FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Hierarchical Multi-resolution Data Structure for Molecular Visualization

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Visual Computing

by

Milena Nowak, BSc
Registration Number 0927584

to the Faculty of Informatics
at the TU Wien
Advisor: Prof. Dipl.-Ing. Dr. techn. Stefan Bruckner

Vienna, 27th July, 2020

Milena Nowak	Stefan Bruckner

Erklärung zur Verfassung der Arbeit

Milena Nowak, BSc
Speckbachergasse 13, 1160 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 27. Juli 2020

Milena Nowak

Acknowledgements

There is a reason the first person who receives thanks is generally the person who supervised the thesis. In Norwegian, the role is described as "leading the way" towards finishing a degree. That job, it turns out, involves a lot of work. Therefore, I would first and foremost like to thank Stefan Bruckner for his patience and clarity.

But academic support is not the only kind needed, so my thanks also go to Marte, who's company made long working hours during short autumn days a lot more pleasant, and Rebecca, for her invaluable moral support and feedback.

I would also like to thank everyone who made it possible for me to spend most of my degree in Bergen, as well as my family for making it clear that I have their support - and a home - no matter where I am.

Kurzfassung

Biomolekulare Datensätze sind nicht nur komplex, sie wachsen auch mit der Entwicklung des Forschungsgebietes. Um ihre Eigenschaften zu erforschen, verwenden Experten unter anderem dreidimensionale Modelle der Datensätze, die aus Millionen individueller Moleküle bestehen. Bei der visuellen Analyse der Modelle kann es außerdem hilfreich sein, durch verschiedene Darstellungsformen strukturelle Eigenschaften zur Geltung zu bringen. Daher gibt es einen Bedarf an flexiblen Datenstrukturen, die es ermöglichten, effizient mit solche Datensätze zu arbeiten.

Vorhandene Lösungsansätze im Bereich großer, auf Punktwolken aufbauender Datensätze verwenden in den meisten Fällen rasterbasierte Strukturen in verschiedenen Auflösungen. Andere Publikation konzentrieren sich dagegen auf die Verbesserung der Oberflächendarstellung, oder auf sich wiederholende Strukturen innerhalb großer Datensätze.

Wir schlagen eine Octree Datenstruktur vor, die die Daten räumlich so aufteilt, dass jeder Datenblock eine ähnliche Menge Datenpunkte beinhaltet, und mehrere, interpolierbaren Auflösungsstufen zur Verfügung . Der Aufbau der Datenstruktur erfolgt in einem Vorverarbeitungsschritt, dessen Resultat gespeichert wird, und daher nur ein Mal nötig ist. Zusätzlich effektivieren wir die Darstellung durch die Verwendung eines Least Recently Used Cache, der das Laden sichtbarer Datenblöcke verwaltet, und durch perspektivenabhängige Detaildarstellung.

In unserer Auswertung zeigen wir, dass insbesondere das Anpassen des dargestellten Detailgrades die Frameraten signifikant erhöht. Die Kombination einer höheren Auflösung im Vordergrund, bei gleichzeitiger Reduktion der Daten im Hintergrund, erhöht die Geschwindigkeit deutlich, ohne, dass die visuelle Qualität beeinträchtig wird. Durch die Optionen, Details zu reduzieren und nur Datenblöcke zu laden, die im Blickfeld der Kamera sind, wird es möglich, Datensätze anzuzeigen, die ansonsten die Kapazität der uns zur Verfügung stehenden Ressourcen überlasten würden. Der Vorteil der auf der Dichte des Datensatzes basierenden räumlichen Aufteilung im Gegensatz zu einer regelmäßigen Aufteilung zeigt sich besonders bei der Verwendung von Algorithmen, die bei der Berechnung Nachbarschaften in Betracht ziehen. Das könnte insbesondere bei der Implementierung eines Solvent Excluded Surface (SES) Oberflächenmodelles, einer der wichtigsten, aber rechnerisch aufwändigen, Darstellungsformen solcher Datensätze, von großem Vorteil sein. Die von uns vorgeschlagene Datenstruktur ist für Datensätze optimiert, die mehrere Millionen Punkte in einem einzigen Zeitschritt beinhalten.

Abstract

The complexity of biomolecular data sets is both high, and still rising. Three-dimensional models of molecules are used in research to test and investigate their properties. Such models can consist of several millions of atoms. Additionally, visual enhancement methods and molecular surface models are helpful when visualizing molecules. There is therefore a demand for efficient and flexible data structures to accommodate such large point-based data sets.

Existing solutions in the field of molecular visualization for large data sets include the use of, in most cases, regular grid-based data structures, as well as levels of detail. Other papers focus on repeating structures or improving the efficiency of surface models.

We propose an octree-based data structure that divides space into areas of similar density, and provides several levels of detail. Our approach is optimized for a single time-step, moving much of the computational overhead into a pre-processing step. This allows us to speed up frame rates for interactive visualizations using visibility culling, least recently used caching based on the pre-built octree data structure, and level of detail solutions such as depth-based level of detail rendering.

In our evaluation, we show that level of detail rendering significantly improves frame rates, especially in the case of distance-based level of detail selection while keeping the amount of details in the foreground high. Both the possibility to reduce the resolution and the caching strategy that allows us to only upload visible parts of the data set make it possible to render data sets that previously exhausted the capacities of our test set-up. We found the main advantage of a density based octree, instead of a regular division of space, to be in neighbourhood-based calculations, such as the clustering algorithm required to build levels of detail. This could prove particularly useful for the implementation of a Solvent Excluded Surface (SES) representation model, which would be an important feature to include when developing the framework further.

Contents

CHAPTER 1

Introduction

The art and science of visualization itself long predates the field of visual computing. Within literature, several varying definitions of the term visualization exist. Broadly speaking, visualization could be described as the meaningful transformation of data or information into graphical representations. More specifically, according to Senay and Ignatius [SI94], "The primary objective in data visualization is to gain insight into an information space by mapping data onto graphical primitives." The necessary ingredients are therefore data that you want to explore or explain, suitable tools to achieve the desired illustration or rendering, and of course certain skills and knowledge to do it successfully. Traditional visualization techniques generally rely on hand-drawn 2D illustrations. Possibly the most well-known historical example of 2D visualization is an 1869 graphic by Charles Joseph Minard shown in Figure 1.1a. It depicts the march of Napoleon's army into Russia in 1812 and 1813, and according to Tufte [Tuf83], "It may well be the best statistical graphic ever produced."

The same frameworks and techniques that are used to communicate patterns or interesting features in a data set, can often also be used to explore data. The term scientific visualization often refers to a specific type of data visualization, created in order to communicate or explore data obtained in a scientific field. Senay and Ignatius [SI94] focus their definition on the aspect of exploration, stating that "Scientific data visualization supports scientists and relations, to prove or disprove hypotheses, and discover new phenomena using graphical techniques." Another similarly famous historical visualization, the map of the cholera outbreak in London in 1854 by Dr. John Snow, is an example of data visualization being used to investigate a hypothesis. Using the statistical visualization shown in Figure 1.1b, Dr. Snow was able to support his hypothesis that the epidemic was caused by water from a contaminated well, and convince the authorities to remove the handle of the well. While Tufte [Tuf97] points out that it is problematic to use density based visualization techniques without factoring in population density, the graphic

remains famous as it was used to successfully explore and convincingly communicate a hypothesis that was later proven to be correct.

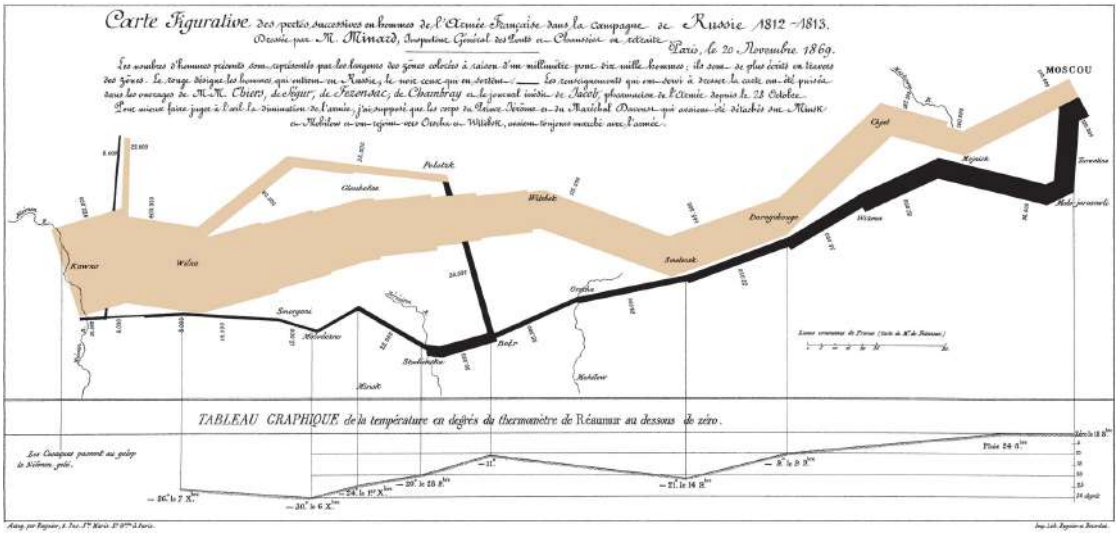
Our work is concerned with molecular visualization, which can be considered a specific branch of scientific data visualization. The definition given by Senay and Ignatius is particularly relevant to our work, which focuses on facilitating interactive exploration of molecular data. As we will see in Section 1.1, there are several widely used conventions on how to visualize molecules using computer graphics. An example of non-digital 3D data visualization from the field of molecular graphics is the original so-called ball-and-stick model, developed by August Wilhelm Hofmann. He built the models for explaining structural formulae out of croquet balls and presented them while lecturing in London in 1865 (see Travis [Tra92]). Versions of the ball-and-stick model are still used in current molecular visualizations in the field of computer graphics.

1.1 Background

Before moving on to the question of how we use scientific visualization in the field of biomolecules, a look at the basics of the matter to be visualized provides some useful context. In the introductory remarks of his book on molecular cell biology, Lodish [Lod00] explains the convoluted structures biological matter consists of. Within an organism, there are organs, which consist of tissue, those in turn consist of cells, and cells are composed of atoms. All biological systems are made up of the same types of atoms and are organized based on similar principles at the cellular level. Biomolecules are divided into macromolecules, such as proteins, lipids, and nucleic acids, and small molecules. Molecules are the material from which cells are built, but they also have other functions, such as carrying messages from cell to cell. Albeit a very short summary of the complex topic of molecular biology, this should provide some idea of why improving the understanding of what he calls "the molecules of life" is important. Lodish suggests that in order to learn about biological systems, we have to regard them one segment at a time. In order to do that, it is helpful to have suitable visualizations at one's disposal.

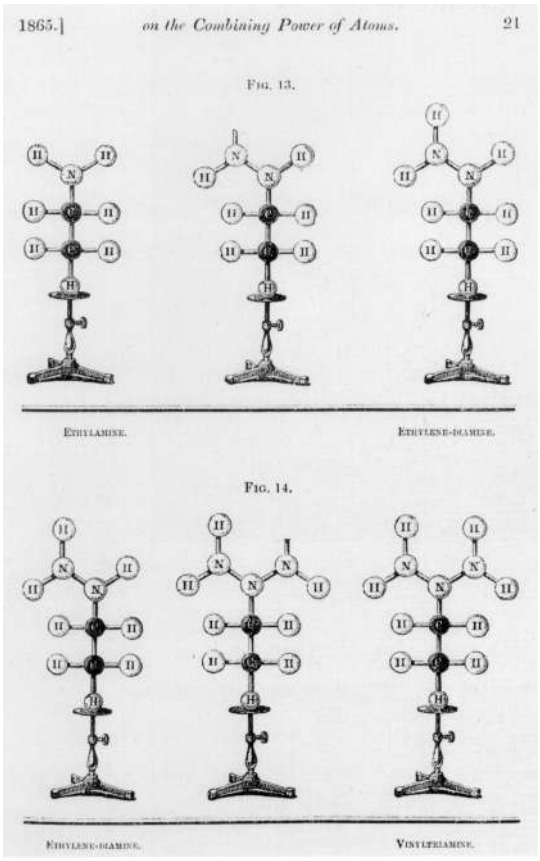
To make molecular structures intelligible, including their properties and interactions, is the primary purpose of molecular visualization according to Kozlíková et al. [KKF+17]. They also provide an overview over the history of molecular illustrations prior to the digital age. Before (interactive) visualizations could be created by computer graphics, depictions were drawn by hand. Atoms were already illustrated as spheres in the 17th century. In 1873, van der Waals [VdW73] approximated the volume occupied by individual atoms and molecules using experiments. The resulting approximate radii for chemical elements have been used to visualize atoms and molecules ever since. More elaborate atomic models described in the early 20th century lead to suggestions of more detailed illustrations. However, the majority of recent papers on molecular visualizations we reviewed in the context of this work still use van der Waals spheres as the basis of their visualizations.

According to Francoeur [Fra02], the earliest efforts to use computer graphics to create



(a) The visualization refered to as "Napoleon's March" designed by Minard in 1869 is a much cited example of data or information visualization. (From Tufte [Tuf83], who considers this to be an excellent statistical graphic)

(b) Dr. Snow used a map-based visualization of a part of London that was hit particularily hard by the cholera epidemic in 1854 to convince authorities of his hypothesis that th origin of the epidemic was a contaminated well. (From Tufte [Tuf97])



(c) Illustrations of the so-called ball-and-stick molecular model made from croquet balls that were introduced by Hofmann in 1865. (From Travis [Tra92])

Figure 1.1: Historical examples of data visualizations

interactive representations of molecular structures were made by a team around Cyrus Levinthal at the Massachusetts Institute of Technology in the mid-1960s. Techniques in the subfield of the interactive visualization of biomolecular structures have mostly been developed during the last two decades (see Kozlíková et al. [KKF+17]). Marschner and Shirley [MS15], define computer graphics as "any use of computers to create and manipulate images", which includes technical illustrations and animations. When the aim is an interactive application, as is the case in our work, the particular challenge is to create the images within the required time frame, that is generally in real-time. Martin [Mar65] points out that different authorities define the term real-time differently. So rather than specifying any particular frame-rate, he defines a real-time system as one which receives and processes data, and returns the results "sufficiently quickly to affect the functioning of the environment at that time.", which is a rather system-oriented definition. Laplante and Ovaska [LO11] introduce the term with the claim that most people would understand real-time to mean something along the lines of "at once" or "instantaneously". As an intuitive definition, they call a system interactive when, in order to avoid system failure, it needs to process information within a specified interval, which is similar to the definition given by Martin [Mar65]. The literature we reviewed, generally uses the term real-time without giving a precise threshold in terms of frames per second (fps), but rather provides the reader with the achieved frame-rates. As an approximate guideline, we can point to the speed at which the human eye can process images, which is 10-12 images per second, or the speeds used in film making history, 16 or 24 frames per second (see Read and Meyer [RM00]).

1.2 Motivation

The basic question of why biomolecules are worth investigating should, among other possible answers, be answerable by the simple fact that they are the building blocks the human body is made of. When it comes to why we need visualizations, Jones et al. [JJS05] point out that there are chemical phenomena that require visualization tools to investigate and describe them, as they are not obvious without visualization. They also note that such tools can be used to visually communicate complex molecular interactions and dynamics that are difficult to describe using only words. In particular, the interactive visualization of complex molecular data is highly valued by domain experts. Craig et al. [CMB13] emphasize that interactive animations can communicate structural information much more effectively than static images.

An important computational aspect of the complexity of molecular data is the amount of data to be processed at interactive rates. Molecular structures can contain millions of atoms, creating very large data sets. For example, the largest single molecule we use in our tests, contains 2,440,800 atoms, while our largest artificially created test set contains 122,040,000 atoms. As we will see in Chapter 2 and Chapter 3, there are certain limitations to the amount of data that can be processed at the same time. While the exact constraints depend on the system, the fact that there is a limit remains true across hardware and software capabilities.

According to our research, atoms in most molecular data sets are represented as point-based data (see Chapter 2). It is therefore possible to take inspiration from existing solutions for large point-based data sets. However, molecular data sets pose some specific challenges that are not necessarily relevant for other point-based data sets. In particular, several different surface models for molecular representations have been proposed and are requested by domain experts. Notable surface models include the Solvent Accessible Surface (Lee and Richards [LR71]), Solvent Excluded Surface (Richards [Ric77]), Ligand Excluded Surface (Lindow et al. [LBH14]), and the Gaussian Surface (Blinn [Bli82]). These are, in various degrees, computationally expensive and partially dependent on neighborhood queries (for more detail see Chapter 2 and Chapter 4 or refer to Kozlíková et al. [KKF+17]). Our aim is to implement a data structure for point-based molecular data that is efficient for neighborhood queries, and flexible in rendering the data.

1.3 Contribution

We propose a hierarchical multi-resolution structure based on an octree, and levels of detail. The data structure, including levels of detail, is built in a pre-processing step once, and then stored for later use. All levels are therefore available at runtime, allowing smooth interpolation between resolutions. Many of the octree-based solutions we came across in our research divide the data set in a regular way and use the internal nodes of the octrees to store different levels of detail. In contrast to this approach, we divide the data set spatially on the original level of detail. The spatial extent of an individual block or node in the tree is determined by the number of data points within it. A block in a sparse region can thus have a larger spatial extent or bounding box than one in a dense region. We store the data for all levels of detail in the leaf nodes. This approach combines many of the advantages of regular grids, such as the possibility to "pick and mix" different resolutions for individual blocks, with advantages of hierarchical spatial division, for example greater control over the amount of data stored in each leaf. We demonstrate the use of our modular structure for optimization strategies including coarse visibility culling, mixed levels of detail, depth-dependent level of detail, and data management via a least recently used (LRU) cache. As our basic surface rendering model, we provide an implementation of the Gaussian Surface Model as proposed by Bruckner [Bru19]. In addition, we include enhancement options such as depth of field and ambient occlusion. The project is implemented in C++ and OpenGL.

1.4 Structure

After analyzing the current state of the art in Chapter 2, we first introduce our data structure (Chapter 3), and then go on to rendering aspects of our implementation (Chapter 4). In the chapter concerning the data structure, we focus mainly on the CPU side of the implementation, explaining the division of data into an octree, and the clustering-based level of detail calculation. The chapter on rendering includes both GPU and CPU components, including the LRU cache, level of detail blending, and screen-space

enhancement and surface methods. In a separate implementation chapter (Chapter 5), we provide additional information about libraries, choice of important data types, and other factors that concern the concrete implementation in C++/OpenGL, rather than algorithms and concepts. Finally, we provide numerical and visual results (Chapter 6), and discuss applications and limitations (Chapter 7)

CHAPTER 2

State of the Art

In our work, we aim to render large sets of biomolecular data at interactive rates, and build a framework that allows the efficient calculation of neighborhood-based algorithms. Rendering biomolecular data has a long history in visualization and computer graphics. Individual atoms are usually defined by their position and rendered as spheres, so they can be considered point-based data. Similar data sets based on points are also used for other visualization tasks, for example fluid simulations. The individual data points are commonly called particles, especially when they are the result of a simulation. We start this chapter by giving an overview over previous work in the field of biomolecular visualization. In the following section, we take a closer look at point-based rendering methods. As our goal is to enable real-time rendering of large data sets, we look into previous work specifically targeting large point-based data. Here, we pay particular attention to the proposed data structures and level of detail methods. In the final part of this chapter, we cover research on the representation of biomolecular data, as well as enhancement methods for point-based data.

2.1 Frameworks and Systems for the Visualization of Biomolecular Data

The data we visualize is biomolecular data consisting of up to a couple of millions of atoms. Our aim is interaction in real-time, and in our current implementation, we focus on a single time-step. In this section, we look at implementations that propose solutions to very similar challenges. In a recent state of the art report, Kozlíková et al. [KKF+17] give an extensive overview over developments and results in the field of visualization of biomolecular structures.

Many researchers publish the results of their work in the form of toolkits. It is outside the scope of this thesis to compile a comprehensive list, so we only mention some of the

most well-known and relevant available tools. The following papers give a more detailed overview over toolkits: Chavent et al. [CLK+11] focus on tools and implementations that use the GPU to accelerate the visualization of large molecular data sets. Johnson and Hertig [JH14] provide a well-structured overview over tools and methods for the visual analysis and communication of biomolecular data from a user's point of view. Dubbeldam et al. [DVVC19] present a survey of existing (bio-)chemical tools and visualization software.

An early example of a toolkit for molecular visualization is RASMOL, a molecular visualization tool presented by Sayle and Milner-White. [SMW95]. One of the most frequently cited tools is VMD by Humphrey et al. [HDS96]. It is mainly used for molecular dynamics simulations. Both Chimera by Pettersen et al. [PGH+04] and PyMOL by Schrödinger et al. [Sch15] are popular extendable frameworks for molecular data mostly written in Python. Herraez [Her06] proposes JMol as a tool to investigate biomolecular data, but also emphasizes possibilities to integrate it into educational web pages. Paraview by Ahrens et al. [AGL05] is frequently used for large molecular data sets, though it is a visualization tool made to handle large point-based volume data sets generally. Megamol by Grottel et al. [GKM+14] is another framework with the capability to render different kinds of point-based data, though its focus is on molecular data. They use a combination of GPU rasterization, ray-casting of glyphs, and image-space filtering to interactively render millions of atoms.

Though we focus on a single time-step in our implementation, molecular dynamics simulation implementations often apply similar optimization strategies to ours, as they usually have to handle very large data sets. The main difference is that the trade-off for computationally expensive pre-processing is more problematic when rendering animations. Hao et al. [HVS04] for example choose to use a simpler data structure, as they aim to render multiple time steps without a significant amount of pre-processing. Instead, they focus their efforts on occlusion culling schemes and levels of detail. Max [Max04] renders large molecular data sets at interactive rates. He also focuses on a single time step, and achieves interactivity by using a hierarchical level of detail data structure. He uses individual atoms as leaves in their tree. In contrast, we use leaf nodes to group the data into neighborhood clusters. Sigg et al. [SWBG06] use GPU splatting of molecular data. Their implementation is somewhat similar to ours, as they use the same kind of data and visualize it based on vdW surfaces. However, they do not use any advanced data structure and only render data sets up to several thousands of atoms in real-time.

Sharma et al. [SKNV04] present an approach to interactively visualize large atomistic data sets. They use a hierarchical view frustum-culling algorithm based on an octree data structure to remove atoms outside the field-of-view. Then, they select atoms which have a high probability of being visible. The selected atoms are rendered as spheres at various LODs, with a resolution derived from an exponential function of the distance from the viewer. They do not include any advanced surface visualization techniques in their viewer. Their octree consists of three levels. Each octree node contains the coordinate bounds of its subspace. A node at level 2 contains a pointer to a structure

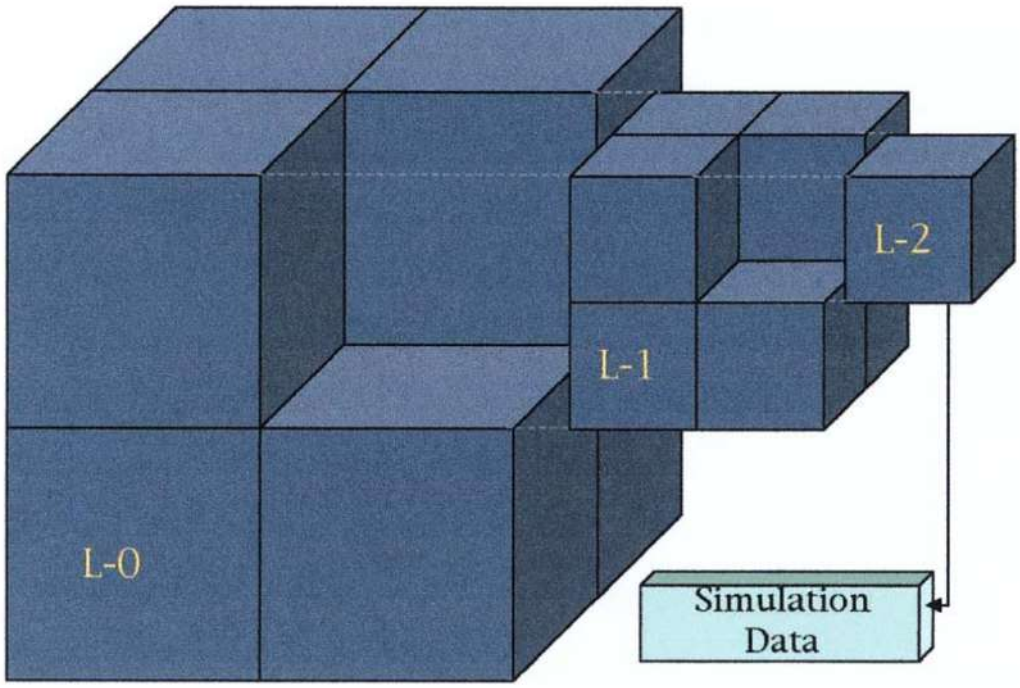


Figure 2.1: Octree data structure by Sharma et al. [SKNV04] that divides the data set into three levels of detail, as illustrated in their publication

that stores the atom data within its bounds, as illustrated in Figure 2.1. This is a similar approach to ours, though we do not use a fixed number of levels. Instead, we subdivide each region until no node contains no more than the given maximum amount of atoms.

Grottel et al. [GRDE10] present a visualization framework for molecular dynamics simulations. They choose a simple data structure and rather focus on optimization strategies. They employ several different methods: data quantization, data caching in video memory, and a two-level occlusion culling strategy involving hardware occlusion queries and maximum depth mipmaps. As molecular dynamics usually implies time-dependent data sets, a particular aim is to alleviate the data transfer bottleneck. Lindow et al. [LBH12] render large molecular data sets by making use of repetitive components found in molecular data. According to the authors, their work is closely related to that of Sharma et al. [SKNV04], and Grottel et al. [GRDE10]. They store atoms of individual components in a grid. The grid is then stored in a three-dimensional texture. This texture is then rendered using ray-casting in an approach similar to methods proposed for volume rendering. Materials are rendered at atomic scale, bridging five orders of magnitude in scale from the smallest details to the overall structure of the data sets. They exploit the fact that biological structures consist of many recurring molecular substructures. Each recurring component is represented by a single grid. All instances of a component are rendered by applying different transformations to that grid. The

authors use a ray-casting algorithm similar to classical voxel rendering methods which utilizes the bounding box of the data. Ray-casting is implemented in the fragment shader. Occlusion culling is accomplished implicitly on the atomic level, based on the underlying grid. Additionally, they use deferred shading to emphasize the global shape of biological structures. Falk et al. [FKE13], who extend the algorithm proposed by Lindow et al. [LBH12], use the same kind of data sets, which consist of many instances of only a couple of different molecules. The biggest change they propose is hierarchical ray-casting. When a traversed grid cell only covers one pixel, the algorithm does not render the data, but only determines whether the cell is empty or not by making a texture lookup. The same principle is used on entire molecules in the scene when their bounding box only covers one pixel. Le Muzic et al. [LMPSV14] optimize this approach by using a LOD scheme and by building visualization elements on the fly rather than using a grid, which allows them to alter atom positions dynamically. They store atom positions in a texture buffer, and use tessellation and geometry shaders to emit atoms, trying to maximize the amount of atoms emitted per vertex call. In addition, they also make use of repetitive structures. Parulek et al. [PJR+14] focus on supporting computationally expensive surface abstractions in real-time for large individual molecules. They achieve that by building a hierarchy based on spatial clustering in a bottom up approach. Clusters are formed based on their location, and each cluster is represented by a sphere with a radius that bounds all particles inside the cluster. For the next level, the error threshold is raised, and the new spheres are used as input for the clustering algorithm. The process is stopped, when only a single cluster remains. Then, they combine that hierarchy with surface abstractions at different levels of computational complexity to build a seamless model. Guo et al. [GNL+15] implement a view-dependent macro-molecule rendering framework, which includes levels of detail. They build their hierarchical model using a spatial clustering method similar to the one by Parulek et al. [PJR+14] and cluster atoms until only a single one is left. In order to find the most suitable distance metric, they compare their object space error as shown in Figure 2.2. LOD selection and view frustum culling are performed on the CPU. Like Lindow et al. [LBH12] and Falk et al. [FKE13], they use repetitive structures found in macro-molecules to optimize performance. For their GPU ray-casting, they use the method proposed by Grottel et al. [GKM+14].

The grid-based approach by Matthews et al. [MEK+17] is optimized for visual enhancement effects such as shadows and ambient occlusion. As their data structure needs to be recomputed every time the scene changes in order to ensure the quality of visual effects, they build it on the GPU. They present the results of their method with molecules of up to about 300.000 atoms. It is important to note that quantitative results of frameworks aimed at optimizing computationally expensive visual effects cannot be directly compared to implementations that focus on utilizing repetitive structures or on rendering a maximum amount of simple spheres at interactive rates. Similarly, the difference between rendering individual molecules and repetitive molecular structures should be noted.

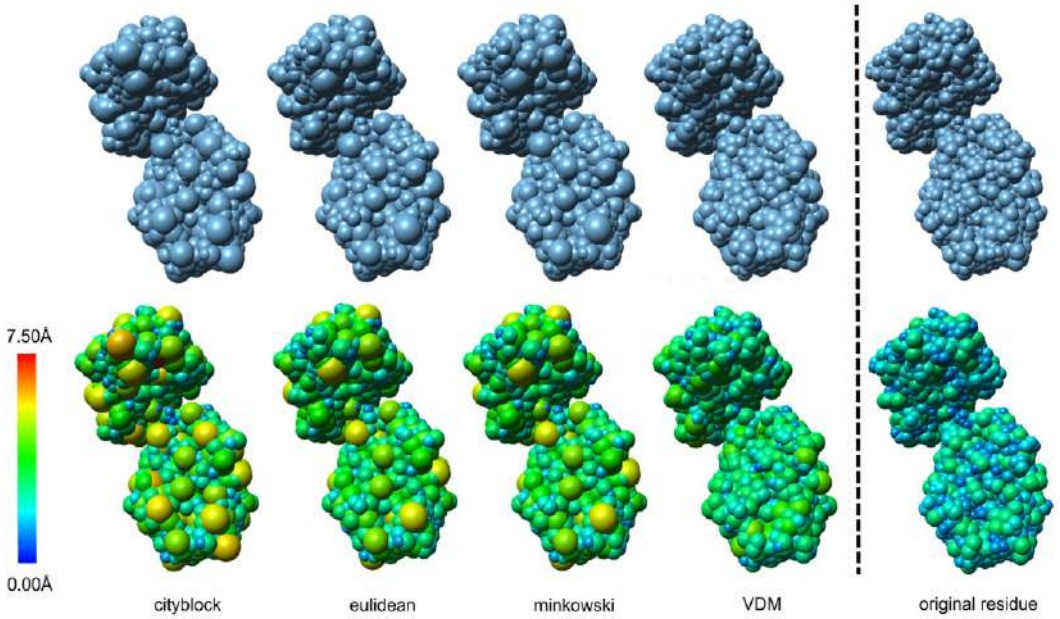


Figure 2.2: Comparisons of the object space error of different distance metrics by Guo et al. [GNL+15]

2.2 Rendering of Point-based Data

Classical representations of molecular data are based on spheres. Early examples that are still in use include van der Waals representations, as proposed by van der Waals [VdW73], and the ball-and-stick representations, first used by Hoffmann [Hof65] in lectures at the Royal Institution of Great Britain. Though proposed approaches vary, they usually fall into one of three categories. The data is either polygonized using algorithms like Marching Cubes (Lorensen and Cline [LC87]), sampled into a density volume representation, or spheres are rendered directly using splatting. A splat or glyph is a two-dimensional graphical element that is projected onto the 2D viewing plane for each voxel or point. According to Ibrahim et al. [IWR+17], ray-casting using per-fragment rays based on splats for each particle, has been accepted as the state of the art, at least in molecular dynamics visualization. However, different rendering techniques, acceleration strategies, and data structures have always been developed in parallel. In a recent report, Reina et al. [RGE19] cover the development of point-based visualization in the last decade.

Polygonal meshes generally yield satisfying visual results, but the approach creates a large number of triangles, and does not scale well beyond a couple of thousand particles. Müller et al. [MCG03] implement both point splatting, and Marching Cubes-based surface reconstruction for SPH simulations. They find the results achieved by iso-surface triangulation via marching cubes more visually convincing. It does, however, slow down rendering, which is not desirable in real-time rendering of large data sets. Hao et

al. [HVS04] use occlusion culling to optimize rendering of time-varying molecules. They build per-frame occlusion maps on-the-fly. Atoms from the sorted list of potential atoms for each frame are projected onto the image plane in a front-to-back order. Instead of checking for overlap of the circles representing atoms, they use two nested squares, one that completely fits within the circle, which is used to build the occlusion map, and one that covers the entire circle to check if the atom has been blocked by previously rendered atoms. They also include a method that chooses an appropriate tessellation resolution according to the amount of space the atom takes up in the final rendering. Keiser et al. [KAD+06] propose an alternative meshing strategy using 3D Delaunay triangulation of particles. Both methods improve performance, but even the reduced mesh is still computationally expensive to render, compared with other representations. Research in recent years has mainly focused on very large and dynamic data sets, so tessellation has mostly been replaced by other methods.

