Varying Regions of Interest for Interactive Visualization of Macromolecules

Hugo Alexandre Pereira Fernandes

Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering

Supervisors: Prof. João António Madeiras Pereira
Dr. Daniel Simões Lopes

Examination Committee
Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. João António Madeiras Pereira
Member of the Committee: Prof. Abel João Padrão Gomes

November 2016
Dedicated to my dear family and friends...
Acknowledgments

I would like to thank my thesis supervisor professor João Madeiras Pereira for trusting me this challenge. A special thanks for my co-supervisor doctor Daniel Simões Lopes who was always there to support me and always believed in my capabilities.

I would also like to acknowledge A-MOP: FCT Project UTAP-EXPL/QEQ-COM/0019/2014, the project that allowed this work to happen, among others in the molecular field. I also acknowledge that I was financially supported by national funds through the Portuguese Foundation for Science and Technology with references IT-MEDEX PTDC/EEISII/6038/2014 and UID/CEC/50021/2013.

Thank you to Jorge Martins, a student that helped me by sharing his previous work in bounding volume hierarchies. I would also like to gratefully acknowledge the support of NVIDIA corporation.

More personally, I would like to appreciate my family support that helped me to reach one of my life goals: learn about my special interest, computer science. To my mother, father and brother, I can not thank enough, even if most of the times I may not show it.

To my closest friends, one of the greatest way to get motivation in hopeless times. Each one is unique and has a special place in my heart.

To my dear girlfriend, one capable of put up with me is certainly a special person. You made of me a better person, made me aware of my skills and value. I hope that you continue being as awesome as you are, supportive, lovely and heartwarming. By the way, good luck for your turn!
Resumo

Esta tese explora uma nova técnica de visualização interactiva de macromoléculas, nas quais apenas zonas de interesse são renderizadas com grande qualidade, deixando as regiões restantes com menos detalhes visuais.

O algoritmo usa tanto a técnica de rasterização como a de ray tracing de uma maneira alternativa ao que se usa habitualmente. Ray tracing, normalmente, é usado para criar imagens inteiras ou para simular efeitos como sombras em programas que usem rasterização. Nesta tese, rasterização e ray tracing são combinados para, em conjunto, criarem a visualização de objectos na cena.

A visualização obtida é uma nova alternativa ao software de visualização molecular existentes, sendo mais adequado para análise de zonas de interesse, ao ajudar os utilizadores interessados em analisar, principalmente, partes específicas de moléculas, focando essas zonas e simultaneamente garantindo um melhor desempenho, que significa maior capacidade de interactividade.

Palavras-chave: Macromoléculas, superfície de solvente excluído, interactividade, traçado de raios, superfícies implícitas, nível de detalhe, zonas de ligação, GPU, regiões de interesse
Abstract

This thesis explores a novel approach for interactive visualization of macromolecules, in which only regions of interest are rendered with high quality, leaving the remaining regions with lesser visual details.

The algorithm created uses both rasterization and ray tracing techniques in a different way that it has been used before. Ray tracing is usually used either to create the entire images or to help simulating features like shadows in rasterization based techniques. In this thesis, both methods are combined to, together, generate the visualization of objects on the scene.

Our visualization is a new alternative to the existing molecular analysis software, more suitable to analyze areas of interest, by focusing those zones and granting better performance, hence having more interactivity capabilities.

Keywords: Macromolecules, solvent excluded surface, interactivity, ray tracing, implicit surfaces, level of detail, binding sites, GPU, regions of interest
Contents

Acknowledgments .......................................................... v
Resumo ............................................................................. vii
Abstract ........................................................................... ix
List of Tables ..................................................................... xiii
List of Figures ..................................................................... xv
Nomenclature ..................................................................... xvii
Glossary ............................................................................ xix

1 Introduction ..................................................................... 1
  1.1 Motivation ................................................................. 1
  1.2 Objectives and contributions ......................................... 2
  1.3 Thesis Outline ............................................................. 2

2 Background ...................................................................... 3
  2.1 Molecular Representations ........................................... 3
  2.2 Graphics visualization ................................................ 5
  2.3 Implicit Surfaces .......................................................... 7
    2.3.1 Polygonization ...................................................... 7
    2.3.2 Ray tracing .......................................................... 8
  2.4 CUDA framework ....................................................... 9
  2.5 Acceleration structures ............................................... 10
    2.5.1 Bounding volumes hierarchy ................................. 11
    2.5.2 Uniform Grids ..................................................... 12
    2.5.3 K-d tree ............................................................. 13
  2.6 Screen-space ambient occlusion ................................... 14
  2.7 Shading models ........................................................... 15

3 State of the art ............................................................. 17
  3.1 Molecular visualization ............................................... 17
  3.2 Level of detail based techniques .................................... 20
4 Implementation

4.1 Architecture ................................................. 23

4.2 Initialization step ........................................... 25

4.2.1 MolecularVisualization class .............................. 25

4.2.2 Scene class .................................................. 25

4.2.3 Rasterizer class ............................................ 27

4.2.4 Raytracer class .............................................. 27

4.2.5 AccelerationStructures class .............................. 28

4.3 Draw event ..................................................... 29

4.3.1 MolecularVisualization class .............................. 29

4.3.2 Rasterizer class ............................................ 30

4.3.3 Raytracer class .............................................. 31

4.3.4 Sphere tracing ............................................... 33

4.4 Protein Database parser ...................................... 40

4.4.1 Sphere tracing range optimization .......................... 41

4.5 Prototype evolution ........................................... 42

5 Experimental tests and results ................................. 47

5.1 Visual quality tests ........................................... 49

5.1.1 Test 1AON-1 ................................................. 49

5.1.2 Test 1AON-2 ................................................. 50

5.1.3 Test 1IGT ...................................................... 51

5.2 Performance tests ............................................. 51

5.2.1 Solution performance ...................................... 52

5.3 Full solvent-excluded surface versus regions of interest ........ 54

5.4 Discussion ..................................................... 54

5.4.1 Visual quality ............................................... 55

5.4.2 Performance .................................................. 55

6 Conclusions ..................................................... 57

6.1 Future Work .................................................. 57

Bibliography ..................................................... 59
# List of Tables

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Iteration of a combination</td>
<td>39</td>
</tr>
<tr>
<td>5.1</td>
<td>1AON-1: Results with the thinnest thresholds</td>
<td>52</td>
</tr>
<tr>
<td>5.2</td>
<td>1AON-1: Results with the thickest thresholds</td>
<td>52</td>
</tr>
<tr>
<td>5.3</td>
<td>1AON-2: Results with the thinnest thresholds</td>
<td>53</td>
</tr>
<tr>
<td>5.4</td>
<td>1AON-2: Results with the thickest thresholds</td>
<td>53</td>
</tr>
<tr>
<td>5.5</td>
<td>1IGT: Results with the thinnest thresholds</td>
<td>53</td>
</tr>
<tr>
<td>5.6</td>
<td>1IGT: Results with the thickest thresholds</td>
<td>53</td>
</tr>
<tr>
<td>5.7</td>
<td>Uniform grid size comparison: Full SES vs ROI</td>
<td>54</td>
</tr>
<tr>
<td>5.8</td>
<td>Frame rate: Full SES vs ROI (thinnest thresholds)</td>
<td>54</td>
</tr>
<tr>
<td>5.9</td>
<td>Frame rate: Full SES vs ROI (thickest thresholds)</td>
<td>54</td>
</tr>
</tbody>
</table>
# List of Figures

2.1 SAS-SES difference ................................................................. 4  
2.2 Implicit Surfaces ................................................................. 5  
2.3 OpenGL pipeline ................................................................. 6  
2.4 Whitted 1980: Ray tracing algorithm ....................................... 7  
2.5 Marching cubes algorithm ..................................................... 8  
2.6 Hart 1994: Sphere tracing algorithm ....................................... 9  
2.7 CUDA architecture .............................................................. 10  
2.8 Bounding volume hierarchy example ....................................... 11  
2.9 Uniform grid example .......................................................... 13  
2.10 K-d tree example ............................................................... 14  
2.11 Crysis SSAO ............................................................... 15  
2.12 SSAO sphere ............................................................... 15  
2.13 SSAO hemisphere .............................................................. 15  
2.14 Shading models ................................................................. 16  
3.1 Parulek et al. 2012: Toroidal function, spherical triangle function and the visual result ................................................. 17  
3.2 Parulek et al. 2013: Comparison between SES representations ................................................................. 18  
3.3 Kanamori et al. 2008: Visual result of a molecule with 2878 metaballs ................................................................. 19  
3.4 Singh et al. 2010: Step length variations ...................................... 19  
3.5 Parulek et al. 2014: Phospholipase bound to lipid membrane ................................................................. 20  
3.6 Bajaj et al. 2004: Secondary structures of hemoglobin ................................................................. 21  
3.7 Lee et al. 2006: Mesh simplification process .................................. 21  
4.1 Initial sketch of our approach .................................................. 24  
4.2 Class domain model ............................................................. 25  
4.3 Frustum scheme ................................................................. 26  
4.4 Kernel blocks linear organization ............................................ 33  
4.5 Kernel blocks rectangle organization ....................................... 33  
4.6 Parulek et al. 2012: Projection from the point to the influence radius ................................................................. 37  
4.7 Evolution of the prototype ..................................................... 43  
4.8 Evolution of the prototype: Image mixing .................................... 44
4.9 Evolution of the prototype: Final result .............................................. 45
5.1 Full SES vs ROI: 1AON-1 ................................................................. 49
5.2 Full SES vs ROI: 1AON-2 ................................................................. 50
5.3 Full SES vs ROI: 1IGT ................................................................. 51
6.1 User interface example ................................................................. 58
Nomenclature

Greek symbols

$\alpha, \beta$ Angles.

Subscripts

$i, j, k$ Computational indexes.

$x, y, z$ Cartesian components.

Superscripts

$\vec{a}$ Vector.
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AABB</td>
<td>Axis-Aligned Bounding Box</td>
</tr>
<tr>
<td>AMOP</td>
<td>Algorithms for Macro-Molecular Pocket Detection</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>BVH</td>
<td>Bounding Volume Hierarchy</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CUDA</td>
<td>Compute Unified Device Architecture</td>
</tr>
<tr>
<td>FBO</td>
<td>Framebuffer Object</td>
</tr>
<tr>
<td>FPS</td>
<td>Frames Per Second</td>
</tr>
<tr>
<td>GHz</td>
<td>Gigahertz</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>K-d</td>
<td>K-Dimensional</td>
</tr>
<tr>
<td>OpenGL</td>
<td>Open Graphics Library</td>
</tr>
<tr>
<td>PBO</td>
<td>Pixel Buffer Object</td>
</tr>
<tr>
<td>PDB</td>
<td>Protein Database</td>
</tr>
<tr>
<td>RAM</td>
<td>Random-Access Memory</td>
</tr>
<tr>
<td>RGBA</td>
<td>Red, Green, Blue, Alpha color model</td>
</tr>
<tr>
<td>RGB</td>
<td>Red, Green, Blue color model</td>
</tr>
<tr>
<td>ROI</td>
<td>Region Of Interest</td>
</tr>
<tr>
<td>SAS</td>
<td>Solvent-Accessible Surface</td>
</tr>
<tr>
<td>SES</td>
<td>Solvent-Excluded Surface</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SSAO</td>
<td>Screen-Space Ambient Occlusion</td>
</tr>
<tr>
<td>Å</td>
<td>Angström</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This project fits in a cooperative program from the University of Texas at Austin, University of Beira Interior and Instituto Superior Técnico, named A-MOP - Algorithms for Macro-Molecular Pocket Detection. Laboratory drug design based on structure consists in predicting which drugs act as ligands to a given protein. When this happens, the connected protein suffers changes on its functionality. To check if a ligand can connect to a protein, it is important to analyze the molecule binding sites. The problems are to correctly identify those sites and being able to visualize it interactively. There are already some binding sites detection algorithms such as Fpocket[1] and LIGSITE[2], but there is still lack of proper visualization techniques to this purpose due to the size of macromolecules, which causes excess of information and low performance.

1.1 Motivation

The motivation behind this work is based on the need to have better molecule interactive visualization techniques, in which algorithms that select zones of interest like cavity detection algorithms can be used to simplify the visualization. With such information, it is possible to reduce the excess of a complete molecule representation’s data that are atoms farther away from selected areas. This can be achieved by granting different levels of detail depending on whether an atom belongs to the region of interest. This variant level of detail technique would also help users to focus their attention in zones that are considered important. The usage of low detailed surface in areas of the molecule also grants a faster rendering and therefore gaining a better frame rate and interaction capability.

A visualization that offer this capability of interaction and different levels of detail based on zones of interest would certainly help drug laboratories, since that drug search for a disease would be easier and faster by using the filtering and analyzing capabilities of this visualization.
1.2 Objectives and contributions

The goal of the thesis is to provide a level of detail based technique that allows the users to visualize regions of interest of a molecule. In this project, without loss of generality, the considered molecule features are protein cavities. The objective is to grant greater level of detail in those parts of the protein than remaining zones. The visualization should be interactive which requires a decent frame rate, about 25 frames per second, allowing the user to explore the molecule in real-time. To reach this goal, different techniques were studied, tested and benchmarked to determine which are the ones that allow such visualization.

