University of California
Los Angeles

Parallel, Data-Driven Simulation and Visualization of the Heart

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Eduardo Ribeiro Poyart

2016

© Copyright by
Eduardo Ribeiro Poyart
2016

Abstract of the Dissertation

Parallel, Data-Driven Simulation and Visualization of the Heart

by

Eduardo Ribeiro Poyart

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2016

Professor Demetri Terzopoulos, Chair

This thesis focuses on the Lagrangian approach to fluid simulation, its parallelization, and its application in the medical imaging and simulation contexts. The fundamentals of Smoothed Particle Hydrodynamics (SPH) are analyzed, and common implementation techniques are shown. We describe our SPH implementation and show a novel approach to particle-mesh collision resolution. We also focus on the data pre-processing step, so that captured time-varying volumetric heart scans can be directly used to drive the simulation, rather than hand-crafted models. Our new mesh interpolation approach generates intermediate steps to allow stable, higher resolution simulations. Multithreading and GPU parallelism are analyzed, and a multi-CPU approach is shown, which allows the simulation to be highly scalable. We present a visualization framework, VSim, and its application to heart simulations, especially for training, education and collaboration purposes. Additionally, we show the relation between Lagrangian fluids and our previously published work on particle-based hair simulation, and we explore ultrasound volume registration methods with the purpose of enabling blood flow simulations in large volumes.

The dissertation of Eduardo Ribeiro Poyart is approved.

Joseph M. Teran

Glenn D. Reinman

Jason Cong

Demetri Terzopoulos, Committee Chair

University of California, Los Angeles

2016

To my wife, Alejandra, and to my mother, father and brothers, for their love, encouragement and endless optimism.

And to my sons Diego and Mateo. Que vocês tenham um futuro brilhante e feliz. – *May you have a bright and happy future ahead.*

Table of Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank my advisor, Professor Demetri Terzopoulos, for his encouragement, extensive knowledge, valuable suggestions and ideas, for the many meetings during the preparation of this thesis, and for believing in it. I would also like to thank Professor Petros Faloutsos, my initial advisor, for counseling, guidance, and for the papers we published together.

Many thanks also to my committee members: Professor Joseph Teran, who patiently taught me the fundamentals of fluid simulation, Professor Glenn Reinman, for accepting me as Teaching Assistant in his class and for a lot of guidance, and Professor Jason Cong, for his valuable comments.

During the course of my studies, I received funding from the National Endowment for the Humanities to develop VSim, and I am very grateful for their support. I would like to express my greatest appreciation for Dr. Lisa M. Snyder and Dr. Scott Friedman for leading that project and for bringing me in to work on it. I also thank Dr. Eric Savitsky and the whole team at SonoSim, Inc. My work with them taught me a lot about medical imaging, was quite inspirational, and also helped fund my studies.

My managers at Google, Matt Hancher and Rebecca Moore, were very encouraging and flexible throughout the final stages of development of my dissertation. I am very grateful for the culture at Google of valuing academic achievement. My colleague Hector Gonzalez graciously endured my defense rehearsal and provided valuable comments. I would like to extend a big thank you to all of them.

I would like to thank my colleagues in the UCLA Computer Graphics & Vision Laboratory, for many discussions and exchange of ideas, and for inordinate amounts of inspiration: Brian Allen, Jingyi (Franklin) Fang, Sharath Gopal, Billy Hewlett, Wenjia Huang, Mubbasir Kapadia, Gergely Klar, Masaki Nakada, Gabriele Nataneli, Kresimir Petrinec, Garett Ridge, Weiguang (Justin) Si, Konstantinos Sideras, Shawn Singh, Matthew Wang, and Lap-Fai (Craig) Yu.

Finally, I thank my wife Alejandra, my sons Diego and Mateo, my mother Neide, my father Eduardo, and my brothers André and Leonardo, for their constant encouragement and unconditional love. This thesis would not have been possible without their support.

Vita

1994	B.S. (Computer Engineering), Pontifícia Universidade Católica, Rio de Janeiro, Brazil
1994–2000	Software Engineer, Rio Datacentro, Pontifícia Universidade Católica, Rio de Janeiro, Brazil. Introduced and maintained Internet services for students and faculty (e-mail, Usenet News, World Wide Web). Developed a web application for student access to academic records.
2000–2002	M.S. (Computer Science), Pontifícia Universidade Católica, Rio de Janeiro, Brazil
2002–2009	Software Engineer, Activision, Inc. Developed the video-games Spiderman 2, Spiderman 3, Spiderman: Web of Shadows.
2011	Software Engineering Intern, NVIDIA, Inc.
2012	Software Engineering Intern, Google, Inc.
2012–2013	Software Engineer, Sonosim, Inc. Developed medical ultrasound training application, and ultrasound volume registration and visualization tools.
2013–present	Software Engineer, Google, Inc. Developed algorithms for Google Maps Engine and Google Earth Engine.

Publications

"Real-Time Hair Simulation with Segment-Based Head Collision." Eduardo Poyart and Petros Faloutsos. *In Motion in Games. Lecture Notes in Computer Science, Volume 6459,* pp. 386–397 (Ronan Boulic, Yiorgos Chrysanthou, and Taku Komura, editors). Springer Berlin / Heidelberg, 2010.

"VSim: Real-time Visualization of 3D Digital Humanities Content for Education and Collaboration." Eduardo Poyart, Lisa Snyder, Scott Friedman, and Petros Faloutsos. *The 12th International Symposium on Virtual Reality, Archaeology and Cultural Heritage* (Prato, Italy, October 18–21, 2011) pp. 1–7.

CHAPTER 1

Introduction

Fluid simulation remains a challenging area in computer graphics. It is a hard problem to model and visualize the complexity that arises from the behavior of real-life fluids. A significant amount of recent work has been done on both the Eulerian and Lagrangian approaches to fluid simulation. This thesis focuses on parallel fluid simulation in the Lagrangian approach, specifically using the technique known as Smoothed Particle Hydrodynamics (SPH).

In the medical imaging and simulation context, the simulation of blood flow and its interaction with surrounding tissue has many applications. One such application is the training of medical students in ultrasonograpy. Fluid simulations can be used to display doppler effects, and they enable medical students to be trained in the acquisition of ultrasound images and the analysis of such effects. Another application is the study of the interaction of blood with structures such as heart valves, and also with external structures such as needles that are inserted into veins. In virtual surgery, a Lagrangian blood flow simulation can show the amount of bleeding. It can also visually affect the surgery simulation, since blood can accumulate during surgery and obscure certain areas if not removed. Simulating this phenomenon in a training system increases realism and better prepares the surgeon for dealing with the problem.

As engineering techniques develop and improve CT and other scanners, resolution and accuracy improves, and heart models extracted using such data-capture techniques become more accurate, with all the important internal cardiac structures faithfully represented. This thesis is aligned with this evolution. Simulating fluid flow in precisely extracted heart models and observing its behavior is a valuable proposition.

In computer graphics, we rely on increasingly high-performance hardware to enable us to create interactive systems—systems that support user interaction and that enable simulation and rendering at a high frame rate. Traditionally, integrated circuit manufacturing improvements over time yielded increases in processor speed. Single CPU and linear memory computer architectures had persisted for some time. When advances in CPU clock speed permitted it, offline simulation techniques could be applied to real-time simulation. Clock frequency increases traditionally increased the running speed of programs, without any need to modify those programs.

However, hardware design has now reached physical limits that have reduced the rate of improvement in clock frequencies (Hennessy and Patterson, 2006). Mobile devices have also been contributing to pushing the computing world in the direction of low power and low performance. For example, a doctor can carry a tablet device and consult it while treating patients. This is another motivation to revisit algorithms that were originally designed for high-power desktop computers.

A promising avenue for the improvement of computational performance is parallel processing. Both desktop and mobile devices are increasingly housing multiple CPU cores and massively multithreaded GPU cores, which entails a change in computing architectures and an associated change in algorithm design. Furthermore, cloud computing services allow low-cost access to large number of machines connected through fast networks.

In the aforementioned context, this thesis presents a method to perform parallel Lagrangian fluid simulation in the presence of kinetic geometric models of the heart captured from patients through CT scans, with little to no human intervention in preparing the simulation meshes. The proposed simulation system is highly parallelizable and readily scales with the number of CPUs.

1.1 Contributions

The thesis advances the state of the art in heart simulation and visualization in the following aspects:

1. We present a novel system to interpolate meshes captured from four-dimensional (4D) heart scans. This interpolation works on arbitrary meshes with heterogeneous geometries and does not require prior annotation of correspondence points. Mesh interpolation makes Lagrangian fluid simulation possible by enabling the heart walls to interact with particles with increased precision.

2. We improve on the heart-blood interaction model by presenting an approach to particle collision response that confines particles to the interior of the simulated volume and increases the physical accuracy of the movement of each particle.

3. We present an extensive study on the speed-up obtained by parallelizing the Lagrangian fluid simulation across multiple computers, which has immediate application in the use of cloud computing services. The speed gain, taking into account the effect of network data transfer, is studied and captured in an equation that describes the scalability of the system with the number of machines used.

4. Our simulator produces novel results: Without prior manual steps of heart mesh modeling or editing, we show a full-fledged simulation of the heart beating and the blood flowing, using a kinetic cardiac mesh entirely captured from a 4D CT scan. Blood flow is driven by the captured data. We are also able to isolate and simulate the left ventricle system.

5. VSim, our previously published related work, is a framework for 3D visualization with focus on supporting education, collaboration, and training. We propose that, using VSim, physicians and researchers can annotate heart features and structures and share real-time interactive 3D visualizations with their communities and students, among other uses. Our contributions in this context are as follows:

(a) We propose a novel camera motion control approach that ensures smooth camera motion during the navigation of a 3D scene.

(b) We propose a narrative creation system for end users.

(c) We present an approach to embedding spatially-localized resources into the 3D model.

6. We also present, in the Appendix, our previously published work on hair simulation, which also performs simulation on a Lagrangian framework, and discuss associated parallelization approaches.

7. As minor contributions, we propose (a) the elimination of one of the free parameters of SPH, the rest density, by using the initial distribution of particles to compute it and (b) a method to remove geometric structures caused by noise in the CT scan.

1.2 Overview

The remainder of this thesis is structured as follows:

Chapter 2 presents an analysis of SPH, and we identify an implementation of the state equations and kernels that produce good results. Methods to enforce incompressibility are also studied.

Chapter 3 focuses on the context of heart simulation. We present our method for heart mesh interpolation and particle collision with the mesh.

Chapter 4 discusses parallelism. We present our implementation of GPU-based SPH, and analyze alternatives. We present our experiments with network communication between servers transferring particle data, to enable highly scalable parallel implementations.

Chapter 5 turns to the topic of visualization, presenting VSim, a framework for 3D visualization in education and collaboration.

Chapter 6 presents our conclusions and discusses future work.

Appendix A discusses ultrasound imaging and how fluid simulations can be done in vessels in ultrasound data, presenting our experiments in ultrasound volume registration, which can generate larger areas of captured volumes for simulation purposes.

Appendix B presents a relevant summary of our published work in parallel hair simulation, which shares Lagrangian simulation attributes with the presented heart simulation.

CHAPTER 2

Background and Prior Work

2.1 Fluid Simulation

2.1.1 Eulerian and Lagrangian Views

Fluid simulation can be based on an Eulerian or a Lagrangian approach. In the Eulerian formulation, space is subdivided with a regular or irregular grid. In the Lagrangian formulation, fluid itself is subdivided into particles, each one carrying mass, position and velocity, and representing an amount of fluid.

Both formulations can lead to parallelizable code. In the Eulerian formulation, parallel methods can be used for preconditioning or solving the linear system of equations resulting from the formulation (McAdams et al., 2010). In a purely Lagrangian formulation, there is no global coupling between particles, and parallelization is more natural. Because of the lack of global coupling, however, incompressibility is not trivial to enforce.

2.2 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a Lagrangian formulation for fluid simulation using particles. It was first described for applications in astrophysical simulation (Gingold and Monaghan, 1977; Lucy, 1977). SPH was recently applied to heart simulation by Guo et al. (2013).

2.2.1 Navier-Stokes Formulation

In this section we present a summary of the Müller et al. (2003) formulation of SPH, and our adaptations to it.

We aim to model the Navier-Stokes equation:

$$\rho\left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\cdot\nabla\mathbf{v}\right) = -\nabla p + \rho\mathbf{g} + \mu\nabla^2\mathbf{v}. \tag{2.1}$$

The left-hand side corresponds to mass times acceleration, and the right-hand side corresponds to the sum of forces acting on a particle. In this dissertation, we shall refer to the three terms in the right-hand side as the force $\mathbf{f}^p$ due to pressure, the force $\mathbf{f}^g$ due to gravity, and the force $\mathbf{f}^v$ due to viscosity.

The idea behind SPH is the following: Each particle models a volume of fluid, and it contains a certain mass, density and pressure, along with its position and velocity. In an Eulerian simulation, it is also necessary to model the conservation of mass:

$$\frac{\partial \rho}{\partial t} + \nabla\cdot(\rho\mathbf{v}) = 0. \tag{2.2}$$

Since we assume particles have constant mass and the number of particles is conserved, the mass conservation equation is not necessary in SPH. Furthermore, since particles move with the fluid in the Lagrangian view, the material derivative term on the left-hand side of (2.1) can be replaced with a simple derivative of velocity, yielding

$$\rho\frac{d\mathbf{v}}{dt} = -\nabla p + \rho\mathbf{g} + \mu\nabla^2\mathbf{v}. \tag{2.3}$$

We also assume that we can interpolate scalar quantities (such as density) anywhere in the volume of fluid by taking a weighted sum of contributions from nearby particles. The formula

$$A(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r}-\mathbf{r}_j, h) \tag{2.4}$$

is a general form of this interpolation. It represents a general quantity A being interpolated at a point $\mathbf{r}$, in which the contribution of each particle j, located at $\mathbf{r}_j$, is weighed by a kernel W. The kernel is a function of the distance between the particle and the interpolated point, as well as a constant h defining the extent of the kernel (in our case, exactly the kernel radius). This equation is used to find the density at each particle, then the force due to pressure, and the force due to viscosity:

$$\text{Density: } \rho(\mathbf{r}) = \sum_j m_j W(\mathbf{r}_i - \mathbf{r}_j, h); \tag{2.5a}$$

$$\text{Pressure: } \mathbf{f}_i^p = -\sum_j m_j \frac{p_j}{\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h); \tag{2.5b}$$

$$\text{Viscosity: } \mathbf{f}_i^v = \mu \sum_j m_j \frac{\mathbf{v}_j}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h); \tag{2.5c}$$

where the pressure is computed from density via the ideal gas equation $p = k\rho$, and k is a gas constant.

This leads to asymmetrical pressure and viscosity forces. The force that particle i exerts on particle j is different from the force that j exerts on i. A solution to this problem is to use the arithmetic mean of the pressures of the interacting particles in the pressure case, and the relative velocity between the particles instead of the absolute velocities of particles in the viscosity case.

Also, since the particles always exert a repulsive pressure force against each other, this formulation would be suited for simulating a gas, but not a fluid. Fluids have bonding forces that act between molecules at close distance, which prevents them from dissipating and occupying the enclosing volume like a gas. In standard SPH, this is modeled by changing the state equation to $p = k(\rho - \rho_0)$. This change introduces a rest density, ρ_0, which is the density that the fluid wants to have if undisturbed. If there is a region of density higher than ρ_0, the fluid particles will exert a repulsive pressure on each other, as expected. If there is a region of density lower than ρ_0, particles will exert a negative pressure, that is, an attractive force on nearby particles. The potential profile resembles a Lennard-Jones potential.

The rest density term ρ_0 represents the density that the fluid wants to have if undisturbed. In fact, if a volume of SPH fluid is left in zero gravity, its particles will attract each other when below the kernel distance, and will repel each other if they are too close. After sufficient time has passed, this volume of fluid will evolve to a more or less uniform density equal to ρ_0.

The final formulation for pressure and viscosity is

$$\text{Pressure: } \mathbf{f}_i^p = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h); \tag{2.6a}$$

$$\text{Viscosity: } \mathbf{f}_i^v = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h); \tag{2.6b}$$

where $p = k(\rho - \rho_0)$.

2.2.2 Kernels

The kernels are used to smear out the effect of each particle in space. They are normalized to have an integral of one, for stability purposes. Many different kernels have been proposed in the literature. Here we present the ones that were used in our implementation.

For the density computation, a sixth-degree polynomial kernel is used, as designed by Müller et al. (2003):

$$W_{\text{poly6}}(\mathbf{r}, h) = \begin{cases} \dfrac{315}{64\pi h^9}(h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise.} \end{cases} \tag{2.7}$$

The gradient of the pressure kernel appears in the pressure computation. Gaussian-like kernels have derivatives approaching zero at the center. Because of that, too little pressure is exerted if particles happen to get close together, leading to clumping. Desbrun and Gascuel

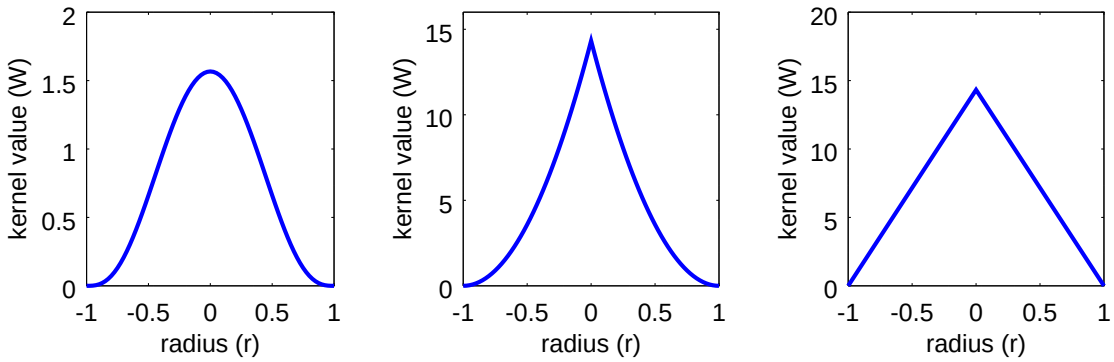

Figure 2.1: *Kernels for density, gradient of pressure, and Laplacian of viscosity.*

(1996) propose the use of a "spiky" kernel for pressure:

$$W_{\text{spiky}}(\mathbf{r}, h) = \begin{cases} \frac{15}{\pi h^6}(h - r)^3 & 0 \leq r \leq h \\ 0 & \text{otherwise;} \end{cases} \tag{2.8}$$

with

$$\nabla W_{\text{spiky}}(\mathbf{r}, h) = \begin{cases} -\frac{45}{\pi r h^6}(h - r)^2(x\hat{\mathbf{i}}, y\hat{\mathbf{j}}, z\hat{\mathbf{k}}) & 0 \leq r \leq h \\ 0 & \text{otherwise.} \end{cases} \tag{2.9}$$

The Laplacian of the viscosity kernel appears in the viscosity computation. Müller et al. (2003) use a kernel such that its Laplacian is as follows:

$$\nabla^2 W_{\text{viscosity}}(\mathbf{r}, h) = \begin{cases} \frac{45}{\pi h^6}(h - r) & 0 \leq r \leq h \\ 0 & \text{otherwise.} \end{cases} \tag{2.10}$$

The above kernels are plotted in Figure 2.1.

2.2.3 Incompressibility

Several methods to enforce to enforce incompressibility in SPH can be found in the literature. The Weakly Compressible SPH (WCSPH) method (Becker and Teschner, 2007) employs stiffer equations of state to achieve less compressibility. In this method, the follow-

ing variation of the gas equation is used:

$$p = B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right), \tag{2.11}$$

with $\gamma = 7$. This equation is one of the forms of the Tait equation (Dymond and Malhotra, 1988), which was originally published by Peter Guthrie Tait in 1888 and relates liquid density to pressure.

When using WCSPH, small time steps must be used because of the stiffness of the equations. As we will see later, we will need to use small time steps for another reason—the movement of the heart mesh and collisions between particles and the mesh. We use WCSPH with good results in our simulation. We modified the Tait equation to improve stability under certain conditions where the mesh movement produced pressures that were too high. The modification was to clamp the pressure computation to a maximum value, as follows:

$$p = \min\left(B\left(\left(\frac{\rho}{\rho_0}\right)^{\gamma} - 1\right), p_{\max}\right), \tag{2.12}$$

where $p_{\max}$ is a high value that we determined by measuring typical pressure values; we used 10^6.

Alternative SPH-based simulation methods are ISPH and PCISPH. In the Incompressible SPH (ISPH) method, Enright et al. (2002) used the pressure projection step from Eulerian fluid dynamics to enforce incompressibility. It allows larger time steps than WCSPH, but at a higher computational cost. In the Predictive-Corrective Incompressible SPH (PCISPH) method, Solenthaler and Pajarola (2009) use a prediction-correction step that propagates density variations across the fluid, in a manner similar to Jacobi iterations for linear systems, up to a point where an error threshold is achieved. The method is more efficient than WCSPH, but at a higher implementation cost.

2.2.4 Recent Advancements in SPH

A large amount of current work is dedicated to high-performance SPH simulation, especially on the GPU. In terms of performance, it is important to notice that the focus of our work is on the analysis of multi-CPU computations and data transfer. As we will see, the scalability on multiple CPUs can extend the performance obtained on a single CPU. Each CPU on such a system can work with a companion GPU to increase the total particle count that can be simulated.

Ihmsen et al. (2014) provide an extensive survey of SPH simulation techniques. This can be complemented by another survey by Weaver and Xiao (2016).