Sampling particles into a volume allows the use of GPU-accelerated volume rendering, using optimizations such as early ray termination and empty space skipping as proposed by Krüger and Westermann [KW03]. Most methods sample point-based data on some type of regular grid. Quiao et al. [QEE+05] visualize point-based atomic simulation data by sampling to a Cartesian grid, taking advantage of graphics texture memory. Navrátil et al. [NJB07] use a regular grid, extracting iso-surfaces using Marching Cubes, which they then render using the toolkit ParaView (Ahrens et al. [AGL05]). Their method allows them to visualize several hundred time-steps containing two million particles. Fraedrich et al. [FAW10] sample particle data inside the view volume into a perspective grid. They compare order-dependent splatting to volume ray-casting, as well as a hybrid approach. In their hybrid method, they add iso-surfaces for fine details which, according to their research, cannot be rendered accurately using splatting. Lindow et al. [LBH12] render large molecular data sets by making use of repetitive components found in molecular data. They store atoms of individual components in a grid, which is then stored in a three-dimensional texture. The grid is rendered using a ray-casting method based on the work of Hadwiger et al. [HSS+05]. Falk et al. [FKE13] extend this algorithm by hierarchical ray-casting, which omits grid cells that cover only part of a pixel. Reichl et al. [RTW13] visualize large SPH simulations using a compressed octree grid, streaming data asynchronously to the GPU. They perform decompression on the GPU, which allows them to integrate it into GPU-based out-of-core volume ray-casting. Knoll et al. [KWN+14] propose a multi-core CPU method for direct volume rendering of particle data using radial basis function kernels. They apply it to both astrophysical and molecular data sets. Schatz et al. [SMK+16] propose a hybrid multi-scale rendering architecture for very large data sets. Large parts of the data set are rendered as a hierarchical density volume, while fine details are visualized using direct particle rendering.

Splatting was one of the first methods used for point-based data, and is still relevant today. Most of the earliest works, for example Csuri et al. [CHP+79] and Reeves [Ree83], use a form of particle splatting to implement special effects, such as smoke. Westover [Wes89] introduces an interactive volume rendering method based on splatting. Laur

and Hanrahan [LH91] propose hierarchical splatting, also for volumetric data sets. They combine the splatting method with a progressive refinement algorithm, based on a pyramidical volume representation. Rusinkiewicz and Levoy [RL00] use hierarchical splatting as a point rendering method for meshes. They construct a hierarchy of bounding spheres based on a k-d tree to represent the surface. Extending this approach, Max [Max04] proposes a method for molecular data. Instead of an arbitrary hierarchy, the natural structure of molecular data is used. In order to better approximate the shapes of objects, the paper proposes the use of ellipsoids instead of spheres. Hopf and Ertl [HE03] extend the method to Smoothed Particle Hydrodynamics (SPH) simulations. They build a hierarchical data structure using PCA clustering.

Reina and Ertl [RE05] build a molecular dynamics visualization framework based on the work of Hopf and Ertl [HE03]. They generate implicit surfaces to render splats directly on the GPU, thus reducing the bottleneck between CPU and GPU. Toledo and Levy [TL04] propose ray-tracing based GPU implementations of new graphics primitives like spheres, cylinders, and ellipsoids. Intersection and lighting are computed in a fragment shader based on an analytic representation of the primitives. Bajaj et al. [BDST04] also render primitives directly on the GPU. For spheres, they use the texture-based method described in one of NVIDIA's Cg Tutorials by Fernando and Kilgard [FK03]. Sigg et al. [SWBG06] propose an algorithm for GPU-accelerated rendering of quadratic surfaces and apply it to molecular data. Each quadratic primitive is represented and rendered as a single vertex. That allows them to compute a screen-space bounding box in a vertex shader with perspective correctness. In our work, we use the same basic approach, rendering one vertex per atom. In order to draw the quad representing the atom correctly, we use the bounding box of the atom defined by its radius and the position of its center and compute the ray-sphere intersection analytically.

Later approaches focus on optimizing splatting methods for even larger data sets. Gribble et al. [GIK+07] propose interactive ray tracing for glyph-based representations on multilevel grids to visualize large time-varying data sets. Their approach is based on the coherent grid traversal algorithm proposed by Wald et al. [WIK+06]. Fraedrich et al. [FSW09] sort SPH data into a page-tree and render it out-of-core. Pages are blocks of data, usually grouped by spatial proximity. To render them, Fraedrich et al. rasterize particle splats, creating a proxy geometry based on equilateral triangles. They work with data sets containing more than 10 billion particles. Falk et al. [FGE10] propose a combination of splatting, texture slicing and ray-casting to render large dynamic particle simulations. Grottel et al. [GRDE10] also base their visualization on glyphs. They propose to use splats and a deferred shading pass that estimates their normal vectors in image space instead of ray-casting individual glyphs. Another paper by Grottel et al. [GKSE12] proposes an ambient occlusion algorithm for molecular dynamics based on density information collected in real-time. They build a density volume by splatting a sphere for each particle into the corresponding volume cell. The density volume is then used to create the ambient occlusion effect in object-space. In their framework MegaMol, Grottel et al. [GKM+14] render spherical glyphs using point-based GPU ray-casting in

their basic rendering mode.

Le Muzic et al. [LMPSV14] save all the required atom positions in a texture buffer. Each atom is represented by a single vertex, the center of the atom. They are rendered as spheres using splatting in the fragment shader. In order to test their framework, they created a scene containing 30 billion atoms, which they are able to render at 10 fps using levels of detail. Wald et al. [WKJ+15] implement a balanced k-d tree data structure for large point-based data sets. They use CPU ray-tracing to render the primitives encoded in the tree. Zirr and Dachsbacher [ZD17] propose a GPU based voxelization technique for particle sets such as SPH data. They render splats on a perspective grid and then compress the obtained information by only storing the entry and exit surface depths for each ray. In order to accelerate ray-casting, they use a screen-aligned implicit quadtree with different levels corresponding to perspective grids with decreasing resolution. They note that the approach can be distributed across multiple GPUs. Xiao et al. [XZY17] combine particle splatting, ray-casting and surface normal estimation techniques. Splatting is used to accelerate ray-casting while surface normals are estimated using using Principal Component Analysis. They also employ GPU-based ray-casting.

2.3 Methods for Handling Large Point-based Data Sets

Most publications that focus on very large particle data sets contain time-steps, such as molecular dynamics visualizations and SPH simulations. Cosmological data sets, such as those used by Schatz et al. [SMK+16] contain up to trillions of particles. Our current framework is aimed at molecular data sets containing millions of atoms.

An important question in handling large data sets is when to process data. Whether it should be pre-processed, streamed, or processed on demand, depends both on the type of data, and on the requirements of the framework. Our implementation relies on pre-processing to build the data structure. Pre-processing has to be done once for each new data structure. As we save the calculated structure to a binary file, it can subsequently be loaded instead of calculating it again. Contrary to this approach, on-demand methods only process the data the visualization requires whenever it is needed. This can slow down the application, but may also decrease the total amount of computations, and space necessary. Data requests can be visibility-driven or query-driven. Streaming is similar to on-demand visualization, except that the data is not demanded, but sent when it becomes available i.e., when calculations are finished. Already available parts of the data set can be rendered on-the-fly, without the delay of having to wait for the entire data set to be processed. This can speed up the creation of visualizations considerably. The disadvantage is that the user has less control over available data than in an on-demand based system.

Large data sets are commonly divided into smaller sub-sets. Several terms are used, but in the context of this work, we will refer to these sets as pages. Paging makes the amount of data easier to handle, both in the pre-processing and the rendering stage. It is especially important for extreme-scale data sets that are too large to fit into memory.

Data structures that divide large data sets into smaller sub-sets can be used to implement visibility culling. Blocks of data that are entirely invisible can be discarded early, saving computational cost. Sharma et al. [SKNV04] visualize data sets consisting of up to a billion atoms. They build an octree data structure, which they use for hierarchical view frustum culling. Zhu et al. [ZCW+04] propose a method for distributed rendering of large molecular data sets. Their approach uses a grid and focuses on partitioning the data set and distributing computations in order to handle large data sets. Gribble et al. [GIK+07] propose interactive ray-tracing of multilevel grids. They achieve interactive frame rates for data of up to 35 million particles. Freaedrich et al. [FSW09] traverse their hierarchical data structure every frame in order to determine the nodes that have to be rendered. Their test data set contains 10 billion particles, and they manage to achieve interactive frame rates. Grottel et al. [GRDE10] use a two-level occlusion culling strategy for large molecular dynamics visualization. They first perform per-cell culling to reduce the upload to the GPU. After that, a maximum depth mipmap is used to cull individual glyphs in the geometry processing stage. We sort our data into an octree data structure, and employ a page-based culling strategy where we discard pages, when their bounding box is entirely outside the view frustum.

2.3.1 Common Data Structures

Despite the continuous growth of hardware capacities, data sets can become too large to fit into GPU memory, and for extreme-scale data even CPU memory. In that case, the data needs to be broken up into smaller parts. When using paging strategies to render data sets that are too large to fit into memory, parts of the data are resident out-of-core, and have to be fetched before rendering. Usually, the currently required data is requested. A working set that fits into memory is then composed and uploaded or updated as needed. Many processing methods require information from the neighbors of individual points. Going through all data points in a set is computationally expensive, especially for very large data sets. Reducing the number of points that have to be considered in each step can speed up rendering considerably. Samet [Sam90] gives a comprehensive overview over different data structures for point-based data.

Which data structure is used depends on the type of data, as well as the priorities of the researchers. The most common types of data structures for point-based data are regular grids, and different variations of trees. Some researchers build structures tailored to a specific data type. Bajaj et al. [BDST04] for example do not use a conventional data structure, but sort their data into a structure they call Flexible Chain Complex, which is based on the hierarchy in Protein Data Bank (PDB) files. In this section, we focus on the two most common types, i.e., grid and tree data structures. We start with some examples of grid-based data structures, but focus in particular on implementations based on octrees, as we use such a data structure in our own work.

Harada et al. [HKK07] dynamically construct a sliced data structure for particle-based fluid simulations. They divide the voxel grid into slices perpendicular to one axis, and define bounding boxes for each slice. Memory for the grid is dynamically allocated,

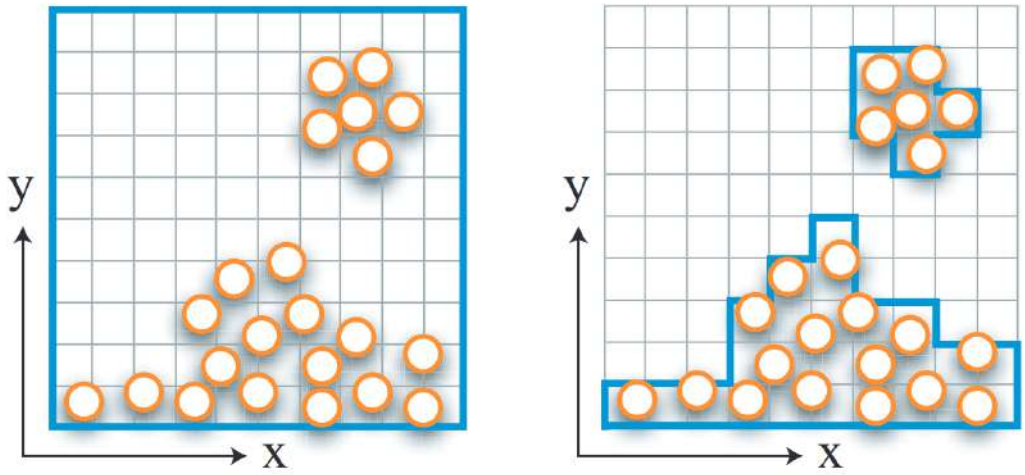


Figure 2.3: Data slices shown in Harada et al. [HKK07]. They compare a fixed grid data structure (left) with the dynamic grid they propose (right)

.

achieving improved cache efficiency, as it culls most of the empty voxels. Figure 2.3 shows a comparison between a fixed and a dynamically allocated grid. In addition to presenting the conceptual data structure, they implement and test it both on the CPU and GPU.

Gribble et al. [GIK+07] use hierarchical multilevel regular grids for particle based volumetric data. As they want to optimise their method for time-varying data sets, grids are constructed on-the-fly. Grottl et al. [GRDE10] use ray-casting on a regular grid, which has the same extent as the bounding box of the data set. They employ GPU ray-casting and deferred shading with smooth normal vector generation. They argue that the simplicity of the data structure allows a higher number of occlusion queries. Fraedrich et al. [FAW10] sample particles onto a uniform 3D grid. Instead of the simulation domain, the grid is fixed to the view volume. Therefore, only the view frustum is discretized. Spacing between grid vertices along the view ray increases logarithmically, resulting in a decreasing sampling rate along the viewing direction. Another implementation that uses a regular grid is presented by Matthews et al. [MEK+17]. They investigate both a fixed grid and a compact grid for visualization of molecular trajectories. Due to more efficient memory access, they conclude that the compact grid outperforms the fixed grid for large proteins.

Rusinkiewicz and Levoy [RL00] implement a hierarchy of bounding spheres based on a k-d tree to represent the surface of a solid object. They use it for view frustum culling, back-face culling, a level of detail scheme, and rendering. Pfister et al. [PZVBG00] sample geometric models into point primitives called Layered Depth Cubes (LDC). These primitives are then stored in an octree data structure. The lowest level of the octree

contains the resolution acquired during sampling. Blocks on higher levels are constructed by sub-sampling by a factor of two. During rendering, the octree is traversed from lowest to highest resolution blocks. View-frustum culling is performed using the block's bounding box. Hopf and Ertl [HE03] argue that hierarchical data structures impose higher memory requirements, so they decouple the cluster hierarchy from the actual point data. They store the information about points in a continuous array, while the hierarchical data structure only stores pointers. In order to reduce memory requirements, they store point coordinates relative to cluster coordinates. Figure 2.4 conceptually shows the hierarchy levels. The finest level, shown on the right side, contains the actual point information.

Losasso et al. [LGF04] propose an unrestricted octree grid to refine and coarsen particle data sets to simulate water and smoke. Max [Max04] builds a tree hierarchy with individual atoms as leaves. Coarser levels of detail are represented as ellipsoidal objects, and are stored at the internal nodes of the tree. Sharma et al. [SKNV04] visualize data sets containing up to a billion atoms. Their octree consists of three levels. Nodes at level 2 contain pointers to the atoms stored in the corresponding subspace. Lee et al. [LPK06] use a bounding tree to achieve view dependent mesh simplification as well as view frustum culling. Raschdorf and Kolonko [RK09] compare different data structures for large particle simulation data sets. They present a hybrid data structure consisting of a loose octree combined with local neighborhood lists. Reichl et al. [RTW13] visualize large SPH simulations using a compressed octree grid from which they stream data asynchronously to the GPU. Wald et al. [WKJ+15] adapt the balanced k-d tree proposed by Bentley [Ben75], and use CPU ray-tracing to render it. Rather than using the tree to store different levels of detail, they use it to reorder the original data set.

We implement an octree data structure in in our framework. The resolution of the tree is determined by a specified maximum number of points per page. Contrary to many implementations, data, including level of detail resolutions, is only stored in leaf nodes.

2.3.2 Levels of Detail

Visualizing several million atoms individually can become quite computationally expensive, even when rendering simple surface models. However, display resolutions limit the amount of information that can be shown at once, so optimization is possible. If the view is "zoomed in", which means that atoms are visible in detail, large parts of the data set are outside the view frustum, and can be culled. In a "zoomed out" view, less or no data may be culled, but many individual atoms are clustered in a space that is rendered as less than a single pixel. That means that the data structure can be shown at a lower resolution, without great impact on the visual result. In such cases or when the only goal is to visualize the general shape of a molecule without need for details level of detail strategies can be used. Keiser et al. [KAD+06] argue that the resolution of the chosen level of detail should be high enough to show fine-scale surface detail, while reducing the number of particles as much as possible in order to minimize the computational overhead.

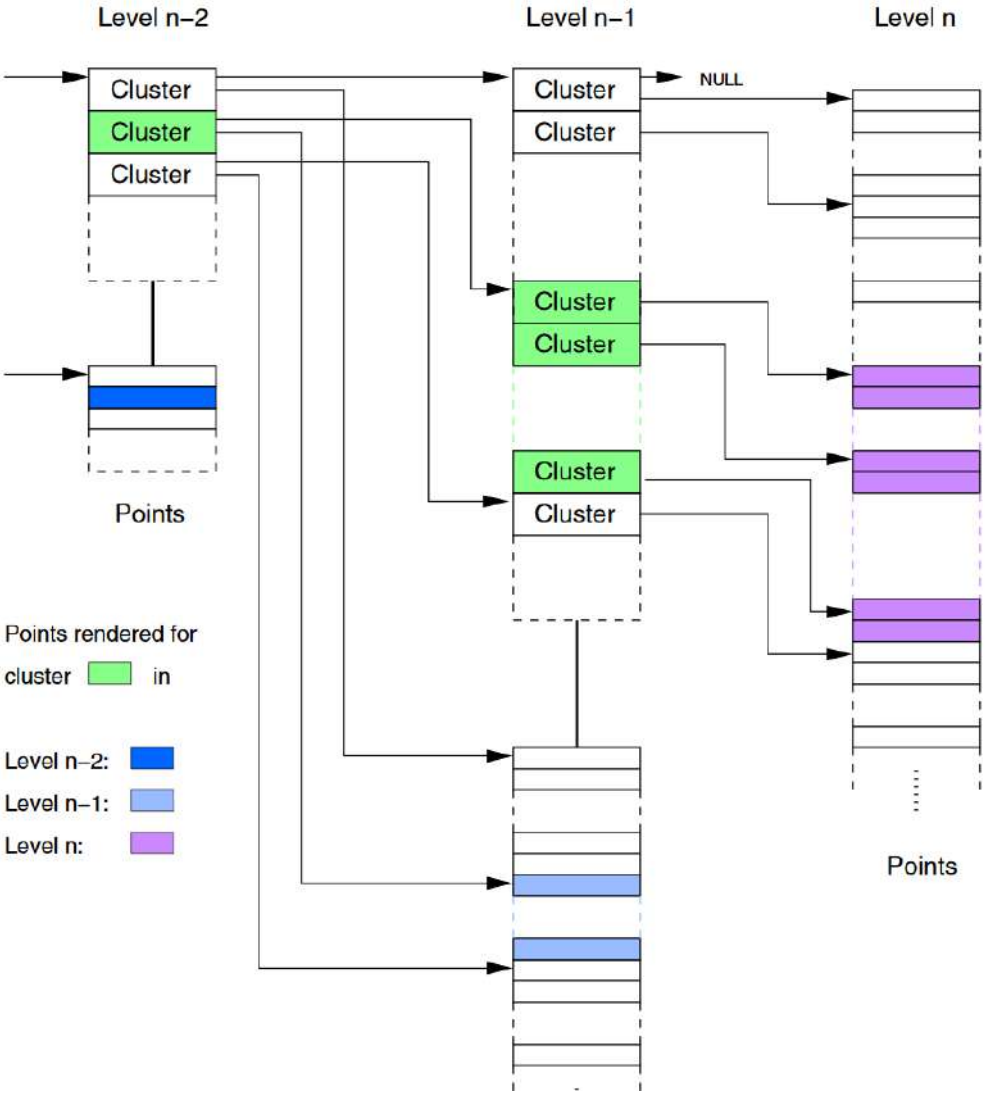


Figure 2.4: The hierarchy levels as implemented and illustrated by Hopf and Ertl [HE03]. The illustrated hierarchy is decoupled from the storage of the underlying point data. Points are stored in a continuous array, while the hierarchical data structure illustrated in the figure only stores pointers.

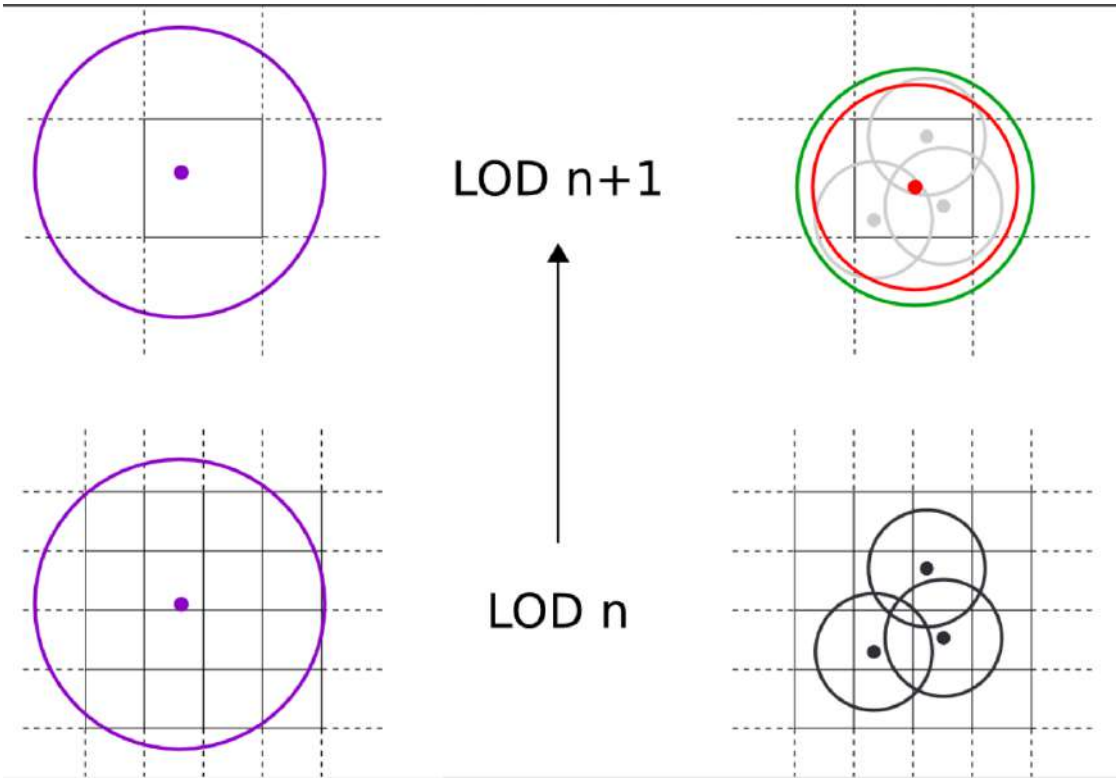


Figure 2.5: Level of detail construction as implemented by Fraedrich et al. [FAW10]. The data structure is constructed bottom-up. Particles are copied to the next level as long as their diameter is larger than the grid sampling resolution, those that are smaller are merged.

They implement a multi-resolution particle-based fluid simulation. Levels of detail are created by splitting and merging individual particles.

For general point-based data, levels of detail are usually created by clustering data points, based on the positions and in some cases radii of the points. Most level of detail structures are built bottom-up. Pfister et al. [PZVBG00] store the resolution acquired during sampling at the lowest level of the octree. Blocks on higher levels are constructed by sub-sampling by a factor of two. During rendering, the levels are traversed from lowest to highest resolution. Hopf and Ertl [HE03] create their LOD hierarchy using principal component analysis splits. The multilevel grid proposed by Gribble et al. [GIK+07] also constitutes a level of detail scheme. The original particles are stored at the finest level. The resolution of the grid is determined such that the number of cells is a multiple of the total number of particles. Coarser levels are imposed over the previous level, where each block corresponds to M $\times$ M $\times$ M blocks of the underlying level. In the implementation discussed in the publication, they use a two-level hierarchy. Fraedrich et al. [FAW10] also build their data structure bottom-up. As long as the diameter of a particle is larger

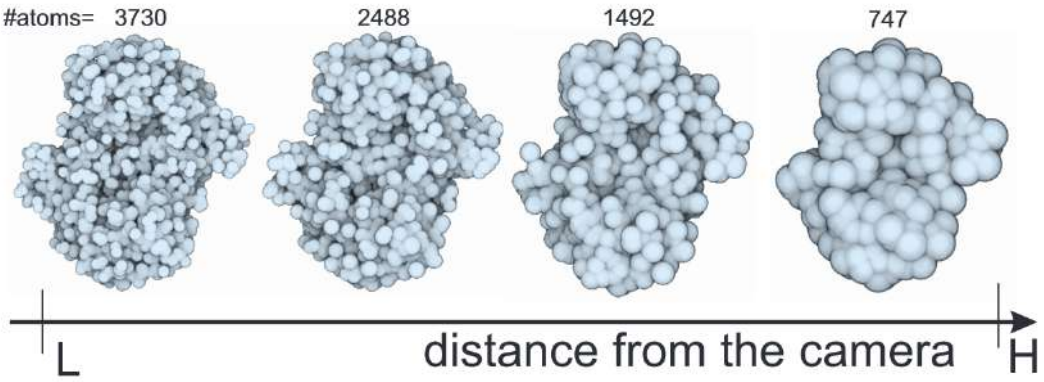


Figure 2.6: At increasing distance from the camera, Le Muzic et al. [LMPSV14] skip atoms, adapting the radius accordingly

than the grid sampling resolution, it is copied to the next level, as shown in Figure 2.5 on the left. Particles that have diameters smaller than that are merged and enlarged to the grid spacing. Reichl et al. [RTW13] build their level of detail hierarchy bottom up in a similar way. However, they store it in an adaptive octree data structure, rather than a hierarchical grid. Le Muzic et al. [LMPSV14] use tessellation shaders to lower the number of rendered atoms according to increasing camera distance. Figure 2.6 shows the same molecule rendered at different levels of detail. One of the goals is to implement smooth transitions between neighboring LODs. They sort the atoms by increasing distance from the center of the molecule's bounding box. Based on that distance, they decide how many atoms are skipped and increase the radius linearly.

For molecular visualization, it is possible to use the natural hierarchy of biomolecular data to implement levels of detail. This approach is proposed by several papers, such as the Flexible Chain Complex by Bajaj et al. [BDST04]. In their method, each level is associated with a geometric representation. Atoms are rendered as single spheres, and residues represented by a bounding sphere. Secondary structures are represented by sets of cylinders and helices. Van der Zwan et al. [VDZLBI11] implement continuous transitions between three separate visual abstraction models: the vdW surface, the so-called ball-and-stick model, and a cartoon rendering. Waltemate et al. [WSB14] also base their visualization on the natural characteristics of biomolecular data. They present an interactive tool, which makes it possible to combine the mesoscopic and molecular level in cell visualization. The result is a visualization of whole cells, with an interactive magnifier that allows the user to investigate the structure and behavior on a molecular level. The method they propose is based on instantly computed local parameterizations that are used to map patches of membrane structures onto regions selected by the user. They render the mesoscopic level based on triangle meshes, and the atomic level using a splatting technique based on local ray-casting of spheres. Parulek et al. [PJR+14] propose a level of detail concept that combines three different surface models in one

visualization. They blend SES, Gaussian kernels and van der Waals surfaces, based on linear interpolation, and implement a shading scheme that allows them to create seamless transitions between the representations. Guo et al. [GNL+15] use a volume based distance metric to select appropriate levels of detail depending on the viewpoint.

2.4 Biomolecular Representation Models and Enhancement Methods

Atoms in molecular data sets can be treated like any other type of point-based data. Depending on the complexity of the scene and capability of the hardware, simple point- or sphere-based representations may be the most suitable, or indeed only possible visualization model. In many cases, however, a lot of additional insight can be gained by using more complex models.

2.4.1 Representation Models

The van der Waals (vdW) surface is defined by the union of spheres proportional to the covalent radius of the individual atoms. It is simple and efficient, and therefore the most commonly used representation for atomic data. Additionally, it serves as a basis for many other surface models. Lee and Richards [LR71] propose Solvent Accessible Surfaces (SAS) as an extension of vdW surfaces. They illustrate which regions of a molecule can be accessed by a solvent. Conceptually, the surface is created at the position of the probe's center, while rolling over the surface, which is the same as extending the vdW radius of each atom by the radius of the probe. The disadvantage of the SAS is that it inflates the surface of the molecule. Richards [Ric77] addresses this issue with the introduction of Solvent Excluded Surfaces (SES). Instead of using the center of the solvent sphere, its boundary is used. This means that the volume of the resulting molecular surface is closer to the vdW surface, while still illustrating accessibility. Solvent Excluded Surfaces show the boundary of the volume with respect to a specific solvent. They are very useful in the exploration and development of substances, e.g., when developing and testing pharmaceuticals. Therefore, they are included in most molecular visualization toolkits. However, the calculation of an SES tends to be computationally expensive. It can either be computed by discretizing space, or by determining the implicit surface equations of its surface patches. Lindow et al. [LBPH10] implement a parallelized version of the contour-buildup algorithm originally proposed by Totrov and Abagyan [TA96]. Krone et al. [KGE11] also propose a parallelized optimization of the contour-buildup algorithm on the GPU. They subdivide calculation tasks in order to optimize them for parallel processing. Parulek and Viola [PV12] present a method that calculates the SES while allowing the user to change parameters interactively. They use local neighborhoods to compute implicit functions representing the surface. Hermosilla et al. [HKG+17] develop a grid-based GPU implementation using ray-marching, which allows smooth transitions between different levels of detail. As one of their main goals is to ensure real-time

interaction, they compute an SES using a low-resolution grid in real-time, and refine the surface progressively in the background.

Another surface representation is the Molecular Skin Surface (MSS), which applies the skin surface algorithm proposed by Edelsbrunner [Ede99] to vdW spheres. The MSS has the advantage of being C1-continuous (so the derivative does not change where two curved surfaces cross). The disadvantage on the other hand is that it has no biophysical background. While an SES approximates a solvent molecule with a sphere, Lindow et al. [LBH14] propose to use the ligand's actual vdW surface instead, showing the surface that a particular ligand can access in more detail. This so-called Ligand Excluded Surface is even more computationally expensive than the SES, so it is usually only applied when pre-computation is feasible. In interactive contexts, the SES or even simpler models are still preferred by most researchers.

Blinn [Bli82] proposes an approximation of molecular surfaces using a Gaussian convolution kernel, commonly known as Metaballs. Parulek and Brambilla [PB13] present a model that closely resembles the SES, while approaching the rendering performance of the Gaussian model. It is based on iterative blending of implicit functions, and is visualized using GPU-based ray-casting. Krone et al. [KSES12] compute interactive surface representations of large dynamic particle data sets. They build a volumetric density map based on Gaussian kernels, using a GPU-accelerated marching cubes algorithm. Structural details can be adjusted interactively. In our framework, we use the method presented by Bruckner [Bru19]. It is an image-space approach to the computation of Gaussian molecular surfaces. As the molecular surface only needs to be computed for visible parts of the molecule, unnecessary computations can be skipped. An on-the-fly list-based data structure is constructed for every frame. To create the list, the atoms in the molecule are rendered as spheres in two separate rendering passes. The first pass is used to identify all intersecting van der Waals spheres for every viewing ray. In the second pass, the sphere of influence for each atom is rendered in the same way. For each pixel, a linked list of the intersections of the spheres of influence is stored. Those are the only spheres that potentially contribute to the visible surface. In order to minimize computations in occluded regions further, visibility-based ray traversal is used.

2.4.2 Enhancement Methods for Point-based Data

Different shading methods can be used to enhance complex molecular visualizations. Color can be mapped either to individual chemical elements, or other properties, such as which amino acid chain it belongs to. In addition, effects such as depth of field, ambient occlusion and, edge-cueing can guide the viewer's eye to important areas of the visualization and combat visual clutter.

Ambient occlusion describes the amount of ambient light that reaches each point in a scene. Objects on the inside of structures, or that are occluded by direct neighbors are darker, so ambient occlusion helps the viewer to correctly judge the depth of structures. Tarini et al. [TCM06] propose a set of techniques to enhance the user's understanding

of rendered molecules. In order to compute ambient occlusion, they use several render passes, creating shadow-maps from the z-buffer. They achieve a contour line effect by drawing a line around each rendered primitive. The thickness of the lines can be rendered dependent on the difference in depth between the primitives it separates. Additionally, they propose to draw halos around atoms. The bigger the distance between a point in the halo and its background, the more opaque the algorithm draws it. Grottel et al. [GKSE12] implement an object-space ambient occlusion algorithm based on local neighborhood information. Their approach is based on the aggregation of particle data into a coarse resolution density volume, which is then used to calculate the ambient occlusion factor. Staib et al. [SGG15] propose an illumination model that supports transparency, ambient occlusion, surface reflection, and ambient illumination based on the emission-absorption model of volume rendering. It runs in real-time for millions of particles and is based on analytic solutions to the volume rendering integral. McGuire et al. [MOBH11] present a screen space ambient occlusion method. Rather than approximating the equations governing indirect illumination in an efficient but error-prone way, they use the full radiometric model until the screen-space sampling step. Simplification is still possible by using the fall-off function to cancel expensive operations. Skåneberg et al. [SVGR15] aim to communicate their spatial arrangement, but also to visualize pairwise atom interaction strengths. In order to achieve that, they propose an analytic approach for capturing ambient occlusion and interreflections of molecular structures in real-time. The neighborhood search required to calculate the influence of atoms, is based on a grid data structure. The cell size can be set and defines a cut-off distance for the interreflections. Matthews et al. [MEK+17] also use ambient occlusion to enhance depth, and shadows to help the user perceive relative motions of parts of the protein. Their algorithm is similar to that of Skåneberg et al. [SVGR15]. They use a regular grid to accelerate shadow casting, but find neighbors per atom rather than per fragment.

Another enhancement method widely used to draw attention to a certain region or feature, is depth of field. Depth of field is an optical effect known from photography, where objects on and around a focus plane are sharp, while the rest is blurred. The blurring factor is called circle of confusion. It can be calculated based on focal length, distance, and aperture. The closer the focused object is to the camera, the narrower the field of focus. Examples of this method can be found in several publications. One example is the method proposed by Falk et al. [FKE13]. They use an image space method to create the effect. They calculate the size of the circle of confusion, which is then used to sample a mipmap chain of the color and depth buffer. In our framework, we implement a depth of field effect based on the work presented by Bukowski et al. [BHOM13].

2.5 Summary and Conclusion

Both the field of biomolecular visualization and large point-based data sets have been researched widely and successfully. However, biomolecular visualization methods generally either focus on smaller data sets, or rely heavily on a priori structural knowledge about the data. Methods aimed at point-based data sets on the other hand can usually handle

extreme scale data efficiently, but do not provide any of the surface methods that are often expected in biomolecular research. We therefore conclude that there is a need for a framework that can handle large data sets without any particular knowledge about the structure of the data that nonetheless allows the use of surface models and enhancement methods. The most common data structures for handling large data sets appear to be regular grids and to a lesser extent various forms of trees, in particular octrees. Levels of detail for point-based data are most often implemented using a bottom up approach to spatial clustering. As a main surface model, biomolecular visualization methods generally use vdW surfaces and often additionally include the possibility of computing the SES. Popular enhancement methods for point-based data include in particular ambient occlusion and depth of field.

