

# **Research on Improving Methods for Visualizing Common Elements in Video Game Applications**

ビデオゲームアプリケーションにおける共通の  
要素の視覚化手法の改良に関する研究

June 2013

Graduate School of Global Information and Telecommunication Studies  
Waseda University

**Title of the project**

**Research on Image Processing II**

**Candidate's name**

**Sven Dierk Michael**

**Forstmann**



## Acknowledgements

Foremost, I would like to express my sincere thanks to my advisor Prof. Jun Ohya for his continuous support of my Ph.D study and research. His guidance helped me in all the time of my research and with writing of this thesis.

I would further like to thank the members of my PhD committee, Prof. Shigekazu Sakai, Prof. Takashi Kawai, and Prof. Shigeo Morishima for their valuable comments and suggestions.

For handling the formalities and always being helpful when I had questions, I would like to thank the staff of the Waseda GITS Office, especially Yumiko Kishimoto san.

For their financial support, I would like to thank the DAAD (Deutscher Akademischer Austauschdienst), the Japanese Government for supporting me with the Monbukagakusho Scholarship and ITO EN Ltd, for supporting me with the Honjo International Scholarship.

For their courtesy of allowing me the use of some of their screenshots, I would like to thank John Olick, Bernd Beyreuther, Prof. Ladislav Kavan and Dr. Cyril Crassin.

Special thanks are given to my friends Dr. Gleb Novichkov, Prof. Artus Krohn-Grimberghe, Yutaka Kanou, and Dr. Gernot Grund for their encouragement and insightful comments.

Last, I would like to thank my family: my parents Dierk and Friederike Forstmann, and my brother Hanjo Forstmann for their support and encouragement throughout the thesis.





## Index

### Index 5

<b>Figure Index .....</b>	<b>9</b>
<b>Table Index .....</b>	<b>14</b>
<b>Chapter 1. Introduction .....</b>	<b>16</b>
1.1 Background .....	16
1.2 Related Work .....	20
1.2.1 Overview .....	20
1.2.2 Terrain .....	20
1.2.3 Static Objects .....	22
1.2.4 Skeletal Animation .....	24
1.3 Purpose .....	26
1.3.1 Terrain .....	26
1.3.2 Static Objects .....	27
1.3.3 Skeletal Animated Objects .....	27
1.4 Approach .....	27
1.4.1 Terrain .....	27
1.4.2 Static Objects .....	28
1.4.3 Skeletal Animation .....	29
1.5 Organization .....	29
<b>Chapter 2. 3D Engine .....</b>	<b>32</b>
2.1 History .....	32
2.2 3D engine in Game Engine .....	33
2.2.1 Game Engine .....	33
2.3 3D Engine Structure and Functions .....	35
2.3.1 Background Pass .....	37
2.3.2 Main Pass .....	37
2.3.3 Post-Processing Pass .....	37
2.4 Visualization by 3D Engine .....	38
2.4.1 Items to be visualized .....	38
2.4.2 Combining multiple modules .....	39
2.5 Module Comparison .....	41
2.5.1 General Comparison .....	41

2.5.2	Terrain Comparison .....	42
2.5.3	Static Objects Comparison .....	43
2.5.4	Skeletal Animation Comparison.....	43
2.6	Conclusion .....	44
<b>Chapter 3. Terrain .....</b>		<b>45</b>
3.1	Goals .....	45
3.2	Related Work.....	45
3.2.1	Procedural Terrain Generation .....	45
3.2.2	Polygonal Visualization of Volumetric Terrains .....	46
3.3	Proposed Method .....	47
3.3.1	Overview .....	48
3.3.2	Differences between the proposed method and Previous Work .....	51
3.4	Clip-Box Algorithm.....	52
3.4.1	Clip-Box .....	53
3.4.2	Data Structure .....	53
3.4.3	Procedural Volume-Data Creation .....	54
3.4.4	Volume-Data to Polygon Conversion.....	55
3.4.5	Nesting.....	58
3.4.6	Moving the View-Point .....	59
3.4.7	Geometry Post-Processing.....	60
3.5	Experimental Results and Discussion.....	62
3.5.1	Implementation .....	62
3.5.2	Immediate Visualization.....	63
3.5.3	Unlimited Terrain Size .....	64
3.5.4	Concurrent execution of generation and visualization .....	66
3.5.5	Demonstration .....	70
3.5.6	Limitations.....	71
3.6	Conclusion .....	71
<b>Chapter 4. Static Objects.....</b>		<b>73</b>
4.1	Goals .....	73
4.2	Related Work.....	73
4.3	Input Data .....	74
4.3.1	Polygon Data Import .....	74
4.3.2	Volume Data Import.....	75
4.3.3	Procedural Voxel Objects: Trees .....	77
4.4	Proposed Algorithm.....	79
4.4.1	Overview .....	79
4.4.2	Improvements over Previous Work .....	80
4.4.3	Difference to Volume Rendering.....	81
4.4.4	Trends in CPU and GPU Development .....	82
4.4.5	Details of the proposed Algorithm .....	84

4.4.6	Pre-Processing.....	86
4.4.7	Level-of-Detail Computation .....	88
4.5	Rendering .....	89
4.5.1	Vanishing-Point .....	89
4.5.2	Concentric Planes.....	89
4.5.3	Plane Parameters .....	90
4.5.4	Rasterizing the Ray Buffer.....	91
4.5.5	Displaying the Ray-Buffer .....	96
4.5.6	Quality Aspects .....	97
4.6	Experimental Results .....	100
4.6.1	Experimental Conditions.....	100
4.6.2	Memory Consumption .....	100
4.6.3	Algorithm Speed .....	101
4.6.4	Rendering Quality .....	104
4.6.5	Comparison to Related Methods and Discussion.....	107
4.7	Conclusion .....	122
<b>Chapter 5. Skeletal Animation .....</b>		<b>125</b>
5.1	Goals .....	125
5.1.1	Spline Skinning .....	125
5.1.2	Deformation Styles.....	125
5.2	Related Work .....	125
5.2.1	Spline Skinning .....	126
5.2.2	Deformation Styles.....	127
5.3	Proposed Method .....	127
5.3.1	Spline Skinning .....	127
5.3.2	Deformation Styles.....	128
5.4	Splines .....	130
5.4.1	Fundamentals .....	130
5.4.2	Spline Aligned Deformation .....	132
5.4.3	Spline Binding.....	132
5.5	Spline Skinning .....	133
5.6	Deformation Styles.....	134
5.6.1	Radial Scale Envelope (Style 1).....	135
5.6.2	Rectangular Scale Envelope (Style2).....	138
5.7	Deformation Styles and Spline Skinning .....	140
5.8	Fast Computation based on the GPU .....	141
5.8.1	Pass 1.....	142
5.8.2	Pass 2.....	143
5.8.3	Pass Three .....	145
5.9	Experimental Results and Discussion .....	145
5.9.1	Experimental Conditions.....	145

5.9.2	Non-Collapsing Geometry .....	145
5.9.3	Small Number of Control Points .....	148
5.9.4	Deformation Style Results .....	148
5.9.5	Computation Speed.....	153
5.9.6	Re-Usability.....	155
5.9.7	Other contributions .....	155
5.9.8	Comprehensive Evaluation.....	155
5.9.9	Limitations.....	158
5.10	Conclusion.....	159
<b>Chapter 6. Conclusion and Future Work .....</b>		<b>161</b>
6.1	Conclusion .....	161
6.2	Future Work.....	163
<b>Bibliography .....</b>		<b>164</b>

## Figure Index

<b>Figure 1.1</b> 3D Engine: Example of a screenshot of DrakenSang, a game by BigPoint/RadonLabs based on the Nebula Device game engine. Screenshot by courtesy of Bernd Beyreuther, head of game-production at BigPoint/RadonLabs. ....	19
<b>Figure 1.2</b> Overview: This diagram shows the three main modules and their data connections. The numbers indicate the chapter in this thesis. ....	26
<b>Figure 2.1</b> Overview of Game Engine and 3D Engine: black triangle: modules explored by this thesis. ....	34
<b>Figure 2.2</b> General 3D engine Overview: This diagram shows the render flow of a common 3D engine including additional post processing modules in a simplified manner. The black triangle marks modules correspond to this thesis' chapters 3 to 5. ....	36
<b>Figure 2.3</b> Main render passes of general 3D engines: upper row: terrain, middle row: static objects, lower row: characters; left column: color buffer, right column: depth buffer (bright: close, dark: far). ....	40
<b>Figure 2.4</b> Illustrated Comparison .....	41
<b>Figure 3.1</b> Overview of the terrain visualization module: Using two threads helps to optimally distribute the rendering and voxel to polygon conversion tasks on modern multi-core-CPU's. ....	49
<b>Figure 3.2</b> Evolution from Clip-Map to Clip-Box (CB); Top left: Nested geometry clip-maps [12] Top right: the Clip-Box based approach as sketch; lower: and the final result as a wire-frame. ....	50
<b>Figure 3.3</b> Clip-Box: Left: the pure Clip-Box geometry; right: CB embedded into the landscape. ....	51
<b>Figure 3.4</b> Adjacency information between surfaces, vertices and voxels. ....	54
<b>Figure 3.5</b> Terrain synthesis: based on CSG (constructive solid geometry) and Boolean operations. ....	55
<b>Figure 3.6</b> Clip-Box connectivity: A simple method (left) yields an erroneous gap, while the improved version (right) solves this problem .....	56
<b>Figure 3.7</b> Seamless connections by improved method.....	56
<b>Figure 3.8</b> Voxel to polygon conversion: Surface creation in the 2D case. ....	57
<b>Figure 3.9</b> Geometry-processing: The four images show the proposed steps to process the initial mesh: (1) direct conversion from volume data (2) smoothed (3) surface subdivision (4) synthetic details. ....	57
<b>Figure 3.10</b> Moving the viewpoint: ○ initial point, ●: the next view-point.....	59
<b>Figure 3.11</b> Caching volume data .....	60
<b>Figure 3.12</b> Triangle subdivision. Upper: adding vertices (white circles), lower: subdivision example.....	61
<b>Figure 3.13</b> Fractal details: Image 1: no fractal details. Image 2: fractal details for the innermost CB. Image 3: Fractal details for the two innermost CBs. Image 4: Fractal details for the three innermost CBs. ....	61
<b>Figure 3.14</b> Smoothing errors and their elimination: Upper-left: Original image with smoothing errors, Upper-right: Result of smoothing; Lower-left: pattern is replaced by lower right pattern. ....	62
<b>Figure 3.15</b> Terrain used for Benchmark.....	63

<b>Figure 3.16</b> Terrain visualization from height-map (real data) Puget Sound region in WA, USA.....	64
<b>Figure 3.17</b> Function plotting and real data: [1] to [3]: three different mathematical functions; [4] conventional iso surfaces.....	65
<b>Figure 3.18</b> Continuous performance: for a flight in the landscape in <b>Fig. 3.5</b> . Upper: polygon vs time, middle: time to visualize one frame vs time, bottom: million polygons per second vs time. ....	68
<b>Figure 3.19</b> Screen-space error: Comparing the highest Clip-Box resolution (192) with lower resolutions: 64 (top-most), 96 (2 <sup>nd</sup> top), 128 (third), and 160 (fourth).....	69
<b>Figure 3.20</b> Examples of synthesizing terrain.....	70
<b>Figure 4.1</b> Rasterization of a single 2D triangle.....	74
<b>Figure 4.2</b> Rasterization of 3D Polygon Data: Imported result of the Happy Buddha PLY Dataset with approximately one million polygons.....	75
<b>Figure 4.3</b> MRI Bonsai Data-Set: A screenshot of the original semi-transparent data rendered in V3, available at The Volume Library. ....	76
<b>Figure 4.4</b> Volume-Data Import: A forest scene created by the voxelized Bonsai data-set. ....	77
<b>Figure 4.5</b> Tree Generation: The tree is created using spheres, random mid-point displacement and finally the L-system.....	78
<b>Figure 4.6</b> Example of Procedural Voxel Trees.....	79
<b>Figure 4.7</b> . Hardware comparison: CPU vs GPU in terms of theoretical GFlops and theoretical memory bandwidth (source: NVidia).....	83
<b>Figure 4.8</b> . Proposed algorithm: (a) side view, (b) top-view. ....	84
<b>Figure 4.9</b> . Pipeline for the proposed method: in “Render Scene” Section, numbers are indicated. ....	85
<b>Figure 4.10</b> Pre-Processing: The initial volume data (left), removal of non-surface voxels (middle), RLE compressed (right). ....	87
<b>Figure 4.11</b> Data structure:Pointer-map: For each pointer-map element’s data, one RLE column is to pointed by a pointer and decoded by RLE . ....	88
<b>Figure 4.12</b> Screen segmentation: VP represents the vanishing point; Seg 1 to 4 refer to segments 1 to 4 respectively. ....	90
<b>Figure 4.13</b> Equidistant and exact raycasting: Left: Equidistant; Right: Exact raycast; Upper: sampling; Lower: Example of rendering.....	91
<b>Figure 4.14</b> Ray mapping: 1 to 4 denote segments 1 to 4; Upper: The temporary buffer with the four segments; Lower: mapping to the screen.....	94
<b>Figure 4.15</b> Skip-Buffer. ....	96
<b>Figure 4.16</b> Smoothing results: Left: without smoothing; middle: smoothed silhouette; right: smoothed interior part.....	97
<b>Figure 4.17</b> Smoothing steps: a) Target pixel; b) Find minimum depth (Z); c) Box-filter with threshold, scaled according to the minimum depth; d) Result. ....	98
<b>Figure 4.18</b> Anti-aliasing (AA): left: Non AA; middle: 2x1 AA; right: 2x2 pixel AA. ....	98
<b>Figure 4.19</b> Normals: The depth-buffer can successfully be utilized to compute normal vectors on-the-fly (Left). These can be utilized for shading and further enhanced with screen space ambient occlusions (Right). ....	99
<b>Figure 4.20</b> Scenes used for tests: Handcrafted mansion (upper-left), Bonsai forest with 3000 trees (upper-right), Hotei, or Happy Buddha, (middle left) and a Procedural Landscape with about 4000 visible trees (middle right), the Stanford Dragon (lower-left) and the Stanford Bunny (lower-right). Unit for the number is given in voxels.....	101
<b>Figure 4.21</b> GPU vs CPU: The GPU version running on an NVidia GTX 285 is compared to the CPU version (Intel Q6600 4x3Ghz).....	102

<b>Figure 4.22</b> Raycasting vs Splatting (1): left: the proposed RLE method; middle: quad splatting; right: triangle splatting.....	103
<b>Figure 4.23</b> Raycasting vs Splatting (2): left: the proposed RLE method; middle: quad splatting; right: triangle splatting.....	103
<b>Figure 4.24</b> Quality: To show the ability to render at high quality, a complex test scene with many fine details was created and rendered at $512 \times 348$ pixel with $2 \times 2$ AA as well as no AA for a comparison. Note that $2 \times 2$ AA successfully removes aliasing artifacts for distant pixels. ....	105
<b>Figure 4.25</b> Raycasting vs Splatting (3): render quality for geometry close to the camera; left: the proposed RLE method; middle: quad based splatting; right: triangle splatting.....	105
<b>Figure 4.26</b> Richtmyer-Meshkov dataset.....	106
<b>Figure 4.27</b> Memory consumption (3): QSplat, Sparse Voxel Octree and Triangle raycasting compared to the proposed method.....	111
<b>Figure 4.28</b> Speed comparison (1): Triangle rasterization compared to the proposed method for multiple camera configurations.....	112
<b>Figure 4.29</b> Speed comparison (2): Triangle rasterization compared to the proposed method for multiple camera configurations.....	112
<b>Figure 4.30</b> Speed comparison (3): Triangle rasterization compared to the proposed method for a complex scene with multiple Imrod models. ....	113
<b>Figure 4.31</b> Speed comparison (4): GigaVoxel and this thesis' Sparse Voxel Octree Raycasting (that was implemented for comparison purposes) is compared to the proposed method for multiple camera configurations. The GigaVoxels [43] screenshots are with courtesy of Cyril Crassin.....	113
<b>Figure 4.32</b> Speed comparison (5): Sparse Voxel Octree Raycasting of Jon Olick is compared to this thesis' Sparse Voxel Octree Raycasting method and to the proposed method. The Sparse Voxel Octree Raycasting screenshot (left) is with courtesy of Jon Olick.....	114
<b>Figure 4.33</b> Speed comparison (6): this thesis' Sparse Voxel Octree and triangle raycasting are compared to the proposed method.....	114
<b>Figure 4.34</b> Speed comparison (7): this thesis' Sparse Voxel Octree is compared to the proposed method and to triangle raycasting. ....	115
<b>Figure 4.35</b> Speed comparison (8): this thesis' Sparse Voxel Octree is compared to the proposed method and to triangle raycasting.....	116
<b>Figure 4.36</b> Speed comparison (9): this thesis' Sparse Voxel Octree is compared to the proposed method.....	116
<b>Figure 4.37</b> Speed comparison (10): the proposed method is compared to QSplat.....	117
<b>Figure 4.38</b> Visual Precision (1): the proposed method is compared to triangle rasterization. ....	117
<b>Figure 4.39</b> Visual Precision (2): the proposed method is compared to GigaVoxels. The GigaVoxels screenshots is with courtesy of Cyril Crassin.....	118
<b>Figure 4.40</b> Visual Precision (3): the proposed method is compared to triangle raycasting. ....	118
<b>Figure 4.41</b> Visual Precision (4): the proposed method is compared to QSplat.....	119
<b>Figure 4.42</b> Visual Precision (5): the proposed method is compared to QSplat, close-up view. ....	120
<b>Figure 4.43</b> Visual Precision (6): the proposed method is compared to Sparse Voxel Octree Raycasting. The Sparse Voxel Octree Raycasting screenshot (left) is with courtesy of Jon Olick.....	120
<b>Figure 4.44</b> Summary of memory consumption per element. ....	121
<b>Figure 4.45</b> Summary of computation speed for high screen resolutions (2048x768).....	121
<b>Figure 4.46</b> Summary (3): visual precision. ....	122

<b>Figure 5.1</b> Proposed spline skinning and deformation styles. Top: spline skinning, middle: spline aligned deformations, bottom: deformation styles. ....	129
<b>Figure 5.2</b> Bend and Twist deformations: left: FFD, middle: SSD, right: spline aligned deformation. ....	129
<b>Figure 5.3</b> Spline functions: In the upper row, the short-listed spline functions are compared. The lower row shows the ability of the spline to adjust the stiffness by parameter $a$ ...	131
<b>Figure 5.4</b> The spline coordinate system: left: the spline function in violet together with the spline's coordinate system, where the spline's tangent is indicated in blue, the normal in red, the bi-normal in green and the origin of each coordinate system in yellow; right: an example deformation. ....	132
<b>Figure 5.5</b> The binding process: (left) the perpendicular mapping of vertex $v$ to the spline by using binary search, (right) a way to determine the search direction in each step. ....	133
<b>Figure 5.6</b> Spline skinning: skinning weights .....	134
<b>Figure 5.7</b> Deformation Styles .....	135
<b>Figure 5.8</b> Radial Scale Envelope: The lower left side shows an example envelope while the upper right side shows the concentric scaling of $v$ in relation to the spline.....	136
<b>Figure 5.9</b> Scale Textures: The upper row shows the three scale textures that were used to create Style 1 in <b>Fig. 5.7</b> and an example object where the textures are applied. The lower row shows the pose-dependent weight calculation to apply the three textures, where red corresponds to the weight of the frontal scale texture, green to the lateral and blue to the radial. ....	137
<b>Figure 5.10</b> Rectangular Scale Envelope: The left side shows the application to an example object while the right side shows the scaling of $v$ corresponding to $bB$ and $bN$ .....	138
<b>Figure 5.11</b> Rectangular Scaling: The upper row shows the three scale functions. The lower row shows the pose-dependent weights. ....	139
<b>Figure 5.12</b> GPGPU based accelerations of the computations.....	141
<b>Figure 5.13</b> Shared Texture 1. Vertex normals ( $N$ ), weights ( $W$ ), bone indices ( $I$ ) and spline offsets ( $O$ ) are stored in one texture. The NWIO pattern applies for the empty space of the texture as well. ....	143
<b>Figure 5.14</b> Shared texture 2. Scale textures and scale curves are stored in one large texture. Each texel's RGBA value in the texture (left) is used as defined on the right side. ....	144
<b>Figure 5.15</b> Spline discretization: Pre-computing all splines is one of the key improvements in the implementation to increase the speed.....	145
<b>Figure 5.16</b> Basic Spline Skinning: Three poses for an animated chest-arm-shoulder model. The binding pose (left), simple bend operation (middle), and, finally, bending combined with two twist operations, for the hand and for the body (right).....	146
<b>Figure 5.17</b> Spline Skinning with a simple muscle deformation style in 8 frames. ....	146
<b>Figure 5.18</b> Facial Animation: Lips and cloth folds can be animated using spline skinning. Up-left the final animation and up-right the bind pose, where each color R,G,B is assigned to one spline. The lower part shows an animation sequence.....	147
<b>Figure 5.19</b> Spline Skinning compared to matrix skinning for multiple joints. ....	147
<b>Figure 5.20</b> Metal: This Figure shows the animation of designed metal, which smoothly deforms as the pose changes. Upper row: deformation styles are applied; lower row: spline skinning without deformation styles.....	149
<b>Figure 5.21</b> Muscles: Created muscles can easily be applied to different characters simultaneously. ....	149
<b>Figure 5.22</b> Muscles on David:middle: without, right: with. ....	150
<b>Figure 5.23</b> Hollow materials: Here an animation of crunching an empty can.....	150



<b>Figure 5.24</b> Self-intersections: Demonstrated are self-intersections (up-left) and the efficient removal of self-intersections (up-right) as well as modeled lateral bulges. The used curves are shown in the lower row. ....	151
<b>Figure 5.25</b> Cloth: This example shows the algorithm's ability to imitate cloth-like wrinkles. Upleft and down-left: non-style version. Up-right and down-right: deformation styles version. The red circles show the affected region. ....	152
<b>Figure 5.26</b> Benchmark results: Spline skinning indicates basic spline aligned deformation, Radial adds Drad, Rect adds Drect and Rect+Radial adds both. ....	153
<b>Figure 5.27</b> SSD, DQS and Spline Skinning Methods Compared (1): upper row: twist operation, lower row: bend operation. ....	156
<b>Figure 5.28</b> QS and DQS Skinning Methods Compared (2). Left: artifact-free twist operation; right: cloth deformation. Images with courtesy of Ladislav Kavan, Skinning with Dual Quaternions [20]. ....	157

## Table Index

<b>Table 2.1</b> Module comparison: In the table, “+” indicates that the feature is available and “-” indicates that the feature is unavailable. For unknown support the fields are left empty.....	42
<b>Table 3.1</b> Performance analysis: In the upper row, update and render times for one CB resolution (128) are analyzed in detail, while the lower row compares the performance of different CB resolutions.....	66
<b>Table 4.1</b> Benchmark Tests: The RLE element count in the frustum (Total), the processed element count (Proc) and the rendered element count in million (Ren). The resolution is stated in voxel. Further, Fps denotes frames per second and Speed is given in million RLE elements per second (Elems/s) .....	100
<b>Table 4.2</b> Memory consumption (1): Triangles compared to the proposed method (voxel) .....	111
<b>Table 4.3</b> Memory consumption (2): GigaVoxels [37] compared to the proposed method for the Sponza scene. The GigaVoxels screenshot is with courtesy of Cyril Crassin .....	111
<b>Table 5.1</b> Texture formats: Here an overview of the used texture and buffer formats.....	142
<b>Table 5.2</b> Timing breakdown: .....	154
<b>Table 5.3</b> Related methods and their features. ....	156



## Chapter 1. Introduction

### 1.1 Background

Since the invention of computers in the 1940's by Konrad Zuse (Z3, 1941 and Z4, 1945), John Presper Eckert and John William Mauchly (ENIAC, 1946)<sup>1</sup>, data acquisition, data processing, and data visualization have been elementary tasks in computer science.

The history in computer graphics (CG) has begun only shortly after the invention of the computer in the 1950's. Already at that time, General Motors started to research and develop the previously mentioned computer aided design system<sup>2</sup> for virtual car design. At that time, graphics had represented entirely by vector graphics on cathode ray tubes (CRT), rather than pixel graphics. Graphics at that time only consisted of dots and lines.

The next milestone in the development of CG was the visualization of avatars in the 1960's. The main elements of today's 3D graphics such as raster-graphics, ray-tracing, texture mapping, bump mapping, reflection mapping, and the depth-buffer were finally developed in the 1970's.

In the 1980's, the visualization quality was further improved, and methods such as 3D graphics related technologies and global illumination were invented. As 3D graphics technologies grew, CG started being used for many applications such as scientific and/or engineering, medicine, art, education and entertainment. The 1980's were also the years when CG became important for the entertainment business. CG appeared in Hollywood movies such as *Star Trek* and *Tron*<sup>3</sup> as well as in 3D video games such as *Cube Quest*<sup>4</sup>.

As the video game market is growing, the segment of the game industry also keeps evolving. Already in 2008, the video games industry had grown larger than the movie industry<sup>5</sup>. Nowadays, the development of a top video-game, a so called triple A (AAA) title, cannot anymore be carried out by a few people in a garage. Large studios are mandatory, and production costs of up to 100 million US Dollars<sup>6</sup> and more<sup>7</sup> have to be taken into account. In the video game market, Japan plays a particularly significant role as its market share is, with a revenue of over 7 Billion USD as of 2008<sup>8</sup>, the world's second largest.

---

<sup>1</sup> **United States Army**, ENIAC Electronic Numerical Integrator And Computer. 1946.

<sup>2</sup> **General Motors, IBM**. DAC-1 Design Augmented by Computer. 1960.

<sup>3</sup> **Lisberger, Steven**. *Tron*. Walt Disney Productions, 1982.

<sup>4</sup> **Simutrek Inc.** *Cube Quest*. 1983.

<sup>5</sup> **TomsGuide**. Video Games Outsell Movies <http://www.tomsguide.com/us/Games-DVD-Blu-ray-Economy,news-3364.html>. 2012.

<sup>6</sup> **DigitalBattle**. Top 10 most expensive video games budgets ever.  
<http://www.digitalbattle.com/2010/02/20/top-10-most-expensive-video-games-budgets-ever>. 2010.

<sup>7</sup> **GameBandits**. *Star Wars – The Old Republic*.  
<http://www.gamebandits.com/news/pc/star-wars-the-old-republic-new-voidstar-trailer-released-20393/>. 2012.

<sup>8</sup> **Analysis: Trends in the Japanese Game Market**. [http://www.gamasutra.com/php-bin/news\\_index.php?story=20461](http://www.gamasutra.com/php-bin/news_index.php?story=20461). 2008.

In the 1980's, first video games were developed independently, without using any third-party software. The reason for that was that most games were developed for arcade machines, which had unique hardware. The program code was very platform-dependent; i.e., written for one platform, thereby it could not be reused on another. On home computers of the time, such as the Commodore 64, Amiga 500 and Atari ST, third-party game engines were not yet common. Games at that time did not require complex algorithms and had much less lines of code, which made the development easier. Therefore, source code was rarely re-used. Examples of re-use of code were sequels to a game, but in that case they re-used the entire game, not an abstract part such as a game engine.

With the evolution of Intel x-86 architecture and the beginning of 3D graphics in the early 1990's, a significant increase in the complexity of game development happened. At that time, the first game engines for 3D games, such as the Doom engine<sup>9</sup> and the Build engine<sup>10</sup> appeared. Both engines used software rendering and provided simple 3D support. Items and characters in the game were represented by billboards. The camera's motion was limited as well, and not able to be translated and rotated with six degrees of freedom. For the level design in Doom, room over room was impossible. Due to several of those limitations, both engines were often referred to as 2.5D game engines, rather than real 3D game engines.

Later in the late 1990's, real 3D engines that could handle arbitrary geometry configurations appeared, such as the Quake engine<sup>11</sup> and the Unreal engine.<sup>12</sup> They did not have 2.5D engines' limitations and for the first time provided support for 3D hardware acceleration. The characters were full 3D models, rather than 2D billboards.

Successors of the Quake engine were named *ID tech*. Newer ID tech as well as the Unreal Engines support more graphical features, realistic physics and better artificial intelligence (AI) for enemies and AI multiplayer characters (bots).

While the ID tech game engines were dominant among games in the 1990's, the Unreal engines became more popular, starting in the year 2000. The first and second most popular game genres using third party game engines were first person shooters and role playing games, respectively.

More important things for game engines in recent history include not only the feature support, but also the support for game consoles, PCs and mobile devices together. In particular, the share of mobile devices increases as they become faster and further have better 3D hardware acceleration.

Due to the continuing increase in complexity, more and more modules of modern game engines are developed by third parties as individual products (middleware). It is not common anymore that the entire game engine is developed only by one single company. A few examples of third

---

<sup>9</sup> **ID software.** Doom Engine. 1993.

<sup>10</sup> **Silverman, Ken.** *Build Engine*. 3D Realms: 1997.

<sup>11</sup> **ID Software.** Quake Engine. 1996. <http://www.idsoftware.com/games/quake/quake>.

<sup>12</sup> **EPIC MEGAGAMES.** Unreal Game Engine. <http://www.unrealengine.com/>.

party software are Speed-tree<sup>13</sup>, Havok<sup>14</sup>, Fork Particle<sup>15</sup>, the Simul Weather SDK<sup>16</sup> and the City Engine<sup>17</sup>.

The term “Game Engine” is defined as a software framework that includes all modules required for producing a video game. One of the modules of game engines is the so called “3D engine” in case of a 3D game and “2D engine” in case of a 2D game, respectively.

Different from the term “Game Engine”, “3D engine” is not strictly defined. In history, often entire game engines have also been entitled as “3D engine”, even if they are very complex.

Recent 3D engines can accommodate multiple applications and can take care of consistencies between the multiple applications such as the view-point and view-angle synchronization so that the visualized CG images generated by the multiple applications are properly combined.

In case of state of the art 3D engines, such as the Nebula 3D engine<sup>18</sup>, as shown in **Fig. 1.1**, objects to be visualized are categorized as follows.

- Terrain
- Static objects
- Skeletal animated objects
- Plants
- Sky and clouds

For each of these objects, an application is embedded into the Nebula 3D engine so that these objects are consistently combined in 3D very efficiently.

---

<sup>13</sup> **SpeedTree**. SpeedTree. <http://www.speedtree.com/>.

<sup>14</sup> **Havok**. Havok Physics. 2000. <http://www.havok.com>.

<sup>15</sup> **Fork Particle**. Fork Particle. <http://www.forkparticle.com>.

<sup>16</sup> **Simul Software Ltd**. Simul Weather SDK. 2009. <http://www.simul.co.uk/>

<sup>17</sup> **ERSI**. City Engine. 2008. <http://www.esri.com/software/cityengine/>.

<sup>18</sup> **RadonLabs**. Nebula Device. <http://sourceforge.net/projects/nebuladevice/>.



**Figure 1.1** 3D Engine: Example of a screenshot of DrakenSang, a game by BigPoint/RadonLabs based on the Nebula Device game engine. Screenshot by courtesy of Bernd Beyreuther, head of game-production at BigPoint/RadonLabs<sup>19</sup>.

For a modern video game, the entire process until a game ends up in the stores has become quite complex and is split up into the following steps:

- Game Development
  - Game Design (game type, storyline, character design, level design)
  - Software development (Research and Development, game engine, tools, web portal)
  - Graphics (modeling, animating, motion capture, painting textures & backgrounds, web)
  - Audio (music composing, sound effects, narration)
- Beta testing phase
- Marketing (TV commercials, internet adverts, getting reviewed by magazines)
- Shipping (via BlueRay, DVD, internet and distributors)

<sup>19</sup> BigPoint. BigPoint. 2002. <http://www.bigpoint.com>.

This thesis, focuses on the research part of the software, in particular the 3D graphics algorithms to be accommodated into the 3D engine. More specifically, this thesis addresses how to improve methods for visualizing terrain, static objects and skeletal animated objects, which are elements common to video game contents.

## 1.2 Related Work

This section reviews the related work to the three topics described in Section 0 as well as an overview of general CG methods related to 3D engines.

### 1.2.1 Overview

In CG, many methods have been developed over time. Basic and quite old methods of CG have been overviewed in Chapter 0. Newer methods relevant to 3D engines are reviewed in this section. In 3D engines it is important to visualize complex geometries with accurate shading at high frame-rates. Concerning shading in real-time, particularly shadows are challenging. Conventional ray tracing based methods have not been suitable in game engines for a long time, as they are too slow. Therefore, faster techniques such as shadow mapping [1] and shadow volumes [2] have been used. They can avoid ray tracing and suit well for triangle based visualizations.

Another methods important for 3D engines include techniques that improve conventional bump-mapping in order to let flat surfaces appear bumpy. Here, parallax occlusion mapping [3], which was first published in 2004, is an evolution that creates much more realistic bumps than previous bump-mapping. Rather than just altering the normal vector, parallax occlusion mapping creates a parallax effect, which lets the bumps on a planar surface appear in 3D. Furthermore, it provides self-shadows and correct silhouettes in object borders on the screen.

Recent methods related to animation evolve existing methods to realistic and physically correct skin deformations, proper cloth simulation [4], or hair animation and visualization. Another work in the area of animation focuses on creating realistic character locomotion animations procedurally. In the area of visualization, splat [5] and voxel-based [6] representations are researched as an alternative to visualizing geometries by triangles. For achieving faster visualizations, level of detail (LOD) was researched, where methods such as HLOD [7] and Far Voxels [8] were proposed to visualize large and complex scenes. To achieve proper lighting without using pre-computations, global illumination in real time was developed by Crassin et al [9]. The computation of global illumination is already complex, even for state of the art ray-tracing methods; therefore, achieving real-time performance is a challenge.

### 1.2.2 Terrain

In the scene, terrain is a ground surface on which characters, buildings or other static objects can be placed. It is one of the most crucial parts of a video game and virtual 3D worlds in general. The research of terrain visualization for CG applications mainly started in 1996 with height-map based



terrains using continuous level of detail [10] and roaming [11]. Over time, as computers evolved, and as hardware acceleration appeared, methods were changed. Nowadays, research focuses not only on improving height map based methods, but also volume data based methods to allow terrain features such as overhangs, caves and arches. One of the most successful movies, “Avatar”<sup>20</sup>, might be the best example for impressive volumetric landscapes, where a fantasy world called Pandora, with large floating rocks is one of the main elements in that movie.

In video games, height-map based methods were commonly used for rendering terrain [11] [12]. While height-map based approaches were largely sufficient for featuring isometric perspective games such as real-time tactics, first person games<sup>21</sup> demand more interesting landscapes, including concavities and overhangs. Therefore, recently height-maps were step-by-step replaced by volumetric terrains such as Pandromeda<sup>22</sup>, [13], [14] so that the above mentioned interesting terrain landscapes can be generated.

The creation of these complicated, and thus interesting, volumetric terrains necessary for long-range walk-through environments can either be achieved by manual operations<sup>23</sup>, or procedural methods [13], [14]. Manual creation by content creators is expensive in terms of time and financial cost and thus should be reduced as much as possible.

Procedural methods save time for creators; however, they produce huge amounts of data, which needs to be stored and loaded again at run-time. To solve this issue, procedural methods need to be integrated into the video game for generating contents at run-time. Furthermore, video game players tend to get bored if the same contents are presented each play repeatedly. To avoid repetitions of the same contents, procedural methods should be able to create new and interesting contents in each play.

So far, none of the existing methods that can visualize volumetric terrains is able to display run-time generated procedural terrain. Overcoming this limitation is an important step in the development though. All existing methods to visualize volumetric terrains require a time consuming pre-processing step in order to convert the terrain data into a format that is optimal for the particular method. Therefore, they need to store the temporary generated data on mass-storage devices. The visualization of procedural volumetric terrain data that is generated on run-time is still an open area of research that has not been solved yet.

The limitations of existing terrain technologies are briefly summarized as follows:

- Pre-calculation: Existing methods that are able to visualize large volumetric terrains in real-time require a time consuming pre-processing prior to the visualization. The pre-processing is required to compute level-of-detail representations of the original high resolution terrain data. Each method uses its own individual level of detail format.
- Limited terrain size: As existing methods need to apply the pre-processing to the entire

<sup>20</sup> Cameron, James. *Avatar the movie*. Twentieth Century Fox Film Corporation, 2009.

<sup>21</sup> First person games are video games where the player sees the scene in ego-perspective. It is equal to seeing through the eyes of the virtual main character.

<sup>22</sup> Pandromeda. <http://www.pandromeda.com>. 2012.

<sup>23</sup> 3D-Coat. <http://3d-coat.com/>. 2012.

terrain data, the terrain size is limited by the memory capacity provided by the hardware. Infinite sized terrains cannot be handled by existing methods.

- Absence in synthetic volumetric terrain generation on the fly: Existing methods do not achieve both, an automatic volumetric terrain generation and the visualization simultaneously. There are methods that are able to generate terrain procedurally, and there are different methods that allow existing data to be visualized in real-time. However, presently no method exists that allows the dynamic, on-the-fly generation of volumetric terrain data, in parallel to the visualization.

### 1.2.3 Static Objects

Traditionally, static objects are visualized using polygon based rendering, which has been more efficient than point-primitive (splatting) or voxel-based rendering (voxel comes from Volume-Pixel) over a long period. However, polygonal models are becoming more and more detailed, leading to dense meshes where each polygon merely covers a few pixels on the screen. Once polygonal meshes become so dense, results of rendering by using polygons, point primitives, or voxels do not show significant differences in quality and rendering speed. This means that voxel and point-based rendering methods gain more importance. The reason for this is that as the rasterized size of voxels, splats and polygons become similar, rendering a voxel or splat employs considerably less computation than rendering a polygon.

Previously, an advantage of polygon-based rendering was the ability to save memory by using repeated textures. Voxel- and splat-based rendering inherently use unique texturing, so there is no benefit in memory consumption from using repetitive texturing. However, according to a recent trend to use unique non-repeating textures for each object on the screen (Megatexture technology<sup>24</sup>), the memory consumption for polygon-based rendering sharply increases. Therefore, if this trend continues, the memory consumption of voxel and splat-based rendering becomes comparable with unique textured polygon rendering.

Voxels are basically three dimensional pixels. Voxels are most commonly used in connection with volume data. Each voxel represents one atom of a volume data set. First, voxels were mostly used in visualizing medical scans (such as CT or MRI). Over time, they also found their application in video games. An early game based on voxels is called Comanche<sup>25</sup>. They used voxels to visualize the terrain. At that time, polygon rendering was not hardware accelerated and therefore quite slow. Voxel graphics provided more details at the same rendering speed.

The founder of one of the most famous game companies, John Carmack, already forecasted the come-back of voxels in 2008<sup>26</sup>. He explained a technique called sparse voxel octree raycasting [15] where sparse means that only surface data is stored as voxels – no solid data. On the CPU, several methods for ray-casting voxel and volume data have already been developed. However,

<sup>24</sup> Carmack, John. The Megatexture technology. 2006.

[http://www.team5150.com/~andrew/carmack/johnc\\_interview\\_2006\\_MegaTexture\\_QandA.html](http://www.team5150.com/~andrew/carmack/johnc_interview_2006_MegaTexture_QandA.html).

<sup>25</sup> NovaLogic. Comanche series. 1992. [http://en.wikipedia.org/wiki/Comanche\\_series](http://en.wikipedia.org/wiki/Comanche_series).

<sup>26</sup> Carmack, John. Id Tech 6, Ray Tracing, Consoles, Physics and more. <http://www.pcper.com/article.php?aid=532>.

since the graphic cards became general purpose computation units that can execute common C-code, a next challenge is voxel ray-casting on the GPU.

Level of detail (LOD) [16] technologies for polygons and voxels and are compared. LOD is important to accelerate rendering and to decrease the run-time memory consumption. For polygonal objects, LOD is usually handled as follows. First, a set of polygonal objects with different levels of detail is created by an artist. Then, at run-time, the proper LOD of the object is selected according to the view distance from the view-point. Voxels can handle LOD more efficiently than polygons, because voxel data can be easily down-sampled for representations in lower details. Therefore, it is not necessary for the artist to design separate models of the same object for each level of detail because the different levels of detail can be generated automatically. Another advantage of voxels over polygons is that Boolean operations (consisting of union, intersection and difference operations) can be applied much easier to voxels than to polygons. With polygons, complex algorithms are required, and taking care of exceptional cases is needed. With voxels, the Boolean operation is simply performed per voxel, which is much simpler as it is very similar to a Boolean value; the voxel is either set or unset, like a Boolean value which is either 0 or 1. Despite all these advantages of voxels over polygons, it should be noted that deformations and skeletal animations in real-time still pose a challenge for voxel-based representations.

Point- and voxel-based rendering are very similar. However, the major difference between voxel-based rendering and point-based rendering is that voxels occupy a well-defined cubic portion of volume in space, while point-based methods usually approximate the geometry by 2D splats, such as in QSplat [5]. Due to the fact that splats are 2D approximations of a 3D object, point-based algorithms need many special processes (such an adaptive sampling) to be robust and efficient. Voxel-based algorithms are generally more robust without the need for such exceptions, because a voxel covers a well-defined cubic portion of space in 3D, rather than a 2D approximation.

In the past, voxel-based methods optimized for sparse surface data used to be applied to the CPU. After the invention of the GPU and NVIDIA CUDA<sup>27</sup>, it is possible for the first time to execute such complex algorithms on the GPU in a parallel fashion. The GPU provides hundreds of single instruction multiple data (SIMD) units, which allow co-execute algorithms in a highly parallel manner. So far, the use of the GPU has been very limited, though; e.g. C-like programs using pointers cannot be executed. This was improved by the development of NVIDIA CUDA. Their API allows, for the first time, to use the GPU as a real general purpose processor.

However, developing efficient ray-casting methods using the GPU is more than a simple implementation task. Additional research is required to optimize the performance for the novel parallel architecture. Furthermore, since the data of large detailed voxel objects consumes a significant amount of memory, which requires further research on efficient data structures and memory compression methods [17]. Another issue of existing voxel visualizations is the aliasing for close geometry. So far, the blocky appearance of voxels close to the camera has not yet been efficiently solved.

---

<sup>27</sup> NVIDIA. Compute Unified Device Architecture (CUDA). 2008.

The limitations of existing methods are summarized as follows:

- Conventional triangle based rasterization as well as splatting based methods do not scale well for complex scenes in terms of memory consumption and rendering speed due to overdraw<sup>28</sup>. The speed for rasterization is linear, which is not as good as raycasting methods which provide logarithmic scaling.
- Existing voxel-based raycasting methods are not memory efficient due to their data structures.
- Triangle based ray-casting methods significantly consume larger memory than other methods, such as splatting.

Therefore, it can be said, that at present no method can provide low memory consumption and high computation speed simultaneously.

## 1.2.4 Skeletal Animation

Animation has been important in CG and visualization ever since. In 1961, the first computer animation by Edward Zajack named *Two-Gyro Gravity-Gradient Attitude Control System*<sup>29</sup> was presented, shortly after the history of CG began. Over time, various methods have been developed to animate virtual characters, where *skeletal animations*<sup>30</sup> of characters that consist of polygons are most important. To animate them in a smooth manner, the mesh vertices need to be interpolated between key-frames. Soon after this was first done in a simple manner, advanced skeletal methods were developed. They allowed the animations to be created in a hierarchically structured manner, such that not every vertex needs to be animated manually.

In the meantime, skeletal character animation has become one of the major components in most digital productions, including cinematic productions and interactive applications such as video games. A challenge in particular is the skin deformation, which is a major component in character animation.

To animate the deformation, many approaches have been developed over time; e.g. free-form-deformation (FFD) based techniques over skeletal methods and advanced algorithms, which also take into account topology of a mesh, physical constraints, and even authentic data from laser-scans.

---

<sup>28</sup> In complex scenes, many objects occlude each other. Since all objects need to be drawn for rasterization, some regions of the screen are drawn over multiple times. This occurrence is defined as overdraw.

<sup>29</sup> **Bell Labs.** A Two Gyro Gravity Gradient Altitude Control System. 1961.  
[http://en.wikipedia.org/wiki/History\\_of\\_computer\\_animation](http://en.wikipedia.org/wiki/History_of_computer_animation).

<sup>30</sup> **Skeletal Animation:** The definition of skeletal animation derives from the hierarchic bone structure used in computer graphics, which is similar or equal to a skeleton. In case that representing an existing skeleton of a real biologic life form is focused, the term “skeletal” is replaced by “skeleton”.

In recent years, the so-called linear blend or matrix skinning [18], also known as skeletal subspace deformation (SSD), has widely been used for skeletal animation. SSD is the most popular method among all authoring tools and interactive applications. The key of SSD's success is that it is simple and well-balanced in terms of quality, speed, and complexity.

On the other hand, SSD is still not perfect, because its deformations expose the well-known candy-wrapper effect for twisting operations and collapsing geometry while bending. This issue is very annoying for artists and has inspired many researchers to propose new solutions and alternative methods. Unfortunately, most of these methods turn out to be not practical, because these are too complex or demanding to accomplish real-time execution. This is critical for real-time applications such as video games, which require fast computations. Also in rendering systems for cinematic productions that do not require real-time, computation speed is an important and non-negligible factor, because time-consuming computations raise the production cost of a movie.

To solve the above-mentioned issues of SSD, the following two quaternion based skinning methods were developed by Ladislav Kavan: Spherical blend skinning (SBS) [19] and Dual Quaternion Skinning (DQS) [20]. They change the interpolation scheme from matrices to quaternions or even dual quaternions. This change cannot prevent deformation artifacts completely, but it successfully avoids effects as collapsing geometry for large bend angles. Furthermore, they preserve a high computational speed, which is not as high as SSD, though. In case of quaternion skinning, about 78% of the speed of SSD is achieved, and in case of DQS, 72% the speed of SSD is achieved ([19], [20]). However, for creating complex deformations of a spine or facial animation for instance, many joints are required. At present, existing skinned skeletal animation methods suited for use in real-time applications do not provide an efficient way to simplify this.

Another important issue to be solved for an animation system's success is the ability for adjusting the degree of freedom of deformation. In this sense, pose-space-deformation (PSD) has made an important advancement that allows artists to design each pose of an animation individually. PSD, which is an animation system that is mounted on top of the basic skeletal animation system, uses the output of the basic skeletal animation system and modifies the output by a post-process. However, PSD has a significant restriction: if an artist enhances a certain pose – for example by modeling a joints muscle- or cloth-like deformation behavior – it is not possible to reuse this particular behavior for any other joint. It needs to be modeled for each character individually.

Existing limitations are summarized as follows:

- Matrix skinning is fast, but exposes deformation artifacts for large bend angles in joints.
- Quaternion Skinning and Dual Quaternion Skinning have smaller artifacts than SSD, but they are not as fast.
- Existing skeletal animation methods suitable for 3D engines require many control joints for creating a spine or complex facial animations. They do not provide an efficient way to

simplify this task.

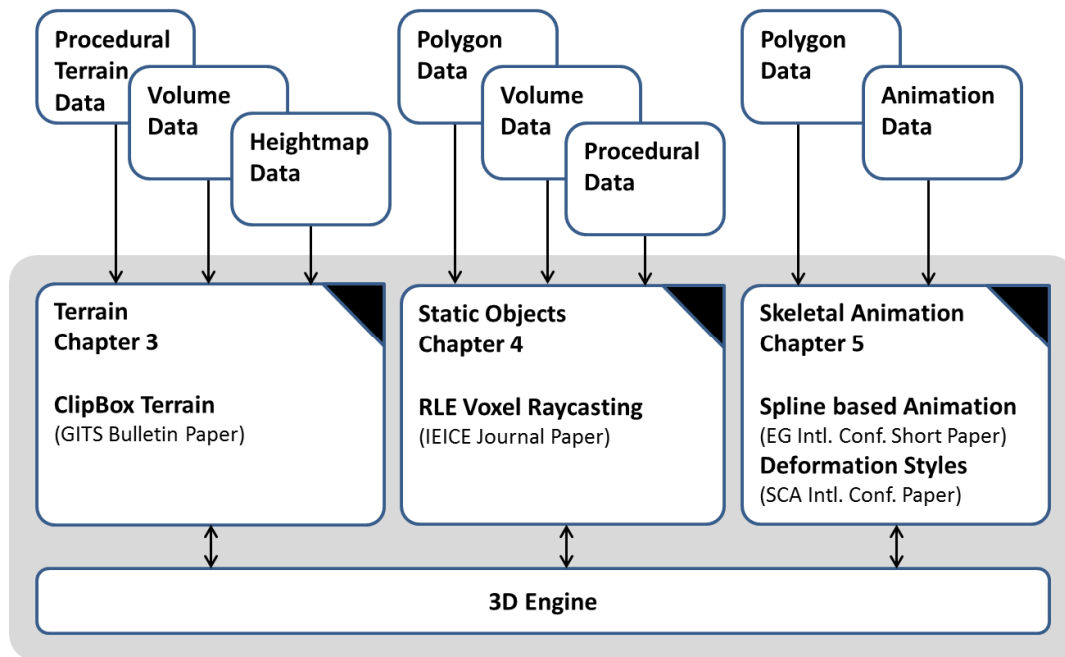
- Custom deformations such as muscle behaviors or cloth wrinkles need to be modeled for each character individually using PSD. Existing methods do not allow re-use of existing custom deformations for different joints or characters.

### 1.3 Purpose

As shown in **Fig. 1.2**, this thesis assumes that the three visualization modules: terrain, static objects and skeletal animation are embedded into a 3D engine.

The purpose of this thesis is to improve the three main visualization modules.

The specific goals of the three modules are described as follows.



**Figure 1.2** Overview: This diagram shows the three main modules and their data connections. The numbers indicate the chapter in this thesis.

#### 1.3.1 Terrain

The goals for overcoming the limitations described in Section 1.2.2 are as follows. First, the pre-computation-free visualization of volumetric terrain data should be achieved, because all existing methods require a pre-computation step prior to the visualization. Second, infinite sized terrain

should be achieved, because existing algorithms can visualize only limited sized terrain as they need to pre-process the entire terrain data. Last, new procedural terrain data should dynamically be generated on the fly in parallel to the visualization, to allow the visualization infinite sized terrains, where this is impossible for the conventional methods.

### 1.3.2 Static Objects

The goals for improving existing methods for visualizing static 3D models with complicated structures are as follows. Complex voxel scenes should be visualized faster than conventional splatting methods and conventional triangle based rasterization. The memory consumption should be lower than the scene's equivalent triangle mesh, and lower than related voxel based raycasting methods.

Therefore, highest rendering speed, and lowest memory consumption for visualizing detailed voxel scenes should be achieved.

### 1.3.3 Skeletal Animated Objects

The goals for improving existing skeletal animation methods are as follows.

#### 1.3.3.1 Skeletal Animation

Collapsing geometries caused by large bend angles of articulated objects, which could be observed in matrix skinning (SSD), should be prevented. Faster computation, artifact-free deformations and more parameters to adjust the deformation compared to quaternion skinning (QS) and dual QS (DQS) should be achieved. The number of joints for complex skeletons should be reduced, where fast rendering speed is preserved, and the artifacts are prevented.

#### 1.3.3.2 Pose Dependent Customization

Different from the previously mentioned pose-space-deformation, two kinds of “re-usability”, which is impossible for existing methods, should be achieved: simple and abstract design of deformation styles for re-usable deformation behaviors such as muscles bulges or cloth wrinkles, and applicability to any number of target characters instantly.

## 1.4 Approach

### 1.4.1 Terrain

A nested Clip-Box approach is proposed to achieve the goals described in Section 1.3.1 as follows. Clip-Boxes allow distant terrain geometry to be visualized with low detail, while geometry close to the view-point is visualized with high detail. A Clip-Box consists of a cubic regular grid of voxels and its corresponding triangulation. For the visualization, multiple nested Clip-Boxes are centered about the viewer. To preserve the placement about the viewer over time, their position,

their voxel data and their triangulation are updated frequently according to the viewer's position changes, concurrently to the visualization.

Since only a small and well-defined region of the entire terrain data around the viewer is required for the visualization, the entire terrain size can be arbitrary size. It is, therefore, possible to visualize infinite sized procedural terrains. The procedural method does not need to compute the entire terrain data. Computing the terrain data within the Clip-Boxes, which surround the viewer, is sufficient. Namely, nested Clip-Boxes allow infinite sized terrains, because the terrain size is independent of the small, constant amount of terrain data contained by the Clip-Boxes that is required for the visualization. Finally, nested Clip-Boxes allow the visualization of arbitrary sized procedural volumetric terrain data, as the procedural data can be computed concurrent to the visualization for the required Clip-Boxes, along with the required Clip-Box updates, where "Procedural" here means "computed by evaluating numerical functions". Therefore, variations of the terrain can further be computed instantly.

Concurrent updates along with the visualization are achieved with a two threaded approach: one thread creates the geometry (geometry thread) and the other thread renders the scene on the screen (visualization thread). The geometry thread repeatedly updates the geometry of all clip-boxes. It therefore loops over all Clip-Boxes and serially creates the procedural voxel terrain data, converts the voxel data into triangles, and smoothes the triangle data. The visualization thread repeatedly loops over all Clip-Boxes and visualizes the latest triangle data of each.

Nested Clip-Boxes are pre-computation free, because their volumetric terrain data is converted into polygons immediately concurrent to the visualization.