The reference that we take as state of the art solution is Parulek et al. 2014 [3], which uses a hierarchy clustering that has a very high execution time. With our solution, we expect to avoid this problem and focus on the analysis of zones of interest, but with the same goal of allowing the users to visualize and explore those high quality molecules interactively.

1.3 Thesis Outline

Chapter 1 addresses the motivation and the objectives of this thesis. Chapter 2 describes the necessary theoretical background to understand the molecular visualization field and the state of the art solutions in section 3 and the implementation of the novel proposed solution in chapter 4. Chapter 5 addresses the experimental visual and performance tests executed and its discussion. Chapter 6 compares the solution obtained with the initial thesis goals.
Chapter 2

Background

Atoms are considered the basic and smallest unit that has element chemical properties. When tied together through chemical bounds formed by the sharing of electrons between atoms, they create sets of atoms which are called molecules. There are several types of molecules, including proteins which are large molecules which contains amino acids that are used in human body for many functions, namely antibodies that help to protect the system.

Proteins are a big target of studies for different fields, including drug design. Some proteins are capable of binding to foreign and malicious molecules and neutralizing them. The connection made between the antibody proteins and the viruses are called binding sites. One point of interest is to study where those binding sites are and their features, so it can be arranged a protein that can overcome the undesired molecule. The scientific community concluded that often the binding sites are localized in cavities, which are, often, also named as pockets.

2.1 Molecular Representations

In molecular representation, usually it is used implicit and parametric surface representations. Implicit surfaces are represented by equation 2.1, where the domain is three dimensional and the result is one dimensional. When the equation result is equal to zero, then it means that the input point stands on the surface. When the result is lower than zero, the input point is inside the surface and when it is higher than the isovalue $k$, it means that the point is outside the surface. The last two cases can be swapped, depending on the intended interpretation.

$$f(x, y, z) = k$$

(2.1)

Parametric representation, on the other side, has the ability to generate points if there are input parameters. It can be represented as in equation 2.2, where the left side of the equation represents the resulting point and the right side is a calculation that works with two parameters $u$ and $v$.

$$(x, y, z) = F(u, v)$$

(2.2)
Van der Waals surface consists in using an implicit sphere centered in atom center position and with the radius corresponding to the atom. More complex molecular representations, such as Solvent Accessible Surface (SAS) [4] and Solvent Excluded Surface (SES) [5] are also used in more detailed visualizations. Either one of these two implicit surfaces are calculated by simulating rolling a sphere with the solvent radius on the Van der Waals surface. As represented in figure 2.1, SAS is determined by the center of the solvent. On the other hand, SES uses the closest point to the Van der Waals surface.

![SAS - SES representation differences. Source: Krone et al. 2009](image)

As described in Zhang and Feng work [7], general implicit distance fields can also be used for molecules. Those fields are based on functions that take into consideration the distance between a point and the atoms centers. The value returned by this function is compared to a threshold value to determine whether the implicit function involves the point. There are different functions for this purpose such as blobby molecules, metaballs and soft objects. Each one has different arguments and result in slightly different representations. The main reason to compare those functions is due to the different computation complexities.

Blobby molecules is based on Gaussian distribution, where the standard deviation of the Gaussian curve represents the influence radius of each atom, while the Gaussian curve height represents the weight of each atom. The function is defined by equation 2.3, where $b$ is the standard deviation of Gaussian curve, $a$ is Gaussian curve height and $d$ is the distance from the point being tested to atom center. Although the square root operation is not necessary (since that in the formula the distance is squared), the exponential calculation is expensive. Other problem of this representation lays on the fact that the function does not have defined boundaries, which means that there are no limits where the functions should be tested when applied to computational algorithms.

$$f(d) = ae^{-bd^2}$$  \hspace{1cm} (2.3)

Metaball is a function that uses the distance between input point and metaball center and the metaball influence radius and the function is determined as represented in equation 2.4. In the equation, $a$ is the atom influence, $b$ is the influence radius of the atom and $d$ is the distance from the point being tested to
atom center. The advantages in the computation when compare to Blobby molecules is that the function is constraint in the interval \([0; b]\) and does not have an exponent operation, but the disadvantage is the computation of square root which is a computationally expensive operation.

\[
f(d) = \begin{cases} 
  a(1 - \frac{3d^2}{b^2}) & \text{if } d \in [0; \frac{b}{3}] \\
  \frac{a}{2}(1 - \frac{d}{b})^2 & \text{if } d \in [\frac{b}{3}; b] \\
  0 & \text{otherwise}
\end{cases}
\]  

Soft objects representation tries to avoid the disadvantages of Blobby Molecule and Metaballs and expressed as in equation 2.5. In the equation, \(a\) is the atom influence weight, \(b\) is the influence radius of the atom and \(d\) is the distance from the point being tested to atom’s center. Since all exponents of \(d\) are pair, there is no need to compute the square root and there is no exponential calculation either.

\[
f(d) = \begin{cases} 
  a(1 - \frac{4d^6}{9b^6} + \frac{17d^4}{9b^4} - \frac{22d^2}{9b^2}) & \text{if } d \leq b \\
  0 & \text{otherwise}
\end{cases}
\]  

Figure 2.2: Comparison between different implicit surfaces: blobby molecule, metaball and soft object. Sources: https://renderman.pixar.com/resources/current/RenderMan/appnote.31.html ; Kanamori et al. 2008 [8] ; http://www.cescg.org/CESCG-2002/MHasan/

2.2 Graphics visualization

In three dimensional graphical visualization, there are two main rendering methods: rasterization and ray tracing. Graphics pipelines that use rasterization are based on projecting three dimensional objects into the camera view plane. Rasterization is then performed, where pixels are drawn depending on the depth of each fragment. Rasterization rendering uses local illumination models, that consists in directly calculate surface illumination by using only light sources information.

OpenGL pipeline [9] in figure 2.3 is an example of a pipeline that involves rasterization. Shortly, OpenGL pipeline is based on the following: the vertex data is sent to the vertex shader, where operations
are made to each vertex and the output is defined by the user-programmed shader program. After those vertex-based operations, clipping is done to select vertices that belong to the camera frustum. Finally rasterization is applied resulting different fragments that can be modified using the fragment shader.

The other rendering method is ray tracing [10], a method that is famous for being a more realistic approach and more detailed representation. A simple ray tracing algorithm is based on rays that come out of the camera's position and calculated intersection with scene objects. Once it finds an intersection, it calculates the color and stores the corresponding pixel. In terms of illumination model, ray tracing can use global illumination, in which, rays are used to test if there is an intersection between the intersected point on object position and the light position, granting a higher level of realism.

Whitted's ray tracing algorithm [11] is based on creating rays from the virtual camera position, that have direction to each near plane pixel. Those rays intersect objects in the scene, and new rays can be generated, such as shadow feelers, reflection and refractions rays, depending on the intersected object material, as illustrated in figure 2.4.

When comparing these two methods, we can conclude that usually rasterization is usually faster since it is based on fast and simple transformations, but ray tracing provides a more realistic visualization since it follows an approach which is closer to reality and it also provides some features that are harder to get in rasterization such as global illumination. However, ray tracing suffers in terms of performance, mainly due to the number of rays that come out of the camera and their intersection calculations with the scene objects.
2.3 Implicit Surfaces

Rendering implicit surfaces visualization is divided into two different sets, the direct and the indirect methods [12]. Ray-tracing, particle-based, non-photo-realistic rendering and volume rendering are some examples of direct methods. Those direct techniques create a representation directly using the surface equation. The indirect methods involve a polygonization, which generates a mesh which can be rendered or used to other purposes, like 3d printing and 3d modeling since the mesh information can be stored.

When applied to high level of detail objects, polygonization loses its speed advantages and even fails to represent finer detailed objects. In terms of quality and fidelity to the real surface, ray tracing often grants more realism and is more accurate in surface tracking, which depends on precision parameters on iterative ray tracing techniques.

2.3.1 Polygonization

In terms of polygonization techniques, as described in Araújo et al. 2015 [12] there are three types of approaches, the spatial decomposition, surface tracking and inflation and shrink-wrap.

The spatial decomposition consists on recursive space divisions and storing the cells that contain the surface. The surface tracking approach starts by using an initial point of the surface and iterates, querying the surface and generating a mesh of polygons. Finally, the inflation and shrink-wrap methods consists on having an initial volume that expands or shrinks until it covers closest to the shape of the surface.

The most famous polygonization algorithms are variations of Marching Cubes and Marching Tetrahedron [13]. These two algorithms are based on having a 3d regular grid containing the surface and creating polygons where the surface intersects. Briefly and for simplicity, Marching Cubes algorithm [14] in two dimensions as represented in figure 2.5 is based in subdividing the space in a two dimensional...
regular grid and marking the vertices that contain the surface. Middle points of the outside edges (represented in purple in the figure) are connected generating the final mesh. The resolution of the regular grid defines the quality of the final result.

Polygonization techniques are used to generate a mesh that represents implicit surfaces that can be rasterized in a fast way. The main problems of using this method are the need to compute the mesh every time the surface changes, less precision the real implicit surface and high memory requirements to highly detailed and large surfaces since it needs to store the mesh vertices data.

![Marching cubes algorithm](http://www.cs.carleton.edu/cs_comps/0405/shape/marching_cubes.html)

### Figure 2.5: Marching cubes algorithm.

#### 2.3.2 Ray tracing

As described in section 2.2, ray tracing technique grants a higher level of realism sacrificing performance. Its main source of overhead lays on intersection calculations between rays and scene objects. Since we are dealing with implicit surfaces, it is usually described by high complexity functions. There are two surface finding methods: analytic and iterative[15].

Analytic methods are based on equation solving, which is faster than the other two methods, but there are no mechanisms to solve to equations higher than forth order polynomials. Iterative methods are famous since they are independent of the function used to describe the implicit surface. It consists in testing values directly on the surface function until the result is close to zero. The problem associated to this technique is the difficulty on setting initial parameters.

John Hart in 1994 [16] introduced sphere tracing which is an iterative surface finding mechanism, that uses functions that return the minimal distance from a point to the surface to set the step size of the marching points. The algorithm consists in a loop where it is calculated the distance function giving as input the marching point. If the distance is lower than a threshold value, that determine the level of approximation to the zero, then it is returned an intersection. Otherwise, as the figure 2.6 shows, the step size is set by the distance returned by the function and the loop continues until the intersection is found. This iterative method variation guides the surface tracking, using the minimal distance to the surface, it is guaranteed that there are no misses in the search and even optimizing the number of iterations needed.
2.4 CUDA framework

Since ray tracing is an algorithm with great capability to be parallelized, modern Graphics Processing Units (GPUs) capabilities can be used in order to grant better performance on the visualization in terms of time to generate an image.

Compute Unified Device Architecture (CUDA) framework is developed by Nvidia and has the goal to explore the computational power of GPUs. Modern GPUs are a great computational resource due to the number of cores when compared to Central Processing Units (CPUs), making it very useful for parallel computation.

In GPU programming [17], there are two terms that help distinguishing computation modes: host and device. Host is the processor that will have the main program execution and that will ask the device (GPU) to execute some work. GPUs resources can be used by launching kernels and allocating memory.

Kernels are functions that can be executed in the host but are mainly executed in the device, but the caller depends on the kernel type. If the kernel is defined as host, it will run in the host and it can be called by other host function. If a kernel is declared as device, then it can only be called by device code. If the kernel type is global then it can either be called by the host or device but will be executed by the device. The most interesting functionalities are the global kernels that can be used by the host to employ heavy work to the GPU.

When launching a global kernel, there are four main structures to consider. Threads are execution controls and have their individual state of the computation. Threads can be grouped in blocks, that can be multi-dimensional, and its main objective is to organize threads. Blocks can be groups in a multi-dimensional grid, which has the similar objective as the blocks, in this case, to organize blocks. Finally there is the concept of warp, which is a group of threads that are executed at the same time in a GPU core, while the execution path of the warp threads are all the same.

Parallelism in GPUs does not behave equally as in CPUs, since GPUs are designed for the architecture SIMD (Single Instruction Multiple Data), which means that all threads in a warp must follow the same computational path. Divergence is the name given to the situation when a thread takes another computational route, forcing a instruction serialization, harming the GPUs advantages.

In terms of memory architecture, there are some different layers: registers, shared, global, texture
and constant memories. Registers have a very small capacity that can be used directly by the thread. They are characterized by being the fastest memories but the scarcest in terms of memory size. Shared memory has a small capacity shared by each thread in a block that is optimized by caching. Global memory has a large capacity that is available to all the threads but reads and writes are very slow. Texture memory is a fairly big-sized and it is optimized for caching and interpolation. Finally constant memory is read-only medium-sized memory that cannot be changed during a kernel execution but that is faster than global memory accesses. There is also the term local memory that is usually misunderstood, since that it is, in fact, an optimization of global memory to avoid coalesced memory accesses.