CHAPTER 3

Data Structure

The goal of our implementation is to render large sets of molecular data at interactive rates, in order to help users to better understand the data. As discussed in the previous chapter, there are several different approaches to similar problems. Our solution is a novel combination of several established approaches. We divide the description and discussion of our methodology into two parts. In this chapter, we focus on the underlying data structure, while rendering aspects of our implementation are covered in a separate chapter.

Large data sets require structure management, especially when the system is expected to perform at interactive rates. The goal is to reduce the size of the working set, i.e., the data that needs to be processed at a specific time, while keeping as much information available as necessary or possible. As it is unlikely that all details of a large data set need to be visible at the same time, it is possible to manage the data in such a way that only the parts that are currently required, are fetched and processed. Similarly, some parts of large data sets are usually in the far distance, which means that they only occupy a very small portion of any given screen, or they are at least partially covered. Therefore, rendering data at different resolutions can also be part of a solution.

Implementations that target extreme-scale data sets, such as Fraedrich et al. [FSW09], need to solve the problem that the data does not fit into the memory of current workstations. In order to make visualizations of the entire data sets feasible, they have to be divided in a suitable way. Many implementations provide level of detail schemes, which reduce the amount of data that needs to be processed for a given frame, without necessarily reducing the visual quality. We aim to optimize rendering for data sets of up to several millions of atoms, without requiring additional information about the data set. As we will see in Chapter 6, achievable frame rates are highly dependent on the amount of points or atoms that are actually within the view frustum for the current frame. Therefore, camera position and level of detail, on which the number of visible data points depends, have a greater influence on the performance than the number of

atoms in the original data set. This makes it somewhat difficult to directly compare numbers. Implementations that rely on repeating instances of molecules, such as for example Le Muzic et al. [LMPSV14], can render up to billions of atoms. Those that focus on graphical aspects, such as enhancement or surface models, like for example Matthews et al. [MEK+17] or Parulek et al. [PJR+14], do not go beyond a few hundred thousand atoms in their test sets. What we are investigating, is how to close this gap, and develop a data structure and rendering setup that makes it possible to render several tens of millions of atoms without prior structural information, using different molecular surface models, and applying enhancement effects.

As we want our data structure to be able to handle large data sets, we need to divide the data into manageable blocks. There are two main bottlenecks where very large data sets can become problematic. Processing or pre-processing steps that depend on neighborhood queries can easily get out of hand if the algorithm needs to visit each individual point for a large data set. Also, the amount of data that can be processed in the rendering step itself, is limited by hardware capacities. Very often, however, not the entire data set is visible at once. Therefore, the answer to both of these issues can be a spatial division of the data. As the density within molecular data sets varies spatially, we want our data structure to provide the option to control the maximum amount of data in each page (spatial block). We also want pages to contain approximately the same, or at least a similar amount of data. Our final choice of data structure is a density based octree with pre-calculated levels of detail.

3.1 Overview of the Components for CPU Data Handling

The main contribution of our work is a data structure that divides large data sets into manageable blocks, which facilitate efficient calculations of neighborhood-based algorithms. Our design is based on a core CPU-based data structure, as well as a set of other components that manage data and rendering. Figure 3.1 illustrates the main components that are involved in handling the data and constructing the structure.

The black arrows illustrate how components are related. Gray components above the dashed line are mainly concerned with rendering and front-end operations, which are covered in Chapter 4. In this chapter, we focus on components responsible for data management on the CPU, which are illustrated below the dashed line in Figure 3.1.The protein is part of the scene component, which is in turn part of the viewer component. At any given time, there is only one instance of each of these components. The viewer contains renderers. The renderer responsible for drawing the data points that represent atoms or clusters, is called sphere renderer. There can be several different types of renderers. Their display methods are called consecutively. We use a separate renderer to render bounding boxes.

Data from the .pdb file (Protein Data Bank, Berman et al. [BWF+00]) is read into our framework in the protein component. As we expect our implementation to be able to handle large data sets, we divide them spatially. The page component represents the

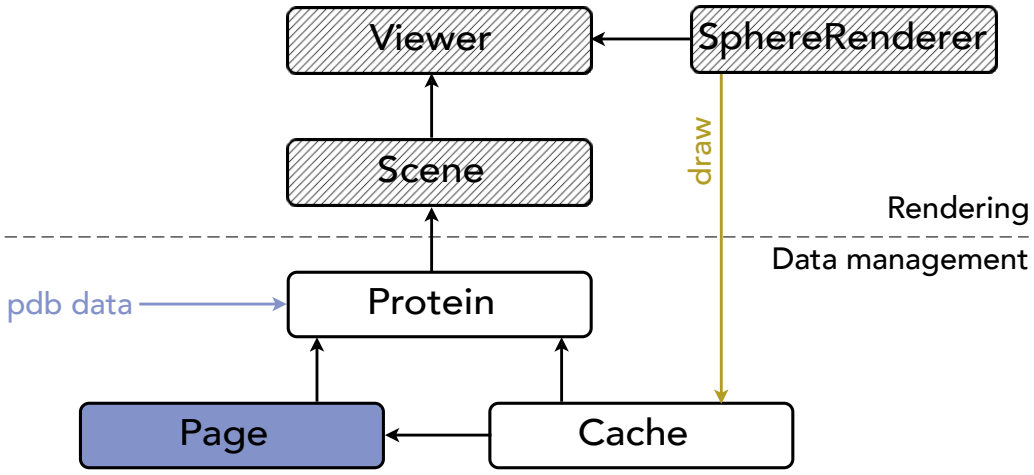


Figure 3.1: The main components involved in handling the data structure. Blue indicates where the actual (per point) data is located. The black arrows show which components contain each other.

building blocks of the resulting data structure. We store a single page, the root of the octree, in the protein component. As we want to control the maximum number of atoms on a single page, the depth of the octree depends on the density of the data structure. Each page that has more than a specified amount of atoms within its bounds is subdivided and stores pointers to its eight child nodes. The actual data points are stored in the pages that are leaf nodes. This allows us to mix pages of different spatial extent, but with a similar amount of data, which we take advantage of in our cache implementation.

The cache component manages which data is uploaded to the GPU. It implements a least recently used cache which is responsible for a list of visible pages and for replacing the ones that are no longer needed.

3.2 The Octree Data Structure

Our main requirements for a data structure are that it has to divide the data in a way that makes it possible to handle data efficiently, i.e., by reducing bottlenecks, as well as make computationally expensive neighborhood-based calculations feasible. We propose an octree-based data structure. It divides the data set spatially into hierarchically organized units, the pages. Within these pages, we save the data at different resolutions, or levels of detail. The data structure is built in a pre-processing step. In this section, we explain the design and implementation at the core of the data structure, the pages and the octree

they form, as well as the construction of the levels of detail.

Structuring available data into blocks that are manageable, both in terms of memory, and the computation of computationally expensive neighborhood queries, can be achieved in several ways. For a comprehensive overview of data structures for point-based data, we refer to Samet [Sam90]. The most common choices are regular grids, and hierarchical tree-based data structures. Several variations of both have been tested and proposed. A grid data structure is a simple and fast way to divide space. Regular and perspective grid solutions are very common in implementations that render their data by sampling it into a volume (for example Qiao et al. [QEE+05], Navratil et al. [NJB07], Knoll et al. [KWN+14], etc.). They are also used in some related implementations that render atoms as spheres directly on the GPU (Grottel et al. [GRDE10], Lindow et al. [LBH12], Falk et al. [FKE13],Matthews et al. [MEK+17]). The disadvantage of regular grids is that they allow little control over the amount of data in each block. It is of course possible to define the size of the cells by specifying a maximum number of points in each cell instead of spatial units. The drawback of adapting the cell size to the densest areas in the data set is that it results in an unnecessarily fine division of space in sparse regions, especially if data points are distributed unevenly. In our framework, we opt for a hierarchical data structure, as it provides greater control over the division of space.

Tree-based data structures are another common choice, especially for implementations that include level of detail schemes. Several variations of trees have been proposed in the context of point-based rendering. Though octrees are most common, k-d trees (Rusinkiewicz and Levoy [RL00], Wald et al. [WKJ+15]) are also used. The child nodes of an octree node divide the space into eight smaller cubes (four squares in the two-dimensional case). In regions with sparse data, lower subdivisions can be used, providing the ability to control resolution. K-d trees are also based on hierarchical space partitioning. While octrees are always partitioned cubically, the splitting planes in k-d trees can be placed arbitrarily in space. Space is divided by one axis-aligned split plane per level of depth, recursively subdividing space into a binary tree. Each node is split into two halves, alternating between axes. In order to achieve a balanced k-d tree, points in the data set are used to partition space. Geometric primitives are usually stored in leaf nodes. In k-d trees, it is not possible to control the number of neighboring cells that have to be examined when using nearest neighbor queries, as split planes can be placed anywhere in the node. Octrees, on the other hand, divide space in a regular manner. Each node is either a leaf, or is equally divided into eight sub-nodes. This makes the divisions more predictable, which makes the data structure easier to handle. It also makes octrees a good choice when a more even division of the data set in terms of the amount of points per page is a priority.

Most solutions that use octree data structures in combination with a level of detail scheme that we have come across in our research, use the nodes throughout the octree to store level of detail data. Often, the original data is stored in the leaves, and coarser levels of detail in internal nodes. This approach creates a structure which is very similar to regular grids with several different resolutions linked together. While this can be

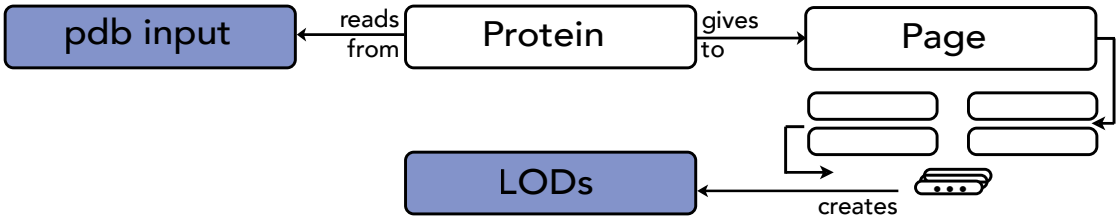


Figure 3.2: Flow of data while creating the data structure. Blue indicates where actual point data is saved. The protein component manages the data structure which consists of nested page components.

very efficient in many situations, the disadvantage is that pages or grid cells at different levels have different spatial extents, which makes mixing levels of detail more difficult. It also makes it harder to achieve a similar amount of data points per node. We use the hierarchical division of space somewhat differently. Instead of storing coarser levels of detail in the internal nodes, we store all levels of detail in the leaves. Space is divided depending on atom density in the original data. Internal nodes simply contain pointers to their children.

3.2.1 Dividing the Data Set into Pages

In a pre-processing step, the set of data points is processed, divided, and stored, building the data structure that is then used throughout interactive rendering. Figure 3.2 illustrates the flow of data through the components. The protein component reads in the original data. At this stage, the data for each point consists of a spatial position and a radius. The protein stores a single page, which serves as the basis for the octree, and which is filled with data and sub-pages in the pre-processing step. The page component is responsible for both subdividing data, and calculating clusters to build levels of detail.

We read atom positions, and information about molecules in the Protein Data Bank (PDB) file format. For each atom, we get the spatial position, and the element name from the file, which we then look up in a table in order to get the radius. This information is stored in a four-component vector.

When being read from the .pdb file, the atoms of such a single time-step are stored in one vector. This vector is then handed over to the basic page, the root of the octree, where the data is divided into sub-octrees, according to the specified maximum number of atoms per page.

The maximum number of atoms possible in our current setup is 32,768 (2^{15}). This is due to numerical limits which are explained in Section 3.2.3. We did not find the limit of 2^{15} atoms problematic, as the clustering algorithm we use to build our levels of detail is dependent on neighborhood comparisons and becomes very inefficient for larger pages.

Algorithm 3.1: Spatial data division. If the number of points in a page exceeds the maximum number of allowed points, it gets eight sub-pages and the data points are divided according to their spatial position. This is done until all pages fulfill the size limit.

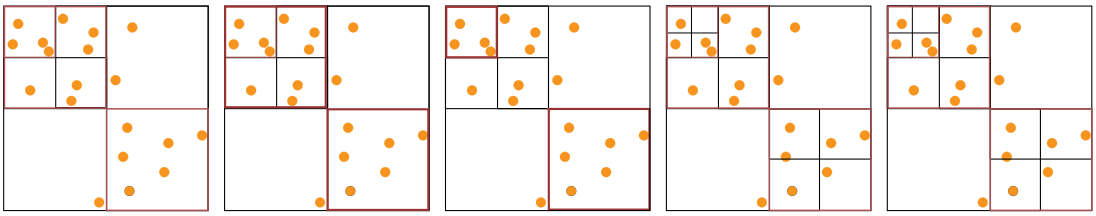
Input: Maximum number of atoms per page max, current page p containing vector of $atoms$, $lowerBound$ and $pageSize$

```
1  if size of atoms > max then
2  |  Create array of 8 subPages in p
   |  foreach a ∈ atoms do
3  |  |  pos = ((a.xyz − lowerBound)) / (pageSize / 2)
   |  |  insert a into vector of atoms in page at position pos in subPages
4  |  end
5  |  foreach page ∈ subPages do
6  |  |  run spatial data division algorithm
7  |  end
8  end
9  else
10 |  return atoms
11 end
```

The pseudo-code shown in Algorithm 3.1 sums up the division of space into pages. Starting with the basic page saved in the protein component, we check if the length of the vector of atoms assigned to it exceeds the maximum allowed number of atoms per page. If it does not, the algorithm ends. If it does, we create a sub-octree, i.e., a two by two by two array of sub-pages that have half the page size of the parent-page in each direction. Then, we iterate over all atoms, and sort them into the sub-pages according to the formula given in algorithm 3.1. For each sub-page, we recursively repeat the process. When the number of atoms assigned to a page no longer exceeds the limit, it is considered a leaf page, and the vector of atoms is saved, instead of creating sub-pages. Once the number of atoms on a page falls under the allowed maximum, we proceed to the creation of the levels of detail, as described in section 3.2.2.

Figure 3.3 shows a simplified illustration of the subdivision of a data set, with a limit of 3 atoms per page. The basic page contains more than 3 atoms, and is subdivided. After the first subdivision, two sub-pages still exceed the limit. Pages are subdivided depth first, so we check all children of the first of these two sub-pages recursively, before moving on to check its siblings. When no page contains more than 3 atoms, we have created the leaves of all branches and the final structure is reached.

The structure of a page can be summed up as follows. Each page saves its position, size, lower and upper bound. A leaf node additionally stores an array containing the atoms that fall within its bounds at different levels of detail. Inner nodes have pointers to their child nodes.



(a) Basic page (b) Subdivision 1 (c) Subdivision 2 (d) Subdivision 3 (e) Final structure

Figure 3.3: Illustration of page subdivision with max. 3 atoms per page

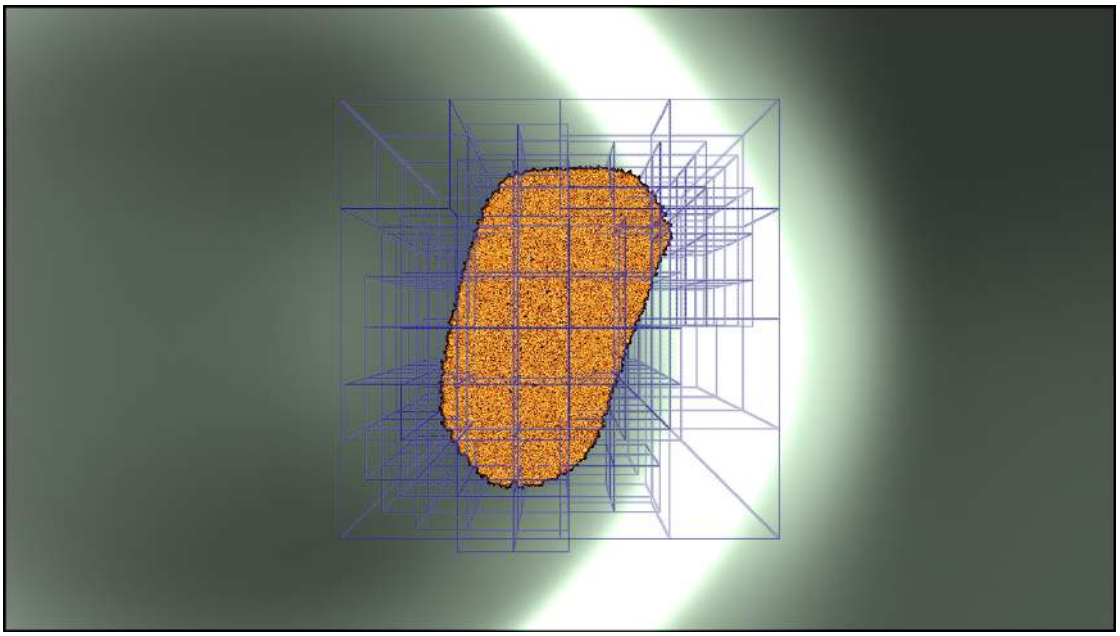


Figure 3.4: Octree structure for the molecule 3j3q. The blue lines show the bounding boxes of pages. In denser areas, more subdivisions are used.

Figure 3.4 shows the leaf nodes of the hierarchical data structure for the molecule 3j3q, which contains 2,440,800 atoms, making it the largest individual molecule currently available in the Protein Data Bank. The maximum number of atoms per page is set to 32,768, the largest number possible in our current setup. The figure shows the molecule at the original resolution. Additionally, the bounding boxes of the pages are illustrated in blue. In denser areas, the molecule is divided into smaller sub-pages while empty and sparse areas are not subdivided any further.

3.2.2 Calculating Levels of Detail

Levels of detail are a common method to increase flexibility, and improve performance. For point-based data, they are usually calculated bottom-up, using a variation of hierarchical

clustering. In order to calculate the clusters that form our coarser levels of detail, we use the hierarchical clustering implementation by Müllner [M+13].

Several researches (Max [Max04], Bajaj et al. [BDST04], Van der Zwan et al. [VDZLBI11], Waltemate et al. [WSB14]) propose using the natural hierarchy of molecules, rather than purely spatial approaches. This method provides the user with additional structural information, as well as a number of different levels of detail. However, the approach is not particularly flexible, as it relies on very specific information, and cannot easily provide intermediate levels of resolution. The main goal of our LOD implementation is to optimize performance, and enable us to render larger data sets, not necessarily the integration of additional structural information. That is why we chose the more flexible approach of hierarchical clustering. Structural levels of detail could be considered in addition to our existing LOD scheme in a future developing step.

Both our own tests, and the results achieved by Parulek et al. [PJR+14], show that density-based clustering does not perform well on the type of data we use. Atoms within most molecules do not show a significant variation in distribution. Therefore, density-based clustering often results in only a single cluster for a large part of the molecule. Hierarchical agglomerative clustering, such as the fastcluster algorithm proposed by Müllner [M+13] which we use, is more suitable. Guo et al. [GNL+15] argue that the algorithm has two limitations: according to their experiments, it is difficult to find the most suitable parameters, and it requires hierarchical abstraction. They use a volume-based distance metric in order to reduce the screen-space error. However, we found the results satisfying, and the control over parameters sufficient.

Hierarchical agglomerative methods cluster input data based on a dissimilarity index, starting with each point in a single cluster. Usually, the index is stored in a matrix, which is by definition reflexive and symmetric. Another possibility is the so-called stored data approach, where input points are handed to the clustering algorithm in vector form, and dissimilarity is specified implicitly. The algorithm by Müllner provides two options to define the size of clusters. The user can either specify the desired number of clusters, or a cut-off distance. Algorithm 3.2 shows a simplified pseudocode of the hierarchical clustering algorithm.

We calculate the dissimilarity matrix based on the Euclidean distance between two respective points. Additionally, the algorithm either needs to know a cut-off distance, or the number of desired clusters. In order to specify a reasonable number of desired clusters manually, one would need some information about the structure of the molecule. Specifying a cut-off distance is more flexible, as the user does not need any particular information beforehand. Therefore, we found it more suitable for our needs and a variety of molecules. In our implementation, we use a cut-off distance of $c = 2l$, where l is the level of detail being calculated.

Müllner [M+13] implements four modes of linkage for hierarchical clustering: Single-linkage (nearest neighbor), Complete, Average and Median. Figure 3.5 shows the results of the different distance calculation formulas for the molecule 2btv at level 1 and 5. The

Algorithm 3.2: Fastcluster by Müllner [M+13] based on a nearest neighbor approach. The algorithm needs a pair-wise dissimilarity (distance) index, as well as the number of points to be clustered, and returns a list of labels that divide the points into clusters.

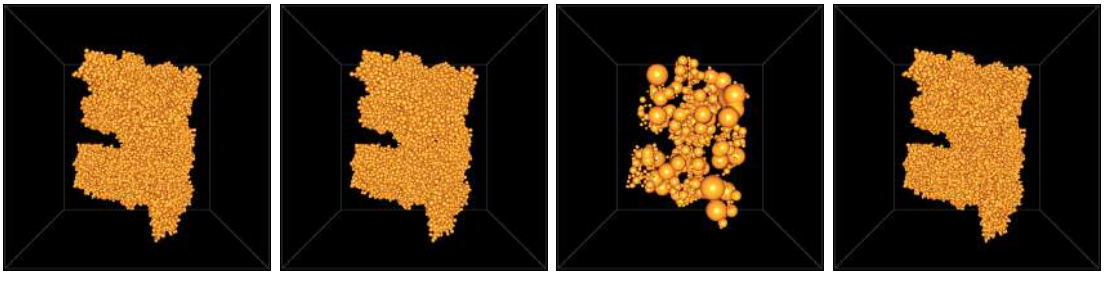
Input: A pair-wise distance index **D**, the number of points to be clustered C
Output: Output list L, number of points per cluster P

```
1  create list of nearest neighbors nn
   calculate priority queue of indices Q, mindist as keys
   for j ← 0 to N − 1 do
2      a ← min element of Q
       b ← nn[a]
       while dist(a, b) ≠ mindist(a) do
3          recalculate nearest neighbours
4      end
5      add (a, b) to L
       update C
       remove a from Q for remaining elements do
6          update distance matrix using formula 3.1
7      end
8      for remaining elements where x < a do
9          update nearest neighbor candidates
10     end
11     update mindist and Q with distance and nearest neighbor
12 end
13 return L;
```

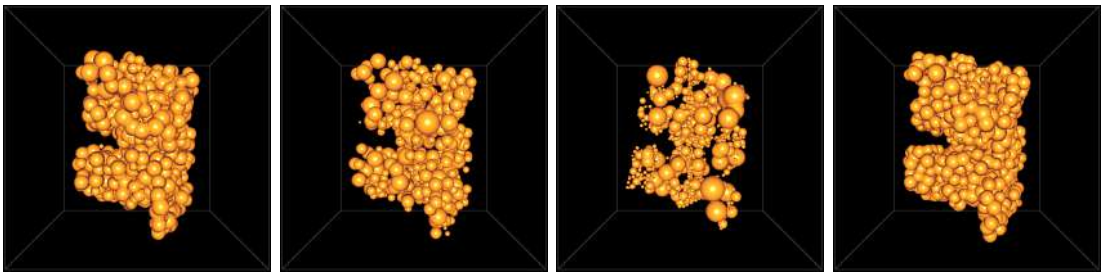
methods that approximate the original molecule best, are Average and Complete. Figure 3.6 shows the number of points the clustering formulas result in. The blue line represents the time it takes to calculate the entire level of detail structure using that metric. The larger the molecule, the more important this factor becomes. We choose the Average method, which achieves convincing results, and is more performant than the Complete method. The Average-linkage method calculates the inter-cluster distance by adding up the distance between all pairs of points in the cluster and then dividing by the number of pairs. It updates the distance with the formula given in Equation 3.1, where I and J are two clusters being joined into a new cluster, and K any other cluster. n_I and n_J are the sizes of the clusters. d is the dissimilarity measure, i.e., the Euclidean distance.

$$\frac{n_I d(I,K) + n_J d(J,K)}{n_I + n_J} \tag{3.1}$$

Algorithm 3.2 shows a pseudocode of the hierarchical clustering method. For a given set of nodes, a pair of closest points is determined. Those nodes n_1 and n_2 are then joined



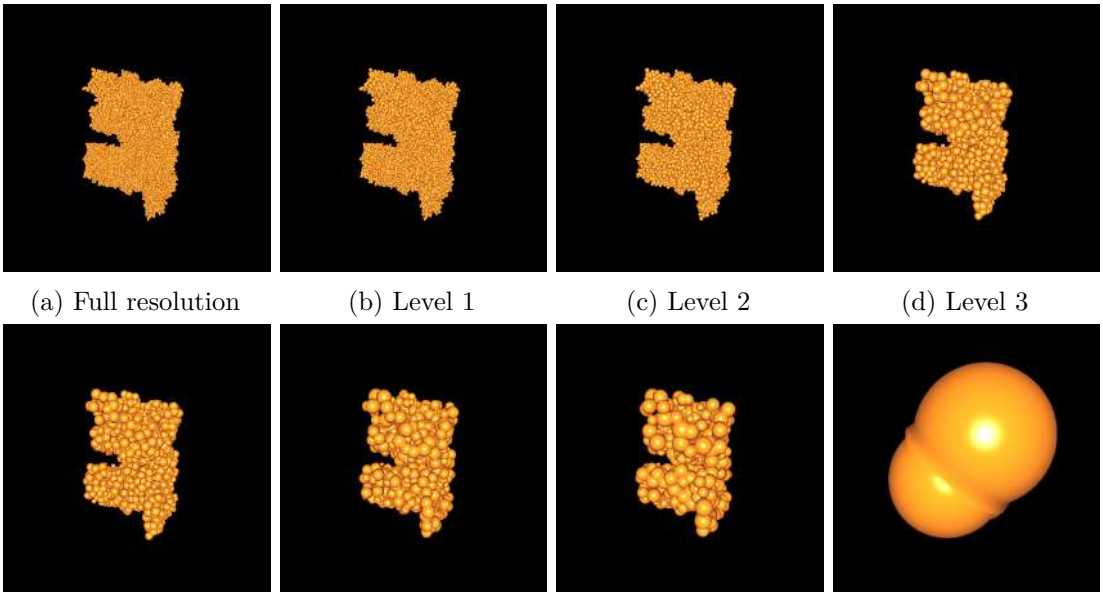
(a) Average level 1 (b) Median level 1 (c) Single level 1 (d) Complete level 1



(e) Average level 5 (f) Median level 5 (g) Single level 5 (h) Complete level 5

Figure 3.5: Levels of detail for the molecule 2btv

Figure 3.6: Number of atoms at levels 1 and 5 and building time, showing the number of resulting clusters for 2btv for level 1 (red) and 5 (green) as well ast the time it takes to build the data structure using that method (blue).



(a) Full resolution (b) Level 1 (c) Level 2 (d) Level 3

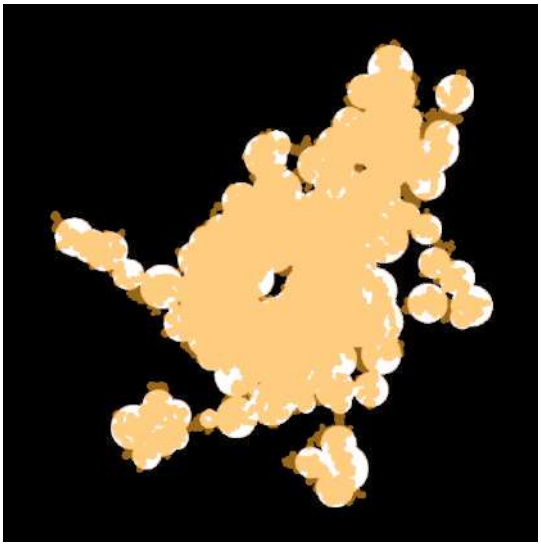
(e) Level 4 (f) Level 5 (g) Level 6 (h) One sphere per page

Figure 3.7: Levels of detail for the molecule 2btv. In the original resolution in the top left, the molecule contains 49,061 atoms, while level 7 on the lower right contains two data points, one per page.

into a new node, which replaces the two original nodes. For n_1 and n_2, the labels and distance are added to the output. In the next step, the dissimilarity (distance) from the new node to all others in the set is calculated. This is repeated until all nodes are clustered based on the cut-off distance or desired number of clusters. The result is a list of labels for all input points. All points that are labeled with the same number are part of the same cluster, and become a single data point in the new, coarser level of detail. Figure 3.7 shows the resulting levels of detail for the molecule 2btv.

In order to create the new data points from the labeled cluster data, we need to find a position, and a radius for each new point. Additionally, we have to choose one point in each cluster that will be the "heir" to the parent on the coarser level. The "heir" is the point into which all other points in the cluster transition when smoothly blending between levels. This information is necessary in order to achieve a smooth transition between different levels of detail, as described in more detail in Chapter 4. We choose the point with the shortest distance to the center, i.e., the position of the new point representing the cluster. We determine the position of the new point p_c by calculating the mean of all points in the cluster $p_c = \frac{\sum_{i=1}^{n} p_i}{n}$. Both Parulek et al. [PJR+14] and Müllner [M+13] choose a radius that covers all contributing atoms. We found that this method blows up the atom's volume unnecessarily. Individual data points on coarser levels of detail no long represent any actual atoms, but rather a cluster of several atoms or sub-clusters. The relevant factor is therefore how visually similar the calculated LODs

(a) When covering the centers of all points in the previous level, the overall volume of the molecule increases with each level.



(b) Using the average distance between points in as a basis for the radius of the new sphere, the volume remains closer to the original.

Figure 3.8: Molecule 1sva. Comparing radius covering all points in the cluster vs average distance using the original resolution (orange) vs. level 5 (white)

are to the original set of atoms. Rather than letting the radius cover all points in the cluster, we calculate the new radius based on the average distance using the following formula in Equation 3.2, where r_c is the radius of the data point on the coarser level of detail that represents the cluster, n is the number of points in the cluster, p_i is the position of a point in the cluster with the radius r_i, and p_c is the position of the new data point, which is the average of all positions within the cluster.

$$r_c = \frac{\sum_{i=1}^{n} \frac{distance(p_i,p_c)}{2} + r_i}{n} \tag{3.2}$$

Figure 3.8 compares our proposed method to a radius that covers all contributing atoms. The only exception we make is for the final level, which is always one sphere per page. In this case, we do choose a radius that covers all spheres on the previous level.

3.2.3 Storage and Encoding of Data Points

For each point, we need to store both the radius, and the parent ID, which links the point and all its siblings to the point that represents the cluster they belong to on the next level of detail. The first three components of the four-component vector that stores each individual point are needed to define the point's spatial position. Therefore, all additional information has to be stored in a single 32 bit floating point. We use 16 bit to

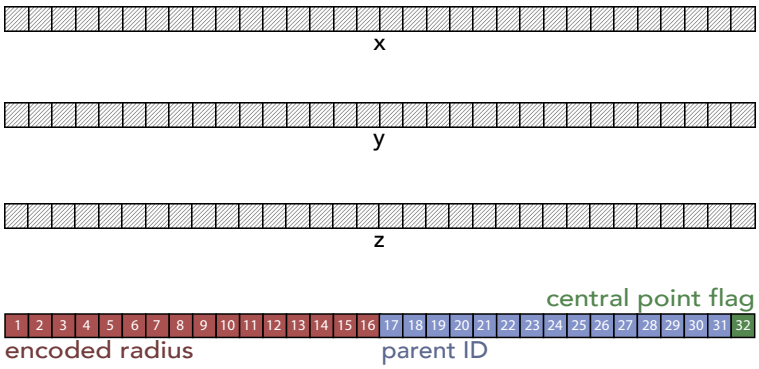


Figure 3.9: A single data point stored in a 4 x 32 bit component vector

store the radius, and 1 to flag the central point in the cluster, i.e., the "heir", leaving 15 to store the parent ID. This allows us to store 32,768 distinct parent IDs, which defines the limit of how many atoms can be in a single page.

Figure 3.9 illustrates how the data for a single point is stored. We encode the radius to use those 16 bits of storage more efficiently. The smallest possible radius for an atom is defined as 1.0, and we assume that even cluster points of accumulated atoms will not exceed a radius of 50.0. As a 15 bit variable can store 32,768 unique values, an encoded radius r_e is calculated for each point's radius r using equation 3.3. This maps r to a value between 0 and 32,768. In the shader, the encoding is reversed before using the radius to render the corresponding sphere.

$$r_e = \frac{r-1}{50-1}32768 \tag{3.3}$$

3.3 Summary and Conclusion

In this chapter, we presented the concept of our data structure and its place within the framework. We implement an octree constructed by nesting page components, that encapsulate the data points. The extent of a pages bounding box is determined by the given maximum amount of point in a page, as well as the density in the region. Additionally, our data structure provides levels of detail that are calculated using agglomerative hierarchical clustering. Levels of detail are linked together using a parent ID, which makes it possible to smoothly interpolate between them, as we will see in the following chapter, where we will also see some of the advantages of dividing space hierarchically based on the density of a region. As the individual pages can be used to limit the amount of data that needs to be considered when applying methods based on neighborhood queries, as well as loaded and rendered per block, it allows us to deal with large data sets that would be problematic without an advanced data structure.

CHAPTER 4

Rendering

When visualizing molecular data, it is important to allow the user to investigate both the overall structure, and small details. Understanding the structure as a whole, requires the user to be able to interact with it in real time. In order to make it easier to see structural details, enhancement methods are helpful. This includes both general visual enhancement methods that help to guide the viewers eyes to important details, such as depth of field and ambient occlusion, and surface representation methods that are specific to molecular data. In the first part of this chapter, we describe how we use our data structure to render data sets. In part two, we give an overview over the enhancement methods implemented in the framework.

4.1 Managing and Rendering the Octree Data Structure

The aim of our work is to render several million atoms in real time, without requiring additional structural information, while facilitating enhancement and surface representation methods. We use a least recently used (LRU) cache, implemented on the CPU, to manage the data structure described in Chapter 3. For any given frame, a certain number of pages are visible, and need to be in memory. This sub-set of data is called the working set.

