CONCOMITANT HIERARCHY CONSTRUCTION AND RENDERING OF LARGE POINT CLOUDS

Vinícius da Silva

Tese de Doutorado apresentada ao Programa de Pós-graduação em Engenharia de Sistemas e Computação, COPPE, da Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Doutor em Engenharia de Sistemas e Computação.

Orientadores: Claudio Esperança
Ricardo Guerra Marroquim

Rio de Janeiro
Maio de 2018

CONCOMITANT HIERARCHY CONSTRUCTION AND RENDERING OF LARGE POINT CLOUDS

Vinícius da Silva

TESE SUBMETIDA AO CORPO DOCENTE DO INSTITUTO ALBERTO LUIZ COIMBRA DE PÓS-GRADUAÇÃO E PESQUISA DE ENGENHARIA (COPPE) DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE DOUTOR EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

Examinada por:

RIO DE JANEIRO, RJ – BRASIL
MAIO DE 2018

Silva, Vinícius da

Concomitant hierarchy construction and rendering of large point clouds/Vinícius da Silva. – Rio de Janeiro: UFRJ/COPPE, 2018.

XIII, 53 p.: il.; 29, 7cm.

Orientadores: Claudio Esperança

Ricardo Guerra Marroquim

Tese (doutorado) – UFRJ/COPPE/Programa de Engenharia de Sistemas e Computação, 2018.

Referências Bibliográficas: p. 46 – 53.

1. Graphics data structures and data types. 2. Object hierarchies. 3. Viewing algorithms. 4. Computational Geometry and Object Modeling. 5. Computer Graphics. I. Esperança, Claudio *et al.* II. Universidade Federal do Rio de Janeiro, COPPE, Programa de Engenharia de Sistemas e Computação. III. Título.

*To the ones I love most: God,
my parents Gilberto and Gedilsa,
my wife Priscila, my brothers
Thiago and Lucas, my nephews
Júlia and Davi and my church.*

Acknowledgments

I would like to thank CNPq for the scholarship offered during my Ph.D.. Without it it would be impossible for me to focus full time on my research.

To the professors at the Computer Graphics Laboratory (LCG/UFRJ) and advisors Claudio Esperança and Ricardo Marroquim for all the help and expertise shared. To the memory of professor Antonio de Oliveira who inspired many students with his simple but logic way to view, discuss and teach advanced Math and Physics topics.

To the external members of the examining board who kindly accepted to share their experience and time to evaluate my work.

To all the coworkers at the laboratory for the good times and the meeting discussions in all these years.

To my parents Gilberto and Gedilsa who created me in a very healthy environment. To my brothers Thiago and Lucas who helped me to develop as a person. To my beloved wife Priscila for all the love and understanding of all these years of hard work. To my nephews Júlia and Davi whose birth changed my way to see life. To God and the people of my church who helped me to keep maintaing hope even on face of failure and difficulties on research.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Doutor em Ciências (D.Sc.)

CONSTRUÇÃO DE HIERARQUIA E RENDERIZAÇÃO CONCOMITANTES DE NUVENS DE PONTOS EXTENSAS

Vinícius da Silva

Maio/2018

Orientadores: Claudio Esperança
Ricardo Guerra Marroquim

Programa: Engenharia de Sistemas e Computação

As abordagens atuais para renderizar nuvens de pontos extensas envolvem um estágio de pré-processamento extenuante em que uma estrutura de dados hierárquica é criada antes da renderização. Esses algoritmos não consideram apresentar os dados antes da conclusão da construção hierárquica. Neste trabalho, apresentamos OMiCroN – acrônimo em inglês para Criação de Hierarquia Multipasso e Obliqua enquanto Navegando – que é o primeiro algoritmo capaz de exibir imediatamente renderizações parciais da geometria, desde que a esta seja disponibilizada na ordem de Morton como um fluxo. Ao usar um algoritmo de ordenação parcial, o OMiCroN é capaz de ordenar os dados, construir a hierarquia e renderizar em paralelo, o que pode começar assim que o primeiro prefixo ordenado dos dados estiver disponível. Na prática, a primeira renderização parcial só precisa aguardar a leitura de toda a geometria não ordenada a partir do disco. OMiCroN também é o primeiro algoritmo a implementar uma abordagem de renderização de baixo para cima, fornecendo detalhes completos desde o início, de forma diferente das abordagens de cima para baixo atuais que começam a partir de uma visão geral dos dados, fornecendo detalhes completos mais tarde no processo. OMiCroN também pode ser usado para apresentar feedback de renderização do processo de criação da hierarquia. Essas características são possíveis usando o "corte oblíquo", uma nova estrutura de dados que separa porções renderizáveis das porções não renderizáveis da hierarquia.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Doctor of Science (D.Sc.)

CONCOMITANT HIERARCHY CONSTRUCTION AND RENDERING OF LARGE POINT CLOUDS

Vinícius da Silva

May/2018

Advisors: Claudio Esperança
Ricardo Guerra Marroquim

Department: Systems Engineering and Computer Science

Current approaches for rendering large point clouds involve a strenuous preprocessing stage where a hierarchical data-structure is created before rendering. These algorithms do not consider presenting data before the hierarchy construction is finished. In this work we present OMiCroN – Oblique Multipass Hierarchy Creation while Navigating – which is the first algorithm capable of immediately displaying partial renders of the geometry, provided the geometry is made available in Morton order as a stream. By using a pipeline sort algorithm, OMiCroN is capable of parallel data sorting, hierarchy construction, and rendering, which can start as soon as the first sorted prefix of the data is available. In practice, the first partial rendering must only wait for the whole unsorted geometry to be read from disk. OMiCroN is also the first algorithm to implement a bottom-up rendering approach, providing full detail at the beginning, unlike current top-down approaches, which start from an overview of the data, providing full detail later in the process. OMiCroN can also be used to present rendering feedback of the hierarchy creation process. These features are made possible using an "oblique cut", a novel data structure that separates the renderable from the non-renderable portions of the hierarchy.

Contents

List of Figures **x**

List of Tables **xiii**

1 Introduction **1**

2 Literature review **5**

2.1 Background . . . 5

2.1.1 Point Clouds and Splatting . . . 5

2.1.2 Space Filling Curves . . . 6

2.1.3 Hierarchical Spatial Data Structures . . . 7

2.1.4 Rendering Front . . . 11

2.2 Large point cloud rendering . . . 11

3 OMiCroN - Oblique Multipass Hierarchy Creation while Navigating **14**

3.1 Oblique Hierarchy Cuts . . . 14

3.1.1 Operator *Concatenate* . . . 15

3.1.2 Operator *Fix* . . . 16

3.1.3 Placeholders . . . 18

3.1.4 Sequence of Oblique Hierarchy Cuts . . . 18

3.2 Oblique Hierarchy Cut Front . . . 19

3.2.1 Insertion of new nodes . . . 20

3.2.2 Substitution of placeholders . . . 21

3.2.3 Choice of substitution level . . . 21

3.3 Sample OMiCroN implementation . . . 21

4 Experiments **30**

4.1 Rendering latency tests . . . 30

4.2 Hierarchy creation and rendering . . . 32

4.3 Rendering parameters . . . 41

5 Conclusion **44**

Bibliography **46**

List of Figures

1.1 A sequence of renderings at different stages of hierarchy creation. OMiCroN is used for sorting, creating and rendering the hierarchy in parallel. 3

2.1 Example of space filling curves for a square. 6
2.2 Z-Order and Morton code illustrative example. 7
2.3 8x8 Bitmap represented by a compressed quadtree. Numbers denote order of nodes. As in [1]. 8
2.4 A recursive subdivision of a cube into octants (left) and the corresponding octree (right). As in [2]. 9
2.5 k-d tree decomposition for the point set {(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)} (as in [3]) and the resulting kd-tree (as in [4]). 10
2.6 A BSP tree for correct drawing of line segments using the painter's algorithm. As in [5]. 11
2.7 Rendering Front example. 12

3.1 OMiCroN overview. A renderable hierarchy is maintained while inserting incoming nodes in parallel. This cycle is repeated until the whole hierarchy is constructed. 15
3.2 The function $span(x)$. It returns the morton code of the rightmost descendant of x in a supposedly full subtree. In other words, it returns the maximum possible morton code of a subtree rooted by x at the morton curve of level l_{max}. In this example, $span(root) = 63$, as the nodes marked in red show. 16
3.3 An example of Oblique Hierarchy Cut and operators *concatenate* and *fix*. A cut C is defined by a delimiting Morton code m_C and a list of roots per level L_C (a). The *concatenate* operator inserts new roots x_0 and x_1 at the deepest level l_{max}, resulting in cut C' (b). Invariant 1.5 does not hold for C' and operator *fix* is used to traverse subtrees bottom-up, inserting nodes until the boundary S is reached. It results in cut C'' for which all invariants hold. 17

3.4 Oblique Hierarchy Cut progression. As operators *concatenate* and *fix* are used, the cuts sweep their associated hierarchy H. Placeholders are marked with a P and the ones used but removed while processing lists bottom-up are also marked with a red X. 19
3.5 Example of valid Oblique Hierarchy Cut Front. The direction of the blue arrows indicate the order restriction imposed by invariant 2.1. The fact that all nodes in the front are not roots in L_C ensures invariant 2.2. 20
3.6 OMiCroN multithreaded prototype, composed of four parallel Tasks: a) Sort, b) Fix, c) Front tracking, d) GPU Load. All threads are represented as purple boxes. a) is responsible of sorting data chunks, creating nodes at level l_{max} and applying concatenate on them (Algorithm 1). b) applies fix, also feeding the front with new placeholders and leaves. This task is parallelized per level. Created placeholders and leaves are first pushed to a per-thread segmented buffer so fix can be fully parallel. After all threads finish, the segmented buffers are compacted and sent to the front (Algorithm 3). c) evaluates the front, rendering, prunning and branching nodes. Before evaluation, it concatenates at the front's end the placeholders in the buffer and chooses a substitution level. A placeholder is substituted at front evaluation if its ancestor leaf is found in the list of the chosen substitution level (Algorithm 5). d) consists of short-lived theads responsible of loading nodes into the GPU. 25

4.1 The datasets used in experiments. Rendered using OMiCroN. 31
4.2 Impact of the number of sort chunks. After a constant time spent reading the input (blue), the first chunk is sorted (red), starting the parallel hierarchy creation and rendering (orange). The first column in all charts corresponds to the case where all input is sorted before the hierarchy creation begins. 33
4.3 Comparison of hierarchy creation with and without parallel rendering. Sorted data is streamed directly from disk. The overhead imposed by parallel rendering is between 20% (David) and 34% (St. Matthew). . 34
4.4 Hierarchy creation over time. Sorted data is streamed directly from disk, parallel rendering is enabled and leaf collapse is disabled unless pointed otherwise. 35

4.5 Rendering comparison of hierarchies with different leaf collapse and parent to children point ratio parameters. The final reconstructions are very detailed even at close range and the differences when the leaf collapse is turned on are almost imperceptible. 37
4.6 Rendering comparison of hierarchies with different leaf collapse and parent to children point ratio parameters. The final reconstructions are very detailed even at close range and the differences when the leaf collapse is turned on are almost imperceptible. 38
4.7 Rendering comparison of hierarchies with different leaf collapse and parent to children point ratio parameters. The final reconstructions are very detailed even at close range and the differences when the leaf collapse is turned on are almost imperceptible. 39
4.8 Rendering comparison of hierarchies with different leaf collapse and parent to children point ratio parameters. The hierarchy for Duomo suffers from lack of density when leaf collapse is turned on because the dataset itself has smaller density in comparison with the others. . 40

List of Tables

4.1 Relationship between the algorithm reconstruction parameters – leaf collapse, parent to children ratio – and memory footprint, total hierarchy creation times, and average CPU usage per frame. 36

4.2 Relationship between the worklist size and performance indicators: front insertion delay, front size, hierarchy construction time and average CPU usage per frame. Numbers refer to the David dataset, no leaf collapse and point ratio 0.25. 36

4.3 Time comparison between OMiCroN with and without parallel rendering and [6]. I/O times are separated for preprocessing and creation tasks. Expections are tasks in [6] which does not report I/O times explictly. In these cases task times also include I/O. The most important lines are titled "Creation", which contain the time needed to create the hierarchy after input and preprocessing, and "Until render", which show the total time before any image is generated, including I/O. They show how OMiCroN minimizes data evaluation delays. . . 42

4.4 Best values for leaf splat tangents size, per model. 43

4.5 Best value ranges for splat tangent multipliers from hierachy level 7 to 3. 43

Chapter 1

Introduction

In recent years, the improvements in acquisition devices and techniques have led to the generation of huge point clouds using automatic or semi-automatic approaches. This phenomenon has also led to a significant increase in complexity of tasks and computational demand when dealing with such large datasets.

Rendering large point clouds is a well-studied topic in Computer Graphics. The usual approach to solve this problem is to build a hierarchical multiresolution datastructure for spatial organization of the data, allowing fast culling and smart choices of subsets that fit a specified resolution or level-of-detail (LOD). However, preprocessing times are incommensurate with the actual time to render them. This preprocessing may require many minutes or even hours for large datasets, depending on the algorithm and the parameters used. Long preprocessing times delay the evaluation of the acquired data, which in turn may compromise an efficient workflow involving dataset acquisition and evaluation. Many digitization campaigns, specially those in situ, would greatly benefit from fast data evaluation, involving just minimal preprocessing delays. Additionally, a good feedback of the hierarchy creation process would be very useful for implementors and systems that use the point cloud hierarchy as a tool for other operations. For instance, a user might stop the hierarchy building after noticing problem in the data as the initial samples are viewed.

The time required for building a hierarchy is related to the nature of the algorithm, which can be incremental, bottom-up or top-down. In particular, incremental methods can start building the structure as soon as the first primitive is available, as is the case of a Bounding Volume Hierarchy (BVH), that can be constructed incrementally by insertion. On the other hand, bottom-up and top-down methods require information about the entire dataset before the building process starts. Nonetheless, a significant drawback of incremental methods is that the quality of the resulting structure is totally dependent on the data order, which may result in very unbalanced hierarchies [7, 8]. This property is the main reason why such algorithms have been mostly left aside by the scientific community. One way to improve the quality of the hierarchy is to shuffle the data beforehand [9]; yet, shuffling the data defeats the main purpose of using incremental methods, i.e., starting the hierarchy construction as early as possible. Another idea to improve the hierarchy created by incremental methods is to adaptively rebuild parts of the hierarchy [8], which can result in hierarchies with quality comparable to the best top-down methods but is less effective for some datasets. Moreover, the authors acknowledge that even its parallel version can be several times slower than an earlier top-down algorithm [10]. Therefore, excluding incremental methods, the time needed for reading a dataset is clearly a lower bound for the preprocessing time required by a high quality hierarchy constructor for large datasets.

Current approaches do not support a concomitant visualization of the hierarchy building process. This is understandable if the build process is too time-consuming, but even supposing that the creation time is relatively short, rendering and building would necessarily have to synchronize their access to the incomplete hierarchy. This synchronization problem is not trivial. Assuming unsorted data, a rendering thread must cope with the fact that any node of the hierarchy can be changed by a point insertion operation as it performs a traversal or sends data to GPU. Conversely, a hierarchy construction thread must preclude other threads from accessing regions of the hierarchy being modified. If we consider that a point insertion can change the hierarchy structure by creating new nodes and deleting others, and that current rendering algorithms apply optimization structures for hierarchy traversal acceleration, we may grasp the depth of the problem. Naive approaches to solve it result in too many possibilities for inconsistencies between the hierarchy creation, traversal acceleration structure management and GPU data loading. The locks needed to synchronize all these tasks might lead to prohibitive delays, mainly in the rendering thread which is expected to achieve realtime performance.

In this work we present OMiCroN (Oblique Multipass Hierarchy Creation while Navigating), a novel stream-based approach for constructing Octrees for large point clouds. Our method aims at presenting data to the user as soon as possible, and, to the best of our knowledge, it is the first capable of sorting a point cloud, constructing a LOD-capable hierarchy, and partially rendering it in parallel. By using the concept of *oblique cuts*, the synchronization between the rendering and hierarchy building threads is restricted to a small set of nodes. Furthermore, differently from previous methods, our algorithm is able to partially render large point clouds in full detail from the very start of the process (bottom-up rendering, instead of top-down). In practice, the first full-detail partial rendering must only wait for the whole unsorted geometry to be read from secondary storage. Consequently, OMiCroN can start rendering right after the dataset is read, leading to the benefits of early data evaluation and the best possible feedback of the process. It is also suitable for creating Octrees on-the-fly in environments where data transferring is a bottleneck. In this case, transferring the input point data, constructing a hierarchy on-the-fly, and rendering it is clearly preferable to transferring a complete spatial hierarchy several times larger. Figure 1.1 shows the overall idea of OMiCroN's objective.

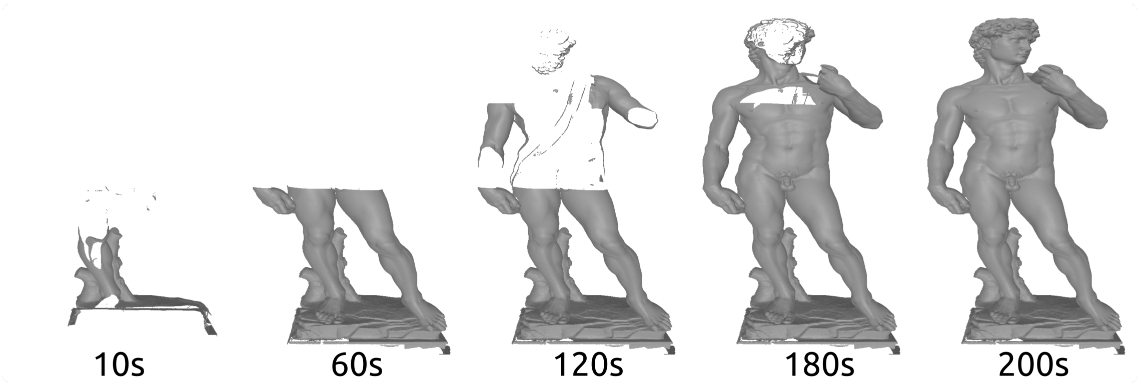


Figure 1.1: A sequence of renderings at different stages of hierarchy creation. OMiCroN is used for sorting, creating and rendering the hierarchy in parallel.