When memory accesses are not coalesced, it means that there are memory accesses patterns do not follow the linear threads sequence. To avoid this problem, the memory accesses positions should depend linearly on the thread index, for example, having an array of $N$ elements, the first thread should read the first position of the array, the second thread should read the second position and so on. This feature improves the gains from caching capabilities.

![CUDA architecture](https://www.researchgate.net/figure/261069154_Schematization-of-CUDA-architecture-Schematic-representation-of-CUDA-threads)

**Figure 2.7:** CUDA architecture. The left side represents the threads organization. The right side illustrates the memory architecture. Source: https://www.researchgate.net/figure/261069154_Schematization-of-CUDA-architecture-Schematic-representation-of-CUDA-threads

### 2.5 Acceleration structures

Acceleration structures are used in ray tracing to minimize the amount of operations when calculating intersections. In this project, there are some variations since it is being used point marching instead of using directly the rays. The scope of this project does not involve molecular dynamics, this means that the positions of the atoms are constant during all execution time. Due to this assumption, the construction of the acceleration structures will not be address in great detail.
Acceleration structures for ray tracing are subdivided in two main categories: acceleration structures based on aggregation and acceleration structures based on partition [18]. Acceleration structures that use aggregation, such as BVH (Bounding Volume Hierarchy) consist in grouping objects, granting that one object belongs to exactly one point in the hierarchy. The other type of acceleration structures, that are based on spatial partition, such as grids and BSP (Binary Search Partition) Trees, usually uses surfaces align to the axis. Those surfaces are subdivided, granting that one point in space belongs exactly to one cell of subdivided surface, but one object can be in multiple cells.

Taking into consideration the GPU computation features describe in previous sections, it is important to minimize the memory requirements since it can determine in which memory type the acceleration structure will be stored, affecting the speed of stores and loads. To generalize, all acceleration structures were stored in global memory, in linear arrays.

In this thesis, the acceleration structures were used to collect the nearest neighbor atoms around the point being tested in the sphere tracing marching. This is the major overhead in the ray tracing computation, affecting the capability of interaction of the user with the visualization.

### 2.5.1 Bounding volumes hierarchy

Bounding volume hierarchy is an aggregation-based acceleration structure that consists in creating bounding volumes that warp the objects of the scene and then grouping those objects with volumes that contain groups of smaller volumes, causing that the root volume will contain all objects of the scene. This structure can minimize the amount of intersection calculations since, for example if the test to box $B$ in figure 2.8 is positive then all box $C$ objects will not be tested.

![Figure 2.8: Bounding volume hierarchy example. At the left, the scene object placement. At the right, the resulting tree.](https://commons.wikimedia.org/wiki/File:Example_of_bounding_volume_hierarchy.JPG)

**Construction**

For the construction of the BVH, we utilized an approach based on Tero Karas work [19], which uses Morton codes to build a binary radix tree. The Morton codes are strings of bits that are created by using the coordinates of a point. The binary radix tree is constructed by calculating the longest common prefix between Morton codes, which results in a tree with Morton codes ordered by point proximity. This tree
is used to create the bounding volume hierarchy tree. To improve parallel construction performance, the parallel threads follow the radix binary tree, starting on the leaf nodes, advancing towards the root.

Although it was not one of the goals of the thesis, we used this more sophisticated approach more suitable to reconstruct the tree since there was already work done by a colleague of the department, which may help advancing future work over this solution.

Traversal

The traversal is similar to typical ray-BVH traversal, with the only difference of using point-volume intersection calculations instead of ray-volume, simplifying the operations since point-volume calculations are much simpler than ray-volume. More details about the retrieval of the atoms during the traversal will be given in the implementation chapter.

To minimize divergence in the traversal of the BVH, we used a stack-based traversal which adds nodes to be processed during the traversal, continuing the computation while the stack is not empty.

Considerations

Since the BVH space complexity of $O(2^n - 1)$, $n$ being the number of atoms, it can be considered, in some situations, to be stored in faster memory types in GPU architecture, increasing the speed of loads during the neighbor atoms retrieval.

Atoms that belong to a binding site are very close together, this means that the bounding volumes will have a higher amount of overlapping, forcing some almost redundant tests diminishing the overall performance.

2.5.2 Uniform Grids

Uniform grids are very simple partition-based structures that consist in dividing the space in uniform cells as shown in figure 2.9, storing references to overlapping objects in each cell.

In this work, the uniform grid has a cell size of $S$, that corresponds to the solvent radius and contains a fixed maximum allowed number of atoms references.

Construction

In the construction of the grid, it is calculated the boundaries of all the objects are calculated and the space is divided depending in the cell size. Then, for each atom it is tested if the influence radius intersects the cell and if the test is positive then the atom id is added to the cell of the grid.

Traversal

Like BVH, point-grid traversal is highly simplified when compared with ray-grid traversal. Taking in consideration the beginning of the grid, the cell size and the point being tested it is trivial to get the index of
the cell in which the point is. More details about those calculations will be presented in the implementation chapter.

Considerations

Uniform grids require much more memory than BVH, forcing to store it in global memory most of the times. The advantage is the fast traversal when compared to BVH, quickly finding the cell that contain the references to the neighbors.

2.5.3 K-d tree

K-d tree is a partition-based acceleration structure that is usually used for nearest neighbor search algorithms.

Construction

K-d tree construction recursive algorithm starts by selecting the root node where all points are considered, then, it proceeds to select one axis. There are several heuristics to decide which axis is chosen but in our case we have chosen that it alternates depending on the depth of the node being computed. Next, it calculates the median of the position values in that axis and stores that value as the division value. Two lists are generated, one with points below the median that will be used by the left child node and the other above the median which will be used by the right child node. The final visual representation of the structure is shown in figure 2.10.
Traversal

Point-K-d tree traversal consists, simply, in checking if the point is below or above the division of the current node to decide to which node the traversal will continue. Although, for this purpose, it is necessary to gather points within a determined range. To fulfill this requirement the condition is modified. Instead of checking only if the position against the value of the node, now it is necessary to check if the minimum bound is below the axis and the maximum bound is above the division. If both conditions occur, the traversal is done in both left and right child. If the current node is a leaf, then it is checked if the point is closer to the corresponding atom than the solvent diameter.

Like BVH, we used a stack-based approach in order to optimize parallel performance.

2.6 Screen-space ambient occlusion

In order to improve depth perception in a scene with an high amount of objects very close to each other, it was used a technique introduced by Martin Mittring [20], in Crysis 2 video game illustrated in the figure 2.11, named screen-space ambient occlusion (SSAO). The main idea of the algorithm is to darken fragments which are surrounded by others that are closer to the camera. The term occlusion is used to quantify the amount of light that hit the fragment. An occluded fragment is hit by less light, while a less occluded is exposed to more light.

The method calculates, in the fragment shader, the value of occlusion for each fragment. The original method is based on simulating a sphere centered in the sampled fragment as presented in figure 2.12, generating a random points inside that sphere and calculating the quantity of points that have higher and lower depth than the values in the depth buffer.

The community has already made some improvements in the Crytek method by swapping the sphere by a normal-oriented hemisphere represented in the figure 2.13. This modification lowers the amount of random points needed to create good results and also ignores the random points that are behind the fragment, which are not necessary to the calculation of the occlusion. This change was presented by
2.7 Shading models

Shading models are used in computer graphics to simulate how light affects a color of a material. Flat shading is one of the simplest ways to perform shading, since it uses one normal vector per face to compute the resulting color. Gouraud shading applies an illumination model to each vertex of the face and then interpolates all the vertex results. Although this technique helps to smooth the shading, it still
has some artifacts.

Phong in 1975 [22] presented one of the most used shading models, which consists in interpolating normal vectors of the objects vertices for each pixel, finalizing by applying the illumination model. Blinn in 1977 [23] improved Phong algorithm, avoiding the repeated calculation of the dot product between the surface reflected vector and surface to eye vector, by introducing the half vector which is calculated by using the surface to light vector and surface to eye vector.

The figure 2.14 shows the visual difference between the shading models referred.

Figure 2.14: Comparison between flat shading, Gouraud shading and Blinn-Phong shading in order. Source: http://diccan.com/Images/gouraud_phong_flat.jpg
Chapter 3

State of the art

This chapter is divided into two sections: state of the art on molecular visualization and techniques that use level of detail.

3.1 Molecular visualization

Julius Parulek and Ivan Viola [24] proposed a solution to compute Solvent Excluded Surface (SES) representation without any precomputations. Due to this improvement, SES representation ray tracing grants some level of interactivity, allowing to change parameters values in real-time. Their solution generates a function that determines the minimal distance from a point to the surface. The method starts by collecting the closest atoms to the point being tested. Depending on the number of closest atoms and the intersection between pairs and triples of atoms, there is a different approach to build the final function. In the case where there are no atoms within the range, the resulting function is determined depending on the solvent radius. In case there is only one atom in the closest set, the function used is Van der Walls. When exactly two atoms are in the set, it is applied a toroidal implicit function, and finally when three or more atoms exist, it is used a more complex spherical triangle implicit function. The figure 3.1 shows the final result on this solution and the toroidal and spherical triangle surface.

![Figure 3.1: Parulek et al. 2012 [24]: (a) Toroidal implicit function; (b) Spherical triangle implicit function; (c) Final visual result.](image)
Although SES representation is seen as the most reliable molecular representation, Parulek and Brambilla [25] proposed a representation with visual results close to SES as shown in figure 3.2. The advantage of this new algorithm is that it has linear complexity, which is better than previous solution [24] cubic complexity. Since that there is still a significant visual difference between this representation and SES (which is considered by the community as one of the more suitable for molecular analysis), this solution was not used.

Figure 3.2: Parulek et al. 2013 [25]: Comparison between representations. On the left side is Parulek and Brambilla modified SES representation and on the right is the original solvent excluded representation.

Kanamori et al. 2008 [8] proposed a solution to render a good amount of animated metaballs with good frame rate. Their solution starts by calculating intersection points to each visualization ray to spheres that represent each metaball maximum range. This information allows to define limits to the ray-surface intersection. If there was intersection with only one sphere, then it is only computed a ray-sphere intersection. In case of multiple intersection, a Bézier form equation is formed and solved using Bézier clipping method. The innovative part of their solution is the metaball selection method to use on isosurface test. Their method consists in having textures with intersected spheres data, and efficiently updating the values. The method can also be applied for molecular visualization as demonstrated in figure 3.3.

In 2010, Singh and Narayanan [15], proposed adaptive marching points algorithm, which is based on adapting marching point step size according to the distance to the surface. The first step is to get the two intersection points between the ray being processed and the near and far planes. Then, the starting point of the marching point algorithm is set to the near plane intersection point and the step length is set to a default value. A cycle is iterated, checking first if the marching point has not exceeded far plane intersection point. After that, a root finding method is performed in order check if there is a zero of the implicit function between the current marching point and the next one. If a zero is found, then this interval is isolated and solved.

Otherwise, the marching point is incremented by using a function that depends on the distance to
the implicit surface as described in equation 3.1 and figure 3.4, where \( b \) is the default step size, \( S(p(t)) \) is a function that returns the distance from the point \( p(t) \) to the closest point of the implicit surface and \( D_f \) is the direction of the ray. Finally, \( T_1, T_2 \) and \( T_3 \) are threshold values. The step size can be a quarter of the default length if the region is a silhouette, it can be half of the default \( b \) if the point distance to the surface is below a defined threshold \( (T_1) \), it can have the double of the default value if the distance is higher than a larger threshold \( (T_2) \) and it is set as the default length otherwise.

\[
stepLength = \begin{cases} 
\frac{b}{4} & \text{if } |S(p(t))| \leq T_1 \text{ and } |\nabla S(p(t)) \cdot D_f| \leq T_3 \\
\frac{b}{2} & \text{if } |S(p(t))| \leq T_1 \\
2b & \text{if } |S(p(t))| > T_2 \\
b & \text{otherwise}
\end{cases}
\]

(3.1)
3.2 Level of detail based techniques

Parulek et al. 2014 [3] proposed a solution based on applying different implicit surface representations based on the distance to each atom. Atoms nearest to the camera are rendered with Solvent Excluded Surface representation, the next atoms are rendered with Gaussian kernels and finally the more distant atoms are rendered using spheres. Linear interpolation is used to connect the representations. The more distant atoms are grouped in clusters to reduce memory requirements and to speed up the rendering process. The precomputations of the clustering algorithm however takes long time, that we avoid in our solution. The level of detail solution is displayed in figure 3.5.

![Figure 3.5: Parulek et al. 2014 [3]: Phospholipase bound to lipid membrane with 2808 clusters of atoms.](image)

TexMol [26] is a molecular visualization software that also applies a level of detail based technique, although in a different way when compared to last method described. Molecular data is classified and generates an hierarchy. The lowest level of the hierarchy has individual atoms, which are represented with Van der Waals spheres. The second lowest level, residues, which include amino acids and nucleotides, which are represented by the minimum bounding sphere that contains all the atoms that compose the structure. The third level has the molecular secondary structure, which contains helices, turns and sheets. Helices and sheets are represented with sets of cylinders and helices. Turns are displayed as the atoms cluster that compose it. Finally, the last and higher level are backbone chains that are incarnated as the set of atoms that define the backbone chain. Unlike previous methods, this level of detail technique does not apply automatically depending on the molecular structure and user input. The solution creates images like the one exhibited in figure 3.6.

