGGRAPH Courses*.
- Ju, T., Losasso, F., Schaefer, S., & Warren, J. (2002). Dual contouring of hermite data. *ACM Trans. Graph.*, 21(3), 339-346. doi: 10.1145/566654.566586

- Kajiya, J. T. (1986). The rendering equation. *SIGGRAPH Comput. Graph.*, 20(4), 143-150. doi: 10.1145/15886.15902
- Kaplanyan, A., & Dachsbacher, C. (2010). *Cascaded light propagation volumes for real-time indirect illumination*. Paper presented at the Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games, Washington, D.C.
- Karras, T. (2012). Maximizing Parallelism in the Construction of BVHs, Octrees, and k-d Trees. *High Performance Graphics 2012*.
- Kavan, L., Collins, S., & O'Sullivan, C. (2009). *Automatic linearization of nonlinear skinning*. Paper presented at the Proceedings of the 2009 symposium on Interactive 3D graphics and games, Boston, Massachusetts.
- Kay, T. L., & Kajiya, J. T. (1986). Ray tracing complex scenes. *SIGGRAPH Comput. Graph.*, 20(4), 269-278. doi: 10.1145/15886.15916
- Keller, A. (1997). *Instant radiosity*. Paper presented at the Proceedings of the 24th annual conference on Computer graphics and interactive techniques.
- Kennedy, J. (2006). Swarm Intelligence. In A. Zomaya (Ed.), *Handbook of Nature-Inspired and Innovative Computing* (pp. 187-219): Springer US.
- Klimaszewski, K. S., & Sederberg, T. W. (1997). Faster ray tracing using adaptive grids. *Computer Graphics and Applications, IEEE*, 17(1), 42-51. doi: 10.1109/38.576857
- Kobbelt, L., & Botsch, M. (2004). A survey of point-based techniques in computer graphics. *Comput. Graph.*, 28(6), 801-814. doi: 10.1016/j.cag.2004.08.009
- Kontkanen, J., Tabellion, E., & Overbeck, R. S. (2011). Coherent Out-of-Core Point-Based Global Illumination. *Computer Graphics Forum*, 30(4), 1353-1360. doi: 10.1111/j.1467-8659.2011.01995.x
- Koo, Y.-M., & Shin, B.-S. (2005). *An efficient point rendering using octree and texture lookup*. Paper presented at the Proceedings of the 2005 international conference on Computational Science and Its Applications - Volume Part III, Singapore.
- Lacoste, J., Boubekeur, T., Jobard, B., & Schlick, C. (2007). *Appearance preserving octree-textures*. Paper presented at the Proceedings of the 5th international conference on Computer graphics and interactive techniques in Australia and Southeast Asia, Perth, Australia.
- Lacoste, J., Perin, C., & Jobard, B. (2008). *Progressive transmission of appearance preserving octree-textures*. Paper presented at the Proceedings of the 13th international symposium on 3D web technology, Los Angeles, California.
- Lafortune, E. P., & Willems, Y. D. (1993). *Bi-directional path tracing*. Paper presented at the Proceedings of CompuGraphics.
- Laine, S. (2010). *Restart trail for stackless BVH traversal*. Paper presented at the Proceedings of the Conference on High Performance Graphics, Saarbrücken, Germany.
- Laine, S., & Karras, T. (2011). Efficient Sparse Voxel Octrees. *IEEE Transactions on Visualization and Computer Graphics*, 17(8), 1048-1059. doi: Doi 10.1109/Tvcg.2010.240

- Lauterbach, C., Garland, M., Sengupta, S., Luebke, D., & Manocha, D. (2009). Fast BVH Construction on GPUs. *Computer Graphics Forum*, 28(2), 375-384. doi: 10.1111/j.1467-8659.2009.01377.x
- Lefebvre, S., & Hoppe, H. (2006). *Perfect spatial hashing*. Paper presented at the ACM SIGGRAPH 2006 Papers, Boston, Massachusetts.
- Lefebvre, S., Hornus, S., & Neyret, F. (2005). Octree Textures on the GPU. In M. Pharr (Ed.), *GPU Gems 2* (pp. 595-613): Addison-Wesley.
- Lei, Z., & Xu, D. Q. (2009). Real-Time Rendering of Highly Complex Dynamic Scenes Based on Parallel Multi-Core Architectures. *2009 International Conference on Information Management and Engineering, Proceedings*, 593-597. doi: Doi 10.1109/Icime.2009.126
- Lengyel, E. (2010). *Voxel-Based Terrain for Real-Time Virtual Simulations*. PhD Thesis, University of California, Davis, CA.