The technical contributions of this work are:

- it introduces the concept of Hierarchy Oblique Cuts, that allows parallel data sorting, spatial hierarchy construction and rendering;
- it restricts the preprocessing of input data to a very fast and flexible Morton code based partial sort;
- it allows for on-the-fly Octree construction for large point clouds;
- it renders full detail data from the very beginning as a consequence of bottom-up hierarchy construction;
- it provides immediate visual feedback of the hierarchy creation process.

This thesis is organized in the following manner. In Chapter 2 a literature review is presented. We start in Section 2.1 with an overview of the necessary background for discussing the problem at hand and for describing OMiCroN. In Section 2.2 we review the currently stablished techniques for construction of hierarchical structures for large point clouds with the objective to scope OMiCroN's scientifical contribution. In Chapter 3 we describe our method, starting with an overview and proceeding with an in-depth discussion about the two central concepts of Hierarchy Oblique Cuts and Oblique Hierarchy Cut Fronts presented in Sections 3.1 and 3.2, respectively. In Section 3.3 we present the parallel version of the OMiCroN algorithm, describing a proof-of-concept application for processing and rendering large point clouds. In Chapter 4 we describe the experiments to measure the preprocessing, rendering and memory efficiency of the algorithm. Finally, conclusions, limitations and future work directions are presented in Chapter 5.

Chapter 2

Literature review

In this Chapter we review current literature about large point cloud rendering, discussing why a novel approach such as OMiCroN is necessary. We also state its contributions to the field. Starting by a review of background concepts necessary for a better understanding of the problem, we develop to a discussion about multiresolution and LOD structures, establishing a comparative argument about the lack of a stream-and-feedback-based algorithm in the field.

2.1 Background

Our work depends on several major concepts: Point Clouds and Splatting; Space Filling Curves, in particular the Z-order curve; Hierarchical Spatial Data Structures, in special Octrees; and Rendering Fronts. The theory behind them is summarized in this section.

2.1.1 Point Clouds and Splatting

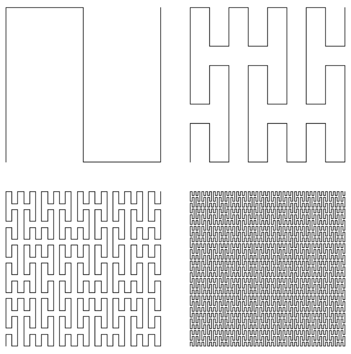
While the use of points as rendering primitives was introduced very early in Computer Graphics [11, 12], their widespread adoption only occurred much later, as discussed on extensive survey literature [13–18]. Many algorithms were presented from that period on, proposing improved image quality by changes in the kernel logic, better spatial management by the use of multiresolution and LOD structures, and integration of the out-of-core paradigm, resulting in systems that can handle extremelly large point clouds.

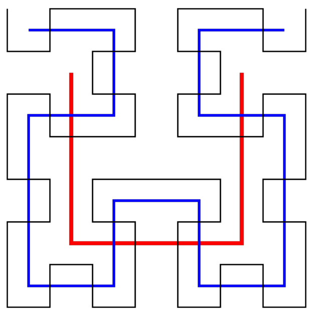
A naive algorithm for rendering point-sampled surfaces is to project them on an image and assigning the point colors to the pixels closest to their projections. The problem of such algorithm is the presence of holes in the resulting image unless the surface is sampled at pixel scale in all surfaces. Additionally, the final color of a pixel would be dependent of projection order in that algorithm.

Splatting is a technique created to solve the aforementioned problems, rendering images from points independently of density and projection order. The concept was introduced in the scope of point-based rendering by [19] and [20]. [19] brought the concept from the scope of volumetric rendering [21], while [20] used the term Surfel to define the same idea. The final popularization of splats came with the proposal of a robust mathematical framework for rendering splats [22]. The main idea is quite simple and consists of distributing the properties of the points among their projected neighborhood. The contribution of each point in a pixel must be weighted by the distance between the pixel and the point center so more distant points contribute less than closer ones. This is done by using a bidimensional gaussian filter centered at the point position, which is excelent for such task because of its quick decay from its center.

2.1.2 Space Filling Curves

Space filling curves are curves whose range contains an entire square, cube or more generally an n-dimensional hypercube. A curve of this type defines a map from its associated hypercube into an 1-dimensional interval. Its construction is iterative and each iteration results in a curve containing more of the hypercube. In the limit of the iterations the curve contains the entire associated space. Peano, Hilbert, Sierpiński, Z-order, Schoenberg and Jordan are examples of such curves. They differ in how the iterations are performed. Figure 2.1 shows the first iterations of Peano, Hilbert and Z-order curves for a square. A detailed discussion about space filling curves can be found in [23]. In this work we will focus on Z-order curves.


(a) Peano curve. As in [24].


(b) Hilbert curve. As in [25].

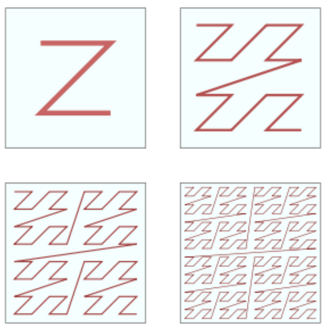
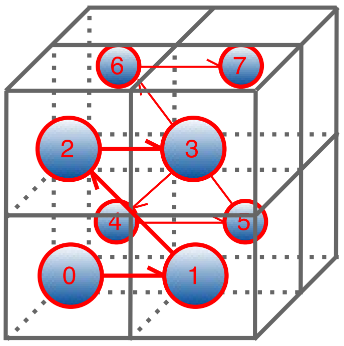
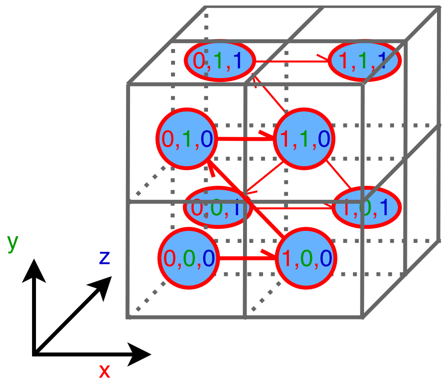

(c) Z-order curve. As in [26].

Figure 2.1: Example of space filling curves for a square.

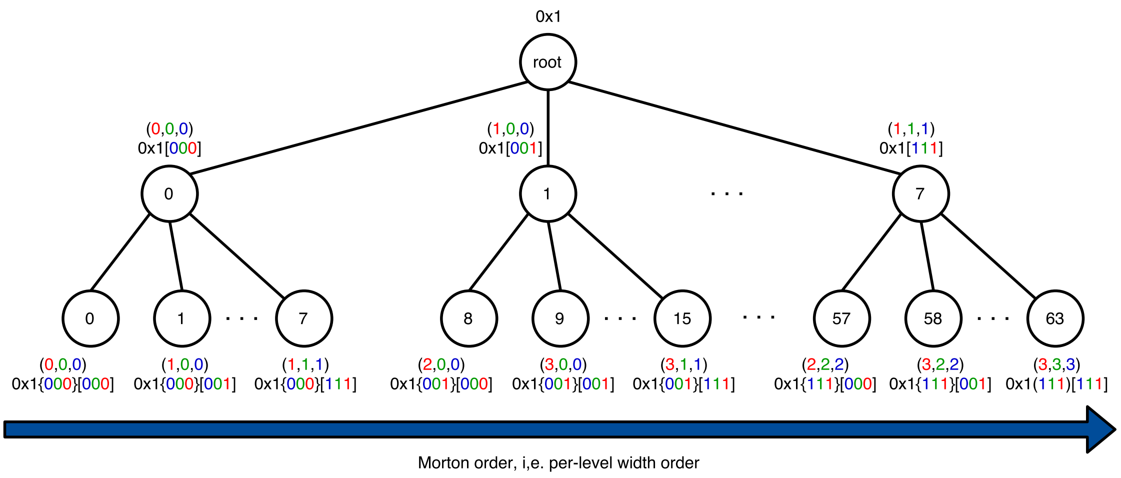
Morton [27] proposed a linearization of 2D grids, later generalized to n-dimensional grids. It results in a z-shaped space-filling curve, called the Z-order curve. The order in which the grid cells are visited by following this curve is called Morton order or Z-order. The associated Morton code for each cell can be computed directly from the grid coordinates by interleaving their bits. Figure 2.2 illustrates the above concepts.



(a) Z-order curve.



(b) Relationship between Morton order and grid coordinates.



(c) Morton order and associated hierarchical representation. Order is indicated inside nodes, coordinates and Morton codes outside them. The Morton code for the n-th child of a parent node with code x is x concatenated with the binary (bit-interleaving) representation of n. Coordinate values and interleaved bits share color. Parent code is between curly brackets and node index between square brackets. A prefix bit is used to avoid ambiguity.

Figure 2.2: Z-Order and Morton code illustrative example.

2.1.3 Hierarchical Spatial Data Structures

Hierarchical spatial data structures are used for structured organization of space based on recursive decomposition. They are very important representation techniques used in Computer Graphics, Image Processing and Robotics.

An hierarchical representation of data is useful because it enables focus on subsets of it. This property results in an efficient representation, improving execution times. Much of this efficiency comes from their ability to eliminate large portions of the space from computation, a process known as culling. Indeed, a test in the shallower parts of the hierarchy can save an exponential number of tests in deeper parts of the hierarchy because of the hierarchical nature of the datastructure. Another use of hierarchies is to fastly and adaptively query coarse or fine-grained instances of the data, allowing minimization of memory footprints. This is known as level of detail (LOD) and is extensively used in realtime applications.

Hierarhical spatial data structures are guided by a regular or irregular decomposition principle. Regular hierarchical structures subdivide the space in equal parts on each level, while irregular structures subdivide based on input. The resolution of the structure is the number of times it is subdivided and can be fixed beforehand, or it may be dependent of properties of the data. Examples of hierarchical spatial structures include Quadtrees, Octrees, Kd-trees, Range trees and Binary Space Partitions, to name a few. An overview of several of these kinds is given in the next sections. A more extensive discussion about Hierarchical Spatial Data Structures and their applications can be found in [28] and [29].

Quadtrees and Octrees

Quadtrees [30] recursively subdivide 2-dimensional space in four square or rectangular quadrants. The data structure consists of a tree with nodes with exact four children and leaves. There are several types of quadtrees based on what kind of data they represent. Examples include the Region Quadtree, Point Quadtree, Point-region Quadtree, Edge Quadtree and Polygonal map Quadtree. Figure 2.3 shows an example of a Quadtree and an extensive discussion about them can be found in [31].

Octrees [32] are the three-dimension analog of Quadtrees. They recursively subdivide the space in eight octants, resulting in a tree with eight children and leaves. Figure 2.4 shows an Octree.

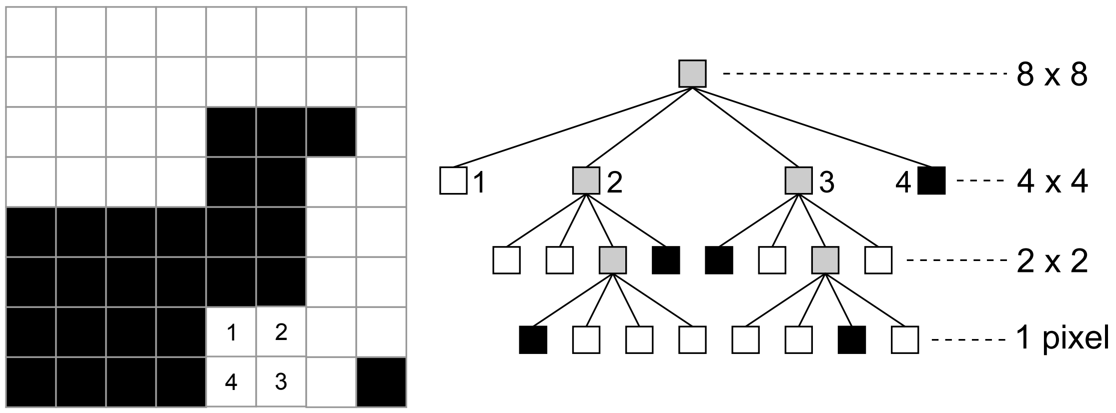


Figure 2.3: 8x8 Bitmap represented by a compressed quadtree. Numbers denote order of nodes. As in [1].

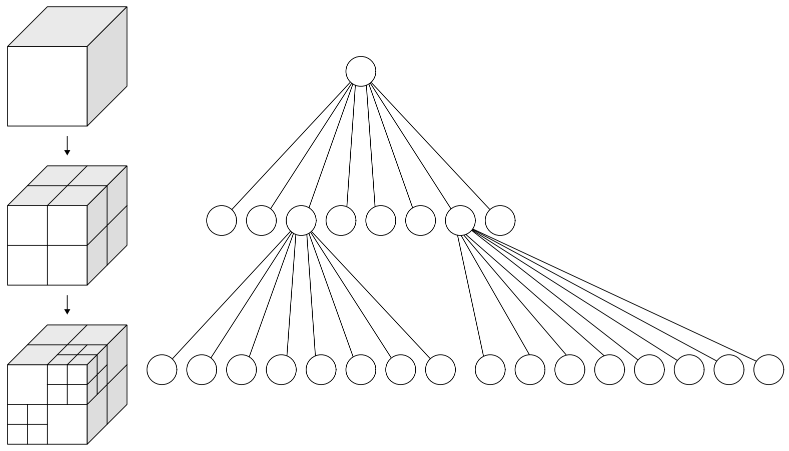


Figure 2.4: A recursive subdivision of a cube into octants (left) and the corresponding octree (right). As in [2].

Kd-trees

A Kd-tree (short for k-dimensional tree) [33] is a structure for recursive subdivision of a k-dimensional space. It consists of a binary tree, with each non-leaf node corresponding to a subdivision in a dimension. This subdivision can be tought as an hyperplane cutting that dimension in two half-spaces. The resulting two half-spaces are represented by the two children of that node. Leaves contain the actual data, grouped by regions resulted from all subdivisions done in the traversal from root to them. Figure 2.5 contains an example of a Kd-tree for a point set. Kd-tree applications include range searches and nearest neighbor searches.

The construction of a Kd-tree revolves around choosing the hyperplanes. There are many possible ways to do such task. Given the flexibilty of the data structure, the construction can be customized to fit better a specific problem at hand. However, the canonical method is to sort the points in an axis of a dimension and select the hyperplane to be the median of the points in that axis. This procedure is then applyed recursively for the points in each half-space of the generated hyperplane, cycling the dimension axis. The recursion ends when a threshold of number of points per node or recursion depth is achieved. This algorithm results in a balanced Kd-tree, in which all leaves have approximately the same depth.

Operations as point insertion and removal are also defined. The insertion can be done by traversing the Kd-tree, testing in which half-planes the point lays for each hyperplane encountered. However, this procedure can increase the depth of the tree, resulting in imbalance and worse performance. If the tree becomes too unbalanced, it must be necessary to reconstruct it. It is necessary because the traditional tree rotation technique used to balance binary trees does not work in this case, since Kd-trees are multidimensional. Removal can be done by traversing the tree in search of

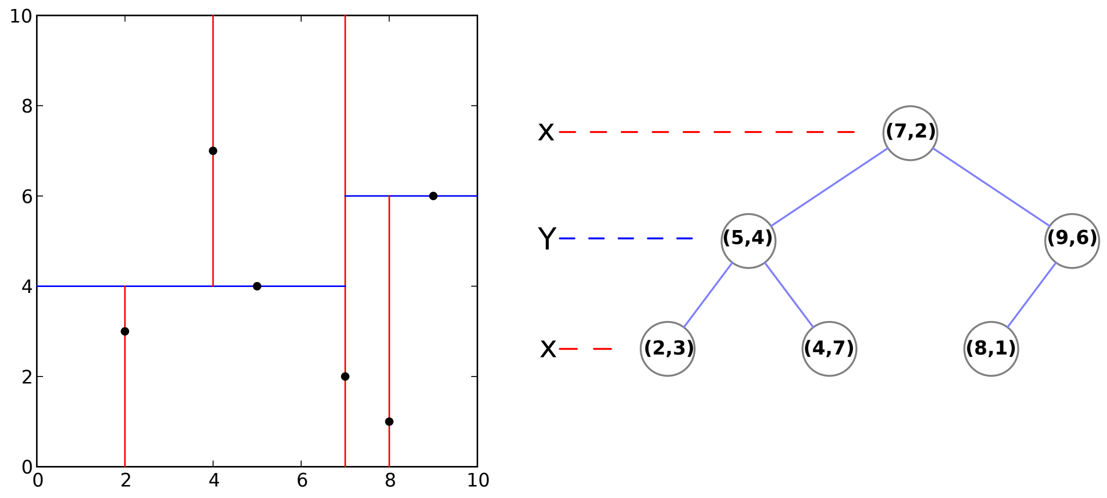


Figure 2.5: k-d tree decomposition for the point set {(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)} (as in [3]) and the resulting kd-tree (as in [4]).

the node where the point belongs and recreating all nodes that descend this one.

The same hierarchical subdivision achieved by Quadtrees and Octrees can be done using Kd-trees. For that it suffices to choose the hyperplane always in the half of the current half-space.

Binary Space Partitions (BSPs)

Binary Space Partitioning [34, 35] is the generalization of the binary subdivision of space. As such, it generalizes Quadtrees, Octrees, Kd-trees and any other Hierarchical Spatial Data Structure that can be reduced to a binary subdivision procedure. BSPs do not restrict the orientation of the subdivision hyperplanes in any way. The hyperplanes are totally dependent on the purpose of the subdivision. Applications of BSPs include tests for fast mesh culling or sorting in realtime applications. For example, they are vastly used in games to define interiors, restrict gameplay area or to sort objects back to front in respect with camera.