Goswami et al. (2010) achieved interactive simulation of 250k particles entirely on the GPU, although without particular concern for incompressibility. Krog and Elster (2012) simulated a similar number of particles on the GPU. Nie et al. (2015) present a PCISPH system implemented on the GPU, capable of simulating 60k particles at 30 fps, interacting with fixed meshes.

2.3 Heart Simulation

A significant amount of previous work has been done in heart simulation, with varying degrees of automation in the heart model generation.

The idea of immersed boundaries was developed by Peskin to address the interaction with the heart walls in a Navier-Stokes simulation, first by analyzing only the fluid flow around a heart valve in 2D (Peskin, 1972), subsequently extended to the full heart in 2D (Peskin, 1977), then in 3D (Peskin and McQueen, 1989). In the immersed boundary treatment, fluid is simulated in the Eulerian domain. Within a volume of fluids, there are moving boundaries, such as the heart valves. These boundaries are considered massless and made of an elastic material, therefore Hooke's laws are used to simulate their behavior. They are modeled in the Lagrangian domain (with the fluid still remaining in the Eulerian domain), and are represented with points and their displacement from their unstressed location. During the

simulation, the boundaries are replaced with a force field which approximates their real effect on the fluid.

Kovács et al. (2001) presented two approaches that use the immersed boundary method. The flow is solved in the Eulerian domain by a Navier-Stokes solver, and the heart mesh is hand generated, including the valves. Details about the heart model are given in another work by Peskin and McQueen (1996).

2.4 Computed Tomography

One of the methods used to acquire volumetric data from the human body is Computed Tomography (CT). In particular, 4D CT scanning methods (Vedam et al., 2003) provide an end-result dataset consisting of multiple 3D volumes, temporally distributed along a cycle (for example the respiratory cycle or the cardiac cycle). One of the acquisition methods consists of acquiring multiple slices over several cycles, with subsequent binning of the slices into their appropriate phase in the cycle. Among other uses, 4D CT scans have been employed to fight cancer tumors with better accuracy (Keall, 2004).

In general, 4D CT scans have low temporal resolution. Recent methods whose temporal resolution is considered high are on the order of 100ms (Leng et al., 2008). Even with those high temporal resolution methods, the number of samples in a cycle of a slow-beating heart, for example at 60 bpm (one beat per second), would be on the order of 10.

2.5 Mesh Interpolation

As will be discussed in Section 3.4, because of the low temporal resolution of 4D CT scans, a mesh interpolation algorithm is employed to produce intermediate meshes. Here we review previous work on mesh interpolation.

Chen and Medioni (1992), working on the problem of aligning partial range-scans of an object in order to reconstruct the whole object, have devised the Iterative Closest Point

method. In the context of object reconstruction, there exists a rigid transformation T that can be applied to mesh P to bring it into alignment with mesh Q. Their formulation is as follows. First, a set of control points in the surface is defined, and an initial transformation T_0 is defined which brings P into near alignment with Q. Then the following process is repeated iteratively:

1. Apply the current transformation T to the control points in P and their normals.
2. Find the intersection of the line defined by the normal and passing through the point with the surface Q.
3. Compute a tangent plane in Q at the intersection position.
4. By least-squares, find the transformation T that minimizes the sum of squared distances between each control point and its tangent plane in Q.

Rusinkiewicz and Levoy (2001) have performed a comparison of many variants of the original ICP description, and present their own combination of these variants optimized for performance.

Our mesh interpolation approach has similarities to mesh morphing algorithms. A survey of mesh morphing techniques was provided by Lazarus and Verroust (1998). Generally, morphing methods for triangular meshes require user input for the specification of feature pairs (Lee et al., 1999). The problem of finding vertex correspondences and dealing with non-topologically equivalent meshes is hard; for our purposes, as we will see, we do not need to have vertex-to-vertex equivalence and we support meshes with different topologies.

Two methods that work on the same class of morphing problems as our approach—that is, automated solution for the correspondence problems—are the approach by Hong et al. (1988) and the approach by Kanai et al. (1997). The former finds correspondence between facets first, then correspondence between vertices. The latter embeds the meshes into a unit disk on the plane using harmonic maps; then, correspondence can be established and the mesh can be interpolated.

In the related area of morphing hexahedral and tetrahedral meshes, Staten et al. (2012) provide a survey comparing six different morphing methods.

2.5.1 Active Contour Models

Active contour models are an energy minimization approach to the extraction of high-level information from image data. They were initially developed with the use of 2D splines to detect features in 2D images (Kass et al., 1988). A spline, called a *"snake"*, is defined. An energy functional describes internal energies in the snake due to bending, external energies applied by the image, and user-defined energies that allow interactive manipulation. An implicit Euler method is used to solve the internal energies; an explicit Euler iterative method is used to make the snake respond to external forces. The user places the snake close to a desired image feature as an initial step; the energy functional is minimized, and each step of iteration brings the snake closer to image contours.

Active contour models can be extended to the third dimension, and instead of an image, a 3D volume can be used. For example, (McInerney and Terzopoulos, 1997) define a *T-surface*, which is a closed triangular mesh. In this mesh, vertices are treated as masses, which are connected by springs. Internal and external forces are applied as before, including an inflation force that makes the mesh expand and seek features in the image. The vertices are moved in an explicit Euler step. Then, the T-surface is reconstructed with new triangles obtained from the intersection of the previous T-surface with the edges of a simplicial discretization of the volume.

T-surfaces are relevant to our problem of extracting a mesh from a volumetric image and interpolating this mesh. Although we used a method based on marching cubes to extract meshes, they remain an interesting avenue to explore. Active contour models incorporate prior knowledge, particularly our knowledge that a normal heart should be composed of four chambers, so they are applicable for these cases. We also need to support the extraction of meshes from a heart that may have a defect, for example a ventricular septal defect (leakage between the left and right ventricles). T-surfaces remain applicable in this case, since they

adapt to the topology that they find in the image, by means of merging when it intersects other parts of itself.

2.6 Visualization And Collaboration

Two pieces of real-time software for exploring three-dimensional models have previously been developed at UCLA, but both have reached the end of their usefulness. Active development was stopped on the original proprietary software developed for the Urban Simulation Team (*uSim*) and it is only available for Linux systems. Similarly, development was terminated on the freely available software from UCLA's Academic Technology Services (*vrNav*) because of the complexities of improving its interface, adding functionality, and maintaining the requisite software dependencies. Many aspects that influenced the research on VSim were studied during development of uSim (Jepson et al., 1996, 1995; Liggett et al., 1995; Friedman, 1994).

Camera control has been studied with various approaches, including a declarative control language in which camera actions are textually described in a non-interactive way (Christianson et al., 1996), an automated camera planner making real-time decisions based on predefined high-level information provided textually by the user (Bares and Lester, 1997), and camera control by hierarchical finite state machines, which are a combination of low-level "camera modules", controlling geometric placement of the camera, and high-level "idioms", selecting camera modules and, timing between shots (He et al., 1996).

Turner et al. (1991) introduce a dynamic, rather than kinematic, camera control system. We found that, for our purposes, such a controller must be over-damped so as to avoid oscillations around the desired position. The first-person camera smoothing system developed for VSim, described in Section 5.2.2, can be considered similar to an over-damped dynamic system, but it is implemented in a simple and efficient way.

Zeleznik and Forsberg (1999) control the camera using the mouse and a single button. This is achieved through mouse gestures that perform different control functions. Depending

on the way the user initiates the gesture (e.g., vertically vs. horizontally), a different type of control is engaged (e.g., zoom vs. translation on the film plane).

CHAPTER 3

Data-Driven Simulation of Fluid in the Heart

In this chapter, we describe our novel approach for fluid simulation in the heart. In particular, using our method for mesh extraction, noise removal and interpolation, and mesh-particle interaction, we are able to automatically extract a model from a 4D CT scan, make it pump blood, and visualize the results. The model is incomplete in terms of internal structures (septum, valves), but a promising blood pumping simulation is obtained nevertheless. The septum and valves may be added as artificially-inserted planes in the simulation, as we will describe.

3.1 Geometric Data Extraction

An important goal of our work is to use captured data for the geometry of the heart, with the fluid simulation taking part inside of it and interacting with it. That is, instead of aiming to hand-model the heart and obtain a clean, simple, guaranteed watertight mesh, the goal is to use real patient data and, to the extent possible, automate the preparation of this data for the simulation.

For experimentation purposes, we use the sample volume data available in the Osirix web site (Pixmeo SARL, 2015), in particular, the dataset called MAGIX. The MAGIX dataset consists of a 4D cardiac CT scan, with 10 frames from a full heartbeat cycle. Each frame is a 3D volume of the chest region, with $512 \times 512 \times 76$ sample points in a regular grid. The volume is provided in DICOM (Digital Imaging and Communications in Medicine) format (ACR and NEMA, 1985; Spilker, 1989). For many of the operations below, we use algorithms provided by the Visualization Toolkit (VTK) (Kitware, Inc., 2015; Schroeder et al., 2006).

The volumes are converted into a mesh by applying a Marching Cubes algorithm. Marching Cubes (Lorensen and Cline, 1987) is an isosurface extraction algorithm that works by finding triangle edges at locations where the isosurface intersects a cube. We use VTK's marching cubes implementation, vtkMarchingCubes. This implementation uses an acceleration structure (in VTK called a *locator*) to quickly find and merge coincident vertices. Without it, the operation of including a new vertex would require a linear search on all previously included vertices.

The volumetric data contains density values at each sample point. Finding the isosurface value at which to extract the surface means finding a good boundary value between the density of solid heart material (the heart muscle) and liquid (the blood). This step can be performed by a doctor in an interactive system; this will ensure that the isosurface value corresponds to the correct interface.

If the goal is to extract only the left side of the heart, the side that receives oxygenated blood from the lungs and pumps it to the body, the problem becomes easier. We performed experiments with the left side only. We also performed experiments with the extraction of both the left and right sides by using a different value, but the *interventricular septum*, the wall that divides the two ventricles, was not fully isolated in this way due to low contrast. Figure 3.1 shows a CT slice; highligted in yellow is the isosurface value that defines the left circulatory system.

A property of the isosurface extracted from the CT volume is that it has no boundary edges in it, except at the faces of the enclosing volume. Boundary edges are defined as edges that are part of only one triangle. Figure 3.2 shows the boundary edges of the first frame of the cardiac data; they indeed all lie at the faces of the enclosing volume. Therefore, if a scanned volume fully encloses the heart, and as long as there is enough contrast between the heart walls and its interior, the resulting meshes will have an interior heart volume that is watertight. Boundary edges will be on arteries and veins, at the points where they exit the capture volume. The method of mesh extraction itself does not create holes, non-manifold areas or stitching problems through which particles can escape. (We define "bad stitching"

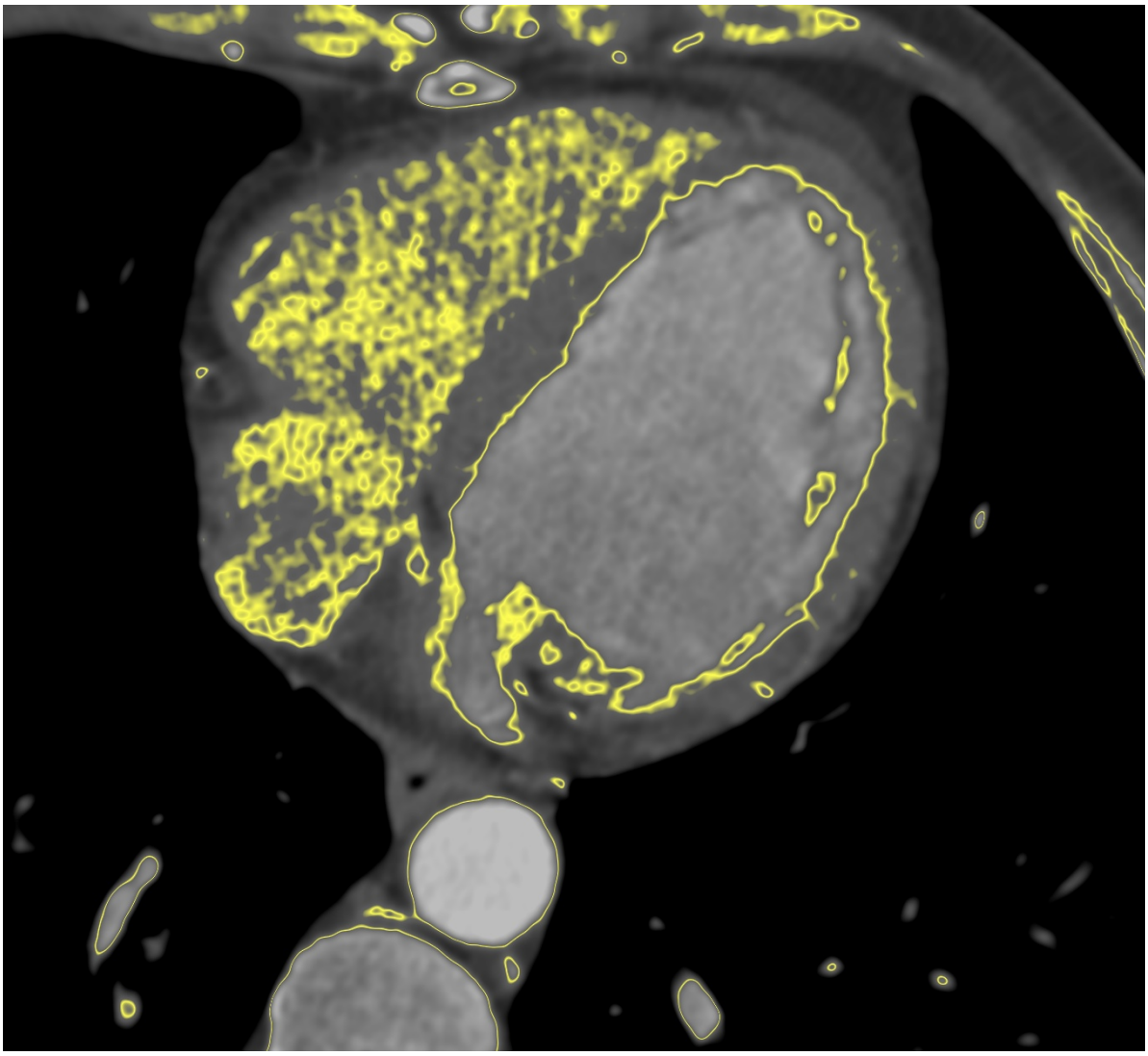

Figure 3.1: *Isosurface extraction. The CT slice is seen from the top; the left side of the heart corresponds to the right ventricle, and the right side to the left ventricle. The value in yellow represents the interface between the inner heart wall and the blood in the left ventricle.*

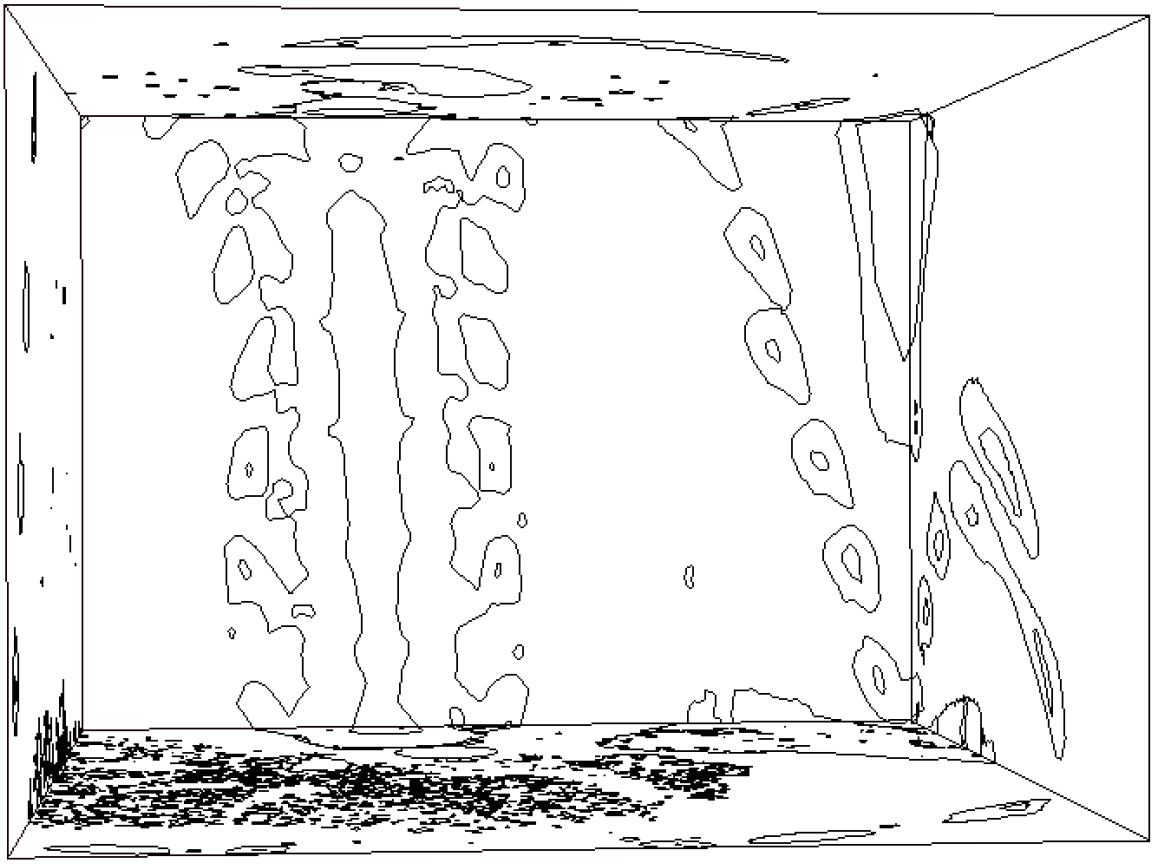

Figure 3.2: *Boundary edges of the extracted mesh. All boundary edges lie at the faces of the enclosing volume.*

as duplicated vertices at the same location, each of them connected to a different part of the mesh, where in reality there should be a single vertex).

Although no holes in the heart isosurface are created due to boundary edges, there can still be holes due to thin structures or low contrast. These holes are not easily detectable through automated means, since they are part of a manifold mesh structure. A thorough visual inspection showed that these holes did not occur in the heart chambers themselves. The heart walls have a large thickness and contrasting density to the surrounding tissue. On the other hand, thin blood vessels showed a few such holes. These holes should not affect the results of the simulation in a significant way.

In one of our experiments we used an isosurface value that extracted all heart chambers, as opposed to the left system only. In that experiment, the interventricular septum was not properly extracted from the volume data. It is possible to simulate the septum, as well as the valves, using planes defined in mesh space. The septum plane should be positioned at the septum location, and during the simulation it prevents particles from crossing in either direction (i.e., behaving exactly like any other polygon in the mesh). The valve planes are unidirectional: they allow particles to pass through in one direction, but not in the other.

3.2 Mesh Simplification

After the isosurface extraction, we performed mesh simplification. VTK's mesh decimation algorithm is applied at this step. Its implementation is similar to the approach by Schroeder et al. (1992). It works as follows:

1. Define *error* as the error introduced if a vertex is removed and the surrounding area is retriangulated (measured as distance from vertex to plane).

2. Each vertex is added to a priority queue based on the computed *error*.

3. The vertex with highest error is removed and the error is recomputed for the surrounding vertices.

4. The process is repeated until there are no more vertices to remove.

To ensure that the decimated mesh maintains good characteristics of not having holes or bad stitching, we disallow vertex splitting and topology changes. These settings can be done through the API in the VTK implementation. We also disallow boundary vertex deletion to preserve good shape at the boundaries, and also because the additional reduction would be small.

Figure 3.3 shows the original mesh extracted by the Marching Cubes algorithm, and the decimated mesh obtained after applying the mesh simplification algorithm.

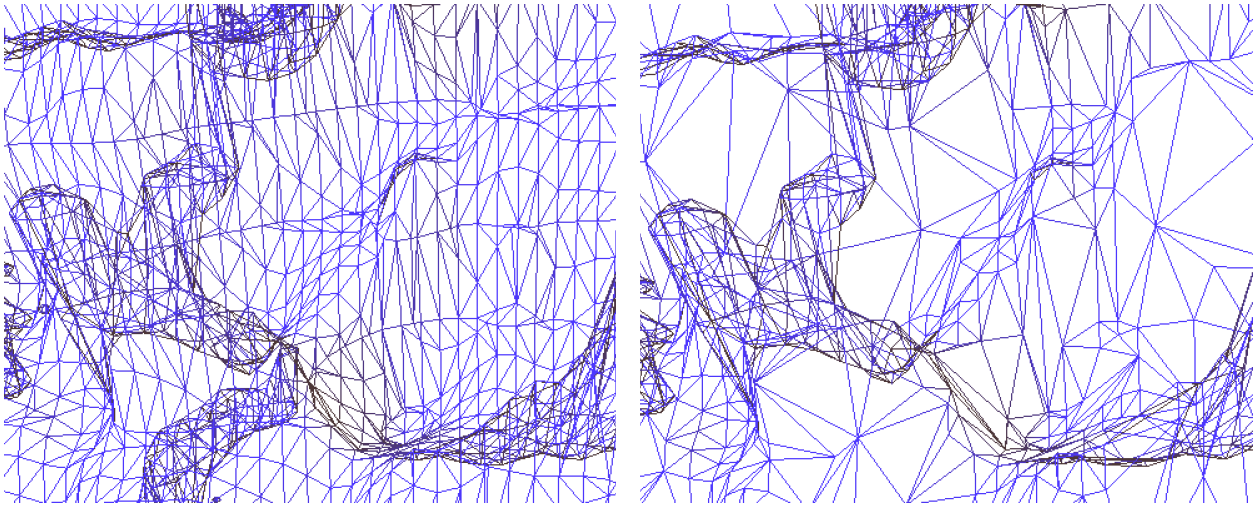

Figure 3.3: *A section of the heart wall is shown as a wireframe. Left: full mesh extracted via Marching Cubes. Right: simplified mesh.*

Figure 3.4 shows the first frame of the complete extracted mesh. Two views of the same mesh are shown. Bones are present in the resulting mesh, exactly as obtained by the isosurface extraction. With our goal of minimizing manual steps, manually removing bones is not desirable. The presence of bones does not incur a significant performance penalty; the spatial subdivision structure used during the simulation provides an $O(\log n)$ search for nearby triangles.