### 1.4.2 Static Objects

This thesis proposes a parallel voxel based raycasting approach for visualizing run-length-encoded (RLE) voxel data sets. The proposed method achieves the visualization by raycasting the scene in vertical planes that are perpendicular to the ground plane. For each plane, only one ray is casted into the RLE structure, where the result of each vertical plane is stored in a temporary 2D image buffer as single column. The temporary buffer is then mapped to the screen for achieving the final visualization. In addition to only casting one ray per column on the screen, efficient visibility culling by an extended floating horizon algorithm together with early ray termination are the main properties to provide a high speed.

Due to visibility culling and early ray termination it is possible to visualize voxel scenes faster than by using triangle based rasterization or basic splatting.

Due to RLE data compression, lower memory consumptions per element than triangle based rasterization, triangle based raycasting and related methods for voxel based raycasting methods are achieved.

The final rendered result is post-processed by a novel image filter that smoothes edges of large voxels close to the camera. The filter computes smooth and precise opaque edge-preserved results based on the data in the depth buffer and achieves a higher rendering quality than existing voxel



based raycasting methods. The accurate visualization per-pixel due to raycasting provides more precise results than conventional splatting. Splats are only an approximation of the space they occupy, while voxels are well defined cubes.

### 1.4.3 Skeletal Animation

This thesis proposes a spline skinning based approach combined with deformation style.

#### 1.4.3.1 Spline Skinning

The proposed spline skinning is a combination of spline aligned deformations and conventional SSD. While SSD uses vertex weights to blend simple matrices, spline skinning uses them to blend multiple splines curves. The deformation for a certain point of the spline is achieved by using the spline's Frenet frame.

As spline aligned deformations do not expose artifacts common to SSD, QS and DQS, deformation artifacts are avoided by spline skinning, where SSD is the most common method for animation in current 3D engines.

Furthermore, splines can help to simplify complex skeletal animations, such as a spine or facial animations, by replacing multiple common joints by one spline.

In addition, computations per vertex can almost be reduced to those of SSD by storing a fixed number of samples per spline-curve in a temporary buffer prior to the main deformation per vertex on a per frame basis. This buffer containing the spline samples is then used for computing the deformation. Spline skinning can therefore be computed significantly faster than QS and DQS, which is used in modern 3D engines, such as the CryEngine3 and Unreal Engine 4.

#### 1.4.3.2 Deformation Styles

To achieve re-usable deformation styles, pose-dependent scaling is applied per spline as post process to the spline skinning method. To achieve flexible scaling, deformations are defined in an abstract manner by three scale textures and three scale curves. Once defined, these deformation styles allow the creation of muscles and other custom deformations that can be applied to any number of characters simultaneously, because the deformation style is defined independent from the geometry. Even self-intersections can be prevented by proper modeling of the scale functions, because the scale functions can adjust the bulging of the deformed geometry depending on the bend angle.

## 1.5 Organization

This thesis is organized as follows, where Chapters 3 to 5 are illustrated in **Fig. 1.2**, together with their publications.

Chapter 2 explains 3D engines and game engines. A brief history of 3D engines and game engines, reviews of existing engines, and explanations of their components as well as a comparison of their features are described.

Chapter 3 proposes large scale polygon-based volumetric terrain generation and visualization. After a survey of related work, and the the proposed method are stated, the experimental results are presented together with discussions.

Chapter 4 proposes a visualization of static objects by using high resolution voxel volumes. After a survey of related work and the proposed method are stated, the experimental results are presented with discussions.

Chapter 5 proposes skeletal animation and deformation styles. After a review of related work and the proposed combined method are stated, the achieved experimental results are presented with discussions.

Chapter 6 concludes this thesis. In addition, future work is described.



## Chapter 2. 3D Engine

### 2.1 History

In history, the first 3D engines were used to operate with simple wire-frame models, such as the one used in the game Flight Simulator FS1 in 1980<sup>31</sup>. At that time, 3D engine was not a common term yet. The next development was the visualization of filled polygons, as in the game Rescue on Fractalus<sup>32</sup>.

The first 3D engines that supported perspective correct texture mapping<sup>33</sup> in software at interactive frame-rates were the 3D engines of Descent<sup>34</sup> and the ID Tech1 engine<sup>35</sup>. ID Tech 1 was used for the game Quake.

Then, with the introduction of 3D hardware acceleration in the years 1996-1998 by mostly 3DFX voodoo graphics cards, both technologies were supported for the first time, where popular engines were the Unreal 1 engine<sup>36</sup> and the ID Tech 1 & 2 engine. Hardware based rendering is obviously faster, but since most users at that time did not have an additional hardware accelerator card, software based rendering was still supported for compatibility reasons. Novel features at that time were transparent water layers, reflections on the floor, shadows, lens flare effects maps, and spherical volumetric fog by Unreal Engine.

The following generation, in the years 1998-1999, step by step abandoned software rasterization, and only hardware accelerated rendering remained. At that time, the novelties of the ID Tech 3<sup>37</sup> engine were tessellated nurbs surfaces as well as environment-mapped materials.

A major step towards having large game-worlds was video-game Elder Scrolls III Morrowind<sup>38</sup> based on the Gamebryo Engine<sup>39</sup> in the year 2002. It was one of the first games that provided large 3D outdoor game environments.

The following generation of engines, such as the CryEngine 1<sup>40</sup> in 2004, was able for the first time to visualize large outdoor height-map-based terrains populated with thousands of plants. First, it was used in the videogame FarCry<sup>41</sup>. Additional features included shadow mapping [1], high

---

<sup>31</sup> **SubLogic Corporation**. subLOGIC Flight Simulator. 1980.

<sup>32</sup> **LucasFilm Games**. Rescue on Fractalus. 1984.

<sup>33</sup> Perspective correct texture mapping: Unlike affine texture mapping which is fast to compute but shows discontinuities, perspective correct texture mapping does not expose discontinuities, but is slower to compute.

<sup>34</sup> **Parallax Software and Interplay**. Descent. 1994. <http://www.interplay.com/>

<sup>35</sup> **ID Software**. ID Tech 1. 1993.

<sup>36</sup> **EPIC MEGAGAMES**. Unreal Game Engine. <http://www.unrealengine.com/>.

<sup>37</sup> **ID Software**. ID Tech 3. 1993.

<sup>38</sup> **Ubisoft**. The Elder Scrolls III: Morrowind. Bethesda Game Studios, 2002. <http://morrowind.de.ubi.com/>.

<sup>39</sup> **Gamebase USA & Gamebase Co., Ltd.** Gamebryo Engine. 1997. <http://www.gamebryo.com/>

<sup>40</sup> **CryTek**. CryENGINE 1. 2006. <http://www.crytek.com/cryengine/cryengine1/overview>

<sup>41</sup> **CryTek**. FarCry. 2004. <http://www.crytek.com/games/far-cry/overview>

quality water rendering, bump mapping and efficient use of level of detail. The ID Tech 4<sup>42</sup> engine that appeared in the same year for the first time used shadow volumes [2]. Shadow volumes provide a better quality but are more complex and use more memory bandwidth.

Remarkable features of the CryEngine 2<sup>43</sup> in the following generation in 2007, were volumetric voxel-based terrains, screen-space ambient occlusion [21], parallax occlusion mapping [3], light beams and light shafts, motion blur, depth of field, high dynamic range lighting, subsurface scattering and ambient illumination based on real time ambient maps similar to [22] .

In 2009, CryEngine 3<sup>44</sup> was released. Its new features include approximate global illumination using light propagation volumes, irradiance volumes to give color to reflected light, particles that can receive shadows, hardware tessellation, local approximated ray traced reflections, deferred lighting and 3D water. Another novel technology, called mega-texturing [23], was introduced by the ID Tech 5 engine [23].

The feature of the latest engines presented in 2012, such as the Unreal 4 Engine<sup>45</sup>, is voxel cone tracing for pre-computation free global illumination [9].

## 2.2 3D engine in Game Engine

### 2.2.1 Game Engine

#### 2.2.1.1 Overview

The software system of a game, which sits on top of the 3D engine, is called a *game engine*, which includes a 2D, 2.5D or 3D engine. A game engine consists of multiple components, which are overviewed in **Fig. 2.1**. The main purposes of a game engine are as follows:

- Re-usability: A game engine allows speeding up the development of a game significantly. It provides the major functionality required for most of the games. Therefore, the development cost can be minimized.
- Portability: Often, game engines support multiple platforms, such as PC, consoles and mobile devices. Therefore, by using a game engine, the developed game becomes available on multiple platforms at the same time.

The core components are game logic, graphics engine, mass storage access ((I/O), sound, graphical user interface (GUI) and collision detection.

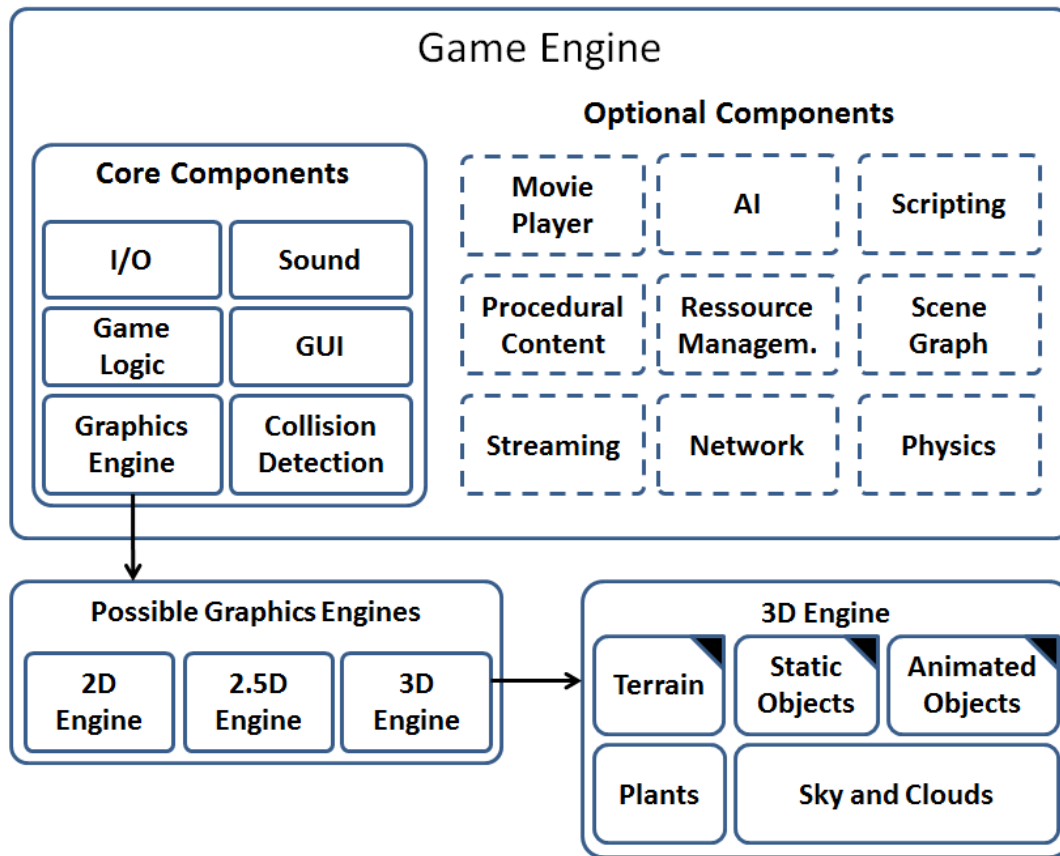
---

<sup>42</sup> ID Software. ID Tech 4. 1993.

<sup>43</sup> CryTek. CryENGINE 2. 2007. <http://www.crytek.com/cryengine/cryengine2/overview>

<sup>44</sup> CryTek. CryENGINE 3. 2009. <http://www.crytek.com/cryengine/cryengine3/overview>

<sup>45</sup> EPIC MEGAGAMES. Unreal Game Engine. <http://www.unrealengine.com/>



**Figure 2.1** Overview of Game Engine and 3D Engine: black triangle: modules explored by this thesis.

Additional components are resource management, physics, network, scripting, scene graph, artificial intelligence (AI), streaming, procedural content creation and a movie player.

### 2.2.1.2 Core components

#### Graphics Engine

This part is responsible to visualize the game graphics. It is the main focus of this thesis.

#### In-Out (I/O)

The I/O component is required to allow the data access to mass storage devices such as the hard-disk or DVD ROM as well as LAN and WAN.

#### Sound

Sound is required for sound effects, background music and narration.

#### Game Logic

Game logic is responsible for connecting and managing the other game modules, which means controlling the components and exchanging the information between them.

#### Graphical User Interface (GUI)

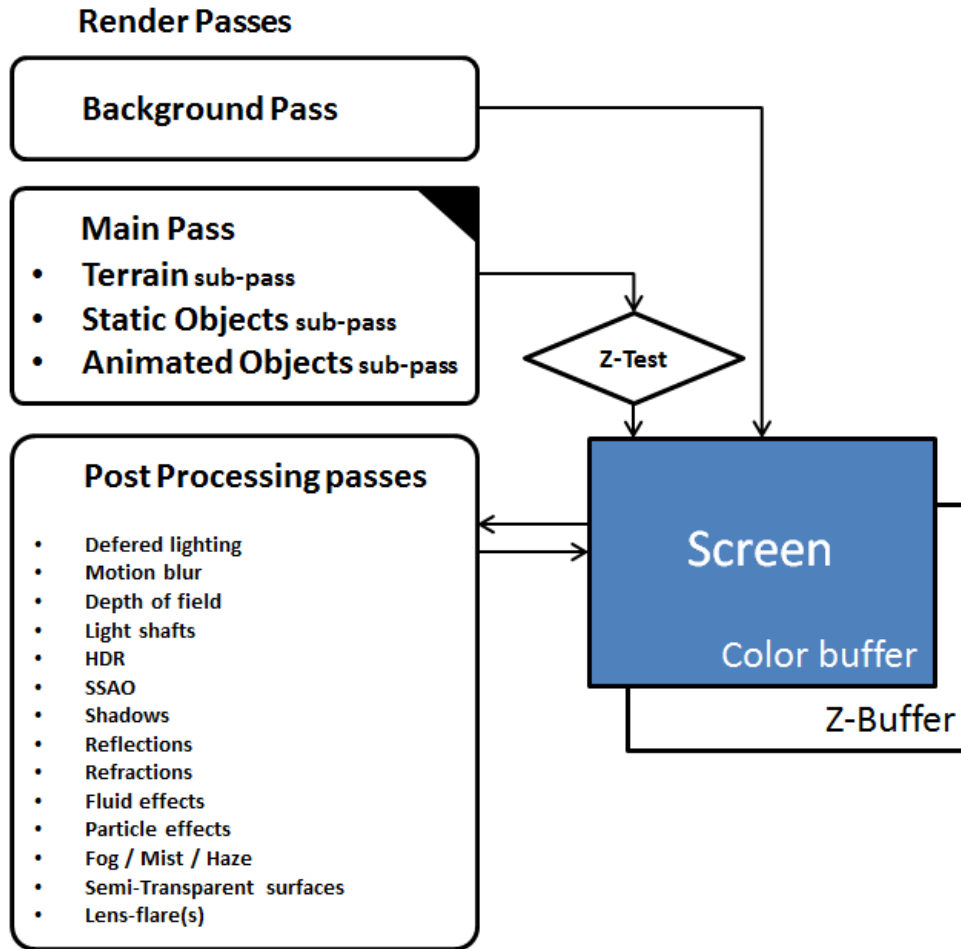
The GUI is required for all types of in-game menus.

#### Collision Detection

Collision detection is required for the interaction between players, the environment and non-player characters (NPCs).

## **2.3 3D Engine Structure and Functions**

A 3D engine solves the visualization task of a game or game engine. A 3D engine uses 3D geometry data as input and visualizes the scene according to changes in the view-point and global camera parameters in real-time. The parameters are given by the game engine and usually depend on user input.



**Figure 2.2** General 3D engine Overview: This diagram shows the render flow of a common 3D engine including additional post processing modules in a simplified manner. The black triangle marks modules correspond to this thesis' chapters 3 to 5.

A simplified visualization flow of a typical 3D engine is shown in **Fig. 2.2**. Multiple passes are required to achieve the final result. The final result is hereby the color buffer, which is displayed on the screen. The attached Z-buffer or depth-buffer is used to handle occlusions of opaque surfaces.

The rendering passes are categorized into three general passes: background pass, main pass and post-processing passes. Terrain, static geometry and animated geometry are included in sub-passes of the main pass. Each pass corresponds to one module. As identical camera parameters are used for each pass, all modules share the same view-point, same view angle and camera orientation, same focal length, same field of view and the same rendering resolution.

The function of each pass is described in the following.



### 2.3.1 Background Pass

The background pass sets the background and initializes the depth-buffer. The types of backgrounds range from single colored backgrounds over using sky-boxes<sup>46</sup> up to simulations of the atmosphere including day/night cycles, cloud rendering, light scattering and weather conditions. An example for complex background rendering is the third-party software True Sky<sup>47</sup>. Such complex backgrounds that include volumetric lighting effects are also involved in the post-processing pass.

### 2.3.2 Main Pass

The main pass visualizes the entire 3D foreground geometry. This thesis' main pass consists of three sub-passes: terrain, static objects and animated objects. The result of each sub-pass is merged with the content of the color buffer and depth buffer.

### 2.3.3 Post-Processing Pass

The post-processing pass is responsible for mostly shading and camera effects. Options of the post-processing pass are listed as follows (**Fig. 2.2**):

- Deferred lighting
- Motion blur
- Depth of Field (DoF)
- Light shafts [24]
- High Dynamic Range lighting (HDR)
- Screen Space Ambient Occlusions (SSAO)
- Shadows
- Reflections
- Refractions
- Fluid effects
- Particle effects
- Fog / Mist / Haze
- Semi-transparent surfaces

---

<sup>46</sup> Valve Software. SkyBox. [https://developer.valvesoftware.com/wiki/Skybox\\_\(2D\)](https://developer.valvesoftware.com/wiki/Skybox_(2D)).

<sup>47</sup> Simul Software Ltd. Simul Weather SDK. 2009. <http://www.simul.co.uk/>

## 2.4 Visualization by 3D Engine

### 2.4.1 Items to be visualized

Modern 3D engines split the visualization task into the following general types of objects for the visualization.

- Terrain (Chapter 3)
- Static objects (Chapter 4)
- Skeletal animated objects (Chapter 5)
- Plants (included in Chapter 4)
- Sky and clouds (not included in this thesis)
- GUI (not included in this thesis)
- Combining the modules

These object types are detailed as follows.

#### Terrain

The terrain provides the foundation of the virtual world. All virtual characters and buildings are placed on the terrain. Terrain has a unique geometry which is different from characters and buildings. Terrain geometry is of lower resolution; it is more uniform and changes more smoothly in space. Therefore terrain rendering algorithms are different from algorithms focusing on rendering general 3D objects. Terrain is commonly visualized using height-map based methods. However, also mixed methods using height-map data and volume data exist, such as the one used in the CryEngine. This thesis aims at achieving the goals described in Section 1.3.1.

#### Static objects

Static objects include all kinds of static architecture such as buildings, bridges, stones and rocks. They are usually rendered as static textured polygon meshes. To increase the performance, distant objects are often rendered with a reduced amount of polygon count, which is called level of detail (LOD).

However, with the increasing size of game-worlds such as in GTA V<sup>48</sup> in which entire cities including cars and characters are visualized, the amount of static objects is enormous. Therefore, the detail of each object needs to be reduced dramatically.

#### Skeletal animated objects

---

<sup>48</sup> **RockStar Games**. GTA V. 2008. <http://www.rockstargames.com>.

Skeletal animation is used mostly for animating skeleton based models, such as humans. However, skeletal animation can be used for any complex animation and deformation of organic models in general. It is a core component of every video game and therefore 3D engine. A common middleware for rendering characters has not yet been established. However, it is often coupled with the physics engine such as Havok for simulating realistic and physically correct animation behaviors.

### Plants

Plants include trees, grass and flowers. They are highly detailed and often animated by simple swaying motion. While trunks and branches are commonly visualized by solid polygonal objects, leaves are rendered as two-sided polygons without volume. As most 3D engines have the same needs for plants and vegetation, a middleware called *SpeedTree* has evolved in recent years. *SpeedTree* is commonly used by newer 3D engines such as previously mentioned Nebula 3D engine. To make trees more realistic, recent developments use multiple tricks to add realism as described in GPU Gems [24].

This thesis does not focus on visualizing plants in particular, but plants without motion are included in static objects.

### Sky and clouds

Sky and cloud rendering is not researched in this thesis. Both are often rendered using a simple skybox<sup>49</sup>, which is basically a textured cube on which distant mountains, sun and clouds can be painted. Newer technologies simulate day and night cycles, weather as well as volumetric clouds and light scattering through the clouds. An example is the middleware TrueSky.

### GUI

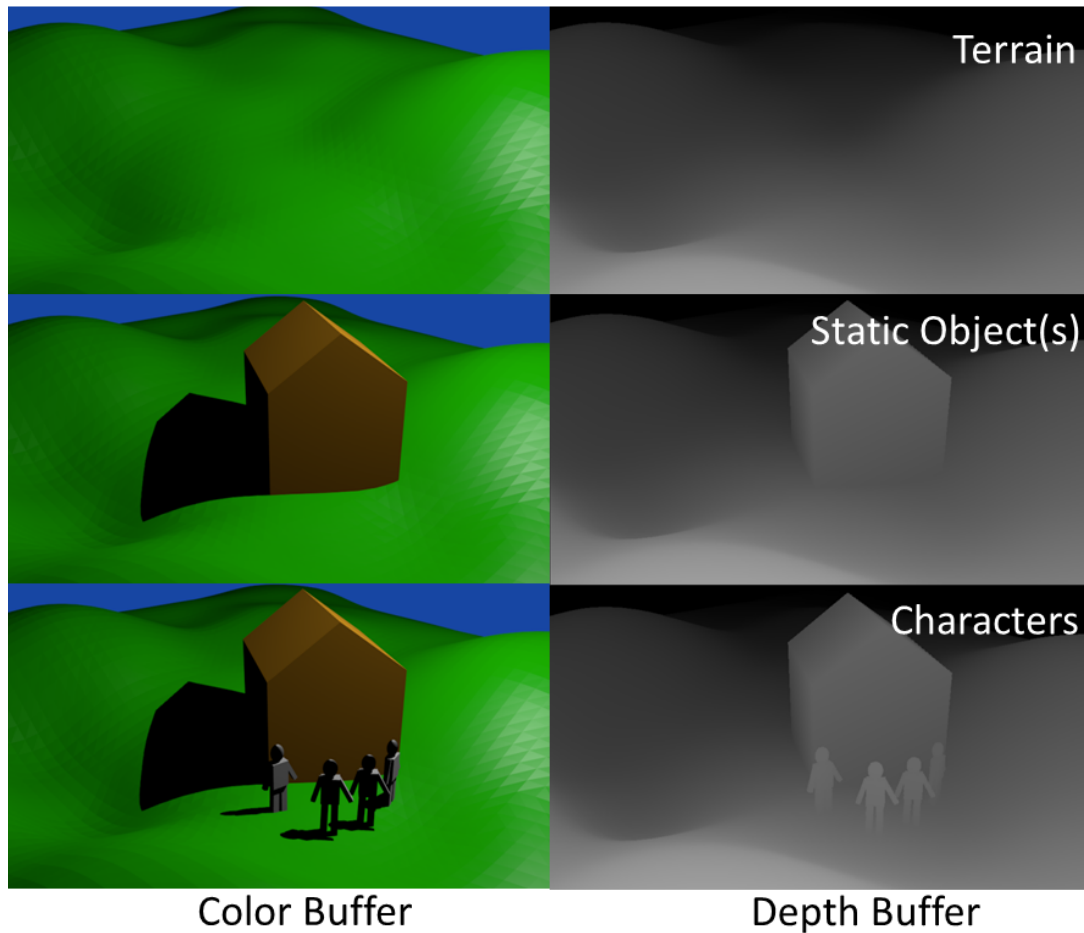
The GUI is used for game menus, but has not yet been standardized. Most games have a custom menu interface.

## **2.4.2 Combining multiple modules**

3D engines combine each module's output in any data format (e.g. polygon, voxel) so as to obtain the final rendering result by utilizing well known depth buffer technology.

---

<sup>49</sup> Software, Valve. SkyBox. [https://developer.valvesoftware.com/wiki/Skybox\\_\(2D\)](https://developer.valvesoftware.com/wiki/Skybox_(2D)).



**Figure 2.3** Main render passes of general 3D engines: upper row: terrain, middle row: static objects, lower row: characters; left column: color buffer, right column: depth buffer (bright: close, dark: far).

The way it works is illustrated in **Fig. 2.3**. The depth buffer stores the depth information for each pixel in the rendered image. It is, therefore, possible to merge the render results of multiple modules by applying per pixel visibility checks using the depth buffer. This technology allows to merge the results from different modules efficiently. The culling is automatically carried out by the graphics hardware when drawing triangles. This technology is supported by the first 3DFX Voodoo graphics hardware accelerator cards.

For visualizing each component shown in **Fig. 2.3**, the same camera parameters are used to achieve a consistent result of the depth-map. The three modules (terrain, static objects and characters) store the depth value sequentially to the depth buffer, one after another, so that the result of merging the terrain, static objects and characters are rendered properly in the color buffer (right column in **Fig. 2.3**) without using advanced synchronization technology.

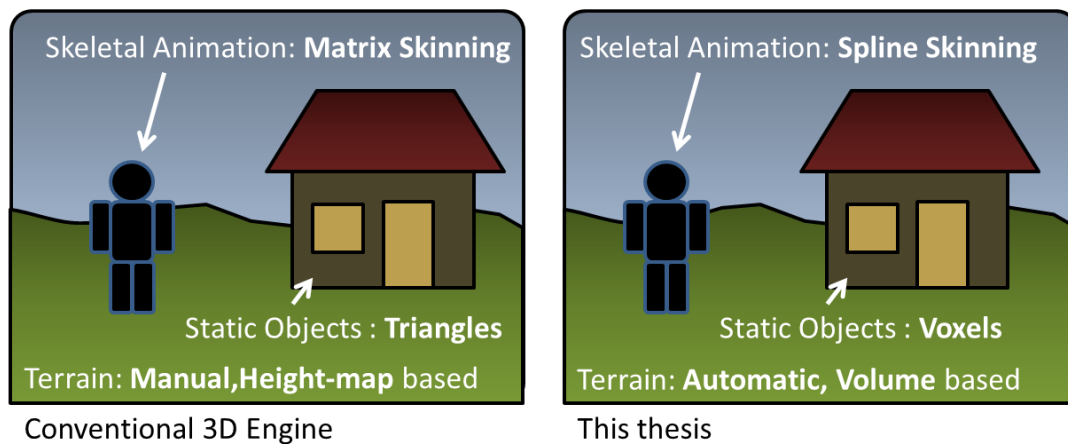
## 2.5 Module Comparison

### 2.5.1 General Comparison

In **Fig. 2.4** the differences between the proposed modules and corresponding modules in conventional engines are illustrated. The major modules for each engine, the proposed and the conventional ones, are completely different from each other as follows:

- While conventional terrain is height-map-based, the proposed is volume-based and automatically generated.
- While conventional static objects are visualized using triangles, this thesis proposes voxel based raycasting.
- While conventional skeletal animation is based on matrix skinning this thesis proposes a spline skinning and deformation styles.

**Table 2.1** shows a more thorough comparison to all of the previously introduced state-of-the-art engines. As shown in the table, the proposed modules provide novel features not present in existing engines that can solve the limitations of the major 3D engines. Each of the modules is detailed in the following subsections.



**Figure 2.4** Illustrated Comparison

**Table 2.1** Module comparison: In the table, “+” indicates that the feature is available and “-“ indicates that the feature is unavailable. For unknown support the fields are left empty.

	Year	Terrain				Static Geometry		
		Heightmap Terrain	Volume Terrain	Runtime- generated Procedural Volume Terrain	Infinite Sized Volumetric Terrain	Triangle rasterization	Raycasting based	Voxel based Ray-casting
Proposed modules		+	+	+	+	+	+	+
CryEngine3	2009	+	+	-	-	+	-	-
Unity3D	2012	+	-	-	-	+	-	-
Infinity Ward4	2009	+	-	-	-	+	-	-
ID Tech5	2010	+	-	-	-	+	-	-
Avalanche Engine	2010	+	-	-	-	+	-	-
Frostbite2	2011	+	-	-	-	+	-	-
Unreal4	2012	+	-	-	-	+	-	-
Anvil Next	2012	+	-	-	-	+	-	-
Unigine	2012	+	+	-	-	+	-	-
Blender	2012	-	-	-	-	+	-	-
Ogre	2012	+	-	-	-	+	-	-

	Year	Skeletal Animation			
		Matrix Skinning	DQS	Spline Skinning	Deformation Styles
Proposed modules		+	-	+	+
CryEngine3	2009	+	+	-	-
Unity3D	2012	+	-	-	-
Infinity Ward4	2009	+	-	-	-
ID Tech5	2010	+	-	-	-
Avalanche Engine	2010	+	-	-	-
Frostbite2	2011	+	-	-	-
Unreal4	2012	+	+	-	-
Anvil Next	2012	+	-	-	-
Unigine	2012	+	+	-	-
Blender	2012	+	-	-	-
Ogre	2012	+	-	-	-

## 2.5.2 Terrain Comparison

Existing engines use manually generated height map or volume terrain. From the 3D engines investigated in **Table 2.1**, CryEngine 3 provides the most advanced terrain technology, which combines voxel based terrain representations with height map based representations. While height maps solve for common terrain, voxels are used for overhangs, caves and arcs.

The terrain in existing engines is often streamed from a mass storage device or from network. This limits the size of the terrain and, as a consequence, the size of the virtual environment, which is based on the local storage device and the time artists spent for its creation.

In the proposed terrain module, the terrain is entirely volume based and created automatically on run-time according to the user-defined parameters. This means that caved terrains with overhangs

and arches can be generated at large size without much effort. Furthermore, modifying parameters allows design variations to be applied easily for generating new and interesting terrains. The size of the terrain that can be generated is only limited by the numerical precision, not by the size of the storage device.

### 2.5.3 Static Objects Comparison

Existing 3D engines use triangles to represent static geometry, as can be seen by the comparison in **Table 2.1**. Important to display many objects with triangles is the use of level-of-detail. Here, CryEngine 3 is the most advanced engine so far. It is able to display a large amount of objects by rendering distant objects with a fewer triangles than objects close to the camera.

Triangle based rasterization works well to a certain degree. However, once the number of objects increases significantly, ray casting could outperform rasterization, because raycasting can handle occlusions very well, while conventional rasterization could lead to heavy overdraw. More precisely, the complexity of rasterization is linear to the number of rendered triangles  $T$ :  $O(T)$ .

The proposed method by this thesis is based on voxel raycasting. Voxel based raycasting scales logarithmic to the number of voxels  $V$  in the scene for raycasting:  $O(\log(V))$ . Another significant issue is the fact that voxel-based objects can store high, evenly distributed details more compactly compared to polygon-based objects. The reason is that voxels combine position and material information. Triangle-based rasterization uses textures, which are addressed with texture coordinates for each triangle. Furthermore, vertex coordinates need to be stored along with the triangles as well, which both gets significant for detailed geometry.

### 2.5.4 Skeletal Animation Comparison

Conventional 3D engines used matrix skinning for performing the skeletal animation. Matrix skinning is fast and simple – however - it exhibits significant deformation artifacts for strong bending operations. To overcome these limitations, dual quaternion skinning is used by the latest 3D engines, such as CryEngine 3 (**Table 2.1**).

Compared to matrix skinning, also the proposed spline skinning algorithm can solve the deformation limitation, as spline based deformations are free from artifacts. While the dual quaternion skinning can solve for collapsing geometry in strongly bent regions, it does not provide as many deformation parameters and same speed as spline skinning. Depending on the purpose, one spline can cover multiple common joints at once and therefore simplify the skeleton significantly.

In addition to spline skinning, a deformation styles method is proposed. It allows to model abstract deformation styles like muscle, metal, or cloth-like deformations. These abstract designs can be applied immediately to all characters at once. The deformation styles method can be attached to the spline skinning method as a post-process.

## 2.6 Conclusion

The proposed three modules for procedural volumetric terrain rendering, rendering of large complex static objects and for skinned skeletal animation with deformation styles can solve existing limitations and provide additional features. As a result of different comparisons, it turns out that the CryEngine3 is the most advanced game engine at this point. This thesis assumes that the CryEngine 3 is the 3D engine, in which the three modules explored in Chapter 3 to Chapter 5 are embedded.



## Chapter 3. Terrain

### 3.1 Goals

This chapter proposes a visualization algorithm that can overcome the limitations of existing methods. The goals are summarized as follows.

- Unlimited Terrain Size: The proposed method should be able to handle arbitrary terrain sizes, without the need to store any data on mass media devices.
- Pre-computation Free: Different from existing methods, pre-computations of the entire terrain data should not be required, as this would not allow unlimited sized terrains. In addition, the terrain data and geometry should immediately be generated.
- Generation of Procedural Volumetric Terrain Data on the Fly: The proposed method should allow the generation of terrain data on the fly. The data should not be stored on mass storage devices etc. The data should be synthesized on run-time, in parallel to the visualization. This saves time for the artist and allows quick changes between multiple terrains.

### 3.2 Related Work

#### 3.2.1 Procedural Terrain Generation

In games, procedural terrain generation has already been used. An example of this is the successful video game “The Elder Scrolls II: Daggerfall”, by Bethesda Software. A massive sized terrain (a flat map, no height information) was one of the main elements of this game. They did not use procedural volumetric terrain for their approach; that is why this is only partially related work.

In academia, procedural terrains can be found as well. Prusinkiewicz developed a method for creating fractal height-map based terrains [25]. More advanced method was developed by Peytavié *et al* [14]. They proposed an algorithm to automatically generate large volumetric terrains with caves and overhangs. Their first one did not solve for volumetric terrains. Their second method uses volume data for creating the terrain, but their method does not generate the terrain in parallel to the visualization, and it does not visualize the terrain in real-time. Therefore, it cannot achieve the visualization of large volumetric terrains in real-time.

In other areas, non-game and non-academic, procedural terrain generation was developed as well. Terragen<sup>50</sup> allows the generation of arbitrary height-map based terrains. In Pandromeda<sup>51</sup>, height-map based terrains as well as volumetric terrains can be generated. In both Terragen and Pandromeda a user can freely choose a terrain function. Both methods focus on rendering the terrain offline. They cannot visualize volumetric terrain in real-time.

A method that generates a volumetric terrain for the visualization in real-time is the NVidia Cascades Demo [13], whose terrain function is fixed to Perlin Noise<sup>52</sup>. They create the terrain in a pre-processing step. They cannot generate procedural terrains in parallel to the visualization. Therefore, they cannot visualize terrains that require more memory than physically available.

*Summary: All related methods create the terrain as an offline process, even though they support terrain visualization in real-time as in [13]. However, at present there is no algorithm that can achieve the dynamic, on-the-fly generation of procedural volume data in parallel to the visualization process.*

### 3.2.2 Polygonal Visualization of Volumetric Terrains

Since the proposed algorithm visualizes the terrain volume data as polygonal mesh, methods that visualize large and detailed objects which consist of either polygonal mesh data or opaque volume data are also reviewed.

One published algorithm<sup>53</sup> represents the terrain by a  $512 \times 512 \times 64$  voxel grid and visualizes by using multi-resolution ray casting. They focus only on visualizing height-map based terrain and do not show any examples for volume based terrains. Furthermore, they pre-process the entire terrain data and store it in a special format prior to the visualization. They do not support updates on run-time. Therefore, they cannot visualize arbitrary sized volumetric terrains in real-time.

Another related method is the visualization of large iso-surfaces from volume data. An iso surface represents the boundary surface inside a volume data between values less than the iso value and the ones greater than the iso value. Commonly, the iso value is a user defined constant. Gregorski *et al* [26] present a method that recursively subdivides the scene into diamonds based on pre-calculated error-values to visualize large iso surfaces. The method is basically a three-dimensional extension of the height-map based terrain rendering method that is known as ROAM [11]; that method converts the input data into a special format in a pre-processing step. Their method requires intensive pre-computation and therefore cannot solve for updates in real-time.

For visualizing large meshes, several methods have been invented. Most of them, such as [7] and [27], cluster the input mesh in multi-resolution shapes, such as cuboids or tetrahedrons. These have to be created in a pre-computation step for the dynamic assembly at run-time. The approach presented by Lindstrom [28] is similar. His method clusters vertices in a hierarchical fashion to

<sup>50</sup> Fairclough, Matt. Terragen. 2000. <http://www.terradreams.de>.

<sup>51</sup> Pandromeda. <http://www.pandromeda.com>. 2012.

<sup>52</sup> Perlin, Ken. Perlin Noise [http://en.wikipedia.org/wiki/Perlin\\_noise](http://en.wikipedia.org/wiki/Perlin_noise). 2012.

<sup>53</sup> Visualization Lab, Center for Visual Computing, SUNY Stony Brook. Voxel-Based Flight Simulation <http://www.cs.sunysb.edu/~vislab/projects/flight/>. 1997.

achieve the view-dependent LOD. His method requires intensive pre-computation and therefore cannot solve for updates in real-time.

Other related approaches propose the usage of point sprites, also known as splats, for representing the scene [8], [5]. In [8], a combination of splats and polygons is used, where the polygons solve the geometry near the viewpoint, and splats are used for distant geometry. Their method requires intensive pre-computation and therefore cannot solve for updates in real-time.

A method that utilizes an LOD structure, which is similar to the proposed method, is called GoLD [29], where the mesh resolution is continuously reduced according to distance by switching among several pre-computed detail levels of the initial mesh. The LOD's are computed by vertex removal in order to enable a smooth transition by geo-morphing. Their method requires intensive pre-computation and does not support dynamic generated terrain data.

Video games such as the CryEngine 3 do not reveal their method used for visualizing volumetric terrains. However, a known limitation of their method is that it does not allow the entire large terrain to be represented as voxel landscape. They use local, manually designed voxel boxes to visualize overhangs and caves. The height map based terrain is marked out in those areas for avoiding interferences.

As for the related work in general, including CryEngine 3 and the methods [5] [7] [8] [27] [28] [29], none of them suits for visualizing large, on-the-fly generated volumetric terrain data. All of the aforementioned approaches require intensive preprocessing of the full data set prior to visualization, and they also have to store the complete terrain data to be visualized. Besides the large amount of resources necessary during preprocessing of polygon or volume data as in [8] to create the run-time structure, it is clear that the amount of data generated obviates the application of large walk-through ranges.

### **3.3 Proposed Method**

To solve the limitations of existing algorithms a method based on nested Clip-Boxes is proposed. Nested Clip-Boxes are an evolution of nested geometry clip-maps, which are used for height-map based terrains. Rather than nesting 2D geometry maps, multiple 3D Clip-Boxes are nested in a concentric manner about the viewer. To preserve the concentric nesting while the viewer moves, frequent Clip-Box updates are performed in parallel to the visualization. A Clip-Box consists of a cubic regular grid of voxels and the corresponding triangulation. For performing a Clip-Box update, the following two general steps are performed. First, the procedural terrain volume data for this particular Clip-Box is computed. Second, the volume data is converted into polygons for the visualization. No pre-computed data is required for these two steps. The proposed Clip-Box based terrain visualization can, therefore, achieve the pre-computation free visualization of volumetric terrain data.

Furthermore, only the data present in the Clip-Boxes is required for the visualization; the size of the entire terrain data is independent of the constant amount of data required for the visualization. The proposed method can, therefore, achieve the visualization of arbitrary sized terrains. This

overcomes the limitation of related methods to visualize terrains that do not fit into the available physical memory.

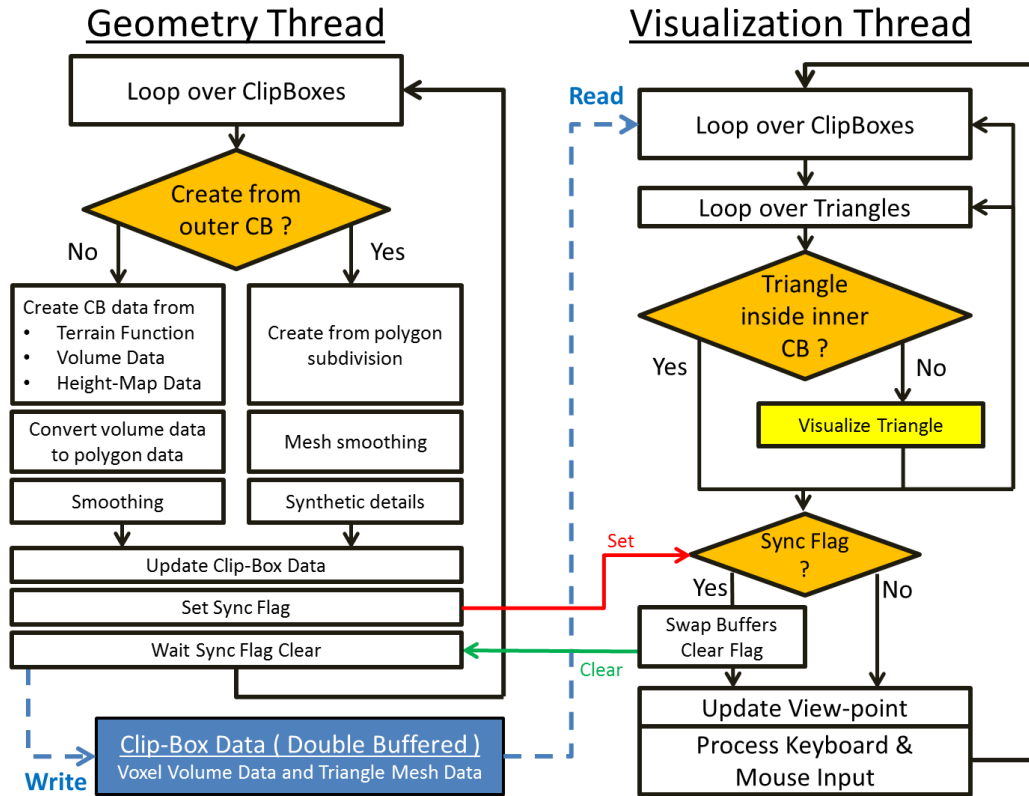
Nested Clip-Boxes furthermore provide the ability to generate procedural volume data in parallel to the visualization. This allows the visualization of synthetic terrains of arbitrary size.

Similar to the proposed approach, other researchers also use functions to generate the terrain (cf. Pandromeda, [25] and [13]).

The proposed method only requires the terrain functions and their parameters for generating the underlying volumetric data. Storing the entire volumetric data generated from these functions is not necessary. Since any arbitrarily explicit function can be chosen for data generation, the walk-through range and the number of levels are limited only by the parametric range of the function. Due to the possible large range of variations, there is a rich number of distinct concavities, overhangs, and other interesting structures that can be generated in run-time. Note, that as procedural creation of volumetric terrains is already addressed by various methods such as, Pandromeda, [25], [13], [14], the main focus of this chapter is on generating the visualized terrain on-the-fly, without relying on any pre-processed data. Different from the proposed method the above mentioned related works, Pandromeda, [25], [13], [14] are not able to create and update the terrain data in parallel to the real time visualization.

### 3.3.1 Overview

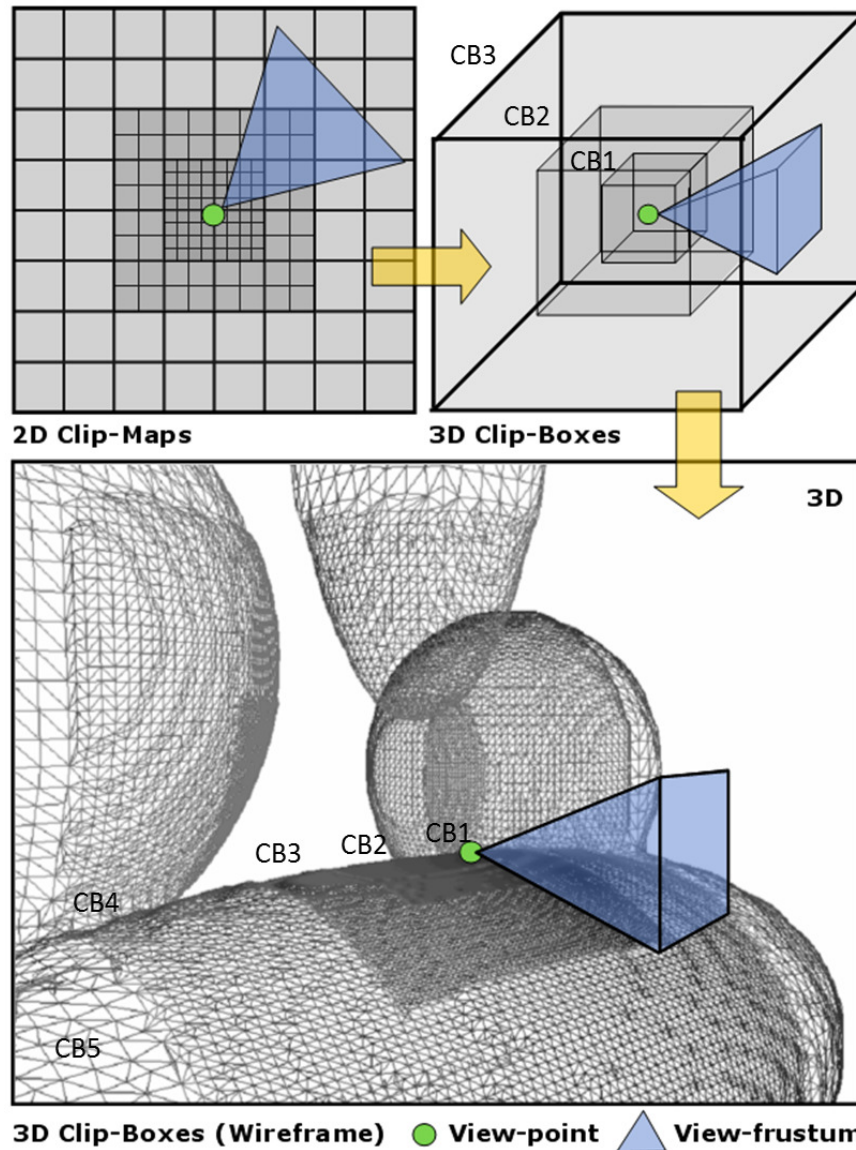
The proposed landscape visualization method consists of terrain synthesis and visualization modules. The synthesis module defines the terrain as three dimensional function defined by the user. The user can adjust the parameters for the terrain generation and for controlling the appearance. The visualization system consists of two threads that run in parallel, as shown in **Fig. 3.1**: A geometry thread and a visualization thread. The geometry thread calculates the procedural terrain and converts it into triangles for the visualization. The computed data is stored in a buffer that can be accessed by the visualization thread later on. Mutual exclusion is achieved by using double buffering (two buffers) for each Clip-Box. The visualization thread then visualizes the data in real-time as triangle mesh.



**Figure 3.1** Overview of the terrain visualization module: Using two threads helps to optimally distribute the rendering and voxel to polygon conversion tasks on modern multi-core-CPU's.

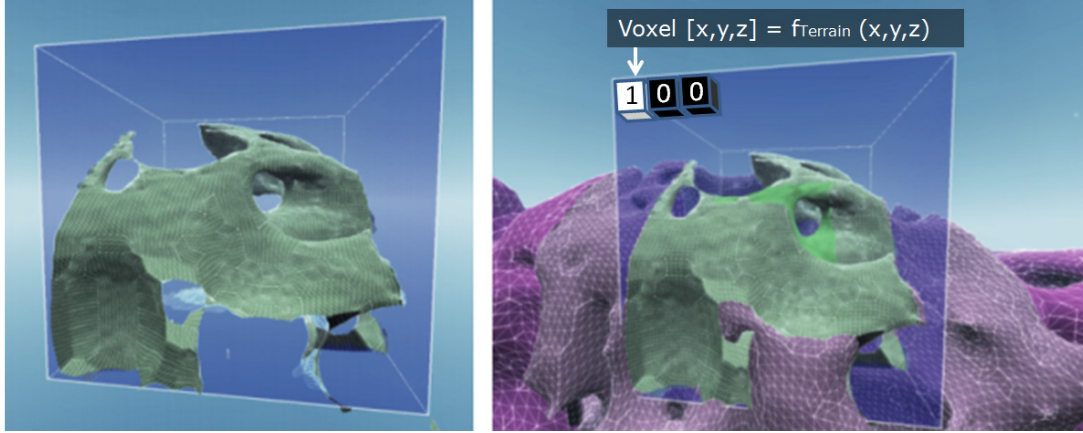
For the visualization module, the CB volume data is created from sampling the procedural terrain function for each voxel inside the CB. For the hardware accelerated visualization on the GPU, the volume data is converted into triangle data. The conversion from volume data to triangles is very similar to visualizing iso-surfaces and can be solved by using one of the conventional algorithms such as marching cubes [30]. However, as the amount of triangles arising from direct volume data to polygon conversion is immense, an efficient level-of-detail (LOD) approach needs to be employed to the proposed system. This is necessary to keep the polygon-count reasonably low for today's graphics hardware.

Nested geometry clip-maps, which derive from clip-maps [31], provide all of required features for the two-dimensional height-map based case. However, they cannot solve the three-dimensional volume-data based case.



**Figure 3.2** Evolution from Clip-Map to Clip-Box (CB); Top left: Nested geometry clip-maps [12] Top right: the Clip-Box based approach as sketch; lower: and the final result as a wire-frame.

Hence, extending the clip-map based terrain visualization approach of Lossaso et.al [12] on geometry clip-maps to the third dimension by introducing nested Clip-Boxes, as shown in **Fig. 3.2**, can solve existing limitations. Clip-Boxes have very similar properties to clip-maps, but are more complex. **Figure 3.3** shows an example of a single Clip-Box (CB). The voxels inside the CB are computed using a terrain function  $f_{\text{Terrain}}$ . In contrast to clip-maps, where nested regular grids suffice to represent the geometry (**Fig. 3.2**), CB's carry complex, rapidly changing mesh-topologies. While each geometry clip-map is represented as a rectangular portion of the landscape's height-map, each Clip-Box represents the iso-surface of a cubic portion of the terrain volume data.



**Figure 3.3** Clip-Box: Left: the pure Clip-Box geometry; right: CB embedded into the landscape.

The proposed algorithm visualizes the terrain using a two-threaded approach, which is shown as a diagram in **Fig. 3.1**. The Geometry Thread with a low update rate creates the geometry from procedural volume data (“Procedural Data”), existing volume data (“Volume Data”) or height-map data, which is converted into volume data (“Clip-Box Data”). Next, the volume data is converted it into polygons (“Convert volume data to polygon data”). For the procedural creation, a user defined mathematical terrain function is evaluated for each voxel of the volume data in  $x,y,z$ . The result is either 0 (voxel not set) or 1 (voxel set). To smooth the created geometry, a smoothing step is finally applied (“smoothing”). After the geometry creation is completed, the created geometry is stored to the one buffer of the Clip Box data that is currently not used for visualization. To communicate with the “Visualization Thread”, a “Sync Flag” is set. The visualization thread checks this flag for each frame to be visualized and updates its reference to the corresponding buffer. After that, the flag is cleared and the geometry thread can continue to update the next CB.

The visualization thread continuously displays the polygons on the screen with a high update rate.

For the procedural terrain function, which computes the landscape volume-data to be used by the nested-Clip-Box algorithm, a relatively simple function that produces landscapes complex enough to prove the efficiency of the proposed method. Since the formula for the terrain generation can be defined by the user, this thesis does not focus on inventing a novel formula.

Section 3.5.3 presents some examples of functions, which are used for the experiments.

### 3.3.2 Differences between the proposed method and Previous Work

The proposed method is basically and extended and improved version of the original version by Lossaso et.al [12] on geometry clip-maps. The extensions and improvements are summarized as follows, where the following summary is detailed in Section 3.4:.

- Different from clip-maps and geometry clip-maps, the proposed method is based on CB’s.

CB's are the three-dimensional extension of geometry clip-maps.

- CB's are significantly more complex, because the topology of their geometry is arbitrary. Different from clip-maps, which are based on height-maps, CB's are based on volume data.
- In case of geometry clip-maps, only regular grids are required. Clip-maps can therefore, re-use the entire geometry, while the geometry of Clip-Boxes needs to be updated according to changes in the view point.
- The proposed method needs to group triangles for efficient rendering and to smooth the mesh generated from volume data to avoid blocky appearances.
- Different from geometry clip-maps, which apply additional procedural details only as a height-map, the proposed method applies them along the normal vector of the generated mesh, which can be any direction.
- Different from geometry clip-maps, which are limited in terms of procedural details for the global height-map data, the proposed method focuses on generating the entire volumetric terrain procedurally.

### 3.4 Clip-Box Algorithm

The proposed nested Clip-Box algorithm utilizes a simple and efficient structure to represent the terrain mesh. Similar to [12], which caches the terrain geometry in a set of nested regular grids, the proposed method caches the geometry in a set of nested Clip-Boxes (**Fig. 3.2**). Once the viewpoint changes, all Clip-Box positions are updated incrementally to preserve the concentric LOD structure.