Lee et al. 2006 [27] described a polygonization level-of-detail technique to allow good frame rate when rendering large size molecules. In the first step of their approach, a set of polygons are created for each molecular model. Then, it is applied a molecular surface generation algorithm that returns simplified models of some parts of the molecule. The important molecular parts such as active sites are not subject to this simplification algorithm. Those models are classified and stored. The models to be used are determined by analyzing the running environment capability and by the distance between the
camera and molecular models. They also adapt their models selection according to the runtime frame rate. If the frame rate is already acceptable for an interactive scene, it avoids lowering the level of detail. If it is not, then it loads lower detailed models. The mesh simplification is shown in figure 3.7.

Figure 3.7: Lee et al. 2006 [27]: (a) (b) and (c) shows the mesh simplification. (a) has 533,774 polygons; (b) has 314,696 polygons; (c) has 100,104 polygons
Chapter 4

Implementation

4.1 Architecture

This work explores a novel approach for interactive visualization of macromolecules, in which only regions of interest are rendered with high resolution, leaving the remaining regions with lesser visual details. The solution combines the advantages of polygonization and ray tracing. The set of atoms detected/classified as regions of interest are rendered using ray tracing and a high precision representation, such as Solvent Excluded Surface. The remaining atoms are rendered with low quality polygonization and rasterization, just to give a molecular context to the user.

The proposed approach of variant levels of detail will have two main advantages: (i) visualization will help users to focus their attention in regions that really matter to the purpose of predicting connection sites between proteins; and (ii) since the potential regions of interest only cover a low percentage of the whole molecule, and those are the ones that will be slower to compute (since that will be based on calculating a complex function for each marching point), it will save computation time, granting better frames per second, which results in more interactivity. Obviously, other molecular features that selects different atoms within a molecule can also be considered to benefit from the level of detail technique. Figure 4.1 illustrates the desired type of result.

With the rise of the GPGPU (general purpose GPU) computations, and taking into consideration the need for cheap, massive computations to evaluate scalar fields (i.e. molecule-describing function) and surfaces for molecules with millions of atoms, we used modern GPUs like Nvidia Maxwell 6000, Nvidia GeForce GTX970 and Nvidia GeForce GTX980 Ti to reach our objectives.

The list of requirements for the project is the following:

1. Parsing a protein database file in order to gather data about atoms position and type
2. Use databases to select zones of interest
3. Render low resolution spheres in regions of low interest atoms positions
4. Convert atoms to billboards at further away distances from camera
5. Use sphere tracing to render solvent-excluded surface for interesting atoms

6. Mix both rendering methods

7. Allow to explore the scene through interaction

The general algorithm follows the steps represented in algorithm 1. The initialization that is described in section 4.2 occurs only once in the execution of the algorithm and it happens in the beginning of the algorithm. The HandleUserInteraction step takes the user input, either keyboard or mouse input, to change the camera position and orientation, that is updated to CUDA code using cudaMemcpy function in the beginning of the draw event. Besides the camera interaction, the user can also swap between acceleration structures using keyboard input. This is specially useful to compare performance between the acceleration structures. The draw event is described in section 4.3 is constantly occurring to display the resulting images for the visualization.

The class domain model is displayed in figure 4.2, containing as the most important classes, the MolecularVisualization, Raytracer, Rasterizer and Scene.

The main function of the program starts by instancing MolecularVisualization object and setting up GLUT and GLEW libraries, defining glutDisplayFunc, glutIdleFunc, glutReshapeFunc, glutMouseFunc,
Algorithm 1: Algorithm overview

1: **Initialization**
2: **while** running **do**
3:  **HandleUserInteraction**
4:  **Draw**
5: **end while**

Figure 4.2: Class domain model

glutPassiveMouse callbacks, redirecting to MolecularVisualization functions to be handled. Then, the MolecularVisualization initialization function is called and glutMainLoop that starts the OpenGL main loop, which consists in checking the events in the queue and execute its callback functions defined in the previous setup.

### 4.2 Initialization step

#### 4.2.1 MolecularVisualization class

In the initialization function of MolecularVisualization class, CUDA framework is initialized by checking if the compute capability has the minimum requirements. Next, it propagates the initialization function to its child objects: Raytracer, Rasterizer and Scene.

#### 4.2.2 Scene class

In the Scene initialization function, PDBParser is called to load the input files: the Van der Waals radius compilation file, two protein database files and the camera position and orientation file. It starts by
parsing the Van der Waals file, storing every element and its corresponding radius for later use. Then, it reads the camera properties file and sets those values to the camera object. Afterwards, the whole molecule .pdb file is read, saving in a list the id, position and then the radius, by looking up on the stored Van der Waals radius list, of the atoms existing on the file. Finally, it parses the regions of interest files, creates a new list and swaps the atoms selected on the file, from the whole molecule atoms list to the new ROI list.

    The instruction control returns to the Scene initialization function which initializes the light position, the light color and the ROI atoms to the CUDA code. The atoms are stored in a CUDA texture, since it exploits the caching capacity and the read speed, and are not changed anymore during the execution since we are not considering molecular dynamics.

    In the end of the scene initialization function, it also initializes the camera, setting up a perspective projection camera with the input position and orientation of the read file. It also passes to CUDA code the camera current position, the camera near plane bottom-left corner, the right vector and the up vector, that are illustrated in figure 4.3, in order to later be calculated the rays origin and direction. To calculate the near plane bottom-left corner position, we need to find the width and height of the plane, by using equation 4.1 and 4.2, where $\alpha$ is the horizontal opening angle of the camera, $\beta$ is the vertical opening angle of the camera and $d$ is the distance between the camera position and the near plane.

\[
viewWidth = 2 \times \tan\left(\frac{\alpha}{2}\right) \times d
\]  

\[
viewHeight = 2 \times \tan\left(\frac{\beta}{2}\right) \times d
\]

![Figure 4.3: Illustration of the frustum variables](image-url)
Then, we translate a point from the camera position $\text{eye}$, by using the camera orientation vector $\mathbf{at}$ and the distance to the near plane $d$ as represented in equation 4.3.

$$\text{viewCenter} = \text{eye} + d \mathbf{at} \quad (4.3)$$

Afterwards, we translate a point the previous $\text{viewCenter}$, by using the camera up and right vectors and the $\text{viewWidth}$ and $\text{viewHeight}$ of the near plane by using equation 4.4.

$$\text{viewDownLeftCorner} = \text{viewCenter} + \frac{\text{viewHeight}}{2} \mathbf{-up} + \frac{\text{viewWidth}}{2} \mathbf{-right} \quad (4.4)$$

To calculate the right and up vectors that can be used instantly by the CUDA kernel, we scale the vector so that each pixel only has to apply dot product those vectors by its index to find the initial position we follow equations 4.5 and 4.6.

$$\mathbf{camUp} = \frac{\text{viewHeight}}{\text{resY}} \mathbf{up} \quad (4.5)$$

$$\mathbf{camRight} = \frac{\text{viewWidth}}{\text{resX}} \mathbf{right} \quad (4.6)$$

Those parameters will allow that CUDA kernel, for each pixel, to calculate the ray direction and initial position, using equation 4.7, allowing to start the sphere tracing algorithm at the correct position for each pixel.

$$\text{initialPosition} = \text{viewDownLeftCorner} + \text{pixelIndex} \cdot \mathbf{camRight} + \text{pixelIndex} \cdot \mathbf{camUp} \quad (4.7)$$

### 4.2.3 Rasterizer class

The **Rasterizer** class reads and initializes all fragment and vertex shaders needed for the execution: billboards shaders, depth shader, SSAO blur shader, SSAO geometry shader, SSAO lighting shader, SSAO occlusion shader and a simple texture displaying shader. Afterwards it issues the model loader to load the sphere model used in the visualization. This sphere model is both used to the near atoms outside the zones of interest and to create the simulated region of interest influence surface used to improve sphere tracing algorithm, this is explained in more detail in the section 4.4.1. Lastly, it initializes the vertex buffer objects that will be used to render the billboard atoms and also initializes all the framebuffer objects used to control the buffers used on the draw event.

### 4.2.4 Raytracer class

In terms of initialization of the **Raytracer** class, it only stores the combinations twelve choose three, resulting in one hundred and twenty combinations that will be used to calculate the spherical triangles in the SES the sphere tracing algorithm. All the remaining initialization to the ray tracing algorithm is already done in the Scene class, which loaded the region of interest atoms and the camera parameters.
It propagates the initialization request to the *AccelerationStructures* class.

### 4.2.5 AccelerationStructures class

The class that assumes the responsibility to create the acceleration structures does most of its work in the program initialization. It constructs the uniform grid, the bounding volume hierarchy and the kd-tree, all those acceleration structures will be transferred to the GPU and the one in current use is used to improve the neighbor atoms collection.

#### Uniform grid

To create the uniform grid, the minimum and maximum positions of the atoms of interest are stored. Since that we need to have the influence surface those values and incremented and decremented, correspondingly, by $2 \times S + R_{\text{max}}$, $S$ being the solvent radius and $R_{\text{max}}$ being the maximum radius of the ROI atoms. Afterwards, the uniform grid initial position is set to the minimum position found and the side size is set to the solvent radius, although it can be changed depending on how much memory is available and how good is the performance. Each cell can have a fixed upper limit amount of atom identifiers, in our test this value is set to 60 identifiers.

The atoms in the regions of interest are iterated and checked on which cells they are contained. When the test passes, the atom identifier is added to the first available slot. In this case, since this is the default acceleration structure, it is transferred to the CUDA code for being used in the search of neighbors atoms.

#### Bounding volume hierarchy

For the bounding volume hierarchy it was used a more sophisticated algorithm which is already suitable to be reconstructed every frame, since there was already some work done in this area by a department colleague and it will also be some work in advance for the future if this visualization is extended to support molecular dynamics.

The algorithm starts by using the minimum and maximum position values of the atoms to find the dimensions of the box which is formed by these positions. For each atom, it is calculated the normalized position by using equation 4.8, where $n$ is the normalized position of a atom, $\text{max}$ is the maximum atom position and box are the box dimensions: width, height and depth. This normalized position is converted to Morton codes and all the codes are stored in a list. The list is then sorted by using radix sort algorithm and sent to GPU, along with default bounding volume hierarchy, by storing $2N - 1$ BVH node structures, where $N$ is the number of ROI atoms. These transfers are done using cudaMalloc and cudaMemcpy functions.

$$(n_x, n_y, n_z) = \frac{(\text{max}_x, \text{max}_y, \text{max}_z)}{(\text{box}_w, \text{box}_h, \text{box}_d)} \quad (4.8)$$

The kernel responsible to build the BVH from the Morton codes starts by the leaves of the tree.
Determines the direction of the range by checking the longest common prefix from the two parent nodes. Then it computes the upper bound for the length of the range and finds the other end and split position using binary search. It outputs the child pointers and computes the bounding boxes for all the hierarchy nodes. The BVH is now built and already stored in the GPU, ready to use.

**K-d tree**

For the construction of the K-d tree, \(2^N - 1\) tree nodes are allocated, being \(N\) the number of atoms in regions of interest. A cycle starts on the root node and a list of all the atoms to consider. The algorithm starts by choosing an axis based on the current depth, alternating between \(x\), \(y\) and \(z\). The axis position values of the atoms are used to calculate the median which will determine the plane which will divide the atoms into two groups, one for each side of the plane. The algorithm is recursive, for computing the left side of the tree, the list of atoms that are on the lower side of the plane are sent, and the same for the right side. For each node it is stored the median value, the axis, references to both children and the atom identifier if it is a leaf node. The tree is completed stops when there is no atom left to consider. Finally it is transferred to GPU to be used by the ray marching kernel.

### 4.3 Draw event

#### 4.3.1 MolecularVisualization class

The *MolecularVisualization* class is responsible for managing the resources to enable the connection between the two techniques: rasterization and ray tracing. As represented in the pseudo-code algorithm 2, it starts by invoke *Rasterizer* class to do its work first then it maps a color buffer with the result of the rasterization technique, a depth buffer with the depth of the influence surface created by the ROI atoms and a depth buffer with the values of rendering the non region of interest areas. Those buffers are used by the ray tracing algorithm as described in section 4.3.3. The *Raytracer* class computes its work that is stored in a PBO which then is used to initialize a texture that is rendered using a simple texture to screen shader and the result is flushed to the screen.