An example of BSP for ordering line segments for correct rendering given a viewing position V is given in Figure 2.6. The technique used for such task is known as the painter's algorithm and consists of rendering the segments starting from the background and proceeding until the foreground is reached. A BSP tree is constructed using the orientation of each segment as a partition plane. This task also subdivides the segments into smaller ones for which the visibility query can be answered without ambiguity. The rendering is done by a recursive traversal of the BSP tree, drawing first the segments that are on the oposite side of the one where V is in, then the partitioning segment and finally the ones on the same side as V.

The drawback of BSPs is that its construction is costly in comparison with other

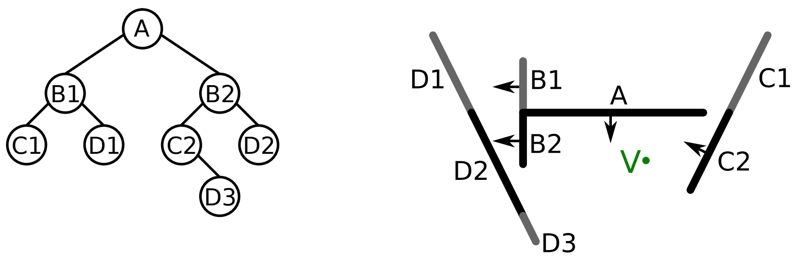


Figure 2.6: A BSP tree for correct drawing of line segments using the painter's algorithm. As in [5].

less-generic approaches. Usually BSPs are calculated in preprocessing steps in order to increase performance.

A final remark about Hierarchical Spatial Data Structures is that Morton codes extend naturally to regular spatial subdivision schemes, thus they are usually used in conjunction with structures such as Octrees and regular Kd-trees. Figure 2.2 illustrates an Octree with an embedded Morton code curve, and its associated hierarchical representation.

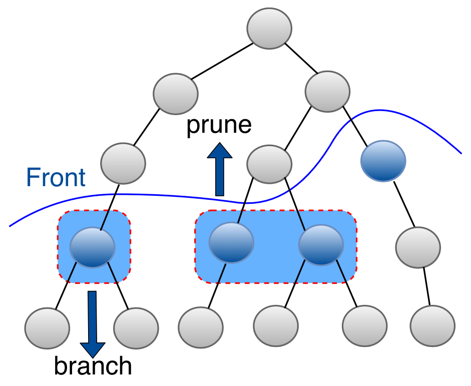
2.1.4 Rendering Front

Front tracking [36] is a technique used to optimize sequential traversals of hierarchies by exploration of spatial and temporal coherence. This approach has been successfully applied in many works [37–39]. It consists of starting the traversal at the nodes where it stopped in the preceding frame instead of starting at the root node for every new frame. The nodes from which the traversal starts compose the front. A Rendering Front is a front used in a hierarchical structure for rendering purposes.

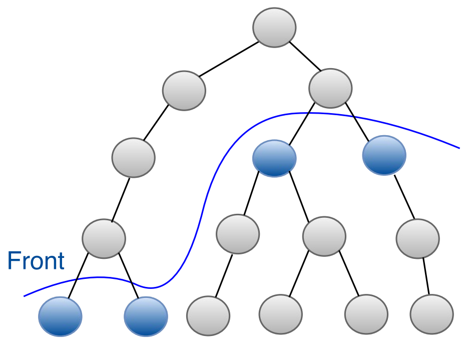
Fronts have two basic operators: *prune* and *branch*. The *prune* operator traverses the hierarchy up, removing a group of sibling nodes from the front and inserting their parent. The *branch* operator works in the opposite direction, by removing a node from the front and inserting its children. Figure 2.7 depicts a front and the two operators.

2.2 Large point cloud rendering

QSplat [19] is the seminal reference work on large point cloud rendering. It is based on an out-of-core hierarchy of bounding spheres, which is traversed to render the points. Since its main limitation was the extensive CPU usage, QSplat was followed by works focused on loading more work onto the GPU. For example, Sequential Point Trees [40] introduced adaptive rendering completely on the graphics card by



(a) A front and operations to be performed.



(b) The front after the *prune* and *branch* operations.

Figure 2.7: Rendering Front example.

defining an octree linearization that can be traversed efficiently using the GPU architecture. Other methods used approaches relying on the out-of-core paradigm, such as XSplat [41] and Instant Points [42]. XSplat proposed a paginated multiresolution point-octree hierarchy with virtual memory mapping, while Instant Points extended Sequential Point Trees by nesting linearized octrees to define an out-of-core system. Layered Point Clouds [43] proposed a binary tree of precomputed object-space point cloud blocks that is traversed to adapt sample densities according to the projected size in the image. Wand et al. [44] presented an out-of-core octree-based renderer capable of editing large point clouds and Bettio et al. [45] implemented a kd-tree-based system for network distribution, exploration and linkage of multimedia layers in large point clouds. Other works focused on parallelism using multiple machines to speed-up large model processing or to render on wall displays using triangles, points, or both [46–50].

More recently, relatively few works have focused on further improving the rendering of large point clouds, such as the method by Lukac et al. [51]. Instead, more effort has been concentrated on using established techniques in domains that require the visualization of large datasets as a tool for other purposes. For example, city visualization using aerial LIDAR [52, 53], sonar data visualization [54] and, more prominently, virtual reality [55–58].

An important discussion concerns which approach best exploits parallelism when creating a hierarchy. A good way to address this question is to study GPU algorithms, which must rely on smart problem modeling to achieve maximum degree of data independency, increasing throughput in a GPU manycore environment. Karras [59] made an in-depth discussion about this subject. His major criticism of other methods is that top-down approaches achieve a low degree of parallelism at the top levels of the tree, generating underutilization of processing resources at early stages of hierarchy construction. Bottom-up methods do not suffer from this problem because the number of nodes grows exponentially with the hierarchy depth, providing sufficient data independency and a good degree of parallelism.

While the aforementioned papers present very useful and clever methods to implement or use large point cloud rendering, none of them considers presenting data to the user before the full hierarchy is created. For example, implementors of systems that use large point cloud rendering as a tool could use the visual feedback given by the algorithm in order to check if the data is presented properly, without having to wait for the full hierarchy to be available. Additionally, in environments where data transfer is a bottleneck, the input data could be transfered and the hierarchy constructed on-the-fly, instead of transferring the full hierarchy which is several times larger. Another important point is that all algorithms for large point cloud rendering start by presenting data top-down, at the coarser levels of the hierarchy. While this initial presentation is useful for providing an overview of the model, there are use cases where early full detail inspection of parts of the data would be preferred. In the usual top-down rendering approach, full detail is available only much later in the process, after the hierarchy is fairly well populated. For example, if the goal is to evaluate the quality of the model, rendering lower resolution versions early may not be useful at all. Since bottom-up approaches provide good parallelism degrees early and the GPU's memory is initially empty, why not render the model in full detail from the beginning, working upwards in the hierarchy as coarser nodes are available and needed? OMiCroN was designed with these observations in mind. The result is a very flexible algorithm, focused on presenting full-detail data to the user as soon as possible.

Chapter 3

OMiCroN - Oblique Multipass Hierarchy Creation while Navigating

The main idea behind OMiCroN is simple: maintain a flexible data-structure to represent incomplete hierarchies that works in a streaming fashion. For this purpose, we define two main operators: *concatenate* and *fix* (Figure 3.1). Starting from an initial (possibly empty) renderable hierarchy, nodes from the maximum level are inserted using the *concatenate* operator. Then, the hierarchy is evaluated in a bottom-up manner, inserting ancestors of the concatenated nodes into the renderable hierarchy using the *fix* operator. These two operators ensure that at least one part of the hierarchy is ready for rendering while the rest is being constructed in parallel. With this simple stream-based approach, not only latency is hidden, but it allows obtaining the best possible realtime feedback while the data-structure is being constructed, i.e., the actual rendering. Next, we formally detail the operators and how OMiCroN works.

3.1 Oblique Hierarchy Cuts

OMiCroN is based on the novel concept of Oblique Hierarchy Cut which we introduce here. Given a conceptual expected hierarchy H, with depth l_{max}, an Oblique Hierarchy Cut C consists of a delimiting Morton code m_C and a set of lists $L_C = \{L_{C,k}, L_{C,k+1}, \cdots, L_{C,l_{max}}\}$, where k is the shallowest level of the hierarchy present in the cut. Each node N is uniquely identified by its Morton code m_N and these two concepts are interchangeable from now on. Let also $span(x)$ be a function that returns the maximum Morton code at level l_{max} of a supposedly full subtree rooted by Morton code x (see Figure 3.2). C also has the following important invariants (see Figure 3.3):

1.1 m_C has level l_{max}.

1.2 $L_{C,l}$ contains subtrees of L_C rooted by nodes at level l.

1.3 All subtrees in L_C are disjoint.

1.4 $L_{C,l}$ is always sorted in Morton order.

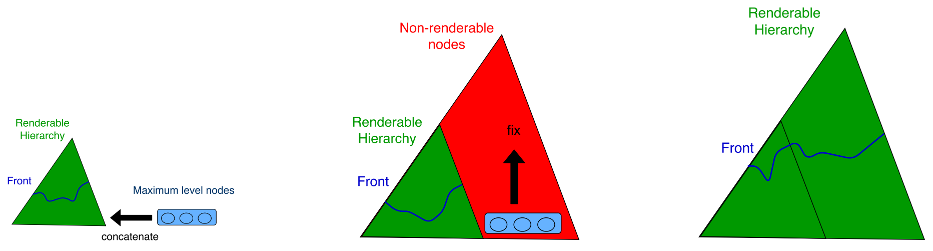
1.5 All nodes N with $span(m_N) \leq m_C$ are in one of the subtrees in L_C.

We now formally define the two operators, *concatenate* and *fix*, as well as the important concept of *Placeholder* nodes.

3.1.1 Operator *Concatenate*

The operator *concatenate* is defined as $C' = concatenate(C, \{x_0, \cdots, x_n\})$ with $m_C < x_0 < ... < x_n$. This operator incorporates new l_{max} level leaf nodes $\{x_0, ..., x_n\}$ to C, resulting in a new cut C'. The operator itself is simple and consists of concatenating all new nodes in a list $L_{C,l_{max}}$, resulting in $L_{C',l_{max}}$. This operator is illustrated in Figure 3.3.

In order for C' to be an Oblique Hierarchy Cut, all invariants must hold. Invariant 1.1 can be maintained by letting $m_{C'} = x_n$. Invariant 1.2 holds by the definition of *concatenate*, since the insertion of the leaf nodes occurs at the correct list $L_{C,l_{max}}$ at level l_{max}. Invariants 1.3 and 1.4 are ensured by the fact that $m_C < x_0 < ... < x_n$, also established in the definition of *concatenate*. Invariant 1.5, however, does not hold, since some of the ancestors A_x of the new nodes $\{x_0, ..., x_n\}$ may have $m_C < span(A_x) \leq m_{C'}$, but are not in any subtree of $L_{C'}$ after concatenation. In fact, it would be absurd if they were, since all nodes N_C in C have



(a) Initial (possibly empty) renderable hierarchy and *concatenate* operator.

(b) The *fix* operator: node ancestors are inserted into the hierarchy.

(c) After the *fix* operation the renderable hierarchy is expanded.

Figure 3.1: OMiCroN overview. A renderable hierarchy is maintained while inserting incoming nodes in parallel. This cycle is repeated until the whole hierarchy is constructed.

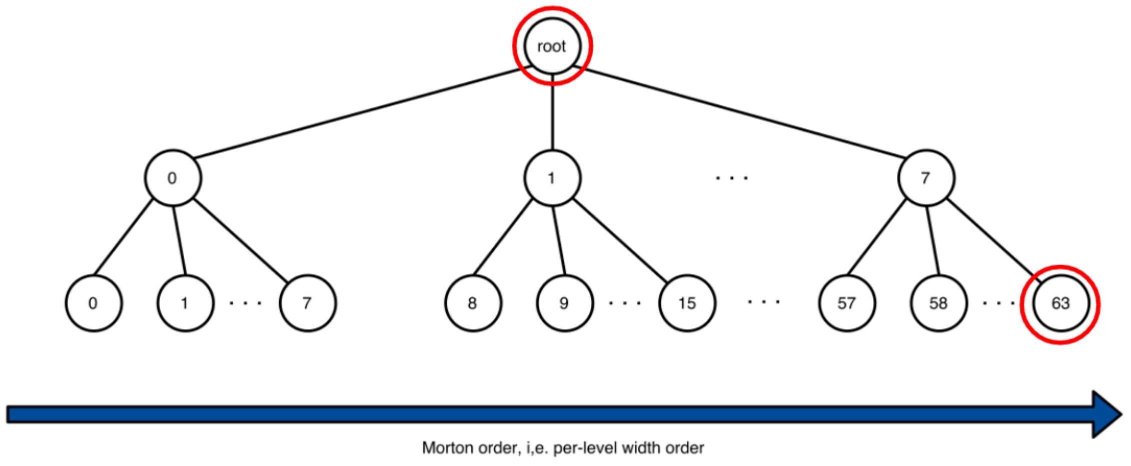


Figure 3.2: The function $span(x)$. It returns the morton code of the rightmost descendant of x in a supposedly full subtree. In other words, it returns the maximum possible morton code of a subtree rooted by x at the morton curve of level l_{max}. In this example, $span(root) = 63$, as the nodes marked in red show.

$span(N_C) \leq m_C$ (invariant 1.5), $m_C < m_{C'}$, and the *concatenate* operator only inserts nodes greater than m_C at level l_{max}.

3.1.2 Operator *Fix*

To resolve invariant 1.5, we define the $C'' = fix(C')$ operator, whose purpose is to insert the offending nodes in subtrees of $L_{C'}$, resulting in $L_{C''}$, while maintaining all other invariants intact. To achieve this, *fix* first defines the set of offending nodes A^*_x as a subset of A_x with $span(A^*_x) \leq m_{C'}$. Second, it identifies all subtree roots in A^*_x whose parents are not in A^*_x. Let S be the set of such parent nodes. To identify these subtrees, the lists are processed in reverse order, that is, beginning with $L_{C',l_{max}}$. For each list, its root nodes are visited in Morton order. The evaluation of a list $L_{C',l}$ works in the following manner: identify the sibling root nodes in $L_{C',l}$; check if their parent is in A^*_x; create a new subtree rooted by their parents at level $l-1$; and move the subtrees from $L_{C',l}$ to their respective parent subtrees in $L_{C',l-1}$. Note, however, that if the parent is in S neither the new subtree is created nor its children subtrees are moved. The resulting $L_{C''}$ will have, thus, only subtrees rooted at nodes whose parents are in S.

In order to guarantee that *fix* is robust enough, all invariants must be checked for correctness after the operation. Since no new l_{max} level nodes are inserted by *fix*, we let $m_{C''} = m_{C'}$ and invariant 1.1 is ensured. Invariant 1.2 holds because the A^*_x nodes are inserted in $L_{C'}$ at the same level they are in H. Regarding invariant 1.3, the nodes in A^*_x are unique and they were not in C', since the only nodes $N_{C'}$ that had $span(N_{C'}) > m_C$ were inserted at level l_{max} by the *concatenate* operator. Thus,

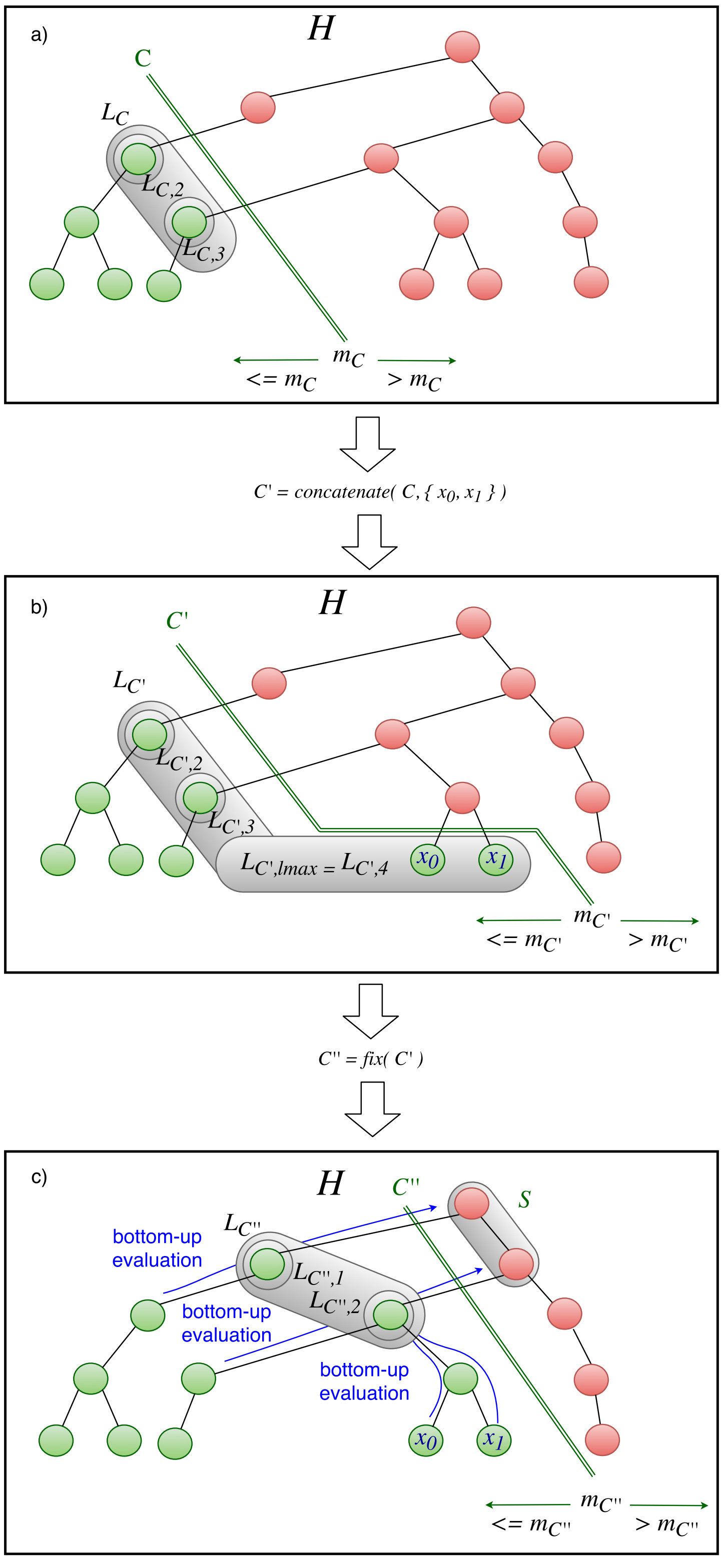


Figure 3.3: An example of Oblique Hierarchy Cut and operators *concatenate* and *fix*. A cut C is defined by a delimiting Morton code m_C and a list of roots per level L_C (a). The *concatenate* operator inserts new roots x_0 and x_1 at the deepest level l_{max}, resulting in cut C' (b). Invariant 1.5 does not hold for C' and operator *fix* is used to traverse subtrees bottom-up, inserting nodes until the boundary S is reached. It results in cut C'' for which all invariants hold.

this invariant holds. Since the subtrees inserted by *fix* are evaluated in Morton order, they are also inserted in this order, maintaining invariant 1.4. Lastly, invariant 1.5 is ensured because the subtrees inserted by *fix* are rooted by nodes whose parents are in S, and S is outside of A^*_x, $m_{C''} < span(S)$, by the definition of A^*_x. Thus S forms a node boundary outside cut C''. The fix operator is illustrated in Figure 3.3.