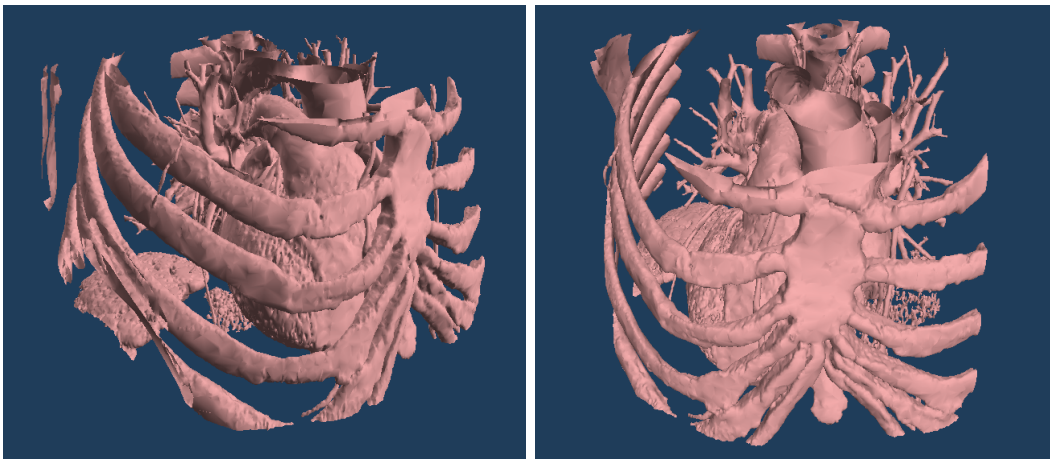

Figure 3.4: *Extracted heart mesh, as rendered by our simulator (two different views). The mesh includes bones, which are not used in the simulation.*

3.3 Noise Removal

The mesh extraction procedure also generated a number of small, isolated structures. They are closed manifold meshes, disconnected from the remaining structures. They are generated by voxel values in the fluid that exceed the isosurface extraction threshold.

We give careful consideration to the cost versus benefit of removing this noise. On the one hand, noise structures appear and disappear in different frames of the simulation, and they may lead to unnatural forces being given to fluid particles and the generation of turbulence. On the other hand, these noise structures tend to statistically concentrate in places where there are actual structures, where the isosurface extraction failed to detect these full structures. This happens at the interventricular septum and at the *chordae tendineae* (fibrous structures inside the ventricles), and it is a desirable effect: simulated fluid will interact with the noise structures in locations where there are *chordae tendineae*, for example, rather than flowing freely and interacting with no structures at all.

A noise removal procedure is described here, although its use can be considered optional. In our mesh interpolation procedure (described in Section 3.4), small noisy structures tended to move in random directions, being attracted by nearby surfaces. Our system ultimately benefitted from noise removal because it reduced this unnatural movement of structures.

We wish to remove small, isolated sub-meshes of the heart mesh. These are almost always closed meshes, except at boundaries of the volume. They may not be convex meshes.

One possible alternative is to compute the volume of each sub-mesh, or to approximate it with the volume of its convex hull. The volume of a convex hull can be computed by finding a point inside the hull (e.g., its barycenter) and computing the volumes of all cones formed with an apex at the center point and with a base at a triangle in the convex hull. The sum of these volumes is the volume inside the convex hull. The volume of a cone is $(1/3)ah$, where a is the area of the triangular base and h is the height from the base to the apex.

We followed another approach, in a novel algorithm described below: box-bounded noise removal. VTK provides us with spatial subdivision structures, *locators*, that accelerate the

search for triangles in a mesh. We first obtain a set S of all triangles that intersect a bounding box centered at one of the vertices of the mesh. The size of this bounding box approximates the size of the structures that we want to remove.

Two conditions must be met so that these triangles are removed:

1. All vertices of all triangles in S must be inside the bounding box (in the general case, a triangle can intersect the bounding box and have vertices outside of it);

2. All triangles adjacent to edges of triangles in the set S must also belong to S.

The first condition catches the case of obviously large structures. The second condition is subtle: It is possible that, in rare cases, an edge of a triangle lies exactly at a face of the bounding box. Through this edge, this triangle may be connected to, and be part of, a large structure outside the bounding box, yet only this triangle happened to fall inside the bounding box. It is not part of a noise structure and should not be deleted. Therefore, by ensuring that all triangles connected to any edges in our set S are not outside of the set S, we eliminate this case.

VTK also provides us with mesh connectivity and navigation information, such that it is efficient to find all vertices of a triangle and all triangles connected to a vertex. Instead of finding triangles adjacent to an edge, we find triangles connected to a vertex, which takes advantage of the efficiency of VTK's data structures and is equivalent in functionality after a simple selection of the desired triangles.

If we find a structure that we need to remove, we mark the triangles in the set S for removal, and mark all vertices in them as visited, and we do not visit them again. In either case, we then move on to the next iteration of the loop by centering the bounding box on the next vertex.

The described box-bounded noise removal approach is more efficient than a precise computation of the volume of the convex hull of small structures. No structure with a volume greater than the volume of the provided bounding box is removed. It also effectively provides

a secondary criterion based on the extent of sub-meshes: Thin sub-meshes of small volume are not removed if they extend beyond the bounding box.

Figure 3.5 shows a view of the inside of the left ventricle, before and after removal of small structures.

3.4 Mesh Interpolation

With the use of SPH for fluid simulation, smaller steps lead to better propagation of pressures, resulting in a fluid that is closer to incompressibility. Each step in SPH is fast and parallelizable, as we will see in Section 4.1. However, as seen in Section 2.4, the temporal resolution of 4D CT scans is low, on the order of 10 frames for a full heartbeat cycle.

The average displacement between consecutive meshes extracted from frames of the MAGIX 4D CT scan was measured using the following method: First, measure the average distance between vertices of mesh n and their closest point in mesh $n+1$. Then, repeat the process using the vertices of mesh $n+1$ and their closest points in mesh n. The mean of the two values is assigned to mesh n as its average displacement. Since many parts of the extracted meshes correspond to non-heart structures such as ribs, only a subset of the meshes was used, cropped to correspond only to one of the heart walls. The result is shown in Figure 3.6.

The average displacement between one CT frame and the next can reach 2.6mm. This displacement is too high, since it is too close to the desired order of magnitude of the SPH kernel radius, which in our experiments is 6mm. In particular, we measured the average distance between particles and their closest neighbors (in a steady-state simulation where particles were at the bottom of a cubic box, using our SPH parameters: $\rho_0 = 0.025$, $k_{gas} = 2.0 \times 10^6$, $h = 6$, $\mu = 8$). We found that this average distance is close to 2.5mm.

To prevent particles from passing through the heart walls, we need smaller displacements. We achieve that by interpolating the mesh using an algorithm created for this purpose, inspired by ICP. ICP finds a transformation that brings a mesh into alignment with another

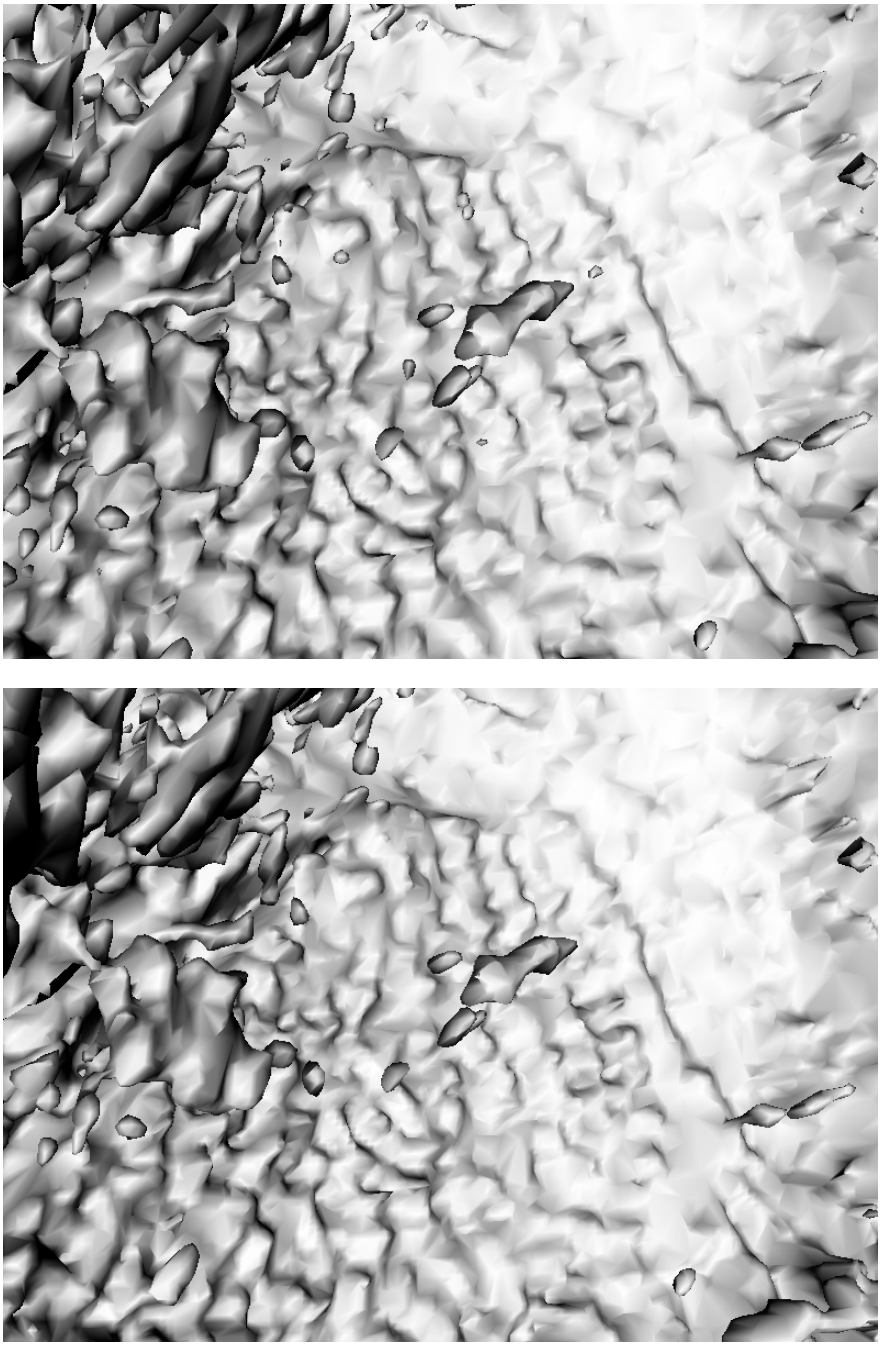

Figure 3.5: *Noise removal. Top: before noise removal; bottom: after noise removal. A few small structures near the center of the image can be seen disappearing. The size used was 1.0mm.*

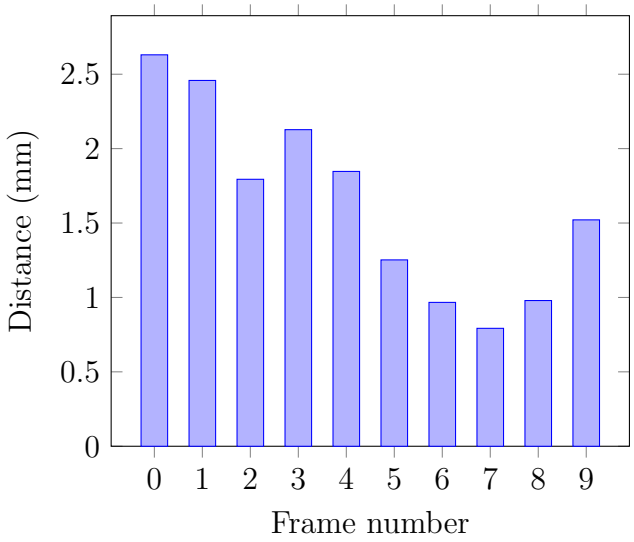

Figure 3.6: *Average distance between a mesh and the next mesh, measured at one of the heart walls.*

mesh; by contrast, our algorithm finds progressive vertex-wise deformations of a mesh to make it approach the shape of another mesh.

3.4.1 Edge-Direction-Preserving Interpolation

Our approach is motivated by the following constraint: We want to deform a mesh to make it globally approach the shape of another mesh, but we want to preserve local features. Many such local features are found in the endocardium; for example, *chordae tendinae*.

An important concept in our approach is bidirectional interpolation. Since we have meshes comprising an entire heart cycle, we always have a previous and a following mesh. We can better preserve shapes if we interpolate a mesh to the mid-point between itself and the previous mesh, and to the mid-point between itself and the next mesh.

A naive mesh interpolation procedure can be described as follows: For each vertex on mesh P, find the closest point in mesh Q and store it as its destination point. Mesh P can then be linearly interpolated to an arbitrary position between the two meshes via a parameter t. This procedure, when applied to heart meshes, has many problems, the two major ones being:

1. Many vertices in mesh P can have the same closest point in mesh Q, which makes many triangles collapse to a single point and generates other long, thin triangles.

2. Different parts of structures may find different closest points in their opposite sides, such that they are stretched in different directions, making them unrealistically change shape.

To overcome this problem, we balance the motion of vertices to their closest points with the preservation of edge directions. Three parameters are used:

1. n, the number of iterations;

2. f_{cp}, the factor between 0 and 1 by which to move vertices towards closest points at each iteration;

3. f_r, the factor between 0 and 1 by which to move edges back to their original directions at each iteration.

Algorithm 1, the interpolation point generator algorithm, is an iterative procedure to successfully move vertices towards their closest points in the target mesh and relax the lengths and orientations of all edges so that they more closely resemble edges in the original mesh. It is composed of two steps: *closestPoint* and *relax*.

In the *closestPoint* step, all vertices in the original mesh are moved towards their closest points in the target mesh by a factor f_{cp}.

In the *relax* step, for each vertex v in the original mesh, all its neighbors w are analyzed. For each neighbor w, the new edge direction is compared with the original edge direction. An interpolated direction, corresponding to a blend between the new and old directions by a factor f_r, is computed. The vertex w is then moved so that the corresponding edge has this interpolated direction.

Note that a vertex may be moved multiple times in the relax step. These two steps are repeated n times. We used the following parameters: $n = 20$, $f_{cp} = 0.7$, $f_r = 0.4$.

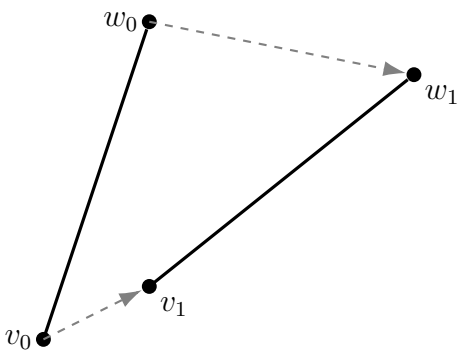

Figure 3.7: *Interpolation point generation: Step 1 – closestPoint function. Points v_0 and w_0 are connected by an edge. From their positions in the original dataset D_0, they are moved to new positions v_1 and w_1 in the working dataset D_1. The amount of displacement is defined by the nearest points in the second mesh (not shown) scaled by a factor f_{cp}.*

Since at each step all vertices are moved to their closest points in the target mesh, there is always a global movement that approaches the shape of the target mesh. The relaxation step attempts to brings local shapes back to their original forms, preventing the mesh from suffering strong local deformations, such as many vertices collapsing to a single point. The algorithm is detailed in Algorithms 1, 2, and 3. Figures 3.7 and 3.8 describe it graphically.

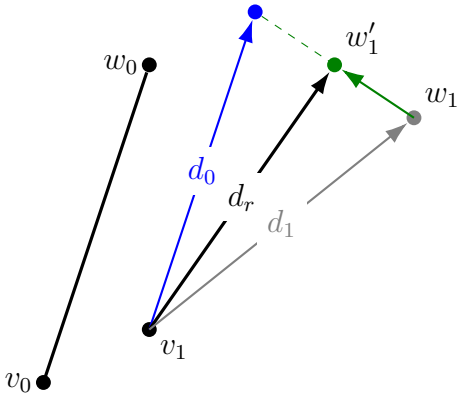

Figure 3.8: *Interpolation point generation: Step 2 – relax function. Point w_1 should be moved so that the edge d_1 more closely approximates its length and direction in the original mesh (d_0). The vector d_0 is parallel to and has the same length as v_0w_0. An interpolation between d_1 and d_0 is computed with a factor f_r, and the direction d_r is found. d_r is added to position v_1 to find the new position w_1'.*

Algorithm 1 Interpolation point generator: main function.

Copy original dataset D_0 to a new working dataset D_1
for n iterations **do**
    closestPoint()
    relax()
end for

Algorithm 2 Interpolation point generator: closestPoint function.

function closestPoint()
    for all vertices v_1 in working dataset D_1 **do**
        Find closest point on target mesh
        Move v_1 toward the closest point by a factor of f_{cp}
    end for
end function

Algorithm 3 Interpolation point generator: relax function.

function relax()
    for all vertices v_1 in working dataset D_1 **do**
        for all edges e between v_1 and a neighbor w_1 **do**
            $v_0, w_0 \leftarrow$ the corresponding vertices in the original dataset D_0
            Compute the edge direction in the original dataset:
                $d_0 \leftarrow w_0 - v_0$
            Compute the current edge direction in the working dataset:
                $d_1 \leftarrow w_1 - v_1$
            Relax the edge direction by a factor of f_r:
                $d_r \leftarrow f_r * d_1 + (1 - f_r) * d_0$
            Apply the relaxed edge direction to the neighboring vertex:
                $w_1' \leftarrow v_1 + d_r$
        end for
    end for
end function

Figure 3.9 shows a slice of the heart wall at two consecutive frames (represented in blue and pink), without interpolation. A yellow sphere of diameter 2.5—our average distance between particles—is shown close to the meshes for size comparison. Figure 3.10 shows the result of our interpolation between those meshes. The blue meshes were interpolated from the original blue mesh in Figure 3.9, up to the halfway point towards the pink mesh. The pink mesh, similarly, was interpolated up to the halfway point towards the blue mesh. A total of 8 interpolation steps were used in this example.

Two sets of destination positions are computed for all vertices of a mesh: the displacement towards the previous mesh (back displacements) and the displacement towards the next mesh (forward displacements). They are stored in custom data fields associated with each vertex, in the VTK mesh data structure (*vtkPolyData*). During the simulation, they can be efficiently accessed. If the current simulation sub-frame is less than the mid-point between two meshes, the forward displacements are used. If it is more than the mid-point, the back displacements of the next mesh are used.

3.4.2 Refinement

To further improve the coherence between meshes at their mid-point, we apply a refinement step after the iterative interpolation point generation step. Recall that the first step finds a displacement for every vertex on mesh A so that it approaches the shape of mesh B on a large scale, while retaining its small-scale details. This displacement moves vertices all the way towards mesh B's non-displaced location. A displacement for mesh B is similarly computed, so that it approaches mesh A's non-displaced shape.

At this stage, if mesh A is interpolated by 0.5 and mesh B is interpolated by -0.5, they should produce two meshes that approximately match. Let us call these meshes A' and B'. The goal of the refinement step is to improve this match. The interpolation point generation step is repeated to find a displacement for mesh A' so that it approached B' and vice versa. The final displacements are the ones obtained at this step.