The algorithm uses the two threads shown in **Fig. 3.1** to handle the Clip Box updates and visualization. The geometry thread converts the input volume data into polygons for each Clip-Box. The input data can be generated from procedural terrain volume data, existing volume data or height-map data, which is converted into volume data. The result is converted into triangle data, which is smoothed to remove its blockiness. The geometry thread then informs the visualization thread that new geometry data is available by setting a flag. If the flag is set, the visualization thread stop its next iteration and update its reference to the new geometry data. The geometry thread processes all Clip Boxes in a loop.

For the visualization thread, it contains a loop over all Clip-Boxes. The loop processes each Clip Box and visualizes the contained triangles. The visualization of the triangles is performed in the inner loop. For each Clip-Box, only triangles that do not lie inside the next inner Clip Box are visualized to achieve the desired level of detail structure.



### 3.4.1 Clip-Box

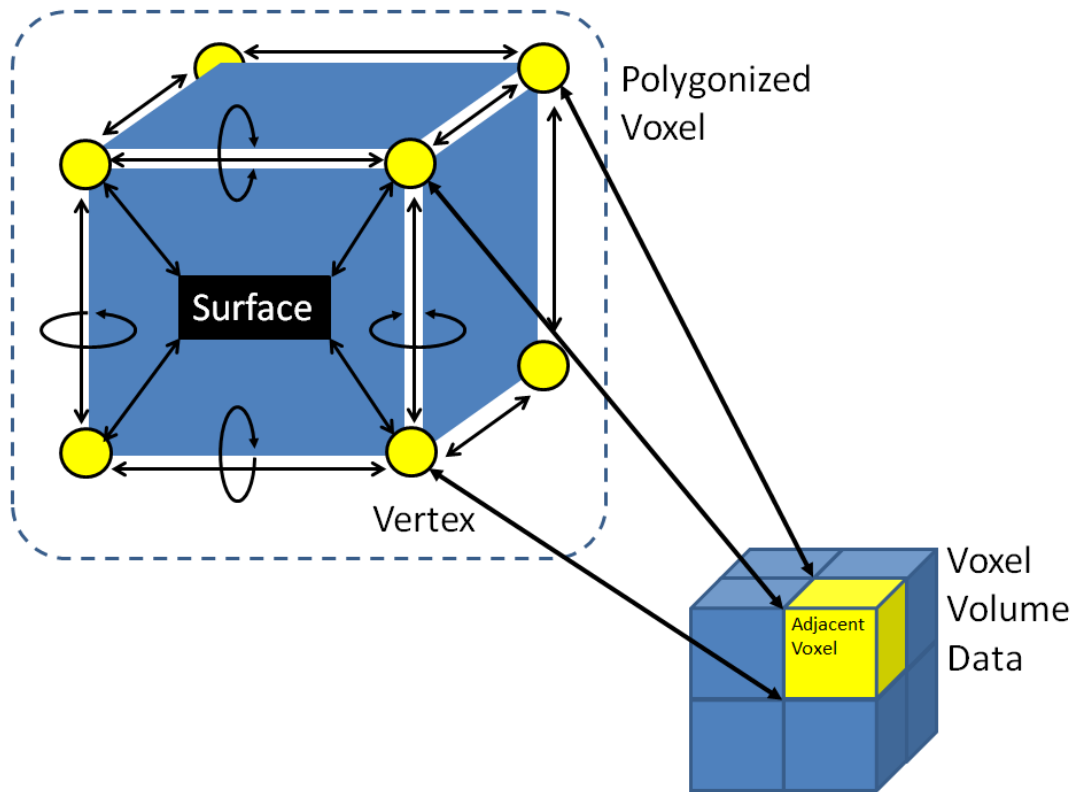
A Clip-Box (CB) is defined as the polygonal conversion of a cubic portion of the entire terrain's volume data. **Figure 3.3** clarifies where a single CB is shown in the left image, where the right image shows the CB embedded into the surrounding landscape. As opposed to clip-maps [12], which preserve simple regular grids with almost constant complexity over time, CBs significantly change their complexity as they are shifted through the volume data.

### 3.4.2 Data Structure

For each CB, 8-bit volume data is stored, where each voxel is either set (opaque) or unset (transparent). The polygon data created from the volume data consists of triangle strips, where each vertex inside the strip carries x- y- and z- coordinates as well as a normal vector. For the conversion, each voxel is considered as a cube with six surfaces and eight shared vertices. Two triangles form each of the six surfaces of a voxel.

In addition to these two structures, adjacency information for each voxel to speed up the voxel-to-polygon conversion process is stored. The links (32-bit pointers) that can be seen in **Fig. 3.4** are utilized as follows:

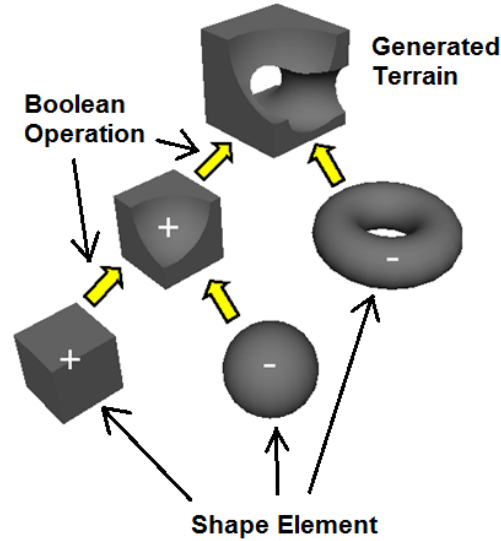
- Voxel to vertex. Required for inserting a new vertex. The link is used to check whether a vertex has already been created for the specific voxel.
- Vertex to vertex. Required for quick smoothing and to link adjacent voxel's vertex. Each vertex has a list of references to at most 6 connected vertices.
- Surface to surface. Required for seeking triangle-strips. Each surface refers to all neighboring surfaces.
- Surface to vertex. Required to access vertices for rendering each surface.
- Vertex to surface. Required for connecting new surfaces. The reference also helps to add the surface-to-surface connections instantly.



**Figure 3.4** Adjacency information between surfaces, vertices and voxels.

### 3.4.3 Procedural Volume-Data Creation

To generate complex terrains, a basic procedural volumetric method, constructive solid geometry (CSG) operations, is employed. CSG is applied to the volume data as shown in **Fig. 3.5** to achieve the desired result. In general, multiple simple shape elements are procedurally added and subtracted from the empty voxel-volume using Boolean operations to create complex landscapes for testing purposes. The required parameters, size and position of each shape element, are generated at random within a user-defined range.



**Figure 3.5** Terrain synthesis: based on CSG (constructive solid geometry) and Boolean operations.

The presented approach is similar to [13], which requires time consuming pre-processing computations; the only difference is that the proposed method computes on-the-fly only the terrain portions contained by the ClipBoxes rather than pre-computing the entire terrain.

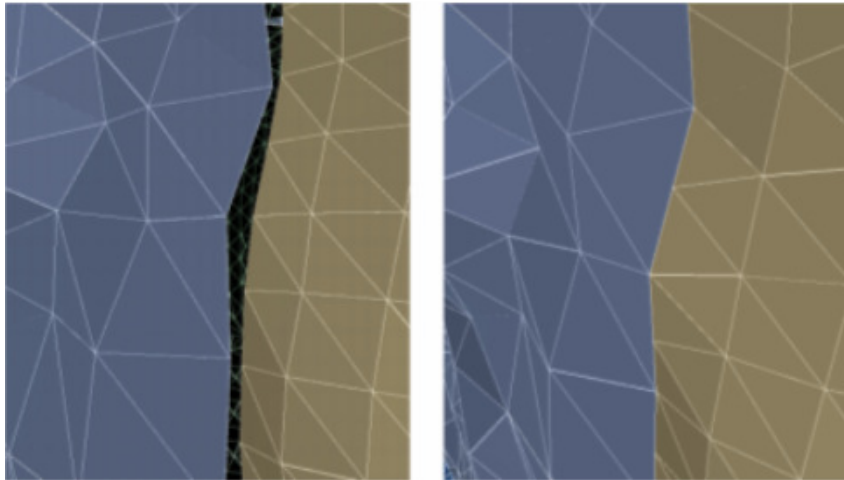
#### 3.4.4 Volume-Data to Polygon Conversion

Concerning the required basic conversion from volume data to polygons, numerous algorithms are available (e.g. [30], [32] or [33]). However, as mentioned earlier, since also LOD has to be employed, these three related algorithms are not directly applicable. In addition, it is necessary to take care of the following two issues: first, how to remove gaps present in LOD boundaries efficiently (**Fig. 3.6**) and second, how to achieve a fast conversion.

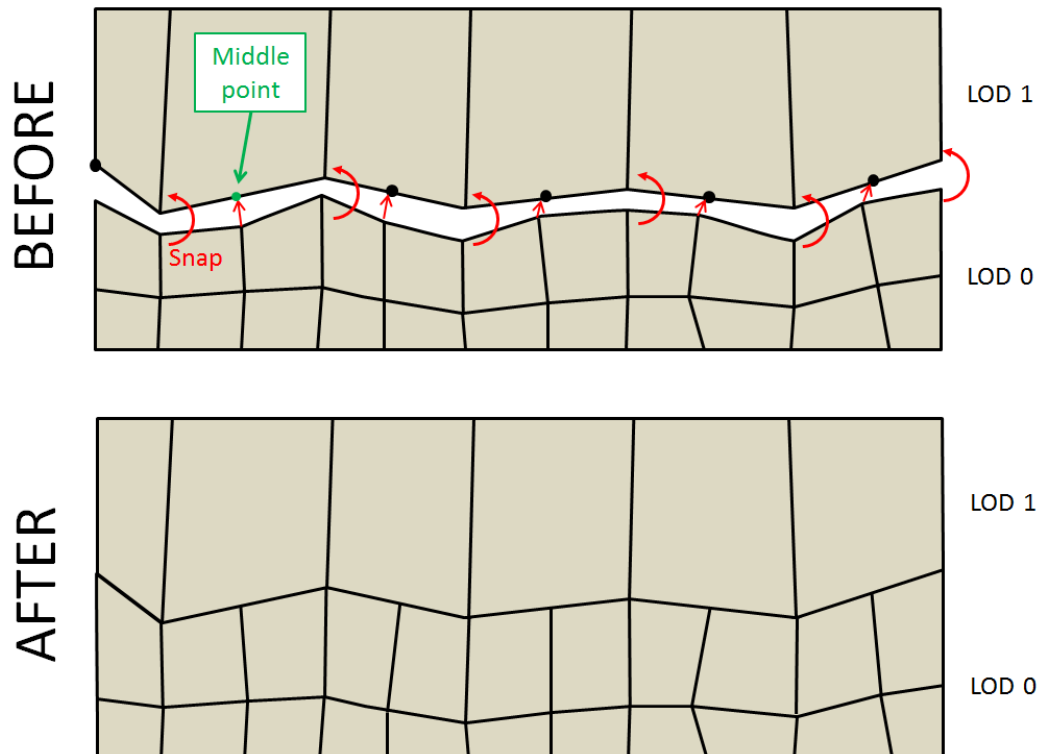
Marching cubes [30] and marching tetrahedra [32] achieve a fast and high quality conversion from volume data to polygons. However, they complicate the welding process for the two LOD boundaries, and also generating adjacency information between vertices gets more difficult.

This thesis solves the gap problem at LOD boundaries by snapping the outer boundary vertices of one LOD to the next higher LOD's inner boundary's vertices, as depicted in **Fig. 3.7**. The upper half of **Fig. 3.7** shows the original result, the lower half shows the seamless result. The vertices of the inner LOD (LOD 0 in **Fig. 3.7**) are snapped to the next higher LOD's boundary (LOD 1). The vertices are snapped to boundary vertices of LOD 1 or to middle points between these vertices.

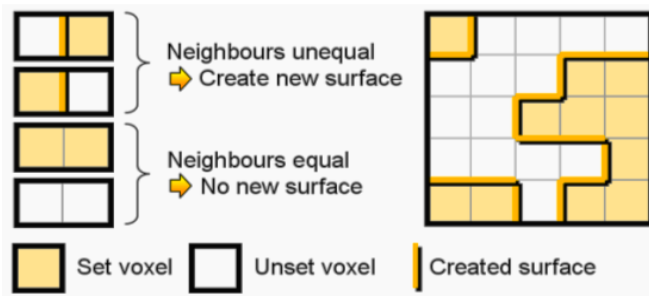
The fast conversion is achieved by using an efficient pointer structure.



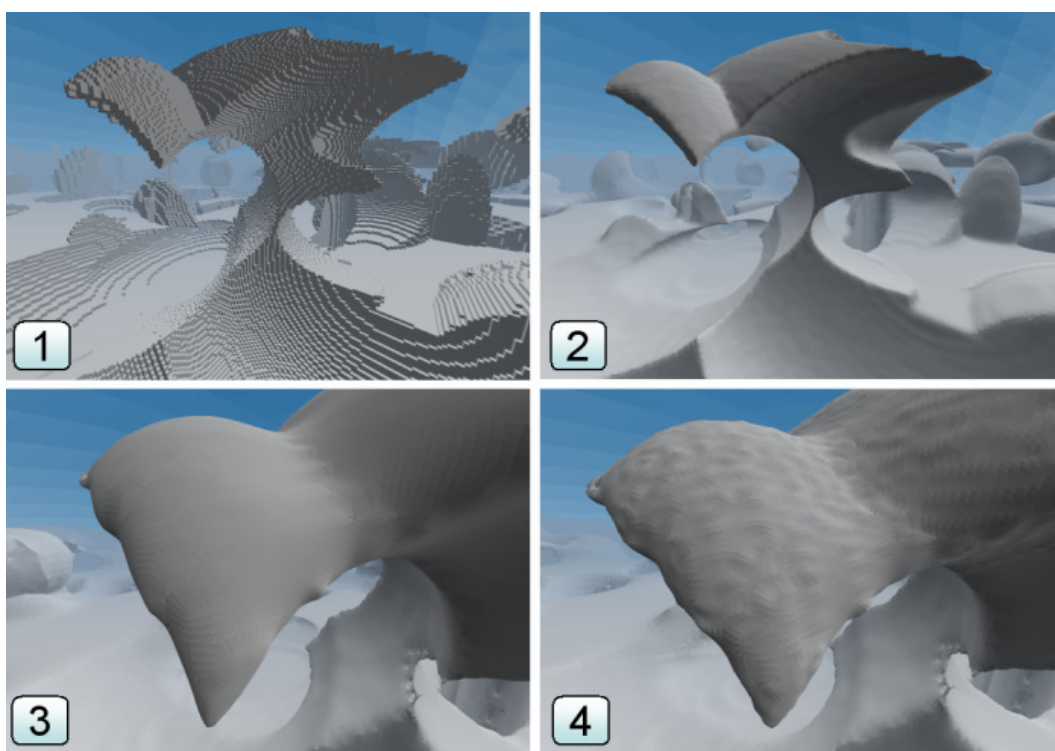
**Figure 3.6** Clip-Box connectivity: A simple method (left) yields an erroneous gap, while the improved version (right) solves this problem



**Figure 3.7** Seamless connections by improved method.



**Figure 3.8** Voxel to polygon conversion: Surface creation in the 2D case.



**Figure 3.9** Geometry-processing: The four images show the proposed steps to process the initial mesh: (1) direct conversion from volume data (2) smoothed (3) surface subdivision (4) synthetic details.

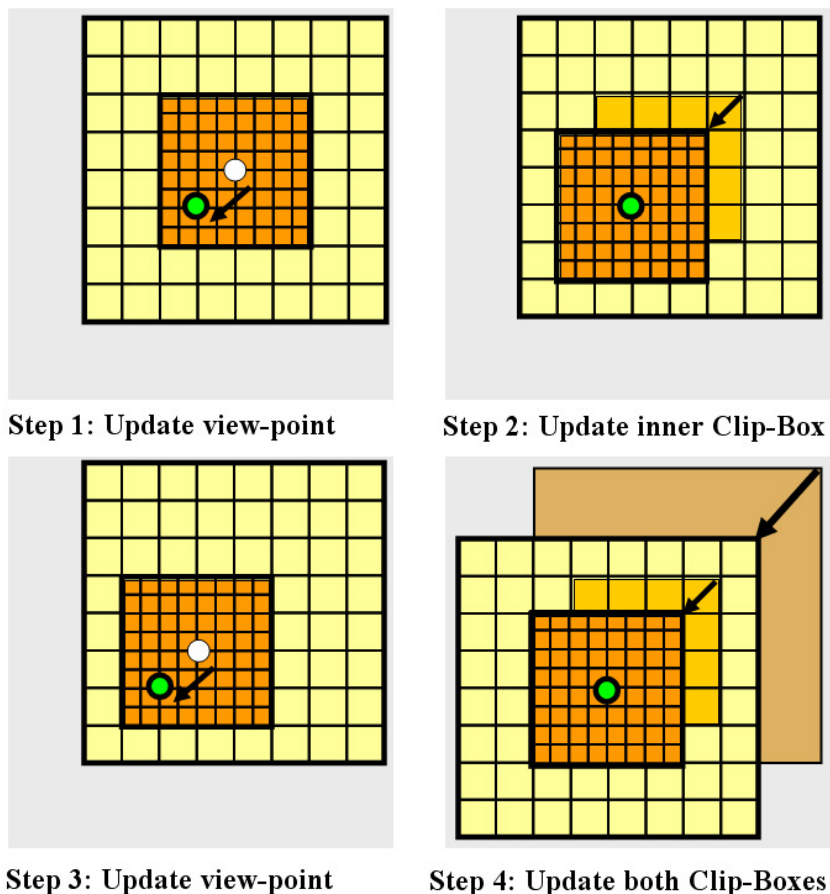
The volume data used in this thesis is binary: each voxel is either set or unset. A simple sketch is shown in **Fig. 3.8**, which demonstrates the voxel-to-polygon conversion for the 2D cases. In case of 2D, a surface is created. In case of 3D, the z-direction is also checked. If the values of the voxel and its neighbors in the x (horizontal) or y (vertical) direction are different surfaces are created. In case of 3D, each voxel is defined as a cube with six quadrilateral surfaces. This allows to connect the geometry of bounding LOD levels efficiently without seams (**Fig. 3.6**) by further enabling the fast creation of adjacency information. The drawback of this approach is obviously a blocky result of the initial polygonal conversion (**Fig. 3.9**, image 1). This is solved by geometry smoothing in a

post-processing step (**Fig. 3.9****Fig. 3.8**, image 2). To weld two LOD levels, the conversion algorithm processes all voxels present in the boundary between two nested CB's as previously explained.

### 3.4.5 Nesting

Nesting is required by the proposed algorithm to achieve LOD, which helps to reduce the number of triangles to be renewed. The LOD and nesting are shown in **Fig. 3.2**. The scale factor for the Clip-Boxes increases exponentially by the power of two, while the number of voxels contained by each Clip-Box remains constant. For example, the size of the innermost CB (CB-1) is  $100 \times 100 \times 100$  voxels, the size of the second innermost CB (CB-2) is  $200 \times 200 \times 200$  and so on; however, the number of voxels contained by each CB is constantly  $100^3$ . This means the voxel size for CB-1 is one, the voxel size for CB-2 is two, four for CB-3 and so forth. This can be seen in **Fig. 3.2**. For each CB, all geometry that overlaps with the next inner CB is spared from rendering.

It is also important that all CBs are connected seamlessly without exhibiting gaps at the border geometry. Gaps occur if the boundaries of two nested CBs are not well connected, as demonstrated in **Fig. 3.6**. Therefore, once the creation of a CB is finished, vertices present in the border are connected properly with the next outer CB to avoid gaps, which welds both Clip-Boxes together. The connection can be achieved efficiently by exploiting pointers of the data-structure. There, each vertex is connected to up to six neighbor vertices, as there might be a neighbor vertex in  $+x$ ,  $-x$ ,  $+y$ ,  $-y$ ,  $+z$  and  $-z$  direction.



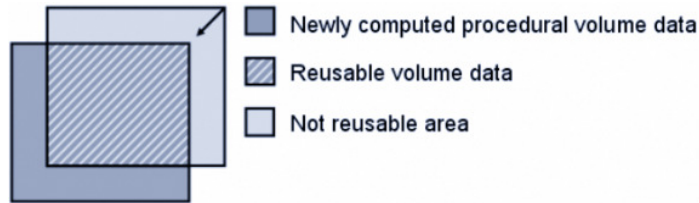
**Figure 3.10** Moving the viewpoint: ○ initial point, ●: the next view-point.

### 3.4.6 Moving the View-Point

In the event that the viewpoint is moved (**Fig. 3.10**), it is important to verify all Clip-Box positions in order to preserve the concentric LOD structure. In an ideal case, all Clip-Boxes are permanently centered about the viewpoint, even if the observer starts moving. However, it is impossible to update all Clip-Boxes fast enough. Therefore, the inner Clip-Boxes is updated more frequent than the outer ones, as done in [12]. For example in **Fig. 3.10** the viewpoint change from step 1 to 2 only requires the inner CB to be updated. The outer CB remains at its position, if the viewpoint change is not significantly large. Moving only the inner CB is possible, as the outer CB accommodates all the geometry enclosed by its volume and can hence fill the gap caused by the displacement of the inner CB. Another advantage of this approach is that the number of displayed triangles can be dynamically adjusted by omitting the innermost Clip-Boxes from rendering.

To minimize the amount of procedural volume data to be computed newly in the event of a Clip-Box-update, the previously computed data is cached and only differential updates (**Fig. 3.11**) are performed on the fly, where the updated portions are referred to as newly computed. After the Clip-Box (CB) is moved, most of the volume data can be reused and only few portions need to be newly

computed by the procedural terrain generation algorithm. The update only includes volume data, but it does not include pointers, vertices or quad surfaces. These are all re-computed from scratch after updating the volume data. Caching of the generated geometry is more complex and left for future work.



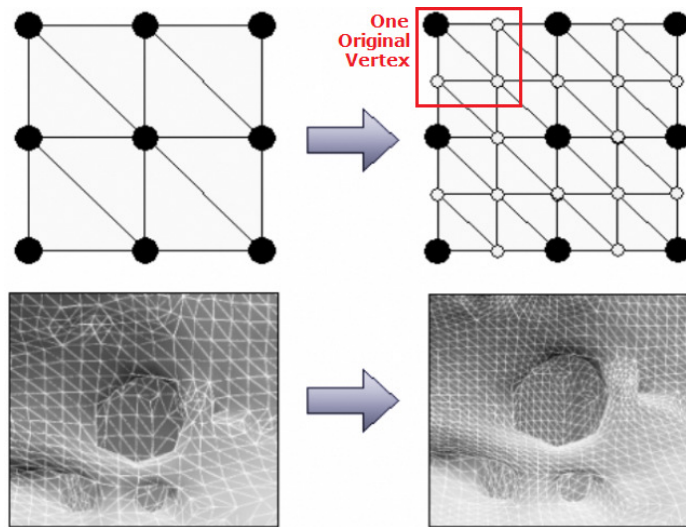
**Figure 3.11** Caching volume data

### 3.4.7 Geometry Post-Processing

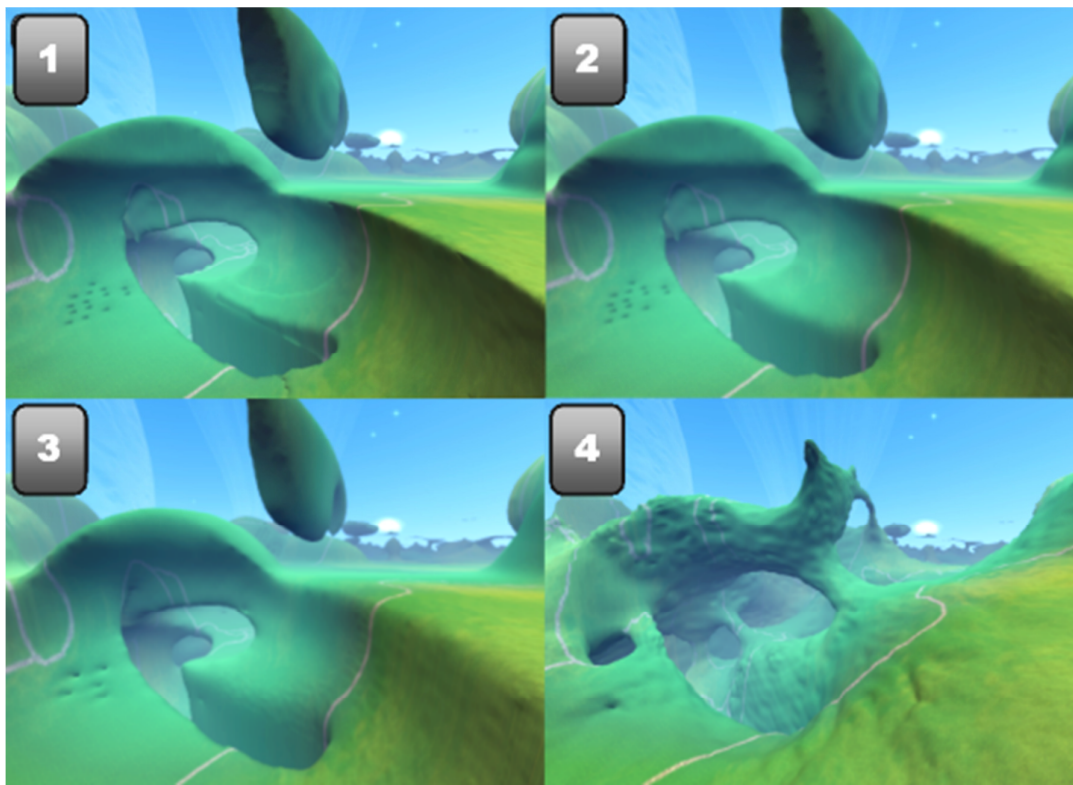
After the surfaces are obtained, smoothing by vertex averaging is applied so that very blocky structures after the initial conversion are significantly suppressed in the mesh. **Figure 3.9** shows the difference between an image without smoothing (image 1) and with smoothing (image 2), where image 1 is very blocky and image 2 is very smooth.

In the event that a high update rate for inner CB's near the viewpoint is needed, the proposed algorithm enables fast creation of Clip-Box geometry from surface subdivision rather than using the more complex extraction from volume data. For surface subdivision, the existing triangle mesh of a CB is used, and each triangle is subdivided into two triangles. As shown in **Fig. 3.12**, for each vertex, three additional vertices are inserted so that the regular grid structure of the CB mesh is preserved as much as possible. This is significantly faster than generating volume data and converting the volume data to triangles. **Figure 3.13** shows the result of creating the CBs from four different LODs, where image 1 shows the original, image 2 shows the innermost CB created from surface subdivision and additional fractal details by random midpoint displacement, image 3 shows the two innermost CBs created from surface subdivision and additional fractal details and image 4 shows the three innermost CBs created from surface subdivision.





**Figure 3.12** Triangle subdivision. Upper: adding vertices (white circles), lower: subdivision example



**Figure 3.13** Fractal details: Image 1: no fractal details. Image 2: fractal details for the innermost CB. Image 3: Fractal details for the two innermost CBs. Image 4: Fractal details for the three innermost CBs.

To make the generated terrain look more interesting, another post-processing is performed. Synthetic details are generated by random midpoint displacement [25], whose effect can be seen in **Fig. 3.9**'s images, images 3 and 4.

To improve the computing speed, a module to group all surfaces into triangle strips is added, allowing cache-optimal rendering. This is done by a depth-first search, utilizing the surface-to-surface connectivity information.

In case of creating many CB's from polygon subdivision rather than volume data, problems near voxel patterns that are equal to the ones shown in **Fig. 3.14**, lower left corner often occur, which results in affecting the smoothed result. In **Fig. 3.14**, those critical regions are indicated by a white circle. To solve this issue, a 2 by 2 pixel filter (lower left in the **Fig. 3.14**), which detects and suppresses these patterns, is employed where in **Fig. 3.14** the dark and white pixels in the filters represent darker and brighter colors in the synthesized image, respectively. Specifically, the left 2 by 2 pixel pattern shown in **Fig. 3.14** is searched and replaced by the right pattern. The result (upper-right) indicates that most of the problematic patterns present in the upper-left image can be successfully eliminated.



**Figure 3.14** Smoothing errors and their elimination: Upper-left: Original image with smoothing errors, Upper-right: Result of smoothing; Lower-left: pattern is replaced by lower right pattern.

## 3.5 Experimental Results and Discussion

### 3.5.1 Implementation

The proposed algorithm was implemented by using C++. For the graphics API, OpenGL was employed. A two-thread approach is used to separate geometry processing from rendering (**Fig. 3.1**). This approach maps well to the current generation of multi-core processors, as each thread is able to occupy one core. It is possible to use the CPU core affinity functions of the operating system to assign each thread to one specific CPU core. In this implementation, affinity

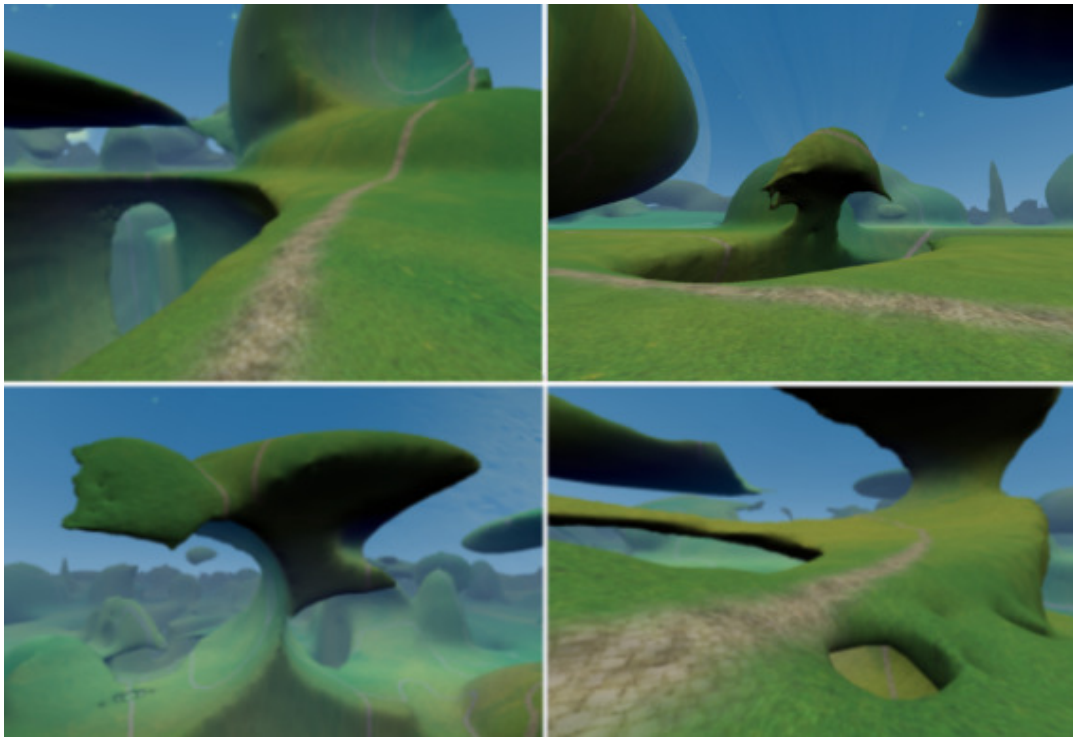
management was left to the operating system. The assignment was verified using task manager. Each thread uses the corresponding CPU core to 100 percent continuously. Load balancing was not implemented. The task distribution of the two threads is as follows.

Thread one, the Geometry Thread, is in charge of computing the CB's mesh. This involves polygon extraction from voxel data, triangle subdivision, mesh smoothing and random midpoint displacement (Synthetic details).

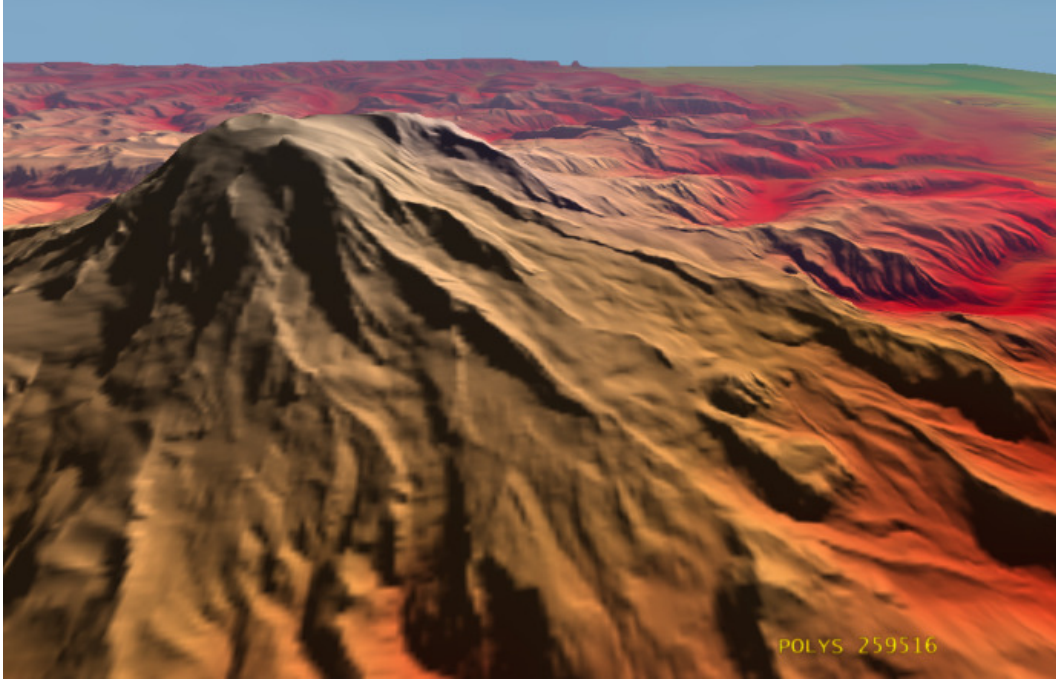
Thread two, the Rendering Thread, is in charge of rendering all CB meshes correctly by sparing the triangles of the next smaller CB inside. As it runs in parallel to the first thread, it is necessary to be aware of the concurrent use of the mesh data. This was solved by implementing a double-buffer system, where each mesh buffer is assigned to one thread. Then, once a CB update is completed, the buffers are swapped synchronously.

### 3.5.2 Immediate Visualization

To demonstrate that the proposed method is pre-computation free, an example terrain consisting of about 50000 Boolean operations is generated and visualized. The terrain data is evaluated concurrent to the visualization without using any pre-computation. The result can be seen in **Fig. 3.15**. The hardware for testing was a dual core Pentium D 3.0 Ghz, equipped with 1GB RAM and an NVIDIA GeForce 8600 GTS graphics card.



**Figure 3.15** Terrain used for Benchmark.



**Figure 3.16** Terrain visualization from height-map (real data) Puget Sound region in WA, USA.

To further demonstrate that the method is pre-computation free and can be used to visualize height map based terrains, the height-map in **Fig. 3.16** is processed. Here, the height-map serves as source for the CB volume data. It is converted into volume data instantly. The height-map and the color-texture are publicly available on the U.S. Geological Survey (USGS) servers<sup>54</sup>. The major difference between rendering height-maps by volume based methods and conventional height-map based methods is the vertical resolution. While the vertical resolution of the proposed volume based method is reduced with each level of detail, height map based methods, such as geometry clip-maps, have a constant vertical resolution such as 16 bit integer per height-map pixel.

### 3.5.3 Unlimited Terrain Size

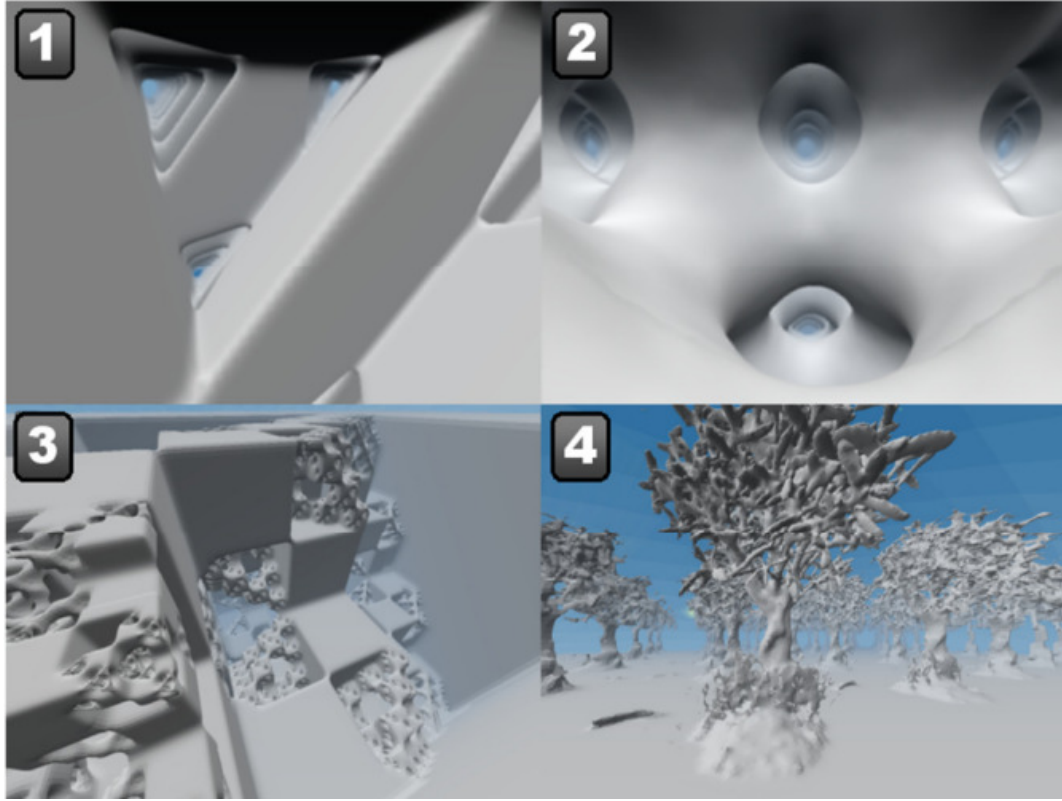
To show that the proposed method is able to visualize unlimited sized terrains and further has application beyond gaming, this section shows its capability to serve as a 3D function grapher to visualize infinite math functions. The proposed method is able to visualize any function

$$f_{Math}: \mathbb{Z}^3 \rightarrow \{0, 1\}, \quad (3.1)$$

that is, the function input is defined as a three dimensional integer coordinate vector (Euclidian space), while the output is defined as zero (represented as air in the visualizer) or one (represented as solid terrain). Results of three generic functions are shown in **Fig. 3.17**, where images one to three show this ability.

<sup>54</sup> United States Geological Survey, "http://www.usgs.gov," 2012.





**Figure 3.17** Function plotting and real data: [1] to [3]: three different mathematical functions; [4] conventional iso surfaces.

In **Fig. 3.17**, three Boolean functions are visualized: (1) exclusive-or, (2) saw-tooth and (3) sine curve. The specific functions, which only return true or false, are represented by Eq.(3.2) below:

$$\begin{aligned}
 f_{XOR}(x, y, z) &= ((x \text{ xor } y \text{ xor } z) \bmod 1000 < 357), \\
 f_{SAW}(x, y, z) &= ((x + y + z) \bmod 1000 < 500), \\
 f_{SIN}(x, y, z) &= (\sin(x) + \sin(y) + \sin(z) < 2)
 \end{aligned} \tag{3.2}$$

As the evaluation and visualization are done immediately, it is further possible to alter the function parameters on run-time.

In **Fig. 3.17**, image 4, the applicability to rendering iso-surfaces [34] is demonstrated. A forest generated from the well-known bonsai tree data set is shown in image (4). The different levels of smoothing can clearly be seen, while the amount of smoothing applied was linear to the size of the CB in order to limit the loss of geometric details. The tree that was used was rescaled to a resolution of  $256^3$  and placed in the landscape 25 times. The tree scene as well as the function plot scene was rendered with a CB resolution of 192 at about 10-15 fps.

### 3.5.4 Concurrent execution of generation and visualization

To analyze the speed performance of the concurrent generation and visualization approach, two benchmarks are conducted using procedurally generated volume data. First, a detailed timing of the algorithm pipeline is measured in **Table 3.1**. Second, an evaluation of the continuous timing behavior of a flight lasting 222 seconds through a landscape is shown in **Fig. 3.18**.

In the first above-mentioned benchmark, the timings for one CB resolution (128) are measured in detail and compared the results with different CB resolutions. In the test, 5 (CB no. 3 to 7) out of the 7 CB's were created from volume-data, whereas the two smallest CB's (no.1 and 2) were created from subdivision and enhanced with random mid-point displacement, which is explained in section 3.4.7. The equivalent size of the visualized data volume is  $2048^3$  voxels.

Concerning the timing evaluation, **Table 3.1** shows that most of the time is spent for the surface extraction process (voxels to polygons). The procedural volume data generation requires relatively less time, which is the result of employing the caching scheme, explained in Section 3.4.6. If caching is switched on, about 80% of a CB's volume data can be reused during a CB update, which reduces the average time for the procedural computation from 100ms to about 20 ms. The rendering time for each CB (CB-1 to CB-7) at resolution 128 (upper half) shows that most time is spent for the innermost CB (CB-1, 161ms), while the outer ones require less time (CB-7, 39.6ms).

In the lower half of **Table 3.1**, different CB resolutions are compared. The Geometry Thread is referred to as *Thread 1* and to the Visualization Thread as *Thread 2*. In **Table 3.1**, the average time to update one CB (CB update avg.) can be seen. It is roughly proportional to the number of processed voxels.

**Table 3.1** Performance analysis: In the upper row, update and render times for one CB resolution (128) are analyzed in detail, while the lower row compares the performance of different CB resolutions.

ClipBox 128	7	6	5	4	3	2	1
Procedural generation	0	0	16	23	114	116	42
Voxels to polygons	0	0	658	674	727	885	745
Subdivision	29	41	0	0	0	0	0
Smoothing	0	2	96	137	152	174	178
Fractal details	4	7	0	0	0	0	0
Surface normals	13	16	28	39	43	41	52
Triangle-strips	4	7	18	61	52	49	76
Total (ms)	51	73	816	933	1088	1265	1093

Render(ms)	1	1	3	6	6	5	6
Polygons (k)	39.6	40.2	90.2	121	127	113	161

ClipBox Resolution	192	160	128	96	64	
CB update avg. (ms)	2671	1315	989	476	162	Thread 1
Render Total (ms)	68	47	29	17	8	Thread 2
Polys Total (k)	1657	1115	692	352	134	
Memory usage (MB)	418	410	265	151	108	

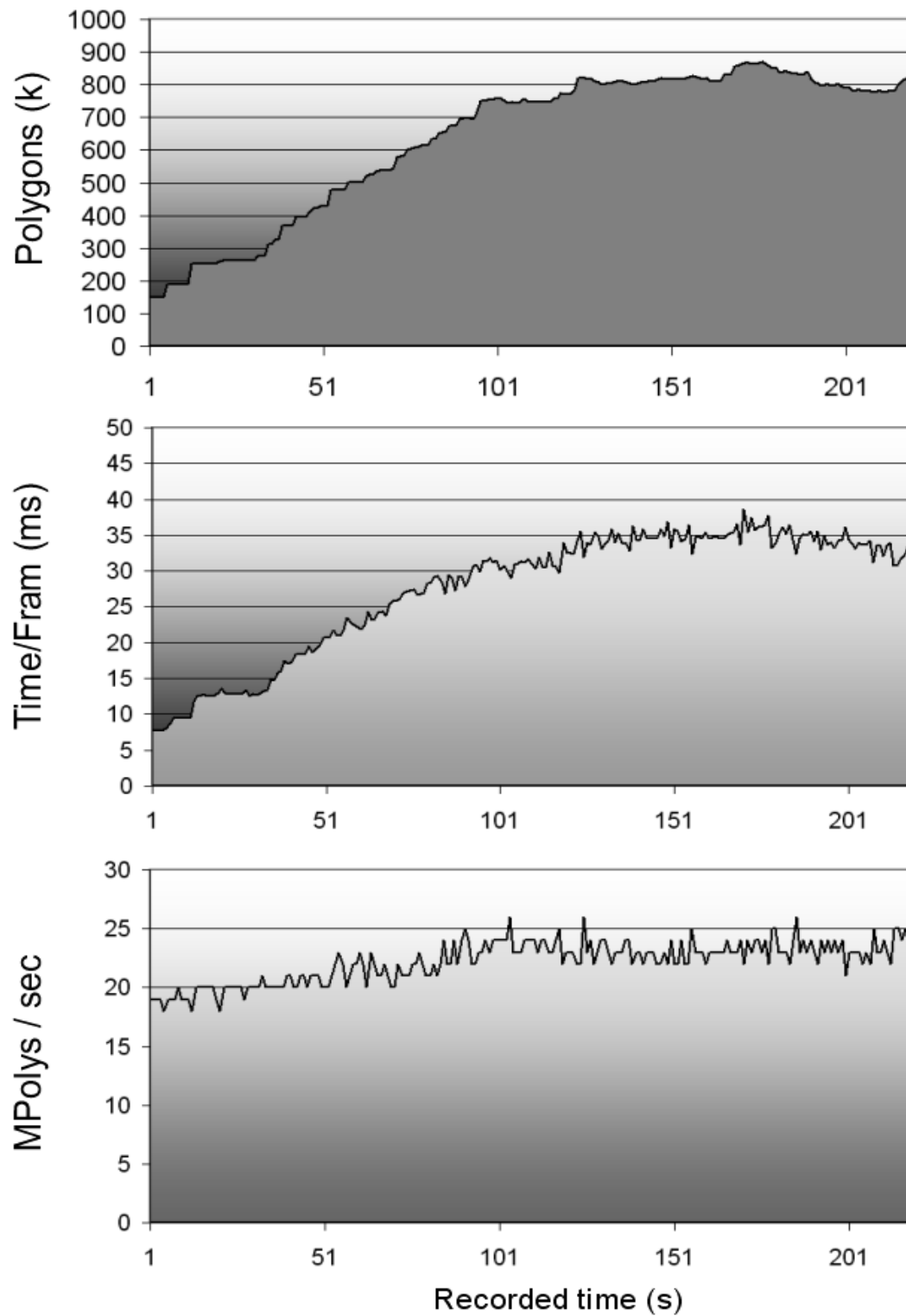
In general, the update frequency for a CB resolution of 128 is sufficient for an interactive exploration at high quality, but it is not well-suited for a fast fly-through. In this case, either lower resolutions such as 96 or 64 are suited well, or increasing the number of CBs created from subdivision can also be helpful, as well as the above mentioned opportunity to reduce the number of CBs. In many cases, an increased number of CBs created from subdivision combined with random midpoint displacement might even be desirable. It often looks more appealing and natural than the initial terrain without using subdivision. In **Fig. 3.13** this behavior is shown in four steps, where each step is equivalent to generating one more CB from subdivision.

In the second performance test, the frame-rate continuity of the proposed method is analyzed. Often, visualization algorithms using LOD have difficulties to provide a continuous frame rate because for many methods the geometry updates cause short stalls in rendering, which can be observed as hic-ups in the frame rate. To confirm that the proposed method does not have this problem, benchmark data over a longer period of time is recorded, while flying through the artificial terrain shown in **Fig. 3.15**. The diagram for the record is shown in **Fig. 3.18**, in which the performance results in terms of polygon throughput (Mpoly/s), time per frame (ms) and the polygon count (in thousand triangles) are shown. Even at polygon-counts around 800k, the triangle throughput remains continuous at about 20 million triangles per second and does not reveal major peaks. If the rendering time per frame (time/frame) is further regarded, smooth changes in proportion to the scene's complexity (Polygons) can be noticed. The proposed algorithm, therefore, does not reveal any problems that might occur due to the LOD. The frame-rate ranged from 25 to 130 frames per second, which is sufficient for interactive applications such as video games.

In order to measure the rendering quality of the visualized landscapes, the landscape of **Fig. 3.15** is analyzed at different Clip-Box resolutions, disabling subdivision and texturing. As a reference, the highest possible resolution that the hardware was able to handle was chosen: a landscape with 7 Clip-Boxes at a resolution of 192. This is equivalent to visualizing a total data volume of  $12288^3$  voxels, which would require roughly 210 GB of memory, assuming each voxel is represented by a single bit. To measure the increased inaccuracy for lower CB resolutions in screen-space, the renderings of lower Clip-Box resolutions were compared to the reference resolution, as can be seen in **Fig. 3.19**. To evaluate the screen-space-error, all images were gray-scaled and each pixel was marked as erroneous if the difference is more than 20 in a range of 0 to 255, which is considered to be noticeable, from the reference image (taken at highest resolution).

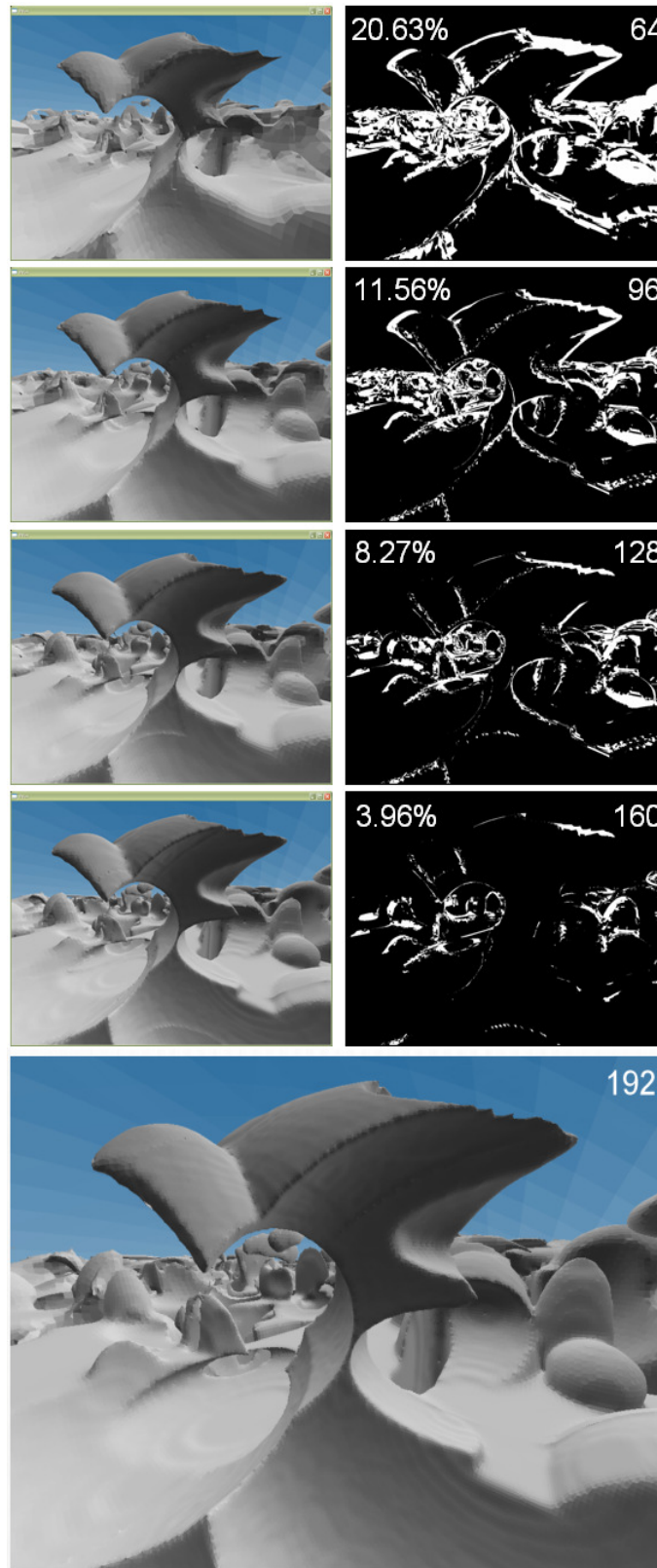
As shown in **Fig. 3.19**, the highest CB resolution is compared to with lower resolutions from 64 to 160, where the percentage of erroneous pixels is calculated. The errors range from 3.96% to 20.63%.

The qualitative results show that good quality renderings are achieved if the Clip-Box resolution is 128 or higher. For lower resolutions, the screen-space error increases significantly and leads to more inaccuracies, particularly at very distant geometry. Concerning the quality in general, an asymptotic error behavior is observed, where the error is roughly halved for each increase in the resolution by 32.



**Figure 3.18** Continuous performance: for a flight in the landscape in **Fig. 3.5**. Upper: polygon vs time, middle: time to visualize one frame vs time, bottom: million polygons per second vs time.