- Lengyel, E. (2011). *Mathematics for 3D Game Programming and Computer Graphics: Course Technology*.
- Lewiner, T., Mello, V., Peixoto, A., Pesco, S., & Lopes, H. (2010). Fast Generation of Pointerless Octree Duals. *Computer Graphics Forum*, 29(5), 1661-1669. doi: 10.1111/j.1467-8659.2010.01775.x
- Li, F. W. B. (2008). Parametric Surface Rendering *Wiley Encyclopedia of Computer Science and Engineering*: John Wiley & Sons, Inc.
- Li, F. W. B., Lau, R. W. H., Kilis, D., & Li, L. W. F. (2011). Game-on-demand:: An online game engine based on geometry streaming. *ACM Trans. Multimedia Comput. Commun. Appl.*, 7(3), 1-22. doi: 10.1145/2000486.2000493
- Li, Y., Wang, T., & Shum, H.-Y. (2002). Motion texture: a two-level statistical model for character motion synthesis. *ACM Trans. Graph.*, 21(3), 465-472. doi: 10.1145/566654.566604
- Lindholm, E., Kligard, M. J., & Moreton, H. (2001). A user-programmable vertex engine *Proceedings of the 28th annual conference on Computer graphics and interactive techniques - SIGGRAPH '01* (pp. 149-158). Not Known.
- Lipman, Y., Levin, D., & Cohen-Or, D. (2008). *Green Coordinates*. Paper presented at the ACM SIGGRAPH 2008 papers, Los Angeles, California.
- Liu, C. K., Hertzmann, A., & Popović, Z. (2005). *Learning physics-based motion style with nonlinear inverse optimization*. Paper presented at the ACM SIGGRAPH 2005 Papers, Los Angeles, California.
- Liu, L., Chambers, E. W., Letscher, D., & Ju, T. (2010). A simple and robust thinning algorithm on cell complexes. *Computer Graphics Forum*, 29(7), 2253-2260.
- Loop, C., & Schaefer, S. (2008). Approximating Catmull-Clark subdivision surfaces with bicubic patches. *ACM Trans. Graph.*, 27(1), 1-11. doi: 10.1145/1330511.1330519
- Lorensen, W. E., & Cline, H. E. (1987). Marching cubes: A high resolution 3D surface construction algorithm. *SIGGRAPH Comput. Graph.*, 21, 163-169.

- Lottes, T. (2009). NVIDIA FXAA (Fast Approximate Anti-Aliasing).
- Luebke, D. P. (2001). A Developer's Survey of Polygonal Simplification Algorithms. *IEEE Comput. Graph. Appl.*, 21(3), 24-35. doi: 10.1109/38.920624
- Maletz, D., & Wang, R. (2011). *Importance point projection for GPU-based final gathering*. Paper presented at the Proceedings of the Twenty-second Eurographics conference on Rendering, Prague, Czech Republic.
- Manson, J., & Schaefer, S. (2010). Isosurfaces Over Simplicial Partitions of Multiresolution Grids. *Computer Graphics Forum*, 29(2), 377-385. doi: 10.1111/j.1467-8659.2009.01607.x
- Marinov, M., & Kobbelt, L. (2005). Automatic Generation of Structure Preserving Multiresolution Models. *Computer Graphics Forum, (Proceedings Eurographics 2005)*, 24(3), 479-486.
- Marroquim, R., Kraus, M., & Cavalcanti, P. R. (2007). Efficient Point-Based Rendering Using Image Reconstruction *Proceedings Symposium on Point-Based Graphics* (pp. 101-108): Eurographics Association.
- McGuire, M., Hennessy, P., Bukowski, M., & Osman, B. (2012). *A Reconstruction Filter for Plausible Motion Blur*. Paper presented at the Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, Costa Mesa, California.
- Merrill, D. G., & Grimshaw, A. S. (2010). *Revisiting sorting for GPGPU stream architectures*. Paper presented at the Proceedings of the 19th international conference on Parallel architectures and compilation techniques, Vienna, Austria.
- Meyer, Q., Eisenacher, C., Stamminger, M., & Dachsbacher, C. (2009). *Data-parallel hierarchical link creation for radiosity*. Paper presented at the Proceedings of the 9th Eurographics conference on Parallel Graphics and Visualization, Munich, Germany.