Figure 4.1 shows the main components involved in rendering the data structure. Components in gray are responsible for back-end data handling, which was discussed in the previous chapter. This chapter is concerned with the components above the dashed line, which are responsible for managing and rendering the data structure that was created in a pre-processing step. The sphere renderer, which is a part of the viewer, manages the rendering process. It has access to the viewer, which contains a scene, in which a protein is stored. The cache, which handles the data using pages of the octree, is accessed by the sphere renderer via the other components. The figure also shows the sphere renderer's

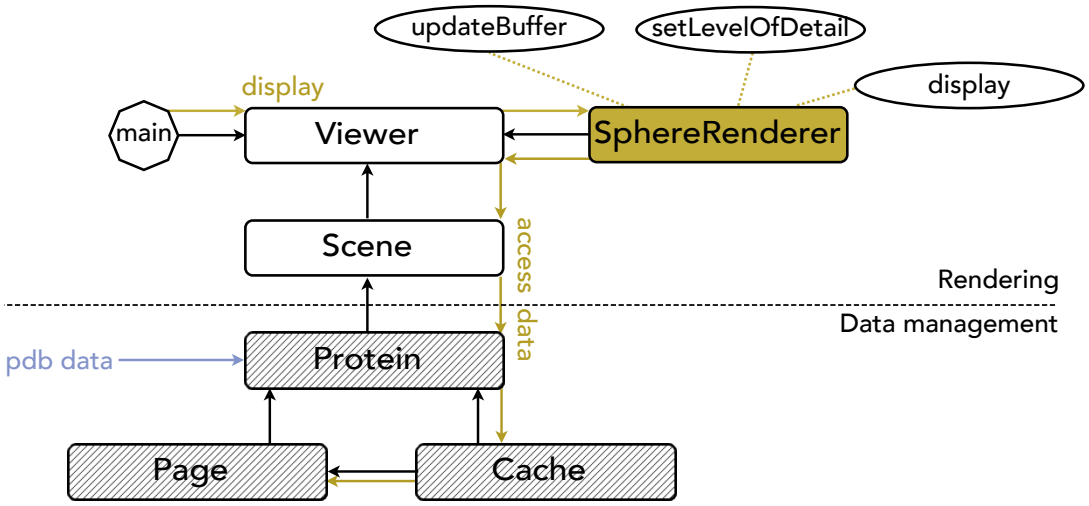


Figure 4.1: Rendering components. The main components responsible for the rendering process are shown above the dashed line. Attached to the SphereRenderer are the component's most important methods.

three most important methods for the rendering process, shown as ellipses connected to the component from which contains them.

4.1.1 Memory Management Using an LRU Cache

When rendering large data sets on a screen with its limited resolution with the entire data set in view, individual spheres at some point become so small that they cover less than a single pixel. Additionally, there is a high chance for data points in the back to be covered up by other geometry. While zooming in to a part of the data set to look at details on th other hand, parts of the data sets are invisible as they remain outside the view frustum. Therefore, it is hardly ever necessary to render the entire, large data set at the same time at full resolution.

In our pre-processing step, we have already spatially divided our data set into pages. This allows us to look at entire chunks of the data set at once, instead of having to check visibility for all individual atoms. We also know the maximum size of each block, as it is possible to choose that when building the data structure. Therefore, we can now use information about the system's limits, or decide how much of its capacity we want our program to use, to determine a maximum amount of pages that can be uploaded to the GPU at the same time. If the allocated capacity is reached, a decision has to be made about which pages to discard. This is where caching comes in as a memory management strategy. The cache helps to efficiently update and manage which parts of the data set

are needed. Data can simply stay in memory, even when it is not visible, until we run out of memory, and the space is needed by another page.

All types of cache have some limit to their memory. When this limit is reached, something has to be removed in order to find space for a new entry. Depending on the application, there are different schemes that can be used. Least recently used (LRU) caches get their name from the fact that they remove the least recently used entry in order to make space for a new one. In this context, least recently used means the page that has not been requested or rendered for the longest amount of time. Consecutive frames generally contain a similar sub-set of data, so data points that have been accessed recently, have a high probability of being visible again in the next frame. Pages that have been out-of-frame longest on the other hand, are more likely to be located in a completely different area of the data set, and are therefore less likely to be needed for the next frame. LRU caches are common in volume rendering (Hadwiger et al. [HSS+05], Reichl et al. [RTW13]), though they have been used in direct particle rendering as well (for example Fraedrich et al. [FSW09]). According to Beyer et al. [BHP15], another common strategy is to use a combination of least recently used and most recently used (MRU) caching. In the suggested strategy, LRU is used as long as there is enough space in the cache. If the working set becomes too large for the cache, the program switches to MRU, to reduce cache trashing, i.e., competition for available slots, resulting in cache misses or overwriting of data blocks that are needed. In our implementation, we choose to implement an LRU cache, and render the molecule at a lower resolution when we run out of cache space instead.

We implement the LRU cache on the CPU side. Usually, an LRU cache is based on two basic data structures, one that implements a queue, and one which provides hash functionality. The implementation of our LRU cache is based on a list, and an unordered map. As a key in the map, we use pointers to the pages. As value, we store a vector containing the level of detail at which the data is uploaded, as well as the position in the cache. If there are unused slots available in the cache at the time when the page is added, the saved position is simply the position of the entry in the map. In case of a full cache, we replace the least recently used page, that is the one at the bottom of the list with the new page, and assign the original position of the replaced page to the new page. This information is used when we use the LRU to update the buffers.

Figure 4.2 shows the molecule 3j3q. The pages' bounding boxes are colored according to the age of the page. Pages shown in green have been accessed recently, while pages shown in red would be the first to be replaced, in case of a full buffer and cache.

4.1.2 Updating GPU Buffers Using the LRU

On the GPU side, we use two buffers that store two neighboring levels of detail, which can be interpolated in the shader. We update them using the LRU cache, and a list of visible pages for the current frame.

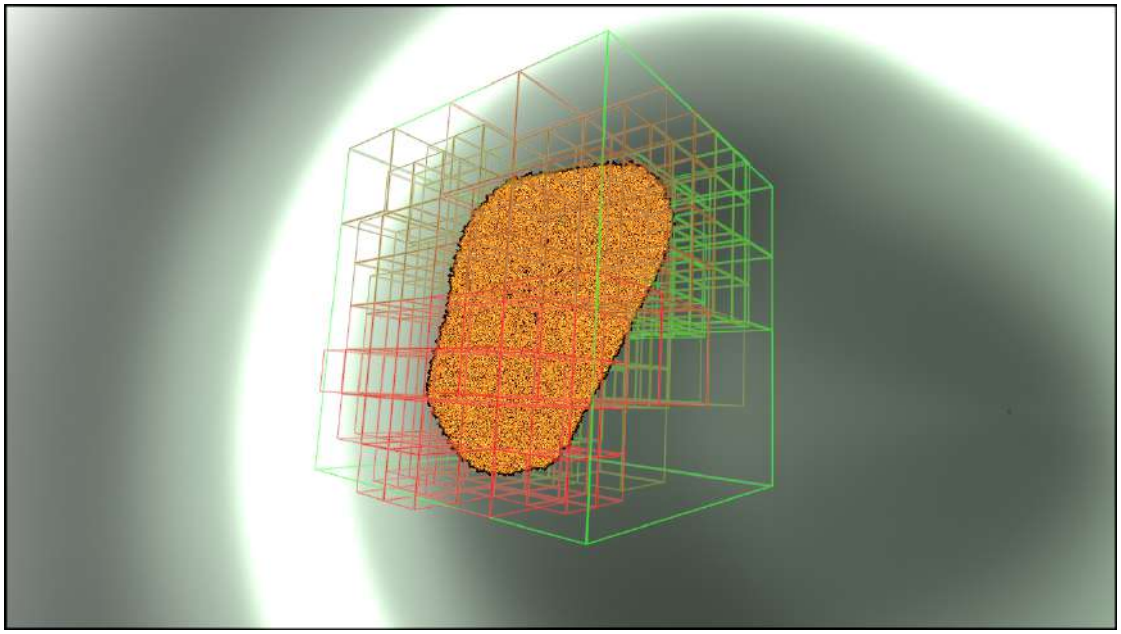


Figure 4.2: Molecule 3j3q with colored bounding boxes for the pages of the octree. The pages that were least recently accessed are shown in red, the most recently accessed pages in green.

To update the rendering list, i.e., the list of at least partially visible pages for the current frame, we perform coarse view frustum culling by checking each page's bounding box against the view frustum. A page that is entirely outside the view frustum, is not added to the rendering list, which effectively means that we are discarding invisible parts of the data for the current frame. After checking if the bounding box of a page lies within the view frustum, we update the LRU cache for all pages that are at least partially visible. The LRU cache saves a new pointer, if the page is not yet resident. If it already exists in the cache, it reorders the page pointers in the list, thus marking the page as most recently used.

To the second buffer, we upload the data for the next coarser level of detail. We want to have two neighboring resolutions available in the shader in order to be able to blend levels of detail smoothly. Both buffers are updated each frame, using the same LRU cache and list of visible pages.

Figure 4.3 illustrates three situations that can arise when adding a page to the cache. The upper row represents the cache, the lower row a buffer it manages. Blue pages have already been added to the rendering list for this frame. In the first case on top of the figure, there is unused space left, so we do not have to delete currently unused data in the buffer. Instead, we assign it to the next available slot in the buffer. The second case in the middle shows a full buffer. However, the number of pages added for the current frame is still lower than the total number of entries allowed in the cache. We replace

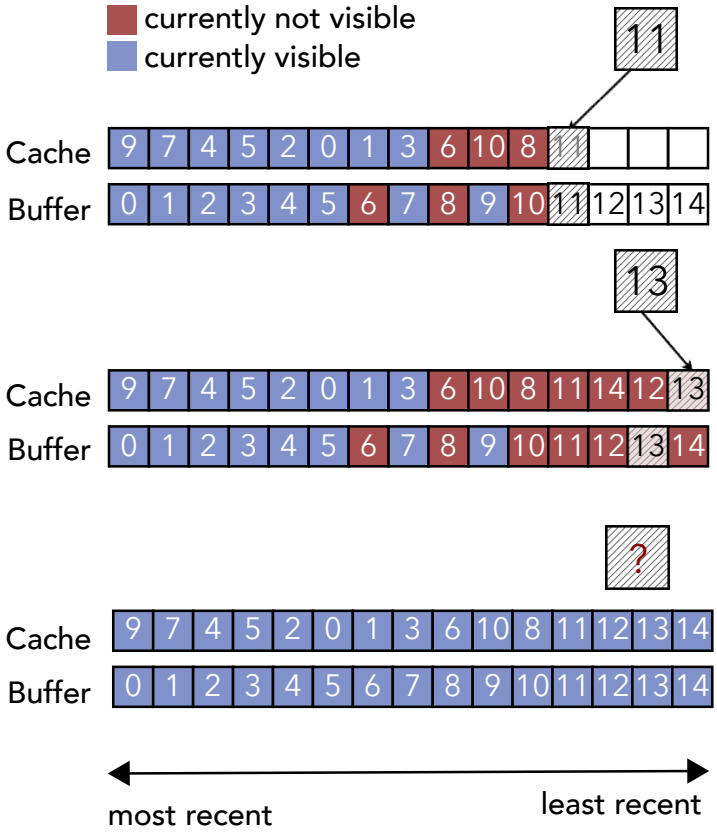


Figure 4.3: Cache and buffer update. The first case in the top row shows a cache with available space left, in the case shown in the middle, the cache is full, but parts of the data are not currently needed, and the last illustration shows a cache that is too small to hold all currently visible pages.

the least recently used page, and assign its slot, in this case number 13, to the new page. The last case on the bottom of the figure shows a full cache. The rendering list for the current frame exceeds the total number of allowed entries in the cache. In this case, we do not replace the least recently used entry, as it is needed in the current frame. Instead, we stop adding pages to the rendering list. The user has to choose to either reduce the resolution, or zoom in to the relevant area of the data set, so that some areas become invisible. It should be noted that, if the entire capacity of the GPU is used, this extreme case only occurs when rendering data sets of several tens or hundreds of millions of atoms, depending on the system.

Which position in the buffer we write the data to, is also managed by the cache. For each page in the rendering list, we check if it is already resident in the buffer by looking it up in the cache. If it is not, but there is still empty space available, it is assigned a position. The offset of a newly added page's data in a buffer is defined as its position in the cache

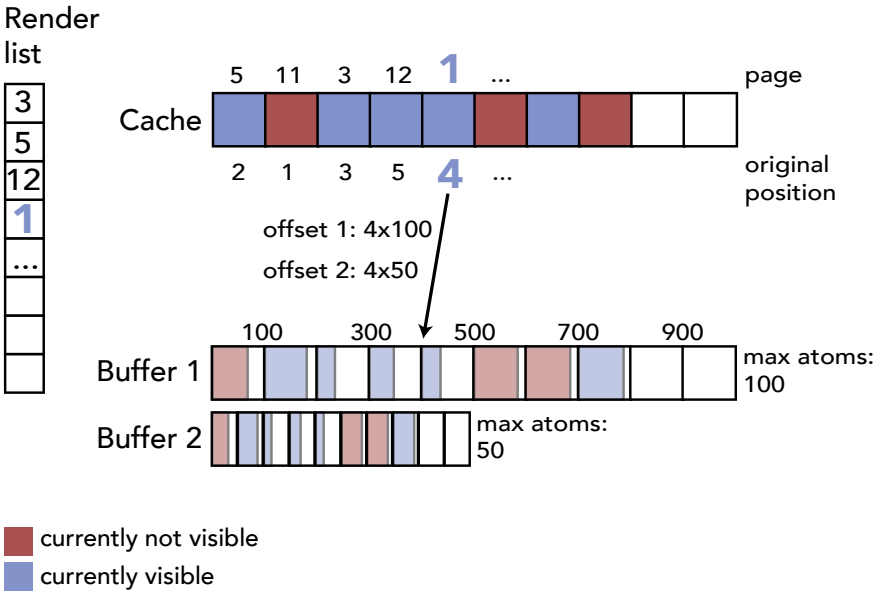


Figure 4.4: The buffer offset is calculated based on the original cache position and the size of buffer chunks.

multiplied by the maximum number of atoms any page at that level of detail contains, as illustrated in Figure 4.4. This offset remains the same until the page is thrown out of the cache. Even if a page is temporarily outside the view frustum, its data remains in the buffers, where it can be used again at need, without having to upload it again. For example, if we add a new page with a maximum number of atoms 100 at level 0 and 50 at level 1, and a page is added to the cache at position 4, the offset when uploading to the first buffer is 400, and that for the second is 200.

When we remove the least recently used page in order to make space for a new page, we need to know which part of the buffer the data associated with the deleted page resides in. This part of the buffer will be overwritten with the data of the new page. That is why the cache saves the original position of a page in the value vector. When we replace a page, the new page gets assigned the original "position" of the page it replaces, and we overwrite the old page's data in the buffers, using the offset derived from the position and the maximum number of points for a page at the currently used level of detail.

The basic level of detail is the most detailed resolution we want to be able to display. Usually, that is the original resolution. As we show the entire molecule as a default view when we start the visualization, the basic level of detail is set to the highest resolution at which all atoms of the molecule can fit into the available space. The user can later change the basic level of detail in the interface. However, that clears the LRU cache and doing so can therefore decrease performance for very large molecules. The basic level of

detail determines the size of buffer chunks, which is why we clear both the cache and the buffers when it is changed.

In addition to the basic level, we define the rendering level, which is the level of detail that we actually show. It can be the same as the basic level, or any level coarser than that. If, for example, we have enough available space for the buffers to view the molecule at the original resolution (level 0), but want to view it at level 2 to increase the frame rate, we can change the rendering level. Instead of replacing and reshaping the entire buffer, which is a costly and inefficient thing to do frequently, we simply re-use buffer chunks and replace each block with its level 2 data when it is needed. Therefore, we only recommend changing the basic level to anything other than full resolution, when the GPU actually runs out of memory.

The rendering level also has to be taken into account when updating the cache and buffers. Therefore, we also save the difference between the basic level, and the rendering level in the value vector associated with a cache entry. If, for example, the global basic level is 1, but we uploaded the data at rendering level 2, we save the number 1, indicating that the resolution saved in the buffer differs from the basic level of detail by one level. When we try to add a page to the cache, we have to check that value as well to see whether it is saved at the desired resolution. If it is not, we update the value vector, and return the offset of that page, which we then use to replace the data in the cache with the new resolution.

Figure 4.5 shows a schematic illustration of the buffers for a given frame. All pages shown in blue are currently at least partially visible, while pages illustrated in red are completely outside the view frustum. The smaller pattern indicates pages rendered at a finer resolution, while the ones covered in coarser dots are saved at a lower resolution. The red pages without a pattern are neither visible, nor resident in the buffer, while the two red-dotted pages were visible in a previous frame, and have not been deleted. Though they are not rendered for the illustrated frame, the data associated with these pages is still in the buffer, as there is space available in the buffer and cache, shown in white in the illustration.

4.1.3 Rendering for Data Sets of Different Sizes

Initially when loading a data set, the entire molecule is in view. We render it at full resolution if possible, and at the lowest available resolution if not. The user can later adjust the position of the protein, as well as choose the level of detail. The goal is to make rendering as efficient as possible. Data that actually needs to be rendered at a given time has to be prioritized when uploading data to the GPU, while invisible data should be discarded as early as possible.

As our standard rendering method, we use the list of visible pages to draw them individually. We also support drawing the entire buffer in a single draw call instead, but in that case, we do not have access to the information that links point to their parents,

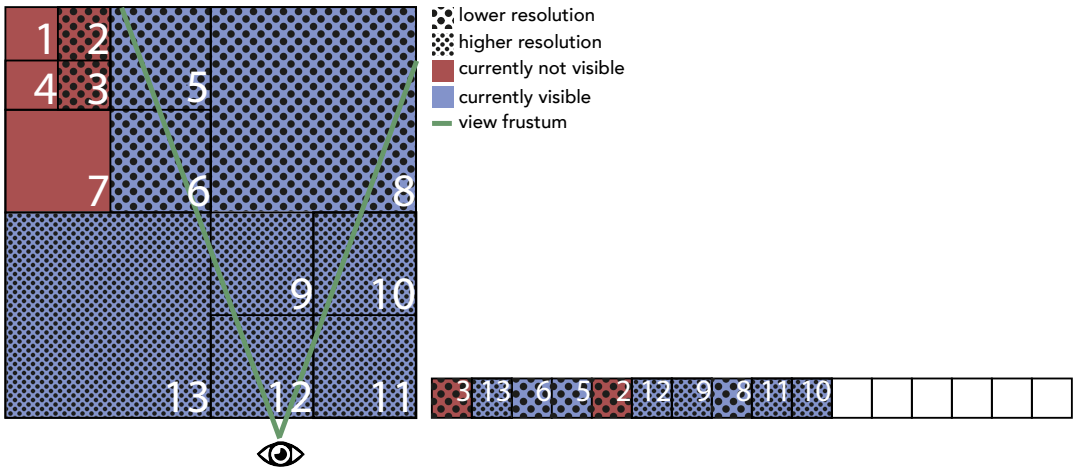


Figure 4.5: The cube on the left is a two-dimensional representation of an octree data structure with different-sized pages. Blue pages are currently at least partially within the view frustum (green), red pages outside it. The dots represent the level of detail at which the data is saved in the buffer. The two red dotted pages are in the buffers because they were previously visible.

as parent IDs are relative to the page, not the entire buffer, so we cannot interpolate between levels of detail.

Though all individual molecules currently available at the Protein Data Bank easily fit into memory using our test hardware, we want our system to be able to handle larger data sets. Therefore, we provide a set of options for very large data sets.

The size of the buffer slots is determined by the maximum number of atoms in a single page at the current basic level. Therefore, the number of pages we can upload also depends on the basic level of detail. By default, that is if the memory capacity allows it, we render the entire data set at the original resolution when starting the program. The maximum number of pages allowed in the cache is generally equal to the total number of pages in the current data set, and the minimum level of detail is 0, or the original resolution. However, the user has the option of limiting the number of rendered pages manually later. If the minimum LOD is for example set to level 2 instead, levels 2 and above can be shown, but not the more detailed levels 0 and 1.

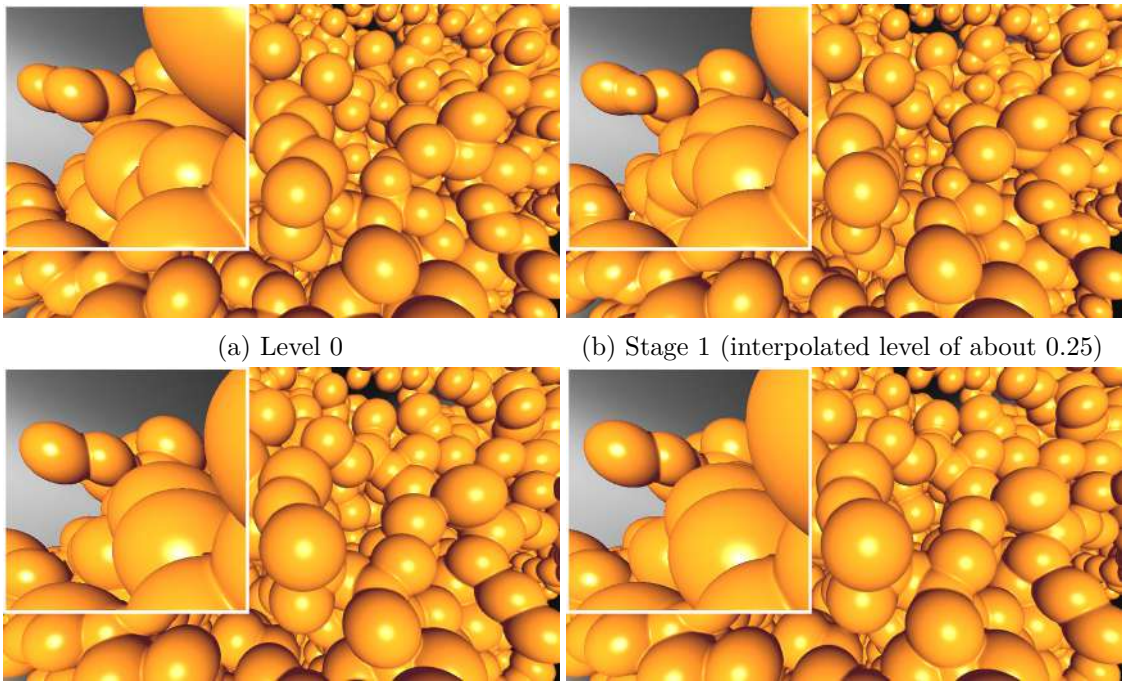
The maximum number of pages of a certain resolution that we can fit onto the GPU can be calculated depending on available memory. If the entire data set at original resolution is too large to fit into memory, we run a GPU memory test to determine that limit for all levels of detail. We then show the resulting limitations in the GUI, giving the user the option of choosing whether to limit the maximum amount of pages, or the minimum resolution. Limiting the number of pages means that only parts of the large data set can be shown at any time. By default, we render the data set at the coarsest available level of detail if the original resolution is not possible. If the user wants to see more details,

they have to change it back manually. The reason we implemented it like that rather than automatically showing the highest possible resolution, even for very large data sets, is that it slows down the system. Additionally, if the data set is very large, and all of it is visible, it is probable that details are too small to be seen anyway. We find it more convenient to be able to handle the entire data set at a "fast" resolution, and change to a more detailed view after zooming into an interesting area.

4.2 Smoothly Blending Between LODs

We want to be able to show more than one level of detail at the same time. Figure 4.6 illustrates the transition from one level of detail to another. As described in Section 3.2.2, we use a flag in the vector's fourth component to mark the point that is closest to the center point of the cluster. The transformation from one level to the next, happens on a scale from 0 to 1, where 0 is the higher resolution level (level 1 in the example) and 1 the lower resolution level (level 2 in this case). Between these two states, all children gradually migrate towards the parent position, i.e. the center of the cluster. For all points except the "heir", the radius decreases from its original value towards 0. The radius of the "heir", that is the point closest to the center, increases from its own original radius to the radius of the parent sphere. Alternatively, one could let all spheres grow and migrate closer to each other without singling out an heir. While that strategy saves the hassle of defining an heir, it leads to issues when rendering Gaussian surfaces. The optimized algorithm proposed by Bruckner [Bru19], which we use to calculate the Gaussian surface, determines where a surface has to be drawn by creating an intersection list. The surface is only created in areas where spheres overlap. If we let all spheres in a cluster migrate towards the center, they overlap more and more, the closer we get to the new level of detail. The more spheres overlap, the larger the surface area. Towards the end of the transition, all spheres are at approximately the same position, and have the same size, leading to almost 100% overlap of many spheres. When we reach the new level, all spheres in the cluster are removed in favor of the single sphere that represents them on the new level. Once that step is performed, there is no overlap anymore, which means that no Gaussian surface is created either, leading to a sudden reduction in the size of the surface. As the goal is a smooth transition, we therefor opted to choose a single heir instead. Thus, all other spheres can be gradually shrunk and then removed, which means that right before the swap, there is usually only a single sphere to be replaced. Even if one or two of its neighbors are left, they are at this point both tiny and within the radius of the now enlarged heir. Therefore, they do not visibly impact the Gaussian surface, which allows for a smooth transition.

The data structure containing the pages at all levels of detail is stored in a nested structure of pages, pointers, and arrays, containing the data in the leaves of the octree. On the GPU, only two levels of detail are available at each given time for each loaded page. It would be possible to upload all available levels, but as the maximum number of data points per page is limited to 2^{15}, it is never necessary to interpolate between more than two levels of detail within a single page. Instead, we choose the level of detail



(a) Level 0 (b) Stage 1 (interpolated level of about 0.25)

(c) Stage 2 (interpolated level of about 0.75) (d) Level 1

Figure 4.6: Molecule 1sva. Transition of the entire molecule between level 0 and 1

that is uploaded to the GPU per page. As mentioned in the previous section, we have a basic level of detail that determines the division of the buffer, and a render level that determines at which level of detail the page is actually rendered. The rendering level can never be a more detailed LOD than the basic level. Apart from this limitation, it can be changed interactively by the user, or determined depending on the frame rate, or distance. The distance-based option reduces the level of detail with increasing distance from the camera, while the frame rate based modus reduces the detail once the frame rate drops below a specified threshold.

4.2.1 Distance-based LOD Adjustment

We are trying to reduce the number of rendered spheres in order to speed up the rendering process, while impacting the visual quality as little as possible. Spheres that are further away from the camera are rendered smaller than closer spheres in perspective projection. Additionally, they have a higher chance of being at least partially covered by other spheres. Reducing the resolution in the back is less likely to be perceived by the user. We can chose the resolution per page, depending on the distance between the camera and the center of the page's bounding box. However, we want to avoid visible artifacts and sudden changes between neighboring pages. Therefore, we additionally smoothly interpolate between levels in the shader.

On the CPU, we choose which level of detail to upload to the buffers, depending on the distance to the camera. The basic level of detail for the entire molecule l_b, is by default level 0, that is the original resolution, but can be coarser if the molecule is too large to fit into memory. The user can choose the distance threshold t, and the amount of tolerance ϵ in the interface. On the CPU, only the distance threshold is used. For all pages where the bounding box center has a distance to the camera d_p, which is greater than that threshold, the next coarser level of detail is uploaded to the buffers. If the basic rendering level for the entire molecule is 1, the level in its secondary buffer would be 2, while for all pages beyond the threshold, level 2 would be uploaded to the first, and level 3 to the second buffer. The information at which level of detail the atoms of a page are uploaded is needed in the shader in order to interpolate between individual points. We encode it based on the difference to the molecule's basic rendering level as shown in equation 4.1. If the basic level of detail is used for a page, the difference is 0, if it is beyond the threshold and the next coarser level is used, the difference is 1. Currently, our system only supports interpolating between three levels of detail. However, it could easily be expanded to several levels of detail if data sets with greater depth differences required it.

$$l_p = \begin{cases} 0, & \text{if } d_p < t \\ 1, & \text{if } d_p \geq t \end{cases} \tag{4.1}$$

In the shader, we use both the distance threshold, and the tolerance to achieve a smooth transition, using the mechanism described in section 4.2. Here, we deal with individual atoms or points representing several atoms, rather than entire pages. Figure 4.7 illustrates the interpolation between three levels in the shader. The pages shown in white are those with center points closer to the camera than the cut-off distance, while the centers of the gray pages are further away. The dashed line in the center of the figure shows the threshold. The blue area around it, defined as $t \pm \epsilon$, shows the tolerance. All points that fall within that area, are rendered at the overlapping level. If the white pages are rendered between level 0 and 1, and the gray pages between 1 and 2, everything within the blue area is rendered at level 1. Equation 4.2 shows how the rendered level of a point is decided. For each atom (point), we have the option of interpolating between the two levels in the buffer, one finer, and one coarser. Which LODs those are, is decided on the CPU, see equation 4.1. As described in section 4.2, interpolation happens on a scale from 0 to 1, where 0 is the finer, and 1 the coarser level of detail. The cases shown in Equation 4.2 are marked with numbers from 1 to 5 in Figure 4.7. In case 1 and 5, the finer and coarser available level are rendered respectively. Case 2, which corresponds to the thick blue line in the illustration, uses the level the pages have in common, for example level 1 in the example where we interpolate between levels 0 and 2. Cases 3 and 4, the areas between the dashed blue lines and the solid blue line, are the areas where actual interpolation happens. The closer the atom or point is to the camera within the extent of the area ϵ, the closer to 0 its interpolation result.

It has to be noted that even with our interpolation system, abrupt level changes can happen if the tolerance area ϵ is too small. Figure 4.7 shows a relatively small ϵ. The red lines show where abrupt changes would happen, as the pages shown in white would be rendered at level 1 and the gray pages at level 2. The correct solution to this issue is simply to choose an appropriate size for ϵ. In practice, the difference is usually not visible, as points in the part of the molecule facing away from the camera are mostly covered by the points or atoms closer to the camera.

$$i = \begin{cases} 0, & \text{if } d < t - 2\epsilon \quad \text{(case 1)} \\ 1 - l_p, & \text{if } t - \epsilon < d > t + \epsilon \quad \text{(case 2)} \\ \frac{d-t-\epsilon}{\epsilon}, & \text{if } t + \epsilon < d < t + 2\epsilon \quad \text{(case 3)} \\ \frac{d-t+2\epsilon}{\epsilon}, & \text{if } t - 2\epsilon < d < t - \epsilon \quad \text{(case 4)} \\ 1, & \text{if } d > t + 2\epsilon \quad \text{(case 5)} \end{cases} \tag{4.2}$$

4.2.2 Frame Rate-Based LOD Adjustment

One of the declared goals of our framework is to achieve interactive rates. We implemented a method that chooses at which level of detail the molecule is rendered, depending on the frame rate. The idea behind this mode is that a user might wish to investigate the overall structure of a very large molecule in real time, without necessarily caring about the finer details at this stage. For large data sets, it might not be possible to render the entire structure at the frame rate the user desires at the original resolution. We let the user specify a target frame rate via the interface. If the average frame rate, as calculated based on the last 64 frames, is lower than the specified limit, a coarser level of detail is chosen for the entire molecule. This method changes the rendering level, not the basic level, so the cache structure remains the same. The method contains a counter that measures the time that has passed since the last change of level. As operations such as updating the atoms saved in the cache to the new rendering level have to be performed, it takes a couple of seconds for the frame rate to stabilize after a change of level. If the frame rate drops below the threshold, we check the counter to see whether at least 10 seconds have passed since the last time the level was changed by the frame rate based rendering method before changing it again. We lower the resolution one level at a time.

4.3 Molecular Surface Rendering

Though enhancement methods and surface models are not the main focus of our implementation, we include basic methods as a proof of concept. The atoms or cluster spheres can be rendered using the van der Waals, or a Gaussian Surface Model. For structural and depth enhancement, our framework includes depth of field, ambient occlusion, and edge enhancement. This section focuses on GPU aspects of rendering, as we implement the effects largely in screen-space. Figure 4.8 gives an overview over the shader passes in the sphere renderer. Gray shaders represent passes for optional screen space effects.

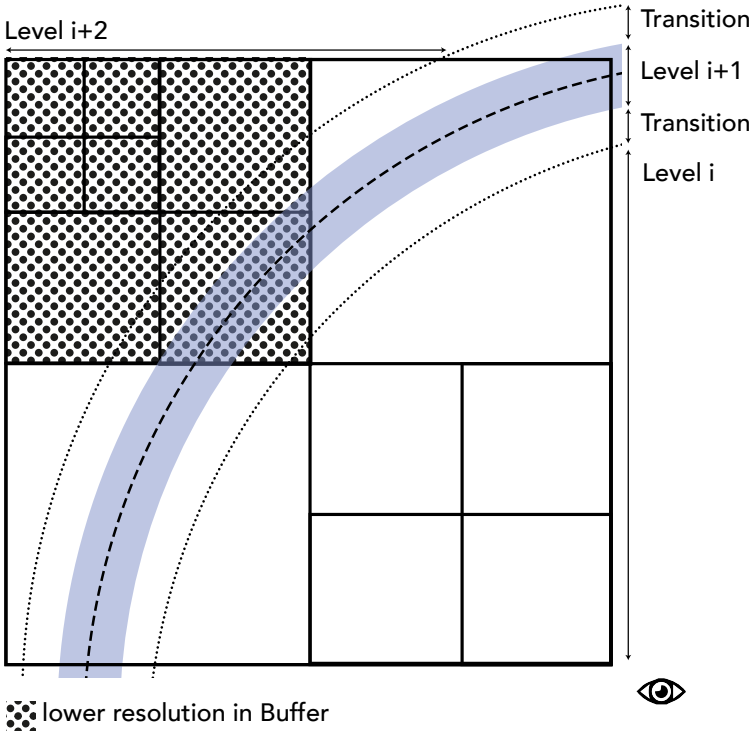


Figure 4.7: Illustration of the smooth transition between levels in distance based rendering. Pages with a center point more than a defined distance from the camera are uploaded at a coarser LOD (dotted pages). Within a parameter-defined area around that distance, we interpolate levels of detail based on the distance.