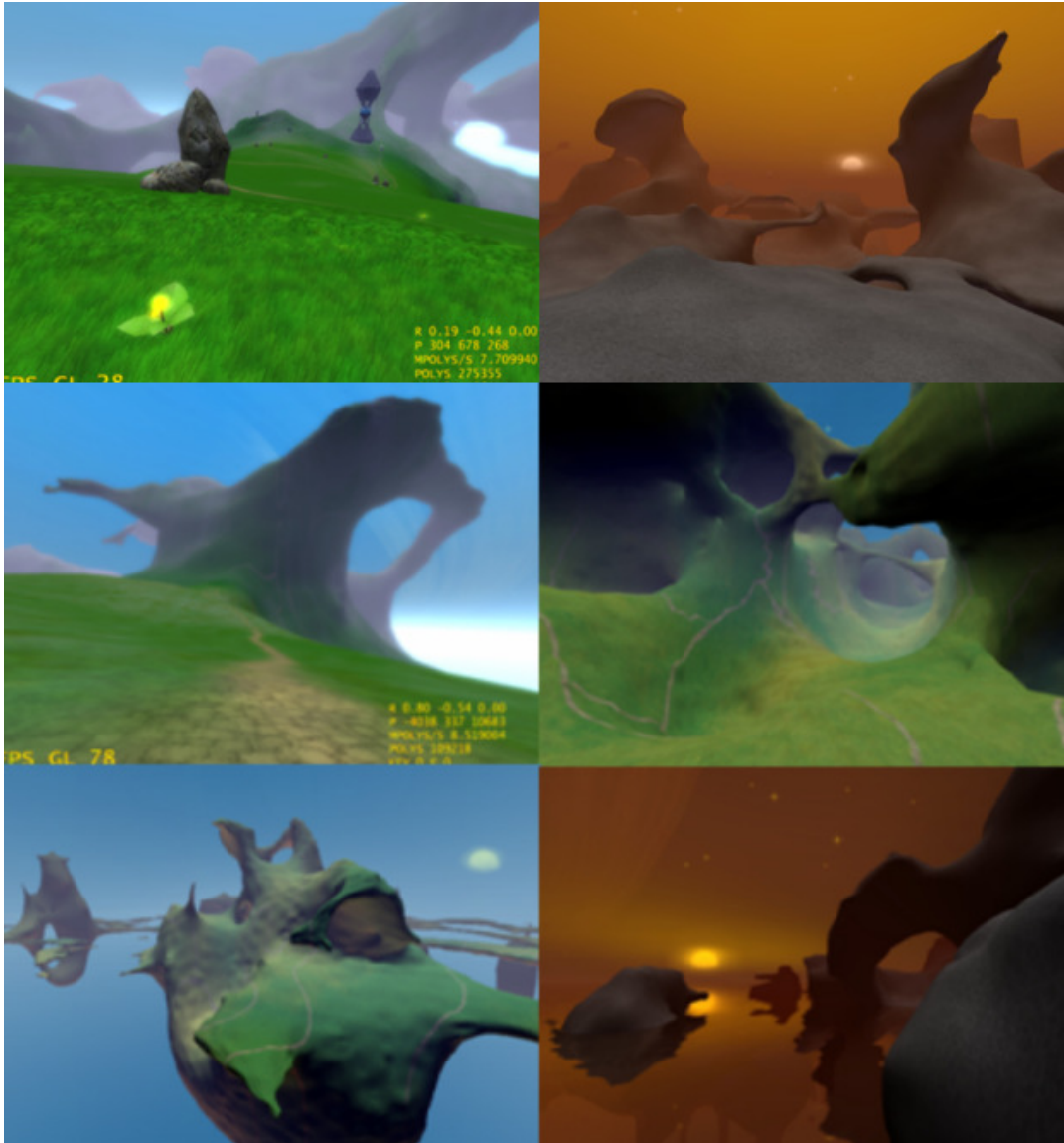


**Figure 3.19** Screen-space error: Comparing the highest Clip-Box resolution (192) with lower resolutions: 64 (top-most), 96 (2<sup>nd</sup> top), 128 (third), and 160 (fourth).

### 3.5.5 Demonstration

Results of synthesizing terrains by the proposed method are shown in **Fig. 3.20**, which demonstrate a variety of terrains can be visualized. The upper-left image shows a terrain that is additionally enhanced by shaders for the grass and handcrafted items to demonstrate the applicability for computer games.

Note that arbitrary 3D terrains such as overhangs, which cannot be generated by conventional height-map based methods, can be generated.



**Figure 3.20** Examples of synthesizing terrain.

### 3.5.6 Limitations

Since the proposed method is based on volume data, the average memory consumption is higher than conventional height-map based methods such as geometry clip-maps. In general, 3D requires more memory than 2D.

Regarding the geometry update of a clip box in case that the view-point is moved, this might be slightly visible in case of low Clip-Box resolutions.

## 3.6 Conclusion

A nested CB based approach that is able to visualize procedural volumetric terrains with unlimited size has been proposed. The nested Clip-Box is an evolution of the nested geometry clip-map, which is used for height-map based terrains. A Clip-Box consists of a cubic regular grid of voxels and the corresponding triangulation. Nested Clip-Boxes allow the immediate and pre-computation free visualization of arbitrary sized volume data. Experiments are conducted using data generated from terrain functions, data from existing volume data sets and height-map data. Experimental results and discussion are summarized as follows:

- *Pre-computation Free*: The immediate and pre-computation free visualization of volumetric terrain data is achieved. This property is proven by the experimental results, which demonstrate that a Clip-Box can be computed from mathematical functions within about one second concurrently to the visualization. This includes the computation of new terrain data on the fly without pre-computations (e.g. accessing to storage devices), which all methods [5] [7] [8] [27] [28] [29].
- *Unlimited Terrain Size*: The proposed method can visualize any arbitrary sized terrain. This is proven by visualizing volumetric terrain data computed from simple mathematical functions, which are defined in an infinite coordinate range. The results of this visualization are demonstrated experimentally.
- *Concurrent Generation and Visualization of Generation of Procedural Volumetric Terrain Data on the Fly*: It turns out that as soon as the terrain generation updates its data, the terrain visualization renders the updated data. The related methods cannot achieve this because they have not the ability.

Future work for improving the proposed method includes re-using the generated triangle data for reducing the updating time of the CB triangle data and improving the filter mechanism employed to remove ambiguous voxel patterns in the CB data.



## Chapter 4. Static Objects

### 4.1 Goals

The goal of this chapter is to solve limitations of existing methods for visualizing static objects by proposing a voxel-based raycasting approach. The goals are summarized as follows:

- Higher computation speed: The proposed approach should be able to visualize better for complex voxel scenes faster than conventional splatting methods, polygon based rasterization and voxel based raycasting methods.
- Lower memory consumption: The proposed approach should consume less memory than related methods for triangle based raycasting

As a summary, the best result of all categories, which are high rendering speeds for complex scenes and low memory consumption should be achieved.

### 4.2 Related Work

This section focuses only on voxel related works. The methods are split into three groups: rendering voxel volume data by using Shear-Warp [35], ray tracing based algorithms and point based rendering. The most widely used methods are briefly overviewed in each group, and their key issues are mentioned. Shear-Warp renders RLE volume data in a front to back manner to a temporary texture. The temporary texture is then mapped to the screen. It has been proven to be very fast for dense, semi-transparent volume data. However, it requires storing three copies of the volume data in memory, as the data is run-length-encoded for each x-, y-, and z- axis independently.

Raytracing methods use tree-like structures such as octrees, KD-trees and bounding volume hierarchies (BVHs) to compress the voxel data and accelerate the raytracing process. An octree-based raycaster proposed by Knoll *et al.* [36] uses a pointer-based octree structure so that large iso-surfaces can be raycast interactively with high quality. The pointer-based octree structure is advantageous in that spatial queries can be made very efficiently. However, it needs to store at least one pointer (usually 4 bytes) for each node, which is more than twice as much as the memory requirement of position data in typical RLE scenes. As a variant of raytracing methods, Gigavoxels [37] uses bricks of volume data in combination with octrees to store voxels for interactive raytracing. The method suits well for raycasting of large, semi-transparent volume data. Its features include filtering for high quality and streaming on demand from the hard-drive to the GPU or CPU for handling data sets that do not fit entirely into CPU or GPU memory. However, Gigavoxels is not optimal for visualizing purely opaque surface data, because this system uses an

uncompressed voxel brick structure, which increases the memory consumption. In GigaVoxels, smoothing of voxels near the camera is solved by utilizing 3D texture filtering.

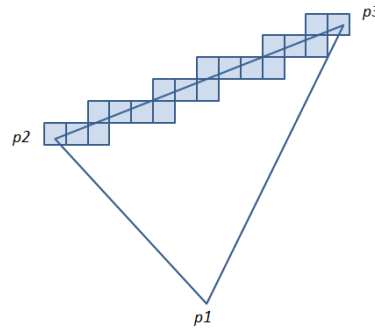
One of the most well-known point based rendering methods, Qsplat [5], inspired many other researchers to propose similar rendering approaches. As an evolution of Qsplat, FarVoxels [8] improved the basic point-based rendering by introducing a hybrid method that also utilizes polygonal rendering for geometries close to the viewpoint. However, as these methods employ either point-based rendering or a combination of point-based and polygonal-based rendering, they suffer from the disadvantages described in this section, compared with voxel-based rendering.

### 4.3 Input Data

For the voxel input data, the proposed algorithm has several conventional and therefore not novel, original or unique data import functions that are necessary to read input data from files and to convert them into voxel data in a pre-processing step prior to the visualization.

#### 4.3.1 Polygon Data Import

This import function can import Stanford polygon (PLY) format data. The format is specified as indexed face list. An example for this format is publicly available at their URL<sup>55</sup>, along with more details on the specification and its development.



**Figure 4.1** Rasterization of a single 2D triangle.

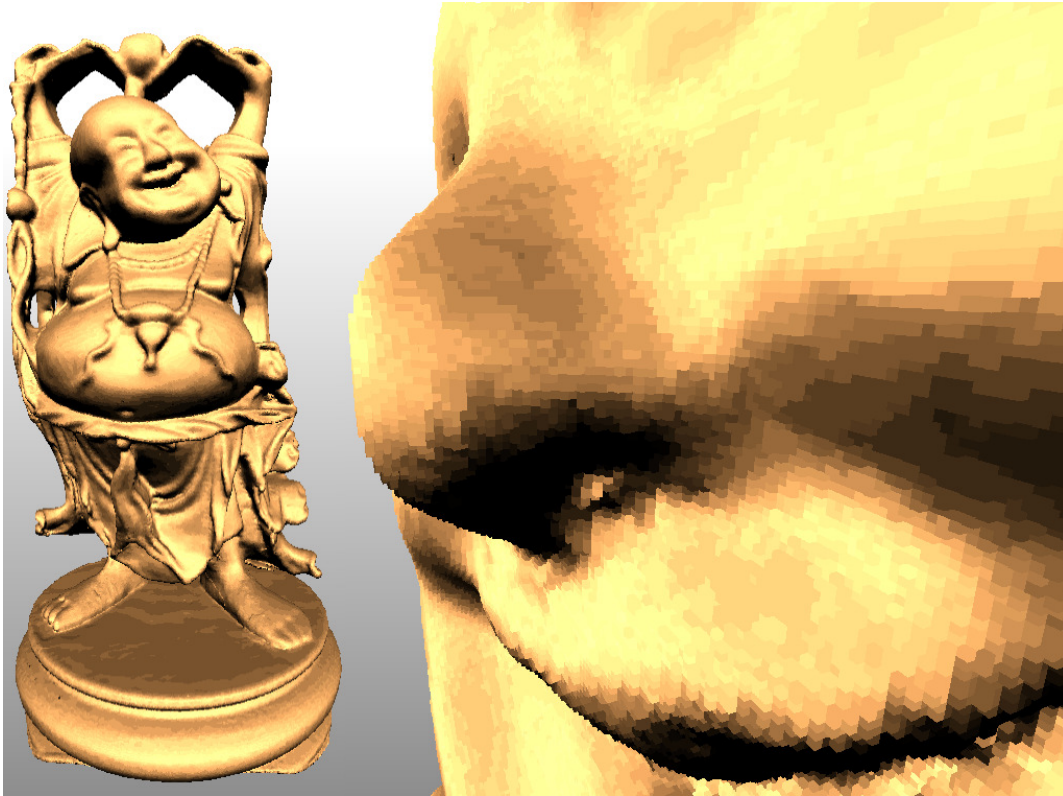
The proposed method accepts only triangulated mesh data. The PLY format supports general polygons without holes. To convert mesh triangles into voxel data, the vertex positions  $p_1$ ,  $p_2$ ,  $p_3$ , of each triangle are converted into the voxel grid coordinate system first (**Fig. 4.1**). Then linear interpolation along two of the three edges ( $p_1, p_2$ ) and ( $p_1, p_3$ ) is carried out. The number of sample points along both edges is defined by the distance between the edges as follows:

$$\text{samples} = 8 \cdot \max(\|p_2 - p_1\|, \|p_3 - p_1\|). \quad (4.1)$$

Next, two interpolated positions along ( $p_1, p_2$ ) and ( $p_1, p_3$ ) are defined as  $p_{12}$  and  $p_{13}$ . The final position is the interpolation between positions  $p_{12}$  and  $p_{13}$ . For each position between  $p_{12}$  and  $p_{13}$  corresponding voxel in the three-dimensional voxel-space are determined and set it to opaque

<sup>55</sup> PLY - Polygon File Format. <http://paulbourke.net/dataformats/ply/>.

color. By doing this, each triangle of the input polygon data can be efficiently rasterized. An example of rasterized result is shown in **Fig. 4.2**.



**Figure 4.2** Rasterization of 3D Polygon Data: Imported result of the Happy Buddha PLY Dataset with approximately one million polygons.

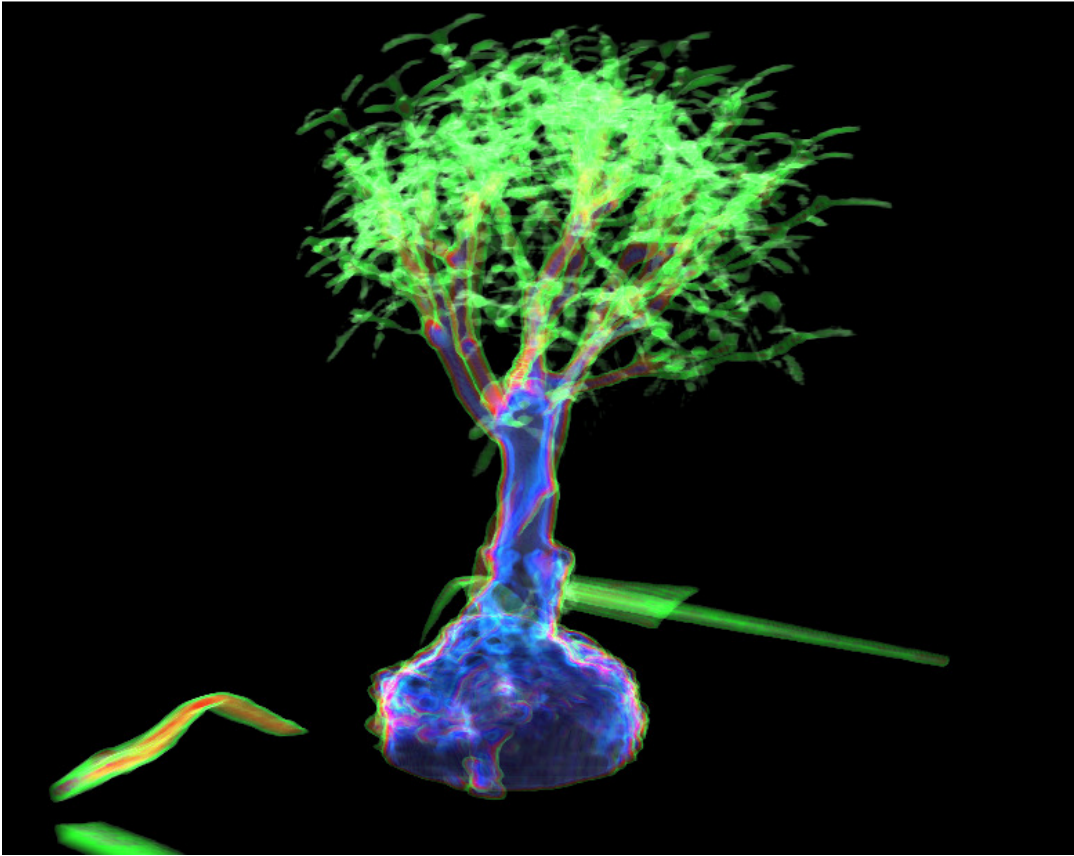
### 4.3.2 Volume Data Import

As a source for sample volume data, The Volume Library<sup>56</sup> is used, in which various data-sets are available for free. Since the proposed visualization method focuses on visualizing opaque data and semi-transparent data sets that derive from medical scans such as MRI (Magnetic Resonance Imaging) or CT (Computer Tomography) scans, a threshold that determines if a semi-transparent voxel is rendered opaque or transparent needs to be applied. This threshold is also known as *iso-value*. The semi-transparent input data is therefore converted into binary voxel data. The imported result from the original bonsai dataset of the volume library (**Fig. 4.3**) can be seen in **Fig. 4.4**. A color gradient to emphasize the voxels that represent the trees leaves is applied in addition to the basic import.

---

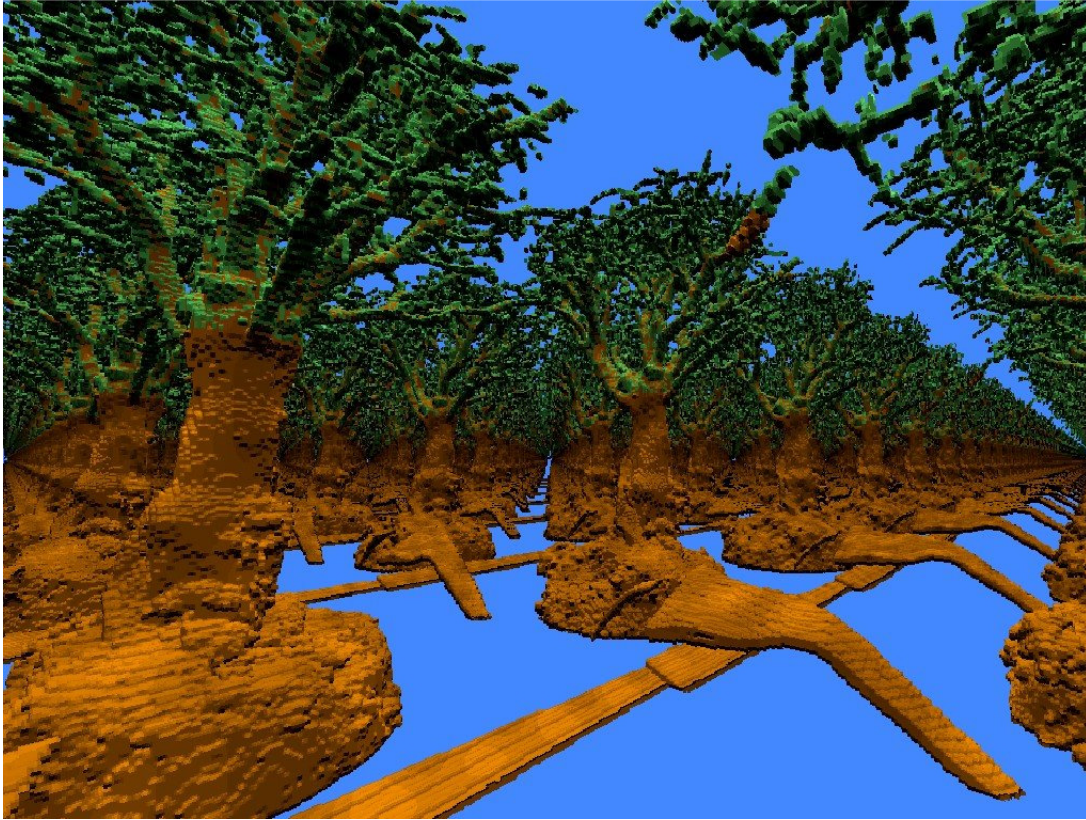
<sup>56</sup> The Volume Library. <http://www9.informatik.uni-erlangen.de/External/vollib/>.





**Figure 4.3** MRI Bonsai Data-Set: A screenshot of the original semi-transparent data rendered in V3, available at The Volume Library.



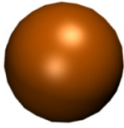


**Figure 4.4** Volume-Data Import: A forest scene created by the voxelized Bonsai data-set.

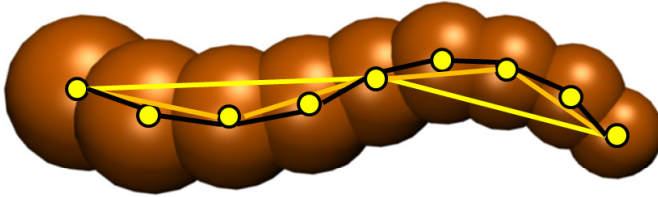
### 4.3.3 Procedural Voxel Objects: Trees

A procedural voxel tree generator based on Lindenmayer systems (L-system, [38]) has been developed to create complex trees with high resolutions. The used algorithm creates a tree based on branches that consist of 3D voxel spheres of variable size, as shown in **Fig. 4.5**. The branches are created by recursively inserting spheres in the middle of two endpoints as shown in that figure. Additionally, random-midpoint-displacement is applied in order to get a more natural result of the branch. The entire tree is generated by starting with one branch from the root and then continuously splitting this branch into two to three smaller branches. Finally inner voxels that are invisible are removed. This operation is done to reduce the memory of the entire dataset. **Figure 4.6** shows the result of trees generated by this method.

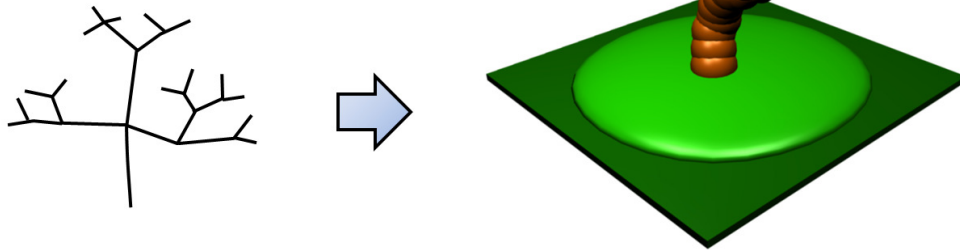
Basic Sphere – The Tree's Atom



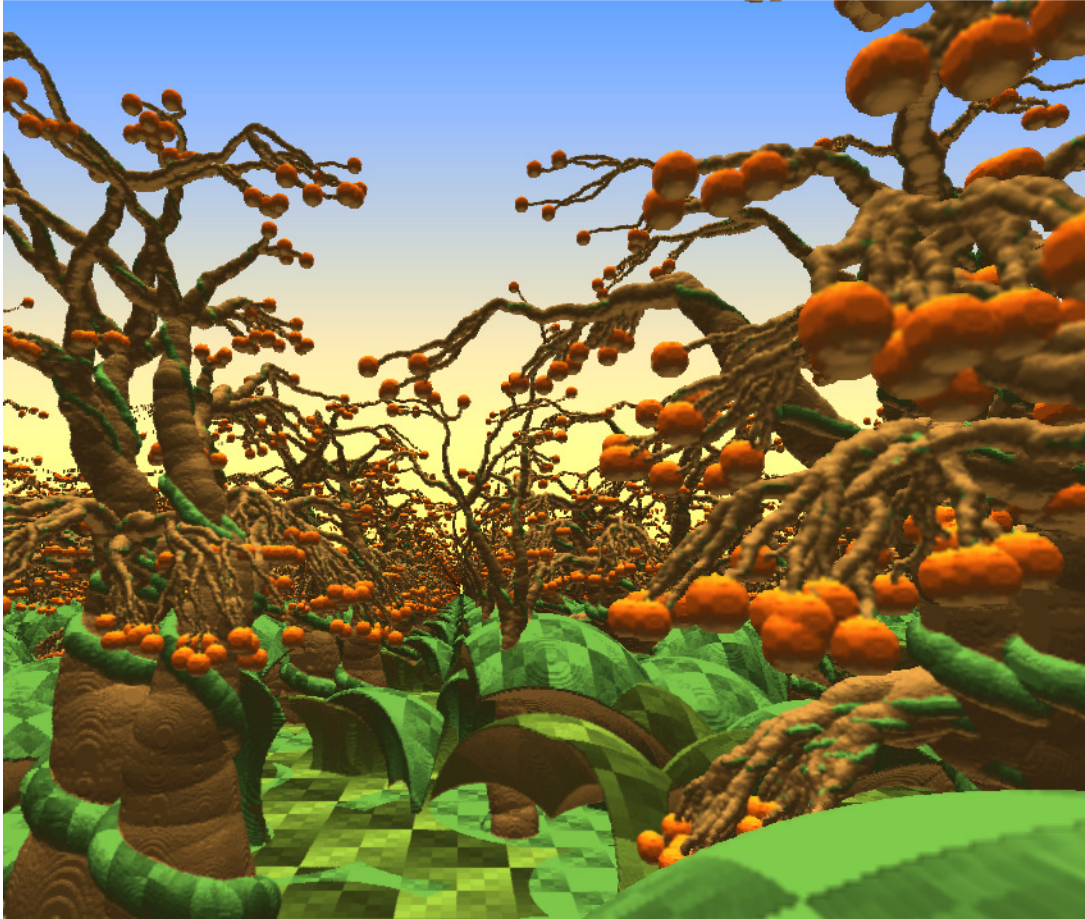
Branch: Created from Spheres and Random Midpoint Displacement



Tree: Created Recursively from Branches



**Figure 4.5** Tree Generation: The tree is created using spheres, random mid-point displacement and finally the L-system.



**Figure 4.6** Example of Procedural Voxel Trees.

## 4.4 Proposed Algorithm

The explanation of the algorithm is split in two sections. Section 4.4 explains the algorithm overview, the proposed approach, the algorithm overview, the pre-processing and the level-of-detail computation. Section 4.5 elaborate on the visualization, which is the heart of the algorithm.

### 4.4.1 Overview

As explained in section 4.1, none of the related methods possesses all of the following two properties: low memory consumption and high rendering performance (fast rendering). Therefore, the purpose of this research is to find an optimal combination of all of the two properties. Furthermore, memory consumption can be reduced by not storing the normal vector inside the voxel data, but recovering the normal as a post-process in screen-space.

Overall, the following goals should be achieved:

- Significantly lower memory consumption compared to other methods.
- High rendering performance even in complex environments at interactive frame-rates from arbitrary viewpoints.
- Support for recovering the voxels' surface normals from the depth buffer.

The proposed approach is based on the so-called “voxel-based forward projection algorithm” developed by Wright *et al.* [39], which renders voxel data with lower memory consumption than the Shear-Warp algorithm. The original voxel-based forward projection algorithm is modified to deal with completely arbitrary voxel data, as it is done in the unpublished work of Silverman<sup>57</sup>. The original forward projection algorithm categorized the data into two groups: terrain, and objects that are placed on the terrain, such as trees and buildings. Each of these two groups of data have its own rendering technique. Silverman's method and the proposed approach store voxel data in a uniform way as RLE data. According to [17], RLE is the second fastest algorithm to decode lossless compressed volume data.

The advantage of storing the data in a uniform way as opposed to categorizing the data into groups is that the data can be rendered using the same algorithm, which results in reducing the complexity.

The chapter is organized as follows. Section 4.4.5 outlines the proposed method. Section 4.4.6 explains the pre-processing. Section 4.5 elaborates on the rendering by the GPU, Section 4.6 evaluates the proposed method experimentally and Section 4.7 concludes this chapter.

#### 4.4.2 Improvements over Previous Work

The proposed method is an extended and improved version of the original work by Wright *et al.* The extensions and improvements are summarized as follows:

- The proposed method is completely optimized for highly parallel single instruction multiple data (SIMD) processing on the GPU and uses newest NVIDIA CUDA technology for the fastest possible visualization. The original method of Wright *et al.* cannot directly be applied to GPU efficiently as it is not optimized for parallel processing and also not aware of features and constraints of the GPU architecture.
  - The proposed method uses the GPU's shared memory to store a local one-bit-per-pixel visibility map, which could significantly improve the speed.
  - The proposed method uses the GPU's texturing technology together with the pixel shader to apply the fast unwrapping of the temporary buffer to the screen. The

---

<sup>57</sup> Silverman, Ken. Voxlap engine. 2003. <http://advsys.net/ken/voxlap.htm>.

original method projected one pixel after another onto the screen, causing holes that needed to be filled in multiple sampling steps. Using such a method on modern GPUs could cause many incoherent memory accesses, which is very inefficient.

- The proposed method uses the Digital Differential Analyzer algorithm for stepping through the RLE voxel data to achieve accurate intersections, which could significantly improve the visualization quality. Each voxel is therefore visualized correctly as a cube. The original method used equidistant sampling, which is more simple, but not as accurate.
- The proposed method uses an advanced floating horizon algorithm, which allows speeding up the rendering significantly compared to the original floating horizon algorithm. It is able to merge disconnected segments. The original method focused on height-map based terrains; therefore disconnected vertical segments were not as important.
- The original method stores only two colors for each vertical RLE element. The proposed method can accommodate up to 64 voxels in one element, which is a significant improvement for complex, dense textured, scenes. Therefore, fewer vertical segments for the same scene are required, which improves the performance and saves memory.
- The proposed method uses perspective correct texture mapping to achieve the vertical coloring of one RLE element. The original method only used two colors per element; therefore they did not require this feature.
- The proposed algorithm applies a post process filtering to the rendering result to remove jaggies at voxel boundaries for voxel close to the camera and create a smooth, yet well-defined silhouette. To further improve the quality, a second part of this filter also applies smoothing across the area enclosed by the smoothed silhouette, which leads to a better result. The original method did not provide this feature.
- The proposed method is able to recover normal vectors from the depth buffer in a post process. This saves memory as the normal vectors do not need to be stored along with the volume data. The original method did not provide this feature.

#### 4.4.3 Difference to Volume Rendering

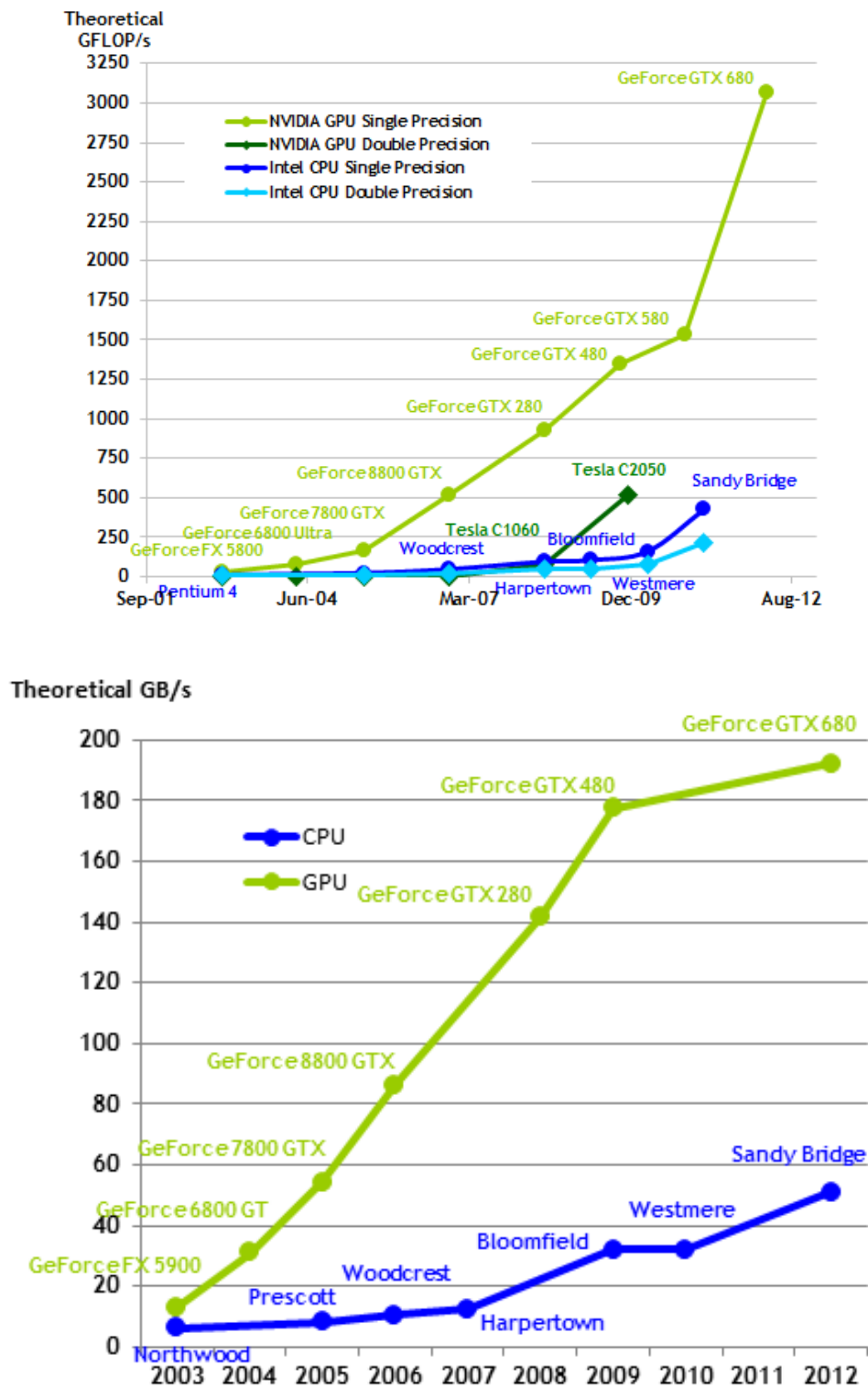
The proposed approach is very different from volume rendering methods, because the proposed method is focused on surface voxel data. Volume rendering methods commonly integrate color and opacity by tracing a ray through the semi-transparent volume data (such as an MRI image) for the final result. They do not directly visualize the surface as intended here. They further have much higher memory consumption as they not only store voxels that represent the surface but the entire solid data.

#### 4.4.4 Trends in CPU and GPU Development

To accelerate the rendering process, the proposed approach, for the first time, integrates the entire rendering algorithm on the GPU by using NVidia's CUDA and the Pixel Shader. The choice for the GPU rather than CPU for computations is clearly shown in **Fig. 4.7**, where performance of CPU and GPU over the past years is compared. It can be seen that the GPU develops much faster in terms of theoretical GFlops (left) and theoretical memory bandwidth as well. GPU can overcome the CPU's two bottlenecks: the floating point performance and the memory bandwidth. It can be seen that also in future versions of CPU and GPU, the GPU could remain faster in terms of floating point performance and memory bandwidth

Until recently, graphics hardware was incapable of supporting random writes, which are crucial for the proposed method. However, now it has become available with NVidia CUDA and OpenCL.



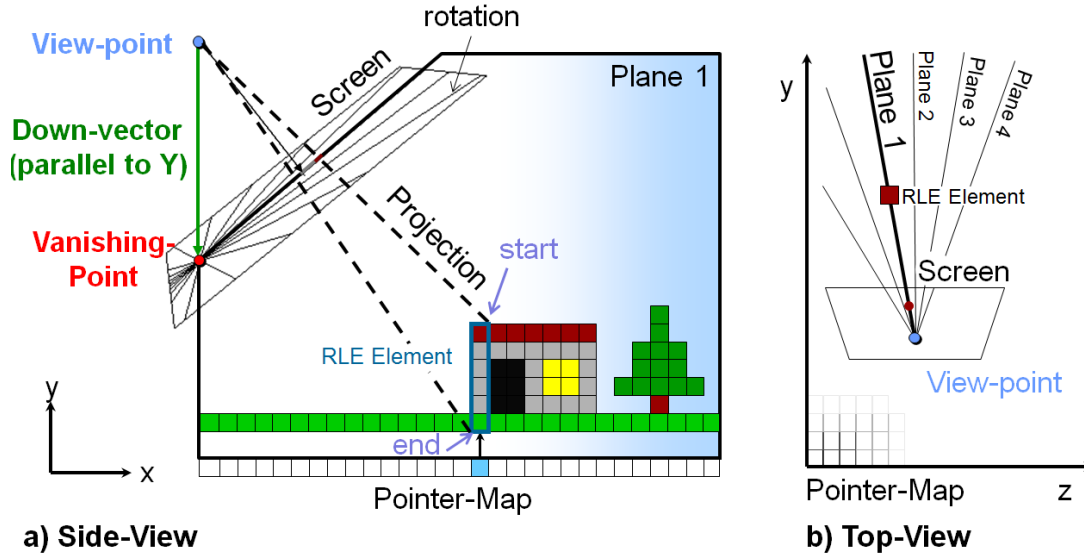


**Figure 4.7.** Hardware comparison: CPU vs GPU in terms of theoretical GFlops and theoretical memory bandwidth (source: NVidia<sup>58</sup>)

<sup>58</sup> NVIDIA, "CUDA Toolkit Programming," NVIDIA. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html> . 2013

#### 4.4.5 Details of the proposed Algorithm

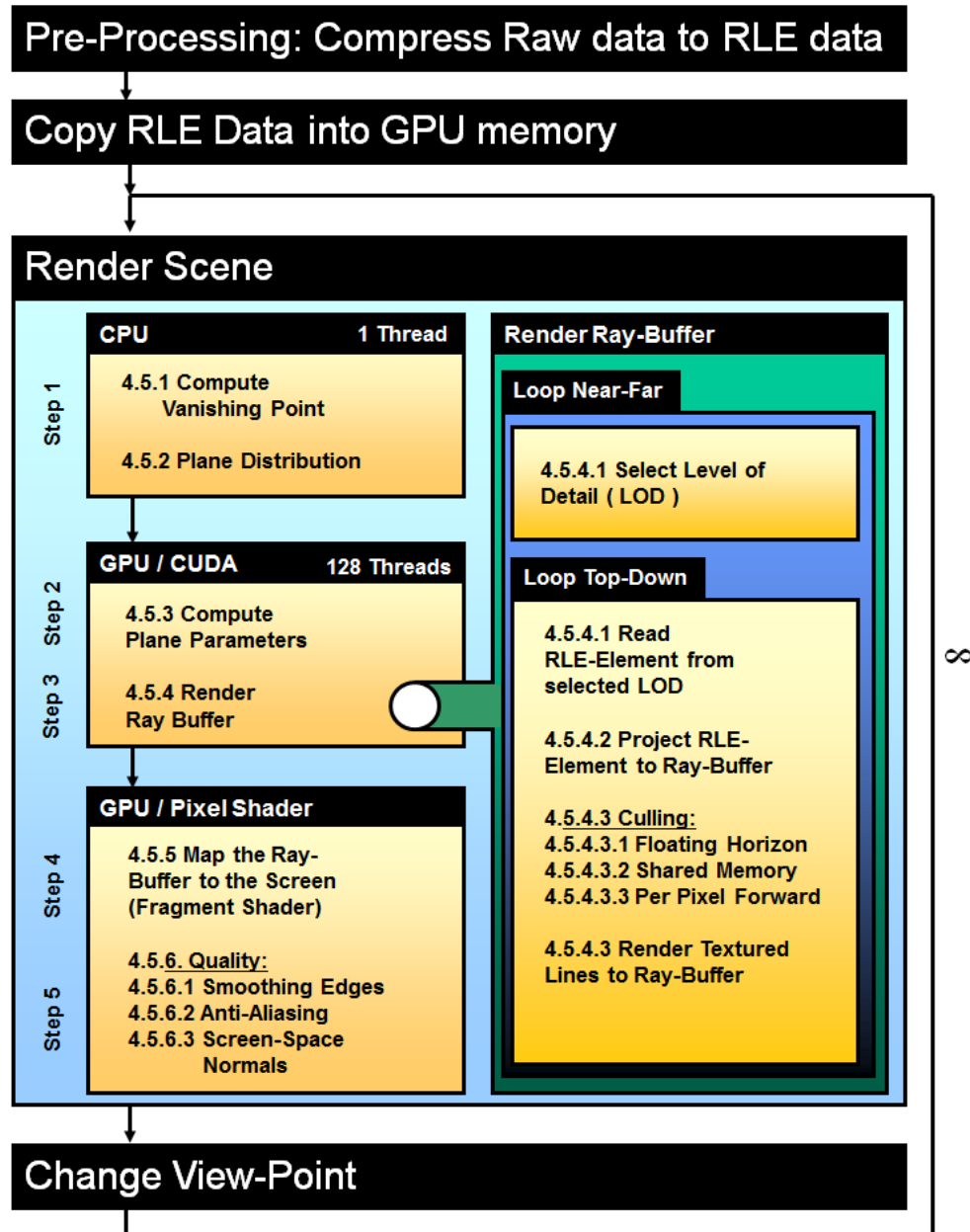
As shown in **Fig. 4.8**, the 3D surface voxel data exists in the x-y-z world coordinate system, where the x-z plane is the horizontal ground plane.



**Figure 4.8.** Proposed algorithm: (a) side view, (b) top-view.

As shown in **Fig. 4.9**, the algorithm consists of a series of steps, starting with the pre-processing step and ending with rendering the scene and changing the viewpoint. In the pre-processing step, the voxel data is run-length-encoded for each LOD in the vertical (y) direction. It is important that the encoding direction is vertical, because this leads to a higher average speed of the algorithm for the general case, when the camera looks towards the horizon. The details of this pre-processing step are described in Section 4.4.6. As shown in **Fig. 4.9**, after copying the RLE data to the GPU memory, the loop for visualizing the RLE data from the viewpoint at each time instant starts. As can be seen in **Fig. 4.8**, the proposed method visualizes the scene in planes that are perpendicular to the x-z plane and share the straight line that passes through the viewpoint and is parallel to the y-axis (Down-vector). Ray casting the RLE data in each concentric plane is done step-by-step from near to far along the x-z plane, while the rasterization is done for each step in the vertical direction (parallel to the y axis) from top to bottom. To be more specific, for each step in the x-z-plane, all the RLE elements in the corresponding column are rasterized by projecting them into the screen space. Since the projection of each concentric plane is a line slanted across the screen space, the results of rendering the planes are stored as temporary bitmap for performance reasons. The temporary bitmap is then mapped to the screen using the Pixel Shader. The render loop consists of the following five pipe-lined major steps referenced as 4.1 to 4.6 in **Fig. 4.9**, corresponding to this chapter's sections 4.5.1 through 4.5.6.





**Figure 4.9.** Pipeline for the proposed method: in “Render Scene” Section, numbers are indicated.

#### Step 1. Compute the vanishing point:

The vanishing point of all concentric plane’s around the downward vector is computed on the CPU. As shown in **Fig. 4.8**, the vanishing point  $vp$  (red dot) is the intersection between the screen plane and the Down-vector. The vanishing point needs to be computed first (Section 4.5.1), before the concentric planes are computed (Section 4.5.2).

**Step 2. Compute the concentric plane parameters on GPU:**

The parameters of each concentric plane, which are needed for the rendering process, are computed on the GPU (Section 4.5.3).

**Step 3. Render the planes on GPU:**

In each concentric plane, a ray is cast in the x-z plane from the x-z-coordinates of the viewpoint to the maximal view-distance. For each x-z-position, the corresponding column of all RLE elements is rasterized from top to bottom for the selected LOD (Section 4.5.4.1) at this distance. For each RLE element, the projection of the coordinates to the ray-buffer is performed first (Section 4.5.4.2). Then, culling is performed (Section 4.5.4.3). Finally, the element is rasterized as a textured line in the ray-buffer (a temporary bitmap) (Section 4.5.4.4).

**Step 4. Display the temporary bitmap on the screen:**

The GPU Pixel Shader is used to rearrange the rows of the temporary texture to a radial pattern of straight lines centered at the vanishing point on the screen (Section 4.5.5).

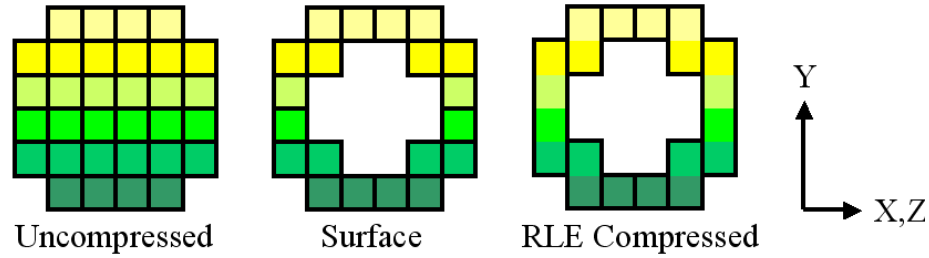
**Step 5. Improving quality:**

In the post-processing step, blocky appearance of voxels is reduced by smoothing the voxels. To improve the rendering quality, smoothing voxels is followed by anti-aliasing (Section 4.5.6).

In order to allow shading computations without storing normal vectors inside the RLE volume data, a special method recovers the normal vectors from the depth buffer (Section 4.5.6.3).

**4.4.6 Pre-Processing****4.4.6.1 Organization**

The original source data to be visualized can either be volume data or polygon data. In case of polygonal data, the voxelization is simply done by rasterizing each triangle is voxelized into a 3D regular grid of voxels. As described earlier, the voxelized data is compressed in the vertical (y-axis) direction from top (larger y coordinates) to bottom (smaller y coordinates) using run-length encoding (RLE). More specifically, each vertical RLE column is compressed separately and referenced by one pointer of a two-dimensional lattice placed in the x-z plane, where the scale-factor of the lattice for the x and z directions are normally uniform, respectively. As shown in **Fig. 4.10**, not all the voxels of a solid volumetric object is run-length-encoded. To reduce the memory consumption, only surface voxels are finally stored, while occluded inner voxels are removed. Section 4.4.6.2 elaborates on the specific data structure of the RLEed voxel data.

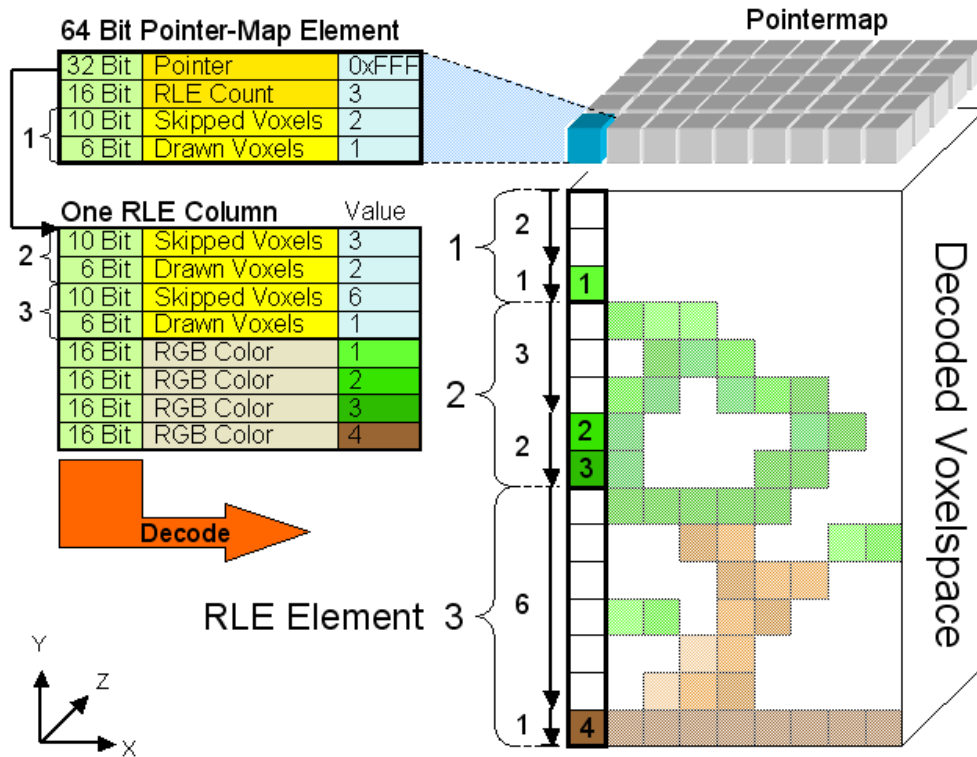


**Figure 4.10** Pre-Processing: The initial volume data (left), removal of non-surface voxels (middle), RLE compressed (right).

#### 4.4.6.2 Data Structure

The data structure of the voxel data should be able to utilize GPU performance as much as possible, and it is therefore optimized based on statistical evaluations of experimental results. As shown in **Fig. 4.11**, the entire data structure basically consists of two parts: the pointer map and the RLE columns. Each element of the pointer map (the lattice in the x-z plane) stores three different variables: the pointer to its corresponding RLE column buffer (described below) the number of RLE elements (defined below) included in that RLE column as well as the first (top-most) RLE element, consisting of “skipped voxels” and “drawn voxels”. An RLE element is defined as a set of two sequences; first a sequence of skipped voxels, and second a sequence of drawn voxels. A skipped voxel corresponds to an invisible, un-set, voxel that is not stored, and a drawn voxel corresponds to a voxel that is stored in the RLE structure with RGB color data. For example, in the decoded voxel-space illustrated in the right side in **Fig. 4.11**, white voxels indicate skipped voxels, and colored voxels indicate drawn voxels, respectively. In the left-most voxel column, the two voxels from the top are skipped (not drawn), and just below there is one (colored) drawn voxel. Therefore, “2” and “1” are stored in the “skipped voxels” field and “drawn voxels” field in the pointer map element of the RLE structure in the left side, respectively.

As shown in **Fig. 4.11**, each RLE column is referred to by a pointer of the pointer map. A referenced RLE column stores the numbers of skipped voxels and the number of drawn voxels starting from the second RLE element. The first RLE element is stored inside the pointer map. In addition, the buffer for RLE column stores the color for each drawn voxel in the order of the voxels’ appearance in the RLE column. To achieve efficient computation by GPU, the number of memory accesses has to be minimized. 64 bit elements are therefore stored in the pointer-map, as 64 bit is the largest amount of memory that can be pulled in one read by the NVIDIA GPU used by this thesis. Note that one 64-bit element includes all the data required to test the visibility of the first (topmost) RLE element. This strategy increases the rendering performance (speed) particularly for large outdoor environments and landscape-like scenes with hills and mountains. This is because one memory read is sufficient to test the visibility for approximately 90% of all rasterized elements according to preliminary studies.



**Figure 4.11** Data structure:Pointer-map: For each pointer-map element's data, one RLE column is pointed by a pointer and decoded by RLE .

#### 4.4.7 Level-of-Detail Computation

As described previously, the individual RLE data for each level of detail is obtained in advance prior to the visualization process. The idea of texture mip-maps is applied to the original RLEed voxel data and generate RLEed mip-volumes<sup>59</sup>.

The original RLEed voxel data has the highest resolution and is used for the LOD that corresponds to the range closest to the view point. As the distance from the viewpoint gets larger, RLEed voxel data with lower resolutions are used. More specifically, suppose that  $lev$  denotes a level of detail, where  $lev$  ranges from 1 (highest resolution) to  $L$  (lowest resolution); the size (length of a side) of one voxel in the level  $lev$  (2) is twice as long as that in the level  $lev-1$ , where linear down-sampling is applied to the voxel data in the level  $lev-1$  so that the voxel data in the level  $lev$  is obtained. For example, an original volume of  $16 \times 16 \times 16$  has four mip-volumes:  $8 \times 8 \times 8$ ,  $4 \times 4 \times 4$ ,  $2 \times 2 \times 2$ , and  $1 \times 1 \times 1$ . As described in the following, the resolution is dynamically chosen by the visualization process, depending on the distance to the viewpoint.

<sup>59</sup> A mip-map is a lower resolution copy of an original image. For a mip-map, a number of downsampled copies of the original image are created, each representing one level of detail. The exact number of these copies is depends on the pixel size (width, height) of the original image. For the visualization, high resolution images are used near the view point, and lower resolution copies of the original are used for distant visualizations to save memory bandwidth [64]. Mip-volumes are the three dimensional extension of mip-maps.

## 4.5 Rendering

The rendering for each frame consists of multiple steps, as displayed in **Fig. 4.9** and described in Section 4.4.5.

### 4.5.1 Vanishing-Point

The vanishing point  $vp$ , the point at which all the concentric planes meet in the screen plane (see **Fig. 4.8**), is computed first. Each plane is projected to the screen as one straight line and all the lines meet at  $vp$ . The vanishing point can easily be obtained by intersecting the vertical line that is parallel to the y-axis and passes through the viewpoint with the screen-plane as follows:

$$vp = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \cdot \frac{-d}{\sin(\alpha_p)}, \quad (4.2)$$

where  $d$  denotes the distance between the camera origin (view point) and screen-plane, and  $\alpha_p$  represents the camera's pitch angle, which is defined as the rotation around the horizontal axis (the x-axis) of the camera coordinate system. A pitch angle of zero means that the optical axis of the camera is horizontal. The vanishing point is projected to the screen space by the following equation:

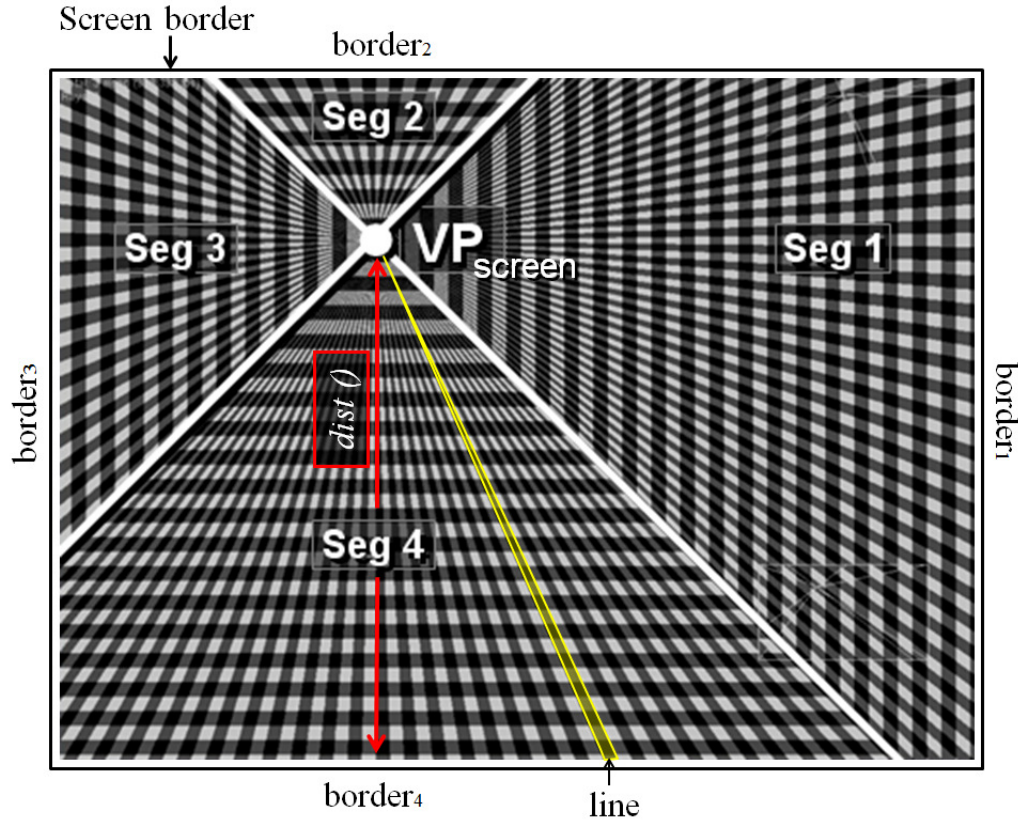
$$vp_{screen} = A_{cam} \cdot vp, \quad (4.3)$$

where  $vp_{screen}$  represents the projection of  $vp$  to the screen space,  $A_{cam}$  represents the  $4 \times 4$  camera matrix. Each plane intersects the screen as one line originated in  $vp_{screen}$  (**Fig. 4.8**).