3.1.3 Placeholders

According to the aforementioned definition of Oblique Hierarchy Cut, H can only have leaves at level l_{max}, since the *concatenate* operator only inserts nodes at this level. Leaves could be inserted into other levels directly, but it would make it difficult for *fix* to efficiently maintain invariant 1.4 since the lists $L_{C'}$ are independent and evaluated in a bottom-up manner. To address this issue, the concept of *placeholder* is defined. A *placeholder* is an empty node at a given level representing a node at a shallower level. More precisely, given a node N at level l, its placeholder $P_{N,l+i}$ at level $l + i$ is defined as the rightmost possible descendant of N at level $l + i$ in a supposedly full tree. In other words, the Morton code of $P_{N,l+i}$ is m_N followed by a bitmask of three 1's for each one of the m additional levels. Note that, with this definition, $P_{N,l_{max}} = span(N)$.

The use of placeholders enables Oblique Hierarchy Cuts to be used with any octree. A leaf X in H with level $l < l_{max}$ is represented by placeholder $P_{X,i}$ such that $l < i \leq l_{max}$ when inserting the subtree of level i at $L_{C'_i}$. Placeholders are used as roots of degenerate subtrees, since there is no purpose for them inside subtrees. Even if not meaningful for H, placeholders ensure invariant 1.4 in *fix* until level l is reached. Figure 3.4 shows the concept of placeholders.

3.1.4 Sequence of Oblique Hierarchy Cuts

Intuitively, a sequence of Oblique Hierarchy Cuts C_i resulting from sequentially applying operators *concatenate* and *fix* until no more leaf nodes or placeholders are left for insertion results in an oblique sweep of H, as can be seen in Figure 3.4. To prove this, let C_{end} be the last cut in this sequence. Because of invariant 1.5, all nodes N in H with $m_N \leq m_{C_{end}}$ will be in subtrees in $L_{C_{end}}$ after fix. Since there are no more placeholders or leaf nodes in level l_{max}, there are no nodes N with $m_N > m_{C_{end}}$ and, thus, S is composed only by the *null* node (parent of H's root node). Since there are no other parents outside the subtrees that have roots with parents in S, and S has only a single element, $L_{C_{end}}$ is composed by a single subtree, named T. Also, T's root has parent equal to the *null* node. Thus, $T = H$, as intuitively suspected.

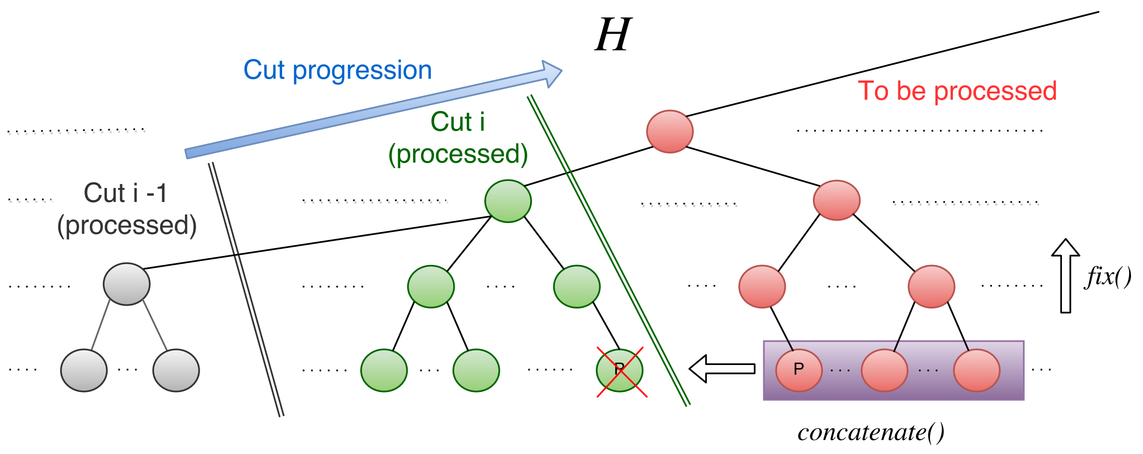


Figure 3.4: Oblique Hierarchy Cut progression. As operators *concatenate* and *fix* are used, the cuts sweep their associated hierarchy H. Placeholders are marked with a P and the ones used but removed while processing lists bottom-up are also marked with a red X.

3.2 Oblique Hierarchy Cut Front

Concomitantly with the building of H with progressive oblique cuts, a rendering process might be traversing the already processed portions of H with the help of a front (see Figure 3.1). Thus, for a given Oblique Hierarchy Cut C, the rendering process will adaptively maintain a front F_C restricted to the renderable part of H. In order to ensure proper independence of F_C with respect to C and other important properties needed later, we define two invariants:

2.1 If F_C is composed of n nodes, named $F_{C,i}$, with $1 \leq i \leq n$, then $span(F_{C,1}) < \cdots < span(F_{C,i}) < span(F_{C,i+1}) < \cdots < span(F_{C,n})$.

2.2 The roots of subtrees in L_C cannot enter the Front.

Invariant 2.1 ensures that sibling nodes will be adjacent in the front, which eliminates searches and simplifies the *prune* operation. Invariant 2.2 is defined because the roots of subtrees in L_C are being moved among lists by the *fix* operator in order to create subtrees at other levels and thus are not safe to enter the front. Note that both invariants impose restrictions on the *prune* operator in order to ensure that all nodes on the front are roots of disjoint subtrees and do not include nodes still being processed. Similarly, placeholders cannot be pruned either since their parents might not yet be defined.

In summary, the evaluation of an Oblique Hierarchy Cut Front consists of three steps:

1. Concatenate new placeholders into the front.

2. Choose the hierarchy level l where candidates for substituting placeholders in the front are to be sought.

3. Iterate over all front nodes, testing whether they are placeholders that can be substituted, and whether they need to be pruned, branched or rendered.

Placeholder and leaf insertions and substitutions will be further described in the next sections. The other aspects of operators *prune* and *branch* work as usual. All valid inner nodes are reachable by *prune* operations from the leaves, ensuring proper rendering capabilities for the cut. An example of a valid Oblique Hierarchy Cut Front is given in Figure 3.5.

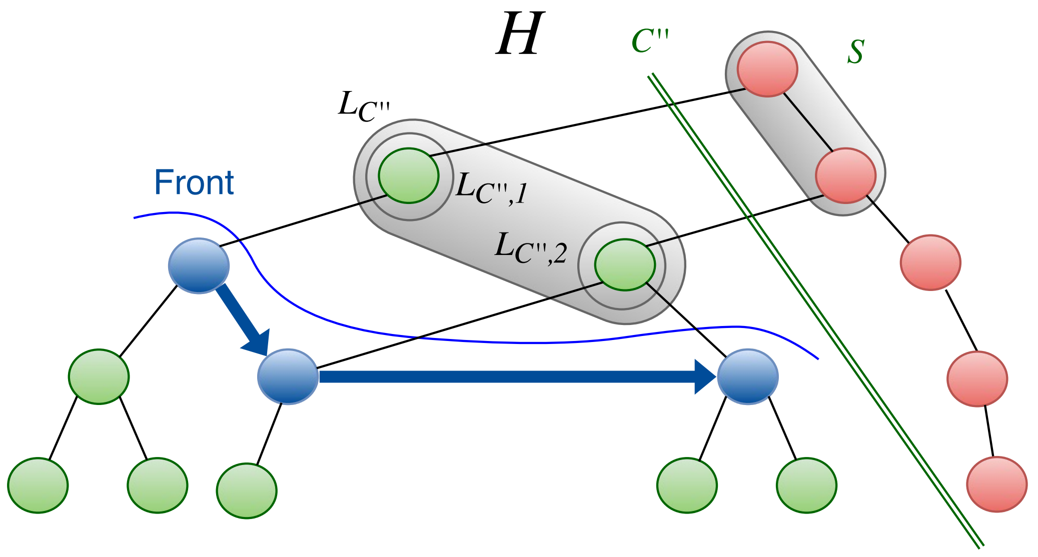


Figure 3.5: Example of valid Oblique Hierarchy Cut Front. The direction of the blue arrows indicate the order restriction imposed by invariant 2.1. The fact that all nodes in the front are not roots in L_C ensures invariant 2.2.

3.2.1 Insertion of new nodes

Since the root of H is only available after all sequential cuts are evaluated, the usual front initialization is not possible for F_C. To insert nodes in the Oblique Hierarchy Cut Front two operators are used: *insertPlaceholder* and *insertLeaf*. In order to simplify leaf and placeholder insertion and substitution, all leaves are first inserted in the front as placeholders and saved in a per-level list of leaves to be replaced. One main reason for this duplication is that new nodes are always inserted as roots in $L_{C,l_{max}}$, and cannot be in the front due of invariant 2.2. Thus, placeholders mark their position until the *fix* operator moves them to other subtrees. The front is, then, continuously checked to see if placeholders can be replaced by leaf nodes. This substitution is detailed in the next section.

The *insertPlaceholder* operator in its turn is simple since it can just concatenate placeholders at the end of the front. This maintains the invariants since placeholders are available at level l_{max} and they are processed in Morton order by *fix*.

3.2.2 Substitution of placeholders

Since the leaf lists are organized by level, and the placeholders and leaves are respectively inserted into the front and into the lists in Morton order, a very simple and efficient substitution scheme is proposed. Given a placeholder and a substitution level l, it consists in verifying if the first element in the leaf list of level l is an ancestor of the placeholder. If it is, the leaf is removed from the substitution list and replaces the placeholder in the front. Since comparison of Morton codes is a fast $O(1)$ operation, the entire placeholder substitution algorithm is also $O(1)$.

Keeping in mind that for each front evaluation a single level l will be checked for substitution, all leaves at level l are guaranteed to be substituted in a single front evaluation. To verify this, note that if P_i and P_{i+1} are sequential placeholders at the same level and L_j and L_k are their leaf substitutes, then $k = j+1$. This comes again from the fact that all insertion lists and front nodes at a given level are in Morton order and that a leaf and its placeholder have a one-to-one relationship. Thus, if P_i is substituted and, as a consequence L_j is removed from the substitution list, then the new first leaf in that list will be L_k, resulting in P_{i+1} being the next placeholder to be successfully substituted at that level. Consequently, for each placeholder in the front we need only to verify the first leaf of the list, and after one evaluation the list for level l will be emptied.

3.2.3 Choice of substitution level

In order to maximize node substitution, l is chosen as the level with most insertions. This is an obvious choice, since the list will be completely emptied after the evaluation, so we are substituting the maximum number of placeholders in one iteration. The nodes not substituted in the current front evaluation are ignored since their corresponding leaves are not in level l. However, the corresponding leaf is already in another list, and it is guaranteed that its substitution will occur within the next $l_{max} - 1$ front evaluations.

3.3 Sample OMiCroN implementation

We have developed a multi-threaded implementation of the OMiCroN algorithm in C++ where the GPU is used only for splat rendering. The implementation follows the algorithms outlined in the previous sections, but a few adaptations are necessary with regard to concurrency control. Figure 3.6 shows a schematic view of the prototype and Algorithms 1 through 8 show in detail the passes necessary to maintain all parallel tasks correctly. The next paragraphs also describe this implementation and algorithms.

The system is composed of several threads, as Figure 3.6 shows. Three of them are persistent: the sorter, the fix's master and the front tracker. The others are short-lived ones: the fix's slaves and the GPU loaders. The synchronization of these threads is done using a per-level (i.e. per-morton-curve) mutual exclusion scheme. Detailed description of them and other important related tasks is in the following.

Sorter is responsible of reading the unordered point cloud and sorting it in chunks. The results of this task are worklists of nodes at level l_{max}, which are concatenated into the cut by pushing them into the list of level l_{max}. Algorithm 1 shows the process in detail while Algorithm 2 shows the concatenate operator in detail. It is important to note the point of mutual exclusion in concatenate, used to ensure that the fix's master thread does not access level l_max simultaneously.

Fix's master applies fix in iterations, one level l at a time, starting at level l_{max}. To perform such task, it pops worklists from level l, feeding itself and a number of issued slave threads with work. Each thread process one worklist of near constant size. The worklist processing consists of finding sibling groups of nodes, creating a parent for them and moving that parent to the list in level $l-1$. Since everything is done in morton order by definition, sibling nodes are always adjacent in the list. This process is done totally in parallel by the use of a per-thread segmented buffer. After all threads finish, the master thread verifies the segmented buffer for duplicates in the segments' boundaries. The duplicates occur when nodes of a sibling group are in different worklists. The master thread also merges segments to maintain near constant worklist size. Finally, a choice between processing the next level $l - 1$ or starting another fix pass from level l_{max} is performed. The alternative with more worklists is chosen so the parallelism degree is increased for the next iteration. Algorithm 3 shows the process in detail.

Parent creation. While fix is creating new parent nodes it also generates placeholders and substitution leaves which must be sent to the front. On one hand, placeholders are always generated for any node processed at level l_{max} so a place for ancestors of that node or for the node itself is guaranteed in the front. On the other hand, leaves are only sent to the front when their grandparent is created. This is done to ensure invariant 2.2. The same reason is behind the postponing of child-to-parent pointer creation in all nodes until their grandparent is created. In order to maintain full parallelism for fix, placeholders and leaves are sent to per-thread segmented buffers before achieving their final destination at front. A final point about this stage is that, in order to maintain the memory within a given budget, it is also possible to enable a very simple optimization, called *Leaf Collapse*. This optimization removes all leaves at level l_{max} which form a chain structure with their parents, i.e., leaves that do not have siblings. The process of creating a parent is detailed in Algorithm 4.

Front tracker evaluates the front, rendering, prunning or branching nodes. It also manages placeholders and leaves inserted by fix. First, it pushes all placeholders to the end of the front, since they necessarily have span() greater than all other nodes in the front. Second, it chooses the level with most leaves inserted as the substitution level *Sl*. Then, all nodes in the front are evaluated in sequence. If a node is a placeholder, a substitution attempt is made in the list of leaves at level *Sl*. This substitution is really fast and consists of checking if the first leaf at the list is an ancestor of the placeholder. This approach is possible because the front is sorted by span() (i.e. in morton order by level), the lists are also sorted in morton order and an ancestry query can be done quickly with morton codes. The substitution is detailed in Algorithm 6. If the placeholder is substituted, then the evaluation proceeds to verify if the node should be prunned, branched or just rendered, as usual for front-based traversals. The heuristic for prunning and branching is the projection of the node's box on the image. In order to perform prunning, a substitution attempt is also done for the next placeholders until a placeholder cannot be substituted or the substituted leaf has a different parent. This is necessary to guarantee that all sibling nodes will be prunned at the same time. Prune is also responsible of releasing nodes from GPU if it has reached its memory quota. Finally, both prune and branch issue the short-lived threads for node loading in GPU if necessary. Algorithms 7 and 8 show prune and branch in detail, respectivelly, while Algorithm 5 contains a detailed description of the front tracking thread.

A simple rendering approach based on *splats* [19] is used in our experiments. OMiCroN nodes contain point splats defined by a center point and two tangent vectors u and v. Parent node creation follows a policy that tries to maintain the ratio between the number of points in a parent and its children, where a parent contains a subset of the splats in its children with scaled tangent vectors. The splats in the rendering queue are used as input for the traditional two-pass EWA filter described in [22]. Several methods for computing the sizes of the projected splats were tested [22, 60–62]. The splat bounding box computation algorithm described in [62] resulted in the best performance-quality relationship and all results reported in this work applied it.