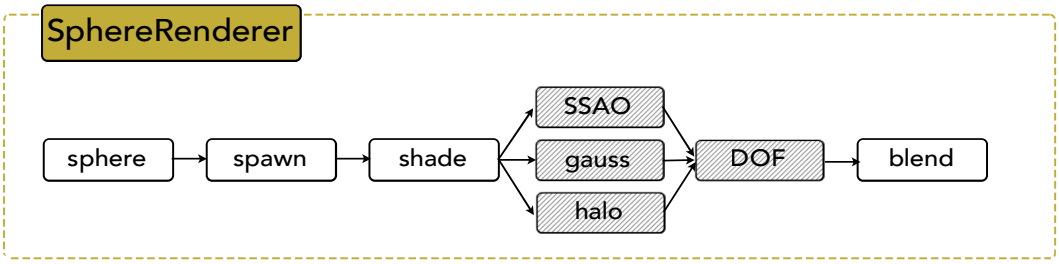


Figure 4.8: Render passes. Shaders shown in gray are optional, the ones in white necessary for our basic rendering process.

The van der Waals Surface Model approximates the molecular surface by rendering individual atoms as spheres using a center position and their van der Waals radii. Our rendering framework is centered around a Gaussian Surface Model as proposed by Bruckner [Bru19], which is based on vdW spheres.

The surface calculation method is achieved in image space, so the algorithm is output sensitive. The influence the calculations have on the achievable frame rate is determined by the atoms on the image plane, rather than the size of the molecule. That makes it particularly suitable for very large structures. The basis of a Gaussian molecular surface is the following density function:

$$\rho(x) = \sum_i e^{\frac{-s||x-c_i||^2}{r_i^2}}$$

c_i and r_i are atom position and radius, s is a scaling factor. The final surface is a union of the Gaussian surface and the vdW surface. Therefore, we can smoothly blend between the two, adjusting the result by tweaking the parameters. In our implementation, the user can do this by changing the sharpness factor and thereby the atoms' sphere of influence. According to Liu et al. [LCL15], it is possible to achieve a good approximation of the SES and SAS with the correct parameters.

To evaluate the density function in image space, an intersection list is created. We define each atom's sphere of influence, which functions as a cutoff radius. Outside the sphere of influence, an atom does not contribute to the density function. Therefore, the density function only needs to be calculated in areas where these spheres overlap, and are not occluded. The size of the sphere of influence is determined as follows, based on the threshold t and the minimum number of contributing atoms at a position N

$$r_i' = r_i \sqrt{\frac{ln(\frac{N}{t})}{s}}$$

We determine the intersection list in a separate rendering pass. The spheres of influence are rendered in the same way as the van der Waals sphers: a single vertex is rendered per atom, and a quad constructed based on the sphere's screen space bounding box. For van der Waals spheres, we stop after the first intersection, as our framework does not support transparency. For each sphere of influence that is not occluded by van der Waals spheres, we save the necessary information as an entry in the list. It contains near and far intersection, center position, the index of the previous sphere entry, as well as the information that is usually stored in the fourth component, i.e. radius, parent ID and heir flag.

Bruckner [Bru19] proposes an adaptation of selection sort for ray traversal. As that algorithm guarantees that the first i elements are correctly sorted after the i-th iteration, and we only need the first intersection with the Gaussian surface, we can stop once that

Algorithm 4.1: Visibility-driven ray traversal, [Bru19]

Input: array of *count* intersection point indices ii, referring to the entry containing information such as *near* and far about the spheres of influence

```
1  ss = 0
   for i ← 0 to count − 1 do
2      k = i
       for j ← i + 1 to count − 1 do
3          if near_ii[j] < near_ii[k] then
4              k = j
5          end
6      end
7      swap(ii[i], ii[k])
       if ss < i then
8          if i ≤ count − 1 or far_i i[ss] < nearii[i] then
9              if interesect(ss, i) then
10                 break
11             end
12             ss = ss + 1
13         end
14     end
15 end
```

intersection is detected. Algorithm 4.1 shows the pseudocode for visibility-driven ray traversal proposed by Bruckner.

To calculate surface intersections, we use sphere tracing in the interval from $near_i + 1$ to $far_j - 1$.

Depth Based Surface

In our implementation, the user can set the sharpness factor s in the interface, which determines the extent of the surface. The radius of the sphere of influence is calculated by multiplying the radius of the atom with a radius scale factor r_s, which is calculated using the sharpness and the contributing atoms constant c_a . For a standard value of $c_a = 32.0$, the radius scaling factor is determined as follows:

$$r_s = \sqrt{\frac{\log c_a e^s}{s}} \tag{4.3}$$

Our depth based surface rendering mode changes the radius scaling factor using the same principles as for distance based LOD rendering (see Section 4.2.1). The user can

choose threshold and tolerance in the interface. All spheres closer to the camera than the threshold are rendered using the radius scaling factor resulting from the chosen sharpness. In the area between threshold, and threshold + tolerance, the scaling factor is linearly decreased to 1, which results in a sphere of influence of the same size as the radius of the sphere itself. As in the case of level of detail rendering, spheres that are further away from the camera, usually take up less screen space and are more likely to be at least partially occluded. They are therefore natural candidates when selectively decreasing resolution or rendering details. Even the optimized molecular surface we use is computationally expensive for large spheres of influence. So for larger surfaces, decreasing calculations on spheres further from the camera is worth the extra calculations required, which is shown in more detail in Section 6.4.3.

4.4 Visual Enhancement for Molecular Rendering

Visual enhancement methods are used to help the viewer to better understand both details, and the overall structure of the data. All of the enhancement methods we include in our framework are screen space methods, based on different variations of blurring passes. In addition, we integrate an ambient occlusion library.

4.4.1 Depth of Field

In photography, depth of field is a result of light rays being broken in the camera's lens. Both photographic cameras, and the human eye are limited in the range of depth they can perceive in focus at the same time. Therefore, we are used to seeing focus as a depth cue. Two of the main reasons depth of field is implemented in computer graphics, are that as depth cue, and to make scenes and objects look more realistic. A realistic looking molecule to the human eye does not exist, given that they are too small for the human eye to see. Depth of field can, however, play an important role when trying to understand three-dimensional structures, irrespective of whether or not we can observe them in the real world. In particle based data visualization, depth of field is often used to emphasize certain parts of the structure by blurring the back- and foreground.

We adapted the implementation by Bukowski et al. [BHOM13] for our framework. Figure 4.9 shows their illustration of the algorithm. It is based on two blur passes, one horizontal and one vertical. In order to achieve the effect, a rendered image of the scene is blurred using a blur kernel to sample and add up neighboring pixels. Two different blurred images are created, one for the background, and one for the foreground, with a stronger blurring effect. The factor that decides which parts of an image are blurred and which sharp, is called circle of confusion (CoC). It is calculated using the parameters aperture, focal length, and focal distance. We store the calculated value in the alpha channel of the blurred texture. The final depth of field image is calculated in a separate shading pass.

In addition, we implement an auto focus option, which sets the region in focus approximately to the current mouse position. We set the focal distance to the z value at the

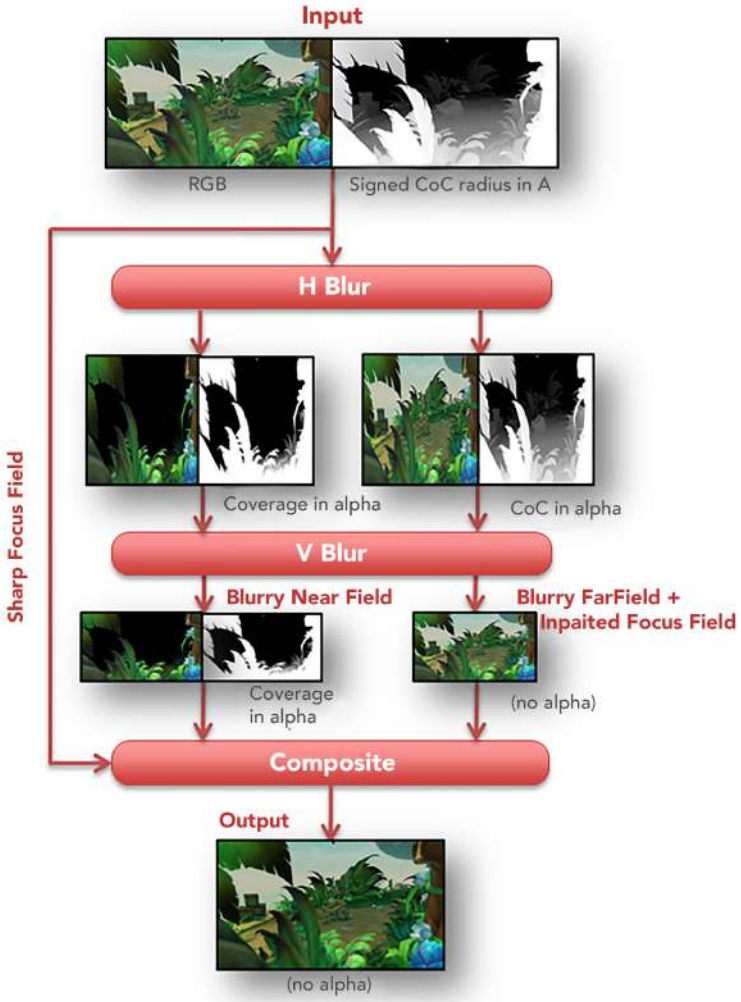


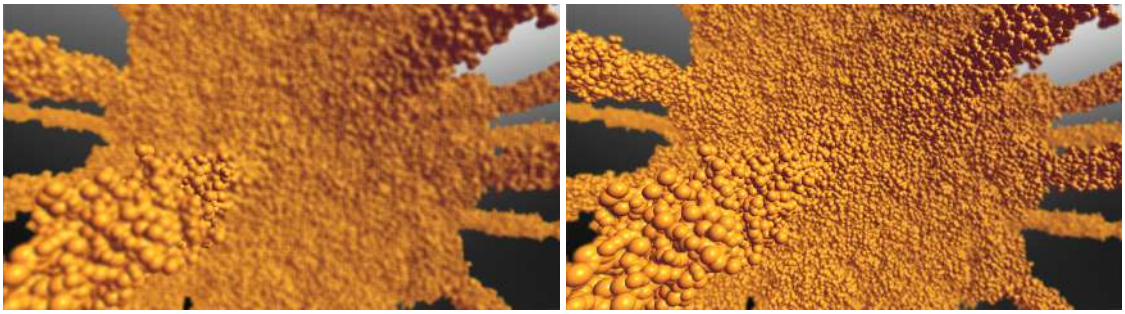
Figure 4.9: Illustration of the depth of field algorithm from the publication by Bukowski et al. [BHOM13]

position of the mouse points at the time the user presses the auto focus button.

Figure 4.10 shows two examples of depth of field blurring. Both screenshots were taken at the same position, and with the same focal length. Screenshot 4.10a uses an F-stop of 0.7, highlighting only a few selected atoms, while screenshot 4.10b, with an F-stop of 11, only blurs distant atoms, while everything closer to the camera is in focus.

4.4.2 Screen Space Enhancement

Ambient occlusion helps to better understand and illustrate the structure by darkening surfaces that are on the inside of a structure. Our framework contains different versions



(a) Blur F-stop 0.7 (b) Blur F-stop 11

Figure 4.10: Depth blur on the molecule 6qz0

of ambient occlusion and edge enhancement. We include the HBAOPlus library for ambient occlusion. Additionally, we implement two versions based on a blurring pass, which the the user can tweak using parameters.

Ambient Occlusion Using the HBAOPlus Library

The NVIDIA HBAO+ (Horizon Based Ambient Occlusion) library is a screen space ambient occlusion library. It is an updated version of the work presented by Bavolia and Sainz [BS09]. It is mainly designed to produce realistic ambient occlusion shadowing.

The library uses the depth buffer as an approximation of the scene. Rays are traced directly in 2D, and the ambient occlusion is approximated from these 2D rays. They use spherical coordinates, with the zenith axis oriented parallel to the Z axis in eye space. They compute incremental horizon angles while stepping forward along a line in eye space, and integrate the ambient occlusion based on the horizon angles.

In our framework, we wrap the library in the SSAO component. The component has its own display method, which we call from within the sphere renderer, if the user chooses to enable ambient occlusion. In the SSAO component, we also set the parameters. Figure 4.11 shows the ambient occlusion effect achieved by the library. Realistic ambient occlusion enhances the structure, and is therefore useful for our purposes. It is not necessarily the only worthwhile technique to use though, given the fact that our goal is to make it easier to understand the structure at hand, rather than light a human-recognizable scene in a realistic way.

Blur-based Structure Enhancement

In addition to the library, we implement screen space enhancement based on two blurring methods, which achieve slightly different results. The first version separable Gaussian blur shader. Is is based on a an incremental Gaussian burring kernel, as described by Turkowski [Tur07], and a GLSL shader published on a blog by Hay [Hay10] [1].

[1] `https://callumhay.blogspot.com/2010/09/gaussian-blur-shader-glsl.html`

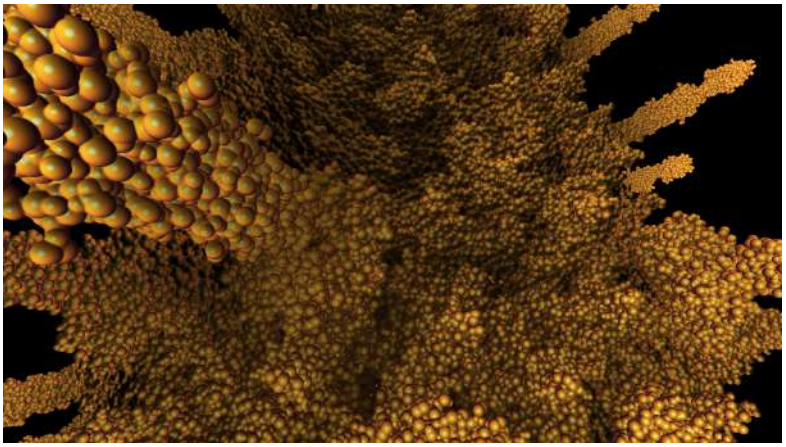


Figure 4.11: Ambient occlusion using the HBAO+ library

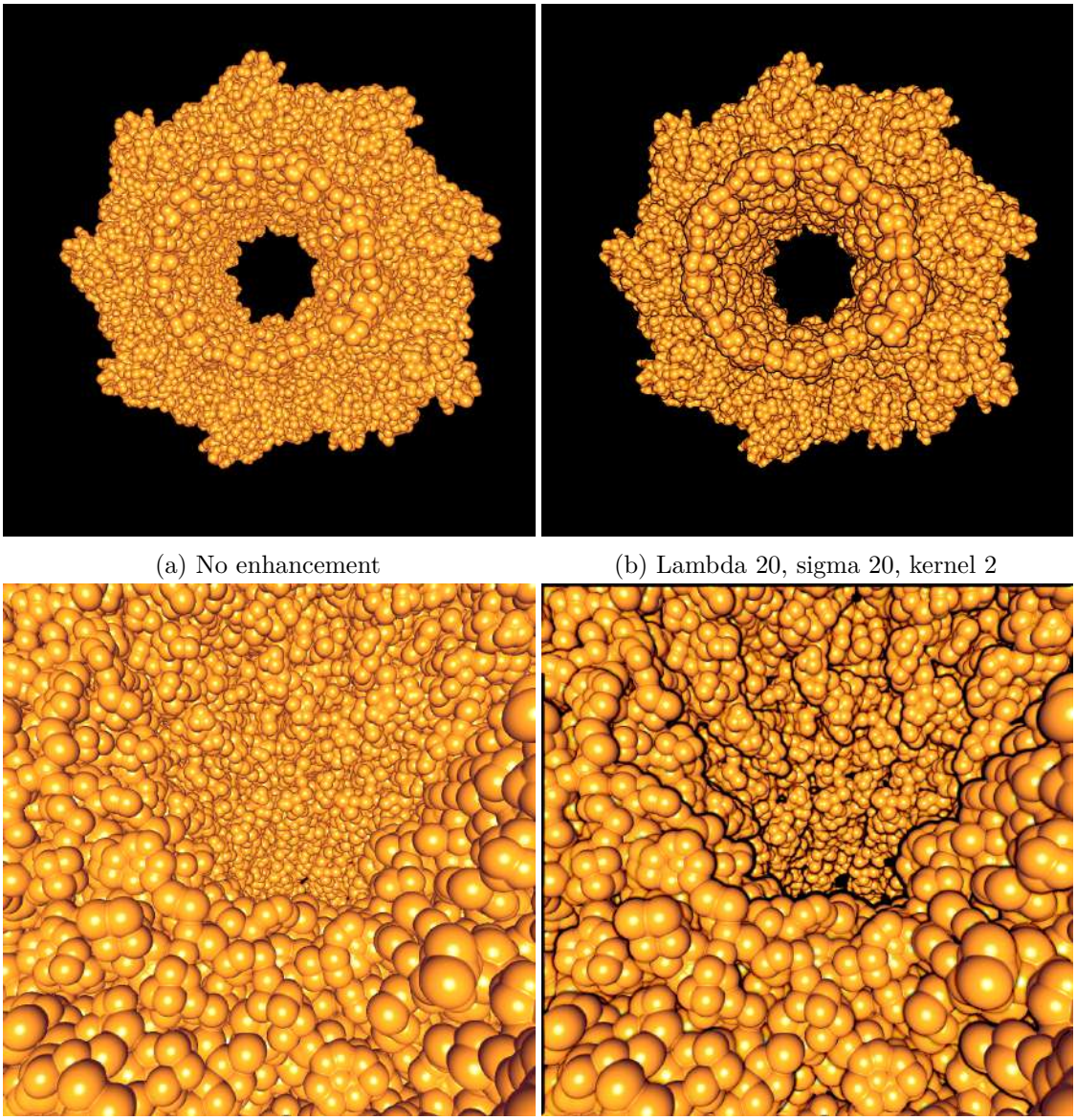
It is used in a post-processing step and takes the output from the previous rendering pass as texture input. The method uses two passes, one horizontal and one vertical, in which the texture is sampled in the given direction. The sampled points are summed up and weighted using the incremental Gaussian coefficients. The incremental Gaussian coefficients are stored in a three dimensional vector and its initial values are calculated as shown in Equation 4.4], where σ can be chosen by the user in the interface.

$$x = \frac{1}{\sqrt{2\pi}\sigma} y = \exp -0.5\sigma^2 z = y^2 \tag{4.4}$$

Figure 4.12 shows examples with two different sets of parameters. The images on the left are without enhancement, while the right side shows images created using the Gaussian blur-based enhancement method. In Figure 4.12b, a smaller kernel size is used, which results in more localized blurring, and thereby limits more to the edges. The image shown in Figure 4.12d is achieved using a larger kernel size, which results in an additional effect that is closer to ambient occlusion and enhances the structures at the core of the molecule.

We also implement another type of blur-based enhancement, which uses a different type of blurring effect. The halo blur shader is based on the works of Bruckner and Gröller [BG07], as well as Rong and Tan [RT06]. Rong and Tan propose a flooding algorithm for Voronoi diagrams. Bruckner and Gröller use a similar approach to create halo fields to enhance volumetric data sets. Each pass, a neighborhood of eight pixels is sampled for each point in the image. At pass n, the sampled point is 2^n pixels away from the point under consideration.

The main difference between this blurring method and the Gaussian blur is that the strength of the effect is determined by the number of passes. Conceptually, the passes are used in a similar way as in the jump-flooding algorithm by Rong and Tan [RT06].



(a) No enhancement (b) Lambda 20, sigma 20, kernel 2

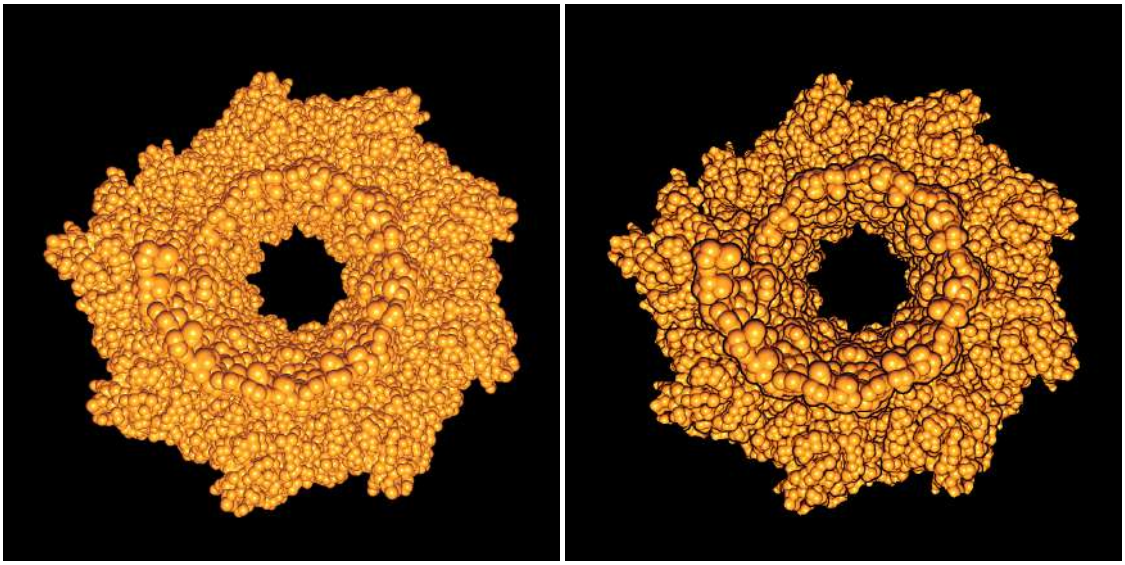
(c) No enhancement (d) Lambda 12, sigma 4, kernel 8

Figure 4.12: Examples of Gaussian blur based enhancement on the molecule 5odv

The distance at which we sample the texture depends on which pass it is. Each pass, we sample pixels closer to our target pixel, until, during the final pass, we sample direct neighbors.

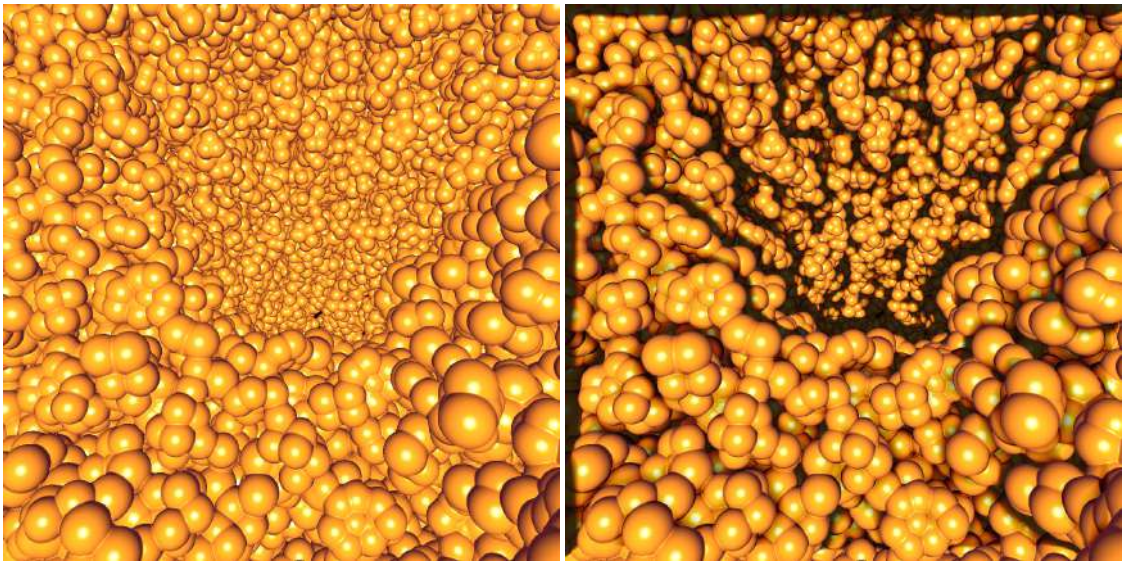
Figure 4.13 shows the results of this rendering method. With fewer passes, such as for example 2 passes in Figure 4.13b, finer details are enhanced. More passes result in a stronger blur, as shown for 32 passes in Figure 4.13d. The result of 16 passes is shown

in Figure 4.13c. Here, the edges become thicker, resulting in an effect that emphasizes larger structures. The disadvantage is that some details are lost in the edges.



(a) Screenshot without the effect

(b) Halo with 2 passes and lambda 50



(c) Screenshot without the effect

(d) Halo with 16 passes and lambda 11

Figure 4.13: Examples of halo-based ambient occlusion

4.5 Summary and Conclusion

In this section, we introduce the methods we use to render the data structure we introduced in the previous chapter. The basic method relies on an LRU cache, a list of

visible pages for each frame, and buffers which we use to upload two different levels of detail to the GPU. The cache has a fixed capacity and saves pointers to the pages that have been added, as well as the pages' rendering level and index in the buffer. We also explain different options for blending levels of detail and look at the special case of very large data structures. In the second part of the chapter, we look at surface models, in particular the Gaussian Surface Model, and screen space enhancement methods.

CHAPTER 5

Implementation

In this chapter, we explain the libraries, tools and technologies used in our implementation. The code is written in C++ and OpenGL. Additionally, we use some libraries and base parts of our implementation on previous work.

Figure 5.1 shows an overview over components and libraries. The components involved

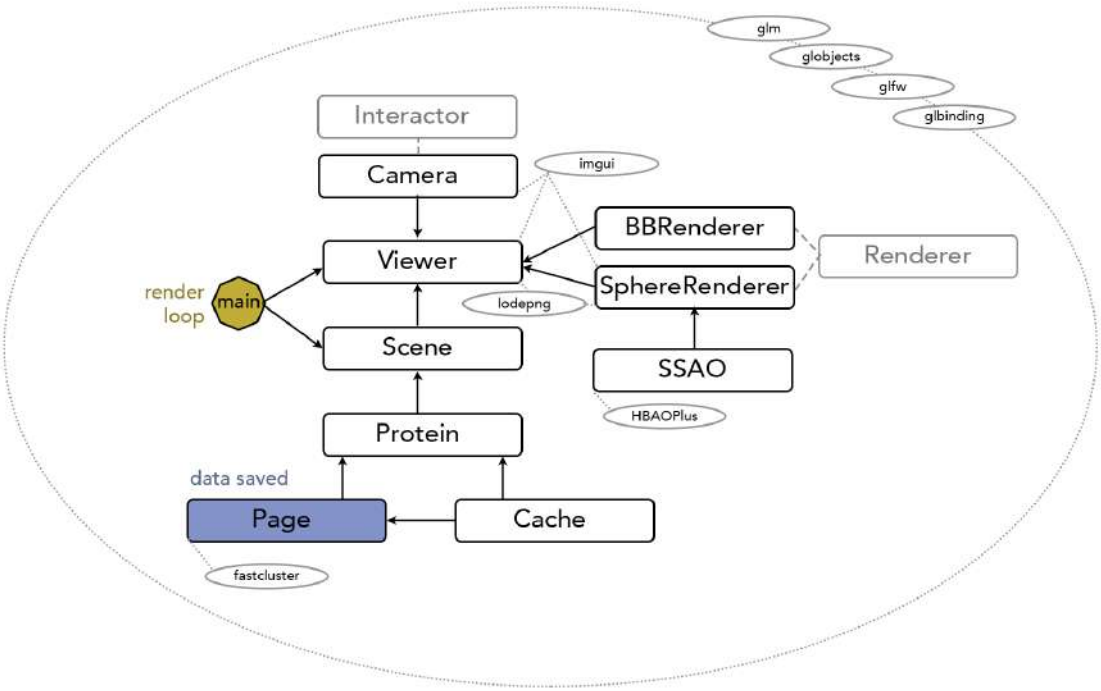


Figure 5.1: System architecture

in rendering and data handling have already been explained in previous sections. Here, they are shown in the context of the entire framework. The viewer is the core of all visualization tasks. It is connected to the data handling components via the scene. It also contains interactors and renderers. Currently, there is one interactor that handles a camera, and two main renderers, the sphere renderer and the bounding box renderer. However, the system is expandable, and can contain as many interactors and renderers as necessary. Libraries are shown in connection with the components that use them. As they interact with most of the components, the GL libraries are illustrated without individual connections to components for the sake of readability.

5.1 Libraries

Many tasks our framework has to handle are standard operations, for which we use libraries when they are not part of the core of our research. In this section, we give an overview over the libraries and how they are integrated into the implementation.

glbinding and globjects

We use glbinding[1] and globjects[2]as wrappers for OpenGL. Globjects is an object oriented wrapper library, which provides an interface for the programmable graphic pipeline. OpenGL objects are mapped directly to C++ classes. Glbinding is a cross-platform C++ binding for OpenGL.

GLM

GLM[3] (OpenGL Mathematics) is a library based on the OpenGL Shading Language (GLSL) specifications. It implements classes and functions in C++, following the naming conventions found in GLSL. We mostly use it for data types such as matrices, but also some of its functions, such as transformations to radians.

GLFW

GLFW[4] is an OpenGL utility library for creating windows, contexts and surfaces and receiving input and events. We use it to create our window, handle keyboard events and measure time.

ImGui

We use the ImGui[5] library to build our graphical user interface. User interface elements are created and drawn each frame, hence the name immediate mode GUI.

[1] https://glbinding.org/
[2] https://globjects.org/
[3] https://glm.g-truc.net/
[4] https://www.glfw.org/
[5] https://github.com/ocornut/imgui

LodePNG

The LodePNG[6] library handles image loading. We use it in the SphereRenderer to load environment maps and in the viewer to save screenshots.

fastcluster

As it is used in part of our core functionality, the fastcluster[7] library has already been described in chapter 3. It is a collection of several variations of the fast cluster algorithm. We use it in the page component, to calculate levels of detail based on iterative clustering.

HBAOPlus

As mentioned in chapter 4, the HBAOPlus[8] library is used for ambient occlusion. It is encapsulated in the SSAO component. That component handles the parameters for the library. It has its own display method, which is called by the sphere renderer, when the user chooses to activate ambient occlusion. The display method requires the view and projection matrix, a framebuffer, as well as depth and normal texture.

5.2 Implementation Choices

In this section, we go through the architecture of our implementation and discuss concrete implementation choices for important components.

5.2.1 Data Structure

On the CPU side, the actual point-based data is stored within nested page components, as marked in Figure 5.1. The single instance of a protein component within our scene handles the basic page, i.e., root of the octree, via a shared pointer. `std::shared_ptr<>` manages shared ownership of an object of the type page. The advantage of smart pointers in C++ is that they manage the destruction of the object they point to. The object's memory is deallocated when the last pointer pointing to it is either destroyed, or assigned to something else. In contrast to other smart pointers such as `std::unique_ptr`, it allows several pointers to point to the same object.

A single page contains a data structure for sub-pages, and one for data points. For each individual page, only one of them is used at any given time. If the page is an internal node in the octree, it stores references to its sub-pages. We store this sub-octree as a 2 by 2 by 2 array of references: `std::shared_ptr<Page>[2][2][2]`. The advantage of a multidimensional array over other structures, such as vectors, is that we can access them in a way that reflects the page's spatial position. Accessing a page in the array

[6] `https://lodev.org/lodepng/`
[7] `https://github.com/dmuellner/fastcluster`
[8] `https://www.geforce.com/hardware/technology/hbao-plus`

using the pattern [1][1][1] for example not only gives us with the reference we need, but also provides information about which area of the parent page it covers.

If a page is a leaf node in the data structure, it does not contain any sub-pages, so no references are stored. Instead, we store the atoms and level of detail cluster points. We store them in a vector of vectors: `std::vector<std::vector<glm::vec4> >`. Individual data points are saved as `glm::vec4`. As the points are later uploaded to the GPU, it is convenient to use the data structures of the GLM library, which provides classes and functions with the same naming conventions and functionalities that GLSL offers in C++. All points at a particular level of a page are saved in a `std::vector`, so there is a `std::vector<glm::vec4>` for each level of detail. The vectors for the different levels are stored in another vector, resulting in the final data structure `std::vector<std::vector<glm::vec4> >`. The advantage of vectors compared to other data structures, such as arrays, is that we do not need to define their size beforehand. While we do know the maximum amount of atoms that can be on a page, most pages are not filled to the maximum, which would leave unused space in an array of pre-defined size. Right now, we calculate a fixed number of levels of detail, but using a vector rather than an array leaves the option of extending it if an additional, coarser level is required.

5.2.2 Cache

Our LRU cache implementation is based on a `std::list` and `std::unordered_map`. The list contains a `std::pair` consisting of a pointer to the page `std::shared_ptr <Page>` and a `glm::ivec2` for the position and rendering level information. As a key, the map also uses the pointer to the page and as a value an iterator over the list.

The C++ data structure `std::list` is a doubly linked list. In contrast to vectors and arrays, it does not store elements at contiguous memory locations. Therefore, insertions and deletions are more efficient, because instead of changing memory, only pointers are changed. This also reflects the way we designed buffer access. While the data of individual pages are stored contiguously, only parts of the memory that have not been used recently are replaced when necessary. Instead, we use index information to point to memory locations. This trait is advantageous in our implementation, because we push pages to the front of the queue each time they are either used or added to keep track of least recent use.

For the map component, we use a `std::unordered_map`, which is the C++ standard library's implementation of a hash map. In contrast to `std::map`, which is based on a binary search tree, it uses a hash table to store data. The order in which the data is added to the cache is tracked by the list, so we do not need the entries in the map to be sorted, nor do we need to know an individual entry's neighbors. We update the map only if the cache does not contain the requested page yet, or when we have to delete a page in order to free up space for new pages. The job of the map is to store the values

associated with the key, without any requirements of order. Therefore, the unordered map is the most suitable type of map.