### 4.5.2 Concentric Planes

Since each plane is projected to the screen as one line that is originated in  $vp_{screen}$ , achieving a complete coverage of the screen by the lines originated in  $vp_{screen}$  is essential. To achieve this, as shown in **Fig. 4.12**, the screen is partitioned into four segments, where the borderlines between adjacent segments meet at  $vp_{screen}$ , and the angle between adjacent borders is 90 degrees. Each line included in the left and right (with respect to  $vp_{screen}$ ) segments is textured in the horizontal direction, while the upper and lower segments are textured in the vertical direction. The number of lines included in each segment depends on the number of pixels on the screen border in this particular segment. This implies that each pixel in the screen border of a segment should be the end of a line (projected plane), whose another end is  $vp_{screen}$ . The number of planes (lines) can be calculated as follows:

$$np_i = 2 \cdot dist(vp_{screen}, border_i), \quad i \in [1 \dots 4], \quad (4.4)$$



**Figure 4.12** Screen segmentation: VP represents the vanishing point; Seg 1 to 4 refer to segments 1 to 4 respectively.

where  $np_i$  denotes the number of planes for a given  $segment_i$ ,  $border_i$  denotes one of the four borders of the screen, and  $dist()$  indicates the computation of the distance in pixels between  $vp_{screen}$  and  $border_i$ . The parameters  $vp_{screen}$  and  $np_i$ , which are computed by CPU, are transferred to GPU for the subsequent computations.

### 4.5.3 Plane Parameters

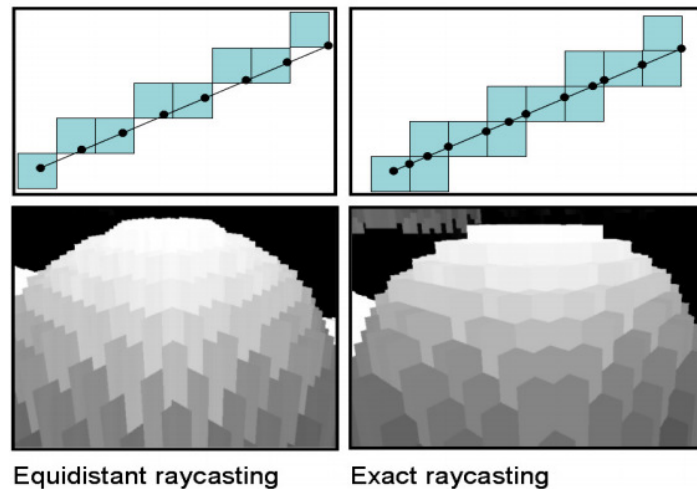
As described in Section 4.4.5, all the calculations described in the rest of Section 4.5 are executed on GPU in a parallel manner by using multiple threads. The number of simultaneous running threads depends on the number of processing units (GPU *cores*) of the underlying hardware. In this case, 240 processing units are available. The parameters to be computed for each plane are as follows (**Fig. 4.8**): the start and end point's (x,y,z) coordinates of the projected line in the screen and the plane's rotation around the y-axis. The start and end points are used for rendering and clipping the projected RLE elements to the screen. The rotation around the y-axis defines the orientation in which marching through the RLE structure is performed (Section 4.5.4).

#### 4.5.4 Rasterizing the Ray Buffer

The RLE elements are visualized in two steps. In the first step elements are rasterized to a 2D temporary ray-buffer, each row of which stores the projected result of one concentric plane. In the second step, the temporary ray-buffer's contents are texture-mapped to the screen.

##### 4.5.4.1 Traversal per Plane

To rasterize the RLE elements to the temporary ray-buffer, the pointer-map is traversed. The map is placed in the x-z plane as shown in **Fig. 4.8** and **Fig. 4.11**. As shown in **Fig. 4.8**, the straight line in which a concentric plane and the pointer-map (x-z plane) meet is considered. For a point (an element of the pointer-map) on the straight line, the RLE elements (voxels) visible from the viewpoint are rasterized in the radial line in which the concentric plane and the screen meet. This process starts from the point just below the viewpoint and traverses the pointer-map in the x-z plane till it reaches the point that corresponds to the predefined maximal distance from the viewpoint. During this traversal, culling, which is explained in Section 4.5.5, is performed for the visibility check. The traversal is not equidistant as it is often done in volume visualization. As shown in **Fig. 4.13**, equidistant traversal performs equidistant sampling of the pointer-map's elements on the straight line.



**Figure 4.13** Equidistant and exact raycasting: Left: Equidistant; Right: Exact raycast; Upper: sampling; Lower: Example of rendering.

This is simple, but leads to errors in the visualization. Instead, an exact grid traversal is applied, which correctly samples all the 2D grid intersections during the traversal according to [40]. In **Fig. 4.13**, the visualization results of the exact traversal and the equidistant traversal are compared. The exact traversal requires slightly more computational effort, but the result is significantly better. During the above-mentioned traversal, LOD needs to be switched according to the distance from the viewpoint. The LOD is selected according to the distance between the viewpoint and a point on the line in which the concentric plane and the x-z plane meet. Suppose  $pd$  is a predefined

distance along the line. From  $p_0$ , the point below the view point, to  $p_1$ , which is away from  $p_0$  by  $pd$  on the line, the RLE data (voxels) with the highest resolution is used for the rasterization; similarly, from  $p_1$  to  $p_2$ , which is away from  $p_1$  by  $pd$ , the second highest resolution is used, etc.



#### 4.5.4.2 Projecting RLE Elements to Ray-Buffer

As mentioned earlier, the visible part of each RLE element is rasterized to the temporary buffer as a textured line, where the  $x$ ,  $y$  and  $z$  coordinates of the start-point  $ps$  and the end-point  $pe$  are the 3D world space coordinates of the particular RLE element. The points  $ps$  and  $pe$  are projected into the screen-space using the camera matrix  $A_{cam}$  as follows:

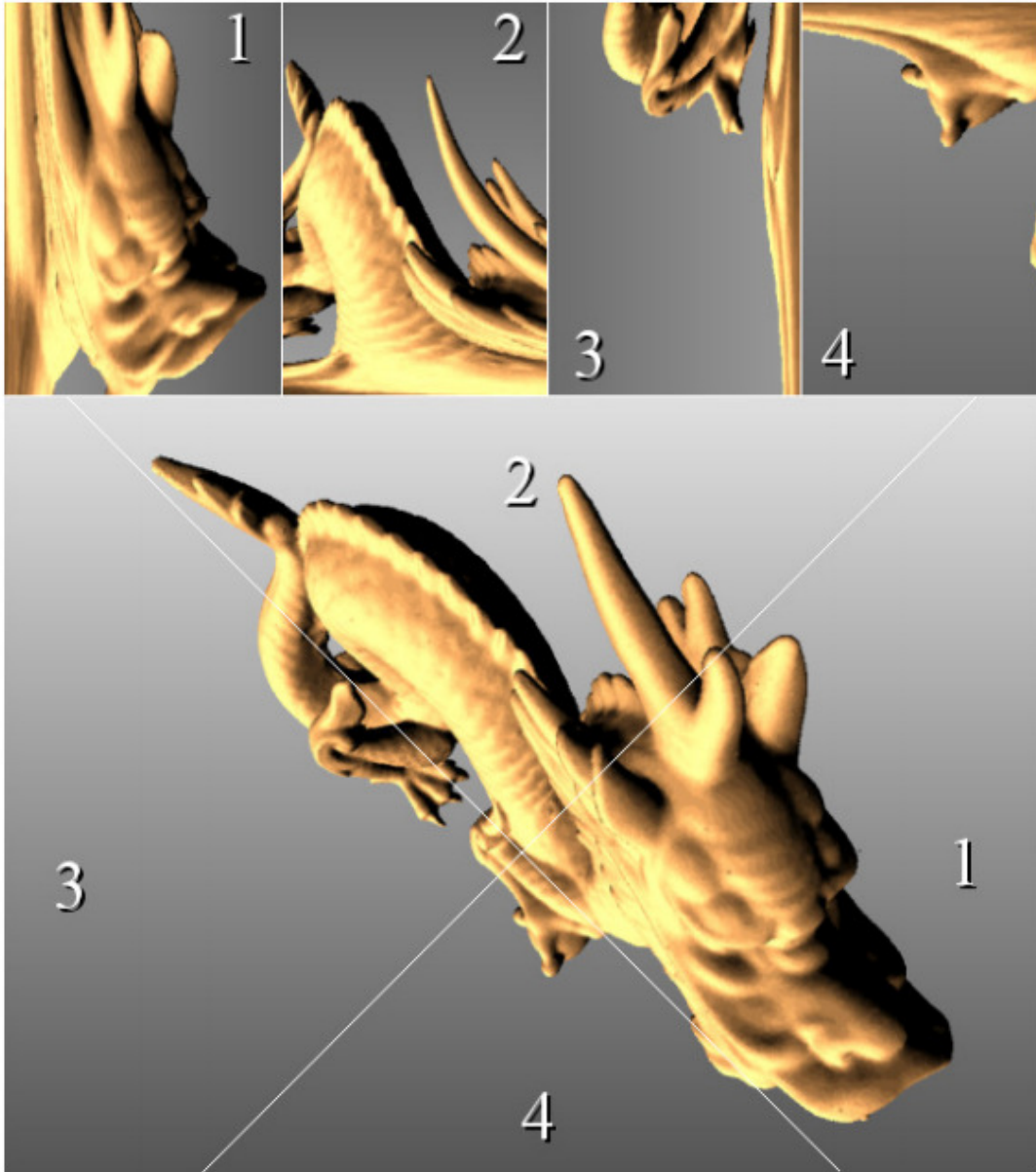
$$\begin{aligned} ps_{cam} &= A_{cam} \cdot ps, \\ pe_{cam} &= A_{cam} \cdot pe, \\ ps_{screen} &= \begin{bmatrix} ps_{cam} \cdot x \\ ps_{cam} \cdot y \end{bmatrix} \cdot \frac{1}{ps_{cam} \cdot z}, \\ b_o(x) &= \begin{bmatrix} pe_{cam} \cdot x \\ pe_{cam} \cdot y \end{bmatrix} \cdot \frac{1}{pe_{cam} \cdot z}. \end{aligned} \quad (4.5)$$

In Eq.(4.5),  $ps_{cam}$  and  $pe_{cam}$  contain the  $x$ ,  $y$ , and  $z$  coordinates of the  $ps$  and  $pe$  in the camera space. The camera space is defined as orthonormal-basis, where the origin is placed at the view-point, the  $z$ -axis a straight line from the viewpoint towards the center of the screen, the  $x$ -axis a straight line towards the origin and parallel to the upper and lower screen border and the  $y$ -axis a straight line towards the origin and parallel to the left and right screen border. The variables  $ps_{screen}$  and  $pe_{screen}$  are the two dimensional ray-buffer coordinates of  $ps$  and  $pe$ . As described in Section 4.5.2, either the horizontal ( $x$ ) or vertical ( $y$ ) component of the start and end coordinates is used for rasterizing RLE elements into the ray-buffer. In the ray-buffer, the projection of each plane is represented as one column, as shown in the upper half of **Fig. 4.14**.

Therefore, either the horizontal ( $x$ ) or vertical ( $y$ ) coordinates of the start and end-point are used to define the vertical 1D position inside the column of the ray-buffer. In **Fig. 4.14**, Segments 1 and 3 use the horizontal ( $x$ ) coordinate, while Segment 2 and 4 use the vertical ( $y$ ) coordinate of  $ps_{screen}$  and  $pe_{screen}$ . After the start and end positions inside the column are determined, visibility culling is performed (detailed in Section 4.5.4.3), before the textured rasterization is done (Section 4.5.4.4).

#### 4.5.4.3 Culling

As described in Section 4.5, culling needs to be performed to render only the visible parts of RLE elements and efficiently skip RLE elements that are invisible. In this work three culling methods are used, including novel and known methods. It is possible to combine these culling methods for optimal performance. However, utilizing all the algorithms simultaneously is not efficient due to mutual interference. It is efficient to use the floating horizon algorithm together with shared memory culling or per pixel forwarding. However, shared memory culling and per pixel forwarding interfere, because they are both executed on a per-pixel-level.



**Figure 4.14** Ray mapping: 1 to 4 denote segments 1 to 4; Upper: The temporary buffer with the four segments; Lower: mapping to the screen.

#### 4.5.4.3.1 Modified Floating Horizon

The well-known floating horizon algorithm, which was used in the original voxel forward projection algorithm [39], is utilized also here. The floating horizon algorithm does not conflict with the other two used culling methods and can hence be used in combination with them. The algorithm works as follows.

For each rendered plane, two offset values the start and end-offset along the projected line in the screen define the bounds of the render-able area and are stored. Once one RLE element that

touches the start or end offset is drawn, this particular offset is updated to narrow the bounding area along the line, which allows to cull more RLE elements.

Using the floating horizon algorithm is possible, because opaque scenes are rendered from near to far, which means that every pixel is drawn only once. However, the basic floating horizon algorithm works well only for height-map based scenes such as mountains. In case of complex scenes such as a tree, unconnected segments rasterized along the line cannot be handled efficiently by the original algorithm. Therefore, a small but significant modification is added to the original method so that good performance is achieved even in complex scenes. The modification is as follows: after one RLE element is rasterized that touches either border, the offsets are updated to enclose this particular RLE element. Pixels next to the new offsets are further tested if they have been drawn already. If they have been drawn already, the bounds are narrowed to enclose these pixels too. Depending on the scene, this modification accelerates the culling process up to two times.

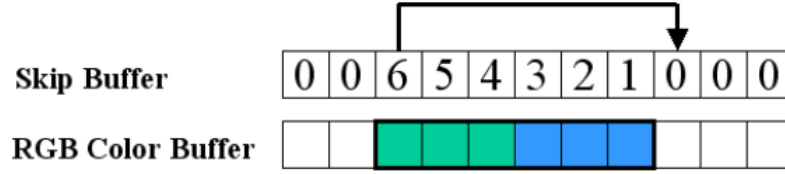
#### 4.5.4.3.2 Shared Memory

The shared-memory culling algorithm takes advantage of the fact that the proposed method draws every pixel in the screen only once. This means a binary map is sufficient to store the visibility information in the screen. This visibility map consumes little memory and therefore fits entirely into the graphic cards shared memory. The hardware used by this thesis, the NVidia GTX series, provides two main types of memory: Global memory and shared memory. The difference between both types is that a memory access to global memory consumes about 300 processor cycles, while an access to the shared memory only requires one cycle. Therefore, using a binary visibility map stored in the shared memory, per-pixel culling works very fast without accessing the slower global memory. Actually shared memory culling accelerates the rendering speed by 40% to 140%, depending on the scene compared to global memory.

#### 4.5.4.3.3 Per Pixel Forward

Lacroute's culling based on per-pixel forwarding [35] is slightly slower and more complex than the previously described shared memory culling, but it is needed for screen-resolutions where the number of simultaneously processed pixels of the screen exceeds the number of bits available in the shared memory. The shared memory is 16384KB in this thesis' case. Using 128 parallel threads leaves 128 bytes or 1024bit for using shared memory culling, which is reduced to effective 900bits due to shared memory reserved for program parameters. Each bit stores the visibility for one pixel. In case of the hardware that was used by this thesis, this happens at screen resolutions with more than 900 pixels in the vertical direction. The per-pixel forward algorithm works as follows: for each pixel in the temporary buffer a relative jump offset is stored. This offset is set to zero in the beginning and is updated to the next empty pixel once an RLE element is drawn as shown in **Fig. 4.15**. Each offset in the skip buffer points to the next free pixel. The RGB color buffer contains the colors of the visualized RLE elements. In this case, this thesis uses a blue and green example pattern, but it could be any other color too.

Since relative jumps help to skip pixels efficiently, a speed-up of approximately 1.08 to 2.0 times compared to not using skip pixels is achieved, which is significantly faster than the floating horizon algorithm alone, but approximately 20% slower compared to shared-memory culling.



**Figure 4.15** Skip-Buffer.

#### 4.5.4.4 Drawing RLE elements as textured Lines

Each RLE element is rasterized into one or multiple columns of the temporary ray buffer as a texture mapped line, using the coordinates of  $ps$  and  $pe$  as the vertical positions in the column. Using texture mapping the overall computation significantly speeds up, because voxels are rendered as a group rather than individually (the data structure is described in Section 4.4.6.2). To achieve a proper appearance, perspective correct texture mapping is applied. Simple non-perspective texture mapping interpolates the 2D texture coordinates, which leads to an approximated but inaccurate visual appearance. Perspective correct texture mapping uses not only the 2D texture coordinates but also the depth coordinate ( $z$ ), which leads to a correct result.

### 4.5.5 Displaying the Ray-Buffer

The texture stored in the temporary ray buffer can be efficiently be mapped to the screen by using the graphics card's Pixel-Shader. To achieve this, the source  $(U, V)$  texture coordinate is calculated in the ray buffer for each target pixel  $(xs, ys)$  on the screen. The mapping is applied in a concentric manner with respect to the vanishing point  $vp$  as shown in **Fig. 4.14**. The formula to compute the source  $(U, V)$  texture coordinates inside the ray-buffer is given by Eq.(4.6).

$$\begin{aligned}
 U_{2,4} &= (xs - vp.x) \cdot |ys - vp.y| + s_{2,4}, \\
 V_{2,4} &= ys - vp.y, \\
 U_{1,3} &= (ys - vp.y) \cdot |xs - vp.x| + s_{1,3}, \\
 V_{1,3} &= xs - vp.x.
 \end{aligned} \tag{4.6}$$

where  $U$  defines the horizontal coordinate inside the ray-buffer,  $V$  the vertical coordinate,  $xs$  the horizontal screen coordinate,  $ys$  the vertical screen coordinate and  $s$  the start-offset that is added for the corresponding segment of the ray-map. The indices of  $U, V$  and  $s$  represent the segment index as numbered in **Fig. 4.13**. The valid range of the texture coordinates  $(U, V)$  as well as the screen coordinates  $(xs, ys)$  ranges from 0 to 1.

### 4.5.6 Quality Aspects

As shown in the flow-chart of **Fig. 4.9**, the quality of the image rendered in the screen is improved at the final stage of the rendering pipeline. Since conventional texture mapping functions of the graphics card are used, texture filtering, which is natively supported by every GPU, can be applied without any performance impact. Two methods are employed to improve quality: smoothing and anti-aliasing. The combination of both algorithms, smoothing and anti-aliasing, can significantly improve the rendered image quality.

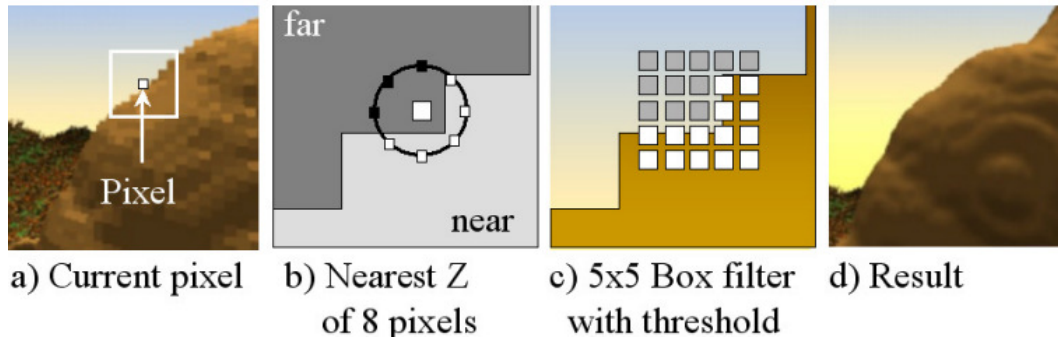
#### 4.5.6.1 Smoothing

Smoothing is applied as a post-process in image-space by the Pixel Shader, where a special smoothing method achieves two types of smoothing in one shader pass: Smoothing of voxel silhouettes and smoothing of voxels close to the camera. **Figure 4.16** shows an example of the result of this method.



**Figure 4.16** Smoothing results: Left: without smoothing; middle: smoothed silhouette; right: smoothed interior part

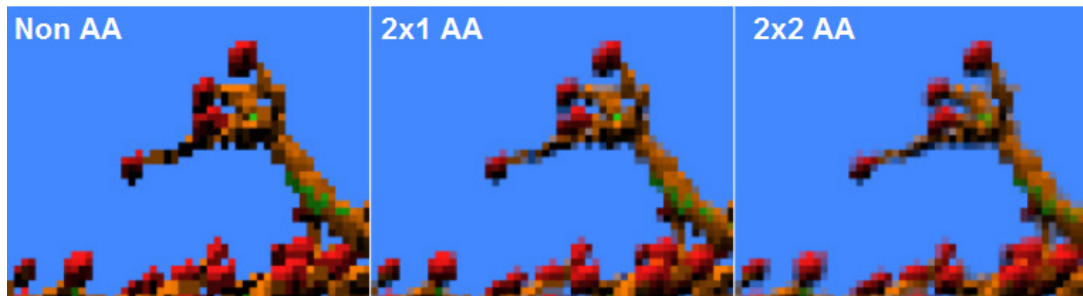
The smoothing consists of multiple steps, as illustrated in **Fig. 4.17**. Step a) shows the target pixel in the original image. In step b), the minimum depth of eight pixels that lies in a circle around the target pixel is searched. The radius is fixed for this operation. In step c) a box filter for  $5 \times 5$  pixels is obtained, where the scale factor of the box filter is determined by the previously obtained minimum-depth. For the smoothing, only pixels, whose depth values are close to the minimum depth, are averaged. Step d) shows the result, which demonstrates both the silhouette and the inner region in the example are smoothed well.



**Figure 4.17** Smoothing steps: a) Target pixel; b) Find minimum depth (Z); c) Box-filter with threshold, scaled according to the minimum depth; d) Result.

#### 4.5.6.2 Anti-Aliasing

For further improvement of the quality, full-screen anti-aliasing (AA) by rendering the scene with a higher resolution and down-sampling the rendered image is applied so as to obtain the target resolution. **Figure 4.18** compares three configurations: without AA (Non-AA),  $2 \times 1$  pixel AA ( $2 \times 1$  AA) and  $2 \times 2$  pixel AA ( $2 \times 2$  AA). For  $2 \times 1$  AA, two horizontal pixel are averaged to one pixel in the visualized image. For  $2 \times 2$  AA, two by two rendered pixel are averaged to one pixel in the visualized image. Obviously,  $2 \times 2$  pixels AA and  $2 \times 1$  pixels AA give the best and second best quality, respectively.



**Figure 4.18** Anti-aliasing (AA): left: Non AA; middle: 2x1 AA; right: 2x2 pixel AA.

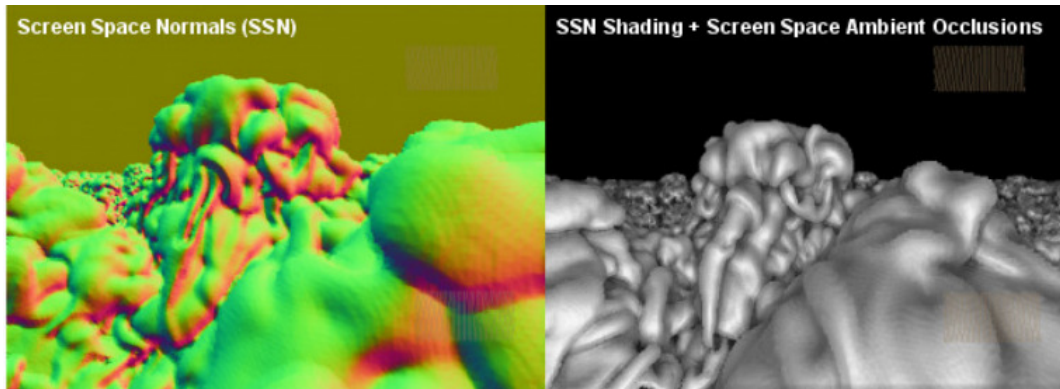
#### 4.5.6.3 Screen Space Normals (SSN)

To visualize large data sets such as the Richtmyer-Meshkov on consumer graphics cards with only 256MB RAM, storing all the surface normal needed for shading is heavy burden for the computation. Instead the surface normal  $n$  for shading can be approximated from a few samples in the depth buffer by Eq.(4.7).

$$\begin{aligned}
 z_s &= \text{Depth}(x_s, y_s), \\
 \Delta x &= x_s - \text{rnd}(1/z_s), \\
 \Delta y &= y_s - \text{rnd}(1/z_s), \\
 dz_x &= z_s - \frac{\text{Depth}(x_s + \Delta x, y_s + \Delta y)}{\Delta x}, \\
 dz_y &= z_s - \frac{\text{Depth}(x_s + \Delta x, y_s + \Delta y)}{\Delta y}, \\
 n &= (1 \ 0 \ dz_x) \times (1 \ 0 \ dz_y);
 \end{aligned} \tag{4.7}$$

where  $x_s$ , and  $y_s$  represent the horizontal and vertical coordinates of a pixel in the screen, respectively;  $\text{Depth}(\cdot)$  represents the depth of the pixel (argument) in the depth-buffer;  $\text{rnd}$  is the random function to achieve an averaged result for multiple samples; and operator  $\times$  for computing  $n$  is the standard vector cross-product.

Note that Eq.(4.7) indicates that the sample region needs to be reciprocal in size to the sampled depth value  $z_s$  of the pixel ( $x_s$  and  $y_s$ ). In case the pixel is close to the camera, a large region is needed and vice versa. To achieve a satisfying result in the experiments, at least 16 samples from the depth-buffer should be used. Since computing a random value by GPU is slow, a random value is sampled from a texture instead. As SSN and SSAO [41] sample the depth-buffer in a similar way, it is possible to efficiently combine both methods in only one shader-pass. An example of the result is demonstrated in **Fig. 4.19**.



**Figure 4.19** Normals: The depth-buffer can successfully be utilized to compute normal vectors on-the-fly (Left). These can be utilized for shading and further enhanced with screen space ambient occlusions (Right).



## 4.6 Experimental Results

### 4.6.1 Experimental Conditions

Experiments with multiple scenes are conducted to evaluate the proposed algorithm in terms of rendering speed, memory consumption, and quality aspects. The scenes used for the experiments are shown in **Fig. 4.20**.

The experimental system consists of a Pentium-D 3.0 GHz Processor with 1 GB of RAM and a GeForce285 GTX (1024MB) graphics board with 240 stream processors. As shown in **Fig. 4.9**, NVidia CUDA is used to compute the ray casting part of the algorithm, while texture mapping the temporary ray-buffer and the post-processing are executed in the Pixel-Shader. The render-resolution for all the tests is set to  $1024 \times 768$  pixels, while the AA setting for improving quality is  $2 \times 1$ , which provides the best tradeoff between quality and performance.

### 4.6.2 Memory Consumption

**Table 4.1** shows the result of benchmark tests, for the six  $1024 \times 768$  pixel scenes shown in **Fig. 4.20**, where bits per voxel indicates the number of bits required for storing the position information of one voxel, taking the pointer-map and mip-maps into account as well. The bits used to store the position of one voxel range from 10.83 to 26.3, which is significantly less than a pointer-based octree, which requires 32 bits only for the tree leaves, and sums up to about  $32 \times (1 + 1/8 + 1/64 + \dots) = 36.8$  bits for the entire tree.

However, in some scenes the proposed algorithm requires more memory than splatting-based algorithms such as QSplat, which only utilizes 13 bits per leaf. As described earlier, the accuracy of splatting-based methods is limited to the size of the splats; therefore, in particular, unreasonably sharp edges tend to degrade the image quality.

**Table 4.1** Benchmark Tests: The RLE element count in the frustum (Total), the processed element count (Proc) and the rendered element count in million (Ren). The resolution is stated in voxel. Further, Fps denotes frames per second and Speed is given in million RLE elements per second (Elems/s)

Scene	RLE Elemtes (in million)			Fps	Speed	Resolution	Compression
1024x768, 2x1 AA	Total	Proc	Ren		Elems/s	x/y/z	Bit/Voxel
Procedural	14.1	7.7	0.58	47.5	365.75	1k/1k/1k	17.9
Bonsai	8.8	8.7	0.35	21.7	188.79	512/512/512	18.25
Buddha	31.3	7.38	0.4	48.2	355.72	1k/2k/1k	12
Mansion	11.48	3.1	0.43	78.5	243.35	1k/256/1k	10.83
Dragon	2.67	1.67	0.29	67.1	112.06	1k/1k/1k	26.3
Bunny	1.34	1.34	0.24	68.2	91.388	1k/1k/1k	22.89





**Figure 4.20** Scenes used for tests: Handcrafted mansion (upper-left), Bonsai forest with 3000 trees (upper-right), Hotei, or Happy Buddha, (middle left) and a Procedural Landscape with about 4000 visible trees (middle right), the Stanford Dragon (lower-left) and the Stanford Bunny (lower-right). Unit for the number is given in voxels.

#### 4.6.3 Algorithm Speed

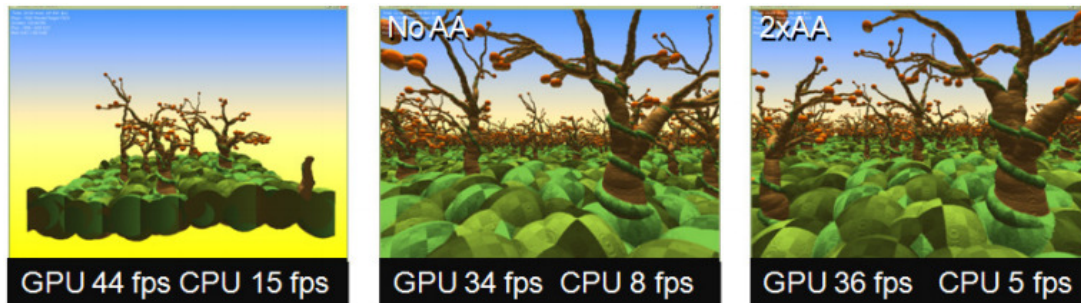
To measure and evaluate the rendering speed, the maximum polygon performance of the graphic card used in this thesis was determined first. In case of rendering as a quad by two textured triangles, rendering speed of 350 Million triangles per second is the limit of the graphic card for rendering triangle strips. However splatting-based rendering reaches 100 Million primitives (splats) per second. **Table 4.1** shows that the proposed algorithm achieves a high count of processed RLE elements per second (Speed, Elems/s); i.e. ranging from 91 to 365.8 Million RLE elements per second. This speed even outperforms the default OpenGL rendering pipeline, whose rendering speed is up to 350 Million disconnected triangles/s.

Further information in **Table 4.1** includes the total number of RLE elements inside the view frustum (RLE Elem total), the number of RLE Elements that have passed the culling test (RLE Elements, ren), frames per second (fps) and the resolution in voxels for the instance single of each dataset (Resolution).

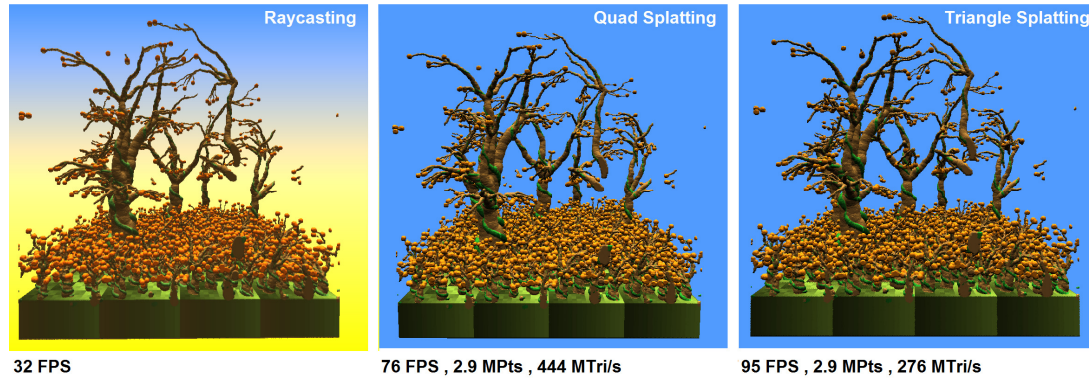
For testing the performance in case of large outdoor areas, scenes containing more than 1000 instances of the same data set are created for the procedural scene and the bonsai scene. The maximal view distance is set to 40000 voxel in both cases.

The results for the large outdoor scenes of the bonsai and the procedural dataset are also listed in **Table 4.1**. They were included in the tests, to evaluate the performance, the compression ratio and the quality as well. The results show that the procedural dataset achieves the highest performance in RLE elements per second. The bonsai data-set achieves not the same high performance as the scene is not suited well for the used culling algorithms.

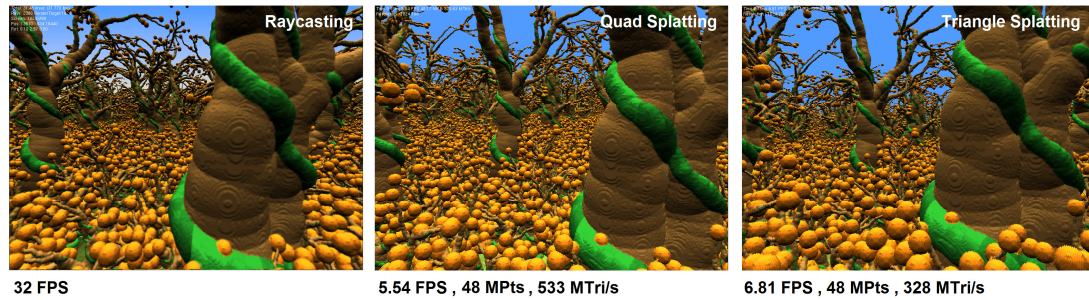
To compare the GPU performance with the CPU performance, the proposed method was executed on the CPU as well. As a result, it turns out that the GPU version, tested on an NVidia GeForce 285, is three to seven times as fast as the CPU version, executed on a test system with an Intel Core2 Quad Q6600 CPU with four cores running at 3 Ghz each and 1GB RAM. The GPU outperformed the CPU by factor of three for simple scenes without AA and factor of seven for complex scenes, with AA enabled. The scenes used for testing are shown in **Fig. 4.21**.



**Figure 4.21** GPU vs CPU: The GPU version running on an NVidia GTX 285 is compared to the CPU version (Intel Q6600 4x3Ghz).



**Figure 4.22** Raycasting vs Splatting (1): left: the proposed RLE method; middle: quad splatting; right: triangle splatting



**Figure 4.23** Raycasting vs Splatting (2): left: the proposed RLE method; middle: quad splatting; right: triangle splatting

For a comprehensive analysis, the proposed method is further compared to common splatting. The comparison was carried out in terms of speed, memory consumption and quality. To achieve fast splatting, each voxel of the voxel data was stored as one splat with position data (three float values for  $x$ ,  $y$  and  $z$ ) and RGB color data. Equal to the voxel data, also the splat data contains multiple levels of details. For the data-set that is used in this test, 12 Million splats are used for the highest level of detail. The complete data-set including all levels of details contains 16.3 Million splats, which requires 261.44 MB when stored as basic splats. The original RLE voxel data containing the equal number of voxels requires only 49.5 MB.

The array of splats is stored on the GPU as vertex buffer object (VBO) for maximal performance. To visualize the splats, the vertex data is sent to OpenGL as vertex array of `GL_POINTS`, and then converted into triangles or quads by the Geometry Shader. A quad consists of two triangles in this case. The hardware for this test was a NVIDIA GTX 580M GPU with an Intel Core i7 CPU and 16GB of RAM.

For rendering a single copy of the  $1024 \times 1024 \times 1024$  voxel data-set, 32 fps are achieved by the proposed method, 76 fps for quad based splatting and 95 fps for triangle based splatting. The results are shown in **Fig. 4.22**. For the two splatting based methods, 2.9 Million splats were

required to visualize the scene. For visualizing 1600 (40 times 40) instances of the same data-set, 32 fps are achieved for the proposed method, 5.5 fps for quad based splatting and 6.8 fps for triangle based splatting. The results are shown in **Fig. 4.23**. For the two splatting based methods, 48 Million splats were required to visualize the scene.

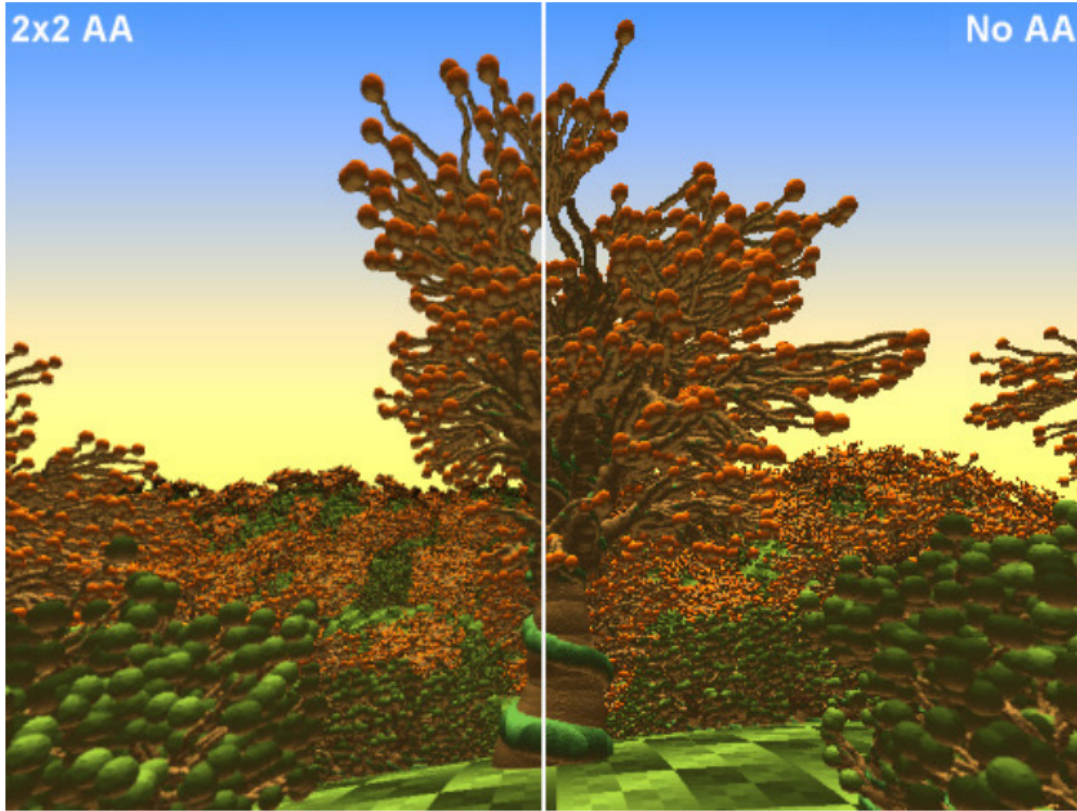
As a result, it turns out that splatting suits well for visualizing single objects, where it is up to three times as fast as the proposed raycasting approach. However, it is much slower for very complex scenes. The proposed method achieves 4.7 times as fast as splatting for the test scene consisting of 1600 instances. The proposed method, therefore, scales better for visualizing complex scenes than conventional splatting.

#### 4.6.4 Rendering Quality

The rendering speed is evaluated in regard to the image quality by measuring the performance for different quality settings. No anti-aliasing,  $2 \times 1$  anti-aliasing and  $2 \times 2$  anti-aliasing (**Fig. 4.18**) were compared.

If the speed for the no anti-aliasing is 100%,  $2 \times 1$  AA and  $2 \times 2$  AA achieve 104% and approximately 80%, respectively. The increase in speed for  $2 \times 1$  AA might be caused by better coalescence for reads from GPU memory. On GPU, coalescent memory reads are very important for high performance. Non-coalescent reads are significantly slower. As a conclusion, half the GPU's processing units must be idle in case of the no AA configuration, because  $2 \times 1$  AA requires two times as many floating-point operations as no AA. As a result of this experiment, the main limiting factor of the proposed algorithm is the memory-bandwidth due to the following reasons. Every rendering algorithm's speed is either limited by the speed of the processing unit (here the GPU) or the speed of the memory. Here, the speed of the memory is the limitation. The memory bandwidth was reduced by employing multiple culling algorithms, but it still remains the limiting factor. To improve that, additional compression schemes to reduce the memory bandwidth might be helpful.





**Figure 4.24** Quality: To show the ability to render at high quality, a complex test scene with many fine details was created and rendered at  $512 \times 348$  pixel with  $2 \times 2$  AA as well as no AA for a comparison. Note that  $2 \times 2$  AA successfully removes aliasing artifacts for distant pixels.



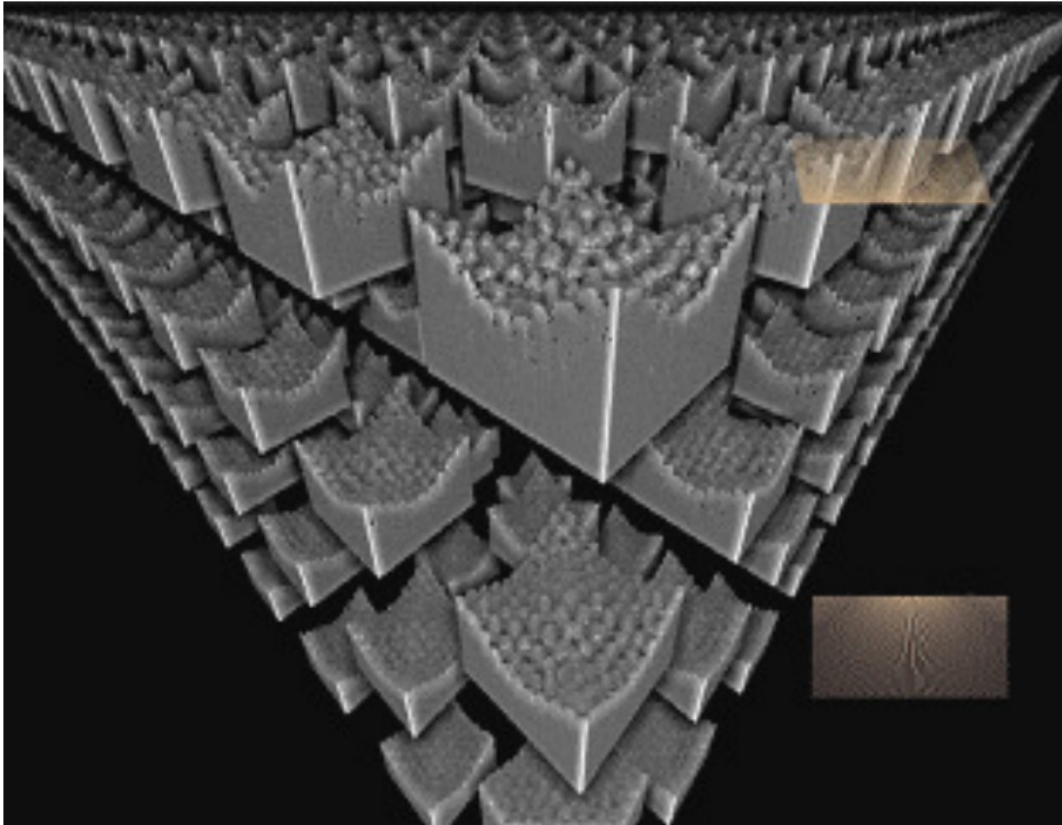
**Figure 4.25** Raycasting vs Splatting (3): render quality for geometry close to the camera; left: the proposed RLE method; middle: quad based splatting; right: triangle splatting

As shown in **Fig. 4.24**, the proposed algorithm is able to achieve high quality renderings for a scene with many fine structures. To facilitate the comparison, the result was rendered using  $2 \times 2$  AA in the left half and no AA in the right half.

Further analysis of the quality was carried out by comparing the previously introduced quad and triangle based splatting to the proposed method. As shown in **Fig. 4.25**, triangle based splats (right) achieve the lowest quality, as they are unable to approximate geometry close to the camera

in a proper manner; quad based splats (center) achieve a better approximation, but their silhouettes and other geometry close to the camera appear blocky and not well defined. The proposed methods result (left in **Fig. 4.25**) achieves the best quality: namely, as silhouettes are smooth and yet opaque. Furthermore, geometry enclosed by silhouettes close to the camera is smooth and looks similar to the result of texture filtering, which is commonly used in the visualization of textured 3D models.

Finally the Richtmyer-Meshkov data set was visualized, with a resolution of  $2048 \times 1920 \times 2048$ . The size of the RLE compressed data of the surface at iso-value 60 is 198 MB including mip-maps. This results in a compression factor of 5:1 in regard to the binary volume data. As this particular data set is very large, no color or shading information was stored along with the voxel data. The surface normal vectors were computed on-the-fly from the screen-space for the visualization, as well as approximated ambient occlusions. For the visualization speed at a resolution of  $1024 \times 768$ , interactive frame-rates were achieved: 15 fps for rendering a single instance of the data-set Fig. 4.21 and 10 fps for rendering the data-set repeatedly as shown in Fig. 4.26. It is possible to render the complete Richtmyer-Meshkov dataset more than 100 times. For the shading, a combination of screen-space-ambient-occlusion and screen-space normal was utilized.



**Figure 4.26** Richtmyer-Meshkov dataset

### 4.6.5 Comparison to Related Methods and Discussion

The proposed method was compared to existing methods in terms of rendering speed, memory consumption and visual precision.

#### 4.6.5.1 Memory Consumption

The proposed method is compared to a basic triangle mesh in **Table 4.2** for multiple scenes. The results shows that a triangle consumes about 9 times as much memory as a single voxel. Comparing the proposed method to GigaVoxels [37] (**Table 4.3**) shows that GigaVoxels uses about 4.8 times as much memory as the proposed method, considering 32 bit color depth for each voxel. Comparing the proposed method to QSplat [5] (**Fig. 4.27**) shows that Qsplat consumes 6 - 9 bytes per splat, which is similar to the proposed method with 4.7 – 6.8 bytes per voxel. For the file-size of the Buddha and the Dragon model, the proposed method's data structure is 1.64 - 1.75x as large as the data structure of Qsplat. The number of splats stored inside both Qsplat models remains unknown for the Buddha and the Dragon model though. The QSplat data values for the data size per splat for the comparison are given by the original QSplat paper. Comparing the proposed method to Sparse Voxel Octree Raycasting method by Jon Olick [15] (**Fig. 4.27**) shows that the run-time structure for octree raycasting uses ~10x as much memory per voxel compared to the proposed method. Comparing the proposed method to GPU Triangle Raycasting, Karras et al, [42] (**Fig. 4.27**) shows that one triangle uses about 9 times as much memory as one voxel in terms of position data. However, additional memory is used by the acceleration structure for ray-tracing.

#### 4.6.5.2 Computational Speed

For the computational speed, the test system was an Intel Core i7-2670QM CPU (2.2Ghz) with NVIDIA GeForce GTX 580M GPU and 16 GB of RAM.

The proposed method is compared to a basic triangle rasterization in **Fig. 4.28** on the test system. The result shows that the proposed method is faster in all cases. The result is further visualized as graph in **Fig. 4.29**, where an increased speed for higher distance can be observed. The proposed method further scales well for rendering complex scenes, as demonstrated in **Fig. 4.30**. While conventional rasterization achieves only 1 fps for visualizing the Imrod model 6 times, the proposed method achieves 30 fps for visualizing it 1600 times.

To compare this thesis' method in terms of speed and to the two voxel octree raycasting methods GigaVoxels by Crassin [37] and Sparse Voxel Octree raycasting by Olick [15], an exemplary sparse voxel octree raycasting method was implemented for this thesis. In **Fig. 4.31**, the exemplary sparse voxel octree raycasting method is compared to GigaVoxels for the San Miguel Scene. The performance values are the original ones stated in Crassin's PH.D thesis, page 114 [43]. The performance of GigaVoxels was measured on a NVIDIA GTX 480 graphics card, which is significantly faster than the test system used here:

- NVIDIA GTX480: (used for GigaVoxels)
- Floating point operations per second: 1.3TFlop/s, RAM memory bandwidth: 177GB/s<sup>60</sup>,
- 3DMark Score: 5810<sup>61</sup>, CL Raytrace Benchmark<sup>62</sup> Score: 136323 points
- NVIDIA GTX580M: (the test system used here)
- Floating point operations per second: 0.95TFlops, RAM memory bandwidth 96GB/s<sup>63</sup>,
- 3DMark Score: 3450, CL Raytrace Benchmark Score: 61154 points

The result shows that the exemplary sparse voxel octree raycasting method is at least as fast as the GigaVoxels, considering that the hardware used here is significantly slower in memory bandwidth and floating point computation speed than the hardware used to measure the performance of GigaVoxels. In **Fig. 4.31**, the exemplary sparse voxel octree raycasting method and GigaVoxels are further compared to the proposed method. The result shows that the proposed method is significantly slower than both methods for low screen resolutions and low voxel resolutions.

In **Fig. 4.32**, the exemplary sparse voxel octree raycasting method is compared to the sparse voxel octree raycasting method of Olick and to the proposed method. It turns out that the three compared methods achieve the same speed in this test using the Imrod model. The hardware for the exemplary sparse voxel octree raycasting method and the proposed method were this thesis' test system. The hardware used by Olick was an NVIDIA GTX 280 graphics card, which provides a similar performance as follows:

- NVIDIA GTX580M (test system used here):
- Floating point operations per second: 0.95 TFlops, RAM memory bandwidth 96GB/s
- NVIDIA GTX280 (hardware used by Olick):
- Floating point operations per second: 0.93 TFlops, RAM memory bandwidth 142GB/s<sup>64</sup>

In **Fig. 4.33**, the inner area of the San Miguel scene is compared for the exemplary sparse voxel octree raycasting method, the proposed method and to triangle raycasting for a high screen resolution,  $2048 \times 768$  pixel. The exemplary sparse voxel octree raycasting method is fastest for this scene (65 fps), whereas the proposed method is slightly slower (50 fps), and triangle raycasting comes last (26 fps).

In **Fig. 4.34**, the proposed method is compared to triangle raycasting and the exemplary sparse voxel octree raycasting method for the Imrod model. Here, all three methods are about the same

---

<sup>60</sup> [http://en.wikipedia.org/wiki/GeForce\\_400\\_Series](http://en.wikipedia.org/wiki/GeForce_400_Series)

<sup>61</sup> <http://community.futuremark.com/hardware/gpu/NVIDIA+GeForce+GTX+480/review>

<sup>62</sup> <http://clbenchmark.com>

<sup>63</sup> [http://en.wikipedia.org/wiki/GeForce\\_500\\_Series](http://en.wikipedia.org/wiki/GeForce_500_Series)

<sup>64</sup> [http://www.nvidia.com/docs/IO/55506/GeForce\\_GTX\\_200\\_GPU\\_Technical\\_Brief.pdf](http://www.nvidia.com/docs/IO/55506/GeForce_GTX_200_GPU_Technical_Brief.pdf) , page 11



speed, where the proposed method and the exemplary sparse voxel octree raycasting method are slightly faster (46 fps) than the triangle raycasting method (42 fps).

In **Fig. 4.35**, the proposed method is compared to triangle raycasting and the exemplary sparse voxel octree raycasting method for the Imrod model with a different camera setting. All three methods achieve about the same performance.

In **Fig. 4.36**, the proposed method is compared to the exemplary sparse voxel octree raycasting method for a very complex voxel scene containing multiple instances of the same tree dataset with 12.6 million voxels. For this scene, which is the most complex of all tested scenes due to numerous fine branches of the trees in the scene, the proposed method achieves 20 fps, which is 1.4 times as fast as the exemplary sparse voxel octree raycasting method with 14 fps. Therefore, the proposed method would also be significantly faster than GigaVoxels for this complex scene.