- Mikkelsen, M. S. (2010). Bump Mapping Unparametrized Surfaces on the GPU. *Journal of Graphics, GPU, and Game Tools*, 15(1), 49-61. doi: 10.1080/2151237x.2010.10390651
- Miklos, B., Giesen, J., & Pauly, M. (2010). Discrete scale axis representations for 3D geometry *ACM SIGGRAPH 2010 papers* (pp. 101:101-101:110). Los Angeles, California: ACM.
- Monahan, J. F. (2011). *Numerical Methods of Statistics*: Cambridge University Press.
- Mortenson, M. E. (2006). *Geometric modeling*: Industrial Press.
- Mount, D. M., & Arya, S. (2010, January). ANN: A Library for Approximate Nearest Neighbor Searching
- Nguyen, D. Q., Fedkiw, R., & Jensen, H. W. (2002). Physically based modeling and animation of fire. *ACM Trans. Graph.*, 21(3), 721-728. doi: 10.1145/566654.566643
- Nickolls, J., & Dally, W. J. (2010). The GPU Computing Era. *Micro, IEEE*, 30(2), 56-69. doi: 10.1109/mm.2010.41
- Ning, X., Li, E., Zhang, X., & Wang, Y. (2010). Shape decomposition and understanding of point cloud objects based on perceptual information *Proceedings of the 9th ACM SIGGRAPH*

*Conference on Virtual-Reality Continuum and its Applications in Industry - VRCAI '10* (pp. 199). Seoul, South Korea.

- Nuzman, D., Rosen, I., & Zaks, A. (2006). *Auto-vectorization of interleaved data for SIMD*. Paper presented at the Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, Ottawa, Ontario, Canada.
- Oda, T., Itoh, Y., Nakai, W., Nomura, K., Kitamura, Y., & Kishino, F. (2006). Interactive Skeleton Extraction Using Geodesic Distance *16th International Conference on Artificial Reality and Telexistence-Workshops (ICAT'06)* (pp. 275-281). Hangzhou, Zhejiang, China.
- Olick, J. (2008). Current Generation Parallelism In Games Retrieved 25 July, 2012, from <http://www.jonolick.com/uploads/7/9/2/1/7921194/olick-current-and-next-generation-parallelism-in-games.pdf>
- Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A. E., & Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1), 80-113.
- Pajarola, R. (2003). Efficient Level-of-details for Point based Rendering *Computer Graphics and Imaging* (pp. 141-146): IASTED/ACTA Press.
- Pantaleoni, J., & Luebke, D. (2010). *HLBVH: hierarchical LBVH construction for real-time ray tracing of dynamic geometry*. Paper presented at the Proceedings of the Conference on High Performance Graphics, Saarbrücken, Germany.
- Pantuwong, N., & Sugimoto, M. (2010). Skeleton-growing: a vector-field-based 3D curve-skeleton extraction algorithm *ACM SIGGRAPH ASIA 2010 Sketches* (pp. 6:1-6:2). Seoul, Republic of Korea: ACM.
- Parent, R. (2012). *Computer Animation: Algorithms and Techniques* (Third Edition ed.): Elsevier Science.
- Parke, F. I., & Waters, K. (2008). *Computer Facial Animation* (Second Edition ed.): A K Peters, Limited.
- Paul, R. P. (1981). *Robot Manipulators: Mathematics, Programming, and Control: The Computer Control of Robot Manipulators*: Mit Press.
- Pauly, M., Keiser, R., Kobbelt, L. P., & Gross, M. (2003). *Shape modeling with point-sampled geometry*. Paper presented at the ACM SIGGRAPH 2003 Papers, San Diego, California.
- Piegl, L. A., & Tiller, W. (1997). *The Nurbs Book*: Springer-Verlag GmbH.
- Provot, X. (1996). *Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior* Paper presented at the Graphics Interface.
- Raman, R., & Wise, D. S. (2008). Converting to and from Dilated Integers. *Computers, IEEE Transactions on*, 57(4), 567-573. doi: 10.1109/tc.2007.70814
- Reeves, W. T., & Blau, R. (1985). Approximate and probabilistic algorithms for shading and rendering structured particle systems. *SIGGRAPH Comput. Graph.*, 19(3), 313-322. doi: 10.1145/325165.325250

- Reffye, P. d., Edelin, C., Françon, J., Jaeger, M., & Puech, C. (1988). Plant models faithful to botanical structure and development. *SIGGRAPH Comput. Graph.*, 22(4), 151-158. doi: 10.1145/378456.378505
- Ritschel, T., Dachsbacher, C., Grosch, T., & Kautz, J. (2012). The State of the Art in Interactive Global Illumination. *Computer Graphics Forum*, 31(1), 160-188. doi: 10.1111/j.1467-8659.2012.02093.x
- Ritschel, T., Eisemann, E., Ha, I., Kim, J. D., & Seidel, H.-P. (2011). *Making Imperfect Shadow Maps View-Adaptive: High-Quality Global Illumination in Large Dynamic Scenes*. Paper presented at the Computer Graphics Forum.