Algorithm 1: Sorter thread

```
Result: Sorts $P$ into morton-ordered chunks of leaves, creates worklists and
        concatenate them into the Oblique Cut.
$P \leftarrow$ read_points();
while $P$ is not empty do
    $Ch \leftarrow$ partial_sort( $P$ );            /* Chunk $Ch$ is removed from $P$ */
    while $Ch$ is not empty do
        $W \leftarrow$ create_worklist( $Ch$ );          /* $W$ is removed from $Ch$ */
        concatenate( $W$ )
    end
end
```

Algorithm 2: concatenate() operator

```
Data: Worklist $W$
Result: Concatenates $W$ into the max depth list of the Oblique Cut.
        Ensures mutual exclusion from Fix thread.
begin mutex $l_{max}$
    $L_{C,l_{max}}$.push_worklist( $W$ );
end
```

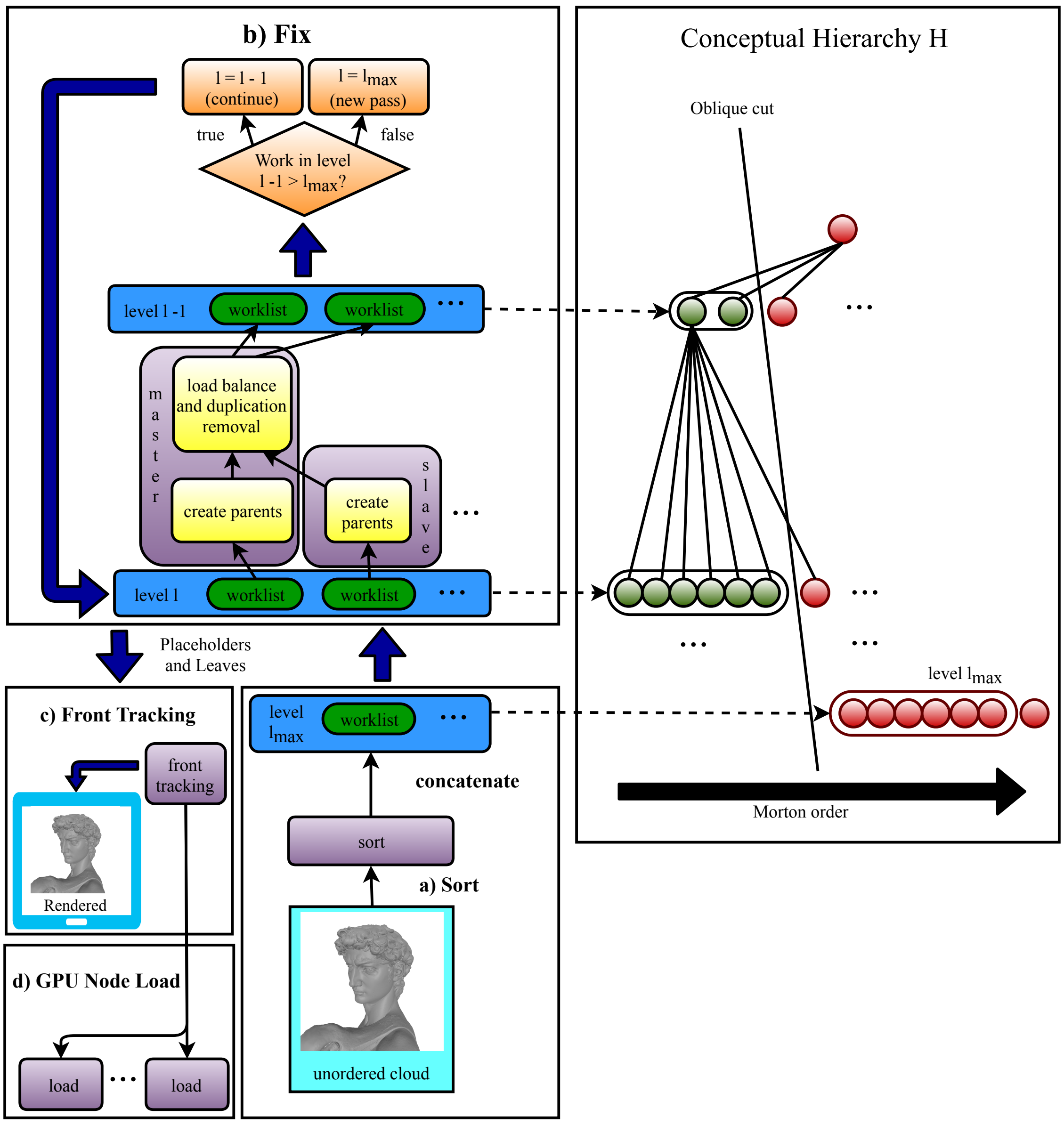


Figure 3.6: OMiCroN multithreaded prototype, composed of four parallel Tasks: a) Sort, b) Fix, c) Front tracking, d) GPU Load. All threads are represented as purple boxes. a) is responsible of sorting data chunks, creating nodes at level l_{max} and applying concatenate on them (Algorithm 1). b) applies fix, also feeding the front with new placeholders and leaves. This task is parallelized per level. Created placeholders and leaves are first pushed to a per-thread segmented buffer so fix can be fully parallel. After all threads finish, the segmented buffers are compacted and sent to the front (Algorithm 3). c) evaluates the front, rendering, prunning and branching nodes. Before evaluation, it concatenates at the front's end the placeholders in the buffer and chooses a substitution level. A placeholder is substituted at front evaluation if its ancestor leaf is found in the list of the chosen substitution level (Algorithm 5). d) consists of short-lived theads responsible of loading nodes into the GPU.

Algorithm 3: Fix master thread

```
Data: Oblique Cut lists $L_{C,l}$, Front $F$
Result: Applies fix by evaluating lists bottom-up, creating parent nodes and
        placeholders.
$l \leftarrow l_{max}$;
while Sorter thread still reading OR sorting do
    while $l > 0$ do
        while $L_{C,l}$ has work do
            // pop_worklists must be exclusive if $l = l_{max}$
            if $l = l_{max}$ then
                begin mutex $l_{max}$
                    $W \leftarrow$ pop_worklists( $L_{C,l}$ );
                end
            else
                $W \leftarrow$ pop_worklists( $L_{C,l}$ );
            end
            foreach $W_i$ in $W$, in parallel do
                foreach sibling group $Sg_j$ in $W_i$ do
                    $P_j \leftarrow$ create_parent($Sg_j$, $i$);
                    $T_i$.push($P_j$);                  /* Per-thread buffer $T_i$ */
                end
            end
            // Check per-thread buffer boundaries for duplicates.
            // Merge them to have constant worklist size.
            remove_duplicates_and_merge( $T$ );
            $L_{C,l-1}$.push_worklists( $T$ );
            // Created placeholders and leaves in create_parent()
               must be sent to the front.  Mutual exclusion from
               front thread is needed.
            begin mutex $l_{max}$
                send_placeholders();
            end
            begin mutex leaf level
                send_leaves();
            end
        end
        if $L_{C,l-1}$ has more work than $L_{C,l_{max}}$ then
            $l \leftarrow l - 1$;
        else
            $l \leftarrow l_{max}$;
        end
    end
end
```

Algorithm 4: create_parent()

```
Data: Sibling group $Sg$, Thread id $t$
Result: Creates a parent node for $Sg$. The pointers to parent nodes are only
        set at grandchild level to ensure invariant 2.2. Accumulates new
        placeholders and granchild leaves in per-thread buffers.
$Pa \leftarrow$ get_points( $Sg$, parent point ratio );
if $leaf\ collapse\ on\ AND\ Sg.size = 1\ AND\ Sg\ at\ level\ l_{max}$ then
    delete( $Sg$ );
else
    $Pa$.set_pointer_to_children( $Sg$ );
end
foreach $Sg_i$ in $Pa.children$ do
    if $Sg_i$ at level $l_{max}$ then
        $P$ = create_placeholder( $Sg_i$ );
        send_to_placeholder_buffer( $P$, $t$ );
    else
        $Gs \leftarrow Sg_i$.get_children();
        foreach $Gs_j$ in $Gs$ do
            if $Gs_j$ is leaf then
                send_to_leaf_buffer($Gs_j$, $Gs_j$.level(), $t$);
            end
            create_pointer_to_parent( $Gs_j$ );
        end
    end
end
```

Algorithm 5: Front evaluation thread

```
Data: Front $F$
Result: Evaluates the front. Renders, prunes or branches nodes and
        substitutes placeholders.
start_frame();
begin mutex $l_{max}$
    // Inserts at the front ending any placeholder sent by Fix .
    insert_placeholders();
end
$Sl \leftarrow$ level with most leaves to substitute;
foreach node $N$ in $F$ do
    if $N$ is placeholder then
        try_substitution( $N$, $Sl$ );
    end
    if $N$ is not placeholder then
        $P \leftarrow N$.parent;
        if $P$ projection $<$ threshold OR $P$ is frustum cullable then
            foreach $Sb$ in $N$.siblings do
                if $Sb$ is placeholder then
                    // Ensures the entire sibling group is there.
                    try_substitution( $Sb$, $Sl$ );
                end
            end
            prune( $P$, $N$ );
        else
            if $N$ projection $<$ threshold AND $N$ is not frustum cullable then
                branch($N$);
            else
                if $N$ is not frustum cullable then
                    $N$.queue_rendering();
                end
            end
        end
    end
end
end_frame();
```

Algorithm 6: try_substitution()

```
Data: Placeholder $P$, substitution level $Sl$, Front $F$
Result: Substitutes a placeholder if an ancestor leaf node is found.
begin mutex $Sl$
    candidate $Ca \leftarrow F$.leaves[$Sl$][0];
    // Next comparison is fast using morton code
    if $Ca$ is ancestor of $P$ then
        $P \leftarrow Ca$;
    end
end
```

Algorithm 7: Prune

```
Data: Parent node $P$, child node $N$, Front $F$
Result: Prunes $N$ and its siblings from the front if conditions are met. The
        parent $P$ takes their place.
if $P$ is loaded on GPU then
    $F$.remove( $N$ and $N$.siblings );
    $F$.insert( $P$ );
    $P$.queue_rendering();
    if reached GPU mem quota then
        $N$.unload_from_GPU();
        $N$.siblings.unload_from_GPU();
    end
else
    // Issues a short-lived thread for data transference to GPU.
    $P$.load_in_GPU();
    $N$.queue_rendering();
end
```

Algorithm 8: Branch

```
Data: Node $N$, Front $F$
Result: Branches $N$ if conditions are met. Its place is taken by its children.
$Ch \leftarrow N$.children;
if $Ch$ are loaded in GPU then
    $F$.remove( $N$ );
    $F$.insert( $Ch$ );
    $Ch$.queue_rendering();
else
    // Issues a short-lived thread for data transference to GPU.
    $Ch$.load_in_GPU();
    $N$.queue_rendering();
end
```

Chapter 4

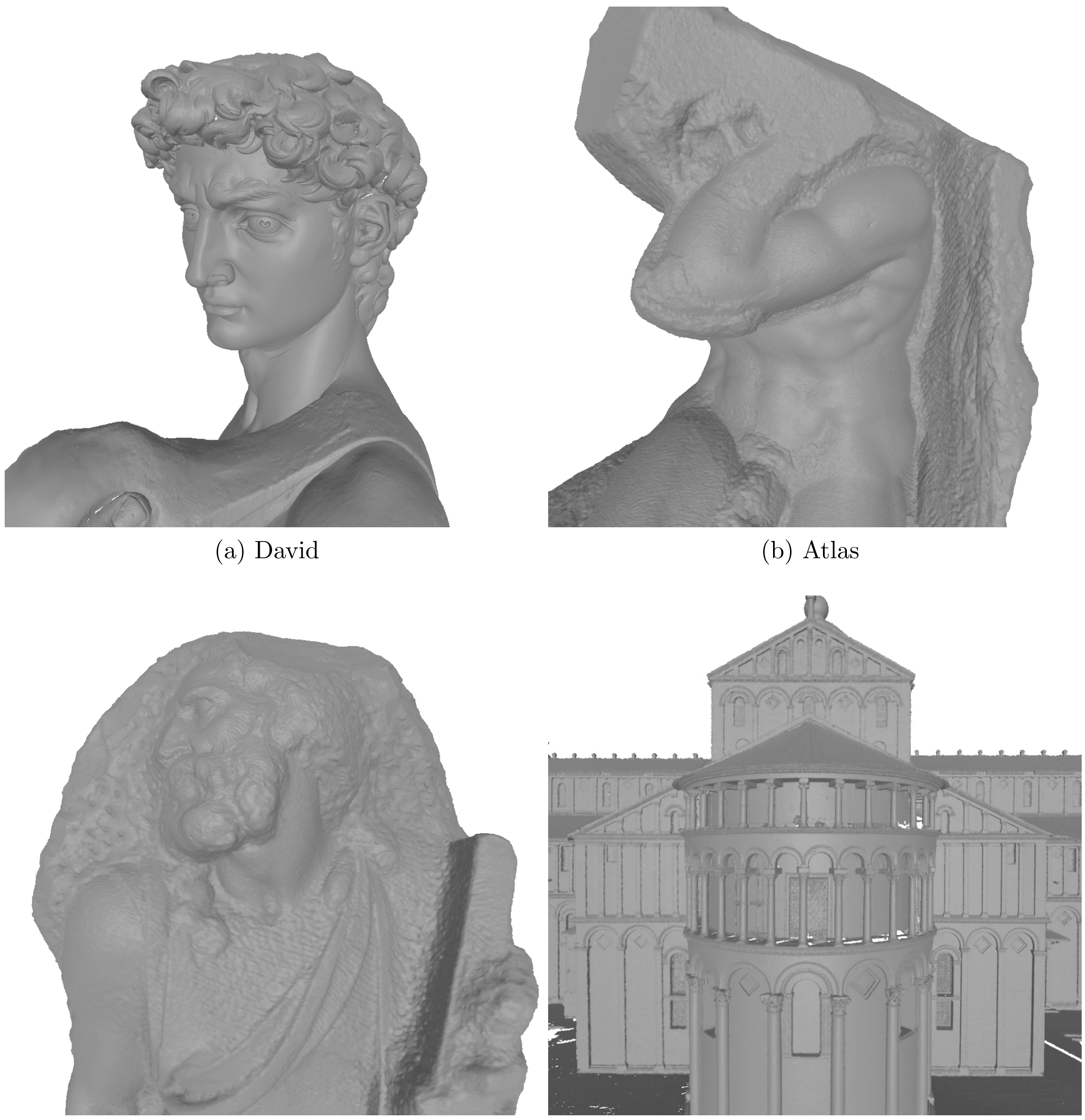
Experiments

The prototype implementation was tested using four point cloud datasets: David (469M points, 11.2GB), Atlas (255M points, 6.1GB), St. Matthew (187M points, 4.5GB) and Duomo (100M points, 2.4GB), all processed with hierarchies with 7 levels (Figure 4.1). Coordinates in all datasets were normalized to the range $[0, 1]$. David, Atlas and St. Matthew were obtained at the Digital Michelangelo Project page.

4.1 Rendering latency tests

To assess the actual delay from the moment the raw unsorted collection of points is available to the moment where rendering actually starts, we must consider the sorting process in some depth. The simplest scenario consists of a separate thread that reads the whole collection, sorts it and streams it to OMiCroN. In this case, OMiCroN must wait at least for the whole collection to be read by the sorting thread, and for the sort itself. In a more elaborate setup, the sorting process might start feeding OMiCroN as soon as a prefix of the sorted collection becomes available. In order to measure these gains, we conducted a set of experiments. Our testbed consists of a desktop computer with an Intel Core i7-3820 processor with 16GB memory, NVidia GeForce GTX 750 and a SanDisk 120GB SSD. The maximum hierarchy depth was set to 7. The same SSD is used for swap and I/O.

The first experiment consisted of consecutively sorting and streaming chunks of the input to OMiCroN. We use the parallel IntroSort available in the Standard Template Library (STL) of the C++ programming language (std::partial_sort() or std::sort()). Parallel rendering and leaf collapse are enabled for these tests. Since rendering starts as soon as the first sorted chunk becomes available, using more chunks allows rendering to start earlier, as shown in Figure 4.2. In particular, increasing the number of sorting chunks can improve the time between the moment input finishes and rendering starts from 5 to 31 times, depending on the size of


(a) David (b) Atlas

(c) St. Matthew (d) Duomo

Figure 4.1: The datasets used in experiments. Rendered using OMiCroN.

the dataset. The price of having this early rendering is that hierarchy creation time may increase up to 4 times for smaller datasets. Nonetheless, the partial sort plays an important role in reducing or eliminating the use of swap during sort and hierarchy creation, resulting in better timings in all aspects for large datasets such as David. This interesting result is demonstrated in Figure 4.2c. The conclusion is that OMiCroN's preprocessing is very flexible. The partial sort can be used to reduce the swap area usage or to reduce the delay for the first rendering. One must respect the inherent tradeoff of increasing the number of chunks to find the optimal number for a given dataset at hand. As a final note, OMiCroN consumes sorted chunks almost as fast as they are produced and streamed, and the hierarchy is finished at most $1s$

after the last byte of the sorted stream is read.

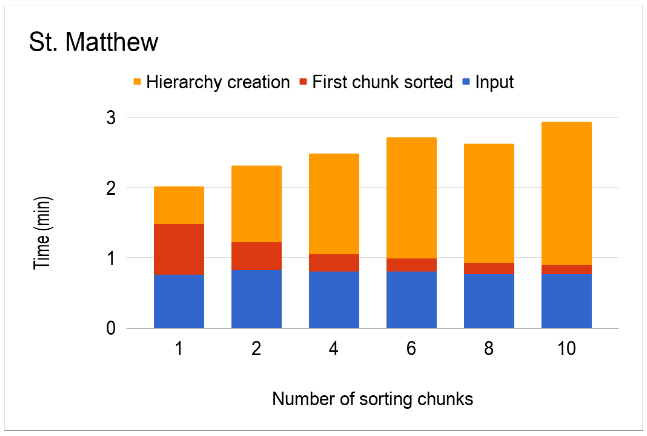
The second experiment consists of profiling and comparing OMiCroN with the parallel rendering activated and deactivated at hierarchy creation time, also evaluating the system core usage while running the algorithm. The purpose of this test is to measure the overhead of parallel rendering and the overall usage of resources. The input for this test consists of the datasets already sorted in Morton order and the data is streamed directly from disk. Leaf collapse is disabled. Figure 4.3 shows the results. The overhead imposed is between 20% (David) and 34% (St.Matthew), which is an evidence that the overhead impact decreases as the dataset size increases. This is a desirable property for an algorithm designed to handle large datasets. The final observation from this experiment is that OMiCroN maintains the usage of all 8 logical cores near 90% with peaks of 100% for the entire hierarchy creation procedure, with parallel rendering enabled or disabled. This fact justifies OMiCroN's fast hierarchy creation times.