In **Fig. 4.37**, the proposed method is compared to QSplat in multiple configurations. The result shows that the proposed method outperforms QSplat for all test scenes by factor 1 - 3.7 with an average of 2.5.

#### 4.6.5.3 Visual Precision

All related methods are compared to the proposed method for visual precision as follows.

In **Fig. 4.38**, triangle based rasterization is compared to the proposed method for multiple camera configurations ranging from near to far. While there is a significantly higher precision for triangle based rasterization for close views, when the size of a voxel on the screen is greater one, the results are similar as far as one voxel is about the size of one pixel. Due to multiple anti-aliasing, triangle based rasterization achieves higher precision in all cases, though.

In **Fig. 4.39**, GigaVoxels is compared to the proposed method. GigaVoxels achieves higher quality for geometry close to the camera than the proposed method by using tri-linear texture filtering. For distant views, similar results are obtained. In **Fig. 4.39**, the upper dragon scene is sampled at  $2048 \times 2048 \times 2048$  voxel and the lower dragon scene at  $1024 \times 1024 \times 1024$ .

In **Fig. 4.40**, the Imrod model is compared to triangle raycasting for multiple views. While triangle raycasting achieves a higher precision for close views, similar results are obtained for far views, where the size of one voxel is equal to one pixel. For the test, the screen resolution was  $2048 \times 768$  pixel. For the triangle model, one million triangles were used. The voxel model was sampled at  $1024 \times 2048 \times 1024$  voxel and represented by 6.8 million voxel.

In **Fig. 4.41**, QSplat is compared to the proposed method for visualizing the entire Lucy model. In **Fig. 4.42**, QSplat is compared to the proposed method for visualizing close-up views of the Buddha model. For distant views as in **Fig. 4.41**, similar results are obtained for both methods. The proposed method renders more accurate than QSplat for close views in **Fig. 4.42**. Further, smoothing is supported by the proposed method, which is not supported by QSplat.

In **Fig. 4.43**, sparse voxel octree raycasting by Olick is compared to the proposed method. Octree raycasting visualizes each voxel as cube, which is equal to the proposed method. In addition, the proposed method supports smoothing. Shading is not considered for the comparison.

#### 4.6.5.4 Summary

This sub-section summarizes the comparison results presented in Sections 4.6.5.1 to 4.6.5.3.

In **Fig. 4.44**, the memory consumption is summarized for all related methods in relation to the proposed method. The proposed methods factor is set to one. It turns out that none of the related methods achieves a lower average memory consumption than the proposed method. Only QSplat is comparable to the proposed method in terms of memory consumption. GigaVoxels requires 4.8 times as much as the memory of the proposed method, and the other remaining methods more than nine times as much as the memory of the proposed method.

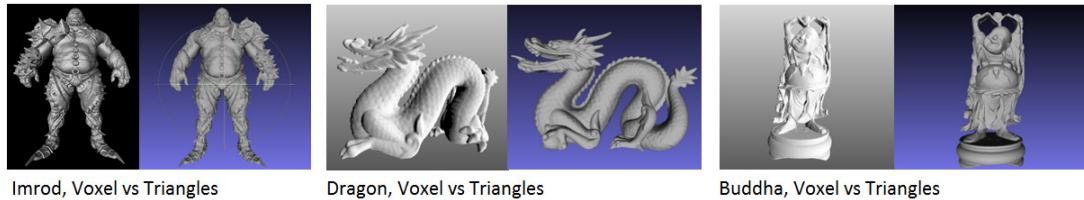
In **Fig. 4.45** the computation speed is summarized for all related methods in relation to the proposed method. As before, the proposed methods' factor is set to one. The results show that none of the related methods achieves a higher average performance for detailed scenes visualized at a high screen resolution. Similar performance is achieved for triangle based raycasting, Sparse Voxel Octree raycasting and GigaVoxels. QSplat achieves in average 0.38 times the performance, and triangle based rasterization achieves only  $1/3000^{\text{th}}$  the performance for complex scenes.

In **Fig. 4.46** the computation visual precision is summarized for all methods related to the proposed method. As before, the proposed method is set as reference. The results show that the proposed method achieves a higher precision than QSplat and a smoother result than Sparse Voxel Octree raycasting, but that the result is not as good as GigaVoxels and triangle based methods. The difference between these methods is significant for geometries close to the camera, where one voxel is larger than one pixel. In case that the camera is far away and one voxel is about the size of one pixel, the results are more similar.

As a summary it turns out that the proposed method achieves the lowest memory consumption among the compared related methods, the highest rendering speed for visualizing complex geometry at high screen resolutions although the proposed method is tied with some other methods. Comprehensive evaluation for the memory consumption and computation speed indicates that the proposed method is best.

**Table 4.2** Memory consumption (1): Triangles compared to the proposed method (voxel)

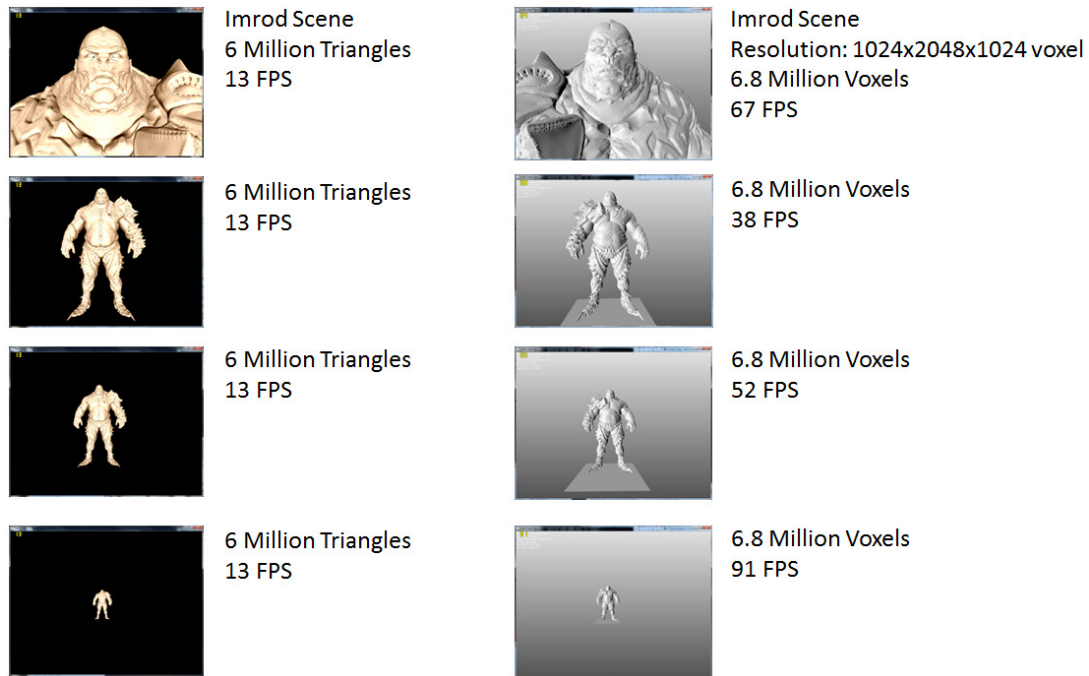
Scene	Triangles			Proposed Method, Voxel, No Color			
	Count (Million)	Size (MB)	Bytes per Triangle	Count (Million)	Size (MB)	Bytes per Voxel	Resolution (in Voxel)
Imrod	18	335	18.5	6.8	14.1	2.1	1024x2048x1024
Dragon	0.871	17.8	20.4	2.9	7.5	3.51	1024x1024x1024
Buddha	1	20.1	20.1	8.8	16.5	1.96	1024x2048x1024

**Table 4.3** Memory consumption (2): GigaVoxels [37] compared to the proposed method for the Sponza scene. The GigaVoxels screenshot is with courtesy of Cyril Crassin

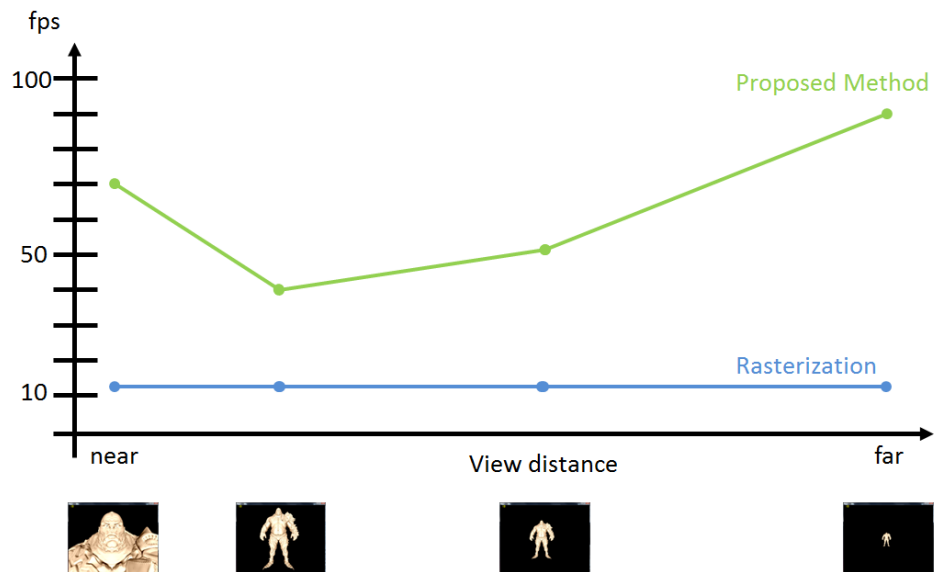
Sponza Scene	GigaVoxels, Voxel, 32 Bit Color	Proposed Method, Voxel, 32 Bit Color	Original, Triangles
Size	70 MB	14.5 MB	7 MB Triangles, 130MB Textures
Resolution	512x512x512 Voxel	512x512x512 Voxel	262.000 Triangles

Qsplat, 16 Bit Normal Vector Data per Splat			
	Scene: Budda Size: 6.8 MB ? Splats ? Byte / Splat		Scene: Dragon Size: 8.3 MB ? Splats ? Byte / Splat
	Scene: David Size: 27 MB 4.2 Million Splats 6.4 Byte / Splat		Scene: St. Matthew Size: 761 MB 127 Million Splats 6 Byte / Splat
Proposed Method, 16 Bit Color Data per Splat			
	Scene: Budda Size: 11.9 MB 2.4 Million Voxel 6.8 Byte / Voxel		Scene: Dragon Size: 13.6 MB 3 Million Voxel 6.2 Byte / Voxel
	Scene: Imrod Size: 32 MB 6.8 Million Voxel 4.7 Byte / Voxel		Scene: Lucy Size: 25.2 MB 5.1 Million Voxel 5.19 Byte / Voxel
Sparse Voxel Octree Raycasting, by Jon Olick		Triangle Raycasting	
	Scene: Imrod Size: ? MB ? Voxel 52 Byte / Voxel		Scene: Imrod Size: 335 MB + ? MB BVH 18 Million Triangles (18.67 Byte + ? Byte) / Triangle

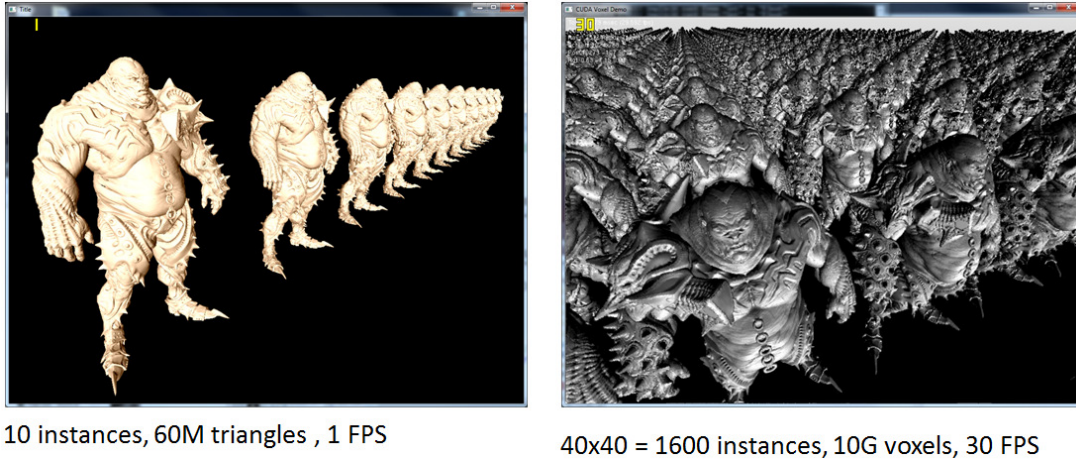
**Figure 4.27** Memory consumption (3): QSplat, Sparse Voxel Octree and Triangle raycasting compared to the proposed method.



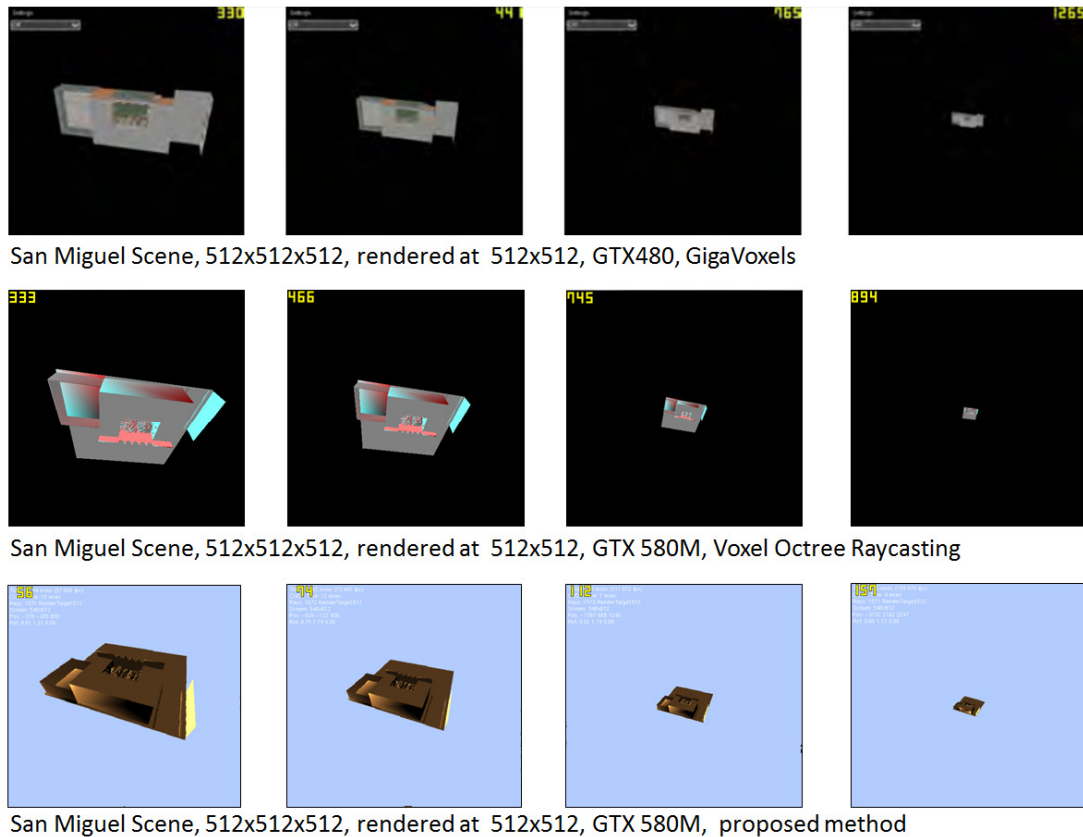
**Figure 4.28** Speed comparison (1): Triangle rasterization compared to the proposed method for multiple camera configurations.



**Figure 4.29** Speed comparison (2): Triangle rasterization compared to the proposed method for multiple camera configurations.



**Figure 4.30** Speed comparison (3): Triangle rasterization compared to the proposed method for a complex scene with multiple Imrod models.



**Figure 4.31** Speed comparison (4): GigaVoxel and this thesis' Sparse Voxel Octree Raycasting (that was implemented for comparison purposes) is compared to the proposed method for multiple camera configurations. The GigaVoxels [43] screenshots are with courtesy of Cyril Crassin



Sparse Voxel Octree Raycasting,  
Jon Olick  
Screen Resolution: 1024x768  
Speed: 60 fps  
Hardware: NVIDIA GTX280



General Sparse Voxel Octree Raycasting,  
This thesis version  
Screen resolution 1024x768  
Speed: 60 fps  
Hardware: NVIDIA GTX580M



Proposed method  
Screen resolution 1024x768  
Speed: 60 fps  
Hardware: NVIDIA GTX580M

**Figure 4.32** Speed comparison (5): Sparse Voxel Octree Raycasting of Jon Olick is compared to this thesis' Sparse Voxel Octree Raycasting method and to the proposed method. The Sparse Voxel Octree Raycasting screenshot (left) is with courtesy of Jon Olick.



General Sparse Voxel Octree Raycasting  
Resolution : 512x512x512 Voxel  
Screen Resolution : 2048x768 Pixel  
0.9 Million Voxel  
Speed : 65 fps



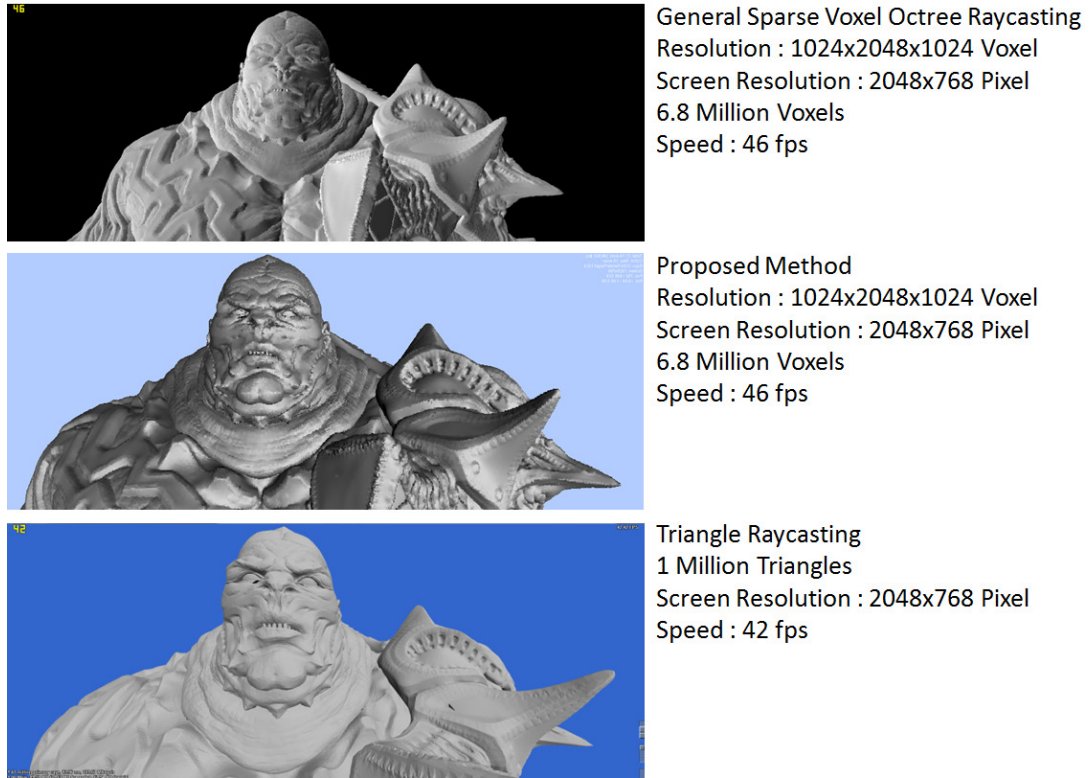
Proposed Method  
Resolution : 512x512x512 Voxel  
Screen Resolution : 2048x768 Pixel  
0.9 Million Voxel  
Speed : 50 fps



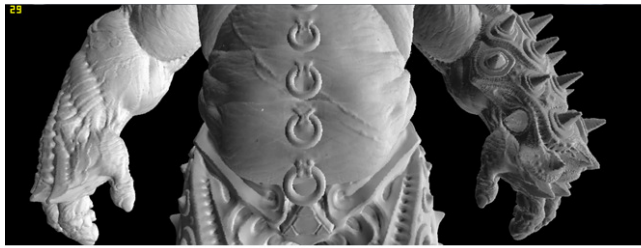
Triangle Raycasting  
1 Million Triangles  
Screen Resolution : 2048x768 Pixel  
Speed : 26 fps

**Figure 4.33** Speed comparison (6): this thesis' Sparse Voxel Octree and triangle raycasting are compared to the proposed method.

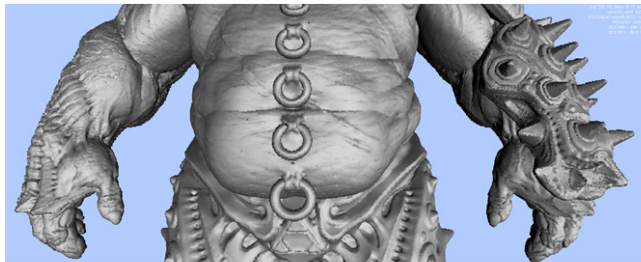




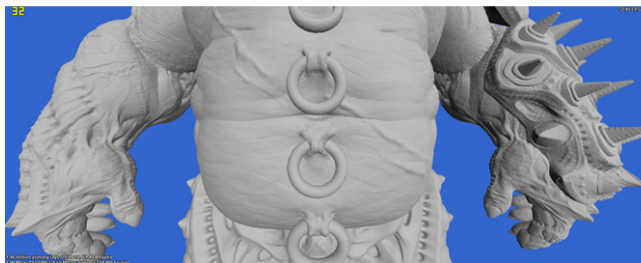
**Figure 4.34** Speed comparison (7): this thesis' Sparse Voxel Octree is compared to the proposed method and to triangle raycasting.



General Sparse Voxel Octree Raycasting  
Resolution : 1024x2048x1024 Voxel  
Screen Resolution : 2048x768 Pixel  
6.8 Million Voxels  
Speed : 29 fps

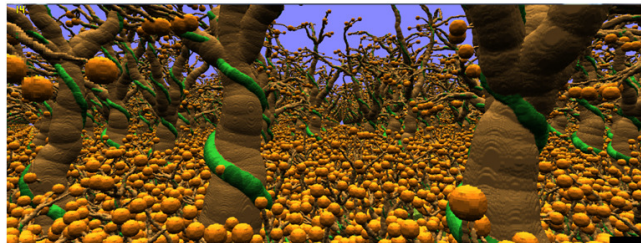


Proposed Method  
Resolution : 1024x2048x1024 Voxel  
Screen Resolution : 2048x768 Pixel  
6.8 Million Voxels  
Speed : 31 fps

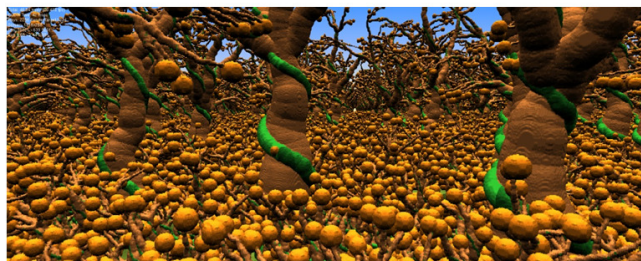


Triangle Raycasting  
1 Million Triangles  
Screen Resolution : 2048x768 Pixel  
Speed : 32 fps

**Figure 4.35** Speed comparison (8): this thesis' Sparse Voxel Octree is compared to the proposed method and to triangle raycasting



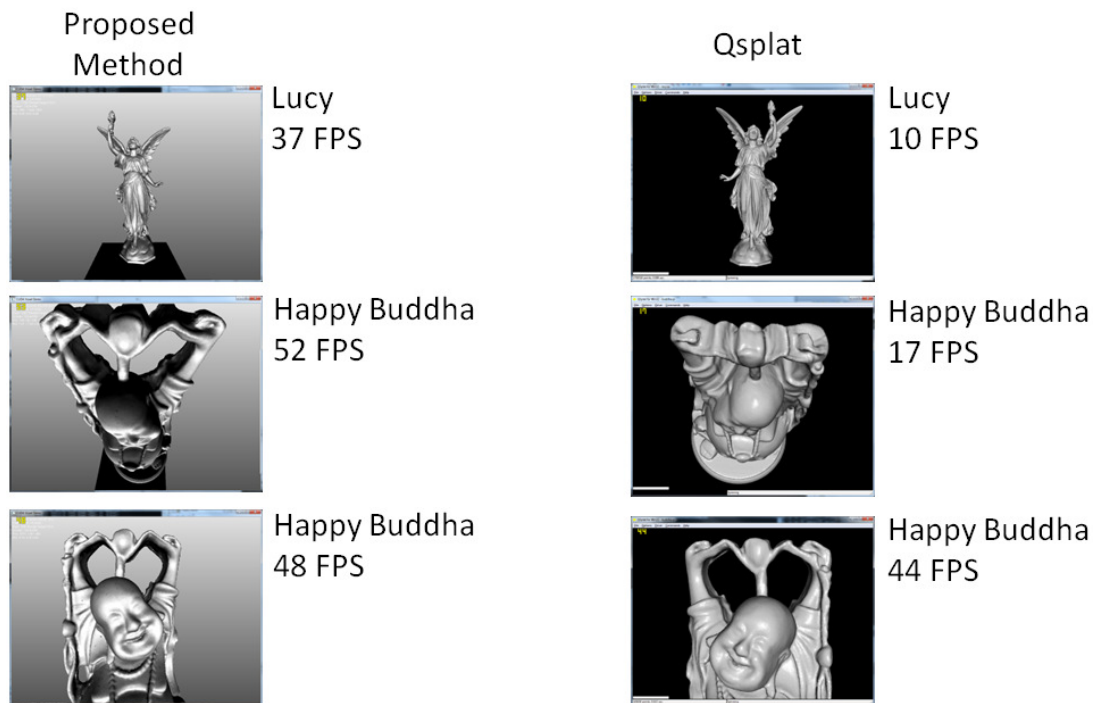
General Sparse Voxel Octree Raycasting  
Resolution : 1024x1024x1024 Voxel  
Screen Resolution : 2048x768 Pixel  
12.6 Million Voxels  
Speed : 14 fps



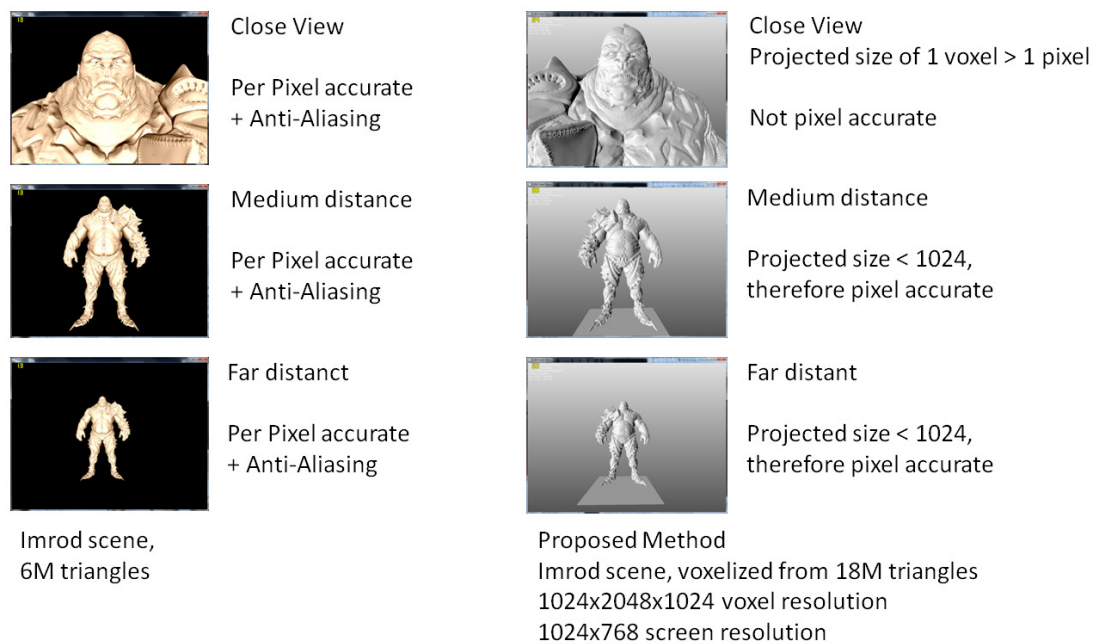
Proposed Method  
Resolution : 1024x1024x1024 Voxel  
Screen Resolution : 2048x768 Pixel  
12.6 Million Voxels  
Speed : 20 fps

**Figure 4.36** Speed comparison (9): this thesis' Sparse Voxel Octree is compared to the proposed method.





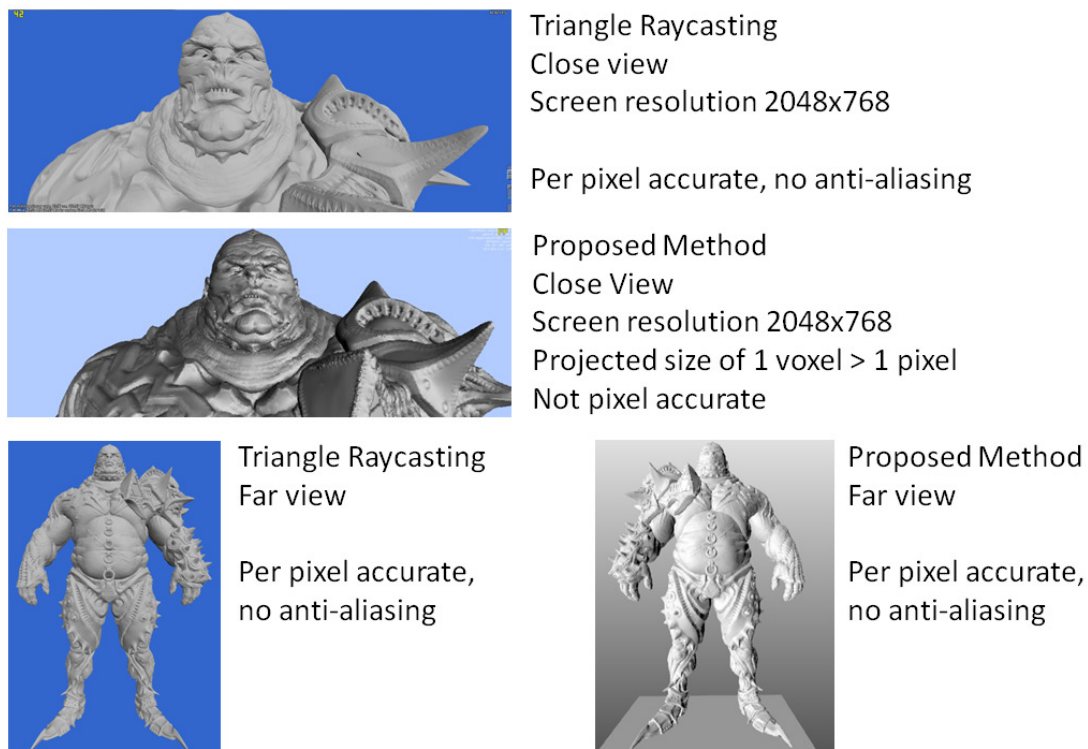
**Figure 4.37** Speed comparison (10): the proposed method is compared to QSplat.



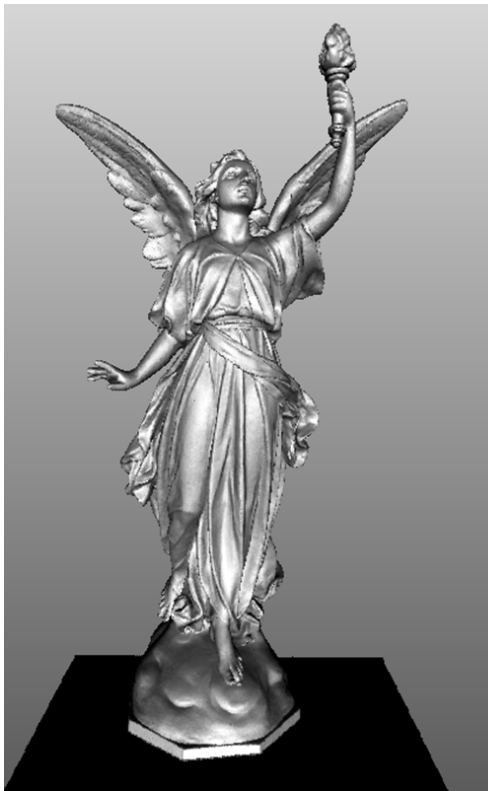
**Figure 4.38** Visual Precision (1): the proposed method is compared to triangle rasterization.



**Figure 4.39** Visual Precision (2): the proposed method is compared to GigaVoxels. The GigaVoxels screenshots is with courtesy of Cyril Crassin



**Figure 4.40** Visual Precision (3): the proposed method is compared to triangle raycasting.

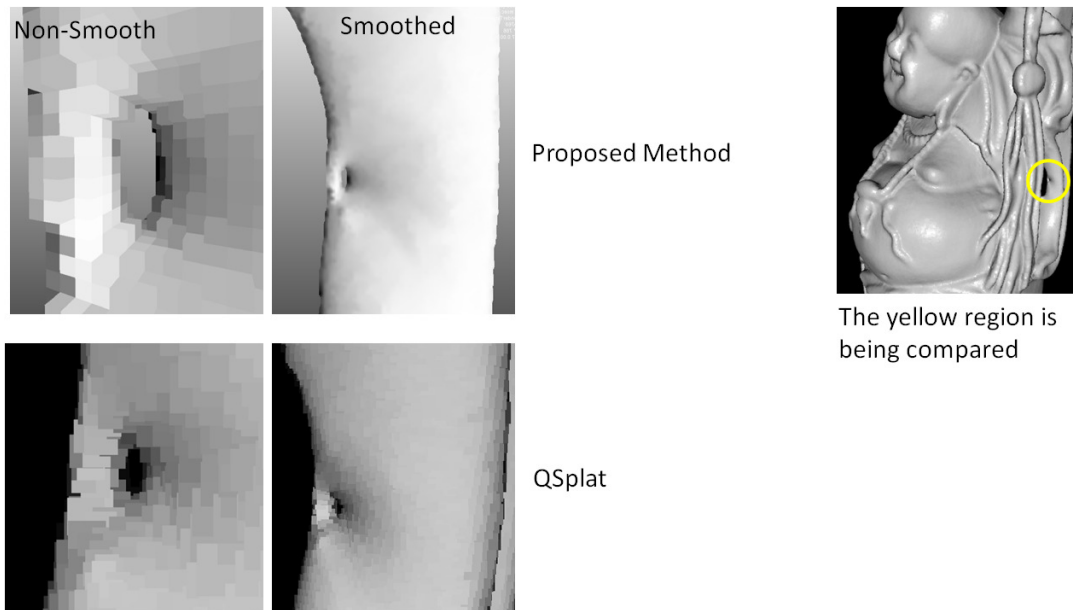


Proposed Method



QSplat

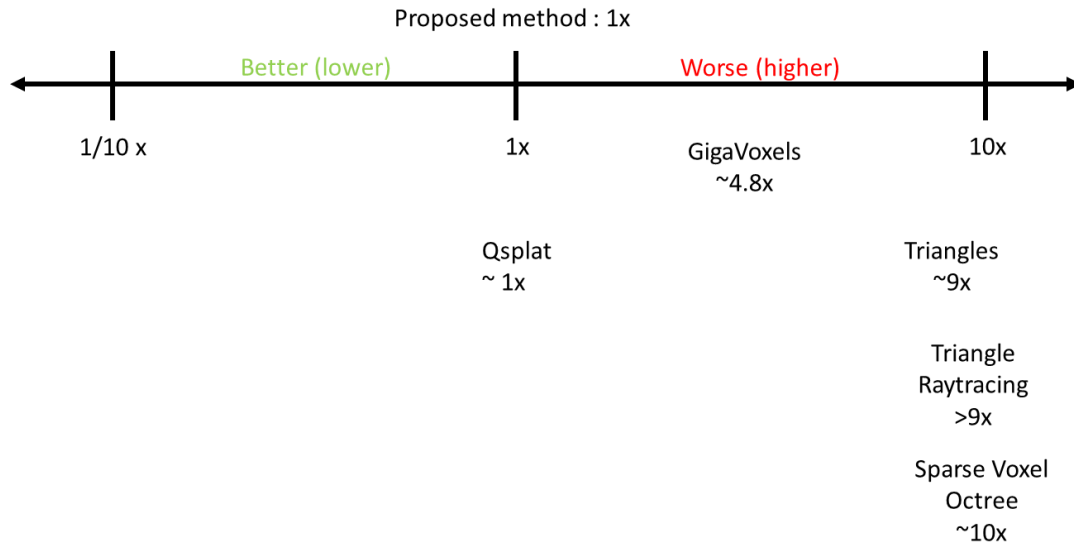
**Figure 4.41** Visual Precision (4): the proposed method is compared to QSplat.



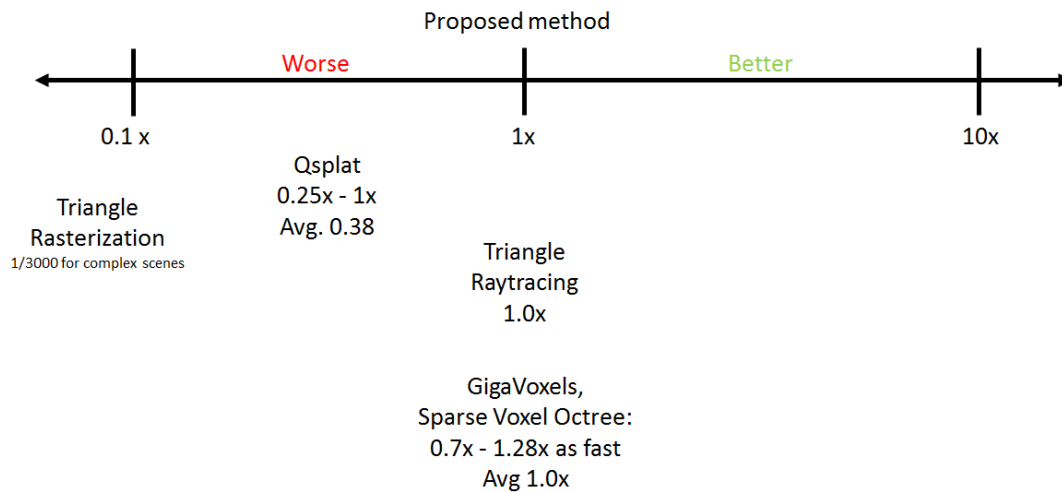
**Figure 4.42** Visual Precision (5): the proposed method is compared to QSplat, close-up view.



**Figure 4.43** Visual Precision (6): the proposed method is compared to Sparse Voxel Octree Raycasting. The Sparse Voxel Octree Raycasting screenshot (left) is with courtesy of Jon Olick.

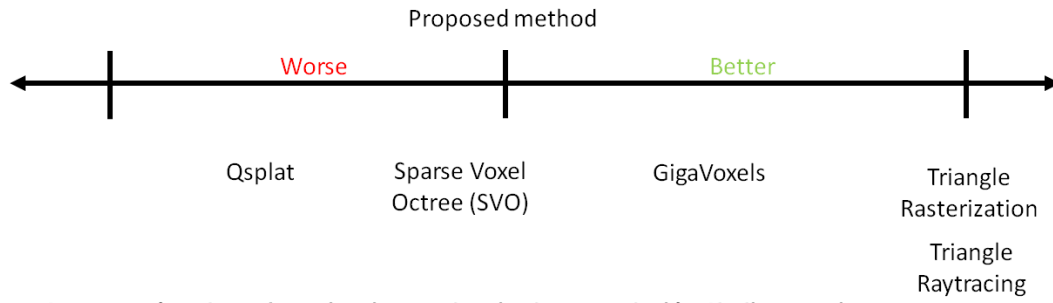


**Figure 4.44** Summary of memory consumption per element.

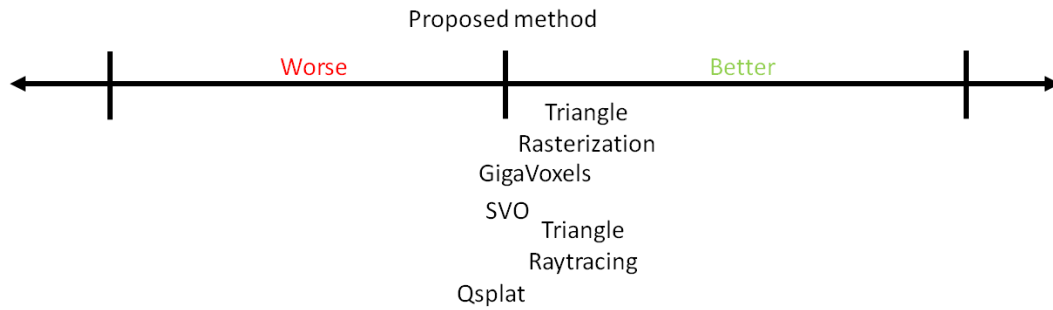


**Figure 4.45** Summary of computation speed for high screen resolutions (2048x768).

**Near Geometry ( projected voxel size > 1 pixel ) : Visible Differences**



**Far Geometry ( projected voxel, splat or triangle size  $\leq 1$  pixel ) : Similar Results**



**Figure 4.46** Summary (3): visual precision.

## 4.7 Conclusion

This chapter has proposed a raycasting based method for the fast visualization of complex RLE compressed voxel data scenes. The proposed method improves the original voxel forward projection algorithm in several ways so that complex scenes are efficiently visualized and so that low memory consumption is achieved.

The experimental results and discussion are summarized as follows.

- **Memory Consumption:** The low memory consumption is achieved using the proposed 16 bit volume data run-length-encoding with 10 bit used for voxel skipping and 6 bit used for counting stored voxels. As a result of applying the proposed method and related methods to multiple data-sets (triangle and voxel data) in the experimental results section, the following results are obtained:
  - The proposed method and QSplat consumes least memory.
  - Gigavoxels, triangles, triangle raytracing and sparse voxel octrees consume more memories.

- Rendering speed: The proposed method and related methods are applied to multiple data sets including voxelized polygon data and procedurally generated voxel data. The following results are obtained:
  - The proposed method, triangle raytracing, GigaVoxels, and sparse voxel octree are fastest.
  - QSplat and triangle rasterization are slower.
- Comprehensive Evaluation: The proposed method is tied with QSplat in terms of low memory consumption, and is tied with triangle raycasting, GigaVoxels and sparse voxel octree in terms of rendering speed. Comprehensive evaluation for these results indicate that the proposed method is best. This comprehensive evaluation result indicates that the goals of this chapter are achieved.

The proposed method uses a special filter method to smooth voxel edges on the screen. The filter method works well for specified distance range of voxels to the camera. For voxels that are very close to the camera, a large area would be required to be smoothed out, but this is not done due to performance reasons.

Future work might include developing better ways to achieve smooth surfaces of the visualized voxel structure on the screen, without having impact on rendering speed or memory consumption.





## Chapter 5. Skeletal Animation

### 5.1 Goals

This chapter proposes novel ways to improve existing skinned skeletal animation methods.

Skinned skeletal animation methods can be utilized to bind a skeleton to any arbitrary triangular mesh for achieving any kinds of complex deformations. Commonly, skinned skeletal animation is used for animating life-forms in general, where most of them are human characters in video games or cinematic productions.

#### 5.1.1 Spline Skinning

- Non-collapsing geometry: The proposed skinned skeletal animation approach should avoid collapsing geometry, which could occur in joints that are bent by large angles in case of conventional matrix skinning.
- Faster computation and flexibility: Higher performance and more flexibility compared to DQS should be achieved.
- Small number of control joints for a spine: As existing methods require many control joints to represent a spine or facial animation, the proposed approach should significantly reduce the number of necessary joints without sacrificing the quality of the deformation.

#### 5.1.2 Deformation Styles

- Reusability: Different from existing methods, the proposed approach should allow the simple and abstract design of deformation styles for re-usable deformation behaviors. The generation of muscle like deformations or the design of cloth wrinkles should be allowed, for the instant application to any number of target characters simultaneously.

For brevity, in the following discussion the proposed skeletal animation module is referred to as *animation system*.

### 5.2 Related Work

Over time, many methods have appeared to achieve the animation of characters, where the most important methods can be divided into Free-Form-Deformation (FFD) [44], Skeletal Subspace Deformation (SSD) [18], shape blending and spline aligned deformations [45]. They form a foundation for many subsequent research approaches and have reappeared in countless variants and combinations since their initial invention. In order to improve the deformation quality and realism for skeletal animations, various methods have been suggested.

### 5.2.1 Spline Skinning

SSD, which is the earliest method, is still the most popular method for skinned skeletal animation today. However, due to deformation artifacts due to large bend angles, many methods have been proposed to improve that. Methods that directly improve SSD are QS [19] and DQS [20]. They change the interpolation domain from matrices to quaternions or even dual quaternions (DQS). This cannot prevent all deformation artifacts, but successfully avoids effects as collapsing geometry by preserving a high computational speed, while their computational speed is still not as high as SSD. In case of quaternion skinning, about 78% of the speed of SSD is achieved, and in case of DQS, 72% the speed of SSD is achieved, where details about their performances are provided by [19] and [20].

A different and more flexible approach is to use skinned spline aligned deformations and apply them to skinned skeletal animation. Two methods to achieve these are [46] (an earlier version of the proposed method that calculates the spline on a per vertex basis rather than pre-computing the spline prior to the per vertex deformation step) and [47]. Concerning Yang et. al's method [47], focuses on the non-real-time case, as their application is a plugin for the commercial software Maya, while [46] focuses on the application in real-time systems by extensive usage of the GPU. In [47], which was developed independently from the method proposed in this chapter at exactly the same time, high performance and the use in real-time applications was not intended. Therefore, performance benchmarks were not provided.

Another related approach in this context is Cornea et al's method [48], which introduced curve skeletons are introduced and discussed in general. Their method focuses on automatically computing of curved skeletons from models rather than utilizing manually created skeletons, for skeletal animations.

A method that extends FFD is a sweep-based FFD [49], which was independently developed the same time from the method proposed in this chapter and appeared in the same conference as Yang et al's skinned skeletal animation approach [46]. The sweep based FFD provides the ability to efficiently model radial deformations by allowing the user to edit cross-sections along spline-curves. Their method however, does not provide skinning; therefore it is not possible to have vertices influenced by multiple spline curves. Furthermore, since they do not focus on real-time applications, they do not provide any performance benchmarks for the deformation time. The proposed system utilizes two variants of sweep-based FFDs to apply the deformation styles to the geometry, as detailed in later sections.

Another sweep-based algorithm is Hyun et al's method [50], which uses the sweep-based deformation to create skinned skeletal animation. The algorithm allows a limited creation of customized deformations, as the user can define virtual muscles, which are taken into account during the animation. Their method achieves good deformation quality, but they achieve barely real-time performance due to complex computations even for a single character. It is, therefore, not suited well for real-time video-game applications.

To provide more realistic deformations, advanced methods such as [51], [52], [53], and [54] were developed. They allow the adjustment of the material stiffness and take physical constraints into

account, which directly affects the deformation. Other methods, such as the volumetric graph laplacian [55] construct an inner graph to preserve the mesh's inner volume while deforming. All of these methods can provide a high quality deformation, but they cannot provide the same high performance as SSD due to their complexity.

### 5.2.2 Deformation Styles

A method that allows the re-use of deformations is Sumner et al's method [56], by which the animation of one mesh may drive the deformation of another, similar mesh. Different from this chapters' goals, their method targets at reusing complete deformations, not deformation's behaviors. They do not allow the creation of an abstract deformation behavior independent of the underlying mesh.

Example based methods allow the pose-dependent modification of animations. Initially pose-space-deformation (PSD) [57] was developed and then was advanced by [58] and [59]. PSD basically allows an artist to individualize particular poses, where intermediate poses are calculated by interpolation. In this case, pose-dependent deformations can also be modeled by the artist, but in a different and more complex way to the proposed method. Instead of directly displaying certain vertices in a certain pose, more abstract design, which could cover all poses is desired. However, PSD and related methods achieve that, because they are tightly bound to the mesh they created.

Cloth simulations were developed to provide realistic cloth appearances. A comprehensive overview can be found in [4]. Conventional cloth simulations include [60], [61], and [62], which allow the design of surface details for the animation. However, cloth deformations cannot achieve the goal of re-usable deformations for skinned skeletal animations designed by an artist. Furthermore, all the three methods apply surface details in direction to the surface normal and depend on the local mesh deformation rather than utilization of skeleton's pose; therefore, their deformation is entirely different from the proposed method.

Different from PSD, algorithm of [62] generates pose-dependent wrinkles procedurally for producing a cloth-like appearance. Since the method is limited to automatic generated wrinkle patterns, they cannot handle arbitrary custom artist designed deformation styles as intended here.

## 5.3 Proposed Method

**Fig. 5.1** shows the two proposed methods: spline skinning (Section 5.5) and deformation styles (section 5.6), where deformation styles is built on top of spline-skinning.

### 5.3.1 Spline Skinning

Spline skinning evolves as a combination of spline aligned deformations [45] (SAD) and conventional SSD. While SSD uses vertex weights to blend matrices, spline skinning uses the weights to blend the results of multiple splines aligned deformations.

As shown in **Fig. 5.1**, SAD consists of two parts to animate a vertex. First, the transformation from world-space into spline-space for the rest pose and second, the transformation from spline-space into the world-space for the animated pose. The proposed spline skinning uses three splines to drive the deformation of a vertex, where the influence of each spline is defined by the skinning weight stored along with the vertex.

SAD provides high-quality bend and twist deformations without exposing unwanted artifacts, which could occur in case of SSD, as shown in **Fig. 5.2**. Another advantage is their fast and stable computation, which is an important property for real-time applications. Concerning the spline function, a special polynomial based spline with variable exponent depends on only three control-points for highest performance.

Since splines may further help to simplify complex skeletal animations, such as a spine or facial animations, it to replace multiple joints of common skeletal animation systems by one single spline. Regarding speed, the computation per vertex can be reduced by SSD. Spline skinning can therefore be computed faster than QS and DQS, which is used in the CryEngine3.

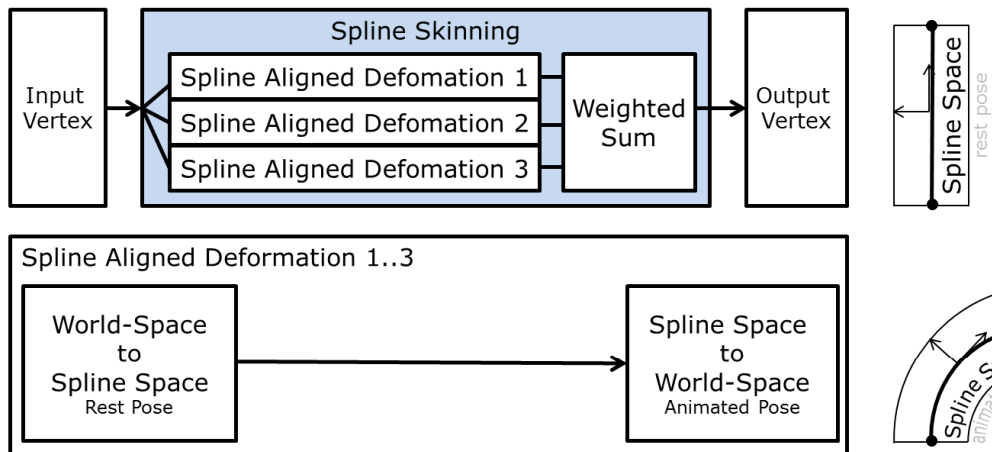
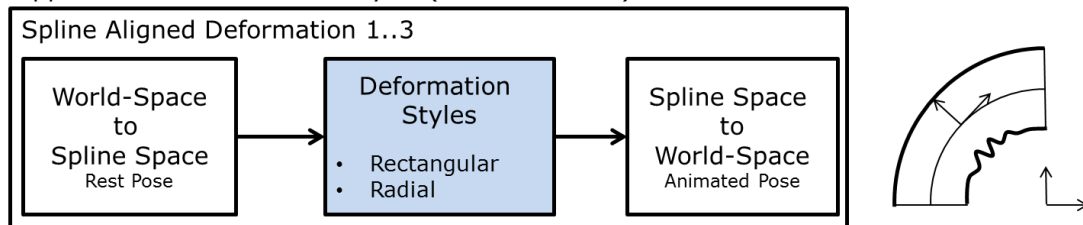
### 5.3.2 Deformation Styles

To achieve reusable and pose dependent deformation behaviors that can be designed in an abstract manner, deformation styles are proposed. As shown in **Fig. 5.1**, bottom, deformation styles are based on spline skinning. They are integrated into the spline skinning's spline aligned deformation module.