Algorithm 2 MolecularVisualization draw pseudo-code

1: Rasterizer.Draw()
2: MapCUDAResources()
3: Raytracer.Draw()
4: UnmapCUDAResources()
5: BindDefaultFBO()
6: BindPixelUnpackBuffer()
7: CopyColorBufferToTexture()
8: EnableTextureShaders()
9: ClearBuffers()
10: DrawTexture()
11: SwapBuffers()
4.3.2 Rasterizer class

Following the pseudo-code draw algorithm 3, the Rasterizer draw function starts by using two lists of atoms stored initially in the Scene object, the list I which contains the atoms of zones of interest and list A with the remaining atoms. It also creates two new lists, B that will contain the atoms that should be rendered as billboards and C that contains the atoms that should be rendered as a simple mesh sphere. To fill those lists, the A list is iterated, and for each atom it is checked if the distance between the camera and the atom is smaller than a limit L.

Before rendering the atoms outside the zones of interest, the ROI influence surface, being S the solvent radius, are rendered with a simple depth shader, that outputs the normalized depth in the color buffer, the framebuffer object used to make this rendering has a texture bound to the color buffer, resulting in the texture having to depth encoded as RGB (red, green and blue) values.

Then it proceeds to use the list of atoms C, which contain the information of position and radius of the atoms that should be rendered as a mesh sphere, to be rendered and applying SSAO algorithm. As explained before, there are multiple framebuffer objects used on this technique. The first is the geometry FBO, which is the only one that receives as the input the original geometry to be rendered, and outputs multiple textures: the position-depth texture, the normal texture, the albedo texture and the a second normalized depth texture. In order to mix the billboard drawing and the spheres drawing, the depth buffer of the geometry framebuffer object is copied to the billboards framebuffer object for later use.

The first texture, as the name implies, has the values of the fragment position as RGB and the normalized depth texture that uses the real frustum far plane distance to be calculated. The frustum far plane distance in this rendering is set to the value of the distance from which the atoms start to become billboards, that is set as 60.0 Ångström as default. This first normalized depth will be used by the SSAO algorithm, unlike the second that will be used to optimize the ray tracing algorithm. The second texture, normal texture, contains the interpolated normal vectors of the fragments. The last texture will be used by the ray tracing algorithm only, and only differs from the first texture depth values because the far plane distance is set to the maximum of the camera, which is 200.0 Ångström as default.

Using all the textures generated by the previous rendering except the last normalized depth texture, the occlusion is calculated by using the corresponding framebuffer object and shaders. It renders a quad since that all the input is stored in textures and the result is stored in a single texture, where the RGBA components are stored with the occlusion value.

The occlusion is then blurred by using a new framebuffer object and shaders again. It iterates through samples of the random values generated in the SSAO class initialization, incrementing a result value and in the end of the cycle it is divided by the area where the random points were generated. The output is stored in a texture that has the RGBA values encoding the result of the calculations made.

The final SSAO step, lighting, receives as input all the SSAO textures generated by the geometry step and the blurred occlusion texture. Using Blinn-Phong shading model, the color components are calculated and are attenuated by using fog technique. The ambient component is multiplied by the occlusion value, that will result in Blinn-Phong shading with occlusion texture. The color buffer is then copied to the billboard framebuffer object.
The draw function ends by rendering the billboards of the list $B$, using the billboard framebuffer object, which contains the copied depth and color buffers. The billboard shaders are enabled and using instancing are rendered. The rendering is optimized since that depth buffer already has values from the spheres rendering, which will result in discarding all the fragments which are in positions where the depth buffer already had values, since every sphere is closer than every billboard.

**Algorithm 3** Rasterizer draw pseudo-code

```plaintext
1: $I \leftarrow \text{ROI}\text{atoms}$
2: $A \leftarrow \text{NROI}\text{atoms}$
3: $B \leftarrow \emptyset$
4: $C \leftarrow \emptyset$
5: foreach atom $\in A$ do
6:   if $\text{DistanceToCamera}(\text{atom}) < L$ then
7:     Append(atom, $B$)
8: else
9:     Append(atom, $C$)
10: end if
11: end foreach
12: $\text{BindDepthFBO}()$
13: $\text{ClearBuffers}()$
14: $\text{EnableDepthShaders}()$
15: $\text{Render}(I)$
16: $\text{BindSSAO}\text{GeometryFBO}()$
17: $\text{ClearBuffers}()$
18: $\text{EnableGeometryShaders}()$
19: $\text{Render}(C)$
20: $\text{CopyDepthBufferToBillboardFBO}()$
21: $\text{BindSSAO}\text{OcclusionFBO}()$
22: $\text{ClearBuffers}()$
23: $\text{EnableSSAO}\text{OcclusionShaders}()$
24: $\text{RenderQuad}()$
25: $\text{BindSSAO}\text{BlurFBO}()$
26: $\text{ClearBuffers}()$
27: $\text{EnableSSAO}\text{BlurShaders}()$
28: $\text{RenderQuad}()$
29: $\text{BindSSAO}\text{LightingFBO}()$
30: $\text{ClearBuffers}()$
31: $\text{EnableSSAO}\text{LightingShaders}()$
32: $\text{RenderQuad}()$
33: $\text{CopyColorBufferToBillboardFBO}()$
34: $\text{BindBillboardFBO}()$
35: $\text{EnableBillboardShaders}()$
36: $\text{Render}(B)$
```

### 4.3.3 Raytracer class

The *Raytracer* class has its major work done in the GPU, but before invoking the CUDA kernel, the class computes the thresholds that will be used in the sphere tracing algorithm. In order to optimize the
frame rate while the user is exploring the scene, we take into account the time difference between the last camera exploration input (camera look around or camera movement) and the current frame, by using the equation 4.9, where $T_{diff}$ is the time difference calculated by the equation, that is clamped between 0 and $T_{focus}$ which is a constant that sets the time difference needed to reach the thinnest threshold, in our case, we set it to 500 milliseconds The $T_{input}$ is the timestamp of the last input interaction and $T_{now}$ is the current time.

$$T_{diff} = \min(T_{focus}, T_{input} - T_{now})$$ (4.9)

To get a linear interpolation on thresholds, based on the time difference calculated, we use equations 4.10 and 4.11, where $SurfaceT_{final}$ is the result value that will be sent to the kernel as the surface threshold value, $SurfaceT_{thinnest}$ is the thinnest possible threshold of the surface and $SurfaceT_{thickest}$ is the thicker threshold allowed. The second equation is identical but applied to the threshold used in the Newton-Raphson method, that will define where to stop the approximation. $NewtonT_{final}$ is the threshold that will be used by the kernel, while $NewtonT_{thinnest}$ and $NewtonT_{thickest}$ are the thinnest and thicker thresholds allowed to the Newton-Raphson threshold. Once the thresholds are calculated, the program allocated the memory required to be stored in the CUDA global memory and are copied using `cudaMemcpy` function.

$$SurfaceT_{final} = \frac{SurfaceT_{thinnest} - SurfaceT_{thickest}}{T_{focus}} \times T_{diff} + SurfaceT_{thickest}$$ (4.10)

$$NewtonT_{final} = \frac{NewtonT_{thinnest} - NewtonT_{thickest}}{T_{focus}} \times T_{diff} + NewtonT_{thickest}$$ (4.11)

To call the sphere tracing kernel, we need to define how many threads there are and how are they structured. The first approach was to create a simple direct mapping, where the thread with identifier one would compute the first pixel, the second thread would compute the second pixel and so on, following the representation in figure 4.4. Taking into account that divergence is a big problem in GPU computation, later the threads were organized in rectangles as represented in figure 4.5. The figures have thickest blocks than the ones used to facilitate the comprehension.

In the case where the blocks are extended horizontally, there is a higher amount of pixels that differ their path in the sphere tracing computation, for simplification, if we consider the pixels that did not draw anything and the ones that found the surface, we can see that there is more change among the linear blocks than the rectangle organized blocks. On the other hand, the global memory accesses that were aligned in the case where we had the linear blocks, now it suffers some amount of misalign accesses in the rectangular blocks, but by experiments, the gain was much higher than the loss, resulting in gains around the 25 milliseconds per frame. The dimensions of the blocks used were 8 by 8 threads and the number of blocks are calculated by dividing the resolution $x$ and $y$ components by the dimensions of each block.

In the kernel invocation it is passed the current acceleration structure identifier, the thresholds and
the reference to the CUDA mapped pixel object buffer where the result will be written.

4.3.4 Sphere tracing

This section will describe the algorithm that is performed in the sphere tracing CUDA kernel. The algorithm described is done for every pixel of the graphical window.
Backed by the pseudo-code 4, it is calculated the mapping of the thread index to the pixel index that it will handle, due to the utilization of a non linear thread arrangement. Following equation 4.12, the x coordinate of the pixel, \( \text{pixel}_x \), is calculated by multiplying the index of the block, \( \text{blockIdx}_x \), and the width of the block, \( \text{blockDim}_x \), and finally adding the thread index x component, \( \text{threadIdx}_x \). The same is done in equation 4.13 to the y component, using the corresponding variables, \( \text{pixel}_y \), \( \text{blockIdx}_y \), \( \text{blockDim}_y \) and \( \text{threadIdx}_y \).

\[
\text{pixel}_x = \text{blockIdx}_x \times \text{blockDim}_x + \text{threadIdx}_x \tag{4.12}
\]

\[
\text{pixel}_y = \text{blockIdx}_y \times \text{blockDim}_y + \text{threadIdx}_y \tag{4.13}
\]

The final mapping is computed by using equation 4.14.

\[
\text{pixelIndex} = \text{pixel}_x + \text{pixel}_y \times \text{res}_x \tag{4.14}
\]

This index is used to access the input depth textures, the depth of the influence surface and the depth of the billboards and spheres rendered before in the \textit{Rasterizer}. The first depth is used to define the starting distance of the sphere tracing algorithm and the second is used to limit the range. In case the starting depth in the texture is the far plane, it means that there was no influence surface detected and so there is no need to compute the sphere tracing in that pixel. The same happens when the limit depth is lower than the starting depth, that means that there is a sphere or billboard closer that the influence surface, hiding it. Even in these cases, the kernel writes using the pixel index again, on the pixel object buffer, to clear previous results.

Afterwards, if the control sequence continues, it loads the thresholds that were stored in the global memory and also calculates the ray origin point by the equation 4.15, using the camera parameters: the view bottom-left corner position (\( \text{initialPos} \) from equation 4.7), the right vector (\( \text{camRight} \) from equation 4.6), the up vector (\( \text{camUp} \) from equation 4.5). Using the ray origin and the camera position (\( \text{eye} \) used in equation 4.3) it is calculated the ray direction by applying the normalization of the vector that results from the subtraction of the ray origin and the camera eye as it is represented in equation 4.16. The procedure that handles the tracing of the ray is called and the result is stored in the unpack pixel buffer \( \text{resultBuffer} \).

\[
\text{rayOrigin} = \text{initialPos} + \text{pixel}_x \cdot \text{camRight} + \text{pixel}_y \cdot \text{camUp} \tag{4.15}
\]

\[
\text{rayDirection} = \frac{\text{rayOrigin} - \text{eye}}{\|\text{rayOrigin} - \text{eye}\|} \tag{4.16}
\]

The \textit{TraceRay} procedure (algorithm 5), receives as input the ray, the thresholds, the selected acceleration structure identifier, the starting depth and the limit depth and outputs the calculated color of the pixel.
Algorithm 4 Main kernel function

1: procedure COMPUTEPixel(surfaceT, newtonT, accStruct, resultBuffer)
2:    Calculate thread index to pixel index mapping
3:    Retrieve starting depth
4:    Retrieve limit depth
5:    if DepthIsValid then
6:        Calculate ray origin and direction
7:        Call TraceRay procedure
8:        Store TraceRay result in resultBuffer
9:    else
10:       Store colorBuffer pixel in resultBuffer
11:    end if
12: end procedure

It calls the sphere tracing function and stores the result. The result contains the distance in which the surface was found and the normal vector of the surface. If the surface was found, it computes Blinn-Phong shading model and returns the calculated color. In the case where the surface was not found within the boundaries established, it loads from the rasterization color buffer the color corresponding to the pixel and returns it to be stored by the `ComputePixel` kernel to the final pixel buffer.

Algorithm 5 TraceRay procedure

1: procedure TraceRay(ray, surfaceT, newtonT, accStruct, startDepth, limitDepth)
2:    Call SphereTrace procedure
3:    if FoundSurface then
4:        Compute Blinn-Phong shading
5:        return PixelColor
6:    else
7:        return ColorBufferPixelColor
8:    end if
9: end procedure

The `SphereTrace` function (algorithm 6), that is called by the previous procedure, receives all the input variables that was given to the `TraceRay` procedure and outputs a pair: the distance where the surface was found and the normal vector that was calculated.

It starts by initializing two new variables: `currentPoint`, having as the initial value the ray origin point and `distanceMarched` that starts at zero. Then the control enters in a loop until the current depth is lower than the limit depth. In that loop, the selected acceleration structure is queried with the current point position, storing the neighbor atoms in a list. The solvent excluded surface function is called to return the current distance to the surface and the normal vector. If the distance is lower than the threshold `surfaceT`, then it is returned the result of calling the SES function. If the surface was not found, the point moves forward in its direction by the distance that was given by SES function and the variable `distanceMarched` is incremented by that same value. Lastly, if the loop ends without finding the surface, the color of the corresponding pixel on the spheres and billboards color buffer is used to define the final image. Note that in the pseudo-code, the distance is used with negative sign, since that is the way that SES function is introduced in the authors work [24].