- Ritschel, T., Engelhardt, T., Grosch, T., Seidel, H.-P., Kautz, J., & Dachsbacher, C. (2009). Micro-rendering for scalable, parallel final gathering. *ACM Trans. Graph. (Proc. SIGGRAPH Asia)*, 28(5), 1-8. doi: 10.1145/1618452.1618478
- Rusinkiewicz, S., & Levoy, M. (2000). *QSplat: a multiresolution point rendering system for large meshes*. Paper presented at the Proceedings of the 27th annual conference on Computer graphics and interactive techniques.
- Saito, T., & Takahashi, T. (1990). Comprehensible rendering of 3D shapes. *SIGGRAPH Comput. Graph.*, 24(4), 197-206. doi: 10.1145/97880.97901
- Samet, H. (1995). Spatial Data Structures. In W. Kim (Ed.), *Modern Database Systems, The Object Model, Interoperability and Beyond* (pp. 361-385). University of Michigan: Addison-Wesley/ACM Press.
- Samet, H. (2006). *Foundations of Multidimensional and Metric Data Structures*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.
- Sanders, J., & Kandrot, E. (2010). *CUDA by Example: An Introduction to General-Purpose GPU Programming*: Addison-Wesley Professional.
- Satish, N., Harris, M., & Garland, M. (2009). *Designing efficient sorting algorithms for manycore GPUs*. Paper presented at the Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing.
- Schaefer, S., & Warren, J. (2004). *Dual Marching Cubes: Primal Contouring of Dual Grids*. Paper presented at the Proceedings of the Computer Graphics and Applications, 12th Pacific Conference.
- Scott, K. (2001). On Proebsting's Law: University of Virginia.
- Sederberg, T. W., & Parry, S. R. (1986). Free-form deformation of solid geometric models. *SIGGRAPH Comput. Graph.*, 20(4), 151-160. doi: 10.1145/15886.15903
- Shannon, C. E. (1948). A Mathematical Theory of Communication. *Bell System Technical Journal*, 27(4), 623-656.
- Sigg, C. (2006). *Representation and Rendering of Implicit Surfaces*. PhD Thesis, Department of Computer Science, ETH Zürich.
- Síleš, B. (2012). Atomontage Engine Retrieved 2 June, 2012, from <http://www.atomontage.com>

- Sims, K. (1990). *Particle animation and rendering using data parallel computation* (Vol. 24): ACM.
- Sloan, P.-P., Kautz, J., & Snyder, J. (2002). Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. *ACM Trans. Graph.*, 21(3), 527-536. doi: 10.1145/566654.566612
- Soler, C., Hoel, O., & Rochet, F. (2010). *A deferred shading pipeline for real-time indirect illumination*. Paper presented at the ACM SIGGRAPH 2010 Talks, Los Angeles, California.
- Stocco, L., & Schrack, G. (1995, 17-19 May 1995). *Integer dilation and contraction for quadtrees and octrees*. Paper presented at the Communications, Computers, and Signal Processing, 1995. Proceedings., IEEE Pacific Rim Conference on.
- Sumner, R. W., O'Brien, J. F., & Hodgins, J. K. (1999). Animating Sand, Mud, and Snow. *Computer Graphics Forum*, 18(1), 17-26. doi: 10.1111/1467-8659.00299
- Sung, M., Gleicher, M., & Chenney, S. (2004). Scalable behaviors for crowd simulation. *Computer Graphics Forum*, 23(3), 519-528. doi: 10.1111/j.1467-8659.2004.00783.x
- Suwelack, S., Heitz, E., Unterhinninghofen, R., & Dillmann, R. (2010). *Adaptive GPU ray casting based on spectral analysis*. Paper presented at the Proceedings of the 5th international conference on Medical imaging and augmented reality, Beijing, China.
- Szirmay-Kalos, L., & Umenhoffer, T. (2008). Displacement mapping on the GPU - State of the art. *Computer Graphics Forum*, 27(6), 1567-1592. doi: DOI 10.1111/j.1467-8659.2007.01108.x
- Tagliasacchi, A., Zhang, H., & Cohen-Or, D. (2009). Curve skeleton extraction from incomplete point cloud. *ACM Transactions on Graphics*, 28(3), 1.