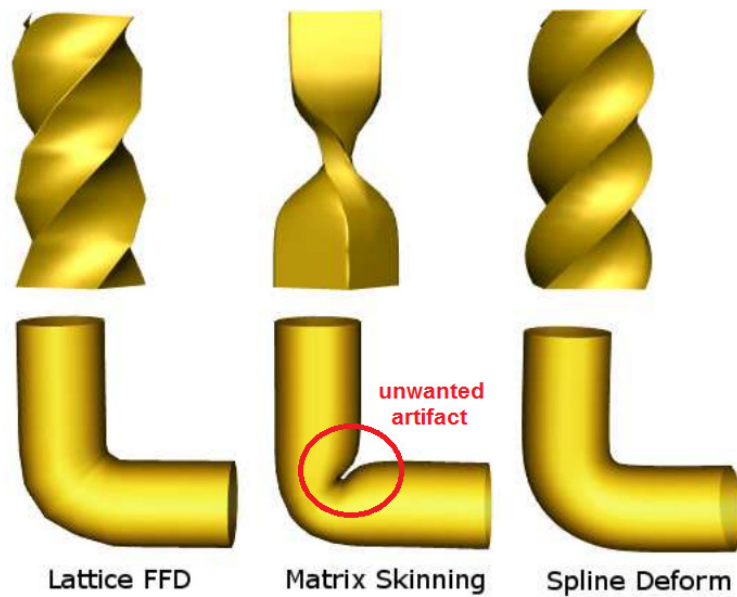
Deformation styles are based on two sweep-based FFD variants, which are attached to each joint. Deformation styles do not require any knowledge about the underlying geometry, which they are applied to, as opposed to PSD.

The first two FFD variants are based on a high-resolution, radial FFD grid, which is wrapped around the spline. They allow to achieve high resolution concentric deformation effects. The effects include metal- and cloth-like deformations or even muscle bulges. The radial FFD grid is driven by three scale textures, which are summed up to a final scale texture using a scale factor (weight) for each texture. The weight depends on the angles of the spline's control points. The three textures are utilized for frontal, lateral and radial scaling.

The second FFD variant is a rectangular scale envelope that is supposed to allow a simple definition of more general scalings. The artist therefore draws three outlining curves for the frontal, lateral and radial direction. Goals of the second variant include the design of folds to prevent self-intersections such as the elbow or the modeling of major lateral bulges for soft-bodies.

**Approach 1 : Spline Skinning (Skeletal animation)****Approach 2 : Deformation Styles (Surface details)**

**Figure 5.1** Proposed spline skinning and deformation styles. Top: spline skinning, middle: spline aligned deformations, bottom: deformation styles.



**Figure 5.2** Bend and Twist deformations: left: FFD, middle: SSD, right: spline aligned deformation.

## 5.4 Splines

### 5.4.1 Fundamentals

In order to achieve spline-aligned deformation, finding a suitable spline-function is required. Arc-spline and the polynomial Bézier-spline are candidates, because both can be computed very fast, which is important for the use of spline based deformation in real-time applications. In order to provide highest speed, three control points are required.

However, both functions do not naturally allow a modification of the curve stiffness without adding another control point. The proposed approach lets the second control point basically represent a joint's rotation center, and hence, additional control points complicate the computation. To achieve a simple and easy handling, three control points  $p_{i \ (i=1,2,3)}$  are chosen. The basic Bézier spline function  $f_b$  is used to create a new spline function  $f_m$ , which provides an additional parameter  $a$  for a continuous variable adjustment of the spline's stiffness. In **Fig. 5.3**, a comparison among the spline functions is shown, together with the function of the additional parameter  $a$ .

The variable  $x$  defines the position on the spline:

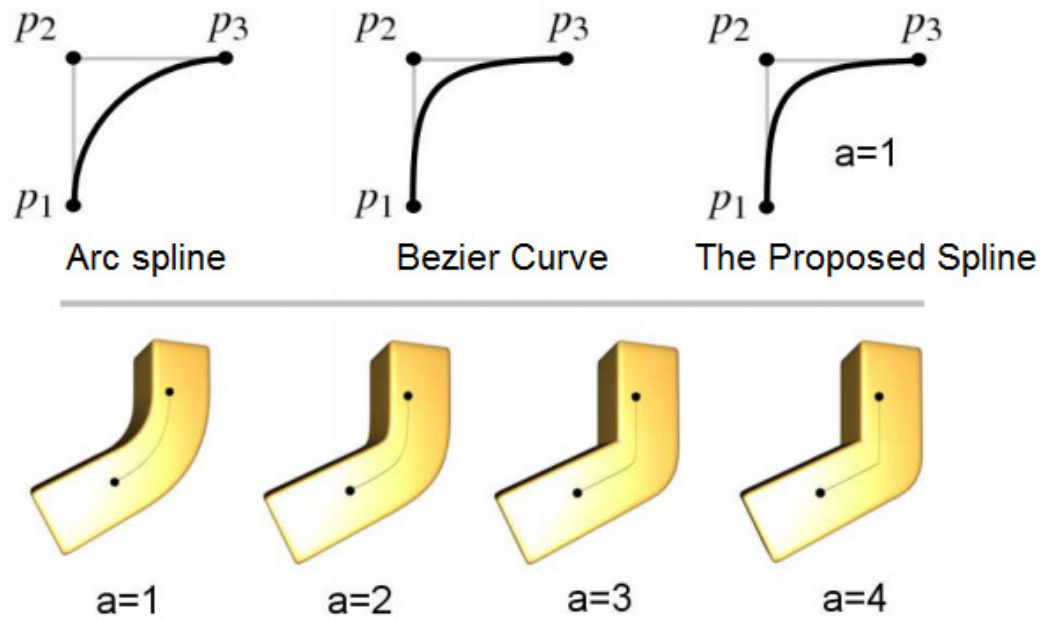
$$\begin{aligned} \forall x \in [0,1], \forall a \geq 2, \\ \Delta_{12} &= p_2 - p_1, \\ \Delta_{23} &= p_3 - p_2. \end{aligned} \tag{5.1}$$

**Conventional Bezier curve in  $\mathbb{R}^3$ ,  $f_b: \mathbb{R} \rightarrow \mathbb{R}^3$ :**

$$\begin{aligned} f_b(x) &= (1-x)^2 \cdot p_1 + (2 \cdot x \cdot (1-x)) \cdot p_2 + x^2 \cdot p_3, \\ f'_b(x) &= 2 \cdot (x-1) \cdot p_1 + (2-4x) \cdot p_2 + 2 \cdot x \cdot p_3. \end{aligned} \tag{5.2}$$

**The modified Bezier curve in  $\mathbb{R}^3$ ,  $f_m: \mathbb{R} \rightarrow \mathbb{R}^3$ :**

$$\begin{aligned} f_m(x) &= p_1 + (1 - (1-x)^a) \cdot \Delta_{12} + x^a \cdot \Delta_{23}, \\ f'_m(x) &= a \cdot (1-x)^{a-1} \cdot \Delta_{12} + a \cdot x^{a-1} \cdot \Delta_{23}. \end{aligned} \tag{5.3}$$



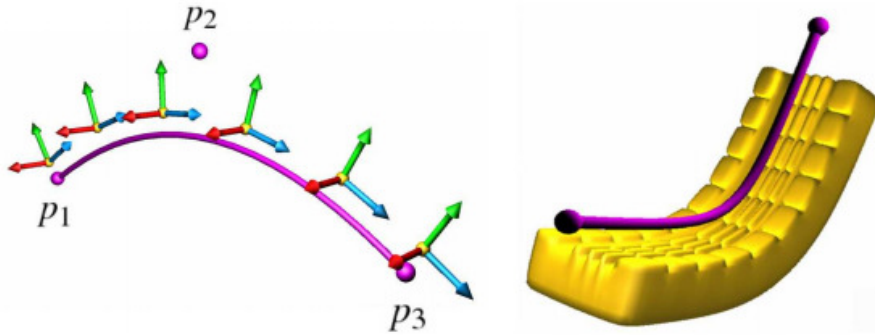
**Figure 5.3** Spline functions: In the upper row, the short-listed spline functions are compared. The lower row shows the ability of the spline to adjust the stiffness by parameter  $a$ .

### 5.4.2 Spline Aligned Deformation

In order to apply the proposed spline with variable stiffness for geometric deformations, it is necessary to define a local coordinate-system around it, the so-called Frenet-frame, as shown in **Fig. 5.4**. A complete orthonormal basis  $b$  can be computed for each position of the spline, where the origin  $b_O$ , normal  $b_N$ , tangent  $b_T$  and bi-normal  $b_B$  are defined as follows:

$$\begin{aligned}
 b_T(x) &= f'_m(x), \\
 b_N &= \Delta_{12} \times \Delta_{23}, \\
 b_B(x) &= b_N \times b_T(x), \\
 b_O(x) &= f_m(x); \\
 B &= [b_N \mid b_B \mid b_T \mid b_O]; \quad (5.4) \\
 B &= \begin{bmatrix} b_N \cdot x & b_B \cdot x & b_T \cdot x & b_O \cdot x \\ b_N \cdot y & b_B \cdot y & b_T \cdot y & b_O \cdot y \\ b_N \cdot z & b_B \cdot z & b_T \cdot z & b_O \cdot z \\ 0 & 0 & 0 & 1 \end{bmatrix}.
 \end{aligned}$$

The origin of the coordinate frame is simply the spline function  $f_m$  itself. Then, the tangent  $b_T$  is equal to the spline's derivative  $f'_m$ . The normal  $b_N$  can be pre-calculated as it is perpendicular to the three control points  $p_{1,2,3}$  and finally the bi-normal  $b_B$  can be computed as cross-product of the normal  $b_N$  and the tangent vector  $b_T$ . The 4x4 transformation matrix  $B$  is built by arranging the four computed vectors as column vectors.



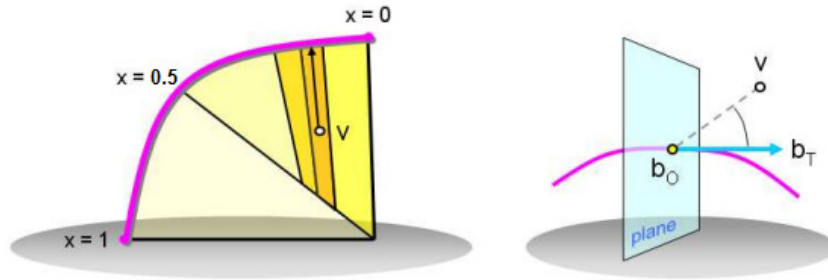
**Figure 5.4** The spline coordinate system: left: the spline function in violet together with the spline's coordinate system, where the spline's tangent is indicated in blue, the normal in red, the bi-normal in green and the origin of each coordinate system in yellow; right: an example deformation.

### 5.4.3 Spline Binding

Prior to the deformation, as shown in **Fig. 5.5** all vertices of the target mesh are mapped perpendicular to a specific position  $x$  on the spline. This is achieved by utilizing a plane based



binary search algorithm, starting at  $x=0.5$ . In the left side of **Fig. 5.5** the color intensity of the segments indicates the iteration depth of the binary search. The right side shows the plane used to determine the direction for the next iteration step.



**Figure 5.5** The binding process: (left) the perpendicular mapping of vertex  $v$  to the spline by using binary search, (right) a way to determine the search direction in each step.

In order to determine the search direction for  $x$  at each step, the perpendicular constraint based on the scalar product  $\langle \cdot | \cdot \rangle$ , the vertex  $v$ , and the plane defined by  $b_T$  and  $b_O$  is utilized as follows:

$$\langle b_T(x) | v - b_O(x) \rangle = \begin{cases} < 0, & \text{if } v \text{ lies in front of the plane,} \\ = 0, & \text{if } v \text{ lies in the plane (solved),} \\ > 0, & \text{if } v \text{ lies behind the plane.} \end{cases} \quad (5.5)$$

The spline-basis representation  $v'$  of  $v$  is defined according to the spline's matrix  $B$ :

$$v' = B^{-1} \cdot v. \quad (5.6)$$

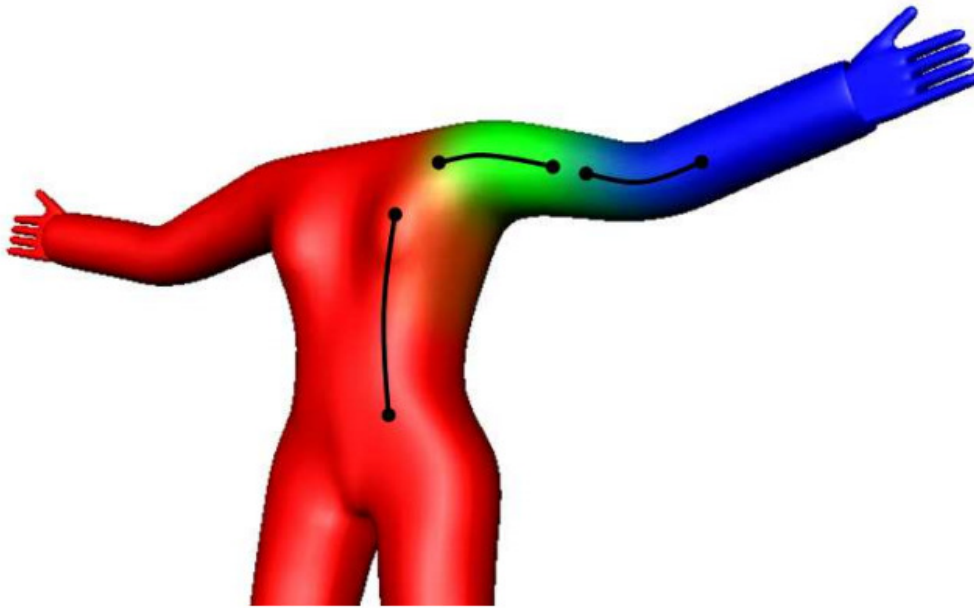
## 5.5 Spline Skinning

To achieve a usable result in character-animation, it is necessary to let each vertex of the geometry be weighted by multiple splines; otherwise it is not possible to have branches (one bone branching into bones two) in the skeleton. For assigning the weights per vertex, the conventional way to paint the weight intensities via color on the geometry similar to a texture map is used. This method has already proven its effectiveness in various 3D authoring tools such as Maya, 3DS Max or Blender. An example can be seen in **Fig. 5.6**, where each of the red, green, and blue colors is assigned to an individual rigged body part. The colors indicate the weight for each spline. Red is related to the body spline, green to the shoulder and blue to the elbow. In overlapping areas the weights are mixed, which results in color transitions. In **Fig. 5.6**, up to three splines influence

a single vertex  $v_f$ . To preserve a correct scaling, all weights  $w_i$  for each vertex must be normalized to one. The resulting formula is written as follows:

$$\begin{aligned} B''_i &= B_i \cdot B'_i{}^{-1}, \\ v_f &= \sum_{i=1}^n w_i \cdot B''_i \cdot v. \end{aligned} \quad (5.7)$$

In Eq.(5.7), two different spline bases ( $B_i$  and  $B'_i$ ) are used. The basis of the actual pose is defined by  $B$ , while  $B'$  defines the basis while binding<sup>65</sup>. In Eq.(5.7), the basis  $B''_i$  is pre-computed, which can be used to write the formula equal to conventional matrix skinning (SSD). In SSD, also two matrices are used. The difference is that the matrices change depending on the position on the spline curve.



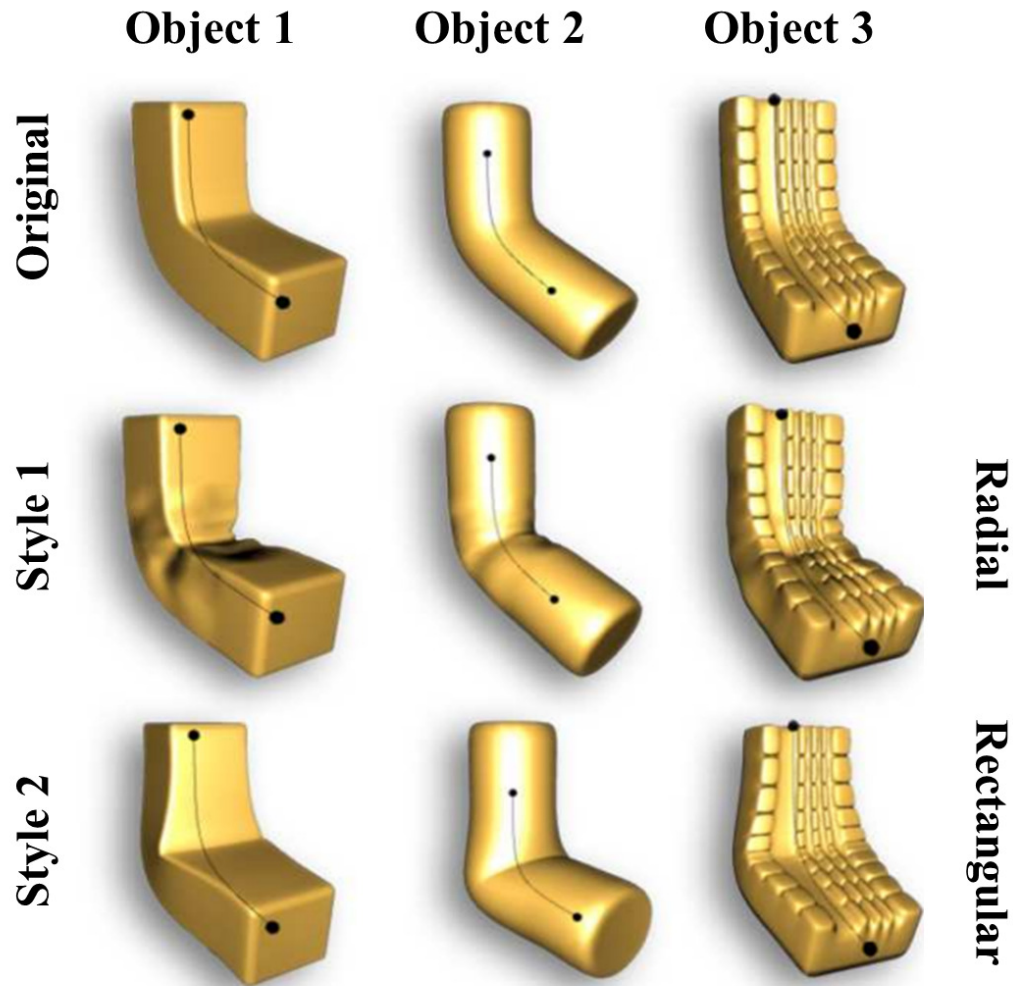
**Figure 5.6** Spline skinning: skinning weights

## 5.6 Deformation Styles

In order to allow a flexible customization by the artist of the spline-aligned deformation (computed as in Eq.(5.7)), deformation styles is used, allowing pose-dependent modeling of a joint's deformation that can be used to represent material behaviors of metal, cloth or muscles. Basic spline skinning does not allow to model such material behaviors. Each style is created from the combination of two pose-dependent FFD variants, where each of the two variant has its own advantages that cannot be replaced by the other. In **Fig. 5.7**, two different deformation styles are equally applied to three objects. The first method applies a radial scale envelope (**Fig. 5.7**, Style

<sup>65</sup> Binding defines the process where the initial skeleton pose is stored for the undeformed mesh

1) while the second method uses a rectangular scale envelope (**Fig. 5.7**, Style 2). The radial scale envelope (**Fig. 5.7**, Style 1) is used to model wrinkles and other high detail deformations. The rectangular scale envelope (**Fig. 5.7**, Style 2) is used to avoid self-intersections and model radial bulges. In the following, the deformation styles are detailed.

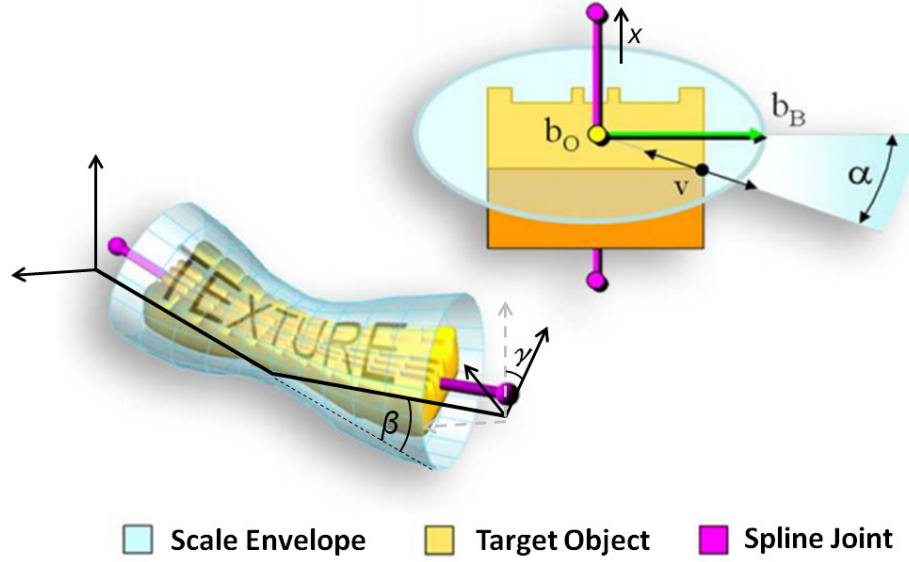


**Figure 5.7** Deformation Styles

### 5.6.1 Radial Scale Envelope (Style 1)

#### 5.6.1.1 Principle

The Radial Scale Envelope is shown as Style 1 in **Fig. 5.7** and is designed to allow high resolution skin deformation effects such as folds and wrinkles. The algorithm basically applies the deformation by concentrically scaling a vertex  $v$  of the target object with respect to the spline origin  $b_O$ , as is shown in **Fig. 5.8**.



**Figure 5.8** Radial Scale Envelope: The lower left side shows an example envelope while the upper right side shows the concentric scaling of  $v$  in relation to the spline

As indicated by Eq.(5.8), the scaling is determined by a radial scale function  $S_{rs}$ , which depends on the two-dimensional position  $(x, \alpha)$  on the envelopes surface and the pose of the joint. The variable  $x$  defines the position on the spline. The pose is defined by the joint's twist  $\gamma$  and the joint's bend-angle  $\beta$ , which is based on the two vectors  $\Delta_{12}$  and  $\Delta_{23}$ . The angle  $\alpha$  denotes the angle between the vertex  $v$  and the spline's bi-normal  $b_B$  with respect to the spline's origin  $b_O$ . Hence, the deformation function  $D_{rad}$  is defined to evaluate the deformed vertex as follows:

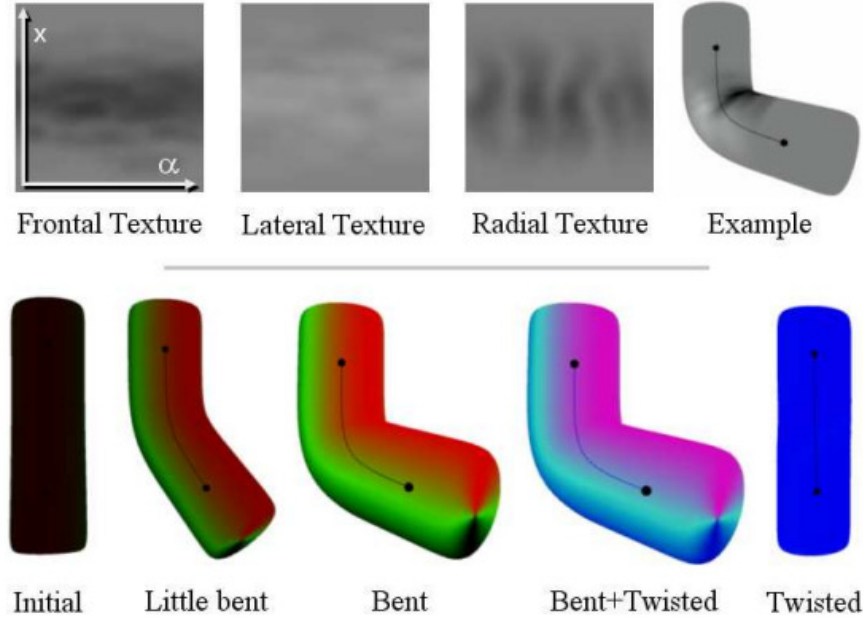
$$D_{rad}(v') = v' \cdot S_{rs}(x, \alpha, \beta, \gamma), \quad \forall \alpha, \beta \in [0, \pi], \forall \gamma \geq 0. \quad (5.8)$$

Since the scaling needs to be applied in spline space,  $v'$  is used instead of  $v$ .

This simplifies the calculation, as the origin in spline-space is  $b_O$  and the multiplication of  $v'$  by any scalar is equal to scaling  $v'$  with respect to  $b_O$ .

#### 5.6.1.2 Radial Scale Function and Textures

The scale function  $S_{rs}$  computes the scaling based on three scale textures, shown in **Fig. 5.9's** upper row. The first texture  $T_f$  is used for frontal scaling, the second for lateral ( $T_l$ ), and the third for radial ( $T_r$ ). The weight distribution for the three textures can be seen in **Fig. 5.9**, lower row. The weight  $w_f$  for the frontal texture  $T_f$  represented by red,  $w_l$  for the lateral  $T_l$  by green, and  $w_r$  for the radial  $T_r$  by blue. The coordinate system  $x, \alpha$  is defined with respect to a cylindrical coordinate system.



**Figure 5.9** Scale Textures: The upper row shows the three scale textures that were used to create Style 1 in **Fig. 5.7** and an example object where the textures are applied. The lower row shows the pose-dependent weight calculation to apply the three textures, where red corresponds to the weight of the frontal scale texture, green to the lateral and blue to the radial.

The scale factor  $S_{rs}$  in Eq.(5.8) for a certain vertex  $v$  is computed by sampling all three textures at the texture-coordinates  $(x, \frac{\alpha}{\pi})$ , and by evaluating the three pose-dependent weights  $w_i$  ( $i=f.l.r$ ) to compute the scaling result  $S_{rs}$  as follows:

$$\begin{aligned}
 & \forall w_f, w_l, w_r \in [0,1], \\
 & \forall w_{\text{sum}}, t_f, t_l, t_r, T_f, T_l, T_r \geq 0, \\
 & w_f = \max(-\cos(\alpha), 0) \cdot \beta, \\
 & w_l = |(\sin(\alpha))| \cdot \beta, \\
 & w_r = \gamma, \\
 & w_{\text{sum}} = w_f + w_l + w_r, \\
 & t_f = w_f \cdot T_f \left(x, \frac{\alpha}{\pi}\right), \\
 & t_l = w_l \cdot T_l \left(x, \frac{\alpha}{\pi}\right), \\
 & t_r = w_r \cdot T_r \left(x, \frac{\alpha}{\pi}\right), \\
 & S_{rs} = \frac{t_f + t_l + t_r}{\max(1, w_{\text{sum}})} + \max(1 - w_{\text{sum}}, 0).
 \end{aligned} \tag{5.9}$$

To achieve a smooth transition between the frontal, the lateral, and the radial scale texture on the 3D object based on the angles  $\alpha$  and  $\gamma$ , functions are used.

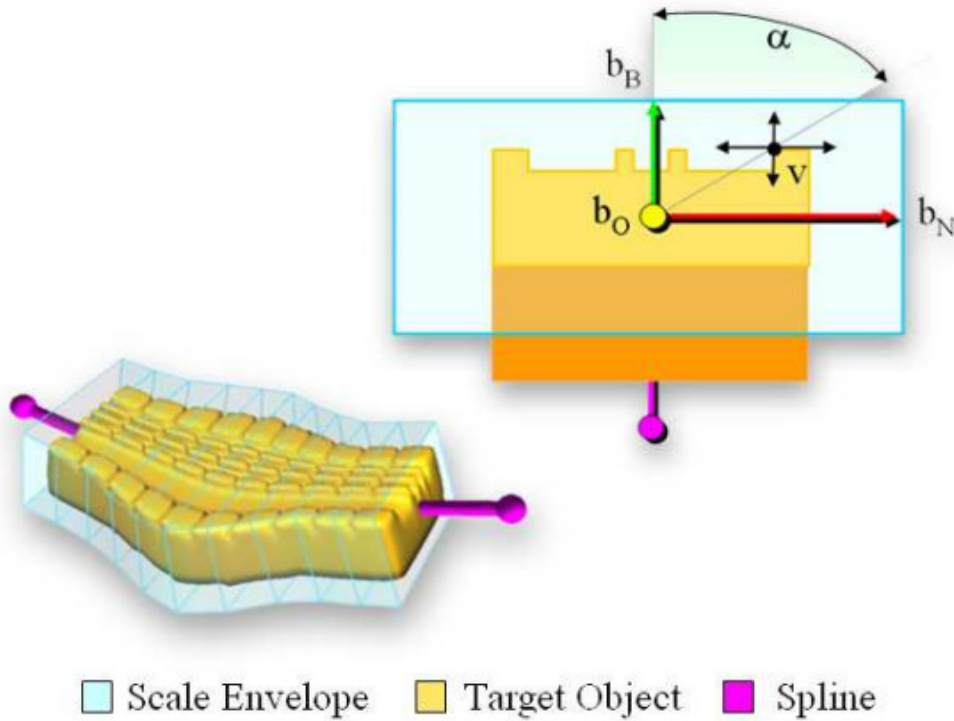
The weights for the frontal and the lateral texture are computed based on the angle  $\alpha$  and the radial weight based on the twist angle  $\gamma$ . In Eq. (5.9),  $w_{sum}$  represents the sum of all weights, and  $t_i$  ( $i=t,l,r$ ) the weighted texture samples. In order to preserve unity scaling for identity textures at any pose, Eq.(5.9) meets the following condition:

$$\forall x, \alpha \text{ such that } T_{f,l,r}\left(x, \frac{\alpha}{\pi}\right) = 1: S_{rs}(x, \alpha, \beta, \gamma) = 1, \text{ for all } \beta, \gamma. \quad (5.10)$$

## 5.6.2 Rectangular Scale Envelope (Style2)

### 5.6.2.1 Principle

The rectangular scale envelope is shown as Style 2 in **Fig. 5.7** and allows the design of the contour's deformations. The algorithm basically applies scaling of a vertex  $v$  in the two directions  $b_B$  and  $b_N$  independently, as shown in **Fig. 5.10**. This is very different from the former radial method that applies a concentric scaling.



**Figure 5.10** Rectangular Scale Envelope: The left side shows the application to an example object while the right side shows the scaling of  $v$  corresponding to  $b_B$  and  $b_N$

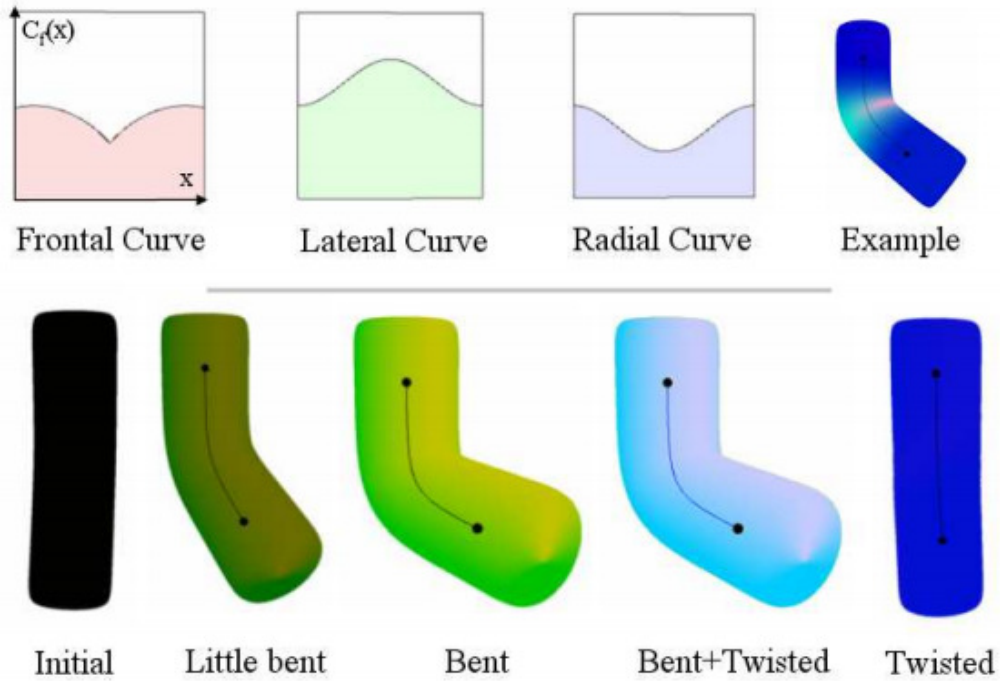
The purpose of the rectangular approach is the modeling of major frontal folds or lateral effects such as creating bulges. In order to design the deformation, the artist needs to have freedom to create three functions, which perform the scaling in the frontal, lateral, and radial direction.

The scaling is performed by two scaling functions  $S_i$  ( $i=f,l$ ), whereas  $S_f$  applies frontal scaling in direction of  $b_B$ , and  $S_l$  the lateral in direction  $b_N$ . More specifically, the deformation function  $D_{rect}$  can be written as follows:

$$D_{rect}(v') = \begin{bmatrix} v'_x \cdot S_l(x, \beta, \gamma) \\ v'_y \cdot S_f(x, \alpha, \beta, \gamma) \\ v'_z \\ 1 \end{bmatrix}. \quad (5.11)$$

The rectangular scaling is also calculated in spline space, and therefore  $v'$  instead of  $v$  is used. The spline-space representation  $v'$  of  $v$  is very handy, because the  $x$ -axis in spline space is along  $b_N$  and the  $y$ -axis along  $b_B$ . This allows an easy handling of the two scaling functions  $S_f$  (frontal) and  $S_l$  (lateral).

Similar to before, the angle  $\alpha$  defines the angle between  $v$  and  $b_B$  with respect to  $b_O$ . The two angles  $\beta$  and  $\gamma$  define the spline's pose.



**Figure 5.11** Rectangular Scaling: The upper row shows the three scale functions. The lower row shows the pose-dependent weights.

### 5.6.2.2 Rectangular Scale Functions

For designing the rectangular scale envelope, the artist can define three curves  $C_{i(i=f,l,r)}$ , each of which directly affects the contour of the deformed object in frontal ( $C_f$ ), lateral ( $C_l$ ) and radial ( $C_r$ ) directions, respectively.

**Fig. 5.11**, upper row shows the three curves that are used to create Style 2 in **Fig. 5.7**. In the example object on the right, the scale factors can be seen while bending. Green represents the lateral and red for the frontal scaling. The three curves are basically applied to the two scaling functions  $S_{f,l}$  the same three directions the textures  $T_{f,l,r}$  have been applied to  $S_{rs}$  before. The only difference to the previous radial scaling is that frontal and lateral scaling are treated separately. The calculation of the lateral weight  $w'_l$  can further be simplified as it does not depend on  $\alpha$  as opposed to the radial scaling; thus,  $S_f$  and  $S_l$  are as follows.

$$\begin{aligned}
 \forall C_f, C_l, C_r, c_f, c_l, c_r, w'_l, w_{sum1}, w_{sum2} &\geq 0, \\
 w'_l &= \beta, \\
 w_{sum1} &= w_f + w_r, \\
 w_{sum2} &= w'_l + w_r, \\
 c_f &= w_f \cdot C_f(x), \\
 c_l &= w'_l \cdot C_l(x), \\
 c_r &= w_r \cdot C_r(x), \\
 S_f &= \frac{c_f + c_r}{\max(1, w_{sum1})} + \max(1 - w_{sum1}, 0), \\
 S_l &= \frac{c_l + c_r}{\max(1, w_{sum2})} + \max(1 - w_{sum2}, 0).
 \end{aligned} \tag{5.12}$$

To achieve the pose-dependent weights that are used to define the importance of each of the curve  $C_{i(i=f,l,r)}$ , a couple of example poses are presented in **Fig. 5.11**, lower row, where the weights are indicated by color. The lower row shows the pose-dependent weights. Red corresponds to the weight of the frontal scale function, green to the lateral and blue to the radial. The intensity of each color represents the intensity value of the respective weight.

For the computation of  $S_f$  and  $S_l$  in Eq.(5.12), three new variables are introduced in addition to  $S_{rs}$ : is, the new lateral weight  $w'_l$ , the frontal weight  $w_{sum1}$  and the radial weight  $w_{sum2}$ .

Similar to the radial scaling function  $S_{rs}$ , the frontal and lateral scaling functions  $S_f$  and  $S_l$  also preserve unity scaling ( $S_{i(i=f,l)}$ ) for identity curves ( $C_{i(i=f,l,r)}$ ).

## 5.7 Deformation Styles and Spline Skinning

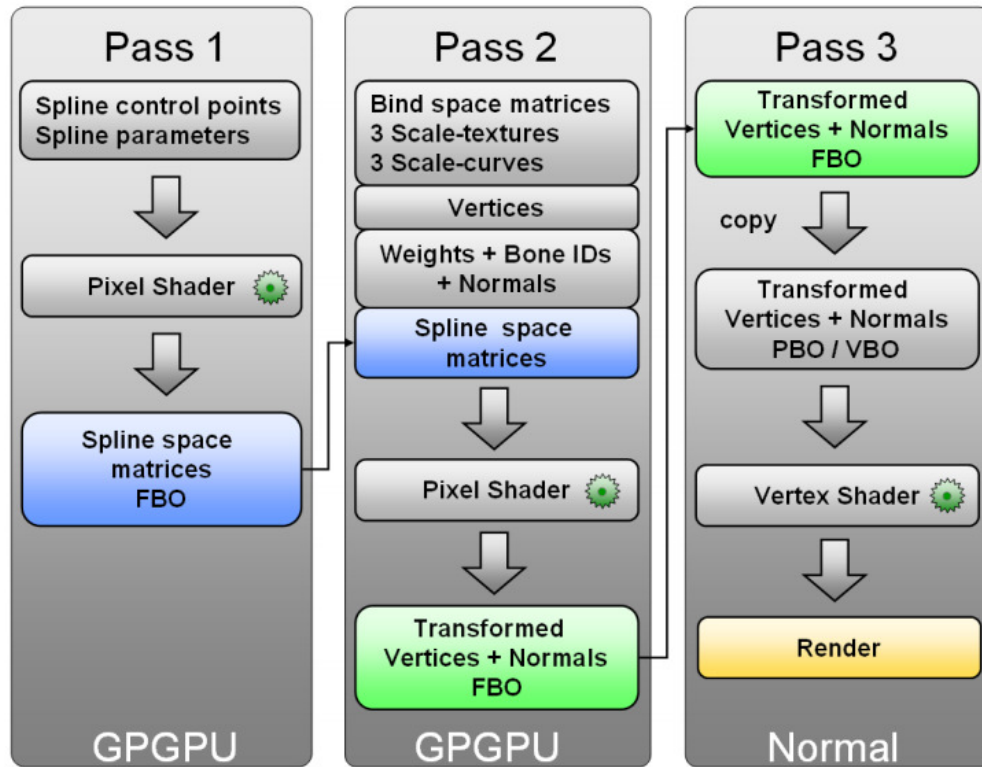
The vertex's eventual coordinates  $v_f$  calculated by the deformation are computed as follows ( $i$  represents the index of the spline curve):

$$v_f = \sum_{i=1}^n w_i \cdot B_i \cdot D_{rect}(D_{rad}(B_i^{-1} \cdot v)). \tag{5.13}$$



In Eq.(5.13), two different spline bases are used:  $B$  and  $B'$ . The basis of the actual pose that changes during the animation is defined by  $B$ , while  $B'$  defines the initial basis for binding that remains constant all the time. The radial deformation, the rectangular scaling, and finally the spline deformation are applied successively for each of the  $n=3$  splines (**Fig. 5.6**). Then the three partial results of each  $i$  are summed up using the skinning weights  $w_i$  for computing the final result.

In transition areas where two or more splines meet, styles are blended automatically, where the blended styles depend on the skinning weights.



**Figure 5.12** GPGPU based accelerations of the computations.

## 5.8 Fast Computation based on the GPU

To accelerate the computations, the OpenGL shading language GLSL and the render-to-vertex-buffer technique [63] are used. The render-to-vertex-buffer technique can be realized in OpenGL by using the frame-buffer-object (FBO) and the pixel-buffer-object (PBO) extension to perform the deformation.

To achieve an efficient implementation of the proposed GPU based algorithm, several improvements, as well as a couple of restrictions are made to the proposed system.

The proposed method consists of three passes, which are overviewed by **Fig. 5.12**. In the first pass, all spline matrices are computed and stored inside a temporary texture. In the second pass, the transformed geometry is computed, and in the third pass the geometry is visualized. The entire deformation, i.e. spline skinning and deformation styles, is applied in pass two.

One major key to this system's speed is the use of the Pixel Shader for computing the deformation rather than using the Vertex Shader. The reason is that not only the Pixel Shader is much more flexible but also it has much more computational power on most recent graphics cards. Using the Pixel Shader for general computations is a well-known technique that is often referred to as general purpose GPU computations (GPGPU), as the computed result of the Pixel Shader is stored in a temporary buffer, the FBO, in GPU memory and might be used for any purpose. The second optimization is of the separation of the evaluation of the spline basis  $B$  and the deformation of the vertices. Instead of computing the spline basis for each vertex individually, which causes heavy overhead for the computation, this implementation samples the spline at a fixed number of positions and stores the result into three temporary textures in the pass 1 in the **Fig. 5.12**. These textures are then passed to the per vertex computation of the deformation in the Pass 2 in **Fig. 5.12**. The pass 3 is responsible for visualizing the object by utilizing the transformed vertices sent from the pass two.

### 5.8.1 Pass 1

Pass 1 samples the spline at a fixed number of positions and stores the resulting spline bases  $B$  ( $3 \times 4$  matrix) into three temporary 16-bit RGBA floating point textures. **Table 5.1** shows all formats used here in detail.

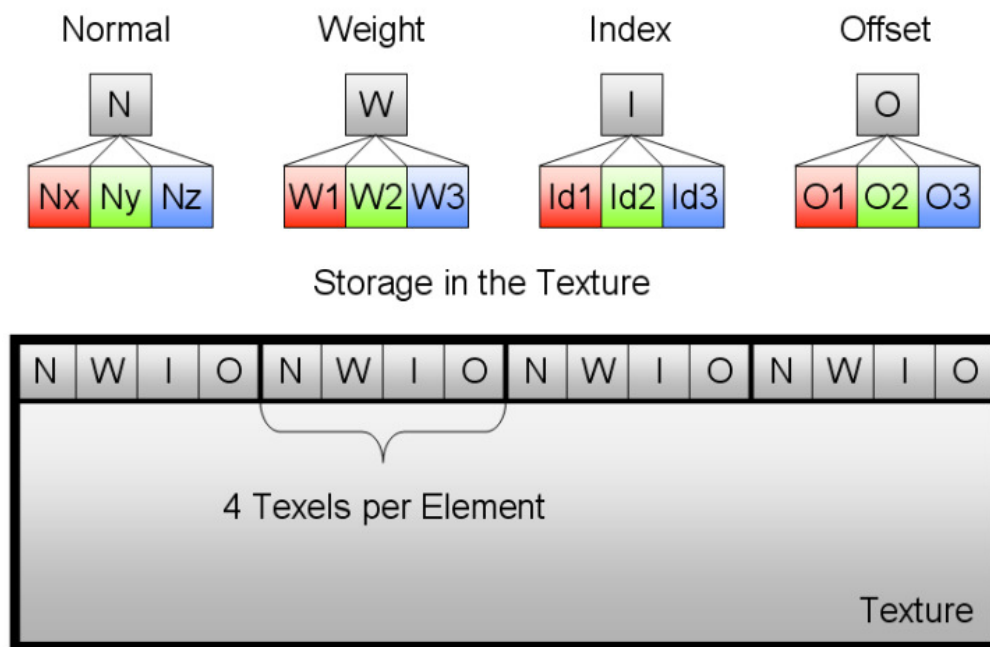
In OpenGL, a pixel shader can use inputs from constant program variables, from textures and from the vertex shader. The inputs for the Pixel Shader program are the spline control points and all the additional spline parameters such as twist and stiffness. These inputs are passed as textures, as this is the only possible solution to manage a large number of spline curves, while the matrix outputted from the pixel shader program is written into three frame buffer objects (FBOs) in parallel. Each FBO is linked to a texture to be used by the pass two.

**Table 5.1** Texture formats: Here an overview of the used texture and buffer formats.

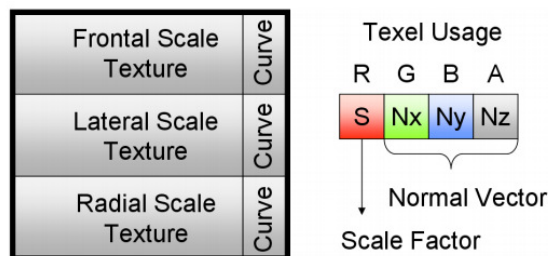
Identifier	Type	Data-type
<b>Spline parameters + control points</b>	Texture	RGBA float 32
<b>Vertices</b>	Texture	RGBA float 16
<b>Weights+BoneIDs+Normals+Offsets</b>	Texture	RGB byte 8
<b>Bind space matrices</b>	3xTexture	RGBA float 16
<b>Spline space matrices</b>	3xTexture	RGBA float 16
<b>Scale Textures+Functions</b>	3xTexture	RGBA float 16
<b>Transformed vertices+normals</b>	PBO/VBO	4x float 32

### 5.8.2 Pass 2

In the second pass, the deformation for each vertex is computed. Of all three passes, this pass is the most important, as it performs the spline skinning and the deformation styles, and also the most time-consuming. For the input, Pass 2 requires 12 textures (**Table 5.1**) that contain all data for the deformation. The 12 textures are divided into the following parameters: the inverse initial matrices  $B_i'^{-1}$  (3 textures), three scale textures (Section 5.6.1) and scale curves (Section 5.6.2)(together one texture), vertices  $v$  (one texture), vertex weights  $w_i$ , bone id's, normal vectors and spline offsets (stored in one texture), and, finally, the spline matrices  $B_i$  computed by pass 1 (3 textures).



**Figure 5.13** Shared Texture 1. Vertex normals (N), weights (W), bone indices (I) and spline offsets (O) are stored in one texture. The NWIO pattern applies for the empty space of the texture as well.



**Figure 5.14** Shared texture 2. Scale textures and scale curves are stored in one large texture. Each texel's RGBA value in the texture (left) is used as defined on the right side.

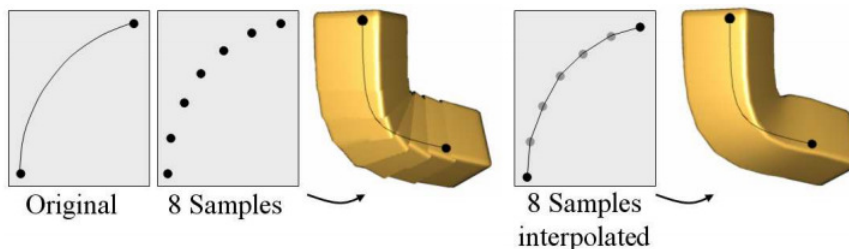
Concerning the texture that stores the vertex weights, bone ID's, vertex normals and spline offsets together, storing all of these elements in an interleaved way is used as optimal cache use as shown in **Fig. 5.13**. This is because the GPU does not read its RAM byte by byte. It reads at least 8 bytes at once from a memory location; therefore, the read accesses are minimized using an interleaved data structure.

The spline offsets are equal to the above mentioned position  $x$  on the spline. The created structure requires 4 pixels in the texture (texels).

In case of the texture sharing the three scale textures and the three scale curves, a small texture atlas as shown in **Fig. 5.14**, left side, is created to reduce the number of used textures.

The large texture consists of six parts: three scale textures and three scale curves. The three scale textures require most of the space, while the three scale curves only require one texel width at the right most column. The right half of **Fig. 5.14** shows how each texel of right column of the texture is used. The scale factor and the corresponding normal vector are stored together in one RGBA texel. This is very similar to storing a bump-map and a normal map in one texture. The stored scale factor is required for the deformation of the vertex, while the normal vector of the scale function as well as the scale curve is needed for deforming the normal vector.

A detail in the pixel shader implementation is the use of texture-filtering during sampling the spline matrices. Texture-filtering helps to efficiently use the graphics hardware to linearly interpolate between two spline matrices. Without this filtering, the spline would show strong aliasing artifacts, as can be seen in **Fig. 5.15**.



**Figure 5.15** Spline discretization: Pre-computing all splines is one of the key improvements in the implementation to increase the speed.

Concerning the deformation, the deformation style was only applied to the most significant spline curve of the three spline curves with the greatest weight. This significantly speeds up the computation without showing major drawbacks in the appearance of the deformed geometry.

### 5.8.3 Pass Three

The third pass is the rendering pass, which visualizes the computed geometry. First, the FBO that has been computed in Pass Two is copied to a pixel buffer object (PBO). The PBO is then used as a vertex buffer object (VBO) for visualizing the mesh as OpenGL vertex array.

## 5.9 Experimental Results and Discussion

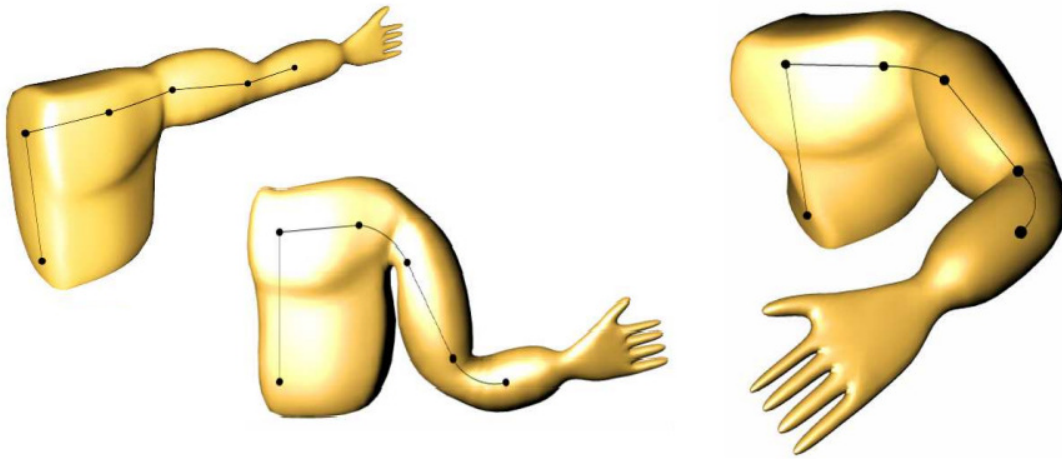
### 5.9.1 Experimental Conditions

For the experiments, an NVIDIA7800 GTX graphics card and a Pentium4 3.2 CPU was used. The data used for the experiments has been common triangle data 3D models.

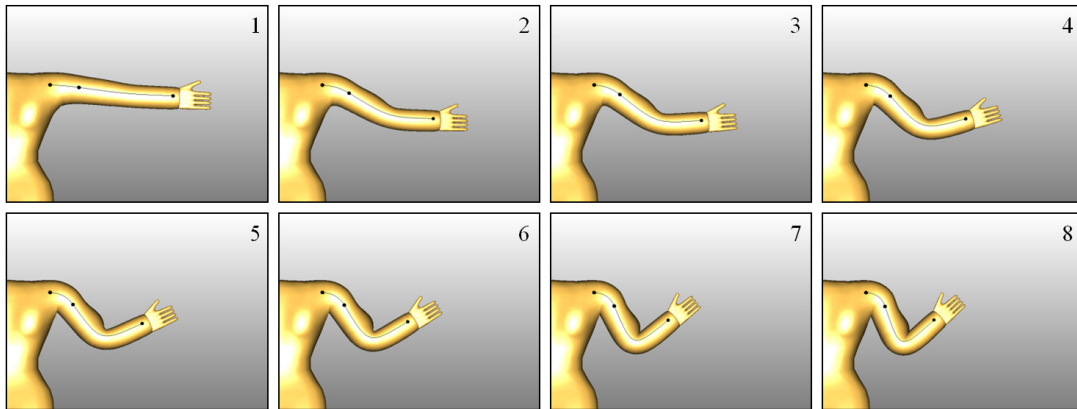
Various poses are created to show the flexibility of the proposed approach. **Figure 5.16** shows three poses: The bind pose (left), a basic bend deformation (middle) and a combination of bending and twisting (right). The underlying skeleton was created by attaching the arm-spline to the upper control point of the body-spline and the elbow-spline to the right control point of the arm-spline. Concerning the pose in the middle, even the large scale deformation near the shoulder does not lead to self-penetration. All the three poses show correct, seamless transitions between deformed and un-deformed areas. A simple animation sequence including a muscle deformation style is shown in **Fig. 5.17**.

### 5.9.2 Non-Collapsing Geometry

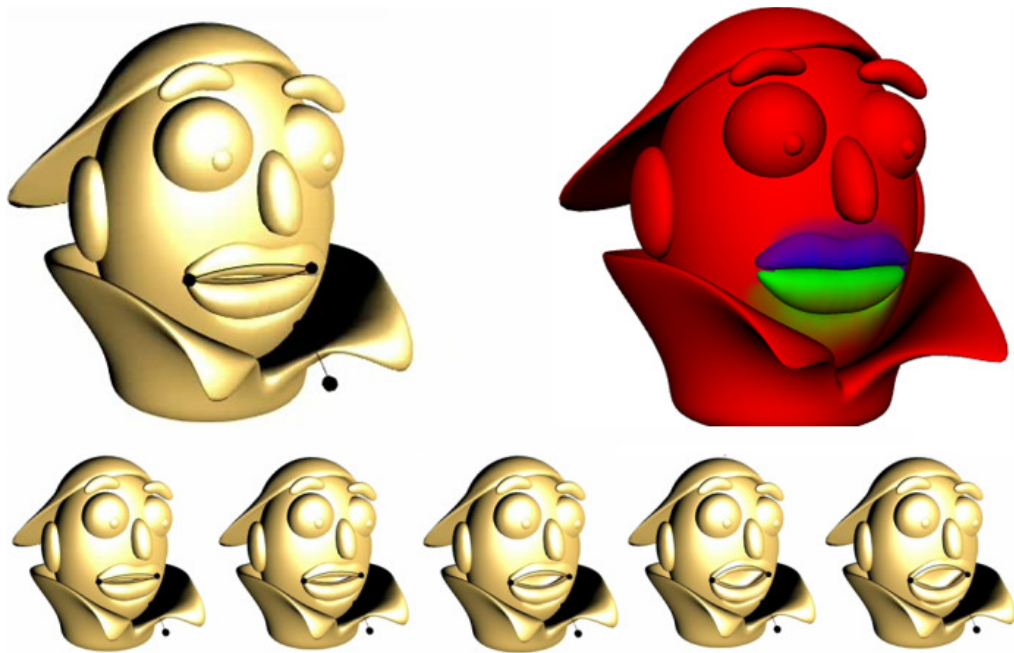
Non-collapsing geometry for the proposed spline skinning is shown in **Fig. 5.2**. There, spline aligned deformation is compared to FFD and SSD. Different from SSD and FFD, spline aligned deformations do not expose deformation artifacts. Further examples including bend and twist operations are shown in **Fig. 5.16**.