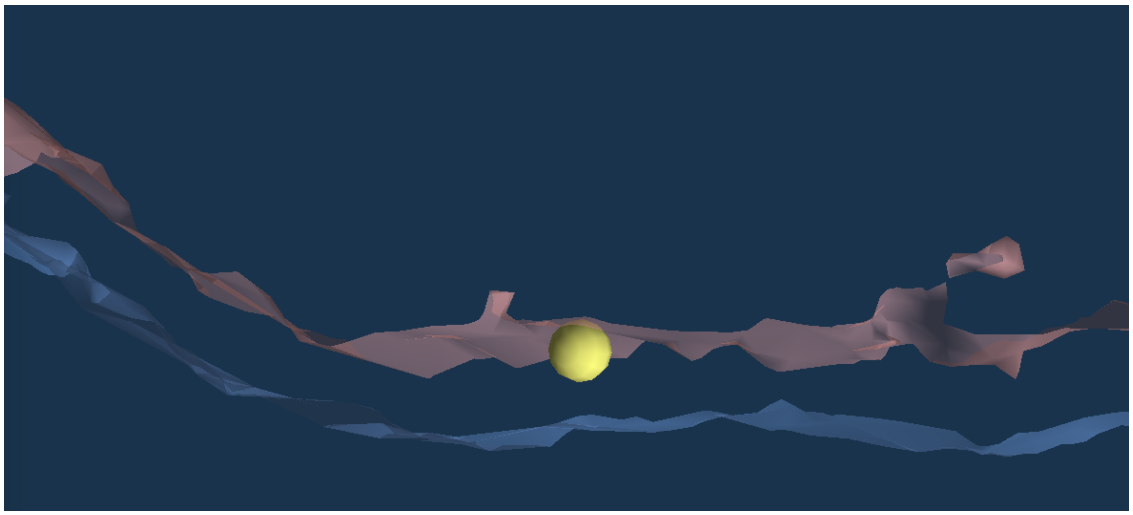

Figure 3.9: *Two consecutive meshes, before interpolation. The scene is zoomed in close to a heart wall. A slice of the heart wall is shown. A sphere of diameter 2.5 is shown for comparison.*

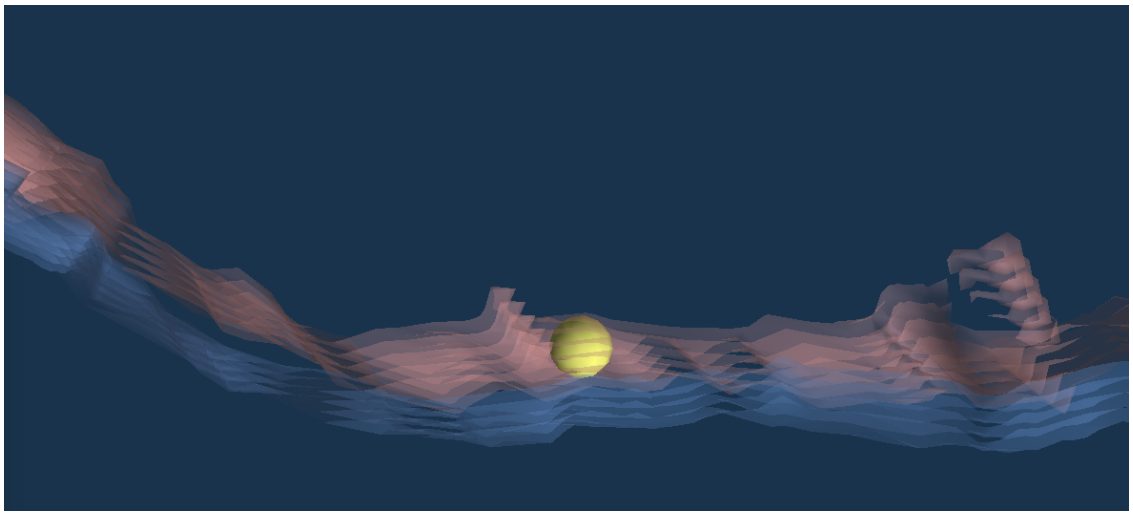

Figure 3.10: *Two consecutive meshes, after interpolation. The blue meshes were interpolated from the original blue mesh in Figure 3.9, and the pink meshes were interpolated from the original pink mesh. Interpolation is done up to the halfway point towards the other mesh.*

Note that after the initial step, the displacement values stored in mesh A take its vertices all the way to mesh B. If we wish to perform a simulation using only this step, we need to interpolate A from 0.0 to 0.5, then B from -0.5 to 0.0 (before repeating with the next mesh). After the refinement step, the stored displacement values take mesh A's vertices to the midpoint between A and B, and the same for mesh B. So, we must now interpolate A from 0.0 to 1.0 such that it reaches the mid point, and similarly B from -1.0 to 0.0. A small modification in the code that performs vertex position update allows that to happen.

Another important observation is that the reference meshes for refinement must be copies of the meshes obtained at the first step of interpolation. The forward interpolation positions for mesh A will be updated to match the mesh B interpolated by -0.5. Then, when processing mesh B, it is necessary to access mesh A interpolated in the forward direction by 0.5. That requires working on a copy, since the forward interpolation positions of mesh A will have been changed at this point.

3.5 Fluid Simulation

We simulate fluids by following the SPH approach of Müller et al. (2003), but using a stiffer state equation to make it perform like WCSPH. We also apply container collision in a modified way.

Experiments were performed first in a simple container, a cube with an open top. In this container, we perform visual validation of the simulation system—we ensure that simulated fluid has the appearance of real fluid. Searching for wall collisions is less costly than on a full heart mesh. The performance was around 10 fps with 10,000 particles, running on a single CPU core. This simplified simulator was then ported to the GPU by using the GPU's global memory to store particle data. We obtained a 2.5× speed-up, increasing the number of particles that can be simulated to 25,000.

Then we move on to the more complex task of simulating in the heart mesh, which was extracted and processed as described in Sections 3.1 to 3.4.

3.5.1 Rest Density

Simulation systems with too many free parameters are hard to configure and tune. It is desirable to infer parameters from other parameters when possible, to remove from the user the burden of experimenting and tweaking them until the results are acceptable.

In our system, we remove the free parameter ρ_0 by automatically detecting the rest density according to the first frame of the simulation. In effect, the user initializes the system by adding particles with a certain distribution. In the first frame, the simulator computes the average ρ assigned to all particles. That value will be used as

$$\rho_0 = \frac{\sum_{i=1}^{n} \rho_i}{n}. \tag{3.1}$$

This can be done since ρ_0 is not used at the density step, only at the pressure step.

The rationale for computing ρ_0 this way is as follows. First, it is common for the user to have a volume where he wants to have an initial number of particles. In our heart simulator, this can be a volume that partially occupies the interior of the heart. The distribution can be either an uniform grid or a random distribution. Second, it is easier and more convenient to define the volume and how many particles should be in it than to pick a value for ρ_0. By initializing the system with this volume of fluid, the value of ρ_0 is effectively defined.

Furthermore, if the user has to initialize a volume of particles and pick his own value for ρ_0, the fluid may end up being subject to unrealistic physical conditions. If the fluid is under excessive pressure, the simulation explodes with particles flying off in all directions. If the fluid is under too little pressure, it condensates in a few areas, forming bubbles of space inside its volume. If, instead, we initialize with the average ρ method, we know that the fluid will be subject to realistic physical conditions.

3.5.2 Collision With the Container

This section describes two approaches for collision handling. The first approach is appropriate for simple, static containers such as our test container. The second approach is

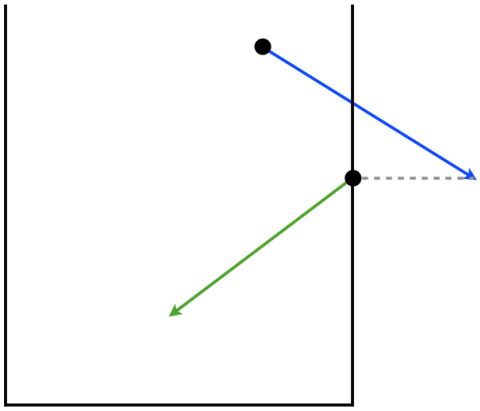

Figure 3.11: *Collision in Muller et al.*

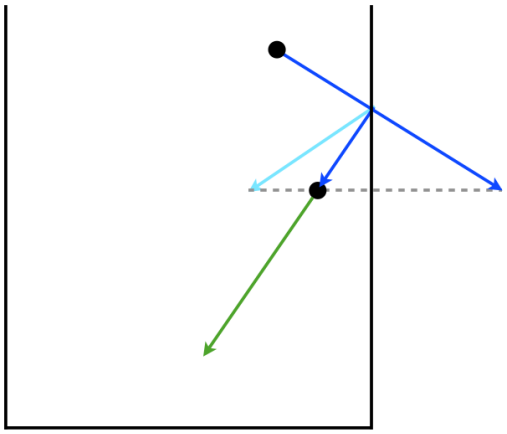

Figure 3.12: *Elastic collision with the container.*

appropriate for moving meshes such as our animated heart mesh. The second approach is more general and it can be used for static containers as well.

3.5.2.1 First Approach: Elastic Collision at the Container Position

For particles that end up outside of the containing volume, rather than simply taking the end position and pushing the particle inside the volume along the normal of the wall, we properly simulate elastic collision with the walls. Particles collide with the container at the exact point of intersection between the particle velocity vector and the container. Their final position is computed by simulating this occurrence as an elastic collision.

Figures 3.11 and 3.12 illustrate the differences between the Muller collision method and the elastic collision methods. The particle travel vector (the vector between the last and current positions) is indicated in blue, and the resulting velocity in green. In our method, we can parametrize the coefficient of restitution of the elastic collision, which causes a reduction in the normal component of the reflected velocity. Although we did not implement a tangential friction component, this is also possible. With this method, particles better conserve their energies, and the extra damping introduced by the simulation system is reduced.

Corner cases can arise when the particle is close to the edge between two faces of the container. To properly resolve them, we iterate again with the remainder of the reflected travel vector, starting from the collision point. If the particle exits the container again, we repeat the procedure and the particle is reflected back inside the container, until it does not exit the container anymore. The new collision method resulted in negligible performance penalty for the algorithm.

Due to pressure from other particles, a layer of particles tend to form along the walls, resulting in a noticeable grouping of particles when seen from certain angles. This is due to the fact that the walls do not exert pressure over the particles. In this sense, the particle layer that forms naturally takes the role of the wall itself and it exerts a higher pressure over the neighboring particles, balancing the system.

As an alternative, the walls could be made to exert pressure on particles, as if they were made of particles themselves; this is usually done by adding ghost particles at points outside of the container that mirror the position of real particles. This works well for simple containers. However, with our goal being to enable simulation inside a complex heart mesh, this is not always possible—it is not straightforward to determine ghost particle positions when the mesh is highly folded and the triangle sizes are the same order of magnitude as the particle spacing. Therefore, we let the naturally occurring layers of particles do its job of balancing the pressure.

3.5.2.2 Second Approach: Collision With Margin

As mentioned, meshes extracted from heart CT scans contain lots of folds and small triangles. Furthermore, the mesh moves from frame to frame, and it is important to avoid particles leaking from the volume as much as possible.

The second approach for particle collision described here is appropriate for use in moving heart meshes. Due to the mesh motion, if a particle is positioned exactly at a container boundary, it is possible that it will be outside of the container at the next step. This can

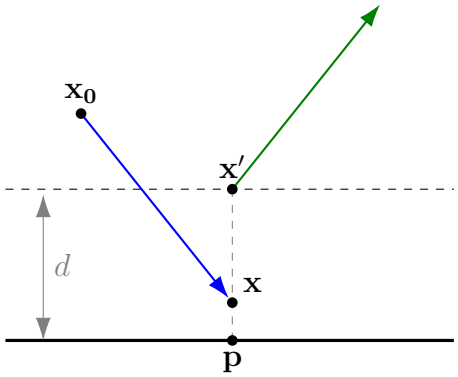

Figure 3.13: *Collision with margin. The mesh is the thick black line. Particles are kept at a distance d from it. When a particle moves from* $\mathbf{x_0}$ *to* $\mathbf{x}$*, it is repositioned at* $\mathbf{x}'$ *and its velocity vector (blue) is recomputed (green).*

happen even with the use of mesh interpolation and small time steps. The algorithm is modified so that particles are made to collide at a distance d from the container.

In practice, a closest-point-in-mesh search is performed. This step is accelerated by the use of a VTK-provided spatial subdivision structure that indexes the mesh, *vtkCellLocator* (in Section 4.1.1, we describe our modifications to this VTK class so that it is made thread-safe). The closest point $\mathbf{p}$ to a particle at position $\mathbf{x}$ can be a vertex, a point in an edge, or a point in a triangle of the mesh. If the distance between $\mathbf{x}$ and $\mathbf{p}$ is less than d, the particle is considered to have collided with the mesh, and the normal of this collision is $\hat{\mathbf{n}} = (\mathbf{p} - \mathbf{x})/(|\mathbf{p} - \mathbf{x}|)$. If this distance is too small for an accurate normal computation, the normal of a triangle containing $\mathbf{p}$ is used.

Once the normal is obtained, the particle position is pushed along the normal to a point $\mathbf{x}'$ so that its distance to the mesh is d. This is once again similar to the Muller approach, with the difference that particles are not positioned at the mesh, but at a distance d from it. This keeps particles from getting too close to the mesh walls, preventing leaks. After the position is corrected, the velocity is updated as before, taking into account the coefficient of restitution. Figure 3.13 shows this approach.

The fact that the particles stay at a distance d from the walls can be viewed as a result of them not being point entities, but something akin to spheres with some radius. Recall that, in SPH, particles are modeled as representing volumes of fluid. These volumes do not

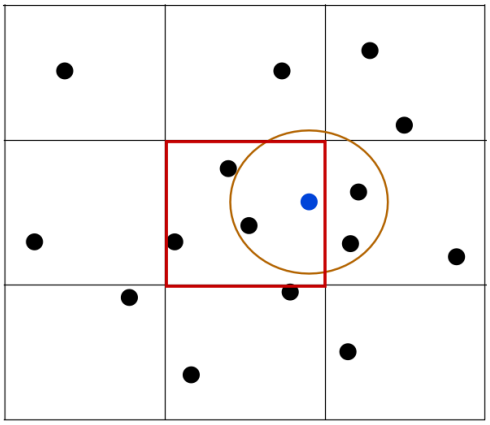

Figure 3.14: *Grid used to accelerate search for nearby particles.*

have any particular shape; in fact, they can be seen as having a fuzzy border due to the interpolation kernels used. However, for rendering, it is common to use spherical particles or some variation of them. The rendering radius can be made to approximately match d and the visual surface of the fluid will closely resemble the heart mesh shape.

3.5.3 Neighborhood Particle Search

If no spatial subdivision data structure is used, all particles must be considered for interaction with all other particles, leading to an algorithm of squared complexity, which is not desirable. In our heart simulation, we use VTK's data structures for particle neighborhood search. However, for our GPU implementation, a GPU-friendly data structure had to be used. We describe it next.

We observe that each particle interacts with other particles within a distance less than h. Thus, we use a grid of voxels of size $2h$ to accelerate the lookup for nearby particles. Each cell holds a list of particles that currently reside in it. As each particle moves, if it changes grid cells, it is removed from the list of particles corresponding to the previous grid cell, and added to the list of the current grid cell. The enumeration of all particles in a cell is trivial with this data structure—it is just a list traversal.

Figure 3.14 illustrates a 2D view of the 3D voxel grid. In this example, we need to compute forces for all the particles in the central cell. Suppose we are computing the forces

for the particle indicated in blue. The brown circle with radius h is the area of actuation of the kernel. For all particles at distances greater than h, the kernel has a value of 0, so those particles need not be considered. If the radius is h and the cell size is $2h$, the only particles that must be considered reside in the $3 \times 3 \times 3$ block of cells around the cell where the blue particle resides. Therefore, this limits the number of cells that must be traversed. The complexity is reduced to $n \times n_c$, where n_c is the maximum number of particles per cell. That is, it is linear in the total number of particles.

We opted not to limit the maximum number of particles per cell. However, limiting this number has another interesting consequence: If done carefully, it helps enforce incompressibility in the volume. Let us say a particle is entering a grid cell, but this cell already is full to capacity. The particle can be made to stay at the edge of the cell from where it is coming. Although this helps with incompressibility, it causes big losses of energy, especially if all particles of the fluid are flowing in an average direction. To mitigate that, the particle may be allowed to keep its velocity value; i.e., it need not be reduced to 0. In the next frame, if other particles in the destination cell move away, the particle that was forbidden to enter may now be allowed in. Clearly, this does not fully solve the energy loss problem, but in reality the primary purpose of this mechanism is not to enforce incompressibility; other methods must be used for that. The cell capacity limitation should be designed to happen rarely.

When designing a SPH algorithm for GPUs with shared memory, the cell capacity limitation also helps with the memory footprint. As we will see in Section 4.2, all particles in 27 adjacent cells must be copied to shared memory, and the amount of shared memory is limited.

3.5.4 Surface Tension

Traditionally, with a smaller number of particles, each particle was viewed as representing a volume of fluid. Surface tension is usually introduced in the algorithm to improve the behavior of the system. With a sufficiently high number of particles, each one represents

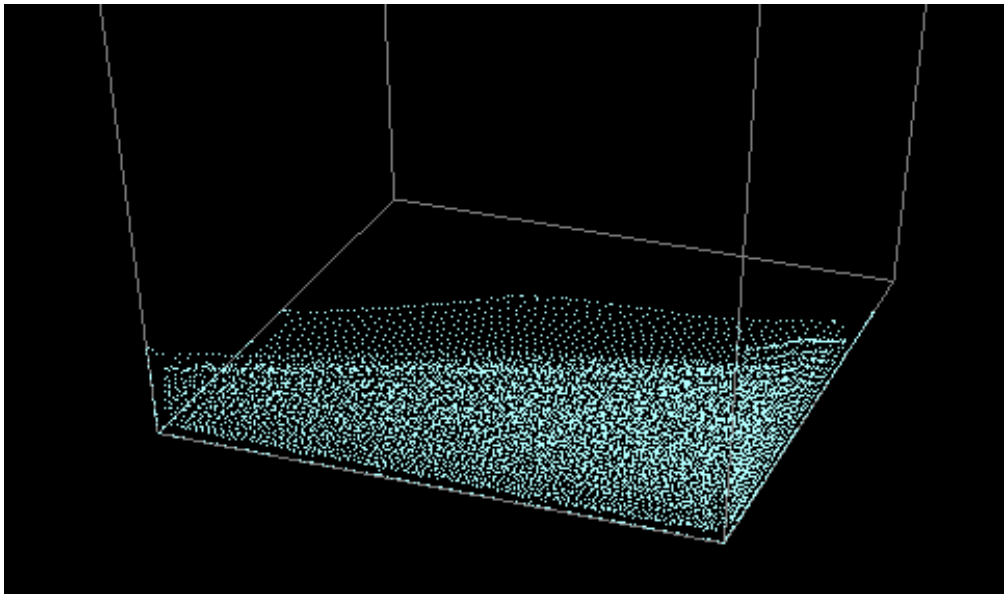

Figure 3.15: *Surface tension.*

a smaller volume of fluid, consequently a smaller number of molecules. Emergent behavior such as surface tension becomes apparent. The effect is described in detail below.

Surface tension is the increased cohesion that happens between molecules on the surface of a fluid. Since surface molecules are not surrounded by other molecules on all sides, the net force pulls these molecules inwards. This, in turn, increases the pressure of the fluid near the surface, which balances the inward-pointing force. The result is that surface molecules are closer together and more strongly bonded than molecules in the bulk of fluid.

Figure 3.15 shows an experiment made with the SPH simulator that was implemented. The container is tilted with its top towards the camera. The top part of the blue particles is a one-particle deep layer at the bottom of the recipient. At the very edge of this layer, it is clear that there is a one-dimensional line of particles closer together—the emerging surface tension of the system. No artificial surface tension term was added, contrary to (Müller et al., 2003).

Interestingly, in recent work, Schechter and Bridson (2012) also noticed emerging surface tension effects in SPH simulations. However, in this case these effects were unwanted. The

intention was that a volume of fluid resting in space in zero gravity did not change shape. So additional steps were taken to counteract the effects of surface tension.

3.5.5 Solid-Fluid Coupling

In our work, the heart walls interact with particles with a one-way effect—the heart mesh drives the particle movement. During the process of capture of the heart model, the heart and fluid interaction happened as it normally does. Any effect that the blood had in the heart walls, such as the effect of pressure and blood movement, became a part of the model itself. At simulation time, the heart drives the blood with one-way interaction and, as an end result, we have a recreation of the full function of the heart, including pumping blood.

A brief discussion on ways to achieve two-way interaction follows. Solid-fluid coupling in SPH can be done by using standard rigid-body simulation methods for the solid objects. For example, a boat floats on an ocean due to a counterbalance of the gravity force actuating on the boat and the water pressure on the boat's hull. If the ocean is simulated using particles, the force and torque exerted by each individual particle can be summed together, resulting in a net force and torque on the boat.

In blood flow simulation, we want to model the forces and torques actuating on various rigid and semi-rigid structures in the body; for example, valves in the heart. The most interesting case are semi-rigid structures. There are three options:

1. Model semi-rigid structures as articulated rigid bodies. This method may be interesting if the structures do not suffer large deformations such as squishing or stretching, but rather small ones such as bending. Care must be taken with the particle collision algorithm in the areas of articulation, so that particles do not get trapped where they should not.

2. Model semi-rigid structures as deformable bodies, with an Eulerian simulation method. The structure is modeled as a volumetric mesh with tetrahedra, and deformable body simulation is performed there. Although Eulerian simulation involves solving linear

systems and it is generally slow, the performance can be kept at a good level if the number of elements is small. Updated sparse Cholesky factors (Hecht et al., 2012) or multigrid methods (McAdams et al., 2010) can be used.

3. Model semi-rigid structures as deformable bodies made with particle systems. This option would keep the modeling consistent between the fluid and the structures. Semi-rigid structures, including structures capable of muscular control, can be modeled by spring/mass systems, with springs added between the particles (Tu and Terzopoulos, 1994). The method is not very expensive in terms of CPU cost.

3.5.6 Simulation Results

We performed experiments both with the left-ventricle only and with a complete heart. The experiments followed the procedure below:

- First, the heart is filled with blood, by using particle emitters that create new particles at every frame. One emitter is positioned at the left atrium in the left circulatory system experiment. In the full heart experiment, two emitters are used: one at the left atrium and one at the right atrium. In this step, a single frame of the moving heart was used, without any movement.

- Then, we started the heart movement: The mesh is successively interpolated towards the next mesh position, then the mesh is replaced by the next mesh at the previous mesh position, and it starts to get interpolated towards the next mesh, and so on.

Movement of blood can be seen as soon as the heart starts pumping. The blood leaves the heart through the aorta, and it can be observed that the rhythm with which the blood travels through the aorta corresponds to the pumping rhythm of the heart. The emitters continually generate particles inside the atria. It is interesting to notice that the simulation can be left running indefinitely and produces a continuous flow of blood.

With the full heart experiment, we noticed that, due to the absence of the interventricular septum (as noted in Section 3.1), blood from the left side of the heart mixed with blood from

the right side. This can be corrected with an artificial septum (also described in Section 3.1). We had good results with the addition of an artificial septum, although this was not desirable due to the manual step that it introduces.