The procedure that computes the solvent excluded implicit function (algorithm 7) has as input the point to be tested (`point`), the neighbors atoms (`neighbors`), the number of neighbor atoms (`neigh-
Algorithm 6 SphereTrace procedure

1: procedure SphereTrace(ray, surfaceT, newtonT, accStruct, startDepth, limitDepth)
2:     currentPoint ← ray.origin
3:     distanceMarched ← 0
4:     while currentDepth ≤ limitDepth do
5:         neighbors ← Query(accStruct, currentPoint)
6:         SESreturn ← SES(currentPoint, neighbors, Length(neighbors), surfaceT, newtonT)
7:         if SESreturn.distance ≤ surfaceT then
8:             return SESreturn
9:         end if
10:        currentPoint ← currentPoint − dot(SESreturn.distance, ray.direction)
11:        distanceMarched ← distanceMarched − SESreturn.distance
12:     end while
13:     return (−1.0, (0.0, 0.0, 0.0)) ▷ returning invalid result
14: end procedure

borsCount) and the thresholds surfaceT and newtonT. It outputs the negative of the distance from the point to the surface and the normal vector of the surface.

As previously explored in the state of the art section, the solvent excluded implicit function depends on the number of neighbor atoms of the point being tested. In case there is no atom which verifies that the distance between the atom center position and the point is lower than the atom radius plus two times the solvent radius, it always returns $-2R$ as the distance to the surface, where $S$ is the solvent radius and a tridimensional zero vector as normal, since it will not be used, because the surface when this happens was not found yet.

When there is one atom within the neighborhood, it is returned the function $VF$ that takes as input the atom position, the atom radius and the point. The function follows the equation 4.17, where $aPosition$ and $aRadius$ are correspondingly the atom position and the atom radius and the point is the point being tested. The normal vector returned is calculated by the resulting vector from the subtraction of the point and the atom position.

$$VF(aPosition, aRadius, point) = aRadius - \|point - aPosition\| \quad (4.17)$$

$$VG(aPosition, aRadius, point) = VF(aPosition, aRadius, point) + S \quad (4.18)$$

$$VGrad(aPosition, point) = \frac{aPosition - point}{\|aPosition - point\|} \quad (4.19)$$

If there are exactly two atoms in the neighborhood, two points are calculated by projecting the current point from each center of the two atoms to the influence radius of the other as illustrated in figure 4.6. The center positions of the first and second atoms are represented in the figure as $c1$ and $c2$ correspondingly, while the influence radius are named as $g1$ and $g2$. The point $p$ and its projections $p1$ and $p2$ in the figure are also used in the lines 2 and 3 of the algorithm 8.

The variable $B_{i,j}$ contains a boolean value, that is true if the distance between the projection points are inside the influence radius of the complementary atom. When this happens, it is used Newton's
Algorithm 7 SES procedure

1: procedure SES(point, neighbors, neighborsCount, surfaceT, newtonT)
2:     if neighborsCount = 0 then
3:         return $[-2.0 \times S, [0.0, 0.0, 0.0]]$
4:     else if neighborsCount = 1 then
5:         return $[VF(neighbors[0], point), point − neighbors[0].position]$
6:     else if neighborsCount = 2 then
7:         return SES2(neighbors, point, newtonT)
8:     else
9:         return SES3(neighbors, point, newtonT)
10:     end if
11: end procedure

iterative formula from lines 7 to 18, to find the intersection between the two spheres created on the center of each atom and its influence radius. There are two stopping conditions: the maximum allowed number of iterations numTries and the stops variable which check if the intersection point is closer than the threshold newtonT. Inside the loop, in line 11, a matrix is created with the format represented in equation 4.20, where $c0$ and $c1$ corresponds to the position of the first and the second atom. The matrix is then inverted and it is used to translate the point nPoint.

Afterwards, it is checked if the distance between the translated point and each atoms influence radius is lower than the Newton threshold, in the case when this happens, the variable stops is triggered and will stop the loop. In the end of the iterations, it is returned the distance between the intersection point and the point that is being tested as the distance to the surface and the normal vector is calculated by subtracting the intersection point and the current point of the ray marching in line 20. If the boolean variable $B_{i,j}$ is false, then there is no intersection between both influence surfaces and it is returned the distance to the closest atom and the vector from the point and the position of the closest atom.

\[
\text{matrix} = \begin{bmatrix}
VGrad(point, c0).x & VGrad(point, c1).x & 0 \\
VGrad(point, c0).y & VGrad(point, c1).y & 0 \\
VGrad(point, c0).z & VGrad(point, c1).z & 0
\end{bmatrix}
\]  \hspace{1cm} (4.20)

Figure 4.6: Parulek et al. 2012[24]: Projection from the point to the influence radius

If there are three or more atoms in the neighborhood, then it is applied a spherical-triangle function (algorithm 9). The process is similar to the previous case, but it is needed to test every combinations of three. To accomplish this, we use the function GetCombinationsCount that returns the number of combinations of three that will be iterated. In order to minimize the number of combinations, the neigh-
Algorithm 8 SES2 procedure

1: procedure SES2(atoms, point, newtonT)
2:     \( p_1 \leftarrow \text{point} - \text{VG}(\text{atoms}[0], \text{point}) \times \text{VGrad}(\text{point}, \text{atoms}[0]) \)
3:     \( p_2 \leftarrow \text{point} - \text{VG}(\text{atoms}[1], \text{point}) \times \text{VGrad}(\text{point}, \text{atoms}[1]) \)
4:     \( B_{i,j} \leftarrow \text{Length}(p_2 - \text{atoms}[0].\text{position}) < \text{atoms}[0].\text{radius} + S \)
5:     \( B_{i,j} \leftarrow B_{i,j} \& \& \text{Length}(p_1 - \text{atoms}[1].\text{position}) < \text{atoms}[1].\text{radius} + S \)
6:     if \( B_{i,j} \) then
7:         \( nPoint \leftarrow \text{point} \)
8:         \( \text{stops} \leftarrow \text{false} \)
9:         \( \text{numTries} \leftarrow 0 \)
10:        while \( \text{numTries} < 8 \& \& \! \text{stops} \) do
11:            \( \text{matrix} \leftarrow \text{CreateMatrix}(\text{nPoint}, \text{atoms}) \)
12:            \( \text{matrix} \leftarrow \text{Invert}(\text{matrix}) \)
13:            \( nPoint \leftarrow \text{Cross}(\text{matrix}, \text{VG}(\text{cAtoms}[0]), \text{VG}(\text{cAtoms}[1]), 0) + \text{nPoint} \)
14:            for \( i \in \{1, \ldots, 2\} \) do
15:                \( \text{stops} \leftarrow \text{stops} \& \& \text{VG}(\text{atoms}[i], \text{nPoint}) \geq -\text{newtonT} + \text{atoms}[i].\text{radius} + S \)
16:                \( \text{stops} \leftarrow \text{stops} \& \& \text{VG}(\text{atoms}[i], \text{nPoint}) \leq \text{newtonT} + \text{atoms}[i].\text{radius} + S \)
17:            end for
18:        end while
19:        \( \text{numTriest} + + \)
20:    end while
21:    return \( [\text{Length}(\text{point} - \text{nPoint}) - S, \text{nPoint} - \text{point}] \)
22: else
23:    \( f_0 \leftarrow \text{VF}(\text{atoms}[0].\text{position}, \text{atoms}[0].\text{radius}, \text{point}) \)
24:    \( f_1 \leftarrow \text{VF}(\text{atoms}[1].\text{position}, \text{atoms}[1].\text{radius}, \text{point}) \)
25:    if \( f_0 > f_1 \) then
26:        return \( [f_0, \text{point} - \text{atoms}[0].\text{position}] \)
27:    else
28:        return \( [f_1, \text{point} - \text{atoms}[1].\text{position}] \)
29:    end if
30: end procedure

bors atoms list is sorted by the distance from the point to each atom radius and the algorithm uses only maximum of \( K_{\text{max}} \) atoms, in our work we use the value 10 atoms.

The loop in line 7 makes the iteration between combinations, and the loop in line 12 iterates inside the chosen combination. The variables \( a, b \) and \( c \) pick a number of the combination to test as in table 4.1. For each iteration inside a combination, the \( \text{SES2} \) function is called, storing its result in the variable \( \text{ret} \). Then the previous \( B_{i,j} \) is extended to take one more atom, \( B_{i,j,k} \). It is checked if the intersection point found is inside the influence radius \((2S + \text{atomRadius})\) of the atoms selected by \( a \) and \( b \) and the shorter radius \((S + \text{atomRadius})\) of atom selected as third atom by \( c \). When one of those conditions does not verify or there was no intersection point in the \( \text{SES2} \) function, then it skips for the next iteration. When the condition is true, Newton iterative formula is again used to find a close point to the intersection made by the three selected atoms. Again, the process is similar, but extended to support three atoms. The modifications are the matrix which is now represented as equation 4.21 and the \( \text{newtonT} \) threshold is now compared with one more atom.

In the end, it is returned the lowest distance to surface calculated and its corresponding normal vector.
Table 4.1: Iteration of a combination

\[
\begin{array}{ccc}
\text{m=0} & 0 & 1 & 2 \\
\text{m=1} & 0 & 2 & 1 \\
\text{m=2} & 1 & 2 & 0 \\
\end{array}
\]

\[\text{matrix} = \begin{bmatrix}
V\text{Grad}(\text{point}, c_0.x) & V\text{Grad}(\text{point}, c_1.x) & V\text{Grad}(\text{point}, c_2.x) \\
V\text{Grad}(\text{point}, c_0.y) & V\text{Grad}(\text{point}, c_1.y) & V\text{Grad}(\text{point}, c_2.y) \\
V\text{Grad}(\text{point}, c_0.z) & V\text{Grad}(\text{point}, c_1.z) & V\text{Grad}(\text{point}, c_2.z)
\end{bmatrix}\] (4.21)
### Algorithm 9 SES3 procedure

```plaintext
1: procedure SES3(atoms, point, newtonT)
2:     combs ← GetCombinationsCount(neighborsCount)
3:     Sort(neighbors)
4:     cAtoms ← GetCloserNeighbors()
5:     SES2ret ← [−∞, [0,0,0,0,0]]
6:     for l ∈ {1,...,combs} do
7:         for i ∈ {1,...,3} do
8:             cAtoms[i] = GetCombination(l)[i]
9:         end for
10:     SESret ← −8
11:     for m ∈ {1,...,3} do
12:         a ← m == 2?1 : 0
13:         b ← m == 0?1 : 2
14:         c ← m == 2 − m
15:         ret ← SES2([cAtoms[a], cAtoms[b]], point, newtonT)
16:         if ret found intersection then
17:             Bi,j,k ← Length(point − cAtoms[a].position) ≤ cAtoms[a].radius + 2.0 × S
18:             Bi,j,k ← Bi,j,k&Length(point − cAtoms[b].position) ≤ cAtoms[b].radius + 2.0 × S
19:             Bi,j,k ← Bi,j,k&Length(ret.intersPoint − cAtoms[c].position) ≤ S + cAtoms[c].radius
20:         else
21:             Bi,j,k ← false
22:         end if
23:         if ret ≥ SESret then
24:             SESret ← ret
25:         end if
26:     end for
27:     if Bi,j,k then
28:         nPoint ← point
29:         numTries ← 0
30:         stops ← false
31:         while numTries < 8 && !stops do
32:             matrix ← CreateMatrix(nPoint, cAtoms)
33:             matrix ← Invert(matrix)
34:             nPoint ← Cross(matrix, VG(cAtoms[0]), VG(cAtoms[1]), VG(cAtoms[2])) + nPoint
35:             stops ← false
36:             for n ∈ {n,...,3} do
37:                 stops ← stops&VG(cAtoms[n].nPoint) ≥ −newtonT + cAtoms[n].radius + S
38:             end for
39:             numTriest++
40:         end while
41:         return [Length(point − nPoint) − S, nPoint − point]
42:     else
43:         return SESret
44:     end if
45: end procedure
```

#### 4.4 Protein Database parser

Proteins are often stored and textually represented in protein database files (extension .pdb) which contains atoms position and type among other molecular data. “The Protein Data Bank (PDB) format provides a standard representation for macromolecular structure data derived from X-ray diffraction and NMR(nuclear magnetic resonance) studies” [28]. This file contains much information about the molecule,
which is divided into sections. The title section contains, among others, the name of the molecule, data about the source, keywords and remark comments. In the primary structure section it can be found sequences of residues of the molecule. Heterogen section incorporates the description of non-standard residues. The sheets and helices of the molecules are in the secondary structure section.