There are two values associated with rendering a page. Both the rendering level and the offset position in the buffer are integer values. As a map only stores a single value per key, we wrap them in a `glm::ivec2`. Though a `std::vector` would work just as well, `glm::ivec2` limits the data that can be stored to two integer values, which is all we want to allow.

5.2.3 Buffers

To upload data points to the GPU, we need a data structure that does not have a size limit, except the actual hardware constraints. We use the `globjects/Buffer.h` wrapper for OpenGL buffer objects. As we want to be able to interpolate between levels of detail, we use two buffers to store point positions of the current basic level, and the next coarser LOD. For each of them, we store a unique pointer: `std::unique_ptr<globjects::Buffer>`. The initial size of the buffers is determined at start up, and they are resized when the basic level of detail is changed by the user. An alternative would be to store position data in Texture Buffer Objects. The maximum size of a Buffer Object is generally only limited by the size of the GPU. Though the size limit is usually higher than for one-dimensional textures, Texture Buffer Objects do have a size limitation. However, as mentioned in chapter 2, there are several papers with similar challenges to ours that do use textures.

Each time a new page is loaded into the buffer, we replace the data in the relevant buffer area with the points in the new page using `setSubData` for both buffers. The buffer containing the basic rendering level is bound to a Vertex Array Object.

As a standard rendering method, we use `drawArrays` on sections of the buffer based on offsets corresponding to individual pages. We iterate over all pages we want to render for a given frame, and call the method `drawPage`. The method gets the pointer to the page we are currently drawing, as well as the shader to be used. We then fetch the two values stored in the cache, i.e. the page's position in the buffer and the rendering level. We draw the relevant part of the buffer using `drawArrays` from *offset* to *end*. The offset is calculated based on the page's position in the buffer, and the maximum atoms per page for the current basic rendering level (which corresponds to the buffer slot size), and the end is the size of atoms in the current page, so we do not draw the entire slot, just the actual points.

We also upload the page position to the shader, because we need it for the second buffer, which we upload to an Interface Block of type buffer, as shown in listing 5.2. On the C++ side we bind the buffer using `m_lowerLOD->bindBase(GL_SHADER_STORAGE_BUFFER, 3)`. In a geometry shader, we use the page position and parent ID to connect the correct parent to points. For each rendered point, we need to get the parent, the point representing the cluster it belongs to, in order to achieve a smooth transition. Listing 5.3 shows the lines of code relevant to retrieving the position and radius of that point.

Listing 5.1: DrawPage method

```
glm::vec2 values = viewer()->scene()->protein()->cache()->get(page);
long offset = values[0]*viewer()->scene()->protein()->maxAtomsPerPage(m_baselevel);

shader->setUniform("pagePosition", (uint)values[0]);
shader->setUniform("levelUp", values[1]-m_renderlevel);

long end = page->atoms(values[1]).size();
m_vao->drawArrays(GL_POINTS, offset, end);
```

Listing 5.2: Interface Block

```
layout(std430, binding = 3) buffer lodBlock
{
Position positions[];
};
```

Listing 5.3: Get parent point

```
#define DECODERADIUS(a) ((a/32768.0f)*(50))

int sphereID = floatBitsToUint(gl_in[0].gl_Position.w);
int parentID = bitfieldExtract(sphereID, 16, 15);
glm::vec3 pos = positions[parentID+maxAtomsPerPage*pagePosition].pos;

sphereID = floatBitsToUint(positions[parentID+maxAtomsPerPage*pagePosition].rad_id);

int encodedRadius = bitfieldExtract(sphereID, 0, 16);
float sphereRadius = DECODERADIUS(encodedRadius);
```

First, we extract the parent ID from the fourth component of our data point. As IDs are defined per page, not for the entire buffer, we have to combine the ID with the same offset we used to render the basic rendering level in order to get the parent's position. The parent point's radius is then extracted from its fourth component and decoded using a macro.

CHAPTER 6

Results

In this chapter, we present the results of our work, compare different settings, and analyze various parts of the implementation. Our tests focus on the two most important aspects of our solution: the performance of the data structure, especially for large data sets, and the effects of our levels of detail solution, both visually and in terms of performance. Additionally, we take a look at the impact of the data set structure on neighborhood based algorithms and the parameters of the enhancement methods.

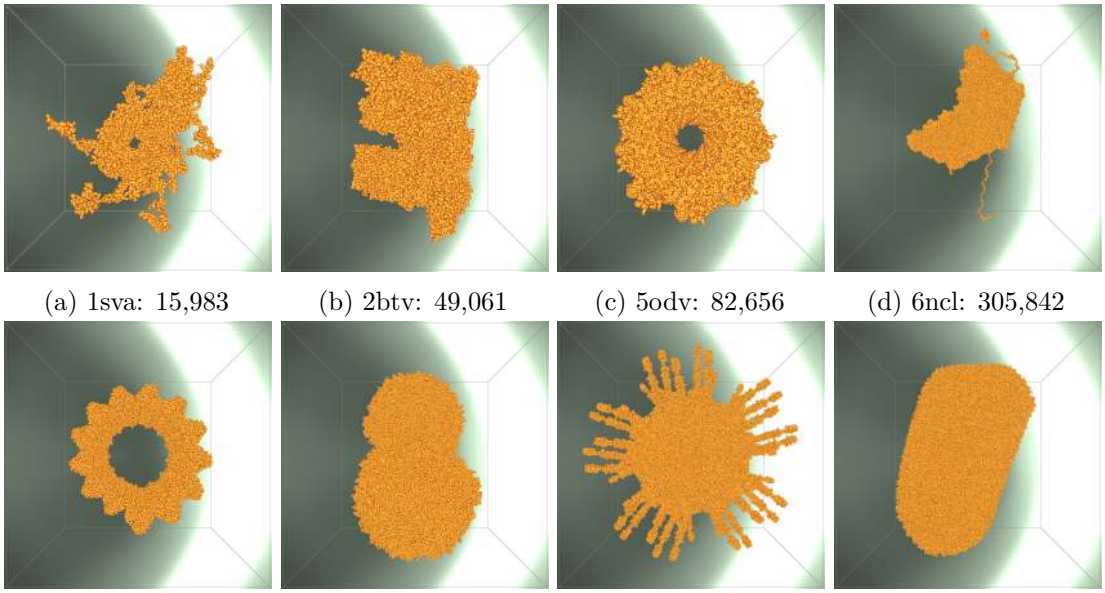
All tests results are generated using an NVIDIA GeForce GTX 960M/PCIe/SSE2 graphics card and an Intel Core i7-7500U @ 2.70 GHz with 16 GB RAM. Everything is rendered in full HD (1920 by 1080 pixels).

6.1 Test Data

In order to test our framework in different situations and investigate how well our implementation scales, we put together a small collection of test data. Our main set consists of eight different-sized molecules, shown in Figure 6.1. The largest molecule currently found in the PDB database is 3j3q, which represents the structure of an HIV-1 capsid. It contains 2,440,800 atoms. To test scalability beyond that, we use two artificially created molecular data sets consisting of 10 and 50 instances of 3j3q, containing 24,408,000 and 122,040,000 atoms, respectively.

6.2 Configuring and Pre-processing the Data Structure

Our implementation focuses on rendering a single time-step. Building the octree and calculating levels of detail is done in a pre-processing step. This speeds up rendering, at the cost of a longer start-up time. In this section, we look at how pre-processing computation scales for different-sized molecules and the impact of data structure configurations.



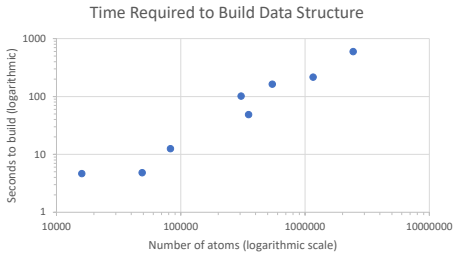
(a) 1sva: 15,983 (b) 2btv: 49,061 (c) 5odv: 82,656 (d) 6ncl: 305,842

(e) 6o2s: 351,728 (f) 4v99: 546,120 (g) 6qz0: 1,161,565 (h) 3j3q: 2,440,800

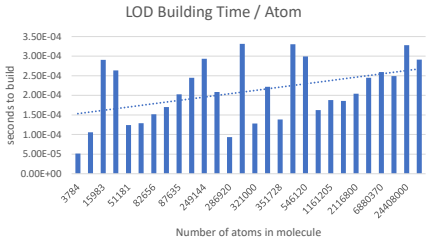
Figure 6.1: Atoms in our test set with the number of atoms they contain

The time it takes to build the octree data structure including levels of detail increases linearly with the number of atoms in the data set, as shown in Figure 6.2a. There are several reasons why one might have to calculate data sets that are larger than the largest single molecule currently available in the PDB database: the database continues to be extended and new molecules added, one may want to render entire cells, or molecular dynamics simulations with several time-steps. Therefore, we look at how the construction of the data structure scales for larger data sets. In Figure 6.2b, we show the time required to build the data set per atom for 26 molecules, and two artificial large data sets. Here, we see that the time required to calculate the data structure including levels of detail actually increases on a per-atom basis. This is due to the overhead of sorting atoms into pages. However, it has to be noted that a simple nearest neighbor algorithm that checks every point in the data set to find neighbors has an exponential execution time. By sorting the atoms into pages with a maximum size, the maximum number of neighbors an atom needs to be compared to is $2^{15} - 1$, even for very large data sets, which results in a linear increase in the time it takes to compute the LOD data structure. The same advantage can be leveraged when calculating any other type of neighborhood-based algorithm. We sill have to sort all atoms and traverse nested pages, which is why the time does increase per atom, but because of the division of the limit to the number of neighborhood queries, the total execution time increases linearly, not exponentially.

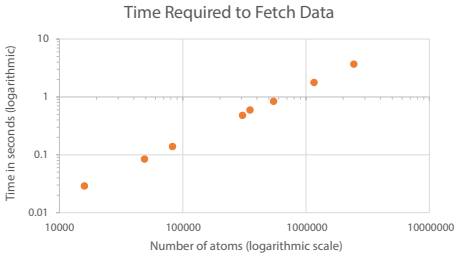
Figure 6.2c shows that the time it takes to load a pre-processed data set also increases linearly with size. However, the overall time required to load a pre-processed data set is negligible compared to building it from scratch each time a data set is loaded. As we



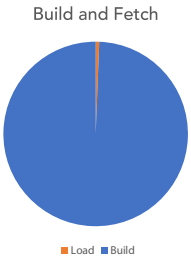
(a) The time it takes to build the data structure increases exponentially with the number of atoms in the data set.



(b) As LOD creation requires neighborhood queries, the time it takes to build them also increases on a per-atom basis.



(c) The time it takes to load pre-processed data also increases exponentially with the size of the data set.

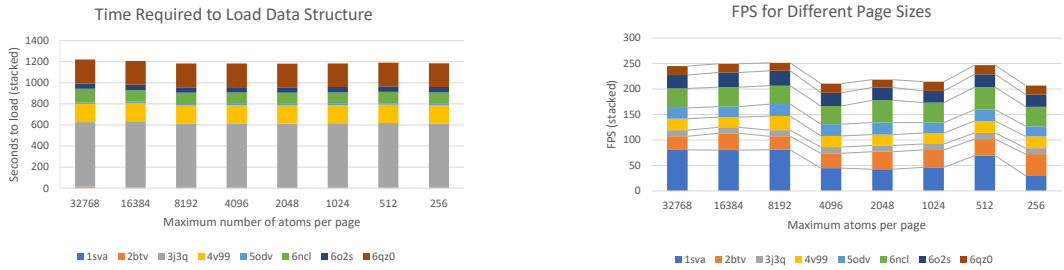


(d) Pre-processed data can be loaded very quickly compared to building the data structure from scratch.

Figure 6.2: Both building and loading times increase exponentially with the size of the data set. As loading only takes a small fraction of the time required to build them, it is well worth saving the data structures.

assume that the same data sets are going to be loaded and viewed several times, it is therefore worthwhile to store the pre-processed data sets. Figure 6.2d shows the time it takes to load a data set relative to the amount of time required to build the same data set. The figure is based on the same data as Figures 6.2a and 6.2c. As building the data structure is a significant overhead for large data structures, we save the data structure into a *.lod, once it is completed.

While the most significant factor determining the building, loading, and rendering times is the size of the data set, page sizes can also have an impact. We compare page sizes ranging from the maximum possible atom count per page of 32,768, to a minimum of just 256 atoms per page. The results are shown in Figure 6.3. The time it takes to load the



(a) The size of pages has very little impact on the time it takes to load a data set

(b) On average, larger pages result in higher frame rates

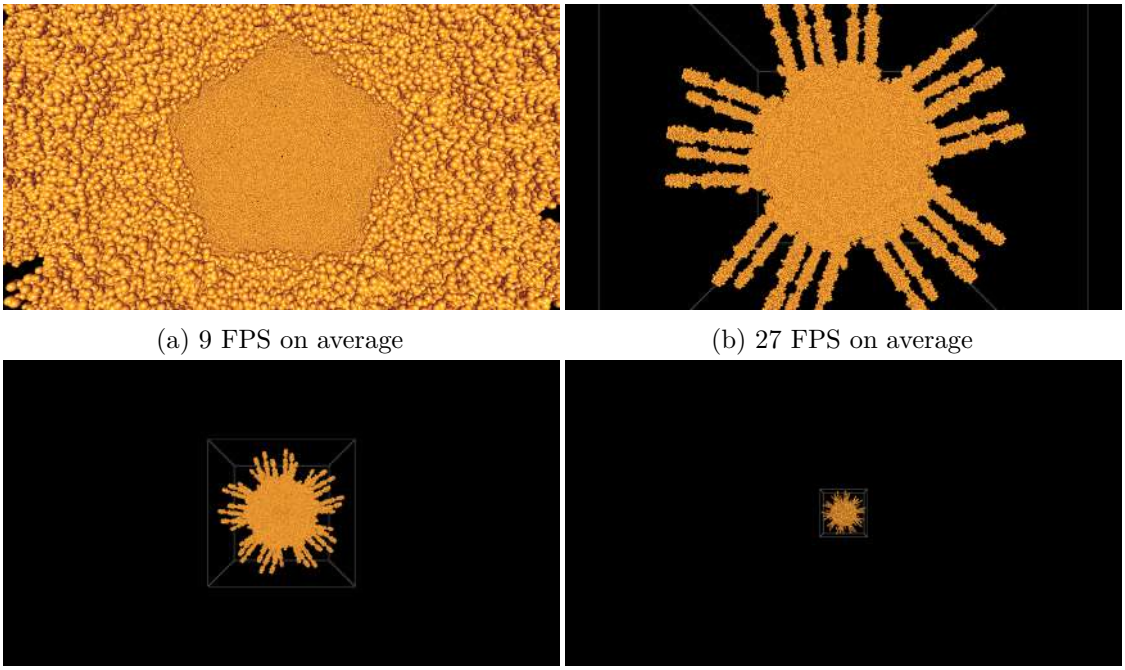
Figure 6.3: When comparing loading time and FPS, we found that larger pages tend to perform slightly better.

data structure remains virtually the same for different page sizes. When looking closely, it becomes visible that the page sizes that result in lower frame rates, are very slightly faster when it comes to loading the data sets. However, the effect is negligible. Our system is also optimized for runtime, rather than start-up time, so the more interesting results are changes in frame rates. We show the average FPS using different page sizes on a set of molecules in Figure 6.3b. While the exact result is also influenced by the structure of the molecule, larger page sizes perform better in most cases. For our remaining tests, we keep the maximum number of atoms per page set to 32,768 (2^{15}).

6.3 Rendering

The main objective of our framework is to render large molecules at interactive frame rates, while keeping the quality of visual enhancement and the level of detail as high as possible.

While frame rates are the most natural quantitative factor when analyzing rendering frameworks, they vary strongly depending on various influences. Apart from hardware and implementation overhead, which are usually constant when comparing data sets within a single framework, the screen fill rate is highly significant. Figure 6.4 shows the same molecule, using the same settings for enhancement methods, etc. at different zoom levels, resulting in a different screen fill rate. In this case, the frame rate lies between 9 and 74 FPS, mostly due to the ray casting implementation. For frame rate based performance tests across data sets, we therefore use a standardized position, where the molecule is centered and scaled in such a way that its bounding box is maximized relative to the viewpoint when using an orthographic camera. The size of the viewpoint in our tests is full HD, that is 1920 by 1080 pixels, and we use a perspective camera with 60



(a) 9 FPS on average

(b) 27 FPS on average

(c) 59 FPS on average

(d) 74 FPS on average

Figure 6.4: Different screen filling (6qz0)

degrees field of view. As a standard setup, all tests are conducted using the positions shown in Figure 6.1. Additionally, we use a Gaussian surface with a sharpness of 16 (see section 6.4.3), but no depth blur or ambient occlusion, unless specified otherwise.

6.3.1 Level of Detail Rendering

The main motivation for using levels of detail, is to reduce the amount of data that has to be rendered in order to achieve performance improvement, as well as making it possible to render data sets that are too large to be handled by a specific set of hardware. In this section, we show how much the number of spheres is reduced for each level, and how much the frame rate increases. We also visually investigate the the perceived loss of detail for coarser levels of detail.

Table 6.1 shows the total number of atoms per level of detail for our test sets. The corresponding screenshots are shown in Figure 6.5. The last column in the table shows the number of pages into which the data is sorted, which also represent the last, coarsest level of detail, where we use a single sphere per page, which is not shown in the figure. Because we use two buffers, with the secondary one coarser than the primary buffer, this last level cannot be chosen as basic level. The numbers in the table are visualized in Figure 6.6. In Figure 6.6a, we show the total amount of data points per molecule and LOD, providing an overview over relative sizes. As that makes it had to see the details

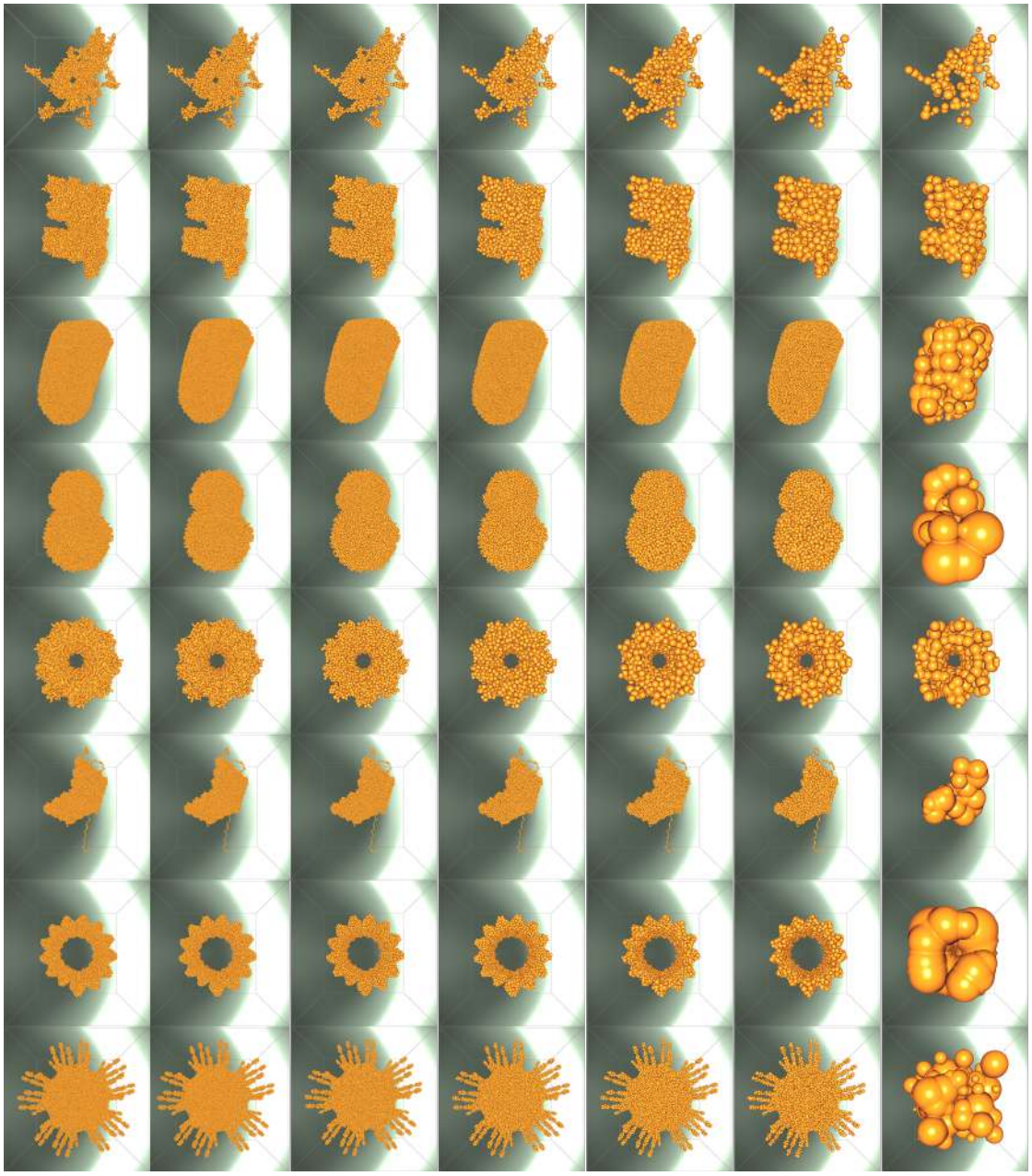
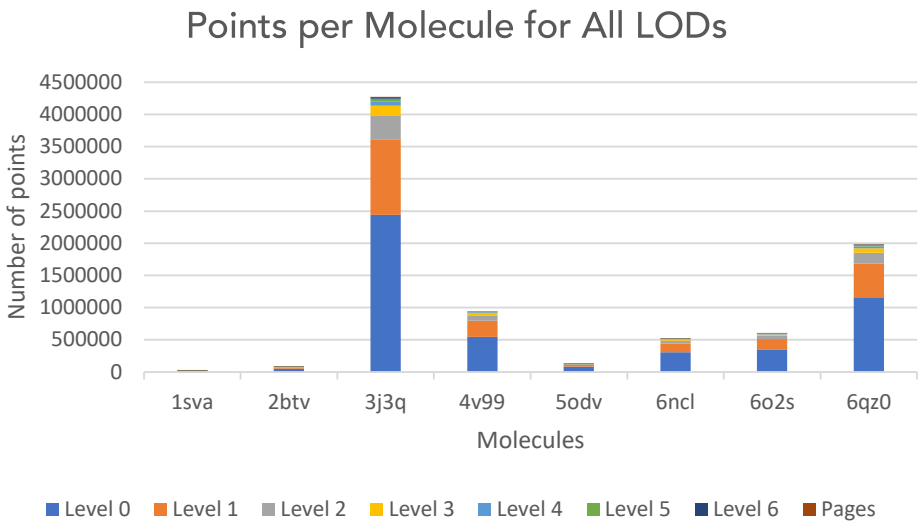
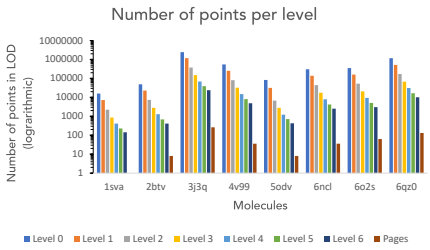


Figure 6.5: LODs (top to bottom: 1sva, 2btv, 3j3q, 4v99, 5odv, 6ncl, 6o2s, 6qz0)

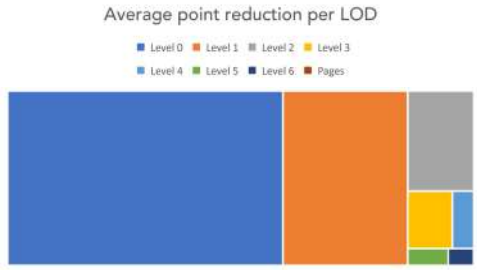
for smaller data sets and coarse LODs, Figure 6.6b shows the amount of points per level next to each other and on a logarithmic scale. Figure 6.6c shows the average amount of points per LOD relative to other LODs. While the exact reduction depends on the structure of the molecule, the average reduction between neighboring LODs is between 40 and 60 percent, depending on the structure and density of the molecule.



(a) Total amount of points in the test molecules across LODs.



(b) Amout of points separated by level and shown in logarithmic scale for easier comparison.



(c) The amount by why the number of atoms is reduced per LOD depends on the structure of the molecule. Generally, points are reduced by about 40-60 percent.

Figure 6.6: Comparing the relative and absolute amount of points per LOD for our test data sets.

Name	Level 0	Level 1	Level 2	Level 3	Level 4	Level 5	Level 6	Pages
1sva	15983	7240	2205	853	409	228	142	1
2btv	49061	22508	7315	2812	1282	689	409	8
3j3q	2440800	1171809	375594	148123	68567	38951	24011	260
4v99	546120	251943	80547	32012	14882	7945	4870	36
5odv	82656	31540	6760	2777	1216	705	428	8
6ncl	305842	138484	44268	17337	7901	4180	2493	36
6o2s	351728	158129	52977	20600	9407	5094	3045	63
6qz0	1161565	520817	171045	67621	30994	16604	10003	131

Table 6.1: Numbers of atoms per level of detail

Our general rendering test for all levels, shown in Figure 6.7, confirms that all molecules currently available in the Protein Data Base can be rendered at interactive rates. The maximum amount of atoms is set to the highest possible level of 32,768 atoms per page. Frame rates for the standard views range from 27 to 84 in the original resolution. As we want our framework to support even larger structures, as well as more complicated enhancement methods, the jump in performance between level 0 and 1 for 3j3q, which contains approximately 2.4 million atoms, from 27 to 55 FPS is particularly interesting. In section 6.3.4, we investigate artificial data sets based on several instances of that molecule.

6.3.2 Caching and Per-page Rendering

Our framework has the capability to render the data structure per page, using a list of visible pages, or to render everything contained within a single buffer at once. In Figure 6.8a, we compare the two rendering methods. In both cases, the frame rates were measured with the molecules in the position shown in Figure 6.1, with the entire molecule in view. For smaller molecules that contain few pages, it does not make a difference whether we render per-page or the entire buffer, but for larger molecules, rendering per page is more effective in our framework. Additionally, features such as blending between levels of detail rely on per-page offsets, which means that if we want to use them, we need to render per page. Our standard method is optimized for per-page rendering, as the buffer is divided into blocks when uploading per page, which leaves empty space (see Figure 6.8b). In order to verify that in our framework, per-page rendering for large data sets is also faster than rendering a full, dense buffer, we compare frame rates for all test sets at the original resolution. As discussed in Section 6.3.4, we also need to divide the data set at some point, when, depending on the hardware, the molecule no longer fits into memory in its entirety.For the remaining tests shown in this chapter, we render blocks of data corresponding to pages individually.

Figure 6.9 demonstrates that only the pages that are actually needed are uploaded. In screenshot 6.9a, the entire molecule 4v99, divided into 36 pages, is loaded at the

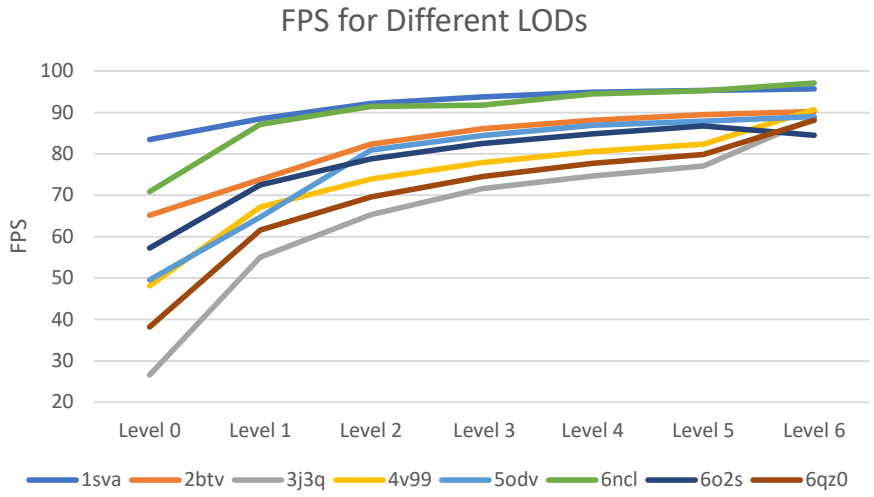
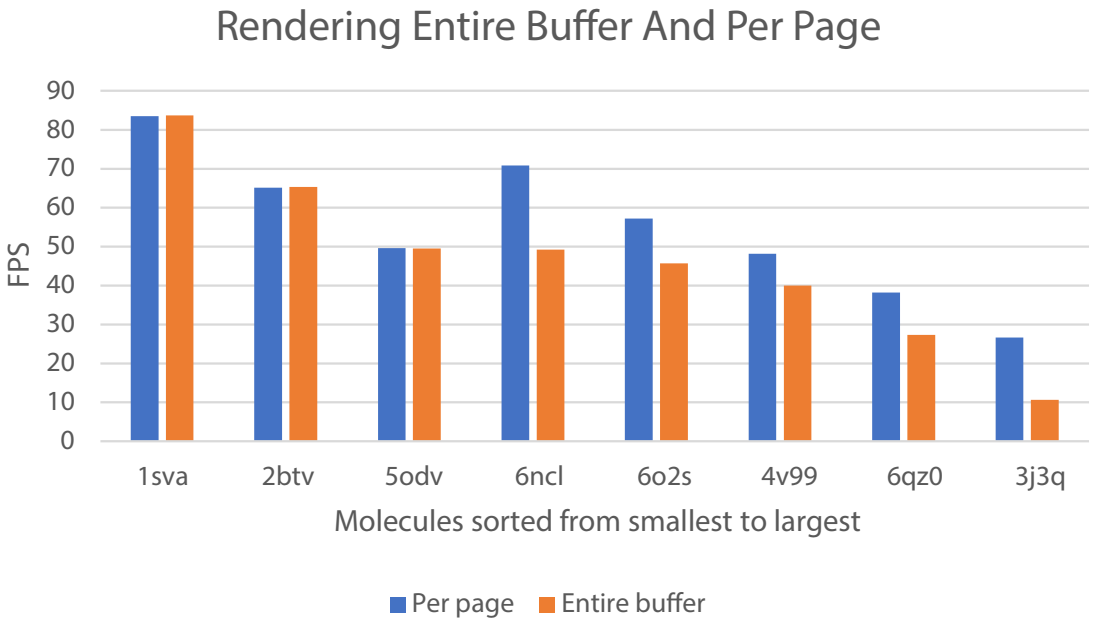


Figure 6.7: Frame rates achieved for the data sets at different levels

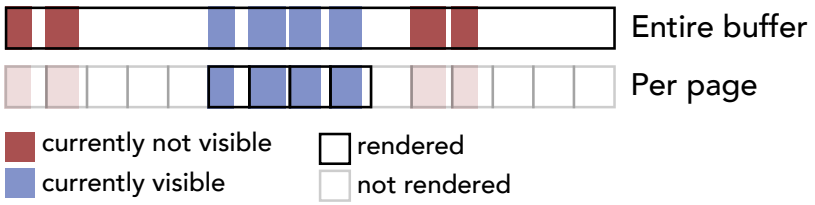
original resolution. The colors represent their order in the cache, from oldest (red) to newest (green). Usually, missing pages are automatically loaded when the perspective changes and they become visible. In order to demonstrate that only pages that are at least partially visible are uploaded to the buffer, we first zoomed in to a position where only part of the data set is visible. Then, we cleared the buffer and disabled cache updating. After that, we zoomed out again to view the entire molecule, without allowing the program to load missing pages. Figure 6.9b shows which pages were already loaded in the previous view, as well as those that are missing, which are enclosed by gray bounding boxes. In order to observe this, we have to manually clear the cache, as pages that become invisible are not automatically deleted, unless the cache runs out of space.

6.3.3 Selecting the Level of Detail

We want to use levels of detail in a way that increases performance as much as possible or necessary, while keeping the required visual quality. Partially, visual quality can be parameterized and automated, but the user has to have to option of choosing or at least influencing the visual result, as the requirements may differ according to the task the user wishes to perform. We provide a set of options that combine manual input and automatic calculation to control the level of detail.

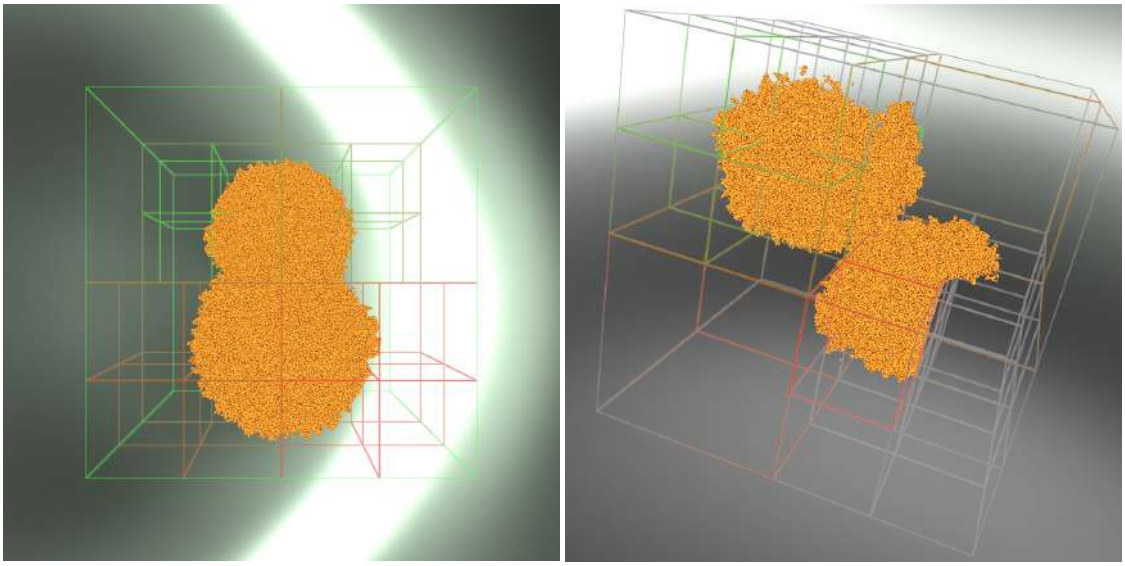


(a) Per page rendering compared to rendering the entire buffer for all molecules at level 0



(b) The buffer is divided into page blocks, which can be drawn individually.