We also noticed, from the observation of rendered simulation videos, that the actuation of the mitral valve (the valve between the left atrium and the left ventricle) was partially captured in the data capture step. In a few of the meshes where it should be closed, we observed an actual closing that prevented blood flow. Even though this did not happen in all the frames where it should, because of a lack of contrast in a few frames, this fact mostly prevented blood from flowing back to the atrium, allowing for a more realistic simulation.

The results were rendered in an offline fashion. During the simulation, particle positions were saved to files; we used the POV-Ray software to render the results using ray-tracing. The POV-Ray primitive of "blobs" was used to represent the fluid. It works by making each particle create a field in space with a radially decaying value in its surroundings. This field is additive among particles. A level set is defined, and rays are traced against this level set.

In Figure 3.16, we see the left ventricle being filled up with blood, and in Figure 3.17 we see one cycle of the heartbeat. In Figure 3.18 we see one cycle of the heartbeat for the full heart experiment. In all these figures, only the blood is rendered (the heart itself is invisible). In the left ventricle, around 8,000 particles were used. The full heart experiment exercised the system at a larger scale and showed its robustness with 25,000 particles.

Mesh interpolation (Section 3.4) and collision with margin (Section 3.5.2) greatly reduce leaked particles. Figure 3.19 shows frames from an experiment identical to the left ventricle experiment shown in Figure 3.17, except that mesh interpolation was disabled for comparison purposes. The benefit of mesh interpolation is very apparent in the frames and in the resulting video. Not only there are fewer particle leaks, but the video shows that the movement is smoother and more natural.

A few particles still escape the interior of the heart volume. This can be seen in Figures 3.17 and 3.18. The remaining leaked particles appear when a mesh is replaced by another mesh with a very different local topology. For example, due to noise or slight bright-

Figure 3.16: *Filling the cardiac left ventricle with blood.*

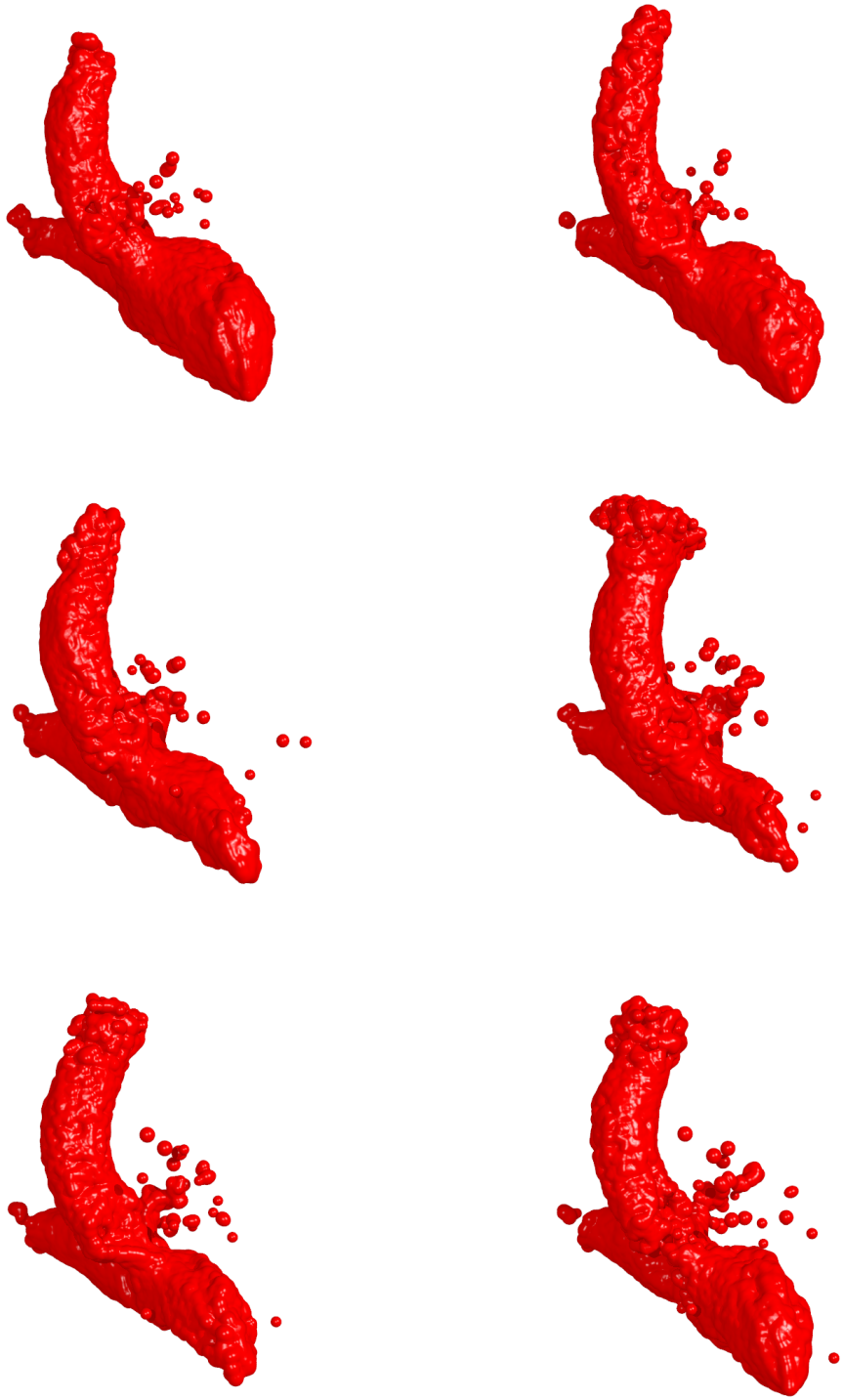

Figure 3.17: *Simulation results for one heartbeat cycle, left ventricle.*

Figure 3.18: *Simulation results for one heartbeat cycle, full heart.*

Figure 3.19: *Simulation results with mesh interpolation disabled, for comparison purposes. The larger number of leaked particles is noticeable.*

ness differences in the CT scan, a mesh where the topology is somewhat smooth in a certain area of the heart wall may be followed by a mesh where there is an indentation in that area. We observed that this actually happens. Due to this fact, particles end up leaking outside of the enclosed volume when meshes are replaced.

After reducing the particle leak to a minimal amount, leaving only the leaked particles due to mesh replacement, we decided to focus on other aspects of the simulation, since the harm caused by these leaked particle was not substantial at this point.

3.6 Rendering

In this section, two rendering approaches are described. First, a fast technique for real-time particle-based fluids is shown. Then, we describe the off-line raytracing approach that we used for our renderings.

3.6.1 Real-Time Approach

A real-time technique for rendering particle-based fluids is described below. This technique is used in many video games and real-time rendering software.

The most costly aspect of rendering free surface fluids is extracting the surface so that it can be shaded and visualized. Instead of explicitly extracting the surface, let us look at the problem in screen space and use the capabilities of the GPU.

In the first step, spheres can be rendered using point-based methods. Only their depth values are rendered to the z-buffer. One sphere is rendered per particle, and they are correctly scaled for perspective. At the end of this step, we have a surface of spheres whose depth values are in the z-buffer.

Then, the z-buffer is blurred. A separable bilateral filter (Pham and Van Vliet, 2005) is used to preserve sharp discontinuities while blurring soft discontinuities. Once the z-buffer is blurred, normals can be extracted using partial differences. With that, we have a smooth field of normals which approximate the normals at the surface of the fluid. Standard Phong

shading can be applied, or a cube map texture lookup can be done to simulate reflections. The surface can be rendered transparently, and an opacity shading can be applied based on the volume thickness at each pixel (this thickness field can be constructed in the sphere rendering pass by using an accumulation buffer as render target). This method was described by van der Laan et al. (2009) and is used in real time rendering of fluids in games.

3.6.2 Ray-Tracing Approach

A ray tracer can be used to extract the surface of the fluid. We used POV-Ray (Persistence of Vision Pty. Ltd., 2004). POV-Ray has been parallelized so that an image can be rendered by a cluster of machines (Fava et al., 1999). However, for animation rendering, each frame as a whole can be rendered by an individual machine or CPU, and intra-frame parallelization is not required.

POV-Ray has support for *blobs* (Blinn, 1982), also called *metaballs* (Nishimura et al., 1985). They are implicit surfaces defined by points and a density function. The points are the locations of fluid particles. The density function defines a scalar field in space. The field is accumulated for all particles, and an isosurface is extracted from it at a threshold t. In POV-Ray, the density function is

$$\text{density} = s\left(1 - \left(\frac{\min(d, r)}{r}\right)^2\right)^2, \tag{3.2}$$

where s is a strength value that we set to 1, d is the distance to the particle, and r is a maximum radius. We obtained good results with $r = 5.0$ and $t = 0.4$.

CHAPTER 4

Parallel Scalability

4.1 CPU Parallelism

Let us review the steps of our SPH simulation:

- *Density step:* For each particle p_i, all its neighboring particles are analyzed, and a density value is stored in p_i based on the application of the density kernel.
- *Force step:* For each particle p_i, once again all its neighboring particles are analyzed, and the forces due to pressure from each neighboring particle are accumulated at p_i. Forces due to viscosity and gravity are also accumulated here.
- *Position step:* For each particle p_i, velocity and position vectors are updated in a forward Euler step. Then, collisions with triangles from the boundary mesh are computed. A final position value is computed and stored in p_i.

It can be clearly seen that, as a Lagrangian particle-based simulation, each of these steps is individually parallelizable. The steps, however, are sequential in nature. Once all density values are computed, the force step can begin; once all forces are computed, the position step can begin.

Each particle can be computed individually, as long as the data structures utilized are thread-safe. Our implementation uses VTK's spatial subdivision structures to perform two tasks: search for nearby particles and search for nearby triangles. For our parallel implementation that used the full heart mesh, we performed experiments with many of these

structures and made modifications to one of VTK's implementations in order to make it thread-safe. These modifications are presented later in this section.

Before delving into the discussion on parallelism and thread-safety, we present a review of spatial subdivision data structures and their implementation on VTK.

Octrees (Jackins and Tanimoto, 1980) are a recursive subdivision of space that can accelerate the search for nearby objects in a 3D model or simulation. In an octree, the space is subdivided into disjoint sets of 8 octants, and each octant is also subdivided, recursively up to a finite number of steps. Each octant is a node in a tree, and its sub-octants are its children. Leafs of the tree contain references to objects in space, for example triangles or points. The number of steps of recursion can be uniform or it can vary for each branch of the tree, particularly to achieve depths in which the number of objects is approximately the same in all leafs.

Octrees can be constructed sequentially or in parallel. Parallel octree construction is particularly useful when dealing with structured data such as voxels. In this context, an octree can be constructed with a bottom-up approach (Lal et al., 1998), in which voxel sets are constructed at each level and then combined and passed to the higher level. Each unit of work is done by a worker, where the number of workers is as many as the number of available CPUs. In VTK, the octree is constructed sequentially.

4.1.1 Single CPU, Multiple Cores

In our parallel implementation, we used as spatial subdivision structures the following VTK classes:

- *vtkOctreePointLocator* to search for closest particles, for the density and force steps;
- a modified *vtkCellLocator* to search for closest triangles in the boundary mesh, for the position step.

At this time, the implementation of VTK's *vtkCellLocator* is not thread-safe when locating the closest cells to a point. Even though it mostly reads data from a read-only tree

constructed at the build phase, it also stores internal work data, such as markers for visited nodes and cell buckets that store neighboring cells. If this data is written from multiple threads, undefined results occur. A semaphore is needed every time a query is issued.

We removed the need for a lock by modifying *vtkCellLocator* so that it is thread-safe. We separated the internal work data structures from the large read-only tree data structure. The work data structures were stored in a new *Context* class. This class can be instantiated once per thread, preventing data races between threads. We structured our parallel loops with an inner loop that works on a *stride* (usually of 100 elements) in a single thread, and an outer loop that is parallelized. With this structure, the *Context* class is created only once per stride, so it does not incur a big cost. The benefit of parallelizing the use of *vtkCellLocator* provided a big gain in performance.

To analyze performance, we divide the algorithm into two steps:

1. *Mesh update step:* vertex update and mesh locator building. Vertex update refers to updating the position of all vertices of the mesh to their interpolated position, according to the current interpolation frame. Mesh locator building refers to building the locator that indexes mesh triangles. These steps are performed once per frame. The vertex update step was parallelized, and the parallel results are shown below.

2. *Simulation step:* particle locator building, density computation, force computation and position update. These steps are iterated multiple times for each frame (i.e., for each *mesh update* step), with the goal of reducing the time step size and allowing pressure propagation. We use 8 iterations. The density, force, and position steps were also parallelized and the results are shown below.

Table 4.1 shows timing measurements for the *mesh update* step; Table 4.2 shows timing measurements for the *simulation* step. For this measurement, the heart mesh was previously filled with 15k particles. The measurements are an average over 20 frames. The mesh locator build step and the particle locator build step are not parallelizable, so their timing is shown only once.

Step	Single threaded	Multithreaded
Vertex update	7.6	5.6
Mesh locator	52.3	

Table 4.1: *Average times in milliseconds for the mesh update step. Timing was done on a hyper-threaded 4-core 2.4 GHz Intel Core i7 CPU, 8 virtual CPUs, 15k particles.*

Step	Single threaded	Multithreaded
Particle locator	2.3	
Density	80.95	17.5
Force	159.8	35.1
Position	411.1	92.9

Table 4.2: *Average times in milliseconds for the simulation step. Timing was done on a hyper-threaded 4-core 2.4 GHz Intel Core i7 CPU, 8 virtual CPUs, 15k particles.*

It can be seen that the most time-consuming step is the position computation. Finding the closest point in the mesh for every particle is a heavy step, but it is necessary for collision handling. Building the mesh locator is also a significant time slice. This time could be saved at the expense of memory by storing all the locators for all the interpolated meshes. The count of these locators would be the number of meshes (in our case, 10) times the number of interpolated frames (in our case 8). However, the time for mesh locator computation does not scale with the number of particles, so the benefit in eliminating it would not be big for a large number of particles.

4.1.2 Multiple CPUs

Within a single CPU, the limitation is the number of cores available. In order to overcome this limitation, we can run the simulation over several machines that communicate over the network. Details about this approach are presented in this section.

The single-CPU SPH simulation proceeds as follows: first, from the positions of the particles, the densities are computed in the *density* step. Then, using positions and densities

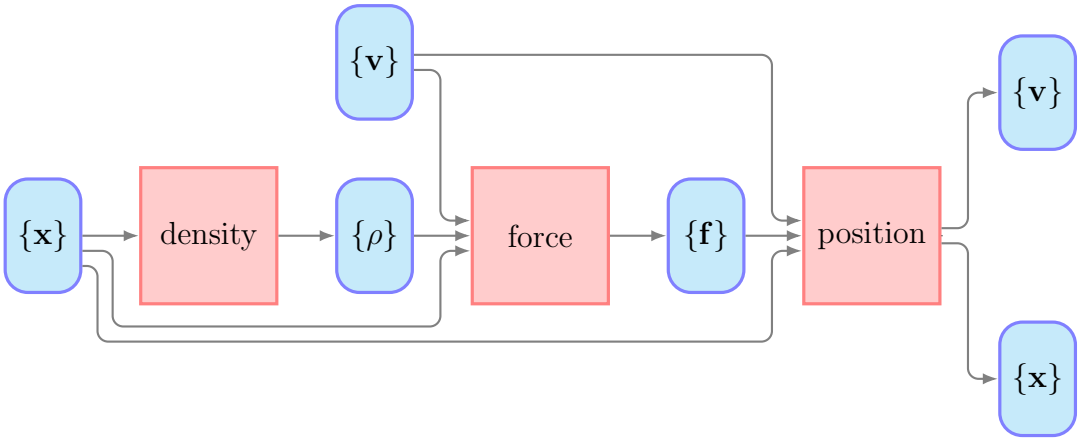

Figure 4.1: *Steps of the SPH computation in a single CPU. Red nodes denote computations; blue nodes denote data.*

as input, forces are computed in the *force* step. Then, using positions, velocities and forces as input, new positions and velocities are computed in the *position* step. Figure 4.1 illustrates the data flow.

We propose a method to parallelize the computations between multiple machines. We define a *master* node and one or more *worker* nodes. The master node drives the process and centralizes the communications. The worker nodes receive data from the master, perform the computations, and send the resulting data back. The master node could be a machine with a small number of CPUs. The worker nodes benefit from having a large number of CPUs; each worker node can internally parallelize the work assigned to it.

Let us assume the workers are initialized with the same parameters as the master (e.g., ρ_0, μ), prior to the start of the simulation. The workers operate on a fraction of the particles, according to their index: if there are n workers and p particles, worker i operates on particles p/i to $((p+1)/i) - 1$.

At the beginning of each frame, the master has all the information about the particles, and it must update the workers as necessary. For the first step, the density computation, the workers need only the positions of the particles. The master sends the positions of all particles to all the workers (p vectors of 3 components). The workers compute the density values, $\{\rho\}$, and send them back to the master. Each worker sends p/n scalars.

The master now has all ρ values, and it can begin the force computation step. This step requires positions, ρ's, and velocities (to compute the forces due to viscosity). The velocities can be sent to the workers simultaneously to their computation of the density step. The densities also must be sent from the master to all workers, since each worker only computed a fraction of the values. Once they have all data, they compute the forces and send back their results (p/n vectors of 3 components).

The position computation can now be performed. The force values are sent to all workers, and they already have positions and velocities. So, they update positions and velocities, and send them back ($2(p/n)$ vectors of 3 components). The simulation frame is now complete. The master can store or render the results, as needed, and begin the next frame. In Figure 4.2, we see the communication and computation timeline described here.

With multi-CPU parallelization, the time needed to perform computations is decreased. On the other hand, time is added for data transfers between nodes. If the added data transfer time t_d is smaller than the time saved by the decrease in computation time, we manage to speed up the simulation.

And what is the decrease in computation time? Each machine works independently on its assigned particles; there is no data dependency during a step. Although there may be a small overhead in starting and ending the loop, let us assume it is small. The computation time with n nodes is t_c/n, where t_c is the computation time for one node.

Thus, the decrease in computation time is the difference between t_c, the sequential computation time, and t_c/n, the computation time of each parallel node. We want this time saving to be greater than the time added by the data transfer step, t_d.

Thus,

$$t_d \;<\; t_c - \frac{t_c}{n} \tag{4.1}$$

is the condition to obtain speed-up; that is, the additional data transfer time should be smaller than the computational time savings.

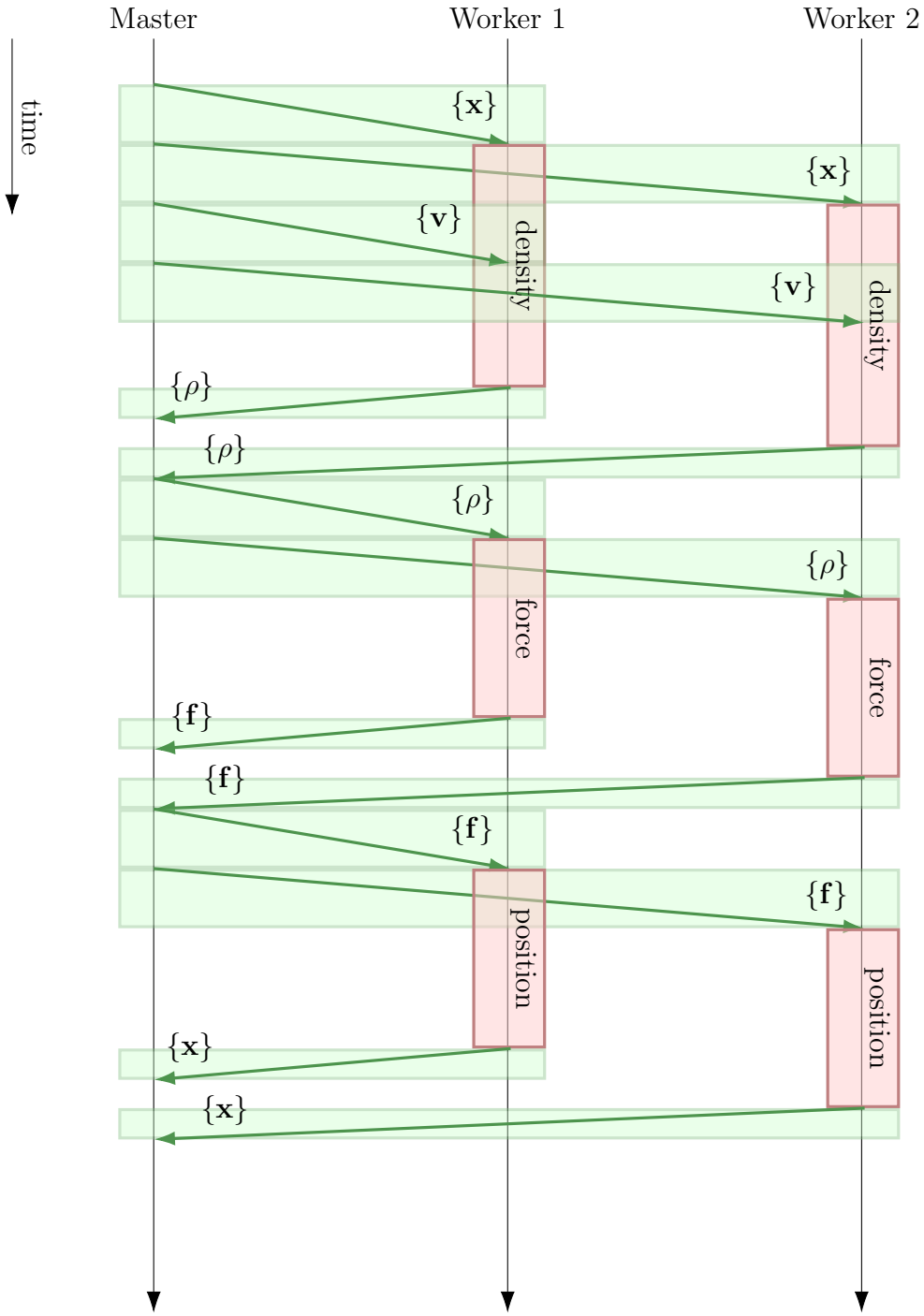

Figure 4.2: *Communication and computation timeline. Time progresses from top to bottom. Green blocks represent data transfer. The network is busy during these periods. Red blocks represent CPU computations; the CPU of a node is busy during these periods.*

If TCP connections are used, then t_d is a function of the number of nodes: the master must send all the position, velocity, density, and force data to all the nodes. We can thus rewrite (4.1) as

$$\begin{aligned} nt'_d &< t_c - \frac{t_c}{n} \qquad \therefore \\ t'_d &< t_c \frac{1 - \frac{1}{n}}{n}. \end{aligned} \tag{4.2}$$

The value of the right-hand side of (4.2) decreases to 0 as n approaches infinity, which shows that there is an upper limit to speed-up. In practice, it is still feasible to use TCP connections and achieve speed-up, if t_d is sufficiently small.