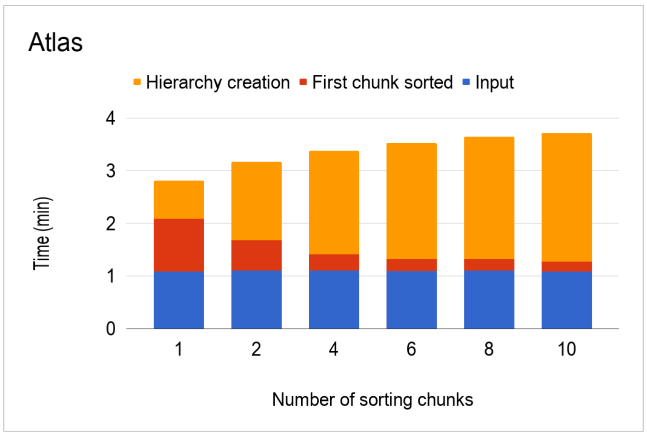
The third experiment generates data for better understanding the hierarchy creation progression over time. It consists of measuring the time needed to achieve percentile milestones of hierarchy creation. The best scenario is a linear progression over time so new data can be presented smoothly to the user while the hierarchy is being constructed. For this test, the sorted data is streamed directly from disk, parallel rendering is enabled and leaf collapse is disabled unless pointed otherwise. The results are presented in Figure 4.4. We can conclude that the hierarchy construction has the expected linear progression. The exception is the David dataset with leaf collapse disabled. This behavior is caused by the hierarchy size, which exceeds available memory, forcing the use of swap area and degradation of performance. This problem is solved when leaf collapse is enabled, as Figure 4.4 also demonstrates.

4.2 Hierarchy creation and rendering

A second set of experiments were conducted to assess OMiCroN's behavior in terms of memory usage and performance. All experiments in this set read a sorted dataset directly from disk. The test system had an Intel Core i7-6700, 16GB memory, NVidia GeForce GTX 1070, and secondary SSD storage with roughly 130 MB/s reading speed. Two main parameters impact OMiCroN's memory footprint: *Leaf Collapse* optimization and parent to children point ratio, as shown in Table 4.1. These also impact the reconstruction quality of the algorithm as can be seen in Figures 4.5, 4.6, 4.7 and 4.8.

Even though limited to datasets that fit in RAM unless swap space is used, OMiCroN can be set up to fit a broad range of memory budgets. For example, David originally occupies 11.2 GB in disk, while its maximum size in memory when





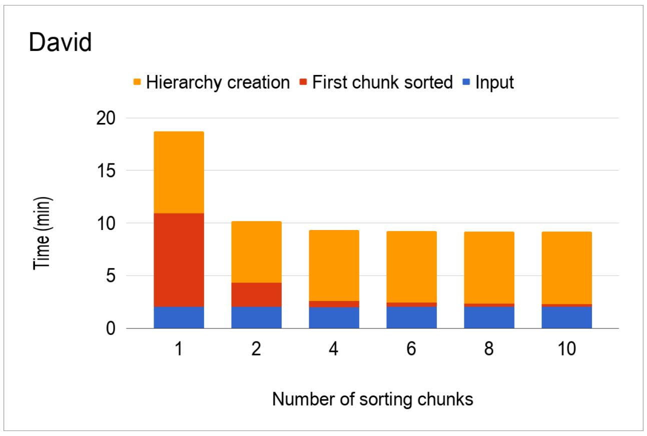


Figure 4.2: Impact of the number of sort chunks. After a constant time spent reading the input (blue), the first chunk is sorted (red), starting the parallel hierarchy creation and rendering (orange). The first column in all charts corresponds to the case where all input is sorted before the hierarchy creation begins.

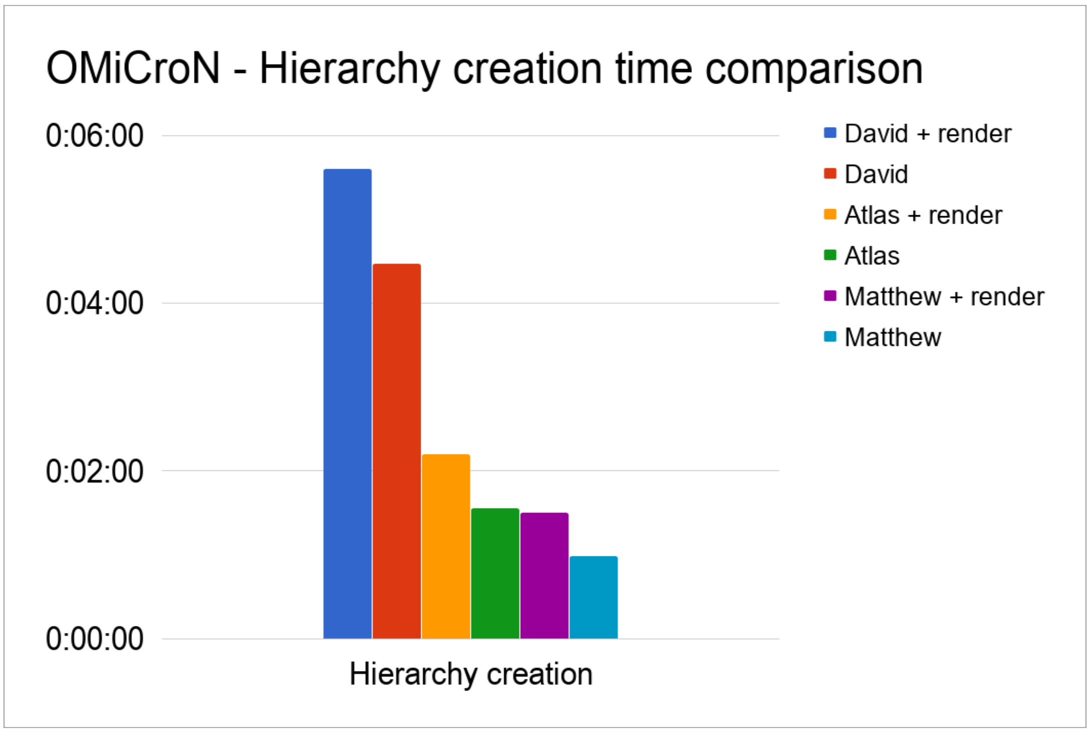


Figure 4.3: Comparison of hierarchy creation with and without parallel rendering. Sorted data is streamed directly from disk. The overhead imposed by parallel rendering is between 20% (David) and 34% (St. Matthew).

using *Leaf Collapse* is 8.5 or 9.9 GB, for parent to children point ratios of 0.2 and 0.25 respectively. In this case, a hierarchy with 0.2 ratio has memory usage of roughly 76% of the original dataset size in disk. Values smaller than these are possible since reconstruction results shown in Figure 4.5 are still acceptable. It is also important to note that the algorithm does not compact in any way the point or Morton code data. The use of such techniques would provide even better memory consumption.

Table 4.1 also shows that the total hierarchy creation times and the average CPU usage per frame are affected by the *Leaf Collapse* optimization. The CPU times were obtained during a rendering session where the camera is constantly moving trying to focus the parts of the model being read from disk. For the David dataset, for example, it takes 88.2s to read the data from disk, while OMiCroN imposes an overhead ranging from 0.66 to 1.6 in the tested scenarios. We also notice that CPU times are probably affected by the *Leaf Collapse* optimization because the hierarchy is simplified when the leaf nodes are removed, resulting in smaller hierarchy fronts.

The worklist size is the parameter that controls the work granularity in the hierarchy creation. In other words, it controls the throughput of new nodes available

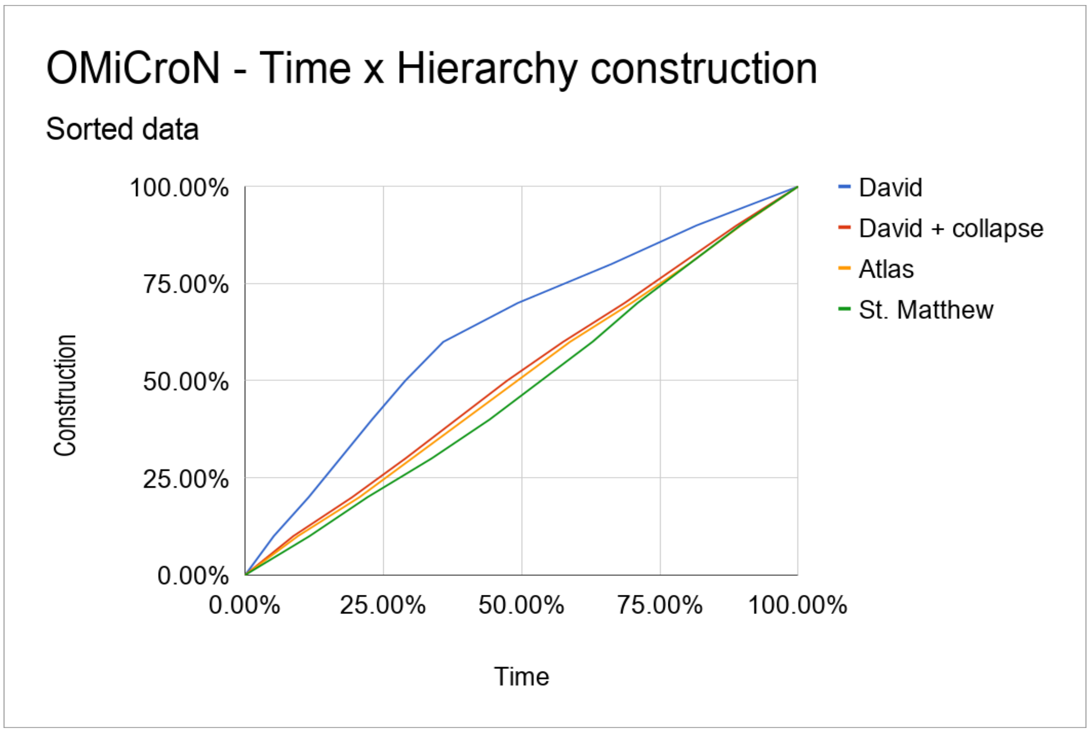


Figure 4.4: Hierarchy creation over time. Sorted data is streamed directly from disk, parallel rendering is enabled and leaf collapse is disabled unless pointed otherwise.

for the hierarchy creation threads to process. Table 4.2 shows the relationship between the worklist size and attributes that are expected to be directly affected by it. It also shows that the front insertion delay scales linearly with the worklist size. As a consequence, larger worklists impose a longer delay for the user to see new parts of the cloud while navigating. Additionally, the optimal worklist size regarding front size is between 32 and 64. Since nodes are processed in a bottom-up manner and smaller fronts are expected to have nodes from shallower parts of the hierarchy, setups with smaller fronts are also expected to have processed more nodes from deeper levels than other setups with larger fronts, given the same time spent in processing. As a consequence, hierarchy construction time is reduced in setups with smaller fronts, as Table 4.2 also indicates. Similarly, benefits in overall performance of front evaluation are obviously related to smaller front sizes, resulting in less CPU overhead.

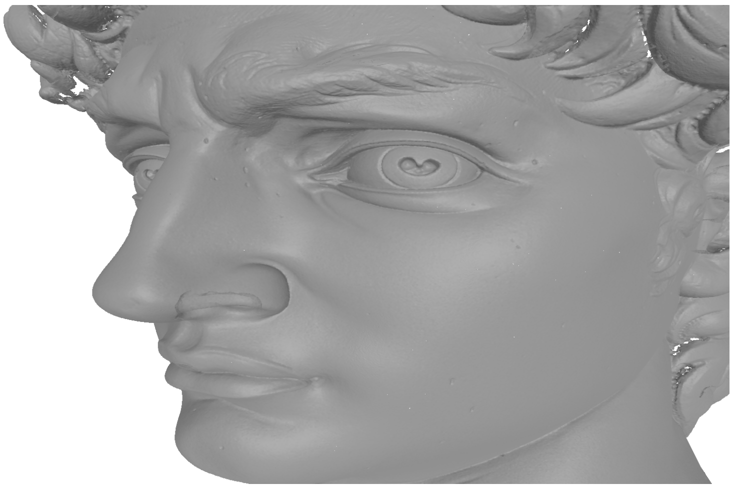
Another experiment we judge appropriate is comparing OMiCroN with the simplest algorithm to build an octree for large point clouds. This way we can evaluate if the relation between OMiCroN's benefits and complexity is worthwhile. The chosen algorithm is a monothreaded top-down octree creator which subdivides nodes when

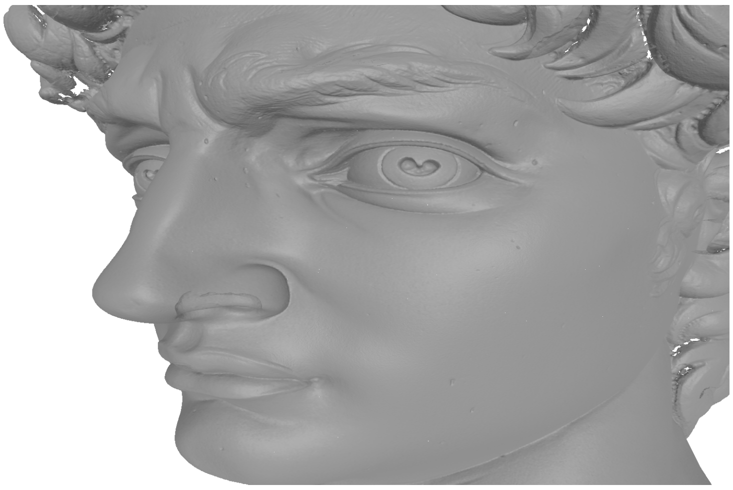
Table 4.1: Relationship between the algorithm reconstruction parameters – leaf collapse, parent to children ratio – and memory footprint, total hierarchy creation times, and average CPU usage per frame.

Model	Coll	Ratio	Mem	Creation	CPU
David	On	0.20	8.5GB	146.3s	7.6ms
David	On	0.25	9.9GB	151.2s	8.8ms
David	Off	0.20	21GB	229.8s	16.7ms
Atlas	On	0.20	2.3GB	77.8s	11.9ms
Atlas	On	0.25	3.0GB	81.9s	11.0ms
Atlas	Off	0.20	11.5GB	120.8s	16.2ms
Matthew	On	0.20	1.7GB	59.6s	13.7ms
Matthew	On	0.25	2.2GB	60.9s	11.6ms
Matthew	Off	0.20	8.4GB	80.6s	25.0ms
Duomo	On	0.20	0.9GB	31.0s	18.2ms
Duomo	On	0.25	1.2GB	32.6s	23.1ms
Duomo	Off	0.20	4.5GB	40.0s	21.9ms

Table 4.2: Relationship between the worklist size and performance indicators: front insertion delay, front size, hierarchy construction time and average CPU usage per frame. Numbers refer to the David dataset, no leaf collapse and point ratio 0.25.

Worklist	Insertion	Front	Hierarchy	CPU
8	127ms	529	274.8s	19.5ms
16	212ms	439	259.8s	17.8ms
32	399ms	401	248.6s	16.0ms
64	831ms	500	258.0s	20.8ms
128	1646ms	506	255.7s	19.7ms


(a) David, leaf collapse on, 0.2 point ratio.


(b) David, leaf collapse on, 0.25 point ratio.

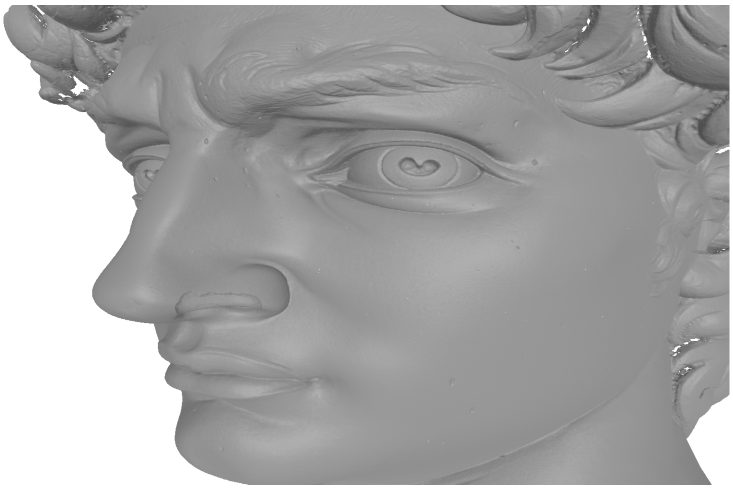
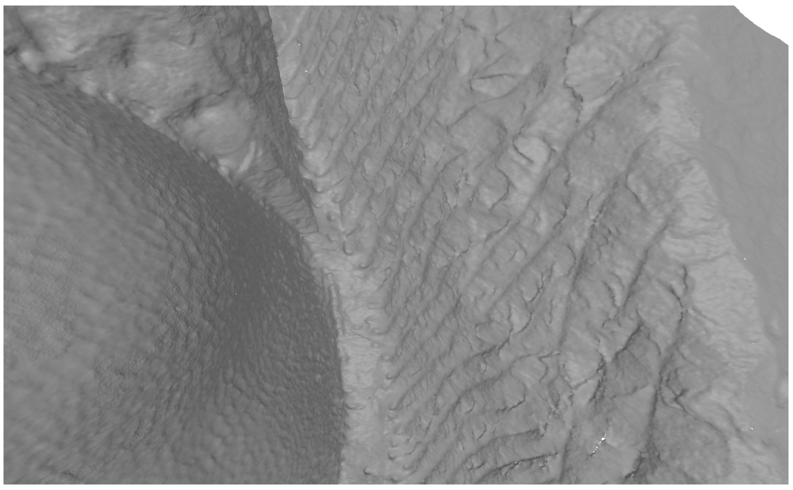
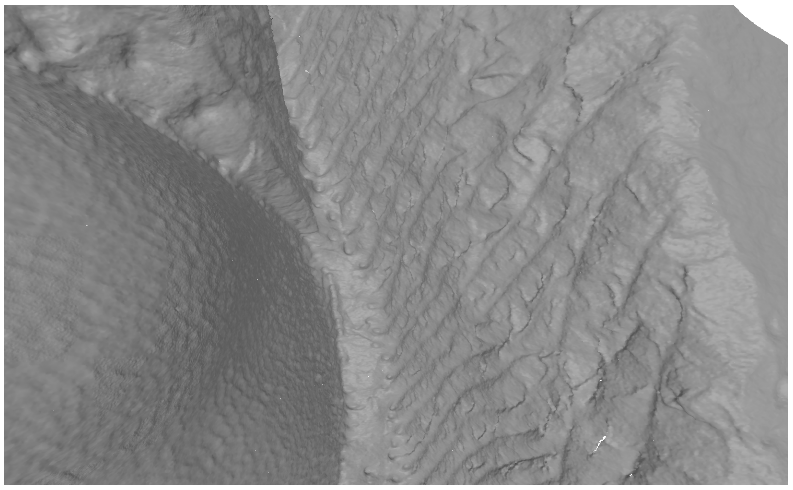

(c) David, leaf collapse off, 0.25 point ratio.