Figure 6.8: Rendering the entire buffer vs. page blocks



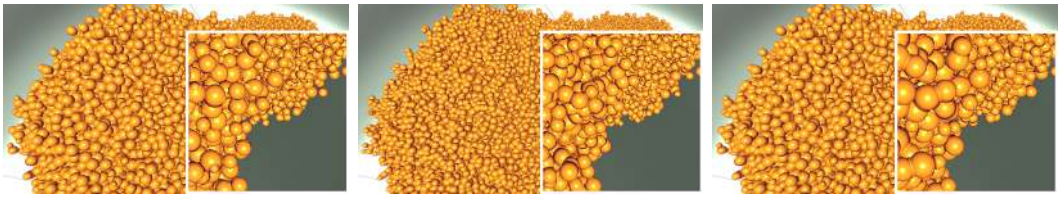
(a) All 36 pages, level 0 (b) Pages are loaded when they are needed.

Figure 6.9: Page replacement (molecule 4v99). Only visible blocks were loaded at a previous position. Stopping the buffer from updating allows us to interact with the molecule and look at which pages were not needed in the previous view.

Manual LOD Selection

The user can set a target frame rate in the interface. When the user chooses a level in the interface, the basic level of detail is changed. This means that the whole buffer is cleared and divided into new chunks, based on the maximum size of atoms per page for the newly set level of detail. For small and medium sized molecules, including all examples in our test set, this task does not cause a visible delay. For very large data sets, such as the examples discussed in section 2.3, it can take several seconds or more. However, very large data sets are the main application for this option. If the user wants to interactively study the entire structure of a data set that is to large to fit into memory at the original resolution, they can choose to set a lower basic rendering level, which allows all pages to fit into memory at the same time.

The smooth transition slider changes the rendering level. The buffer partition is left as it is, but whenever a page becomes visible, the data of the selected rendering level is uploaded instead of the current basic level. The slider starts at the basic level of detail and goes up to the highest level (lowest resolution) available in the system. For example, if the basic level is set to 1, the rendering level cannot be a finer resolution, but starts at 1, as the points contained in a single page at level 0 do not necessarily fit into the buffer slots based on the maximum size of level 1. The slider allows for a smooth transition between all levels, starting with the basic level. If the slider is for example set to 2.5, the molecule will be rendered somewhere between level 2 and 3, as shown in Figure 6.10.



(a) Level 2 (b) Level 2.5 (c) 3

Figure 6.10: Smooth transition (molecule 4v99). We calculate a finite number of discrete levels of detail, but we can interpolate all steps in between them.

Frame Rate Dependent LOD Selection

The timeout window of 10 seconds before the level is reduced to the next lower resolution is based on the observed amount of time in which the frame rate stabilizes in most cases. This mode only lowers the resolution, it does not automatically increase to a finer resolution, even if the target frame rate allows it. Though automatically returning to a better resolution might be convenient in some situations, we prioritize avoiding frequent jumps between levels, which slow the rendering down. This mode also changes the render level, not the basic level.

View Dependent LOD Selection and Interpolation

The idea behind the view dependent level of detail rendering is to reduce the number of atoms towards the back of the scene, as seen from the camera. They are rendered smaller in a perspective projection, and are more likely to be covered by other atoms. Figure 6.11 shows the use of view dependent LODs. In this case, the interpolation is between level 0, the original data, and level 2. Data points close to the camera are rendered at level 0, with a gradual transition to level 1, the pages in the back have level 2 as their rendering level and gradually transition to level 3. While a clear difference is visible between the original resolution shown in 6.11a, and level 1 next to it (6.11b), it is hard to see the difference between original data 6.11a, and the mixed levels 6.11c. The average frame rate increases, as less spheres have to be rendered, while the data points in the foreground remain the same. Figure 6.11d shows where level 0 is used (orange) and where level 1 (violet) is used.

6.3.4 Rendering Very Large Data Sets

In order to test our solution beyond the molecules currently available in the Protein Data Bank, we use two artificial data sets. Data set 1, shown in Figure 6.12b consists of 10 instances of 3j3q, the largest molecule currently available in the PDB. It approximately represents the maximum data set size we can fit into memory as a whole using our test hardware. The exact number of data points that can be uploaded to the GPU not only depends on the hardware itself, but also on whether or not any other processes are using part of it at the same time. Figure 6.12a shows data set 2, consisting of 50 instances

Level	Max pages	Max atoms per page
0	11071	32750
1	25802	15740
2	76011	5067
3	183294	1996
4	365466	933
5	612167	536
6	1569797	341

Table 6.2: Numbers of atoms per level of detail

View	Active pages	Basic level	Render level	FPS	Frozen	Rendering
1	12886	6	6	39	no	page
2	9471	2	2	7	no	page
3	9471	2	4	23	no	page
4	1743	0	0	5	no	page
5	2934	1	1	4	yes	buffer
6	13729 (all)	3	4	22	no	page

Table 6.3: Data set at different views and LODs

of 3j3q. In total, this example contains 122,040,000 atoms, which we cannot load into a buffer at the same time. Instead, it is rendered at level 2 in the screenshot, which is the maximum resolution at which we can show all its pages at the same time. The hierarchical data structure is pre-computed using the standard maximum page size of 2^{15}.

We conduct a number of tests on the special case of a data set that does not fit into the buffer at full resolution on our system. Figure 6.13 contains 6 views at various resolutions of test data set 2 (Figure 6.12a). Table 6.2 gives an overview over the maximum number of pages that fit into the buffer at the same time using the capacity of our hardware. The right hand column shows the maximum number of atoms per page for each level of detail, while the second column shows how many pages could fit into the buffer at the given level. The number of pages in this data set when dividing it into pages of up to 2^{15} atoms per page, is 13,729. Table 6.3 summarizes the configuration details of the individual views shown in Figure 6.13. The column "active pages" refers to how many pages are in the current render list. The render list contains the pages that are at least partially visible in the current frame, unless the buffer is frozen, stopping it and the render list from being updated. The basic level of detail is the level that determines buffer partition, while the render level refers to the resolution at which the data points are actually rendered. The last two columns indicate if the buffer is frozen, and whether we draw the scene page by page, or render the entire buffer.

At startup, we load all pages in the data set. If it exceeds the hardware capacity, the basic level is set to 6. View 1 in Figure 6.13a shows this initial setup. In this view, 12,886 of 13,729 pages are active, i.e. at least partially within the view frustum. At 39 FPS, the frame rate is interactive, but the reduced resolution is clearly visible at level 6. View 2 in Figure 6.13b shows a partial view where about 69% of pages are active, though that does not necessarily correspond to 69% of the actual data being visible. Here, the basic rendering level is set to 2, the highest resolution at which we can view the entire data set. View 3 in Figure 6.13d is identical to Figure 6.13b, except that the rendering level (but not the basic level) is set to 4, which more than triples the frame rate. View 4 in Figure 6.13d shows part of the data set at the original resolution. In order to view part of the data at a higher resolution than is possible for the entire data set at once, the user has to zoom in to the section they would like to view in detail first, and then set the level to the desired resolution. This limits the cache to the maximum amount of pages that can fit into the buffer at that level. When the user moves the camera, pages are replaced using the LRU principle. If the camera is moved in such a way that more pages become active than fit into the cache, the remaining pages are ignored for that frame. We find this behavior preferable to automatically reducing the level of detail, as it is easy to accidentally zoom out too much, thereby triggering an undesired level change. Deleting and restructuring the buffer causes a delay, especially for very large data sets. Switching between basic levels, i.e. restructuring the buffer, should be avoided if it is not what the user actually intends. View 5 in Figure 6.13e shows a rendering of the entire buffer, while it is frozen. As discussed in Figure 6.9, data that remains in the buffer, even though it is not needed, is only visible if the entire buffer is rendered at once, rather than per-page. In this case, we see the part of the data set that we were viewing when the buffer was frozen on the right side. On the left, we see left over data from a previous view. The final view 6 (Figure 6.13f) shows the entire data set with the basic level of detail set to 3, and the rendering level set to 4, which results in 22 FPS, a frame rate usable for interaction.

6.4 Enhancement Methods

In this section, we demonstrate the enhancement methods included in our framework. The included methods are in screen space, which means that they are applied on top of a primary rendering pass. We mainly focus on visual results, and the impact parameter settings have on frame rates.

6.4.1 Blur-based Enhancement Methods for Ambient Occlusion and Edge Enhancement

We summarize different effects based on screen space blurring in Figure 6.14. In all images in the figure, a Gaussian surface with a sharpness of 16 is used. For reference, Figures 6.14a and 6.14b show the molecule with no effects, and the HBAO+ library respectively. The effects based on a Gaussian blur pass are shown in Figures 6.14c, 6.14d, and 6.14e.

View	Method	Kernel/passes	Lambda	Sigma
a	none	-	-	-
b	library	-	-	-
c	gaussian	2	10	20
d	gaussian	10	20	20
e	gaussian	10	18	2
f	library + gauss	4	20	10
g	halo	2	21	-
h	halo	4	50	-
i	halo	64	2	-
j	library + halo	2	50	-

Table 6.4: Rendering details for Figure 6.14

Figures 6.14g, 6.14h, 6.14i show the effects based on halo blurring. Figures 6.14f and 6.14j show a combination off the effect produced by the library, and Gaussian- or halo shader. In general, the effect that can be achieved by the two methods is rather similar. It works effectively for edge enhancement of different thicknesses, depending on the strength of the blur and the parameters. The strength of the blur is influenced by the size of the kernel, or number of passes, depending on the method (see for example 6.14c vs 6.14c), but also the parameters lambda and sigma. Figure 6.14e looks very similar to Figure 6.14c, except for a stronger enhancement of lines around the molecule compared to internal borders, even though it uses the same kernel size as Figure 6.14e. The internal structures enhanced by the library (Figure 6.14b) are hardly visible in Figure 6.14a. When applying halo blurring with a large number of passes, as in Figure 6.14i, where 64 pixels are sampled around each point, the shader achieves an ambient occlusion effect. Compared to the ambient occlusion library, it has the advantage of exclusively darkening deeper structures at the center of the molecule, making them more clearly visible. However, the darkening around the edge of the molecule, where the contrast caused by blurring is strongest, hides details in that area. In conclusion, the blurring based enhancement shaders would have to be developed further in order to sensibly replace the library for ambient occlusion blurring, but they do work well for edge enhancement. We also found a combination of edge enhancement, and blur shaders, as demonstrated in Figures 6.14f and 6.14j, useful to investigate larger and smaller structures within the molecule at the same time.

6.4.2 Screen-space Depth of Field Effect

The depth of field effect can be used to draw attention to specific areas. Its parameters work like the settings of a camera. The user can set the focal length, and the F-stop. Additionally, the maximum circle of confusion (CoC) radius can be set. Figure 6.15 and table 6.5 show different settings for the depth of field shading. From one view to the

View	Focal distance	F-stop	Max COC	Effect
a	-	-	-	-
b	1.4	1.2	6	-
c	1.4	11	6	-
d	4.9	1.2	6	-
e	1.4	1.2	1.5	-
f	1.4	1.2	14	-
g	1.4	1.2	6	HBAO+
h	1.4	1.2	6	Gauss (4/20/9)

Table 6.5: Rendering details for Figure 6.14

next, only a single parameter is changed. The focal distance determines where, relative to the camera, the focus is, while the F-stop determines how deep the area in focus is. On an optical camera, a low value for the F-stop means an open aperture, which lets more light in, but also scatters the light rays, so the area of focus is narrower, which we reflect in our blurring formula. The maximum circle of confusion radius determines the strength of the blur in the blurred areas. A larger radius means that more neighboring pixels have to be considered when blurring. Therefore, this is the only parameter that directly affects the frame rate. It is also possible to combine the effect with ambient occlusion (Figure 6.15g), or edge enhancement (Figure 6.15h). In that case, the render passes for both effects are executed consecutively.

6.4.3 Rendering the Gaussian Surface Model

Though the Gaussian Surface is a less computationally expensive alternative to the full Solvent Excluded Surface, it is the most expensive of the methods we currently provide in our framework. The dominant factor for its performance, is the extent of the surface, as that factor is responsible for the length of the intersection list that has to be checked for each ray. We use the method proposed by Bruckner [Bru19]. A detailed performance analysis of the algorithm itself, is outside the scope of this thesis. In this section, we analyze the method in the context of our work.

Figure 6.16 shows the Gaussian surface with different sharpness parameters, and the resulting frame rate. For the same view and settings, the frame rates range from 29 to 3 FPS. In most use cases, the goal of rendering a Gaussian surface is to approximate an SES, so the sharpness value of 0.5 may not be a particularly realistic case. However, it does demonstrate how dependent the rendering performance is on the parameter of the Gaussian surface. The method as proposed by Bruckner [Bru19], is output sensitive, so its performance depends on its own parameters, and the number of spheres in the view frustum for the current frame, rather than the total amount of atoms in the molecule. The only difference between images in Figure 6.16 is the sharpness factor. This parameter influences the size of the neighborhood that is considered, when determining overlapping

spheres, and thereby the amount of spheres in the intersection list.

As we use a screen space method to build the molecular surface, the data structure used on the CPU is not directly relevant for its performance. The only part of our hierarchical data structure that influences it directly, both in terms of visual results and frame rates, are levels of detail. Figure 6.17 shows three versions of molecule 5odv, using a sharpness factor of 2.5. The images in the top row show the entire screenshot, the row below a cropped out detail. Figures 6.17a and 6.17d show the molecule at the original resolution, without any optimization. The resulting frame rate is 12 FPS. In Figures 6.17b and 6.17e, the extent of the molecular surface is attenuated, based on the camera distance with a threshold of 113, and a tolerance of 87, resulting in a frame rate of 15.5 FPS. For the last view, 6.17c and 6.17f, we use both distance based attenuation, and distance based LOD, with a threshold of 173, and a tolerance 17, and achieve a frame rate of 21.5 FPS. While there are some visible differences, especially between detail view 6.17d and 6.17f, users can set the parameters themselves, and in some cases, a slight loss of detail could well be considered acceptable, given that in this case, the frame rate is almost doubled.

6.5 Conclusion and Comparison to Similar Solutions

Due to variations in hardware and a lack of consistent standards, direct comparisons of measurements such as frames per second, are often not meaningful. As mentioned above, frame rates also depend on other factors, notably the perspective, so even a rendering of the same molecule using the same hardware cannot necessarily be compared directly. Therefore, we mention noteworthy differences to a selection of closely related recent work, and try to give a general comparative overview, rather than presenting a direct comparison of frame rates.

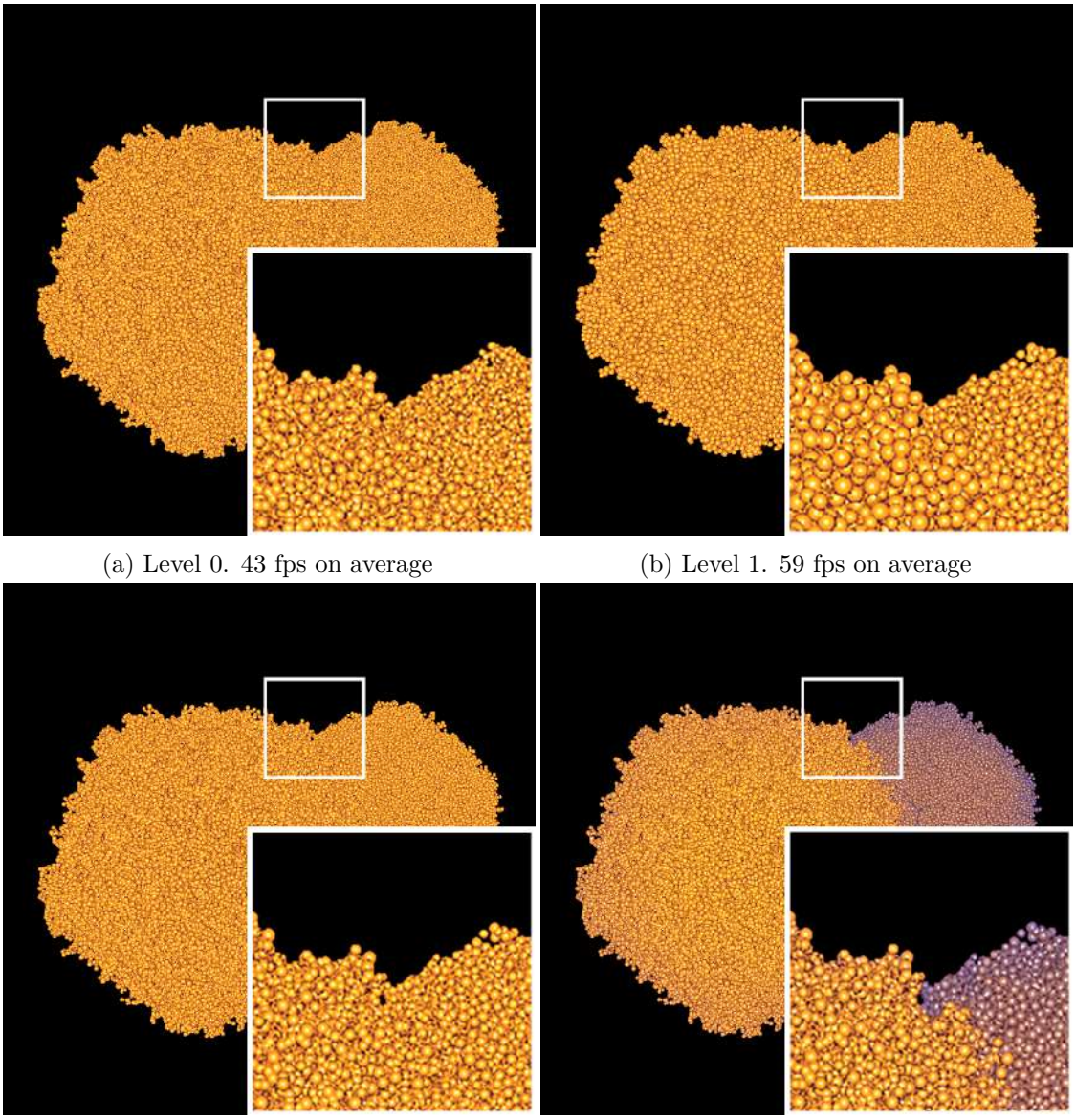
Grottel et al. [GRDE10] use a combination of optimization strategies that include caching, two-level culling, and deferred shading. Figure 6.18a shows the results they present in their paper, for data sets ranging from 107,391 to 100,000,000 points. The main difference to our work is that they do not use a hierarchical data structure. They focus on molecular dynamics simulations, and therefore find the overhead of creating it prohibitive. As our main optimization strategy relies on levels of detail that have to be pre-calculated, a hierarchical data structure is more suitable, and the overhead negligible in our case. Sharma et al. [SKNV04] do use an octree, and levels of detail. However, their octree is limited to three levels, and they use polygons to render their spheres. Levels of detail in that case refer to the resolution ot the sphere, rather than the number of spheres. Their framework manages to render very large data sets (see Figure 6.18b), especially given that their research was already published in 2004. However, they only render points as spheres, and do not include or discuss molecular surface models or enhancement methods. The advantage of their LOD solution compared to ours is that it can be determined on the fly. Our method has the advantage of actually reducing the number of spheres that have to even be considered for culling. Figure 6.18b shows the results achieved in their framework, Atomsviewer, without optimization, using the octree, and a parallel and

distributed version. Their work is a standard to which most other relevant works refer. Contrary to many solutions proposed afterwards, we return to a similar data structure, while using more recent techniques when it comes to rendering.

Parulek et al. [PJR+14] on the other hand focus their work on molecular surfaces. The largest atom size they test is, at 58.674 atoms, relatively small compared to other implementations that include tests up to billions of atoms. However, they present a more advanced surface model than any of the other closely related implementations we found. According to their results, their mixed model, which includes levels of detail and three different surface models, achieves up to 20x the frame rate compared to a full SES representation. Our optimizations do not boost rendering as much, but our system does not (yet) include the more computationally expensive SES, so direct comparisons are not possible. We do, however, use the same clustering method to build our levels of detail. Our surface representation setup is not as advanced as theirs, but we do provide a data structure that is capable of rendering much larger data sets, which could be extended to include such a visualization model.

The work done by Matthews et al. [MEK+17] is similar to ours in that they also focus on a single instance, and investigate the use of data structures to boost performance. They employ a grid-based data structure. The main difference is that they focus on molecular trajectories, so they avoid pre-computations. They achieve interactive results while constructing the data structure on the fly, including ambient occlusion enhancement. However, the largest data set they render in their tests consists of 316,404 atoms. They do not include levels of detail, as the required clustering is computationally expensive. For molecular dynamics or animations, their work is more performant and suitable than ours, but it is much more limited in the sizes it is capable of handling.

A lot of recent work in molecular visualization focuses on instance based rendering (Guo et al. [GNL+15], Le Muzic et al. [LMPSV14], Falk et al. [FKE13], Lindow et al. [LBH12]). They achieve very high numbers when it comes to the absolute results in terms of how many millions or billions of atoms the framework can render. However, this approach is limited to very large data sets that do in fact contain repetitive structures. Lindow et al. [LBH12] mention in their future work section that they recommend the inclusion of LODs and occlusion culling in their framework, which Falk et al. [FKE13] provide. Guo et al. [GNL+15] use a very similar type of hierarchical clustering to ours in order to build levels of detail. They prove that it can be used with instance rendering. None of the given examples use an octree data structure which we argue is more convenient, because it divides space more equally in terms of density. As the papers only provide results for the entire scenes, containing up to millions of repeating instances, frame rates are hard to compare. Our system scales well for large non-repetitive structures well beyond the scale of the basic molecules used in testing the instance based rendering systems. In our opinion, it would however be worthwhile to additionally extend our framework to be able to handle repetitive structures.



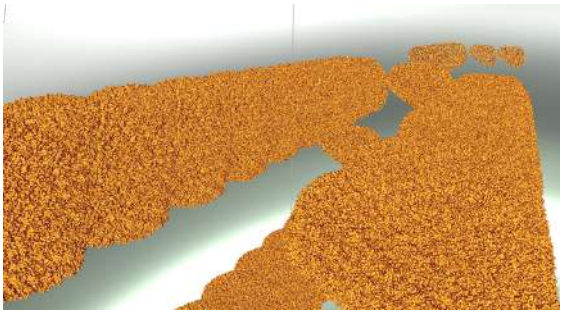
(a) Level 0. 43 fps on average

(b) Level 1. 59 fps on average

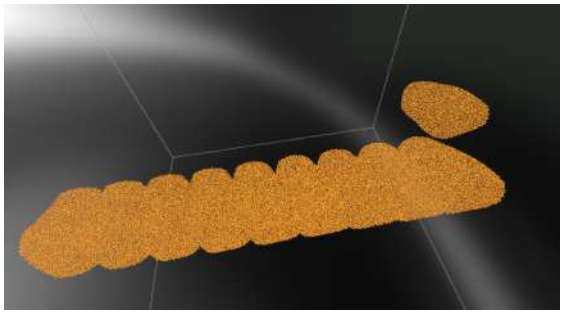
(c) Mixed. 51 fps on average

(d) Interpolation. Yellow: level 0, violet: level 2

Figure 6.11: View dependent level of detail (molecule 4v99). The user can set a distance from the camera and threshold. The resolution of the molecule is reduced in areas far from the current camera position, saving rendering time.

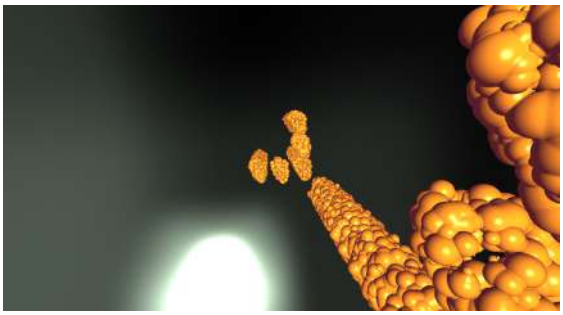


(a) 50 instances of 3j3q at level 2

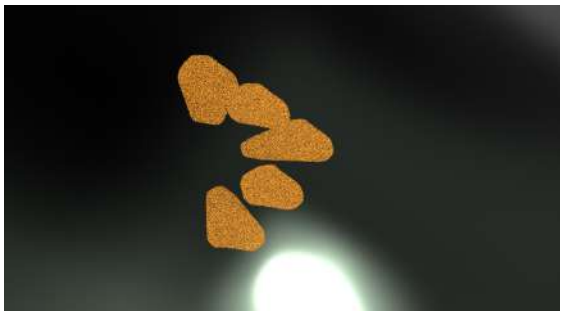


(b) 10 instances of 3j3q, original resolution

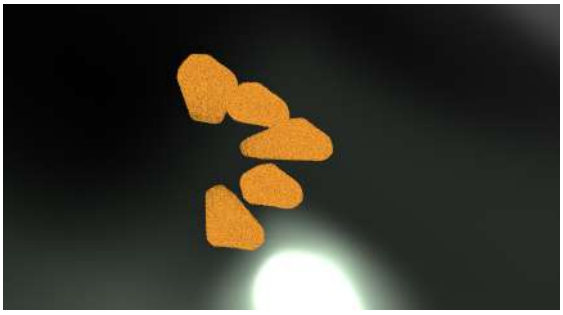
Figure 6.12: Large test data sets



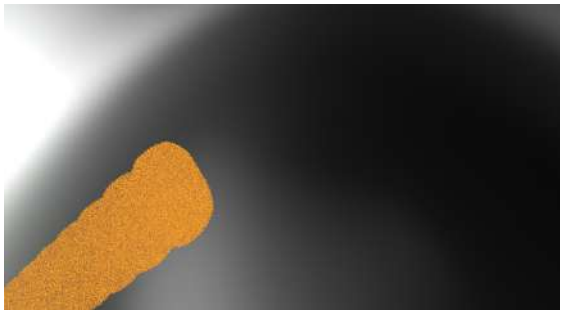
(a) View 1



(b) View 2

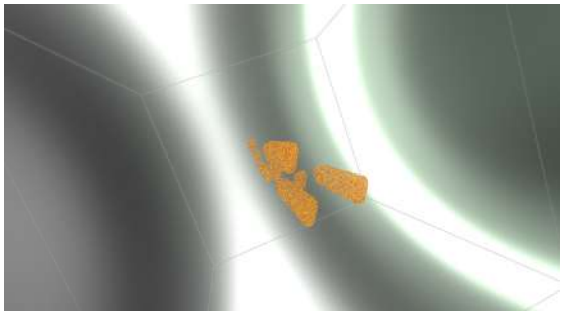


(c) View 3



(d) View 4

(e) View 5



(f) View 6

Figure 6.13: Artificial data set with 50 instances of 3j3q

(a) No effects (b) Library

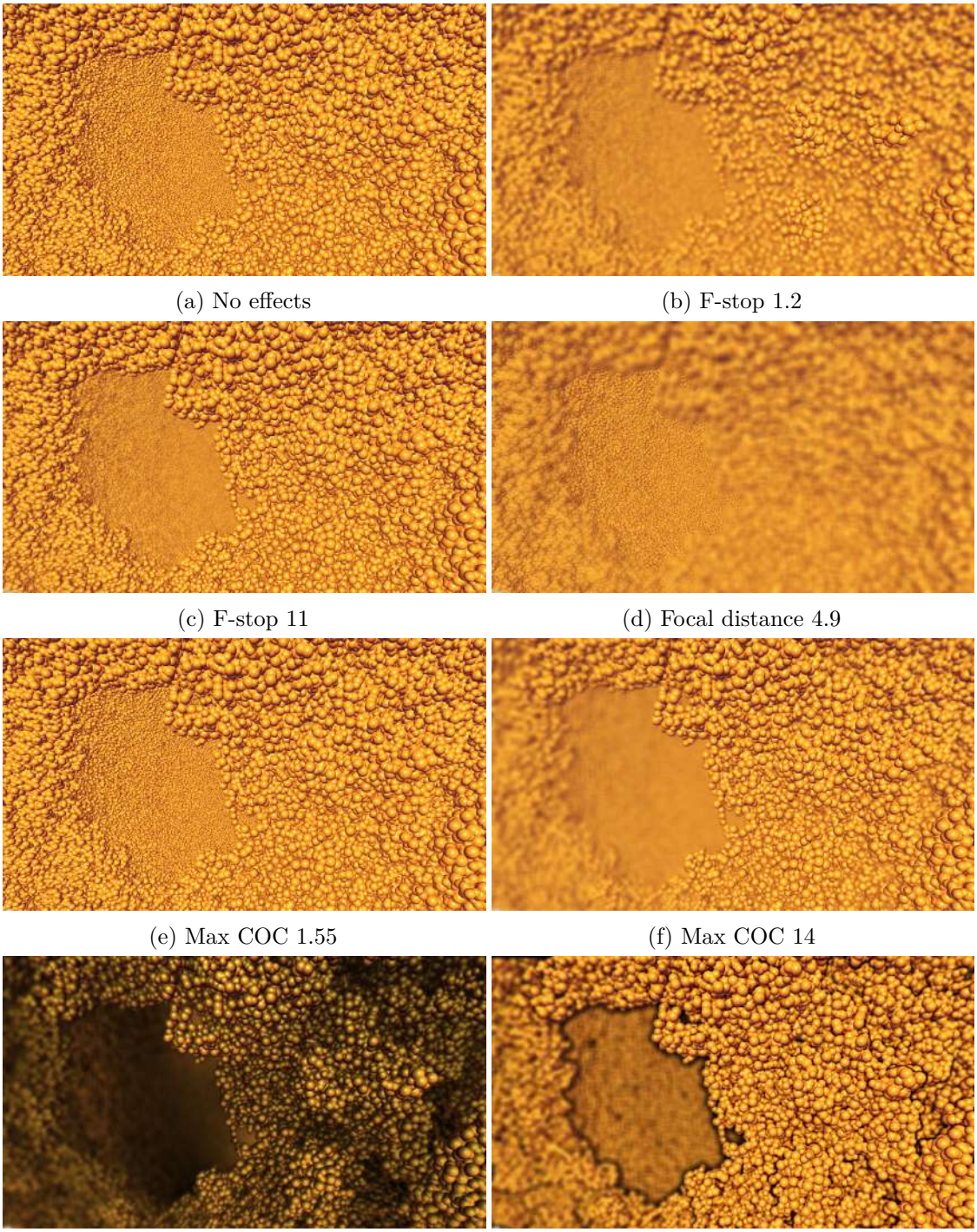
(c) Gauss (d) Gauss

(e) Gauss (f) Library + Gauss

(g) Halo (h) Halo

(i) Halo (j) Library + halo

Figure 6.14: 6ncl with different enhancement effects.



(a) No effects

(b) F-stop 1.2

(c) F-stop 11

(d) Focal distance 4.9

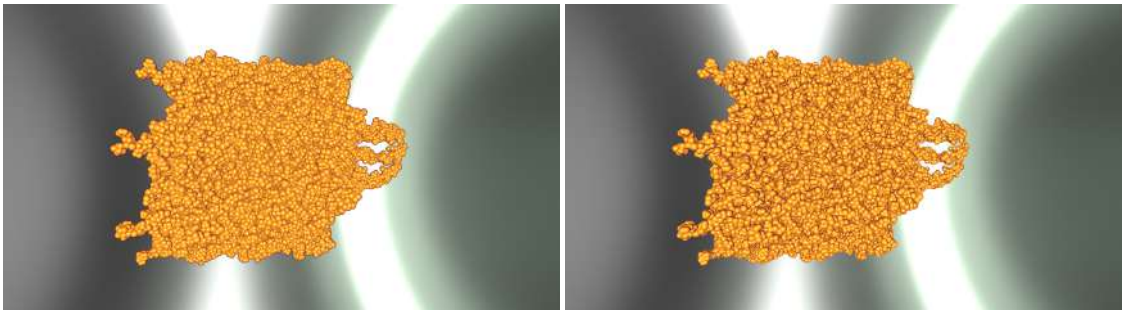
(e) Max COC 1.55

(f) Max COC 14

(g) HBAO+

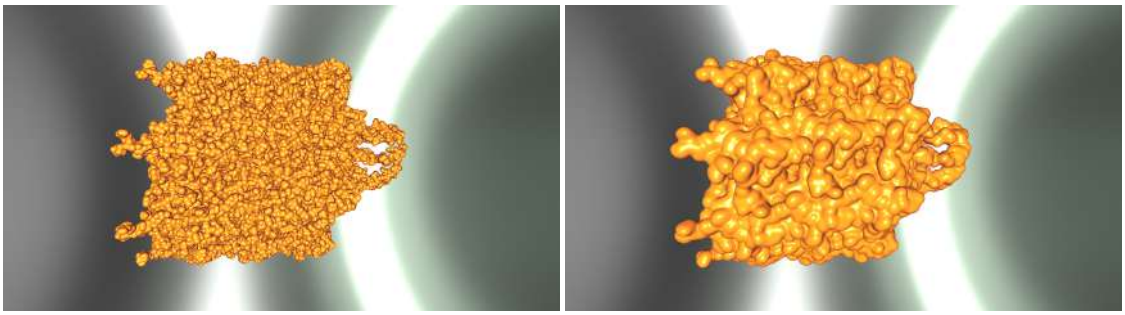
(h) Gaussian edge

Figure 6.15: 6o2s with depth of field effects



(a) Sharpness: -, FPS: 29

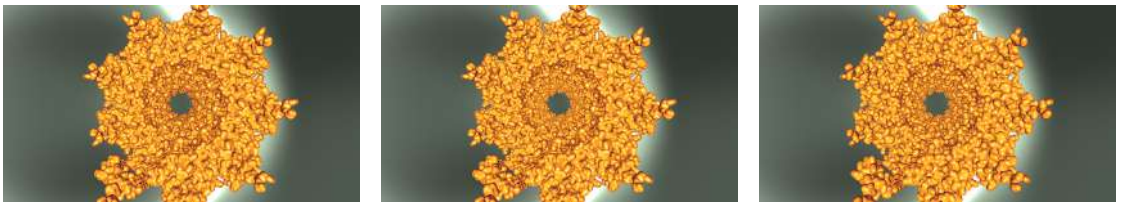
(b) Sharpness: 10, FPS: 25.5



(c) Sharpness: 2, FPS: 15.5

(d) Sharpness: 0.5, FPS 3

Figure 6.16: Gaussian surface (5odv)



(a) Level 0, no optimization, 12 FPS

(b) Level 0, sharpness attenuation, 15.5 FPS

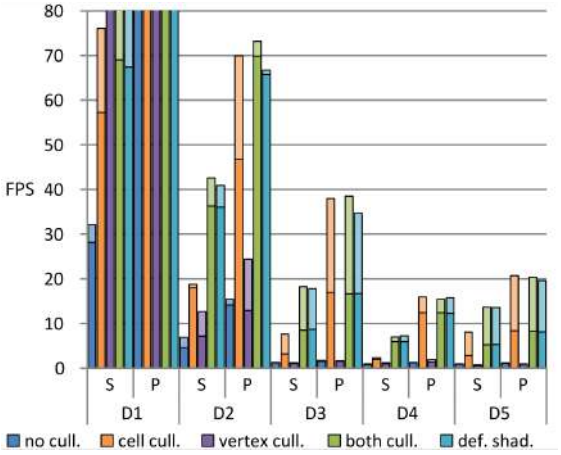
(c) Attenuation, distance based LOD, 21.5 FPS

(d) Detail of 6.17a

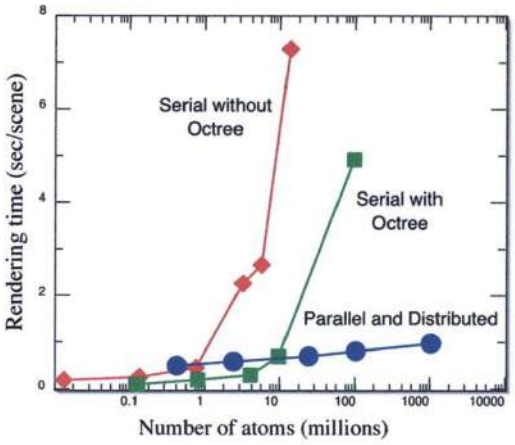
(e) Detail of 6.17b

(f) Detail of 6.17c

Figure 6.17: Gaussian surface (5odv)



(a) D1: 107,391; D5: 100,000,000 [GRDE10]



(b) [SKNV04]

Figure 6.18: Results achieved in related work

CHAPTER 7

Discussion

In this thesis, we present a rendering system for large-scale molecular data sets, including levels of detail, and enhancement methods. Our approach is a novel combination of existing techniques, in order to bridge the gap between medium-sized molecular visualization schemes, and extreme-scale data sets that either rely on repeating structures, or do not provide visual enhancement methods.