4.1.2.1 Multicast

The communication from master to worker can be done via UDP Multicast, and we can further increase the scalability. Multicast (Deering, 1989) is an extension to the IP protocol to allow one host to send datagrams to many others with a single transmission to a single destination address. Although not widely adopted in multi-networking applications, it is very well supported at the level of individual hosts and IP routers (Ratnasamy et al., 2006), so its application is feasible if all hosts are on the same network. With multicast, the master does not need to replicate the data on the network to send the updated densities, velocities, and forces.

The multicast protocol is not connection-oriented—there is no point-to-point connection between hosts. Therefore, the UDP protocol is supported, but not TCP. Without care, if a packet is lost in our simulation, the computation fails. A mechanism must be used to handle transmission errors and dropped packets. Such mechanisms have been defined (Speakman et al., 2001; Adamson et al., 2009) and have been implemented both as open source and as proprietary software. One such solution works as follows: Sequence numbers are assigned to packets, and if a packet is missed by a host, it can send a request for retransmission. The retransmission is also sent via UDP multicast and is ignored by the other hosts. While

the idea of multicast also encompasses wide area networks, here we assume worker nodes are geographically close—essentially machines in a single data center. This simplifies the issue. Physical transmission lines are short and fast, and it is safe to conclude that the rate of dropped packets will be low during a simulation run. Therefore, retransmission schemes protect the system from failures while having little impact in simulation time.

The timeline with multicast is shown in Figure 4.3. It can be seen that, if the number of nodes is large, the use of multicast will have a big positive effect on the speed-up. If the data transfer time is also large, while unicast will be limited in speed-up due to the multiplying effect of the data transfer to each node, multicast will still provide speed-up.

With multicast, (4.2) becomes

$$t'_d \;<\; t_c \left(1 - \frac{1}{n}\right). \tag{4.3}$$

The right hand side of (4.3) goes to 1 as n goes to infinity. So, for a large number of nodes, it is sufficient that $t'_d < t_c$ in order to achieve speed-up. The number of nodes does not inherently limit the scalability.

4.1.3 Experimental Results

We performed experiments simulating the data transmission and their scalability with the number of particles and number of nodes. They consisted of the transmission of buffers of data of the appropriate sizes. We used 64-bit doubles for all values—densities, pressures and positions. Particle simulation was not added to these experiments; the focus was solely on the timing of the data transfer phases. The process is as follows:

1. The master starts, spawns n threads, and waits for connections.

2. The n workers start, each on a different machine connected to the same network. They connect to the master.

3. The master joins the threads (proceeding only when the last worker connects).

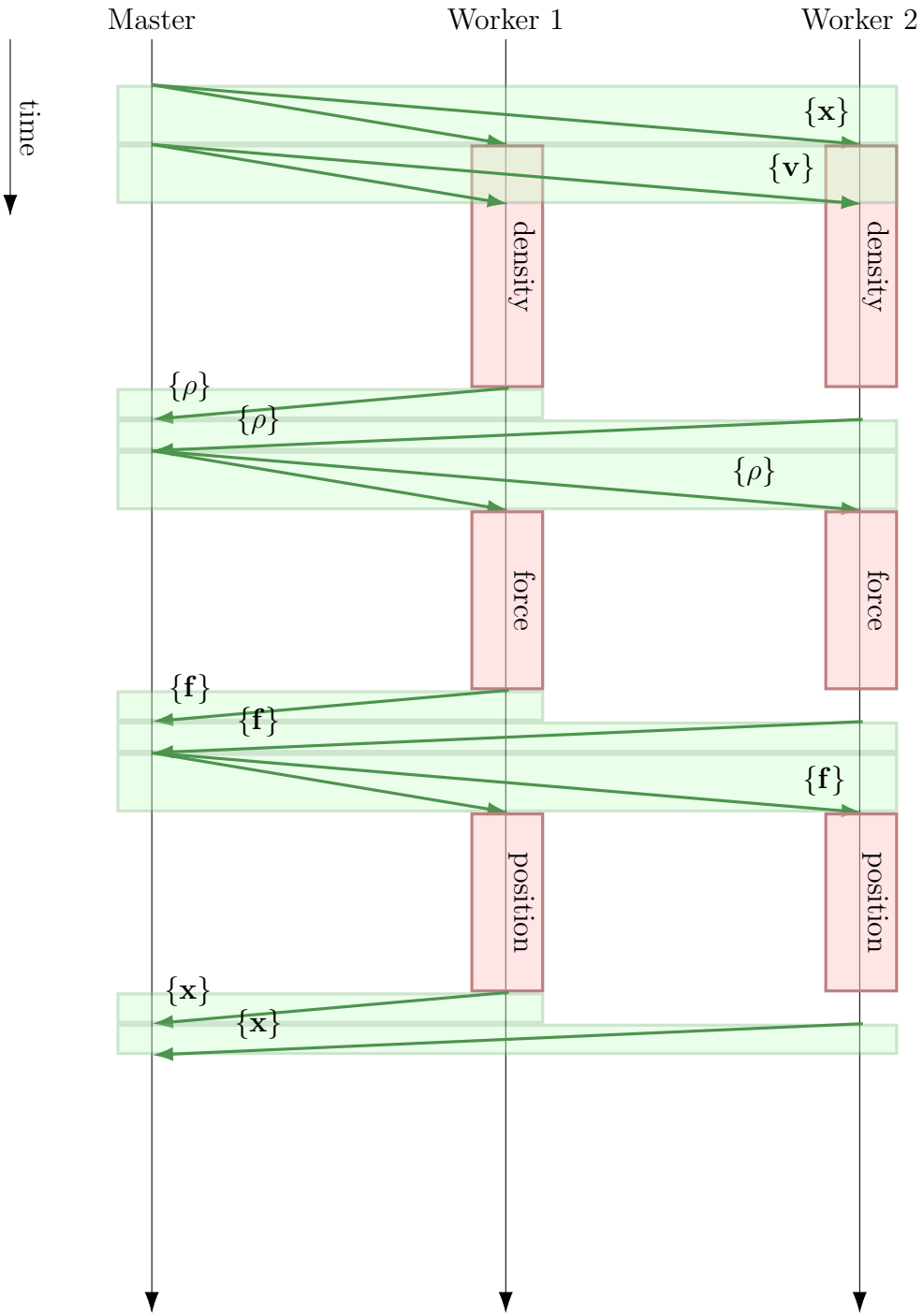

Figure 4.3: *Communication and computation timeline. Multicast is used for master to worker communication.*

4. The master starts a timer.

5. The density step is simulated: Each master thread transmits data to each worker; network contention occurs due to the simultaneous transmissions.

6. Each worker receives their data and transmits back a data buffer representing their results, as if they performed computations. The focus here is on the size of this buffer—we transmit the actual size that would be needed after computations.

7. The master joins the threads and measures the elapsed time since Step 4.

8. Steps 4 to 7 are repeated for the force computation, and then repeated again for the position computation.

9. The elapsed time is recorded. The final timing is measured as an average of 10 whole frames.

In Step 7, the master will have measured the total time needed to transmit data to the workers and receive their responses. Network contention is also naturally measured here. Worker computation time is excluded from this measurement.

This experiment was performed using Google Cloud Compute, a commercial platform that provides access to virtual machines running in a datacenter, allowing for high performance computation and networking at a low cost. Such a platform can be used, for example, by a hospital or research institution that must run high-particle-count simulations. On the hardware side, scalability is only restricted by the number of machines that can be connected to the same network. We used one machine as master and up to 4 machines as workers.

Tables 4.3 and 4.4 show timing measurements for this experiment. Two plots are derived from these tables, with emphasis given for the simulation of 15k particles, so that we can compare it with the experiments performed on Section 4.1.1. In the plot shown in Figure 4.4, we see the measurements for 15k particles, for the unicast case. The corresponding data is shown for the multicast case in Figure 4.5. The effect of network contention can be seen for the unicast case.

1 worker			
Particles	Dens.	Force	Pos.
5000	1.12	1.05	1.99
10000	1.90	1.83	3.46
15000	2.55	2.37	4.83
20000	3.72	3.13	6.59
25000	4.03	4.30	9.10

2 workers			
Particles	Dens.	Force	Pos.
5000	1.19	0.80	1.56
10000	2.03	1.38	2.94
15000	3.26	2.07	5.06
20000	4.29	2.71	6.62
25000	5.87	3.11	8.74

3 workers			
Particles	Dens.	Force	Pos.
5000	2.70	2.13	1.92
10000	3.44	1.59	4.86
15000	4.97	3.58	7.15
20000	7.01	3.23	10.40
25000	9.23	4.12	13.69

4 workers			
Particles	Dens.	Force	Pos.
5000	1.48	1.73	1.85
10000	4.46	1.83	5.76
15000	7.34	2.62	9.12
20000	8.89	4.04	13.53
25000	12.60	5.31	18.25

Table 4.3: *Average data transfer times with TCP unicast. Times are in milliseconds, measured for each step of the simulation algorithm: density, force and position. 1 to 4 workers were used.*

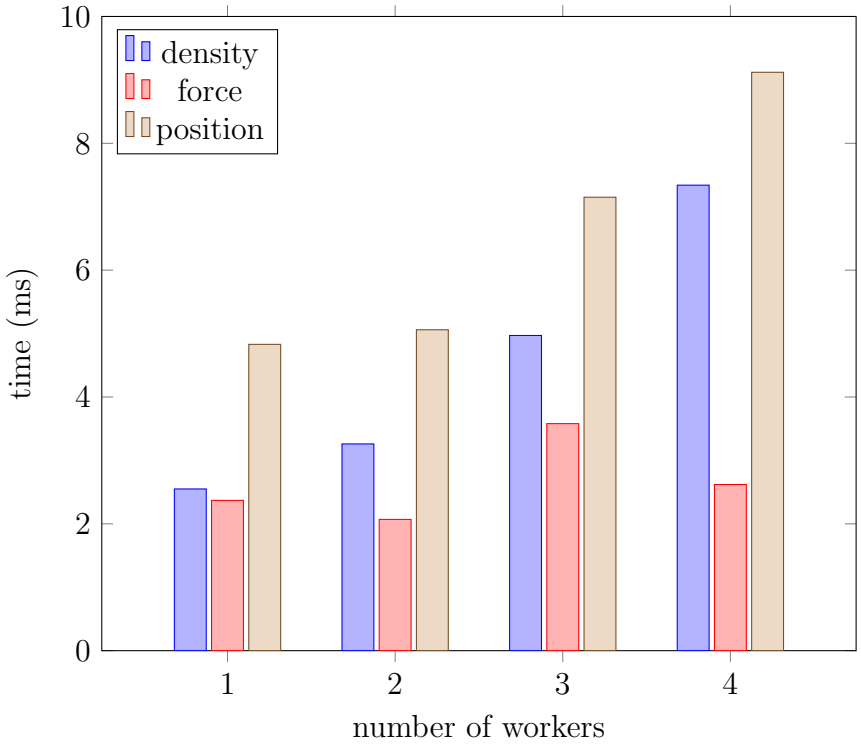

Figure 4.4: *Data transfer times: unicast, 15k particles.*

1 worker			
Particles	Dens.	Force	Pos.
5000	0.91	0.86	1.80
10000	1.60	1.60	3.18
15000	2.17	2.28	4.50
20000	2.84	3.00	5.37
25000	3.35	3.79	7.69

2 workers			
Particles	Dens.	Force	Pos.
5000	0.88	0.73	5.05
10000	1.36	1.16	2.55
15000	1.90	1.67	3.56
20000	2.72	2.33	4.96
25000	3.92	2.92	7.37

3 workers			
Particles	Dens.	Force	Pos.
5000	0.80	0.57	1.21
10000	1.50	1.17	2.30
15000	1.98	1.62	3.79
20000	2.52	1.99	4.65
25000	3.37	2.46	5.78

4 workers			
Particles	Dens.	Force	Pos.
5000	1.07	0.96	1.48
10000	1.37	1.39	2.48
15000	2.15	1.90	3.73
20000	2.65	2.47	4.84
25000	3.04	2.81	5.83

Table 4.4: *Average data transfer times with UDP multicast. Times are in milliseconds, measured for each step of the simulation algorithm: density, force and position. 1 to 4 workers were used.*

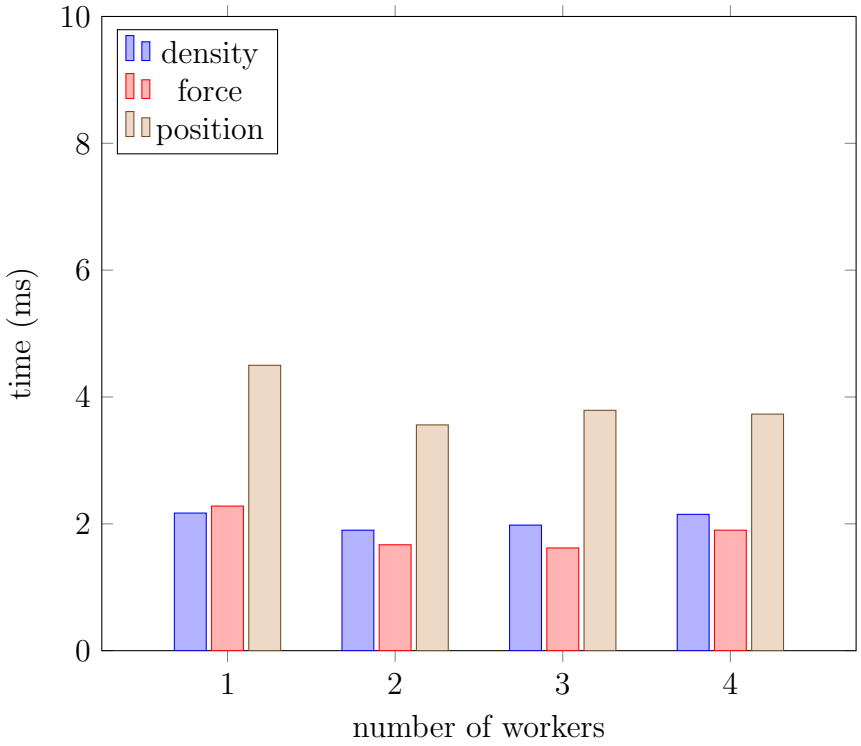

Figure 4.5: *Data transfer times: multicast, 15k particles.*

The multicast experiment was simulated by performing one unicast transmission to one node. This was done because the Google Cloud Compute platform does not expose multicast capabilities to the end user. We used the reasonable assumption that the time for a unicast transmission is roughly equivalent to the time for a multicast transmission. The result of the simulated multicast experiment was that data transfer time was approximately constant, independent on the number of workers. Random variation was observed, assumed due to other network traffic in the datacenter.

As we saw in Table 4.2, the position computation, for 15k particles, takes 92.9 ms in the 4-core multithreaded case. The data transfer time for 15k particles and 4 workers is 9.12 ms for unicast and 3.73 ms for multicast.

With this experiment, we conclude that such a SPH-based simulation can benefit for multi-CPU parallelism, both with multicast and with unicast. The data transfer time is much smaller than the computation time, so the computation can be split among multiple machines with a gain in performance.

Further reduction in data transfer time can be applied with floating-point compression. Recent research in this area presents both lossy and lossless compression algorithms that are fast and adequate for multi-CPU particle simulations (Lindstrom, 2014).

4.2 GPU Parallelism

Current GPU architectures typically contain two types of memory: slower global memory, accessible by all threads, and faster shared memory, which is local to a thread block. There is also a small amount of local memory per thread. Figure 4.6 represents this architecture.

We parallelized our SPH implementation on the GPU using global memory. This implementation is described below. Many challenges are associated with GPU programming. Debugging is difficult, and data dependencies can lead to hard to find bugs. We found that the following steps can help the task of converting code from the CPU to the GPU:

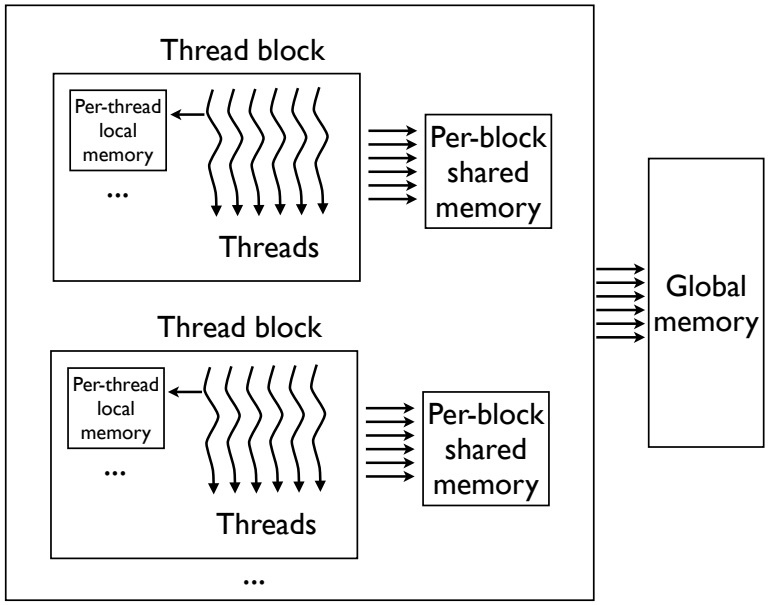

Figure 4.6: *Multithreaded GPU architecture.*

- Start from working, debugged CPU code. Debugging in the CPU is easier and there is a wider range of available tools.
- Port the code to the GPU using global memory. The amount of global memory is relatively large, and global memory is accessible from all threads, simplifying this task. It is desirable, at this point, that the code be structured in a way so that it can be compiled for the CPU or the GPU through a compilation conditional. This helps with eventual algorithmic improvements.
- Optimize the code by using block-shared memory instead of global memory. This is a hard task that involves deep changes, however it benefits from the fact that there is already working code for the CPU and GPU.

On our global memory GPU implementation, the data structure for the per-cell linked list of particles was modified. There are two reasons behind the modifications. First, pointers to CPU memory and GPU memory are different and should not be confused. If a linked list uses pointers on the CPU, these pointers will potentially not point to the same objects in GPU memory. To overcome that, instead of a list, each cell now holds an integer, which is

the index of the first particle in this cell, or a special value of -1 if the cell is empty. Second, it would be costly to gather a linked list of references to particles (potentially fragmented in memory) and copy it to the GPU. So, instead of a separate linked list, particles themselves have an integer index pointing to the next particle on the list, or -1 if it is the last one. That is, the linked list is embedded in the particle data structure. This simplifies the data transfers.

At each frame, all particle and cell data are copied to global memory on the GPU. Since the amount of GPU global memory is relatively large, this is feasible. Particle data consists of the full array of particles, each one containing: position $\mathbf{x}$, velocity $\mathbf{v}$, force, density, and next particle index. There are also two extra fields: $\mathbf{v}_{new}$, the particle's new velocity, and $\mathbf{x}_{new}$, the particle's new position. These fields temporarily hold velocity and position for the current frame being computed. Cell data consists solely of the integer index for the first particle in the cell.

Once all data are copied to the GPU, a kernel is launched to compute the particle densities. The kernel structure consists of one thread per particle. A synchronization barrier is needed after the densities are computed, since the force computation, which is done on the next step, uses density values from for all surrounding particles. Density values are stored in the particle data structure in GPU memory; there is no need to copy this data back to the CPU at this point.

After the synchronization barrier, a second kernel is launched to compute both forces and new positions and velocities. For the force computation, densities of surrounding particles are consulted. Once a particle knows the resultant force acting upon it, it can immediately perform its own velocity and position integration. The reason for the fields $\mathbf{v}_{new}$ and $\mathbf{x}_{new}$ now becomes clear: A particle can compute its new velocity and position, but it cannot update its current velocity and position because other particles may read them when computing their forces.

At the end of this step, all particles have their $\mathbf{v}_{new}$ and $\mathbf{x}_{new}$ values. The CPU takes care of performing collision detection and transferring $\mathbf{v}_{new}$ and $\mathbf{x}_{new}$ to the actual position and velocity fields, $\mathbf{x}$ and $\mathbf{v}$.

Goswami et al. (2010) describe a method for GPU SPH using shared memory. There is a limit to the number of particles per cell, and if a cell contains more than the maximum, it is split and computed in two separate blocks.