There are some other sections but the most important to this work is the coordinate section, more specifically, the "ATOM" records. Lines that start with "ATOM" keyword have several fields: Atom serial number, atom name, alternate location indicator, residue name, chain identifier, residue sequence number, code for insertion of residues, the orthogonal coordinates \((x, y, z)\), occupancy, temperature factor, element symbol and the charge of the atom. The huge amount of data is important to be analyzed by specialists of the molecular field, but to produce the desired visualization, the work only considered atom serial number, coordinates \(x\), \(y\) and \(z\) and the atom element. The atom serial number is used to identify atoms that are selected in the region of interest file, where there is a list of identifiers of atoms. The coordinates are used to position atoms in the visualization world space and atom element is used to detect the atom radius, by looking up in an additional file which contains the relation between element and their size.

4.4.1 Sphere tracing range optimization

In order to optimize the heaviest processing algorithm in the solution, sphere tracing, rasterization technique was used to determine the range limits where the sphere tracing is actually required.

Taking into account that the surface can only exist in a spatial zone in which \(\|p - c\| \leq 2R + r\), (where \(c\) is the atom position, \(p\) is the point being tested by the sphere tracing algorithm, \(r\) is the atom radius and \(R\) is the solvent radius) atoms contained in the regions of interest are drawn using rasterization, but instead of using the atoms real radius, it is used \(R_{\text{new}} = r + 2R\), where \(R_{\text{new}}\) is the new radius and the other symbols are in the same terminology as described before, in order to correspond to the desired influence surface. After the drawing, the depth buffer will contain the depth values of this simulated surface. This data will be used to set the initial depth for each pixel where the sphere tracing algorithm should start.

To avoid searching the solvent excluded surface where there is an overlapping to the regions of the molecule not selected as regions of interest, it is also used an additional z-buffer that results from drawing those spheres and billboards, which is used to determine the maximum depth where the sphere tracing should search for each pixel.

Since that these two techniques, ray tracing and rasterization, use different application programming interfaces (API), the first approach was to read the two depth buffers generated by OpenGL, which are located in the GPU, to the CPU, using \(gl\text{ReadPixels}\) OpenGL function, and then copying from CPU code again to GPU, but using CUDA technology, namely \(cuda\text{Malloc}\) and \(cuda\text{Memcpy}\) functions, so it can be used by the CUDA kernel that executes the sphere tracing algorithm. However, those interactions between GPU and CPU are expensive which translated into about eight milliseconds wasted for every frame. To avoid this problem, the CUDA-OpenGL interoperability functions were used.
In the beginning of the execution, the CUDA-OpenGL resources are initialized, using `cudaGraphicsGLRegisterImage` CUDA function, being them three read-only textures and one pixel buffer object. The three read-only textures are composed by two depth textures and one color texture. The two depth textures are the depth buffer that contains the depth values of the closest fragments to the camera when drawing the influence surface, and the same for the drawing of the spheres and billboards of the regions of least interest. The color texture has the color buffer result of drawing the spheres and billboards just mentioned. The manipulable resource is a pixel buffer object (PBO) that is used to store the results from the sphere tracing kernel.

As shown previously, in algorithm 2, the draw event sequence is the drawing of the rasterizer that will fill the textures by using framebuffer objects, that same texture correspond to the read-only textures that will be used in CUDA code. Afterwards, the interoperability image resources are mapped to be ready for use in the kernel, this is done by using the procedures `cudaGraphicsMapResources`, `cudaGraphicsResourceGetMappedPointer` and `cudaGraphicsSubResourceGetMappedArray` in order to obtain a `cudaArray_t` that is passed to the sphere tracing kernel. At this moment, the resources are ready to be used by CUDA code.

Inside the main kernel in algorithm 4, the depth textures are used to determine the starting and limiting depth of the sphere tracing algorithm. The start depth is the depth stored in the influence surface depth texture, while the limiting depth is the depth stored in the spheres and billboards depth texture. If the limiting depth is lower than the initial depth, means that the ROI surface is hidden due to the existance of a closer region of least interest. In this last case and in the case where the sphere tracing tried to find the surface but reached the limit depth without tracking the surface, then the color texture is used to store the result in the pixel buffer object from the drawing of spheres and billboards. If the surface is found, the resulting color is also stored in the PBO.

In the end of the ray tracing, the resources are unmapped to allow the rasterization in the next draw event to access the textures and then the pixel buffer object has its content copied to a texture and the texture is rendered, having the combination of both ray tracing and rasterization.

### 4.5 Prototype evolution

In order to start getting more familiar with molecular rendering context and problems involved, we started creating a visualization prototype. The first prototype was developed in C++ and used OpenGL to create the graphical window.

The first prototype implementation read a protein database file, collecting the data about atom center position. The scene was also composed by a virtual camera that was placed in a position calculated by analyzing the molecule atoms distribution in space, and its orientation was set to the center of the molecule. In this prototype version, we used the same atom radius and color for every atom. To detect the implicit surface, we had a naive marching point algorithm, which started tracing rays from the camera position in the direction of each pixel. This point iterated through the ray, incrementing a fixed step size each time the previous did not intersect the surface. This process was repeated for a fixed number
of tries. The intersection to the implicit surface was calculated by using metaball function described in 2.4 in section 2.1, which consists in the summation of a polynomial function for each atom. If the result exceeded a defined threshold, then the surface was detected, the color was assigned to the corresponding pixel and the next ray was processed.

The resulting visualization showed a very homogeneous molecule, with no difference in color nor atom sizes. To improve this issue, depending on the atom type, different Van der Waals radius is assigned to the atom. To this purpose, we had created a document with radius values for each available atom type. To solve the issue of having the for every atom, we have created a color vector and for every different atom type a color from this vector is assigned. To improve the visualization realism, we implemented Blinn-Phong illumination model. One light was added to the scene and instead of only color, each atom, at this point, had a material assigned, which contained values for ambient, diffuse and specular component, shine, transmittance and refraction index. To apply the model, it was necessary to calculate the normal vector in the intersection points.

![Figure 4.7: Evolution of the prototype: Initial versions](image)

At this point, the molecule had a much better rendering quality as illustrated in figure 4.7 but the time to generate a single image for short sized molecules was already too slow. Although we expected the major performance gain with parallel GPU-based implementation, we tried a few optimizations related to acceleration structures. The first one was to create an axis aligned bounding box that contained the molecule. This was computed by checking the minimum and maximum value for every atom center and considering its radius. With this AABB, it was possible to avoid performing ray marching for pixel which the ray did not intersect the box, and it was possible to limit the ray marching origin and end, by using the intersection points with the box. This solution was decent for simple a short molecules, but when using larger molecules, the bounding box becomes larger, the intersection happens more often and the marching ray length becomes longer. To improve this issue, we have created bounding spheres for each atom and the ray after detecting an intersection with the general bounding box, it checks the collision with every bounding sphere stored. This will mostly shorten the ray marching size, resulting in fewer implicit surface function calculations and speeding up the rendering. Those changes improved the
frame rate from ten seconds to render a molecule to having close to thirty frames per seconds using the same molecule.

Afterwards, the code was adapted to support CUDA, using the CUDA toolkit 7.5, to execute the sphere marching. At this moment, we already had additional .pdb files that selected the zones of interest. The first implementation in CUDA also used changed from metaball to solvent excluded surface for its implicit surface, that required to fully understand how to implement the theoretical introduction in Parulek and Viola work [24]. Then, it was implemented the first mixing between both images, by using OpenGL glReadPixels function to copy the image result and depth from the rasterization to the sequential CPU code, to be stored in one array that was later transferred to the CUDA code, enabling the mixing as shown in figure 4.8.

![Figure 4.8: Evolution of the prototype: Mixing rasterization and ray tracing](image)

Additionally, more distant atoms not contained in zones of interest were converted to simple billboards and more sophisticated acceleration structures like bounding volume hierarchy, uniform grid and kd-tree. Finally, it was modified the technique used to mix both rendering methods, to avoid transferring data from GPU to CPU and back to GPU, by using OpenGL and CUDA interoperability capabilities which resulted in the final visualization shown in figure 4.9, as described later in section 4.4.1.

The final implementation uses C++, freeglut 2.8.1-1.mp for MSVC [29], GLEW 1.11.0 [30] and CUDA Toolkit 7.5 [31].
Figure 4.9: Evolution of the prototype: Final result
Chapter 5

Experimental tests and results

Our experimental tests were divided into two different methodologies: visual quality tests and performance tests. The visual tests are reported in section 5.1 and are focused in comparing visually the difference between a visualization that uses the same representation for the whole molecule and the result of using regions of interest to differ parts of the molecules. The performance tests in section 5.2 are used to evaluate our solution using concrete measures of performance such as time to render a frame and size of the acceleration structures.

In all the results that are displayed in the next sections, there are some parameters that are common between benchmark scenes which are:

- $ResX = 1280$ pixels
- $ResY = 720$ pixels
- $HorizontalAngle = \frac{\pi}{4}$
- $VerticalAngle = \frac{ResY}{ResX} \times HorizontalAngle$
- $R = 1.4$ Ångström
- $FarPlane = 200.0$ Ångström
- $NearPlane = 0.05$ Ångström
- $DistBillboards = 60.0$ Ångström
- $K_{max} = 10$ atoms
- $SurfaceT_{thinnest} = 0.05$ Ångström
- $SurfaceT_{thickest} = 0.2$ Ångström
- $NewtonT_{thinnest} = 0.005$ Ångström
- $NewtonT_{thickest} = 0.2$ Ångström
ResX and ResY are horizontal and vertical resolution of the view window. In terms of camera parameters, HorizontalAngle and VerticalAngle are the camera opening angles, the FarPlane defines the maximum distance until which the scene objects are rendered, the NearPlane is the minimum distance since which the objects start to be rendered and the DistBillboards is the distance that divides the regions of least interest that are represented with mesh spheres and the atoms that are rendered by using billboards.

In the ray tracing, the $K_{\text{max}}$ is the maximum number of neighbor atoms that are used in the sphere tracing algorithm. The thresholds $\text{SurfaceT}_{\text{thinnest}}$ and $\text{SurfaceT}_{\text{thickest}}$ are the thinnest and thickest threshold values for the surface tracking. Finally, the thresholds $\text{NewtonT}_{\text{thinnest}}$ and $\text{NewtonT}_{\text{thickest}}$ are the thinnest and thickest limits of the threshold values used on the Newton-Raphson method.

Two benchmark scenes were used to test the solution. The benchmark scene 1AON uses the molecule named "Crystral structure of the asymmetric chaperonin complex GroEL-GroES-(ADP)7"[32] with identifier 1AON. The whole molecule protein database file contains 58674 atoms. The second benchmark scene 1IGT contains the "Structure of immunoglobulin"[33] with identifier 1IGT and has 12530 atoms.

In the benchmark scene 1AON, it was used two different positions and orientations of the camera, to analyze several parts of the molecule. The first case will be referred as 1AON-1, where the camera position is $(x, y, z) = (184.090622\text{Å}, -71.322571\text{Å}, -26.695751\text{Å})$ and the camera orientation angles are $\alpha = 4.972095$ radians and $\beta = 0.53315$ radians. The second case will be named as 1AON-2, having the camera position at $(x, y, z) = (149.801498\text{Å}, -11.489871\text{Å}, -26.026388\text{Å})$ and angles $\alpha = -1.341626$ radians and $\beta = 0.2407331$ radians. The $\alpha$ and $\beta$ angles are correspondingly the camera horizontal and vertical rotation angles.

The second benchmark scene 1IGT only has one test, with the camera in position $(x, y, z) = (49.140038\text{Å}, 1.700683\text{Å}, -65.313866\text{Å})$ and view angles $\alpha = -0.6507368$ radians and $\beta = -0.1794062$ radians.
5.1 Visual quality tests

In this section, several images are displayed side by side, comparing representations of molecules that are rendered using solvent-excluded surface for the whole molecule and the visual result of rendering the molecule using zones of interest and distance to either use SES, mesh spheres or billboards representation.

5.1.1 Test 1AON-1

Figure 5.1: Visual comparison between rendering benchmark scene 1AON-1 with SES representation (above) and rendering using our level of detail based on zones of interest (below)
5.1.2 Test 1AON-2

Figure 5.2: Visual comparison between rendering benchmark scene 1AON-2 with SES representation (above) and rendering using our level of detail based on zones of interest (below)
5.1.3 Test 1IGT

Figure 5.3: Visual comparison between rendering benchmark scene 1IGT with SES representation (above) and rendering using our level of detail based on zones of interest (below)

5.2 Performance tests

In order to test the performance of the visualization, the benchmark scenes were tested in different machines. The first machine M6000 is composed by a Intel i7-920 quadcore 2.67 GHz CPU, 6 gigabytes of RAM and a Nvidia Maxwell Quadro 6000 GPU. The machine GTX970 is composed by a Intel i5-4690
quadcore 3.50GHz CPU, 8 gigabytes of RAM and a GPU Nvidia GTX 970 with 4 gigabytes of memory. Lastly, the computer GTX980 Ti has a Intel i7-6700 quadcore 3.40GHz CPU, 16 gigabytes of RAM and a Nvidia GTX980 Ti graphics processor.