Figure 4.5: Rendering comparison of hierarchies with different leaf collapse and parent to children point ratio parameters. The final reconstructions are very detailed even at close range and the differences when the leaf collapse is turned on are almost imperceptible.


(a) Atlas, leaf collapse on, 0.2 point ratio.


(b) Atlas, leaf collapse on, 0.25 point ratio.

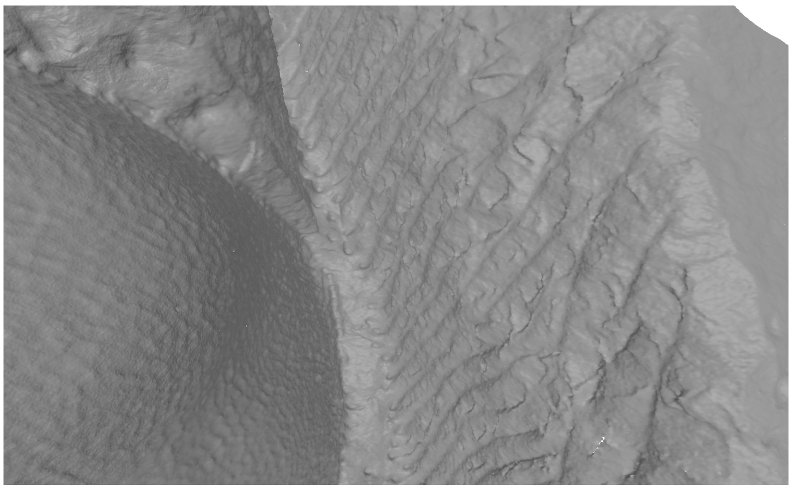
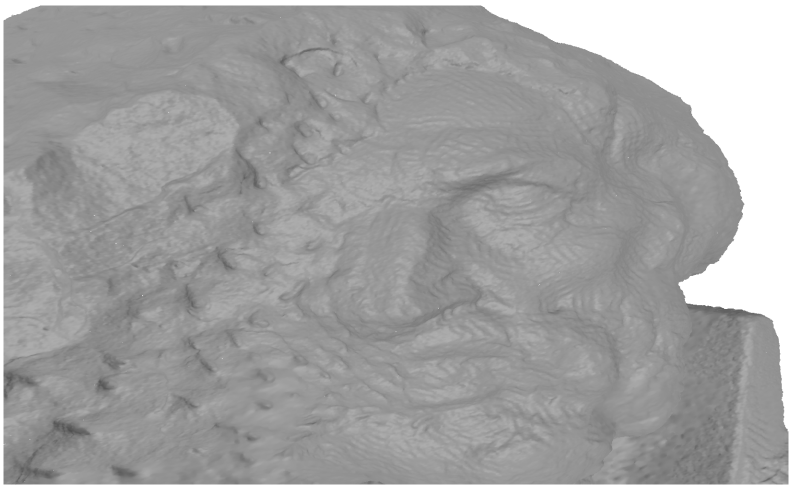
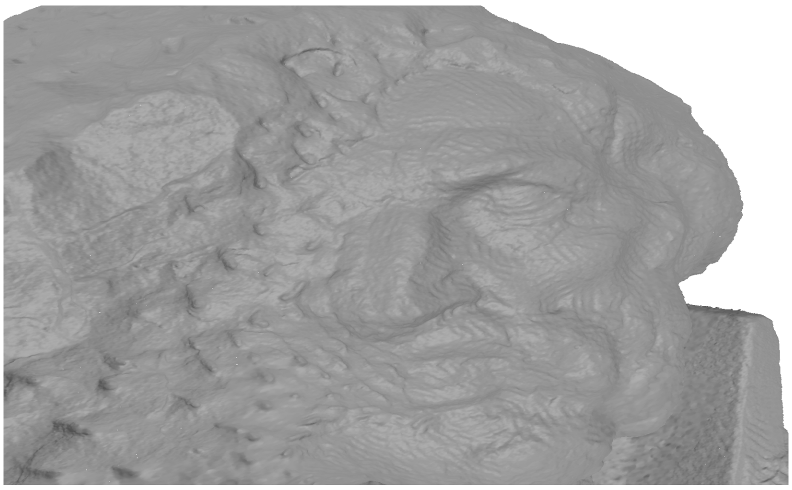

(c) Atlas, leaf collapse off, 0.25 point ratio.

Figure 4.6: Rendering comparison of hierarchies with different leaf collapse and parent to children point ratio parameters. The final reconstructions are very detailed even at close range and the differences when the leaf collapse is turned on are almost imperceptible.


(a) St. Matthew, leaf collapse on, 0.2 point ratio.


(b) St. Matthew, leaf collapse on, 0.25 point ratio.

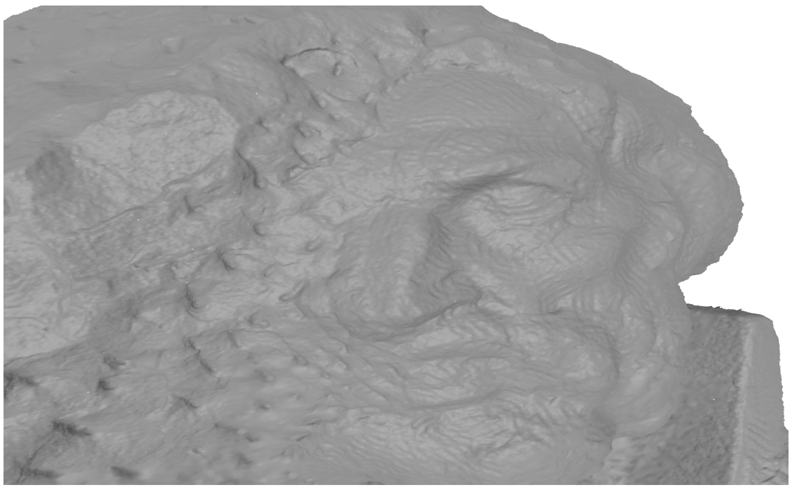
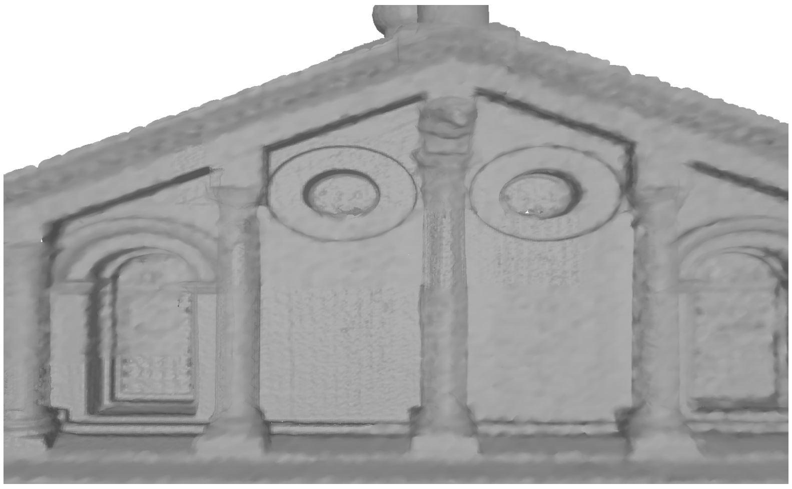
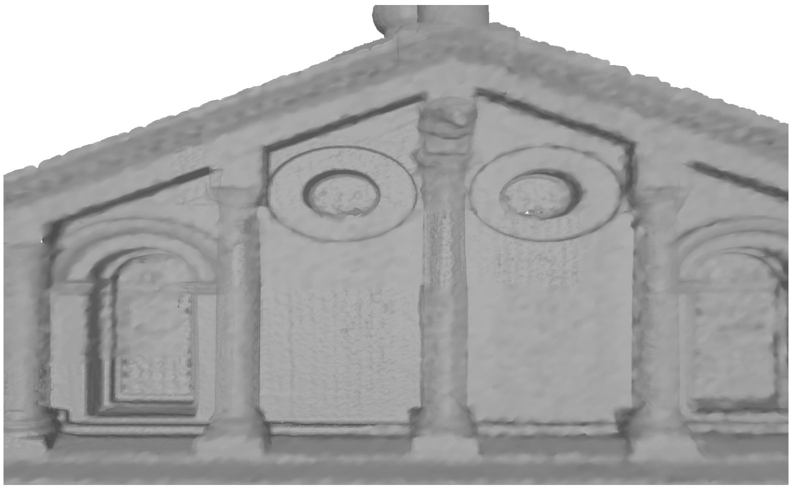

(c) St. Matthew, leaf collapse off, 0.25 point ratio.

Figure 4.7: Rendering comparison of hierarchies with different leaf collapse and parent to children point ratio parameters. The final reconstructions are very detailed even at close range and the differences when the leaf collapse is turned on are almost imperceptible.


(a) Duomo, leaf collapse on, 0.2 point ratio.


(b) Duomo, leaf collapse on, 0.25 point ratio.

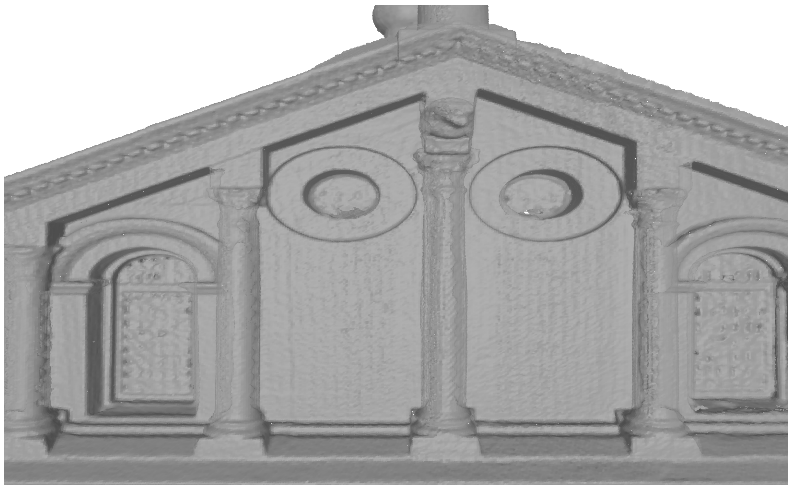

(c) Duomo, leaf collapse off, 0.25 point ratio.

Figure 4.8: Rendering comparison of hierarchies with different leaf collapse and parent to children point ratio parameters. The hierarchy for Duomo suffers from lack of density when leaf collapse is turned on because the dataset itself has smaller density in comparison with the others.

their size in number of points reaches a given threshold $K = 10000$. We have tested it with the Atlas and St. Matthew datasets only, since it exceeded memory limits for David. Including input time, the hierarchy for Atlas was created in 8.6 minutes, while St. Matthew was processed in 3.6 minutes. Comparing these results with Figure 4.2, OMiCroN not only creates the hierarchy several times faster, but it can also start rendering the datasets up to near 8 times earlier than the renderer using the top-down octree creator.

We also found it useful to compare OMiCroN with other algorithms that create hierarchies for large datasets. To this end, we used the binary voxelization method for large meshes described in [6], which is the only freely available implementation of an algorithm for creating Octrees for large datasets we could find. The paper presents another version with per-leaf normals and vertex colors, but it generates more than 100GB of intermediate data for some of our datasets. It should be noted that [6] operates on triangle meshes, and thus the input datasets are roughly twice as big as those containing only the vertices as used by OMiCroN. However, since a voxelization is an abrupt simplification of the original dataset, the difference in input is compensated by the fact that Octree nodes handled by OMiCroN are populated with thousands or millions of points while the Octree nodes in the voxelization are boolean values, resulting in extremely compact Octrees with just a few KBytes. For example, the Octree generated by OMiCroN for the David without leaf collapse has more than 22GB. In our tests, [6] was given a memory quota of 16GB and set to a grid size of 128, which is equivalent to a hierarchy of depth 7. OMiCroN was also run for depth 7, with no rendering, leaf collapse enabled, while sorting was performed with 10 chunks. All input datasets were not sorted. In our tests, OMiCroN finishes building the hierarchy 3 to 5 times faster than [6], which indicates that, even in a traditional setup where preprocessing precedes rendering, OMiCroN is still very competitive. Table 4.3 shows all statistics generated by this test.

4.3 Rendering parameters

The leaf splat tangents vectors u and v have constant length which are scaled up by multipliers creating parent nodes from children. These parameters impose a trade-off between blurriness and hole filling. In practice, a very large tangent size results in blurry reconstructions and poor performance, while a very small value results in holes and fast performance. In our experiments, we used leaf splat tangent vectors with sizes ranging from 0.00002 and 0.00008, while tangent multipliers were set between 2 and 5. Table 4.4 presents the range of leaf splat tangents sizes used in our tests, while Table 4.5 shows the best value ranges for splat tangent multipliers. We arrived at these values from trial and error experiments. It is important to

Table 4.3: Time comparison between OMiCroN with and without parallel rendering and [6]. I/O times are separated for preprocessing and creation tasks. Expections are tasks in [6] which does not report I/O times explictly. In these cases task times also include I/O. The most important lines are titled “Creation“, which contain the time needed to create the hierarchy after input and preprocessing, and “Until render“, which show the total time before any image is generated, including I/O. They show how OMiCroN minimizes data evaluation delays.

(a) David

	Rendering	No rendering	Baert [6]
Preprocessing	6m39s	6m39s	15m36s
Preprocessing Input	2m11s	2m11s	–
Preprocessing Output	3m41s	3m41s	–
Creation	5m36s	4m28s	15m39s
Until render	12m34s	17m1s	31m15s
Nodes	45936	45936	29576
Points	625M	625M	0

(b) Atlas

	Rendering	No rendering	Baert [6]
Preprocessing	48s	48s	5m16s
Preprocessing Input	1m14s	1m14s	–
Preprocessing Output	1m10s	1m10s	–
Creation	2m12s	1m33s	8m1s
Until render	3m12s	4m45s	13m17s
Nodes	49186	49186	71282
Points	340M	340M	0

(c) St. Matthew

	Rendering	No rendering	Baert [6]
Preprocessing	35s	35s	3m45s
Preprocessing Input	54s	54s	–
Preprocessing Output	45s	45s	–
Creation	1m30s	59s	5m54s
Until render	2m14s	3m13s	9m39s
Nodes	54788	54788	59277
Points	249M	249M	0

Table 4.4: Best values for leaf splat tangents size, per model.

Model	u	v
David	0.000037	0.00003
Atlas	0.00008	0.00007
St. Matthew	0.000085	0.000055
Duomo	0.00008	0.00002

Table 4.5: Best value ranges for splat tangent multipliers from hierachy level 7 to 3.

Level	u	v
7	3.8 – 4.7	3.5 – 4.7
6	2.3 – 3.0	1.9 – 2.5
5	2.0 – 2.8	1.5 – 2.3
4	2.0 – 2.5	1.7 – 2.5
3	2.0	2.0

remember that the datasets are normalized, since their scale affects the leaf splats tangent sizes.

The number of front segments is a parameter that can ease the CPU overhead at a cost of more delays in level of detail changes when rendering. After many experiments, we found that values ranging from 1 to 10 provide good results. Larger values excessively increase the front size since placeholders start to accumulate at the end of the front. These placeholders cannot suffer pruning, inhibitting the decrement in front size.

The projection threshold parameter is used to ensure proper level of detail when rendering, and we found that values around 0.2 provide good results. This value refers to the length of a node's projected diagonal in normalized device coordinates, so that nodes smaller than the threshold tend to be pruned while those larger tend to be branched. Larger values result in blurrier images as a consequence of coarser level-of-detail, but accelerate the renderer, while smaller values result in finer images but reduce the rendering speed. All tests in this chapter used a projection threshold of 0.2.

Chapter 5

Conclusion

In this work, we presented OMiCroN, a flexible and generic algorithm for rendering large point clouds. We know of no other method that can render incomplete hierarchies with full detail in parallel with its construction and data sorting. Rather, the vast majority of algorithms in this category rely on heavy preprocessing, which largely outweighs the time complexity of the rendering algorithm proper. OMiCroN, on the other hand, needs only a sorted prefix of the input geometry in Morton code order to start rendering. In practice, this sort can adapt to start rendering models as early as the time needed to read input. OMiCroN's feedback-based design allows construction of Octrees on-the-fly and can help implementors with accurate rendering feedback of the construction process. We also defined the novel idea of Hierarchy Oblique Cut, a strong concept that can be used to apply sweeps on hierarchies.

Additionally, OMiCroN opens the path for new workflows based on streaming of spatially sorted data. Supposing that large scans could be streamed directly in Morton order, the data could be rendered without any delays at all, enabling earlier detection of acquisition problems. Another advantage is that the hierarchical nature of Morton order can be explored, so datasets are sorted only once using a deep Morton code level but can be rendered by OMiCroN using a hierarchy with any level less or equal to the sorting level. This property renders the algorithm even more flexible, since a single sorted dataset can be used with many hierarchy setups.

OMiCroN has limitations. First, it is not out-of-core. We opt to not incorporate this paradigm yet because out-of-core algorithms are known to have non-trivial implementation [50]. Another limitation is the overhead caused by rendering the data while constructing its associated hierarchy, which may drain resources that otherwise could be used for more complex rendering techniques when preprocessing is available. However, the benefits of minimal delay requirements may be compelling for workflows that depend on fast evaluation of acquired data. This fact also makes OMiCroN a good choice for previewing large datasets before applying a more complex and heavy pre-processing technique.