Morton code (Morton, 1966), also referred to as Z-indexing, can be used to sort grid cells such that particles that are close to each other in space are also close to each other in memory. This helps either cache coherence for CPU access (Ihmsen et al., 2011) or data transfer to shared memory on the GPU (Goswami et al., 2010).

CHAPTER 5

Visualization and Navigation

We now focus on the problem of visualizing and moving the camera in order to enable the system to be used for diagnostics, education, and collaboration. Our goal is a simulation system that animates captured heart data and simulates the behavior of fluids inside, and that is easy to use to produce smooth animations. It must be provided with adequate camera controls, and a narrative subsystem to support education and collaboration.

5.1 The Visualization Software VSim

We have previously worked on the development of the visualization software VSim (Poyart et al., 2011). Originally, VSim loaded and displayed 3D models of architectural and archeological importance, with the goal of supporting education and collaboration. In this chapter, we describe the relevant aspects of VSim, such as the ability to create narratives and its camera control model. We propose that the VSim narrative and camera models are good choices for visualizations beyond architectural models, in particular heart structures and fluid movement inside the heart.

VSim addressed the difficulty in sharing digital architectural content by providing a user interface that is intuitive and effective for both spatial navigation in the 3D world and the user's interaction with the software. To address advantages and disadvantages inherent to different camera control methods, a switchable camera control scheme is used, allowing the user to select between two modes. A novel cameral control method used in one of these modes ensures smooth camera motion.

We also proposed a system for the end-user to create narratives which provide predefined navigation paths in the 3D environment, and to add contextual material consisting of text, images, videos, sounds, and web links. This system was designed to minimize the entry barrier for new users, while still being powerful and flexible. We also provide the possibility of adding spatially-localized resources, which do not follow a narrative but are presented to the user at the appropriate locations, for example when looking at a certain heart structure. Finally, we discuss the important issue of enforcing copyrights and branding for the content used in VSim.

VSim is based on a non-trivial integration of known techniques that have been adapted to our problem domain (originally of 3D Humanities Content Visualization, and subsequently extended to medical simulations). Some of these adaptations are novel and a contribution in their own right. These contributions were outlined in Chapter 1.

Section 5.2 presents our two approaches for camera control and navigation in the 3D environment. Section 5.4 describes the system for creating narratives, and Section 5.5 discusses the embedded resources. In Section 5.7 we describe our graphical user interface, its model of smooth motion, and how it can be adapted to touch-screen interfaces.

5.2 Navigation in the 3D Environment

From a computer graphics point of view, the problem of navigating in a 3D environment is essentially one of controlling camera position and orientation. These camera parameters must be updated at every frame, while taking into account user input (keyboard, mouse, joystick and touch-screen devices), and software-driven events (e.g., collision with geometry, or the playback of a narrative).

Video games are a major portion of the entertainment industry, and considering that many video games are essentially 3D virtual environments, it is natural to look at them for inspiration. Furthermore, users already familiar with video-game-style controls, when faced with the transition to a new software, will feel natural if the control mechanics are simi-

lar. However, simply reproducing video-game-style controls does not solve all the problems identified for our real-time visualization software. In particular, while video-game controls are suitable to fast-action aiming at targets, they are not suitable for recording movies with smooth motion, or for pedagogical applications and large-audience situations.

User studies with university faculty made before the development of VSim have covered ease of navigation and use in a classroom setting. Faculty interviewed was, in general, hesitant to try to navigate the 3D models themselves during classes while they talked about them at the same time. The navigation system present here, including the camera controls and the narrative system, was designed to give instructors a non-threatening way to engage with the models as instructional technology.

We have decided to provide the user with two camera control options, called the *uSim* mode and the *first person* mode. These modes are described below. Both of them follow the "flying vehicle control" metaphor described by Ware and Osborne (1990).

5.2.1 uSim Mode

This mode received this name because it is similar to the control mode used in *uSim*, a previous visualization software developed at UCLA (Jepson et al., 1995). The main goal of this mode is to provide movement that is as smooth as possible. A secondary goal is to allow the user to control both camera orientation and velocity with a single hand on the mouse, an important feature for teachers and lecturers that were interested in using the software as a tool during a classroom presentation.

In this mode, we define a point $\mathbf{c}$ in two-dimensional space, residing at the center of the screen. Let $\mathbf{m}$ be the mouse position in the same coordinate system. As the user moves the mouse away from $\mathbf{c}$, the vector $\mathbf{v}_{\text{usim}} = \mathbf{m} - \mathbf{c}$ defines a direction and magnitude. The camera movement at each time interval dt (e.g. at each frame) is defined in the following way: the horizontal component of $\mathbf{v}_{\text{usim}}$ defines a rotation around the vertical (y) axis, and its vertical component defines a rotation around the horizontal (x) axis. The y axis is defined in a

world coordinate system, always pointing up. The x axis is defined in a local, camera-centric coordinate system.

With this system, the mouse position on the screen defines the angular velocity of the camera. An interesting consequence is that even if the user does not provide very smooth movement with his/her hand on the mouse (which is in fact hard to do), smoothing is a result of the fact that his/her hand movements are translated to a first-order derivative of the camera orientation. Any jerkiness is translated to jerkiness in velocity, not in position. Integrating in time the velocity to compute the position has a natural smoothing effect. In both this mode and the first-person mode described below, the spacebar key toggles between the camera-control mode and the mouse-release mode, the latter meaning that the mouse is no longer used to control the camera, and can be moved around to click at user-interface elements like menus and icons.

Camera linear velocity is controlled by the mouse buttons: the left button accelerates and the right button decelerates. For the application of architectural visualization, the user can choose between being allowed to fly, in which case the camera motion vector always coincides with the eye forward vector, or being constrained to ground level, in which case the camera motion vector is a normalized horizontal projection of the eye forward vector. A collision detection system ensures that the camera is always at a constant distance from the ground. If the user attempts to climb over a step that is small enough, the camera is allowed to climb, updating its height for the next section of terrain. To apply this camera system to heart simulation visualization, we allow only the *flight* mode.

5.2.2 First-Person Mode

In this mode, mouse movements are directly translated to camera movements. If the mouse moves from position $\mathbf{m}_0$ to position $\mathbf{m}_1$, the vector $\mathbf{v}_{\text{fp}} = \mathbf{m}_1 - \mathbf{m}_0$ defines the camera rotation. If the user stops moving the mouse, $\mathbf{v}_{\text{fp}} = 0$ and the camera stops moving.

This is the control method used in most first-person computer video games. Notice that in this mode, mouse movements are directly translated to the camera orientation, rather

than to its first-order derivative. Any jerkiness in mouse movement is translated directly to jerkiness in camera rotation. On the other hand, an advantage of this mode is that the user can quickly point the camera to any direction, as if he/she is turning his/her head. It is arguably more intuitive to think that one's hand is directly controlling the look-at vector, rather than to think that it is controlling the camera's angular velocity.

In order to improve the smoothness of the camera's motion in this mode, without compromising the advantage mentioned above, we proceeded as follows: Let p_0 and w_0 be, respectively, the pitch and yaw components of the camera's orientation at time t_0, as defined with respect to an arbitrary global orientation system. The movement vector $\mathbf{v}_{\text{fp}}$ is then translated into "desired" values, p_{desired} and w_{desired}. The current p_0 and w_0 values are interpolated towards p_{desired} and w_{desired} following an inverse exponential law

$$p(t) = p_0 e^{-jr} + p_{\text{desired}}(1 - e^{-jr}), \quad r = t - t_0, \tag{5.1}$$

where t is the current time and j is a scaling constant. An analogous formula is used for the yaw w. An efficient stepwise integration method to approximate this behavior can be implemented as follows. At each frame (a time interval Δt), p is moved towards p_{desired} a distance Δp proportional to the magnitude of the distance between them; i.e.,

$$\Delta p = k \Delta t |p_{\text{desired}} - p|. \tag{5.2}$$

We obtained best results with $k = 8.0$. This value can be intuitively understood the following way: If the speed were to be defined on the first frame and kept constant, p would take $1/8^{\text{th}}$ of a second to reach p_{desired}. The actual time is larger, since the frame rate is usually higher than 8 fps. Even though mathematically this would result in motion that gets slower and slower but never stops, in practice the limits of floating-point accuracy are reached quickly, which makes the motion stop. A threshold point could have been introduced, but, in our tests, we did not have any noticeable residual motion when the user stopped moving the mouse, so the threshold test was not necessary.

This method essentially absorbed a large amount of high-frequency, unintended mouse jerkiness and provided a smoother movement while keeping high interactivity.

Linear velocity of the camera is controlled by the keys **w** (forward), **a** (left), **s** (back) and **d** (right). This control scheme is used in numerous first-person computer games. However, in our case, the velocity does not change immediately, but linearly increases and decreases over a short period of time, limited by a maximum velocity. We used 0.66s for this time value, and our conclusion is that it should be kept small. We essentially follow the same philosophy as for camera rotations—there should be smoothness in motion, but at the same time the user should feel crisp and responsive camera control.

5.3 Look-Aside

During interactive visualization or in demonstrations, while moving around the environment, often it is desirable to look at a feature that attracts your attention, while your movement continues in the original direction. The analogy is as follows: When walking down a pathway at UCLA with foreign colleagues and showing them the place, you notice a building that attracts your attention. You point it to them, and they turn their heads to look at it, while continuing walking and talking about it.

We designed a novel mechanism for doing exactly that—looking to the sides without changing your direction. A keypress is defined to act as a *look-aside* key. We used the *shift* key for that purpose. When it is pressed

- the movement vector of the camera is kept constant, regardless of camera orientation movements, and
- the camera is free to look around and to point at objects of interest while maintaining its original movement vector.

When the look-aside key is released, the current camera orientation may not be the same as the movement vector. The user may or may not have pointed the camera back to the

forward vector. How to reconcile the two? One could think about smoothly rotating the camera so that it points back at the movement vector. We found that this is not desirable, since the user took the action of pointing the camera somewhere else. The approach taken was the opposite—smoothly interpolate the movement vector so that, over a small period of time, it matches the current camera vector again. The big advantage of that approach is that the camera control is always at the user's hands, never being taken away from them.

The interpolation used is a regular vector interpolation, rather than spherical: If the camera is pointed close to 180 degrees from the movement vector (such that you are moving backwards while pressing the look-aside key), the end result is that your movement comes close to a stop and then starts again in the forward direction. With spherical interpolation, there would be a strong, undesirable sideways movement in the middle of the interpolation.

We found that this mechanism works best in the first-person mode, but it can also be used in the uSim mode.

5.4 Narratives

Another feature of our framework is the ability to create *narratives*—a mechanism for both the content contributor and the end-user to create arguments, tours and lesson plans, augmented with on-screen multimedia content. The user adds *narrative nodes* (keyframes) defining the camera position and orientation, and organizes these nodes in a timeline. Figure 5.1 shows the narrative editor UI and its constitutive elements. Figure 5.2 is a detailed view of part of the narrative editor bar. Even though VSim's narrative concept was originally developed in the context of architectural visualization, the same concept can be applied to medical visualization, except for the fact that there is no ground plane or "walking" simulation, everything else is applicable.

Nodes in a narrative are created by simply positioning the camera and clicking on the "+" button. With this action, a narrative node is added to the timeline. Data associated with nodes include camera position and orientation, time to remain in the node, and node overlay

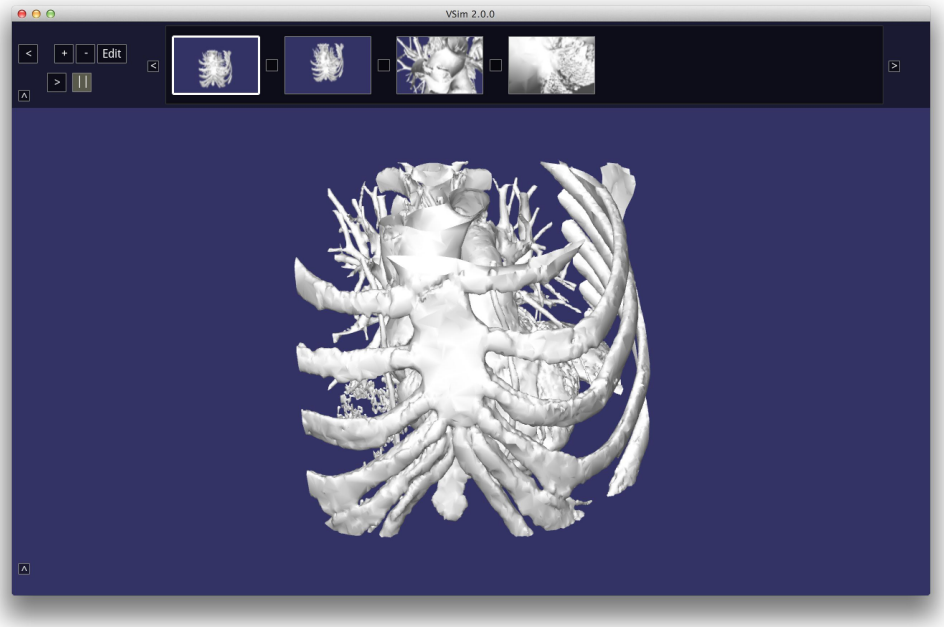

Figure 5.1: *The narrative editor. The central area of the top bar contains narrative nodes and transitions. The control areal on the left contains buttons for creating and removing nodes. The Edit button launches the overlay editor for the selected node.*

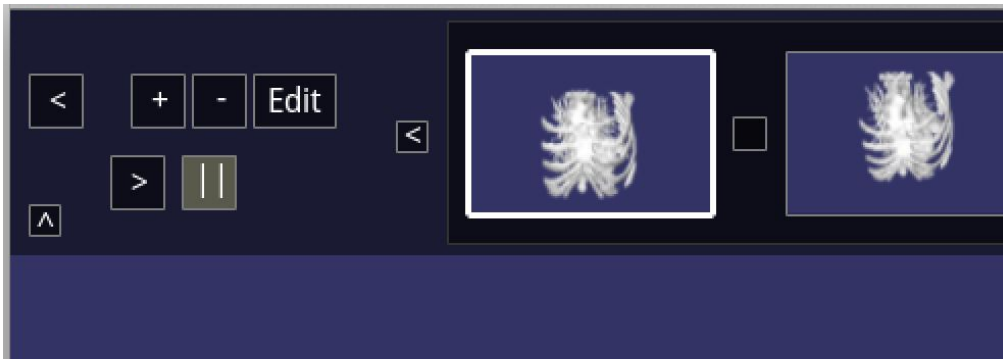

Figure 5.2: *A detailed view of the narrative editor bar.*

content (described in Section 5.4.2). The user can edit the timeline by directly manipulating and reorganizing the nodes.

When playing the narrative, the camera position is interpolated between nodes in the manner described below.

5.4.1 Camera Movement

The camera follows straight lines between nodes. Second-order continuity at nodes was not a requirement for the first version of the software, since users were mostly interested in stopping at nodes and showing content there. It can be added in the future, if needed.

Within the straight line followed by the camera, its position is interpolated in an ease-in/ease-out fashion using a cubic function as described below. We start with the following function:

$$y = x - x^3. \tag{5.3}$$

From this function, the x axis is scaled so that the inflection points $x = \pm 1/\sqrt{3}$ fall at 0 and 1, and the y axis is scaled so that the minimum and maximum in that range also fall at 0 and 1. We input a linear interpolant into this function as x, with values ranging from 0 to 1. The output y is a cubic interpolant that is used both for camera position and rotation (as a quaternion *slerp* interpolant). By using this method for ease-in/ease-out instead of a quarter-sine, we avoid the expense of a sine computation.

5.4.2 Overlays

Each keyframe, or node, can be enriched with textual information, images, videos, and sounds—called *overlays*. The user can freely lay out these elements on the screen in 2D space while in a keyframe. When a narrative is playing and a keyframe is reached, its overlay fades in and is displayed for an user-determined amount of time, or until the user presses a key. This function essentially allows the content contributor and end users to

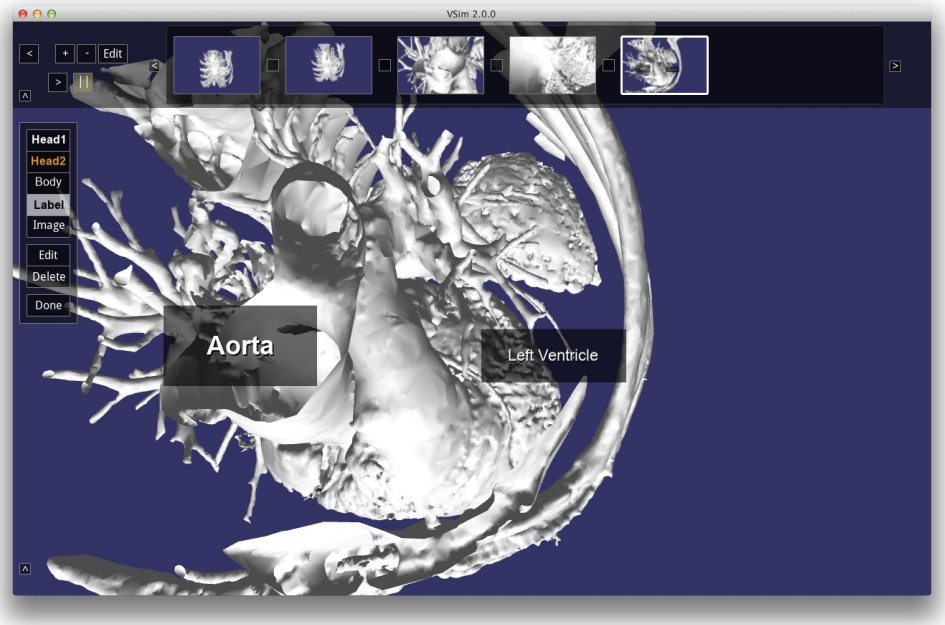

Figure 5.3: *The overlay editor.*

augment the virtual world with multi-media content. Figure 5.3 shows the overlay editor with a text element added to the scene.

5.5 Embedded Resources

Users may wish to explore the model through free navigation; i.e., not following a narrative. It is natural to think that these users should also be presented with contextual multimedia elements (*embedded resources*) associated with locations in the environment. As an example, when the user is viewing a structure in the heart such as a valve or an artery, an embedded resource could show documents related to the structure observed—pictures of the actual organ, text, video and audio recordings, and so on. Due to the fact that the user should not be interrupted in his/her continuous navigation, embedded resources do not pop up on screen, but rather, icons appear on the *embedded resource bar* at the bottom of the screen. The user can choose to view them or not.

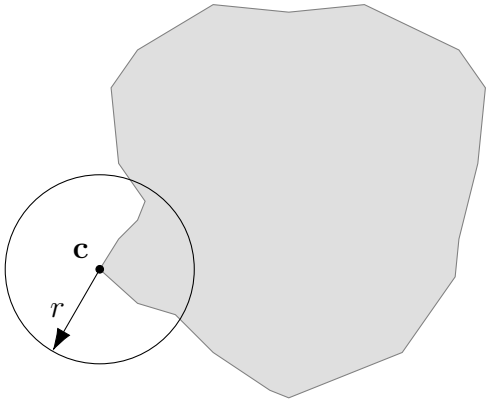

Figure 5.4: *Embedded resource positioning: naive approach. The gray shape is the model being viewed. A point in space* **c** *and a radius* r *are used to describe the area of interest. This point is both the point of interest and the center of camera influence. Notice that the user can see resources associated with point* **c** *in the model, whether he is on the outside or on the inside of it. The author would like the resource to be seen from the outside of the model only, which this approach does not capture.*

An important question is how to define the area in which an embedded resource icon should appear. A naive approach would be to associate the resource with a point in space, and make it appear whenever the camera is inside a sphere of a certain radius, centered at this point. However, this would show resources that are behind the user. Another approach is to combine the sphere with a specific camera angle, which does not entirely solve the problem, as the user could be, for example, on the inside of the heart, whereas the structure of interest can only be viewed from the outside, or vice-versa. Figure 5.4 illustrates this problem.

The approach that we took is described as follows: The content creator defines a resource's location in the model through a data structure composed of

1. target **t** (3D position),
2. camera position center **c** (3D position), and
3. camera area radius r (scalar value).

The embedded resource becomes available to the user and appears in the embedded resources bar whenever the following conditions are true:

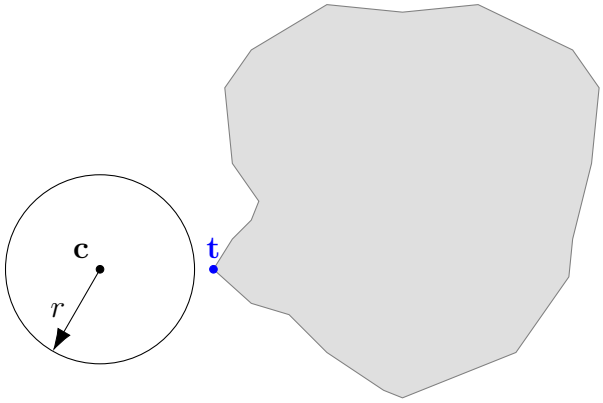

Figure 5.5: *Embedded resource positioning: our solution separates the center* **c** *from the target point* **t**. *The embedded resource is activated if the camera is inside the sphere and the target* **t** *is in the field of view. The author can express the fact that the resource should appear when the user is outside of the model and looking at the point of interest.*

1. The scene has the target point **t** in the field of view.

2. The camera is inside a sphere of radius r centered at **c**.

Past approaches essentially had the center **c** and target **t** combined as the same value. By separating them, we allow resources to appear whenever the user is, for example, *in front* of the heart and looking *at* the heart. Figure 5.5 shows a model, in gray. Points **c** and **t** are shown, as well as the sphere defined by radius r. If the user, in his/her virtual navigation, steps inside the sphere and turns the camera towards the point of interest **t**, both conditions are met and the embedded resource appears. This corresponds to the author's intention—he/she can now express the fact that the resource should appear when the user is in front of the model, not inside of it, and looking at the relevant feature in the model.