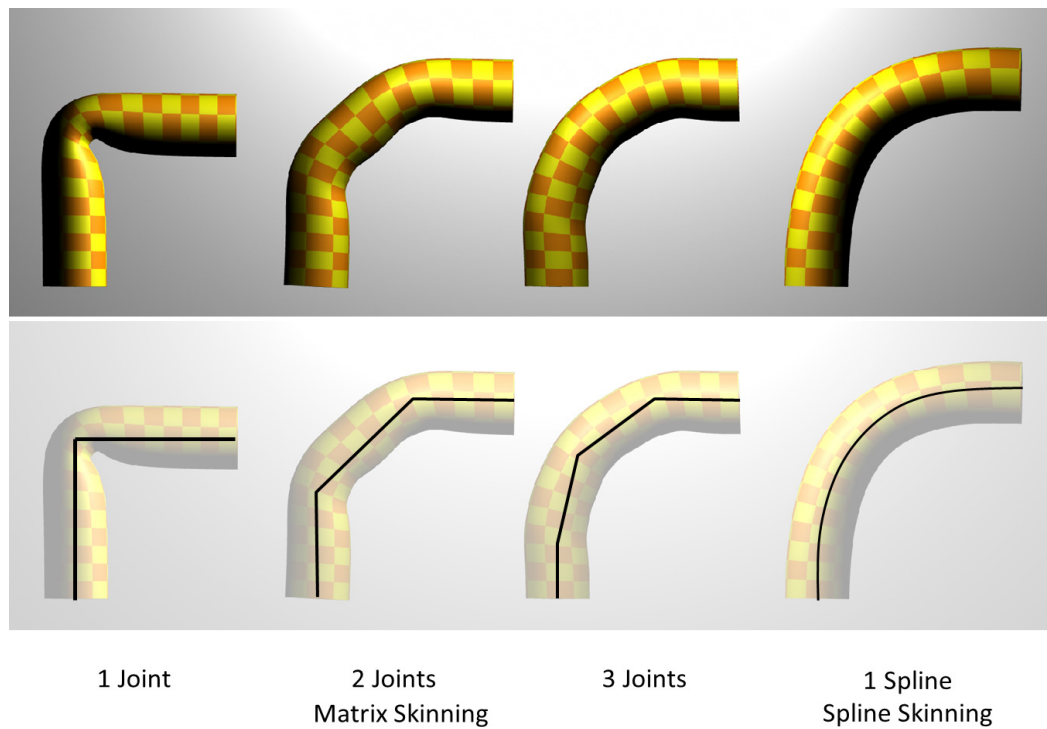
**Figure 5.16** Basic Spline Skinning: Three poses for an animated chest-arm-shoulder model. The binding pose (left), simple bend operation (middle), and, finally, bending combined with two twist operations, for the hand and for the body (right).



**Figure 5.17** Spline Skinning with a simple muscle deformation style in 8 frames.



**Figure 5.18** Facial Animation: Lips and cloth folds can be animated using spline skinning. Up-left the final animation and up-right the bind pose, where each color R,G,B is assigned to one spline. The lower part shows an animation sequence.



**Figure 5.19** Spline Skinning compared to matrix skinning for multiple joints.

### 5.9.3 Small Number of Control Points

It is possible to apply skinned skeletal animation efficiently to facial animation as **Fig. 5.18** demonstrates. Using just two splines, it is possible to represent the lips of a character. Other methods such as DQS or SSD require more joints for achieving the same result. In **Fig. 5.18**, the final result can be seen on the left, while the right side shows the initial binding pose including vertex weights for each spline.

A direct comparison from single spline to matrix skinning using multiple control joints to approximate a curve is demonstrated in **Fig. 5.19**.

### 5.9.4 Deformation Style Results

**Fig. 5.20** demonstrates metal-like behaviors, where a cuboid is used for creating snapshots from various poses.

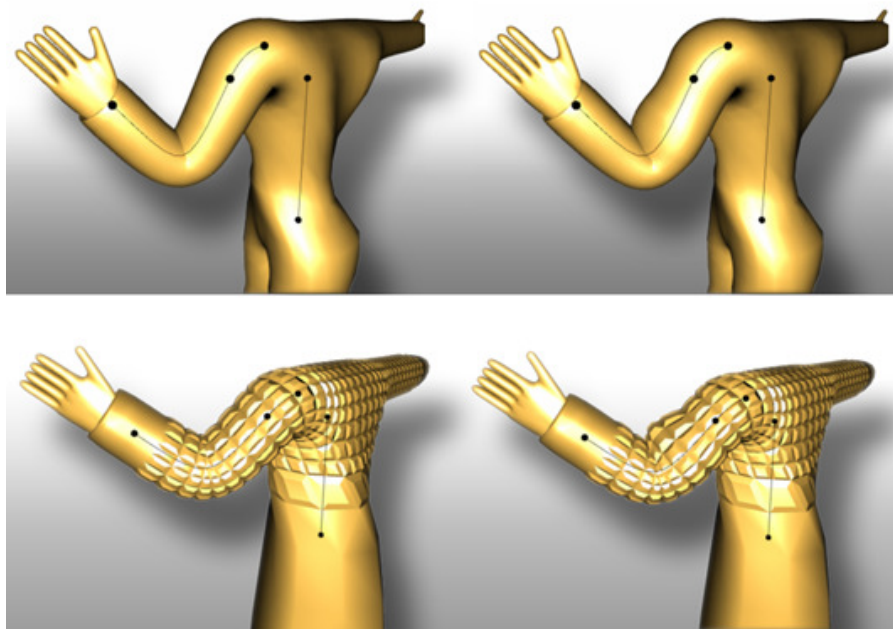
In **Fig. 5.21**, the proposed method's ability to design transferable muscle bulges is presented. The left side shows two characters using conventional spline-skinning, while the right side includes deformation styles.

Concerning deformation styles, further results can be seen in **Fig. 5.7**, where two styles are applied to three objects. Since the object's shape is very different, the geometry independence of the proposed method can be demonstrated successfully.

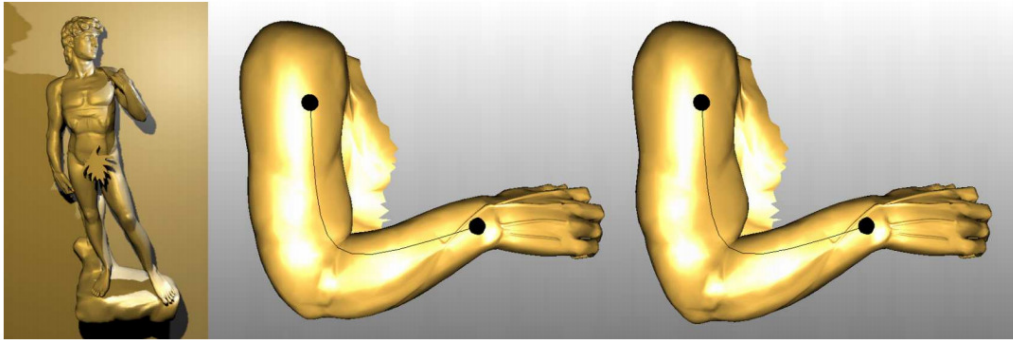




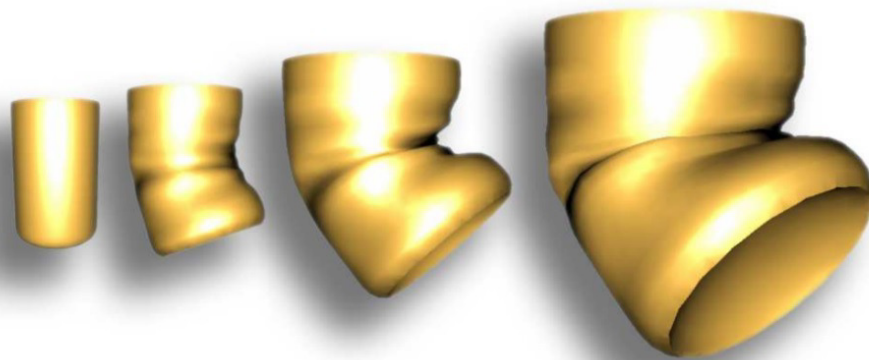
**Figure 5.20** Metal: This Figure shows the animation of designed metal, which smoothly deforms as the pose changes. Upper row: deformation styles are applied; lower row: spline skinning without deformation styles.



**Figure 5.21** Muscles: Created muscles can easily be applied to different characters simultaneously.



**Figure 5.22** Muscles on David:middle: without, right: with.



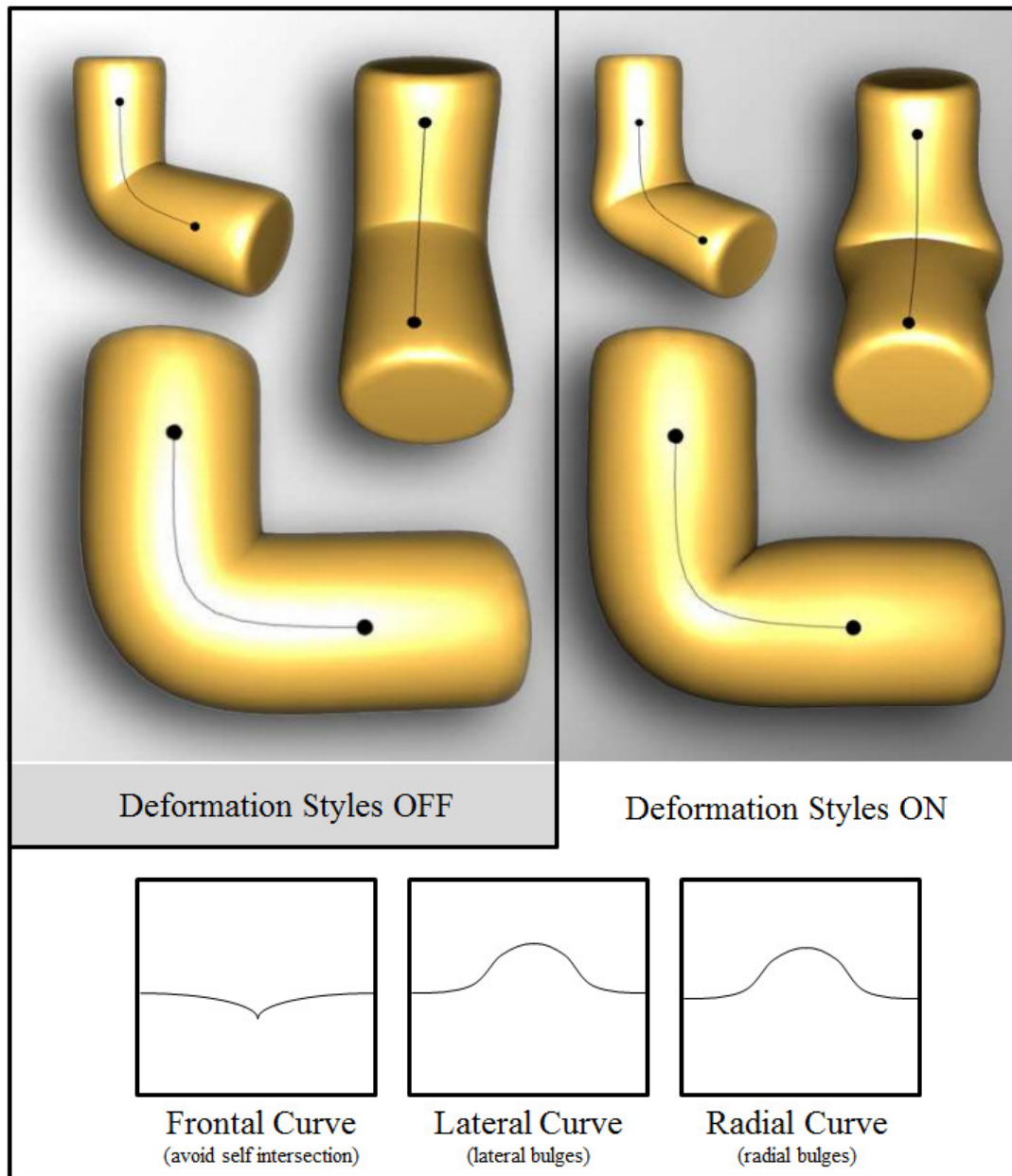
**Figure 5.23** Hollow materials: Here an animation of crunching an empty can.

It can be confirmed that the surface details of the character in the lower row work well along with the applied muscle-style. It is also possible to apply the style to an anatomically correct human body, as shown in **Fig. 5.22**. The same style is applied to the right arm of the David statue. The rightmost and middle images show the results with and without style, respectively.

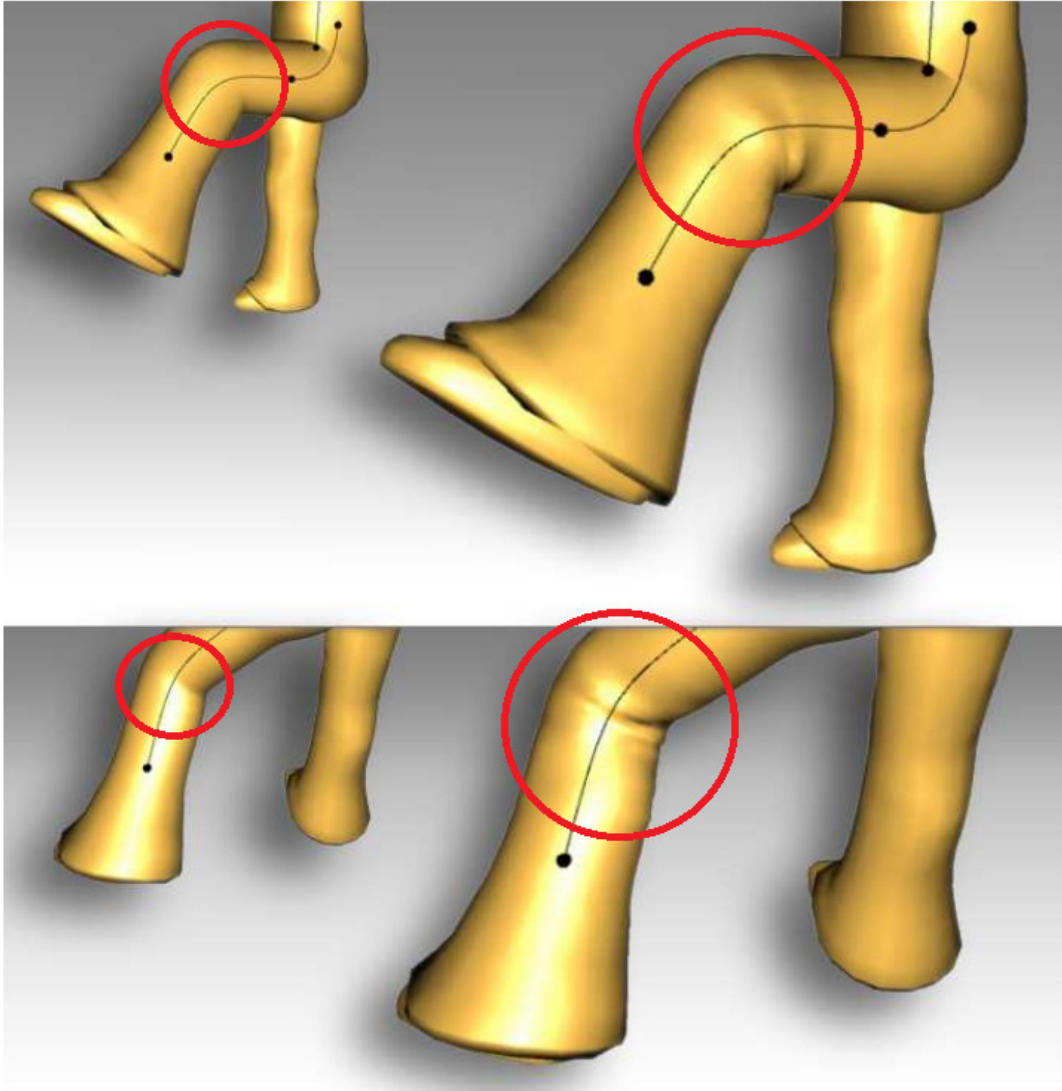
An example for modeling the deformation of a hollow material can be seen in **Fig. 5.23**. The realism is improved by displacing the spline origin  $b_o$  along the binormal  $b_B$  while applying  $D_{rad}$  and  $D_{rect}$ .

The successful prevention of self-intersections by using deformation styles can be seen in **Fig. 5.24**. It is an example for designing lateral bulges, where all the three curves that are shown in **Fig. 5.10** are modified by defining the curves as functions. The imitation of cloth is demonstrated by **Fig. 5.25**, where the applied cloth style showing wrinkles near the knee region can clearly be recognized. The textures that are used in the results were painted using conventional imaging tools. However, for an improved workflow, interactive texture-painting in a WYSIWYG<sup>66</sup> fashion might be advantageous.

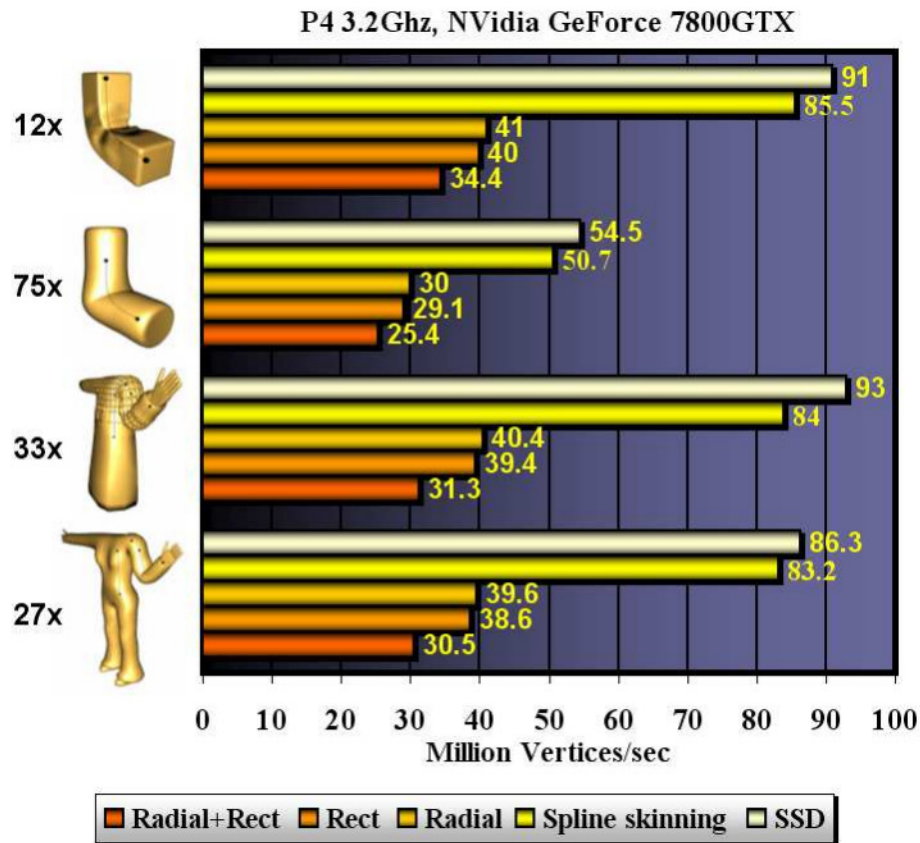
<sup>66</sup> WYSIWYG is the abbreviation of “What You See Is What You Get”.



**Figure 5.24** Self-intersections: Demonstrated are self-intersections (up-left) and the efficient removal of self-intersections (up-right) as well as modeled lateral bulges. The used curves are shown in the lower row..



**Figure 5.25** Cloth: This example shows the algorithm’s ability to imitate cloth-like wrinkles. Upleft and down-left: non-style version. Up-right and down-right: deformation styles version. The red circles show the affected region.



**Figure 5.26** Benchmark results: Spline skinning indicates basic spline aligned deformation, Radial adds Drad, Rect adds Drect and Rect+Radial adds both.

### 5.9.5 Computation Speed

The benchmark of the proposed method is shown in **Fig. 5.26**, where Radial+Rect indicates that radial and rectangular deformation styles are enabled, Rect indicates that only the rectangular deformation styles are enabled, Radial indicates that only the radial deformation styles are enabled, Spline Skinning indicates that deformation styles are turned off, and SSD represents matrix skinning. The number left to the objects represent the number of copies that were rendered simultaneously to achieve a vertex count of about one million for benchmarking each scene.

In case that only basic spline skinning is used, the proposed algorithm gets close to SSD and reaches the speed of 85 Million vertices per second, while the original SSD reaches 91 Million vertices per second.

If the proposed deformation method (Radial+Rect) is switched on, the speed decreases to 30M vertices per second, which is still satisfactory for real-time applications.



Even each scene used (one scene corresponds to one row in **Fig. 5.26**) for benchmarking consists of about one million vertices, the speed is not equal for all objects, which might be caused by an implementation issue of the proposed method. The rendering context of the frame-buffer-object (FBO) is switched for each object, which is a relatively expensive operation. However, there is no direct relation to computing the deformation.

The detailed timing is shown in **Table 5.2**. It is created to measure the proposed method's performance in more detail. The timing is investigated for two scenes – cuboid and cylinder (upper-most row and second upper-most row in **Fig. 5.26**).

**Table 5.2** Timing breakdown:

Scene	12x Cuboid	75x Cylinder
Vertices Object	12528	92526
Vertices Scene	12 x 12k = 0.94M	75 x 92k = 1.11M
Spline Samples	12 x 3 x 32 = 1152	75 x 3 x 32 = 7200
<b>Timing (Scene)</b>		
Spline Matrices	0.3 ms	0.3 ms
Deform (Radial)	7.8 ms	8.7 ms
Deform (Rect)	12.6 ms	9.7 ms
Deform (Spline)	5.5 ms	7.9 ms
Copy FBO to VBO	1.9 ms	4.9 ms
Render Scene	3.4 ms	4.5 ms
Total	31.7 ms	36.1 ms
Vertices/sec	34.4M	25.4M

The skinning based on three spline-curves, each of which are approximated by 32 samples, was also computed for each of the two objects. The scale texture resolution is  $32 \times 64$  pixels, concerning the timing breakdown, it can be seen that the major time consumption is caused by the two deformation methods (radial and rectangular deformations), followed by the spline-skinning (spline deformation). Pre-calculating the spline matrices requires much less time.

The step called *Copy FBO to VBO* is required by the OpenGL architecture and does a complete copy of all vertices from the FBO to the vertex-buffer-object (VBO). The final step for rendering the scene is relatively fast, as complex lighting evaluations were not included. Character scenes consisting of about 1 Million vertices were created for benchmarking, in order to get representative results.

**Summary:** The proposed spline-based skinned skeletal animation system outperforms the old version of the proposed spline skinning method [46] by factor three for the basic spline skinning without deformation-styles. The proposed method's speed without deformation-styles further gets

close to the performance of SSD, which is often referred to as the fastest skinned skeletal animation system. In case of deformation styles are added, the proposed method still shows a very competitive speed, as the vertex deformation rate remains high at 30 Million vertices per second on the utilized testing system.

### 5.9.6 Re-Usability

The proposed method allows the abstract design of pose-dependent deformation behaviors for the imitation of complex material deformations. Once designed, a style can immediately be applied to an arbitrary number of joints simultaneously, as shown in **Fig. 5.21**. This saves time for the artist during the modeling phase, and may further save memory during run-time, as each style needs to be stored only once.

### 5.9.7 Other contributions

The following items are not included in this chapters' goals, but it can be said that these are features of the proposed method.

**Simplicity:** The proposed algorithm is based on simple mathematics and does not contain complex data structures or the requirement of comprehensive mathematical libraries. It is assumed that the implementation is feasible in a reasonable time without complications. Furthermore, the designed deformation styles cover the complete pose-space of a joint and hence avoid the usage of radial basic functions for the interpolation between certain poses.

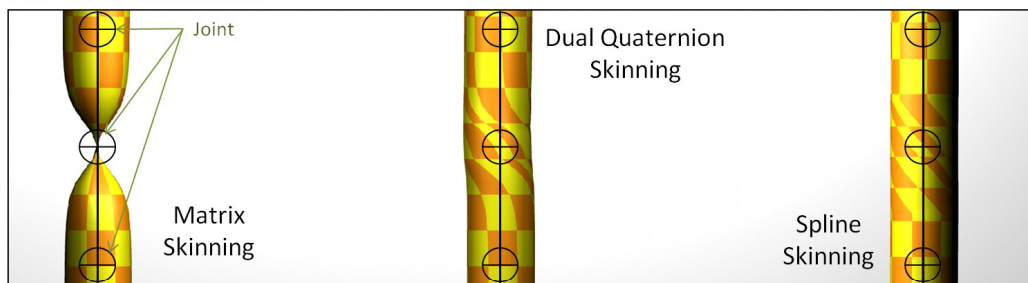
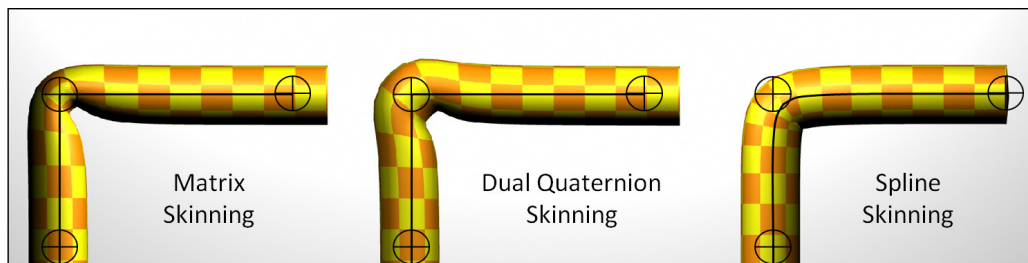
**Memory consumption:** In contrast to the GPU-based spline-aligned skeletal animation system [46], where each vertex and each normal of the animated mesh were required to be stored three times (once for each spline), the proposed algorithm requires them to be stored only once.

### 5.9.8 Comprehensive Evaluation

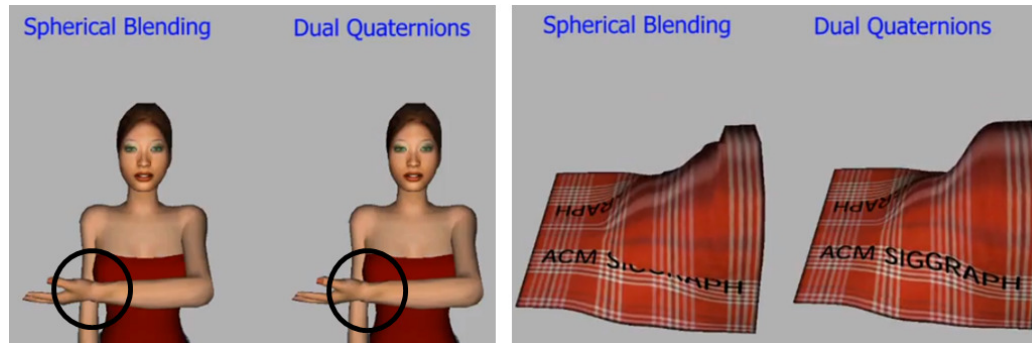
For the comprehensive evaluation, multiple related methods are compared: the proposed spline skinning, the proposed deformation styles, QS, DQS, matrix skinning, and matrix skinning with PSD. **Table 5.3** shows the general comparison in terms of speed, deformation artifacts, support to solve for multiple conventional joints, re-usability and support for custom pose dependent deformations. **Figure 5.27** and **Fig. 5.28** show the side by side comparison of matrix skinning (SSD), QS, DQS and the proposed spline-skinning.

**Table 5.3** Related methods and their features.

Method	Speed	Re-usable Deformation Styles	Custom Pose-Dependent Deformations	Deformation Artifacts	Solves for multiple conventional joints
Matrix Skinning (SSD)	<b>100%</b>	No	No	Candywrapper	No
QS (spherical interpolation)	34%	No	No	Discontinuity	No
QS (linear interpolation)	78%	No	No	Discontinuity	No
DQS	72%	No	No	Bulging	No
Spline Skinning	<b>92%</b>	No	No	<b>None</b>	<b>Yes</b>
Spline Skinning with Deformation Styles	<b>35%</b>	<b>Yes</b>	Yes	<b>None</b>	<b>Yes</b>
Matrix Skinning with PSD (GPU)	84%	No	Yes	None (if hand-modeled)	No

**180° Twist Deformation****90° Bend Deformation****Figure 5.27** SSD, DQS and Spline Skinning Methods Compared (1): upper row: twist operation, lower row: bend operation





**Figure 5.28** QS and DQS Skinning Methods Compared (2). Left: artifact-free twist operation; right: cloth deformation. Images with courtesy of Ladislav Kavan, Skinning with Dual Quaternions [20]

As for the performance compared in **Table 5.3**, it turns out that basic matrix skinning (SSD) is the fastest method, followed by spline skinning as second fastest, then PSD, QS (linear), DQS, spline skinning with deformation styles and QS (spherical).

#### Basic Skinning

Matrix skinning is fastest but not artifact-free, as shown in **Fig. 5.27**. Matrix skinning exposes the candy-wrapper effect as can be seen in **Fig. 5.27**, upper row, left for twisting deformations. It further exposes collapsing geometry, **Fig. 5.27**, lower row, left, for bending deformation.

Methods that improve upon this short-coming are QS and DQS. Both methods solve for the candy-wrapper artifact exposed by SSD by changing the interpolation domain from matrices to quaternions and dual quaternions. The efficient prevention of the candy-wrapper effect is demonstrated in **Fig. 5.28**, left side: the hand geometry does not collapse near the wrist during the twist operation for QS and DQS. The advancement of DQS over QS is demonstrated in **Fig. 5.28**, right side. Discontinuities for the cloth deformation exposed by QS are avoided with DQS.

DQS improves on matrix skinning and QS, but it exposes bulging artifacts while bending, as **Fig. 5.27**, lower row, center shows.

The proposed spline skinning, **Fig. 5.27**, left (upper and lower row), can solve for this issue and deform the geometry artifact-free at high performance.

As for PSD, it is an example based method built on top of SSD. It will show artifacts equal to SSD if no example poses are defined. It is up to the artist to modify each vertex for each pose individually to avoid any artifacts or create custom deformations. Therefore, even PSD is fast and able to solve for deformation artifacts, but it does not solve them automatically as spline skinning. Furthermore, PSD requires extra memory for each customized vertex and each pose. Creating an artifact-free deformation result for every pose using PSD requires a significant amount of memory.

### Custom Deformations

For achieving custom deformations PSD and the proposed Deformation Styles are compared in **Table 5.3**. As for the computation speed, PSD [58] is significantly faster than the proposed Deformation Styles. Further, both methods are able to avoid self-intersections and to create custom deformation styles such as muscles or cloth. Here, PSD is more flexible in terms of the vertex movement. While the proposed deformation styles method is limited to a concentric vertex movement with respect to the spline, PSD allows arbitrary movements of the vertices. However, in case of PSD, each vertex for each pose needs to be hand crafted, while deformation styles lets the artist create the deformations for all poses at once with three textures and three curves. Moreover, deformation styles allows one style template to be applied immediately to any number of target joints and target characters, which is a significant improvement over existing methods including PSD. It saves time for the artists and reduces the memory consumption as well.

In total, the proposed method is best.

## **5.9.9 Limitations**

### **5.9.9.1 Spline Skinning**

Even though the proposed method has many advantages in the design of high quality deformations, there are also certain limitations. The first one is the volume preservation. Since the proposed method is completely dependent on the artist's design, it is up to the artist to design a deformation that seems to preserve the volume or one that models a hollow material and does not preserve it.

### **5.9.9.2 Deformation Styles**

The second limitation is related to PSD. As opposed to PSD, which allows each vertex of a target mesh to be modeled pose-dependently in an arbitrary manner, the proposed deformation styles method can only affect vertices by the constraints of sweep based FFD; therefore, in an orthogonal direction to the spline curve, as mentioned in the previous section. The last issue is concerned with self-intersections. The proposed method neither computes nor automatically prevents them; however, the artist can create deformation styles that give the impression of an intersection-free deformation.

## 5.10 Conclusion

This chapter has proposed two skeletal animation methods that can achieve the four goals, which are categorized into two groups: (1) artifact-free, fast computation, and few control joints, and (2) re-usability. To achieve the goal group (1), this thesis has proposed a Spline Skinning based on spline aligned deformations and blending multiple spline curves using vertex weights. To achieve the goal group (2), this thesis has incorporated deformation styles into the above-mentioned Spline Skinning so that pose dependent deformations can be designed by defining three scale textures for detailed deformations and three scale curves, which can be re-used for skeletal objects with any number of joints.

Experiments for exploring the validity of the proposed method were conducted using multiple triangle models together with manually rigged skeletons. The experimental results and discussions are summarized as follows.

### Goal group (1)

- **Artifact-free:** The proposed method does not expose deformation artifacts, while related methods expose the following problems: Matrix skinning exhibits collapsing geometry and the candy-wrapper artifact, DQS exhibits unnatural bulges for bending deformations, and QS exhibits discontinuities for complex deformations.
- **Fast computation:** As a result of bench marks, it turns out that SSD is the fastest, and the proposed method is second fastest (90 to 96% of SSD), but faster than QS and DQS.
- **Reducing the number of joints:** The experimental results show that the proposed method achieves a complex facial lip animation with only two spline curves and spine deformations by only one spline.
- **As a result of the comprehensive evaluation** it is confirmed that the proposed method works best for the above-mentioned three goals.

### Goal group (2)

- **Re-usability:** Experimental results show that the proposed method can apply a deformation style to two different characters successfully.

For the proposed spline skinning, it is limited to a number of three control points. Future work might include experiments with more control points. For deformation styles, the limitation is in terms of computation speed as the proposed method is significantly slower than PSD. Future work

might include improving the speed and further applying the deformation textures to the surface of the geometry as bump map.

## **Chapter 6. Conclusion and Future Work**

### **6.1 Conclusion**

The goal of this thesis is to improve methods for visualizing common elements in video game applications by overcoming the limitations of existing methods. The common elements which this thesis deals with include terrain, static objects, and skeletal animation. This thesis explores how to improve each of the three common elements as follows.

Chapter 1 is the introduction of this thesis. This chapter claims that this thesis explores how to improve the visualizations of the common elements to be embedded into a 3D engine.

Chapter 2 explains 3D engines and their relationship to Game Engines as well as 3D engines' functions.

Chapter 3 has proposed a nested CB (Clip-Box) based approach that is able to generate procedural volumetric terrains with unlimited size without pre-computation in parallel to visualization. A CB consists of a cubic regular grid of voxels and the corresponding triangulation. Nested Clip-Boxes that utilize mathematical functions that define the terrain allow the immediate and pre-computation free generation and concurrent visualization of arbitrary sized volume data. Experiments using data generated from terrain functions, data from existing volume data sets and height-map data prove that the proposed method can generate terrains with unlimited size without pre-computation and then visualize them concurrently.

Chapter 4 has proposed a raycasting based method for the fast visualization of complex RLE compressed voxel data scenes without consuming much memory. The proposed method improves the original voxel forward projection algorithm in several ways so that complex scenes are efficiently visualized with low memory consumption is achieved. For low memory consumption, this thesis proposes a new data structure for RLE. For fast computation, the proposed method is completely optimized for highly parallel single instruction multiple data processing on the GPU and uses newest NVIDIA CUDA technology. Experimental results show that the proposed method and Qspat consume least memory, and that the proposed method and some related methods are fastest and faster than Qsplat. The comprehensive evaluation based on these results indicate that the proposed method is best in terms of the goals of this chapter.

Chapter 5 has proposed two skeletal animation methods that can achieve the four goals, which are categorized into two groups: (1) artifact-free, fast computation, and few control joints, and (2) re-usability. To achieve the goal group (1), this thesis has proposed a Spline Skinning based on spline aligned deformations and blending multiple spline curves using vertex weights. To achieve the goal group (2), this thesis has incorporated deformation styles into the above-mentioned Spline Skinning. Results of experiments that confirm whether the goal group (1) is achieved show that the proposed method achieve artifact-free and small number of control joints as opposed to related works and that the computation speed is the second fastest. Comprehensive evaluation based on these results indicates that the proposed method is best. Results of experiments that confirm whether the goal group (2) is achieved show that the proposed method can apply a deformation style to two different characters successfully.

## 6.2 Future Work

In case of procedural volumetric terrain, Chapter 3, additional studies to include better frame-to-frame caching of the generated geometry as well as exploring different methods for the visualization could be conducted. As graphic cards become more versatile, ray-casting the volume data rather than converting the data into polygons becomes an option.

The future work of voxel ray-casting, Chapter 4, includes exploring ways of streaming the voxel data into the GPU, in order to allow large and complex scenes to be visualized.

For skeletal animation and deformation styles, Chapter 5, further research can be carried out on rendering detailed surfaces with few polygons. For that, deformation styles might not only be applied to the geometry, but also the surface by using bump-mapping e.g.

For long term, based on this thesis' achievements, the proposed modules could be embedded in modern 3D engines such as CryEngine 3.

## Bibliography

- [1] L. Williams, "Casting curved shadows on curved surfaces," *SIGGRAPH '78*, vol. 12, no. 3, pp. 270-274, 1978.
- [2] F. Crow, "Shadow Algorithms for Computer Graphics," *SIGGRAPH 1977*, vol. 11, no. 2, pp. 242-248, 1977.
- [3] Z. a. T. N. Brawley, Parallax Occlusion Mapping: Self-Shadowing, Perspective-Correct Bump Mapping Using Reverse Height Map Tracing., 2004, pp. 135-154.
- [4] V. MAGNENAT-THALMANN, "From early draping to haute couture models: 20 years of research. The Visual Computer 21, 8–10 (2005), 506–519.," 2005.
- [5] M. S.Rusinkiewicz, "'QSplat: A Multi-resolution Point Rendering System for Large Meshes" Siggraph 2000, 343 – 352," 2000.
- [6] J. R. Wright and J. L. Hsieh, "A voxel-based, forward projection algorithm for rendering surface and volumetric data," in *VIS'92: Proceedings of the third conference on Visualization*, 1992.
- [7] D. W. C.Erikson, "'HLODs for faster display of large static and dynamic environments",SI3D '01, 111—120," 2001.
- [8] F. E.Gobbetti, "'Far Voxels: A Multi-resolution Frame-work for Huge Complex 3D Models", Siggraph 2005, 878 – 885," 2005.
- [9] C. a. N. F. a. S. M. a. G. S. a. E. E. Crassin, "Interactive Indirect Illumination Using Voxel Cone Tracing," *Computer Graphics Forum. (Proceedings of Pacific Graphics 2011)*, 2011.
- [10] P. a. K. D. a. R. W. a. H. L. F. a. F. N. a. T. G. A. Lindstrom, "Real-time, continuous level of detail rendering of height fields," *SIGGRAPH '96*, 1996.
- [11] M. D. M. C. M.-W. Duchaineau, "'ROAMing terrain: real-time optimally adapting meshes", VIS'97, 81—88," 1997.
- [12] H. F.Losasso, "'Geometry Clipmaps: Terrain Rendering Using Nested Regular Grids", Siggraph 2004, 769-776," 2004.
- [13] M. T. Ryan Geiss, "'NVIDIA Demo Team Secrets – Cascades", technical presentation at the Game Developers Conference 2007," 2007.
- [14] E. G. S. M. J. G. Adrien Peytavie, "Arches: a Framework for Modeling Complex Terrains," *Eurographics 2009, Volume 28, pp.457–467*, 2009.
- [15] J. Olick, "Beyond Programmable Shading," in *Course at Siggraph*, 2008.
- [16] J. H. Clark, "Hierarchical Geometric Models for Visible Surface Algorithms," *Communications of the ACM*, vol. 19, no. 10, pp. 547-554, 1976.
- [17] P. K. a. J. F. a. F. D. a. D. Bartz, "'Lossless Volume Data Compression Schemes, SimVis 2007, pp. 169-182," 2007.
- [18] L. R. T. D. MAGNENAT-THALMANN N., "Joint-dependent local deformations for hand animation and object grasping. In Proceedings of Graphics Interface '88 (1988), pp. 26–33.," 1988.
- [19] Z. J. Kavan L., "Spherical blend skinning: a real-time deformation of articulated models. In SI3D '05: Proceedings of the 2005 symposium on Interactive 3D graphics and games , pp. 9–16.," 2005.
- [20] C. S. O. C. Z. J. Kavan L., "Dual quaternions for rigid transformation blending. Technical report TCD-CS-2006-46, Trinity College Dublin," 2006.
- [21] V. Kajiin, *Screen space ambient occlusion*, CryTek, 2007.
- [22] M. N. S. N. Pattanaik, "Real-Time Realistic Rendering," *ASC2002 23rd Army Science Conference 2002*, 2002.
- [23] J. v. Waveren, "ID tech 5 Challenges," in *Siggraph 2009*, 2009.



- [24] H. Nguyen, *Gpu gems 3*, Addison-Wesley Professional, 2007.
- [25] M. P.Prusinkiewicz, ""A Fractal Model of Mountains with Rivers", *Graphics Interface '93*, 174-180," 1993.
- [26] B. Gregorski, ""Interactive View-Dependent Rendering of Large IsoSurfaces", *Visualization 2002*, 475 – 484," 2002.
- [27] F. E. F. F. R. P.Cignoni, ""Adaptive tetrapuzzles: efficient out-of-core construction and visualization of gigantic multiresolution polygonal models", *Siggraph 2004*, 796—803," 2004.
- [28] P.Lindstrom, ""Out-of-Core Construction and Visualization of Multiresolution Surfaces", *SI3D '03*, 2003, 93-102," 2003.
- [29] G. G. F. P. C. L. Borgeat, ""GoLD: Interactive Display of Huge Colored and Textured Models", *Siggraph 2005*, 869 – 877," 2005.
- [30] H. W.E.Lorensen, ""Marching cubes: A high resolution 3D surface construction algorithm", *SIGGRAPH '87*, 163—169," 1987.
- [31] C. M. C.C.Tanner, ""The clipmap: A virtual mipmap", *ACM SIGGRAPH 1998*, 151-158," 1998.
- [32] R. a. A. H. G. G.M.Treece, ""Regularised marching tetrahedra: improved iso-surface extraction ", *Computers and Graphics 1998*, 23(4):583-598," 1998.
- [33] T. J. a. F. L. a. S. S. a. J. Warren, ""Dual contouring of hermite data", *SIGGRAPH '02*, 339—346," 2002.
- [34] C. D. Hansen and C. R. Johnson, "Visualization Handbook," in *Visualization Handbook*, Academic Press, 2004, pp. 7-11.
- [35] P. G. Lacroute, ""Fast volume rendering using a shear-warp factorization of the viewing transformation", *SIGGRAPH '94*, pp.451--458, 1994," 1994.
- [36] I. W. S. P. C. H. Aaron Knoll, ""Interactive Isosurface Ray Tracing of Large Octree Volumes", *IEEE Symposium on Interactive Ray Tracing*, pp.115-124," 2006.
- [37] C. Crassin, F. Neyret, S. Lefebvre and E. Eisemann, "GigaVoxels: ray-guided streaming for efficient and detailed voxel rendering," in *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, 2009.
- [38] A. Lindenmayer, "Mathematical models for cellular interaction in development," *Journal of Theoretical Biology*, vol. 18, pp. 280-315 1968.
- [39] J. R. W. a. J. C. L. Hsieh, ""A voxel-based, forward projection algorithm for rendering surface and volumetric data", *Visualization'92* pp.340--348," 1992.
- [40] J. Amanatides and A. Woo, "A fast voxel traversal algorithm for raytracing," *Eurographics'87*, pp. 3-9, 1987.
- [41] M. Mitting, ""Advanced Real-Time Rendering", *3D Graphics and Games Course*, Chapter 8, pp.113-115, *SIGGRAPH 2007*," 2007.
- [42] T. A. a. S. L. a. T. Karras, "Understanding the Efficiency of Ray Traversal on GPUs -- Kepler and Fermi Addendum," *NVIDIA Technical Report*, no. NVR-2012-02, 2012.
- [43] C. Crassin, "GigaVoxels:A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes," p. 207, 2011.
- [44] P. S. R. SEDERBERG T. W., "Free-form deformation of solid geometric models. j-COMP-GRAPHICS 20, 4, 151–160.," 1986.
- [45] F. E. SINGH K., "Wires: a geometric deformation technique. In *SIGGRAPH '98* (1998), pp. 405–414.," 1998.
- [46] O. J. Forstmann S., "Fast skeletal animation by skinned arc-spline based deformation. In *Eurographics Short-Papers* (2006), pp. 1–4.," pp. 1-4, 2006.
- [47] S. A. Z. J. J. YANG X., "Curve skeleton skinning for human and creature characters: Research articles. *Comput. Animat. Virtual Worlds* 17, 3/4 (2006), 281–292.," 2006.
- [48] S. D. M. P. CORNEA N., "Curveskeleton applications. In *Visualization*, 2005. VIS 05. IEEE (2005), pp. 95–102.," 2005.
- [49] K. M.-S. YOON S.-H., "Sweep-based freeform deformations. *Computer Graphics Forum (Eurographics'06 proc.)* 25, 3 (2006), 487–496.," 2006.

- [50] Y. S.-H. C. J.-W. S. J.-K. K. M.-S. J. B. Hyun D.-E., "Sweep-based human deformation. The Visual Computer 21, 8-10 (2005), 542–550., " 2005.
- [51] M. G. M. K. L. Botsch M., "Primo: Coupled prisms for intuitive surface modeling. In Eurographics Symposium in Geometry Processing (2006), pp. 11–20., " 2006.
- [52] J. D. W. J. B. L. S. A. C. M.-P. Decaudin P., "Virtual garments: A fully geometric approach for clothing design. Computer Graphics Forum (EG'06 proc.) 25, 3, 625–634., " 2006.
- [53] G. S. C. B. D. T. P. Z. Capell S., "Interactive skeleton-driven dynamic deformations. In SIGGRAPH '02: ACM SIGGRAPH 2002 Papers, pp. 586–593., " 2002.
- [54] J. D. S. A. POPA T., "Materialaware mesh deformations. In SMI '06: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06), p. 22., " 2006.
- [55] H. J. S. J. L. X. B. H. G. B. S. H.-Y. ZHOU K., "Large mesh deformation using the volumetric graph laplacian. In SIGGRAPH '05: ACM SIGGRAPH 2005 Papers, pp. 496–503., " 2005.
- [56] P. J. SUMNER R. W., "Deformation transfer for triangle meshes. In SIGGRAPH '04: ACM SIGGRAPH 2004 Papers, pp. 399–405., 2004.
- [57] C. M. F. N. LEWIS J. P., "Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In SIGGRAPH '00, pp. 165–172., " 2000.
- [58] N. U. TAEHYUN RHEE J. L., "Realtimeweighted pose-space deformation on the gpu. Computer Graphics Forum (Eurographics'06 proc.) 25, 3 (2006), 439–448., " 2006.
- [59] C. F. R. I. C. F. SLOAN P.-P. J., "Shape by example. In SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics, p. 135–143., " 2001 .
- [60] M.-T. VOLINO P., "Fast geometrical wrinkles on animated surfaces. In 7th International Conference in Central Europe on Computer Graphics and Visualization (1999), pp. 55–66., " 1999.
- [61] J. A. T. K. Jing F., "Modeling wrinkles on smooth surfaces for footwear design. Computer-Aided Design 37, 8, 815–823, " 2005.
- [62] C. M.-P. Larboulette C., "Real-time dynamic wrinkles. In CGI '04: Proceedings of the Computer Graphics International (CGI'04), pp. 522–525., " 2004.
- [63] . T. Scheuermann, "Render to vertex buffer with d3d9, " in *Siggraph 2006 Course 3: GPU Shading and Rendering*, 2006.
- [64] J. Ewins, "MIP-map level selection for texture mapping, " *IEEE Transactions on Visualization and Computer Graphics*, vol. 4, no. 4, pp. 317-329, 1998.

## ***Publication List***

### ***[ Journals and Transactions ]***

Sven Forstmann and Jun Ohya, "Efficient, High-Quality, GPU-Based Visualization of Voxelized Surface Data with Fine and Complicated Structures", IEICE Transactions on Information and Systems, Vol.E93-D, No.11, pp.3088-3099, (Nov. 2010). (Related to Chapter 4)

Sven Forstmann and Jun Ohya, "Visualizing Large Procedural Volumetric Terrains Using Nest Clip-Boxes", GITS/GITI 紀要 2010 - 2011 (早稲田大学国際情報通信研究科／国際情報通信研究センター), 査読付き論文, pp.51-61, (2011.10). (Related to Chapter 3)

### ***[ International Conferences ( Reviewed ) ]***

Sven Forstmann, Yutaka Kanou, Jun Ohya, Sven Thuerling and Alfred Schmitt, "Real-Time Stereo by using Dynamic Programming", 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Workshop: Real Time 3D Sensors and Their Use, CD-ROM Proceedings (8 pages), June 27 - July 2, 2004.

Sven Forstmann and Jun Ohya, "Visualization of large Iso-Surfaces based on nested Clip-Boxes", SIGGRAPH2005 Posters, Conference Select CD-ROM Disc 2, 1 page, 2005.7-8.

(Related to Chapter 3)

Sven Forstmann, Jun Ohya, Waseda University, "Fast Skeletal Animation by skinned Arc-Spline based Deformation", Eurographics 2006 , pp.1-4, (Sep. 2006) (Related to Chapter 5)

Sven Forstmann, Jun Ohya, Artus Krohn-Grimberghe, Ryan McDougall, "Deformation Styles for Spline-based Skeletal Animation", SCA '07: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer animation, pp.141-150, Aug. 2007.

(Related to Chapter 5)

### ***[ Presentations at Domestic Academic Meetings ( Non-Reviewed ) ]***

Sven Forstmann, Alfred Schmitt, Sven Thuerling, Jun Ohya, Yutaka Kanou, "Realtime Stereo Vision", PRMU2003.

Sven Forstmann and Jun Ohya, "Interactive Visualization of Large ISO-Surfaces", 国際情報通信研究公開研究発表会予稿集, pp.8-9, 2005.10. (Related to Chapter 3)

Sven Forstmann, 大谷 淳, "Interactive Visualization of Large ISO-Surfaces", FIT2005 (第4回情報科学技術フォーラム), pp.365-366, (2005.9). (Related to Chapter 3)

Sven Forstmann and Jun Ohya, "Visualization of Large Caved Terrains", 電子情報通信学会技術研究報告, Vol. 105, No. 608, ITS2005-64, pp.101-106, (2006.2). (Related to Chapter 3)

Sven Forstmann and Jun Ohya, "Procedural Spline-Skeletons for Organic Structures and Adaptive Architecture", 2007 年電子情報通信学会総合大会, A-16-23, p.336, (Mar. 2007). (Related to Chapter 5)

Sven Forstmann, Jun Ohya, "Skeletal Animation by Spline aligned Deformation on the GPU", 電子情報通信学会技術報告, Vol. 106, No. 608, IE2006-283, pp.47-52, (Mar. 2007). (Related to Chapter 5)

Sven Forstmann, Jun Ohya, "Visualizing run-length-encoded volume data on modern GPUs", IE2007-317 PRMU2007-301, pp.355-358. (Related to Chapter 4)

Sven Forstmann, Jun Ohya, "Parallel Forward Projection of Large Voxel-Volumes on the GPU", IEICE Tech. Rep., CS2008-41, pp. 11-16, Dec. 2008. (Related to Chapter 4)

**[ Co-authored Publications ]**

Igor Goncharenko, Mikhail Svinin, Sven Forstmann, Yutaka Kanou, and Shigeyuki Hosoe, "On the Influence of Arm Inertia and Configuration on Motion Planning of Reaching Movements in Haptic Environments", Proc. of World Haptics 2007: The Second Joint Eurohaptics Conference and Symposium on Haptic Interfaces for Virtual Environment and Teleoperator Systems, pp.33-38, Mar. (2007), Tsukuba, Japan

Li Jen Chen, Jun Ohya, Shunichi Yonemura, Sven Forstmann, Yukio Tokunaga: Prompter ".  
Based Creating Thinking Support Communication System That Allows Hand-Drawing. HCI (2)  
2009: 783-790