Regarding future directions, OMiCroN has several possible paths to follow. The splat renderer uses parameters set manually during the experiments, since it was not the focus of this work, rather we concentrated our efforts on the hierarchy construction and high-level rendering management. However, it could be further improved by developing methods to automatically find the optimal parameters, such as initial u and v vectors, and a better hierarchical representation of the splats [63]. Also, although OMiCroN is limited to models that fit in RAM, it is in the path to be integrated with the out-of-core paradigm. This statement is justified by the node collapse concept, which can be substituted by a flush to secondary storage. [6] have already shown successful use of out-of-core paradigm for Morton-ordered data. Moreover, in theory, OMiCroN's deepest abstraction layer could be modified to use the algorithm in other Computer Graphics problems involving the use of Morton-ordered hierarchical structures, such as raytracing, voxelization and reconstruction.

Bibliography

[1] MUŁA, W. "Bitmap 8x8 pixels represented with compressed quadtree. Number denotes order of nodes." https://commons.wikimedia.org/wiki/File:Quad_tree_bitmap.svg, September 2008. Released in public domain.

[2] WHITETIMBERWOLF. "Schematic drawing of an octree, a data structure of computer science." https://commons.wikimedia.org/wiki/File:Octree2.svg, March 2010. Under Creative Commons Attribution-Share Alike 3.0 Unported license (https://creativecommons.org/licenses/by-sa/3.0/legalcode).

[3] KIWISUNSET. "A visualization of results obtained by running the Python kd-tree-construction program on [(2,3), (5,4), (9,6), (4,7), (8,1), (7,2)]". https://commons.wikimedia.org/wiki/File:Kdtree_2d.svg, April 2006. Under GNU Free Documentation License Version 1.2 or later (https://www.gnu.org/licenses/fdl-1.3.en.html) and Creative Commons Attribution-Share Alike 3.0 Unported (https://creativecommons.org/licenses/by-sa/3.0/legalcode).

[4] MYGUEL. "The binary tree corresponding to the 2D kd-tree example". https://commons.wikimedia.org/wiki/File:Tree_0001.svg, August 2008. Released into the public domain.

[5] CHRISJOHNSON. "Untitled". https://commons.wikimedia.org/wiki/File:Example_of_BSP_tree_traversal.svg, June 2012. Under Creative Commons Attribution-Share Alike 3.0 Unported license (https://creativecommons.org/licenses/by-sa/3.0/legalcode).

[6] BAERT, J., LAGAE, A., DUTRÉ, P. "Out-of-Core Construction of Sparse Voxel Octrees", *Computer Graphics Forum*, v. 33, n. 6, pp. 220–227, 2014. ISSN: 1467-8659. doi: 10.1111/cgf.12345. Disponível em: <`http://dx.doi.org/10.1111/cgf.12345`>.

[7] ERICSON, C. *Real-Time Collision Detection.* Boca Raton, FL, USA, CRC Press, Inc., 2004. ISBN: 1558607323, 9781558607323.

[8] BITTNER, J., HAPALA, M., HAVRAN, V. "Incremental BVH construction for ray tracing", *Computers & Graphics*, v. 47, pp. 135–144, 2015.

[9] GOLDSMITH, J., SALMON, J. "Automatic creation of object hierarchies for ray tracing", *IEEE Computer Graphics and Applications*, v. 7, n. 5, pp. 14–20, 1987.

[10] KAY, T. L., KAJIYA, J. T. "Ray tracing complex scenes". In: *ACM SIGGRAPH computer graphics*, v. 20, pp. 269–278. ACM, 1986.

[11] LEVOY, M., WHITTED, T. *The use of points as a display primitive.* Chapel Hill, NC, USA, University of North Carolina, Department of Computer Science, 1985.

[12] GROSSMAN, J. P., DALLY, W. J. "Point Sample Rendering". In: Drettakis, G., Max, N. (Eds.), *Rendering Techniques '98*, pp. 181–192, Vienna, 1998. Springer Vienna. ISBN: 978-3-7091-6453-2.

[13] SAINZ, M., PAJAROLA, R. "Point-based rendering techniques", *Computers & Graphics*, v. 28, n. 6, pp. 869–879, 2004.

[14] KOBBELT, L., BOTSCH, M. "A survey of point-based techniques in computer graphics", *Computers & Graphics*, v. 28, n. 6, pp. 801–814, 2004.

[15] ALEXA, M., GROSS, M., PAULY, M., et al. "Point-based computer graphics". In: *ACM SIGGRAPH 2004 Course Notes*, p. 7. ACM, 2004.

[16] GROSS, M. "Getting to the Point...?" *IEEE Computer Graphics and Applications*, v. 26, n. 5, pp. 96–99, 2006.

[17] GROSS, M., PFISTER, H. *Point-Based Graphics.* San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2011. ISBN: 0123706041, 9780080548821.

[18] RAMOS, F., HUERTA, J., BENITEZ, F. "Characterization of Multiresolution Models for Real-Time Rendering in GPU-Limited Environments". In: *International Conference on Articulated Motion and Deformable Objects*, pp. 157–167. Springer, 2016.

[19] RUSINKIEWICZ, S., LEVOY, M. "QSplat: A Multiresolution Point Rendering System for Large Meshes". In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '00, pp.

343–352, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN: 1-58113-208-5. doi: 10.1145/344779.344940. Disponível em: <http://dx.doi.org/10.1145/344779.344940>.

[20] PFISTER, H., ZWICKER, M., VAN BAAR, J., et al. “Surfels: Surface Elements As Rendering Primitives”. In: *Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’00, pp. 335–342, New York, NY, USA, 2000. ACM Press/Addison-Wesley Publishing Co. ISBN: 1-58113-208-5. doi: 10.1145/344779.344936. Disponível em: <http://dx.doi-org.ez29.capes.proxy.ufrj.br/10.1145/344779.344936>.

[21] WESTOVER, L. “Interactive Volume Rendering”. In: *Proceedings of the 1989 Chapel Hill Workshop on Volume Visualization*, VVS ’89, pp. 9–16, New York, NY, USA, 1989. ACM. doi: 10.1145/329129.329138. Disponível em: <http://doi-acm-org.ez29.capes.proxy.ufrj.br/10.1145/329129.329138>.

[22] ZWICKER, M., PFISTER, H., VAN BAAR, J., et al. “Surface Splatting”. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH ’01, pp. 371–378, New York, NY, USA, 2001. ACM. ISBN: 1-58113-374-X. doi: 10.1145/383259.383300. Disponível em: <http://doi.acm.org/10.1145/383259.383300>.

[23] SAGAN, H. *Space-filling curves*. Springer Science & Business Media, 2012.

[24] DE CAMPOS, A. M. “3 first steps of the building of the Peano fractal curve”. https://commons.wikimedia.org/wiki/File:Peanocurve.svg, June 2007. Under Creative Commons Attribution-Share Alike 3.0 Unported license (https://creativecommons.org/licenses/by-sa/3.0/legalcode).

[25] RICHARDS, G. “First, second, and third order Hilbert Curves overlayed, with the lines getting thinner and darker as the order increases.” https://commons.wikimedia.org/wiki/File:Hilbert_curve_3.svg, July 2008. Released into the public domain.

[26] EPPSTEIN, D., HESPERIAN. “Four levels of the Z curve, showing the square that is eventually filled by the curve. Redrawn from bitmap image en:Image:Zorder mirrored.png.” https://commons.wikimedia.org/wiki/File:Four-level_Z.svg, April 2008. Under GNU Free Documentation License Version 1.2 or later (https://www.gnu.org/licenses/fdl-1.3.en.html) and

Creative Commons Attribution-Share Alike 3.0 Unported (https://creativecommons.org/licenses/by-sa/3.0/legalcode).

[27] MORTON. *A computer oriented geodetic data base and a new technique in file sequencing.* Relatório Técnico Ottawa, Ontario, Canada, IBM Ltd., 1966.

[28] DE BERG, M., VAN KREVELD, M., OVERMARS, M., et al. "Computational geometry". In: *Computational geometry*, Springer, pp. 1–17, 2000.

[29] MEHTA, D. P., SAHNI, S. *Handbook Of Data Structures And Applications (Chapman & Hall/Crc Computer and Information Science Series.).* Chapman & Hall/CRC, 2004. ISBN: 1584884355.

[30] FINKEL, R. A., BENTLEY, J. L. "Quad trees a data structure for retrieval on composite keys", *Acta Informatica*, v. 4, n. 1, pp. 1–9, Mar 1974. ISSN: 1432-0525. doi: 10.1007/BF00288933. Disponível em: <`https://doi.org/10.1007/BF00288933`>.

[31] SAMET, H. "The Quadtree and Related Hierarchical Data Structures", *ACM Comput. Surv.*, v. 16, n. 2, pp. 187–260, jun. 1984. ISSN: 0360-0300. doi: 10.1145/356924.356930. Disponível em: <`http://doi.acm.org/10.1145/356924.356930`>.

[32] MEAGHER, D. J. *Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer.* Electrical and Systems Engineering Department Rensseiaer Polytechnic Institute Image Processing Laboratory, 1980.

[33] BENTLEY, J. L. "Multidimensional Binary Search Trees Used for Associative Searching", *Commun. ACM*, v. 18, n. 9, pp. 509–517, set. 1975. ISSN: 0001-0782. doi: 10.1145/361002.361007. Disponível em: <`http://doi.acm.org/10.1145/361002.361007`>.

[34] SCHUMACKER, R. A., BRAND, B., GILLILAND, M. G., et al. *Study for applying computer-generated images to visual simulation.* Relatório técnico, GENERAL ELECTRIC CO DAYTONA BEACH FL APOLLO AND GROUND SYSTEMS, 1969.

[35] FUCHS, H., KEDEM, Z. M., NAYLOR, B. F. "On Visible Surface Generation by a Priori Tree Structures". In: *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '80, pp. 124–133, New York, NY, USA, 1980. ACM. ISBN: 0-89791-021-4. doi: 10.1145/800250.807481. Disponível em: <`http://doi-acm-org.ez29.capes.proxy.ufrj.br/10.1145/800250.807481`>.

[36] KLOSOWSKI, J. T. *Efficient collision detection for interactive 3 D graphics and virtual environments*. Tese de Doutorado, State University of New York at Stony Brook, 1998.

[37] EHMANN, S. A., LIN, M. C. “Accurate and fast proximity queries between polyhedra using convex surface decomposition”, *Computer Graphics Forum*, v. 20, n. 3, pp. 500–511, 2001.

[38] LAUTERBACH, C., MO, Q., MANOCHA, D. “gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries.” *Comput. Graph. Forum*, v. 29, n. 2, pp. 419–428, 2010.

[39] ARGUDO, O., BESORA, I., BRUNET, P., et al. “Interactive inspection of complex multi-object industrial assemblies”, *Computer-Aided Design*, v. 79, pp. 48 – 59, 2016. ISSN: 0010-4485. doi: https://doi.org/10.1016/j.cad.2016.06.005. Disponível em: <`http://www.sciencedirect.com/science/article/pii/S0010448516300628`>.

[40] DACHSBACHER, C., VOGELGSANG, C., STAMMINGER, M. “Sequential point trees”. In: *ACM Transactions on Graphics (TOG)*, v. 22, pp. 657–662. ACM, 2003.

[41] PAJAROLA, R., SAINZ, M., LARIO, R. “Xsplat: External memory multiresolution point visualization”. In: *Proceedings IASTED Invernational Conference on Visualization, Imaging and Image Processing*, pp. 628–633, 2005.

[42] WIMMER, M., SCHEIBLAUER, C. “Instant Points: Fast Rendering of Unprocessed Point Clouds”. In: *Proceedings of the 3rd Eurographics / IEEE VGTC Conference on Point-Based Graphics*, SPBG’06, pp. 129–137, Aire-la-Ville, Switzerland, Switzerland, 2006. Eurographics Association. ISBN: 3-905673-32-0. doi: 10.2312/SPBG/SPBG06/129-136. Disponível em: <`http://dx.doi.org/10.2312/SPBG/SPBG06/129-136`>.

[43] GOBBETTI, E., MARTON, F. “Layered Point Clouds: A Simple and Efficient Multiresolution Structure for Distributing and Rendering Gigantic Point-sampled Models”, *Comput. Graph.*, v. 28, n. 6, pp. 815–826, dez. 2004. ISSN: 0097-8493. doi: 10.1016/j.cag.2004.08.010. Disponível em: <`http://dx.doi.org/10.1016/j.cag.2004.08.010`>.

[44] WAND, M., BERNER, A., BOKELOH, M., et al. “Interactive Editing of Large Point Clouds.” In: *SPBG*, pp. 37–45, 2007.

[45] BETTIO, F., GOBBETTI, E., MARTON, F., et al. "A Point-based System for Local and Remote Exploration of Dense 3D Scanned Models." In: *VAST*, pp. 25–32, 2009.

[46] HUBO, E., BEKAERT, P. "A data distribution strategy for parallel point-based rendering". In: *WSCG '2005: Full Papers: The 13-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision 2005 in co-operation with EUROGRAPHICS: University of West Bohemia, Plzen, Czech Republic, p. 1-8.*, 2005.

[47] CORRÊA, W. T., FLEISHMAN, S., SILVA, C. T. "Towards point-based acquisition and rendering of large real-world environments". In: *Computer Graphics and Image Processing, 2002. Proceedings. XV Brazilian Symposium on*, pp. 59–66. IEEE, 2002.

[48] CORRÊA, W. T., KLOSOWSKI, J. T., SILVA, C. T. "Out-of-core sort-first parallel rendering for cluster-based tiled displays", *Parallel Computing*, v. 29, n. 3, pp. 325–338, 2003.

[49] GOSWAMI, P., MAKHINYA, M., BÖSCH, J., et al. "Scalable Parallel Out-of-core Terrain Rendering." In: *EGPGV*, pp. 63–71, 2010.

[50] GOSWAMI, P., EROL, F., MUKHI, R., et al. "An efficient multi-resolution framework for high quality interactive rendering of massive point clouds using multi-way kd-trees", *The Visual Computer*, v. 29, n. 1, pp. 69–83, 2013. ISSN: 1432-2315. doi: 10.1007/s00371-012-0675-2. Disponível em: <`http://dx.doi.org/10.1007/s00371-012-0675-2`>.

[51] LUKAC, N., OTHERS. "Hybrid visualization of sparse point-based data using GPGPU". In: *Computing and Networking (CANDAR), 2014 Second International Symposium on*, pp. 178–184. IEEE, 2014.

[52] GAO, Z., NOCERA, L., WANG, M., et al. "Visualizing Aerial LiDAR Cities with Hierarchical Hybrid Point-polygon Structures". In: *Proceedings of Graphics Interface 2014*, GI '14, pp. 137–144, Toronto, Ont., Canada, Canada, 2014. Canadian Information Processing Society. ISBN: 978-1-4822-6003-8. Disponível em: <`http://dl-acm-org.ez29.capes.proxy.ufrj.br/citation.cfm?id=2619648.2619672`>.

[53] RICHTER, R., DISCHER, S., DÖLLNER, J. "Out-of-core visualization of classified 3d point clouds". In: *3D Geoinformation Science*, Springer, pp. 227–242, 2015.

[54] FEBRETTI, A., RICHMOND, K., DORAN, P., et al. "Parallel processing and immersive visualization of sonar point clouds". In: *Large Data Analysis and Visualization (LDAV), 2014 IEEE 4th Symposium on*, pp. 111–112. IEEE, 2014.

[55] POTENZIANI, M., CALLIERI, M., DELLEPIANE, M., et al. "3dhop: 3d heritage online presenter", *Computers & Graphics*, v. 52, pp. 129–141, 2015.

[56] TREDINNICK, R., BROECKER, M., PONTO, K. "Experiencing interior environments: New approaches for the immersive display of large-scale point cloud data". In: *Virtual Reality (VR), 2015 IEEE*, pp. 297–298. IEEE, 2015.

[57] OKAMOTO, H., MASUDA, H. "A Point-Based Virtual Reality System for Supporting Product Development". In: *ASME 2016 International Design Engineering Technical Conferences and Computers and Information in Engineering Conference*, pp. V01BT02A052–V01BT02A052. American Society of Mechanical Engineers, 2016.

[58] PONTO, K., TREDINNICK, R., CASPER, G. "Simulating the experience of home environments". In: *Virtual Rehabilitation (ICVR), 2017 International Conference on*, pp. 1–9. IEEE, 2017.

[59] KARRAS, T. "Maximizing parallelism in the construction of BVHs, octrees, and k-d trees". In: *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics*, pp. 33–37. Eurographics Association, 2012.

[60] ZWICKER, M., RÄSÄNEN, J., BOTSCH, M., et al. "Perspective Accurate Splatting". In: *Proceedings of Graphics Interface 2004*, GI '04, pp. 247–254, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. Canadian Human-Computer Communications Society. ISBN: 1-56881-227-2. Disponível em: <`http://dl.acm.org/citation.cfm?id=1006058.1006088`>.

[61] BOTSCH, M., SPERNAT, M., KOBBELT, L. "Phong Splatting". In: *Proceedings of the First Eurographics Conference on Point-Based Graphics*, SPBG'04, pp. 25–32, Aire-la-Ville, Switzerland, Switzerland, 2004. Eurographics Association. ISBN: 3-905673-09-6. doi: 10.2312/SPBG/SPBG04/025-032. Disponível em: <`http://dx.doi.org/10.2312/SPBG/SPBG04/025-032`>.

[62] WEYRICH, T., HEINZLE, S., AILA, T., et al. “A Hardware Architecture for Surface Splatting”. In: *ACM SIGGRAPH 2007 Papers*, SIGGRAPH ’07, New York, NY, USA, 2007. ACM. doi: 10.1145/1275808.1276490. Disponível em: <`http://doi.acm.org/10.1145/1275808.1276490`>.

[63] WU, J., ZHANG, Z., KOBBELT, L. “Progressive Splatting”. In: *Proceedings Eurographics/IEEE VGTC Symposium Point-Based Graphics, 2005.*, pp. 25–142, June 2005. doi: 10.1109/PBG.2005.194060.