- Talton, J. O., Gibson, D., Yang, L., Hanrahan, P., & Koltun, V. (2009). *Exploratory modeling with collaborative design spaces*. Paper presented at the ACM SIGGRAPH Asia 2009 papers, Yokohama, Japan.
- Terzopoulos, D., Platt, J., Barr, A., & Fleischer, K. (1987). Elastically deformable models. *SIGGRAPH Comput. Graph.*, 21(4), 205-214. doi: 10.1145/37402.37427
- Thrane, N., Simonsen, L. O., & Ørbæk, A. P. (2005). *A comparison of acceleration structures for GPU assisted ray tracing*. University of Aarhus.
- Ünsalan, C., & Erçil, A. (2001). Conversions between Parametric and Implicit Forms Using Polar/Spherical Coordinate Representations. *Computer Vision and Image Understanding*, 81(1), 1-25. doi: <http://dx.doi.org/10.1006/cviu.2000.0881>
- Vassilev, T., Spanlang, B., & Chrysanthou, Y. (2001). Fast Cloth Animation on Walking Avatars. *Computer Graphics Forum*, 20(3), 260-267. doi: 10.1111/1467-8659.00518
- Veach, E. (1998). *Robust monte carlo methods for light transport simulation*. Stanford University.
- Voelcker, H. B., & Requicha, A. A. G. (1977). Constructive solid geometry (pp. 46). Rochester, N.Y.: University of Rochester. Production Automation Project.



- Wald, I., Ize, T., & Parker, S. G. (2008). Special Section: Parallel Graphics and Visualization: Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Comput. Graph.*, 32(1), 3-13. doi: 10.1016/j.cag.2007.11.004
- Wald, I., Mark, W. R., Günther, J., Boulos, S., Ize, T., Hunt, W., . . . Shirley, P. (2009). State of the Art in Ray Tracing Animated Scenes. *Computer Graphics Forum*, 28(6), 1691-1722. doi: 10.1111/j.1467-8659.2008.01313.x
- Walter, B., Bala, K., Kulkarni, M., & Pingali, K. (2008). Fast Agglomerative Clustering for Rendering. *Interactive Ray Tracing*.
- Walter, B., Fernandez, S., Arbree, A., Bala, K., Donikian, M., & Greenberg, D. P. (2005). *Lightcuts: a scalable approach to illumination*. Paper presented at the ACM SIGGRAPH 2005 Papers, Los Angeles, California.
- Wang, R. Y., Pulli, K., & Popovic, J. (2007). Real-time enveloping with rotational regression *ACM SIGGRAPH 2007 papers*. San Diego, California: ACM.
- Wang, Y.-S., & Lee, T.-Y. (2008). Curve-Skeleton Extraction Using Iterative Least Squares Optimization. *IEEE Transactions on Visualization and Computer Graphics*, 14, 926-936.
- Watt, A. H., & Watt, M. (1992). *Advanced animation and rendering techniques: theory and practice*: ACM Press.
- Willcocks, C., & Li, F. (2012). Feature-varying skeletonization. *The Visual Computer*, 28(6), 775-785. doi: 10.1007/s00371-012-0688-x
- Williams, L. (1983). Pyramidal parametrics. *SIGGRAPH Comput. Graph.*, 17(3), 1-11. doi: 10.1145/964967.801126
- Yagel, R., & Shi, Z. (1993, 25-29 Oct 1993). *Accelerating volume animation by space-leaping*. Paper presented at the Visualization, 1993. Visualization '93, Proceedings., IEEE Conference on.
- Yang, Y., Xiang, P., Kong, J., & Zhou, H. (2010). *A GPGPU compiler for memory optimization and parallelism management*. Paper presented at the Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation, Toronto, Ontario, Canada.
- Zhang, H., Manocha, D., Hudson, T., & Kenneth E. Hoff, I. (1997). *Visibility culling using hierarchical occlusion maps*. Paper presented at the Proceedings of the 24th annual conference on Computer graphics and interactive techniques.
- Zhou, K., Gong, M., Huang, X., & Guo, B. (2011). Data-Parallel Octrees for Surface Reconstruction. *IEEE Transactions on Visualization and Computer Graphics*, 17(5), 669-681. doi: 10.1109/tvcg.2010.75
- Zordan, V. B., Majkowska, A., Chiu, B., & Fast, M. (2005). *Dynamic response for motion capture animation*. Paper presented at the ACM SIGGRAPH 2005 Papers, Los Angeles, California.