7.0.1 Summary and Contributions

The main contributions of this thesis are a hierarchical, multi-resolution data structure, and a least recently used caching system tailored to it. We believe that our proposed combination can bridge a gap between research focusing on advanced rendering methods for medium-sized molecules, and those concerned with extreme scale data sets, but with much more limited visualizations options. Our tests were conducted on data sets containing up to over 100 million individual atoms. While our artificial large-scale data sets do contain several instances of the same molecule, we render them as a single molecule. At about 2.44 million atoms, 3j3q may be the largest single structure to be found in the Protein Data Base right now, but it is certainly not the most complex molecular data set anyone will ever want to render. It therefore pays off to have a framework capable of rendering much larger structures, without having any prior information about its components. We propose to use our data structure in a way that combines the advantages of hierarchically divided space, and linear buffers. Contrary to many implementations that use a regular grid as a basic data structure, the pages in our octree contain a similar and controllable amount of atoms each. The advantage of this structuring becomes evident when calculating clusters in order to build levels of detail. It makes the maximum size and the approximate maximum computation time for neighborhood-based algorithms on each page predictable and balanced. As discussed in Section 3.2.2, division of space is necessary for some tasks, such as in our case the hierarchical clustering algorithm. The division of space into density-based pages that result in fewer, and more predictable

units can be effectively used whenever neighborhood queries are necessary. Atoms on the same page are guaranteed to be neighbors with a given amount of maximum distance. While we do not currently use this feature, octrees also make it relatively easy to access neighboring pages, compared to other trees, such as the kd-tree. Therefore, they extend the possibilities of neighborhood queries even beyond neighbors on the same page.

The LRU cache makes sure that the buffer is managed efficiently. Only pages that are needed are ever uploaded to the GPU, and they remain there until the space is needed by something else. This does not make much of a difference for small to medium sized molecules, but is highly relevant when dealing with very large data sets. Our system of differentiating between basic level of detail and rendering level makes it possible to quickly change between the advantages of more detailed and coarser, faster resolutions, without having to upload all the data from scratch. For large data sets, uploading to the GPU is one of the main bottlenecks. We provide a flexible approach to levels of detail, that also makes it possible to blend different resolutions per page, or based on distance measurements.

7.0.2 Limitations

When it comes to the scale of data sets, there are two main limitations. Firstly, the data set, including all the different LOD resolutions does have to fit into CPU memory, in order for our implementation to be able to handle it. Secondly, we assume that the coarsest level of detail fits into GPU memory. While our system is designed not to crash even if the coarsest level exceeds memory, we only tested it up to a maximum size of about 100 million atoms. This data set fits into the GPU at LOD 2, this is approximately a quarter of the original atoms.

Most other frameworks for molecular visualization provide the option of rendering a Solvent Excluded Surface, which our framework currently lacks. We provide an implementation of the less computationally expensive Gaussian surface, and assume that our neighborhood-based division of the data set would be an asset when implementing SES, but it is not included in the current version of the framework.

Our page-based buffer slots allow us to mix different levels of detail, which is highly useful when trying to find an optimal compromise between maximum resolution in the foreground. It also helps to reduce computations where they do not have an effect. However, it also leaves buffer space unused. Therefore, we do not use the entire capacity of GPU memory. It is possible in our system to upload a data set as a whole instead of per page, but this comes at the expense of loosing the advantages we described. Levels of detail are calculated per page, so if it is desirable to upload everything without unused space on the GPU, the data has to be uploaded at the original resolution, without advantages such as block-wise visibility culling.

When it comes to our implementation of visual effects, limitations include the fact that we do not provide transparency, and currently disregard additional molecular information such as chains and residues.

7.0.3 Future Work

There are several aspects of our work that could be extended to address some of the limitations. We think that it would be worthwhile to extend our data structure in such a way that it can be used for repeating structures. While one of the features that sets our proposed solution apart from other recently presented work in the field, is the fact that we do not rely on the existence of, or prior knowledge about, repeating structures, there is no reason not to combine the advantages of both approaches.

The other source of very large molecular data sets are molecular dynamics simulations. Our level of detail scheme makes a pre-processing step necessary, which is not ideal for animation-based visualizations. However, we believe that it would be possible to develop a smart pre-processing system that makes use of the similarity of consecutive time-steps.

When it comes to visual effects, the main component that would be desirable to have is the Solvent Excluded Surface. Implementing and optimized version of it would be the next step in that department. There are also some additional effects that we think could profit from a data structure divided into equal neighborhoods, such as the interreflection effect proposed by Skånberg [SVGR15].

Additionally, there are a couple of tweaks to our existing system that could be worth exploring. One such example is the interactive, manual selection of resolutions and effects. Selecting bounding boxes of pages, or drawing areas on the screen, and assigning resolution and effects, could give researchers some useful and dynamic tools, especially for very large data sets or computationally expensive methods such as the SES. This type of effect could make excellent use of the flexibility of mixed-block rendering, where areas in focus could be replaced with denser, high-resolution blocks, while the area that are not being investigated could be rendered at a lower resolution to increase frame rates.

CHAPTER 8

Conclusion

Many different solutions for rendering biomolecular data sets in particular and large point-based data sets in general already exist. In recent years, there has been a lot of research into rendering whole cells containing billions of atoms by taking repetitive structures into account, based mostly on the work of Lindow et al [LBH12]. However, these publications only work on structures that do contain repeating structures, and require additional structural information. In our research into the state of the art of solutions targeting large biomolecular data sets, we observed a certain lack of attention for very large data sets, without additional structural information that still need to be rendered at interactive frame rates, while using computationally expensive surface models. We see two specific advantages of our proposed data structure. The division of space into blocks containing a similar amount of data, which can be controlled by the user, is advantageous for algorithms that rely on neighborhood queries. Depending on the method, only the block itself, or the block and its direct neighbors, need to be taken into account. Blocks of equal size require approximately the same amount of time to calculate, which makes calculations predictable. Moreover, there are several possible applications for the flexible buffer updating system that allows us to mix different levels of detail.

The main disadvantage of our proposed solution is the computationally expensive pre-processing step, which makes it impractical for more than one time-step. However, we feel that the data structure we propose is a valuable contribution to the are of large point based-data sets, especially when neighborhood-based calculations are expected, or rendering at different resolution in a single frame are used. Of course, the data sets we presented as challenging to our system, could already be rendered without an advanced data structure on systems with higher capacity. But while it can be expected that hardware capacities will continue to increase, so will the complexity of data sets that need to be visualized. If the current maximum amount of atoms per page no longer makes sense for a system with higher capacity, simply adding a few bits, in order to store longer IDs, would make it possible to save more points per page. The system remains the

same, whether for a few thousand atoms, or entire cells, organs, or complex organisms. Even on a system with hight capacity, the possibility of using simple neighborhood-based algorithms with a linear increase in execution time, instead of an exponential increase, is worthwhile.

List of Figures

List of Tables

Bibliography

[AGL05] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. In *The Visualization Handbook*, pages 717–731, 2005. doi: 10.1016/B978-012387582-2/50038-1.

[BDST04] Chandrajit Bajaj, Peter Djeu, Vinay Siddavanahalli, and Anthony Thane. Texmol: Interactive visual exploration of large flexible multi-component molecular complexes. In *Proceedings of IEEE Visualization*, pages 243–250, 2004. doi: 10.1109/VISUAL.2004.103.

[Ben75] Jon L Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975. doi: 10.1145/361002.361007.

[BG07] Stefan Bruckner and Eduard Gröller. Enhancing depth-perception with flexible volumetric halos. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1344–1351, 2007. doi: 10.1109/TVCG.2007.70555.

[BHOM13] Mike Bukowski, Padraic Hennessy, Brian Osman, and Morgan McGuire. The skylanders swap force depth-of-field shader. In *GPU Pro 4: Advanced Rendering Techniques*, pages 175–186, 2013.

[BHP15] Johanna Beyer, Markus Hadwiger, and Hanspeter Pfister. State-of-the-art in gpu-based large-scale volume visualization. *Computer Graphics Forum*, 34(8):13–37, 2015. doi: 10.1111/cgf.12605.

[Bli82] James F Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, 1982. doi: 10.1145/357306.357310.

[Bru19] Stefan Bruckner. Dynamic visibility-driven molecular surfaces. *Computer Graphics Forum*, 38(2):317–329, 2019. doi: 10.1111/cgf.13640.

[BS09] Louis Bavoil and Miguel Sainz. Multi-layer dual-resolution screen-space ambient occlusion. In *SIGGRAPH Talks*, pages 45–45, 2009. doi: 10.1145/1597990.1598035.

[BWF+00] Helen M. Berman, John Westbrook, Zukang Feng, Gary Gilliland, T. N. Bhat, Helge Weissig, Ilya N. Shindyalov, and Philip E. Bourne. The protein data bank. *Nucleic Acids Research*, 28(1):235–242, 2000. doi: 10.1093/nar/28.1.235.

[CHP+79] Charles Csuri, Ron Hackathorn, Richard Parent, Wayne Carlson, and Marc Howard. Towards an interactive high visual complexity animation system. In *Proceedings of ACM SIGGRAPH*, pages 289–299, 1979. doi: 10.1145/965103.807458.

[CLK+11] Matthieu Chavent, Bruno Lévy, Michael Krone, Katrin Bidmon, Jean-Philippe Nominé, Thomas Ertl, and Marc Baaden. Gpu-powered tools boost molecular visualization. *Briefings in Bioinformatics*, 12(6):689–701, 2011. doi: 10.1093/bib/bbq089.

[CMB13] Paul A Craig, Lea Vacca Michel, and Robert C Bateman. A survey of educational uses of molecular visualization freeware. *Biochemistry and Molecular Biology Education*, 41(3):193–205, 2013. doi: 10.1039/B5RP90005K.

[DVVC19] David Dubbeldam, Jocelyne Vreede, Thijs JH Vlugt, and Sofia Calero. Highlights of (bio-) chemical tools and visualization software for computational science. *Current Opinion in Chemical Engineering*, 23(1):1–13, 2019. doi: 10.1016/j.coche.2019.02.001.

[Ede99] Herbert Edelsbrunner. Deformable smooth surface design. *Discrete & Computational Geometry*, 21(1):87–115, 1999. doi: 10.1007/PL00009412.

[FAW10] Roland Fraedrich, Stefan Auer, and Rudiger Westermann. Efficient high-quality volume rendering of sph data. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1533–1540, 2010. doi: 10.1109/TVCG.2010.148.

[FGE10] Martin Falk, Sebastian Grottel, and Thomas Ertl. Interactive image-space volume visualization for dynamic particle simulations. In *Proceedings of SIGRAD*, pages 35–43, 2010.

[FK03] Randima Fernando and Mark J Kilgard. *The Cg Tutorial: The definitive guide to programmable real-time graphics*. Addison-Wesley Longman Publishing Co., Inc., 2003. isbn: {9780321194961}.

[FKE13] Martin Falk, Michael Krone, and Thomas Ertl. Atomistic visualization of mesoscopic whole-cell simulations using ray-casted instancing. *Computer Graphics Forum*, 32(8):195–206, 2013. doi: 10.1111/cgf.12197.

[Fra02] Eric Francoeur. Cyrus levinthal, the kluge and the origins of interactive molecular graphics. *Endeavour*, 26(4):127–131, 2002. doi: 10.1016/S0160-9327(02)01468-0.

[FSW09] Roland Fraedrich, Jens Schneider, and Rüdiger Westermann. Exploring the millennium run-scalable rendering of large-scale cosmological datasets. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1251–1258, 2009. doi: 10.1109/TVCG.2009.142.

[GIK+07] Christiaan P Gribble, Thiago Ize, Andrew Kensler, Ingo Wald, and Steven G Parker. A coherent grid traversal approach to visualizing particle-based simulation data. *IEEE Transactions on Visualization and Computer Graphics*, 13(4):758–768, 2007. doi: 10.1109/TVCG.2007.1059.

[GKM+14] Sebastian Grottel, Michael Krone, Christoph Müller, Guido Reina, and Thomas Ertl. Megamol—a prototyping framework for particle-based visualization. *IEEE Transactions on Visualization and Computer Graphics*, 21(2):201–214, 2014. doi: 10.1109/TVCG.2014.2350479.

[GKSE12] Sebastian Grottel, Michael Krone, Katrin Scharnowski, and Thomas Ertl. Object-space ambient occlusion for molecular dynamics. In *Proceedings of IEEE PacificVis*, pages 209–216, 2012. doi: 10.1109/PacificVis.2012.6183593.

[GNL+15] Dongliang Guo, Junlan Nie, Meng Liang, Yu Wang, Yanfen Wang, and Zhengping Hu. View-dependent level-of-detail abstraction for interactive atomistic visualization of biological structures. *Computers & Graphics*, 52(C):62–71, 2015. doi: 10.1016/j.cag.2015.06.008.

[GRDE10] Sebastian Grottel, Guido Reina, Carsten Dachsbacher, and Thomas Ertl. Coherent culling and shading for large molecular dynamics visualization. *Computer Graphics Forum*, 29(3):953–962, 2010. doi: 10.1111/j.1467-8659.2009.01698.x.

[Hay10] Callum Hay. Gaussian blur shader (glsl), 2010. Accessed: 2020-07-28, https://callumhay.blogspot.com/2010/09/gaussian-blur-shader-glsl.html.

[HDS96] William Humphrey, Andrew Dalke, and Klaus Schulten. Vmd: visual molecular dynamics. *Journal of Molecular Graphics*, 14(1):33–38, 1996. doi: 10.1016/0263-7855(96)00018-5.

[HE03] Matthias Hopf and Thomas Ertl. Hierarchical splatting of scattered data. In *Proceedings of IEEE Visualization*, pages 433–440, 2003. doi: 10.1109/VISUAL.2003.1250404.

[Her06] Angel Herraez. Biomolecules in the computer: Jmol to the rescue. *Biochemistry and Molecular Biology Education*, 34(4):255–261, 2006. doi: 10.1002/bmb.2006.494034042644.

[HKG+17] Pedro Hermosilla, Michael Krone, Victor Guallar, Pere-Pau Vázquez, Àlvar Vinacua, and Timo Ropinski. Interactive gpu-based generation of solvent-excluded surfaces. *The Visual Computer*, 33(6):869–881, 2017. doi: 10.1007/s00371-017-1397-2.

[HKK07] Takahiro Harada, Seiichi Koshizuka, and Yoichiro Kawaguchi. Sliced data structure for particle-based simulations on gpus. In *Proceedings of GRAPHITE*, pages 55–62, 2007. doi: 10.1145/1321261.1321271.

[Hof65] August Hoffmann. On the combining power of atoms. In *Proceedings of the Royal Institution*, pages 401–430, 1865.

[HSS+05] Markus Hadwiger, Christian Sigg, Henning Scharsach, Khatja Bühler, and Markus Gross. Real-time ray-casting and advanced shading of discrete isosurfaces. *Computer Graphics Forum*, 24(1):303–312, 2005. doi: 10.1111/j.1467-8659.2005.00855.x.

[HVS04] Xuejun Hao, Amitabh Varshney, and Sergei Sukharev. Real-time visualization of large time-varying molecules. In *Proceedings of the High-Performance Computing Symposium*, pages 109–114, 2004.

[IWR+17] Mohamed Ibrahim, Patrick Wickenhäuser, Peter Rautek, Guido Reina, and Markus Hadwiger. Screen-space normal distribution function caching for consistent multi-resolution rendering of large particle data. *IEEE Transactions on Visualization and Computer Graphics*, 24(1):944–953, 2017. doi: 10.1109/TVCG.2017.2743979.

[JH14] Graham T Johnson and Samuel Hertig. A guide to the visual analysis and communication of biomolecular structural data. *Nature Reviews Molecular Cell Biology*, 15(10):690–698, 2014. doi: 10.1038/nrm3874.

[JJS05] Loretta L Jones, Kenneth D Jordan, and Neil A Stillings. Molecular visualization in chemistry education: the role of multidisciplinary collaboration. *Chemistry Education Research and Practice*, 6(3):136–149, 2005.

[KAD+06] Richard Keiser, Bart Adams, Philip Dutré, Leonidas J Guibas, and Mark Pauly. Multiresolution particle-based fluids. Technical report, Swiss Federal Institute of Technology Zurich, Department of Computer Science, 2006. doi: 10.3929/ethz-a-006780981.

[KGE11] Michael Krone, Sebastian Grottel, and Thomas Ertl. Parallel contour-buildup algorithm for the molecular surface. In *Proceedings of IEEE BioVis*, pages 17–22, 2011. doi: 10.1109/BioVis.2011.6094043.

[KKF+17] Barbora Kozlíková, Michael Krone, Martin Falk, Norbert Lindow, Marc Baaden, Daniel Baum, Ivan Viola, Julius Parulek, and Hans-Christian Hege. Visualization of biomolecular structures: State of the art revisited. *Computer Graphics Forum*, 36(8):178–204, 2017. doi: 10.1111/cgf.13072.

[KSES12] Michael Krone, John E Stone, Thomas Ertl, and Klaus Schulten. Fast visualization of gaussian density surfaces for molecular dynamics and particle system trajectories. In *Proceedings of EuroVis (Short Papers)*, pages 67–71, 2012. doi: 10.2312/PE/EuroVisShort/EuroVisShort2012/067-071.

[KW03] Jens Kruger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings of IEEE Visualization*, pages 287–292, 2003. doi: 10.1109/VIS.2003.10001.

[KWN+14] Aaron Knoll, Ingo Wald, Paul Navratil, Anne Bowen, Khairi Reda, Michael E Papka, and Kelly Gaither. Rbf volume ray casting on multicore and manycore cpus. *Computer Graphics Forum*, 33(3):71–80, 2014. doi: 10.1111/cgf.12363.

[LBH12] Norbert Lindow, Daniel Baum, and Hans-Christian Hege. Interactive rendering of materials and biological structures on atomic and nanoscopic scale. *Computer Graphics Forum*, 31(3pt4):1325–1334, 2012. doi: 10.1111/j.1467-8659.2012.03128.x.

[LBH14] Norbert Lindow, Daniel Baum, and Hans-Christian Hege. Ligand excluded surface: A new type of molecular surface. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2486–2495, 2014. doi: 10.1109/TVCG.2014.2346404.

[LBPH10] Norbert Lindow, Daniel Baum, Steffen Prohaska, and Hans-Christian Hege. Accelerated visualization of dynamic molecular surfaces. *Computer Graphics Forum*, 29(3):943–952, 2010. doi: 10.1111/j.1467-8659.2009.01693.x.

[LC87] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM SIGGRAPH Computer Graphics*, 21(4):163–169, 1987. doi: 10.1145/37401.37422.

[LCL15] Tiantian Liu, Minxin Chen, and Benzhuo Lu. Parameterization for molecular gaussian surface and a comparison study of surface mesh generation. *Journal of Molecular Modeling*, 21(5):113, 2015. doi: 10.1007/s00894-015-2654-9.

[LGF04] Frank Losasso, Frédéric Gibou, and Ron Fedkiw. Simulating water and smoke with an octree data structure. *ACM Transactions on Graphics*, 23(3):457–462, 2004. doi: 10.1145/1015706.1015745.

[LH91] David Laur and Pat Hanrahan. Hierarchical splatting: A progressive refinement algorithm for volume rendering. *ACM SIGGRAPH Computer Graphics*, 25(4):285–288, 1991. doi: 10.1145/122718.122748.

[LMPSV14] Mathieu Le Muzic, Julius Parulek, Anne-Kristin Stavrum, and Ivan Viola. Illustrative visualization of molecular reactions using omniscient intelligence and passive agents. *Computer Graphics Forum*, 33(3):141–150, 2014. doi: 10.1111/cgf.12370.

[LO11] Phillip A Laplante and Seppo J Ovaska. *Real-time systems design and analysis: tools for the practitioner*. John Wiley and Sons, 2011. isbn: 978-0470768648.

[Lod00] Harvey F. Lodish. *Molecular Cell Biology*. W.H. Freeman, 2000. isbn: 978-1464109812.

[LPK06] Jun Lee, Sungjun Park, and Jee-In Kim. View-dependent rendering of large-scale molecular models using level of detail. In *Proceedings of the International Conference on Hybrid Information Technology*, pages 691–698, 2006.

[LR71] Byungkook Lee and Frederic M Richards. The interpretation of protein structures: estimation of static accessibility. *Journal of Molecular Biology*, 55(3):379–400, 1971. doi: 10.1016/0022-2836(71)90324-X.

[M+13] Daniel Müllner et al. fastcluster: Fast hierarchical, agglomerative clustering routines for r and python. *Journal of Statistical Software*, 53(9):1–18, 2013. doi: 10.18637/jss.v053.i09.

[Mar65] James Martin. *Design of real-time computer systems*. Prentice Hall, 1965. isbn: 978-1114207875.

[Max04] Nelson Max. Hierarchical molecular modelling with ellipsoids. *Journal of Molecular Graphics and Modelling*, 23(3):233–238, 2004. doi: 10.1016/j.jmgm.2004.07.001.

[MCG03] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 154–159, 2003. doi: 10.2312/SCA03/154-159.

[MEK+17] Nick Matthews, Robert Easdon, Akio Kitao, Steven Hayward, and Stephen Laycock. High quality rendering of protein dynamics in space filling mode. *Journal of Molecular Graphics and Modelling*, 78(1):158–167, 2017. doi: 10.1016/j.jmgm.2017.09.017.

[MOBH11] Morgan McGuire, Brian Osman, Michael Bukowski, and Padraic Hennessy. The alchemy screen-space ambient obscurance algorithm. In *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics*, pages 25–32, 2011. doi: 10.1145/2018323.2018327.

[MS15] Steve Marschner and Peter Shirley. *Fundamentals of computer graphics*. CRC Press, 2015. isbn: 978-1568814698.

[NJB07] Paul Navratil, Jarrett Johnson, and Volker Bromm. Visualization of cosmological particle-based datasets. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1712–1718, 2007. doi: 10.1109/TVCG.2007.70526.

[PB13] Julius Parulek and Andrea Brambilla. Fast blending scheme for molecular surface representation. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2653–2662, 2013. doi: 10.1109/TVCG.2013.158.

[PGH+04] Eric F Pettersen, Thomas D Goddard, Conrad C Huang, Gregory S Couch, Daniel M Greenblatt, Elaine C Meng, and Thomas E Ferrin. Ucsf chimera—a visualization system for exploratory research and analysis. *Journal of Computational Chemistry*, 25(13):1605–1612, 2004. doi: 10.1002/jcc.20084.

[PJR+14] Julius Parulek, Daniel Jönsson, Timo Ropinski, Stefan Bruckner, Anders Ynnerman, and Ivan Viola. Continuous levels-of-detail and visual abstraction for seamless molecular visualization. 33(6):276–287, 2014. doi: 10.1111/cgf.12349.

[PV12] Julius Parulek and Ivan Viola. Implicit representation of molecular surfaces. In *Proceedings of IEEE PacificVis*, pages 217–224, 2012. doi: 10.1109/PacificVis.2012.6183594.

[PZVBG00] Hanspeter Pfister, Matthias Zwicker, Jeroen Van Baar, and Markus Gross. Surfels: Surface elements as rendering primitives. In *Proceedings of ACM SIGGRAPH*, pages 335–342, 2000. doi: 10.1145/344779.344936.

[QEE+05] Wei Qiao, David S Ebert, Alireza Entezari, Marek Korkusinski, and Gerhard Klimeck. Volqd: Direct volume rendering of multi-million atom quantum dot simulations. In *Proceedings of IEEE Visualization*, pages 319–326, 2005. doi: 10.1109/VISUAL.2005.1532811.

[RE05] Guido Reina and Thomas Ertl. Hardware-accelerated glyphs for mono-and dipoles in molecular dynamics visualization. In *Proceedings of EuroVis*, pages 177–182, 2005. doi: 10.2312/VisSym/EuroVis05/177-182.

[Ree83] William T Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Transactions On Graphics*, 2(2):91–108, 1983. doi: 10.1145/357318.357320.

[RGE19] Guido Reina, Patrick Gralka, and Thomas Ertl. A decade of particle-based scientific visualization. *The European Physical Journal Special Topics*, 227(14):1705–1723, 2019. doi: 10.1140/epjst/e2019-800172-4.

[Ric77] Frederic M Richards. Areas, volumes, packing, and protein structure. *Annual Review of Biophysics and Bioengineering*, 6(1):151–176, 1977. doi: 10.1146/annurev.bb.06.060177.001055.

[RK09] Steffen Raschdorf and Michael Kolonko. Loose octree: a data structure for the simulation of polydisperse particle packings. Technical report, Clausthal University of Technology, 2009.

[RL00] Szymon Rusinkiewicz and Marc Levoy. Qsplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH*, pages 343–352, 2000. doi: 10.1145/344779.344940.

[RM00] Paul Read and Mark-Paul Meyer. *Restoration of motion picture film*. Elsevier, 2000. isbn: 978-0750627931.

[RT06] Guodong Rong and Tiow-Seng Tan. Jump flooding in gpu with applications to voronoi diagram and distance transform. In *Proceedings of the Symposium on Interactive 3D Graphics and Games*, pages 109–116, 2006. doi: 10.1145/1111411.1111431.

[RTW13] Florian Reichl, Marc Treib, and Rüdiger Westermann. Visualization of big sph simulations via compressed octree grids. In *Proceedings of the IEEE International Conference on Big Data*, pages 71–78, 2013. doi: 10.1109/BigData.2013.6691717.

[Sam90] Hanan Samet. *The design and analysis of spatial data structures*. Addison-Wesley Reading, MA, 1990. isbn: 978-0201502558.

[Sch15] LLC Schrödinger. The pymol molecular graphics system, version 1.8. 2015.

[SGG15] Joachim Staib, Sebastian Grottel, and Stefan Gumhold. Visualization of particle-based data with transparency and ambient occlusion. *Computer Graphics Forum*, 34(3):151–160, 2015. doi: doi:10.1111/cgf.12627.

[SI94] Hikmet Senay and Eve Ignatius. A knowledge-based system for visualization design. *IEEE Computer Graphics and Applications*, 14(6):36–47, 1994. doi: 10.1109/38.329093.

[SKNV04] Ashish Sharma, Rajiv K Kalia, Aiichiro Nakano, and Priya Vashishta. Scalable and portable visualization of large atomistic datasets. *Computer Physics Communications*, 163(1):53–64, 2004. doi: 10.1016/j.cpc.2004.07.008.

[SMK+16] Karsten Schatz, Christoph Müller, Michael Krone, Jens Schneider, Guido Reina, and Thomas Ertl. Interactive visual exploration of a trillion particles. In *Proceedings of the IEEE Symposium on Large Data Analysis and Visualization*, pages 56–64, 2016. doi: 10.1109/LDAV.2016.7874310.

[SMW95] Roger A Sayle and E James Milner-White. Rasmol: biomolecular graphics for all. *Trends in Biochemical Sciences*, 20(9):374–376, 1995. doi: 10.1016/S0968-0004(00)89080-5.

[SVGR15] Robin Skånberg, Pere-Pau Vazquez, Victor Guallar, and Timo Ropinski. Real-time molecular visualization supporting diffuse interreflections and ambient occlusion. *IEEE Transactions on Visualization and Computer Graphics*, 22(1):718–727, 2015. doi: 10.1109/TVCG.2015.2467293.

[SWBG06] Christian Sigg, Tim Weyrich, Mario Botsch, and Markus H Gross. Gpu-based ray-casting of quadratic surfaces. In *Proceedings of the Symposium on Point-Based Graphics*, pages 59–65, 2006. doi: 10.2312/SPBG/SPBG06/059-065.

[TA96] Maxim Totrov and Ruben Abagyan. The contour-buildup algorithm to calculate the analytical molecular surface. *Journal of Structural Biology*, 116(1):138–143, 1996. doi: 10.1006/jsbi.1996.0022.

[TCM06] Marco Tarini, Paolo Cignoni, and Claudio Montani. Ambient occlusion and edge cueing for enhancing real time molecular visualization. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1237–1244, 2006. doi: 10.1109/TVCG.2006.115.

[TL04] Rodrigo Toledo and Bruno Levy. Extending the graphic pipeline with new gpu-accelerated primitives. Technical report, INRIA, 2004.

[Tra92] Anthony S Travis. August wilhelm hofmann (1818–1892). *Endeavour*, 16(2):59–65, 1992. doi: 10.1016/0160-9327(92)90003-8.

[Tuf83] Edward R Tufte. *The Visual Display of Quantitative Information*. Graphics Press, 1983. isbn: 1930824130.

[Tuf97] Edward R Tufte. *Visual and statistical thinking: Displays of evidence for making decisions*. Graphics Press Cheshire, CT, 1997. isbn: 978-0961392130.

[Tur07] Ken Turkowski. Incremental computation of the gaussian. In *GPU Gems 3*, pages 877–890, 2007.

[VdW73] Johannes Diderik Van der Waals. *Over de Continuïteit van den Gas- en Vloeistoftoestand*. Sijthoff, 1873.

[VDZLBI11] Matthew Van Der Zwan, Wouter Lueks, Henk Bekker, and Tobias Isenberg. Illustrative molecular visualization with continuous abstraction. *Computer Graphics Forum*, 30(3):683–690, 2011. doi: 10.1111/j.1467-8659.2011.01917.x.

[Wes89] Lee Westover. Interactive volume rendering. In *Proceedings of the Chapel Hill Workshop on Volume Visualization*, pages 9–16, 1989. doi: 10.1145/329129.329138.

[WIK+06] Ingo Wald, Thiago Ize, Andrew Kensler, Aaron Knoll, and Steven G Parker. Ray tracing animated scenes using coherent grid traversal. *ACM Transactions on Graphics*, 25(3):485–493, 2006. doi: 10.1145/1141911.1141913.

[WKJ+15] Ingo Wald, Aaron Knoll, Gregory P Johnson, Will Usher, Valerio Pascucci, and Michael E Papka. Cpu ray tracing large particle data with balanced pkd trees. In *Proceedings of IEEE SciVis*, pages 57–64, 2015. doi: 10.1109/SciVis.2015.7429492.

[WSB14] Thomas Waltemate, Björn Sommer, and Mario Botsch. Membrane mapping: combining mesoscopic and molecular cell visualization. In *Proceedings of the Eurographics Workshop on Visual Computing for Biology and Medicine*, pages 89–96, 2014. doi: 10.2312/vcbm.20141187.

[XZY17] Xiangyun Xiao, Shuai Zhang, and Xubo Yang. Real-time high-quality surface rendering for large scale particle-based fluids. In *Proceedings of the ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 1–8, 2017. doi: 10.1145/3023368.3023377.

[ZCW+04] Huabing Zhu, Tony Kai-Yun Chan, Lizhe Wang, Wentong Cai, and Simon See. A prototype of distributed molecular visualization on computational grids. *Future Generation Computer Systems*, 20(5):727–737, 2004. doi: 10.1016/j.future.2003.11.023.

[ZD17] Tobias Zirr and Carsten Dachsbacher. Memory-efficient on-the-fly voxelization and rendering of particle data. *IEEE Transactions on Visualization and Computer Graphics*, 24(2):1155–1166, 2017. doi: 10.1109/TVCG.2017.2656897.