5.2.1 Solution performance

In this section, the results of applying our solution of regions of interest to the benchmark scenes are described.

Test 1AON

The molecule with identifier 1AON has a total of 58674 atoms, 898 of which were selected by the regions of interest protein database files. Although we have two different tests to this molecule, the size of the acceleration structures do not vary by the different camera parameters. The following list contains the resulting acceleration structures sizes:

- Uniform grid memory size: 45.4 megabytes
- BVH memory size: 0.07 megabytes
- K-d tree memory size: 0.03 megabytes

In terms of frame rate performance, the results are displayed in the table 5.1 for the test 1AON-1 using the thinnest thresholds and in the table 5.2 using the thickest thresholds.

<table>
<thead>
<tr>
<th></th>
<th>M6000</th>
<th>GTX970</th>
<th>GTX980Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPS</strong></td>
<td><strong>Time(ms)</strong></td>
<td><strong>FPS</strong></td>
<td><strong>Time(ms)</strong></td>
</tr>
<tr>
<td>Uniform Grid</td>
<td>21.7 46</td>
<td>20.4 49</td>
<td>30.5 33</td>
</tr>
<tr>
<td>BVH</td>
<td>15.9 63</td>
<td>13.3 75</td>
<td>21.1 47</td>
</tr>
<tr>
<td>K-d tree</td>
<td>16.4 61</td>
<td>13.7 73</td>
<td>21.7 46</td>
</tr>
</tbody>
</table>

Table 5.2: 1AON-1: Results with the thickest thresholds

<table>
<thead>
<tr>
<th></th>
<th>M6000</th>
<th>GTX970</th>
<th>GTX980Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FPS</strong></td>
<td><strong>Time(ms)</strong></td>
<td><strong>FPS</strong></td>
<td><strong>Time(ms)</strong></td>
</tr>
<tr>
<td>Uniform Grid</td>
<td>29.9 33</td>
<td>32.5 31</td>
<td>48.5 21</td>
</tr>
<tr>
<td>BVH</td>
<td>25.1 40</td>
<td>22.7 44</td>
<td>35.5 28</td>
</tr>
<tr>
<td>K-d tree</td>
<td>25.4 39</td>
<td>23.2 43</td>
<td>36.2 28</td>
</tr>
</tbody>
</table>

The test 1AON-2 have the results shown in the table 5.3 using the finnest thresholds and in the table 5.4 using the coarsest thresholds.
Table 5.3: 1AON-2: Results with the thinnest thresholds

<table>
<thead>
<tr>
<th></th>
<th>M6000</th>
<th>GTX970</th>
<th>GTX980Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS</td>
<td>Time(ms)</td>
<td>FPS</td>
<td>Time(ms)</td>
</tr>
<tr>
<td>Uniform Grid</td>
<td>11.8</td>
<td>85</td>
<td>18.9</td>
</tr>
<tr>
<td>BVH</td>
<td>10.5</td>
<td>95</td>
<td>13.1</td>
</tr>
<tr>
<td>K-d tree</td>
<td>10.6</td>
<td>94</td>
<td>13.5</td>
</tr>
</tbody>
</table>

Table 5.4: 1AON-2: Results with the thickest thresholds

<table>
<thead>
<tr>
<th></th>
<th>M6000</th>
<th>GTX970</th>
<th>GTX980Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS</td>
<td>Time(ms)</td>
<td>FPS</td>
<td>Time(ms)</td>
</tr>
<tr>
<td>Uniform Grid</td>
<td>16.3</td>
<td>61</td>
<td>27.6</td>
</tr>
<tr>
<td>BVH</td>
<td>13.9</td>
<td>72</td>
<td>20.4</td>
</tr>
<tr>
<td>K-d tree</td>
<td>14.3</td>
<td>70</td>
<td>21.3</td>
</tr>
</tbody>
</table>

Test 1IGT

The molecule 1IGT has a total of 12530 atoms, 159 of which are in regions of interest. The following list contains the resulting acceleration structures sizes:

- Uniform grid memory size: 11.69 megabytes
- BVH memory size: 0.01 megabytes
- K-d tree memory size: 0.01 megabytes

The performance outcome is in the table 5.5 using the thinnest thresholds and in the table 5.6 using the thickest thresholds.

Table 5.5: 1IGT: Results with the thinnest thresholds

<table>
<thead>
<tr>
<th></th>
<th>M6000</th>
<th>GTX970</th>
<th>GTX980Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS</td>
<td>Time(ms)</td>
<td>FPS</td>
<td>Time(ms)</td>
</tr>
<tr>
<td>Uniform Grid</td>
<td>18.7</td>
<td>53</td>
<td>16.2</td>
</tr>
<tr>
<td>BVH</td>
<td>14.1</td>
<td>71</td>
<td>11.3</td>
</tr>
<tr>
<td>K-d tree</td>
<td>13.9</td>
<td>72</td>
<td>11.2</td>
</tr>
</tbody>
</table>

Table 5.6: 1IGT: Results with the thickest thresholds

<table>
<thead>
<tr>
<th></th>
<th>M6000</th>
<th>GTX970</th>
<th>GTX980Ti</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPS</td>
<td>Time(ms)</td>
<td>FPS</td>
<td>Time(ms)</td>
</tr>
<tr>
<td>Uniform Grid</td>
<td>27.9</td>
<td>36</td>
<td>27.5</td>
</tr>
<tr>
<td>BVH</td>
<td>21.1</td>
<td>47</td>
<td>18.6</td>
</tr>
<tr>
<td>K-d tree</td>
<td>20.6</td>
<td>49</td>
<td>17.8</td>
</tr>
</tbody>
</table>
5.3 Full solvent-excluded surface versus regions of interest

Using the best setup of the ones tested in the performance section 5.2, the machine GTX980Ti and uniform grid as acceleration structure, the following tables describe the performance comparison between rendering the benchmark scenes using solvent-excluded surface for the whole molecule (full SES) and rendering the molecules using regions of interest (ROI).

The table 5.7 shows the comparison of the uniform grid size between full SES and ROI implementations.

<table>
<thead>
<tr>
<th></th>
<th>Full SES</th>
<th>ROI</th>
</tr>
</thead>
<tbody>
<tr>
<td>1AON</td>
<td>429.66 Megabytes</td>
<td>45.44 Megabytes</td>
</tr>
<tr>
<td>1IGT</td>
<td>226.22 Megabytes</td>
<td>11.69 Megabytes</td>
</tr>
</tbody>
</table>

The table 5.8 shows the comparison of the frame rate between full SES and ROI implementations, using the thinnest thresholds.

<table>
<thead>
<tr>
<th></th>
<th>Full SES</th>
<th>ROI</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPS</td>
<td>Time(ms)</td>
</tr>
<tr>
<td>1AON-1</td>
<td>2.8</td>
<td>357</td>
</tr>
<tr>
<td>1AON-2</td>
<td>3.4</td>
<td>294</td>
</tr>
<tr>
<td>1IGT</td>
<td>3.3</td>
<td>303</td>
</tr>
</tbody>
</table>

The table 5.9 shows the comparison of the frame rate between full SES and ROI implementations, using the thickest thresholds.

<table>
<thead>
<tr>
<th></th>
<th>Full SES</th>
<th>Regions of Interest</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>FPS</td>
<td>Time(ms)</td>
</tr>
<tr>
<td>1AON-1</td>
<td>4.7</td>
<td>213</td>
</tr>
<tr>
<td>1AON-2</td>
<td>5.3</td>
<td>189</td>
</tr>
<tr>
<td>1IGT</td>
<td>6.6</td>
<td>152</td>
</tr>
</tbody>
</table>

5.4 Discussion

The results obtained were within the expectations and the goals of the project. Using the GPU capabilities, the sphere tracing algorithm is processed at the same time for each pixel granting better performance than the sequential implementation. The mixing between the rasterization rendering and the ray tracing rendering was a challenging but successfully solved problem, with all the improvements
done in the project evolution, the drawback of combining both techniques was abolished, keeping the advantages of smaller acceleration structures and the capacity of focusing the user’s attention on the regions of interest.

5.4.1 Visual quality

The visual quality tests in section 5.1, the figures 5.1, 5.2 and 5.3 the benchmark scenes that are rendered using SES for the whole molecule are very homogeneous and confusing. One of the goals of this work was for being able to select zones of interest in order to focus those important parts of the molecule, while leaving the rest of the molecular context. Using the additional files that create those regions of interest, the visual representation change depending on the importance of every atom to the user. Our solution used sphere tracing and SES to render the zones of interest, while the rest of the molecule is rendered with mesh spheres and billboards, depending on the distance of the atoms to the camera. Visually, the fact that the regions of interest are rendered with a more detailed representation and have a color that stands out, grabs the user attention and helps in handling the excess of information.

In the future, questionnaires will be made to gather feedback from the molecular field specialists about our solution.

5.4.2 Performance

Solution performance

Based on the performance results shown in the section 5.2, the overall conclusions are that the performance of the visualization is dependent on the region of interest atoms being rendered as demonstrated by the difference of the performance benchmark scenes in subsection 5.2.1. On other side, the frame rate scales well with the increase number of atoms, since that the great difference in the number of atoms between the benchmark scenes in tests \(1AON\) and \(1IGT\), the frame rate is not drastically affected. The difference in the rendering time between the coarsest thresholds and the thinnest thresholds is in average 22 milliseconds, granting smoother camera movements while the user is exploring the scene.

In terms of machines, the rating performance shows that GTX970 is with lower results, followed by \(M6000\) and having GTX980 Ti has the best among the three machines. GTX980 Ti highlights in the results, having in every case considerably better frames per second and was chosen as the best machine for the last performance test in the section 5.3.

Uniform grid, as expected, is the acceleration structure with better results, since that there is no traversal, only a simple calculation to get the index of the grid in which a point is. It is the one that requires more memory space for a long margin but, even in the case of the larger tested molecule \(1aon\) the memory does not reach 50 megabytes, and the current GPUs have 2 gigabytes or more. The uniform grid was selected for the last performance test in the section 5.3.

In the competition for the second place in the acceleration structures ranking, bounding volume hierarchy and k-d tree have very similar results. The difference between the two acceleration structures lays on the structure of the molecule. If the atoms of the regions of interest are very close together,
k-d tree shows better performance, however if the atoms are in more separate regions, then bounding volume hierarchy has better rating. The reason behind this difference is the high amount of overlapping of bounding volumes in the scenarios with atoms focused in one region.

With the usage of OpenGL-CUDA interoperability capabilities, there was a gain around 12 milliseconds per frame when compared to using OpenGL read pixels option.

**Full solvent-excluded surface versus regions of interest**

The scenarios in section 5.3 shows that displaying the molecule using the full solvent-excluded surface, the memory requirements increased substantially when compared with using regions of interest. When all the molecule is selected as solvent-excluded surface, the size of the acceleration structures increases, reducing the performance of the traversal and increasing the number of combinations computed in the sphere tracing algorithm, affecting negatively the overall performance. The solution without regions of interest also do not optimize the sphere tracing with the techniques described in section 4.4.1.
Chapter 6

Conclusions

After analyzing the existing solutions to render molecules, we came to the conclusion that there are two main types of rendering techniques used: polygonization combined with rasterization and ray tracing. Polygonization has performance as the main advantage, while ray tracing provides an higher level of realism. There are few solutions that use level of detail, and the ones that use it, use the level of detail mostly in the distance from the camera to the molecule. Most solutions do not use molecular features to affect their visualization.

In an attempt to provide a different solution that avoids some of the disadvantages enumerated before, we present a novel solution proposal. The proposed visualization uses molecular features to change the level of detail of molecular zones. It is used high quality ray tracing to render atoms near the zones of interest, leaving the rest of the atoms to be rendered with low quality polygonization and rasterization. The combination takes advantages of ray tracing and polygonization, while causing an interesting side effect. This consequence is the fact that the difference between zones with high quality and zones with lower will cause a blur effect and therefore forcing the user to focus his vision in zones with higher detail. Those zones are the ones selected by the molecular feature, such as cavities, that are the ones that are more interesting to analyze, the remaining atoms with lower detail will only give some molecular context.

Additionally, the usage of both ray tracing and rasterization is exploited to get some optimizations. The depth of the fragments generated through rasterization are used in the sphere tracing algorithm to limit the depth in which the marching points track the surface, improving the overall frame rate.

6.1 Future Work

It is planned to make a questionnaire to molecular field specialists in order to get a more concrete feedback on the visual quality of our level of detail based on regions of interest solution.

In the future, it would be interesting to adapt the visualization to support molecular dynamics. This would imply that all the acceleration structures had a very short construction time, so that it could be rebuilt every frame and updated to the CUDA code. The support for molecular dynamics would be
interesting to analyze a full sequence of molecular features like the binding action between a protein and a ligand, to evaluate how the atoms change during this phenomenon.

A second idea is to create a user interface, that would certainly complement the visualization and making it more useful to the users. Features like selecting atoms to get information about them or to convert the neighbor region to be a region of interest would make the visualization more complete and flexible as illustrated in figure 6.1.

Figure 6.1: Example of a future user interface