If desired, a flag can be added to the embedded resource data structure to inhibit the need for a target; this would allow the authoring of resources that "attract attention", independent of the direction the user is looking; however, we have concluded that this mode should not be the default.

An inside-sphere test has to be made on every frame and for each resource, and there could be many resources associated with a scene. The use of spheres makes this computation efficient. Other, more complex shapes could be used as area of influence, but spheres are

adequate in most cases, and they also simplify a content creator's method of thinking about the problem.

5.6 Restrictions and Enforcing Copyright

We incorporated a lightweight model of restrictions, through which the content creator can have some level of assurance that his/her content will not be easily modified, and that his/her branding elements will not be easily removed.

The content creator can set a series of flags that are associated with his redistributable file (a file that combines the 3D model with meta-information about narratives and embedded resources). These flags enable or disable VSim functionality for, among other things:

1. Adding and removing a branding bar, which can show, for example, the institution logo.
2. Modifying narratives.
3. Modifying embedded resources.
4. Keeping the user restricted to paths defined by narratives; i.e., no free-form navigation.
5. Creating video files.
6. Increasing the screen resolution beyond a certain limit.

The initial implementation is lightweight, and it is a step toward a more secure implementation. We decided not to add encryption and the associated burden of encryption keys. Taking into account the fact that the software is open-source, it is not impossible for someone with the source code in hand to circumvent the restrictions. However, it is not easy for someone without programming expertise to do so, which is adequate for our requirements. Security can be enhanced in future versions by means of encrypted data files, building on top of our current system.

5.7 Graphical User Interface

Two ideas guided the graphical user interface design—it should be intuitive and easy to use, and it should be adaptable to emerging technologies such as tablet devices and touch-screen interfaces.

We envision a typical user of the software to be a lecturer in a discipline in humanities. This user may have little or no experience with other 3D navigation software and with tools to create narratives and movies. We designed a *narrative bar* residing on a *control bar* at the top of the screen. It consists of *nodes* (large rectangles corresponding to keyframes, containing a thumbnail of the scene) and *transitions* (smaller rectangles in between nodes). Accessing a node opens up a window to control node aspects (time to remain in node, fade in/out time, and others). Accessing a transition, similarly, allows the user to control variables such as transition timing. Adding and removing nodes is done through buttons labelled "+" and "−", a concise and efficient representation.

All GUI animations are smooth. For example, the side scrolling of the narrative bar (when there are more nodes than the screen can fit) is visually a rolling movement rather than a jump. This gives the user important visual cues. Similarly, the whole control bar (which is semi-transparent) can be closed to reveal the whole 3D scene, also with a smooth movement. We found that the most pleasant motion was achieved not linearly, but by making the speed follow an inverse exponential law, similarly to the first-person camera (Equations (5.1) and (5.2)).

The UI was designed on a style that is adequate for porting of the software to tablet platforms. All interactive elements (widgets) are large (by being "finger-sized", they can be used with touch-screen input, as opposed to mouse clicks). Rather than separate windows with OS-specific features, the widgets are rendered using OpenGL on top of the 3D environment. A custom widget system was developed for that purpose. By sidestepping this OS dependency, the widget system should work out-of-the-box on touch-screen platforms, and the look-and-feel is preserved. Minor OS-specific elements are still used; e.g., on dialog boxes to open files, which should not present a big portability problem.

CHAPTER 6

Conclusions and Future Work

6.1 Conclusions

We have discussed Smoothed Particle Hydrodynamics, its application in fast fluid simulation, and we presented our implementation and discussed the challenges involved. We provided an extensive analysis of three parallelization approaches, (1) multithreading on a multi-core CPU, (2) offloading computations to a general-purpose GPU, and (3) a multi-machine networked parallelization approach. This latter option allows highly scalable parallelization and it can be implemented to run on virtual machines in existing low-cost cloud services. We analyzed the scalability with the use of both unicast and multicast protocols, and we showed that the data transfer times over the network are small enough to provide speed-up and scalability.

A method to use acquired volumetric scan data to drive the simulation was shown. An important goal of this thesis was to perform simulation on heart data acquired from patients, while keeping the manual, hand-crafted data preparation to a minimum. We showed how this is possible with no hand-crafted preparation, although due to scan resolution the results suffered from inter-ventricular fluid leakage. This is an important step towards the goal of simulating a fully extracted heart mesh, including the inter-ventricular septum. This goal becomes more and more realistic with higher resolution and higher accuracy 4D scanning technologies.

To achieve Lagrangian simulation contained by the extracted heart mesh, we devised a novel mesh interpolation method using a variant of Iterative Closest Points. This allows a smoother and more stable simulation. Our particle-wall interaction, another new aspect,

produced good results both in simple "dam-break" tests and in the actual heart data. Finally, we simulated a heart mesh filled with 25k particles and made it successfully pump simulated blood.

We argued that although it is a software system that we originally developed for visualization and collaboration in the archaeological and architectural domains, VSim, with its novel camera control and interaction method as well as its capabilities for the creation of narratives and embedded resources, can also be applied to medical simulation, providing benefits when used for medical training, research, and collaboration. VSim is currently being used at UCLA in an experimental phase for classroom presentations, in the area of visualizing architectural and archaeological content. The initial response of users that have been exposed to the prototype software has been positive.

The novel camera control method described for VSim is especially applicable. There are advantages and disadvantages to both camera control modes that were tested, so we followed the route of implementing both, and allowed easy switching between them. The *first-person* mode is easy to use and provides fast and direct camera control, whereas the *uSim* mode has smooth motion appropriate for video recording and single-handed control appropriate for classroom presentations. We believe that offering both modes was the ideal solution.

We have also approached GUI issues with an aim to keeping the software portable and adaptable. By developing a custom widget library in OpenGL, we have seamless integration of UI elements with the 3D environment, and the possibility for future adaptation for touch-screen interfaces. We found that the benefits of developing such a custom library offset its cost.

Narratives in VSim allow end users and content creators to provide argumentation within the 3D environment. Embedded resources allow the augmentation of the 3D environment with relevant content for users that choose to perform free navigation. The elegant scheme developed for the activation of embedded resources based on camera position and orientation has met all our requirements, allowing the content creator to express when and where the resources should appear in a variety of scenarios.

6.2 Future work

The results presented in this dissertation can be extended in several ways:

- Adding valves to the heart to replace missing structures extracted from the CT data, and generating a more accurate simulation. With that, we can observe blood flowing separately in the left and right ventricles and being pumped through their separate circulatory systems.

- Using higher resolution volumetric scans and fine-tuning the parameters in order to extract the intra-ventricular septum.

- Simulating blood flow inside vessels in ultrasound and MRI data volumes. A level set can be extracted from the data to identify vessel boundaries, and SPH particle simulation can be performed inside these volumes. Some questions that can be answered are: How many particles are needed for an accurate simulation, how close can we get to real-time, and what simulated parameters (e.g., blood pressure) can we extract from the simulation?

- Experimenting on fluid coupling with solid structures, such as needles, and semi-solid structures, such as heart valves. For example, two goals can be attempted, (1) use 4D ultrasound of a beating heart to simulate the flow of blood being driven by the heart imagery itself, and (2) simulate a needle insertion, with the consequential bleeding and turbulence effects in the blood and vein.

- Extending the research into the Computer Architecture area to define modifications to current parallel architectures that can accelerate particle-based simulations.

Additionally, it is interesting to consider a piece of work that we previously published, on particle-based hair simulation, which is presented in Appendix B. The relationship between that work and Lagrangian simulations is that hair simulation for animation purposes can benefit from results obtained in Lagrangian fluids, particularly with regards to GPU parallelization.

Finally, our experiments with ultrasound volumes, which are presented in Appendix A, have demonstrated the feasibility of automated volume registration, and how it can be made more robust through multiresolution pyramids. The medical team in our research group is doing a great deal of manual work after capturing ultrasound volumes for training purposes. Volume registration is one of their future challenges. Reducing the amount of work needed for registration can have huge benefits, even if the need remains for an initial, approximate manual registration. Future work would involve validating our experiments with our heart blood flow simulation by comparing them with 4D ultrasound, which can capture the actual blood flow and associated tissue deformations. Also, it should be possible to devise a procedure for ultrasound capture that enables automated volume registration and can be used to provide large volumes of data to drive simulations of interest.

APPENDIX A

Ultrasound Data for Circulatory Simulation

Ultrasound imaging plays a very important role in medical care. Ultrasound equipment is small, light, and inexpensive compared to magnetic resonance (MR) imaging or computed tomography (CT) equipment. It is non-invasive and produces no radiation. CT imaging is a fast method but uses radiation. MR imaging can take a half hour or more, and it cannot be used if the patient has a pacemaker. Both methods are expensive. Ultrasound, on the other hand, can be used in the emergency room to detect problems, such as organ rupture or internal bleeding, that may require immediate surgery. Ultrasound is real-time, so it can provide image guidance for procedures. It can also show the tissue response as the physician performs manipulations while scanning. Due to the low cost, more hospitals have more ultrasound equipment than other sophisticated imaging equipment. The drawback of ultrasound is the low resolution and noise in the produced images.

A.1 Ultrasound Imaging

The principle of ultrasound imaging is as follows: A transducer positioned at the tip of a probe is held in contact with the patient's skin. This transducer emits ultrasonic waves, which travel through tissue. A reflection happens whenever there is a change in density. The transducer captures these reflections and transforms them into images. Usually, the transducer emits waves in a fan-like shape, so the image produced is a planar slice through the body corresponding to the plane where the ultrasound waves travel. Ultrasound probes can also be curved, linear, or have multiple configurations, performing more complex scans.

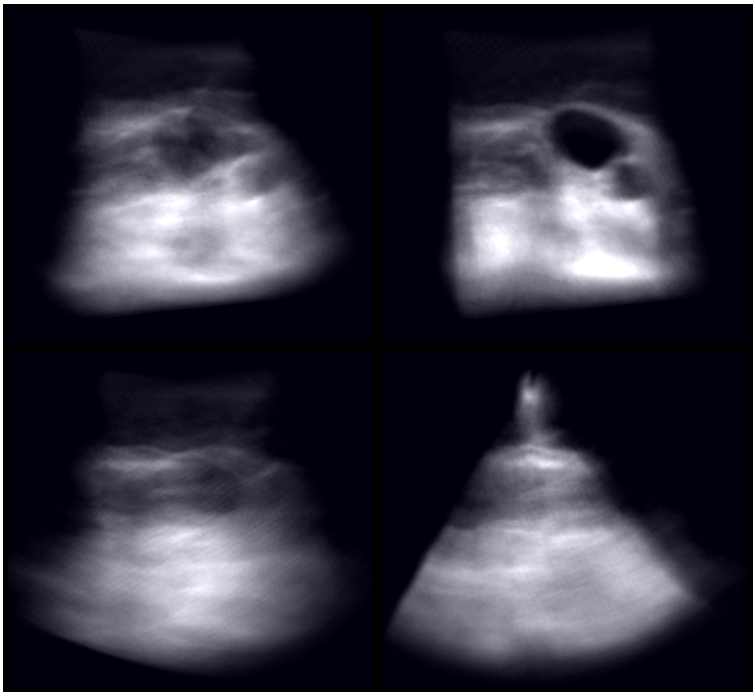

Figure A.1: *An ultrasound volume seen from different angles. Two blood vessels can be clearly seen in the top-right view.*

A special type of ultrasound probe can have a moving transducer that, instead of a single slice, captures a volume. The moving transducer basically captures several parallel slices automatically as the probe is held still, and the hardware and software processes it into a volume of data. Trained physicians can acquire volumes with little to no distortion due to hand movement. The output from the volume acquisition process is a three-dimensional grid of intensity values, which correspond to different densities of body tissues. Figure A.1 shows a volume captured this way. These images were rendered on a tablet device where they can be viewed and manipulated.

Rendering and manipulating ultrasound volumes in tablet devices can have many interesting applications in medicine. The rendering method we used is described below. It allows interactive manipulation of the volumes.

Slices are taken from the volume in the x, y and z directions and stored in textures. The textures have color set to white and alpha set to the intensity value coming from the volume. The resolution of each texture is the full resolution of the volume, but the number

of textures in each direction is a fraction of the full volume. We used a decimating factor of 4. This improves frame rate by reducing the number of pixels that must be filled, for devices that are limited by pixel fill rate, at the cost of resolution in the z direction in camera space, which has less effect in image quality than resolution in the x and y directions.

All these textures are rendered in their respective positions and orientations. To correctly visualize the volume, each pixel must be the integral of all the values along its direction. The usual alpha blending for transparent surfaces in computer graphics is governed by the following equation:

$$c = \alpha c_t + (1 - \alpha)c_0, \tag{A.1}$$

where c is the final color, c_t and α are the color and alpha (opacity) values of the texture being rendered, and c_0 is the color that previously exists in the frame buffer. This is an order-dependent method (transparent surfaces must be rendered from back to front) and, furthermore, it does not produce the expected integration for volume rendering. We instead used the following equation:

$$c = \alpha c_t + c_0. \tag{A.2}$$

This performs the expected integration of intensity values and it is order-independent. All OpenGL-compliant graphics hardware supports this rendering mode.

An extension of this method can be implemented in the following way: As the user rotates the volume and the normals of one of the three sets of textures approach a right angle with respect to the camera direction, these textures have diminishing impact on the resulting image. A customized shader can be used in order to modulate the whole texture with lower alpha values, according to the dot product between the texture normal and the camera direction, and starting from a certain angle, the set of textures can be dropped altogether.

Volumes captured with ultrasound can be used for medical training. A medical student can learn and practice in a computer-based environment using software that takes such a volume and re-slices it according to the position and orientation of a handheld probe attached

to the computer. This training probe tracks the student's hand movements using sensors not unlike the accelerometers commonly found in smartphones and tablets. Thus, a very natural way of controlling the probe can be achieved, and the image that is viewed (the re-slicing of the captured volume data) is very similar to those produced by real ultrasound equipment.

Another feature of ultrasound imaging is the ability to use the Doppler effect to detect movement in the images. This is commonly used to find blood vessels by detecting blood flow. In the produced image, which is generally grayscale, the software can draw red or blue pixels according to whether blood is flowing towards or away from the probe. This results in pulsating red/blue areas in the image that indicate the presence of blood vessels.

Ultrasound imaging can have enough resolution to capture small veins and arteries, in addition to large ones. With a large enough volume, it can capture the branching and merging structure of these vessels, and fluid simulation can be done inside of them.

Training systems can benefit from the addition of Doppler visualization. For that, a fast simulation system is essential. Two steps must be done:

1. registration of ultrasound volumes, to construct larger volumes from several scans stitched together, and

2. segmentation of blood pathways, which can be done by a level set method that constructs a blood vessel mesh (with its potential ramifications to other vessels) from the intensity image, or by other methods such as "snakes", or active contour models (Kass et al., 1988).

These two steps are analyzed in the following sections.

A.1.1 Human Circulatory System Simulation

Blood flow simulation in the human circulatory system can have more impact on health care than just for training purposes. It can be used to study the blood interaction with walls of arteries and veins, with valves (especially in the heart), and with other structures.

A special type of ultrasound probe can capture so-called 4D volumes; i.e., three-dimensional time-varying volumes. One can use a time-varying level set extracted from a 4D volume to drive fluid simulation, with interesting applications.

Needle insertion is an interesting research avenue. The objective is to study the interaction between blood and needle. One of the effects is the appearance of turbulence on the blood flow. Studying and predicting the amount of bleeding when a needle is inserted is another interesting example. Simulations such as those can be validated by ultrasound imaging itself. For example, a volume can be captured without and with a needle inserted. Simulation can then be applied to the former volume, with a virtual needle added to it. Then, comparisons can be made with the latter volume in order to validate how accurate the simulation is. The predicted and real amount of bleeding can also be compared. Since SPH offers the power to model viscosity, the blood's actual viscosity can be fine tuned, and the effects of changes in viscosity can also be observed.

Not only can blood flow simulation be applied to ultrasound data (potentially captured just-in-time in the emergency room), but also to high resolution MRA and CTA data (the A stands for angiography). As an example, the simulation of blood flow in a brain angiogram and its interaction with thicker or thinner vessel walls can be useful for the detection of aneurysms or areas where rupture of vessels can potentially occur.

A.2 Volume Registration

The dimensions of the acquired volumes on a single scan are usually small, with sides that are on the order of a few centimeters in length. In order to simulate the flow of fluids in a larger region, a number of volumes can be scanned with overlap and automatically or semi-automatically stitched together.

We have experimented with registration in the following way: We have opted to use intensity-based, rather than feature-based registration, since automatic feature tracking across different scans is not feasible with ultrasound, due to noise, shadow effects, and overall

softness of edges. We started with a typical ultrasound volume. A copy of this volume is made with a linear translation p in one direction. Then, we perform registration between the two copies. The registration algorithm finds a resulting position, and we measure the distance between the correct and the obtained positions.

In this experiment, only translation was used. Although in the future we plan to repeat the experiment with rotations, we assume that our method of capturing volumes will have a very small, if any, rotational component compared to the translational component. It is possible that this can be defined as part of the volume capture procedure for the physician to follow.

The algorithm used is gradient descent. Starting from a current position p_n, the position is updated as follows:

$$p_{n+1} = p_n + k\frac{\partial f(p_n)}{\partial p_n}, \tag{A.3}$$

where p_{n+1} is the new position, k is a constant, and f is an image metric that measures the similarity between two images. The image metric used was the sum of squared differences of intensity between voxels:

$$f(p) = \sum_{x,y,z}(I_1(x,y,z) - I_2((x,y,z)+p))^2. \tag{A.4}$$

To reduce the amount of data that must be processed, we perform the image metric computation only on a random sample of voxels in the overlapping region, rather than on the full image. For volumes with $300 \times 300 \times 300$ voxels, we used 100,000 samples.

We performed the described registration with several values of p, increasing from 0 to 40 voxels away from the registered position. The resulting minima found by gradient descent are the correct global minima up to a certain distance, beyond which the algorithm mostly fails and yields a local minimum.

The upper left plot in Figure A.2 shows the results with no noise added to the image. Gradient descent found the correct registration when the starting distance was up to around 25 voxels from the starting position. This corresponds to 8% of the volume dimension.

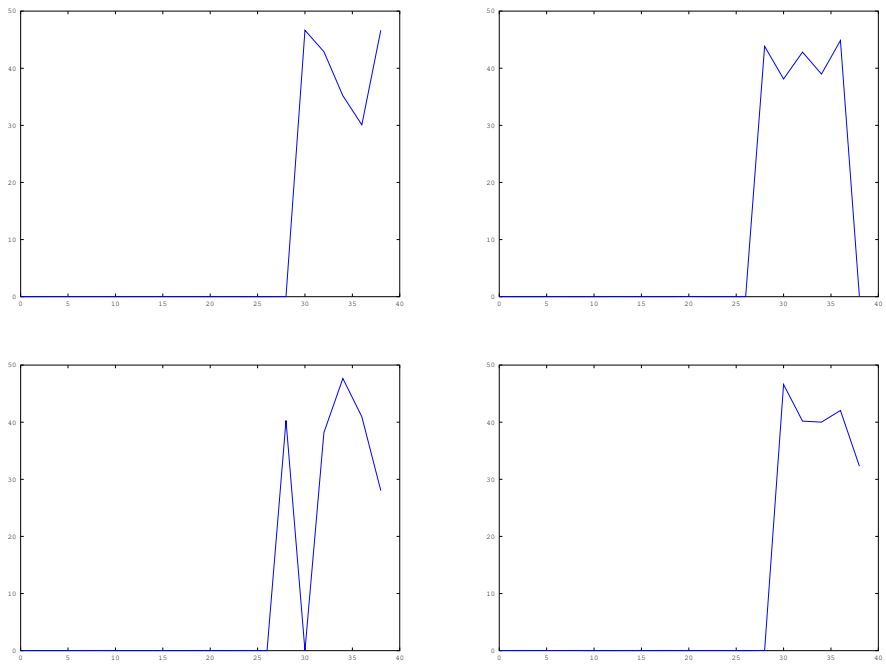

Figure A.2: *Registration: initial distance versus registration error. The horizontal axis is the starting position in voxels away from the true registration position. The vertical axis is the registration error—the length of the vector from the true registration position to the computed position. The first plot corresponds to no added noise; the second, third, and fourth plots correspond to a noise range of 16, 32 and 64 intensity units, respectively.*

We repeated the experiment with random noise added to the volumes. The noise follows a uniform distribution. The range of this distribution was 16, then 32, then 64 units of intensity. This was added to the image information, which consisted of voxels with values from 0 to 255. An interesting result that can be observed is that the noise did not have a big impact on the maximum distance that leads to good registration.

Multiresolution pyramids are a common technique in image registration, where an image is subsampled to lower resolutions and registration is performed in those lower resolutions first, giving a rough alignment that can be used as a starting point, which can is then refined at the higher resolutions. This technique can also be used with volumes; i.e., 3D images. We can perform registration in subsampled volumes first, and get to the area of convergence within 25 voxels from the correct point.

A.3 Segmentation

Volume segmentation is a related area in which work can be done. If we consider the case of ultrasound volume segmentation for simulation purposes, for example for needle insertion simulation, the work is currently done manually. This process is very labor intensive.

Level sets and mesh extraction can be used for segmentation. Analysis of the available data shows that the interior of vessels usually displays a nicely defined region of zero-values of intensity, in contrast to the vessel walls which are captured as high intensity. Other methods, such as "snakes" (Kass et al., 1988) and other active contours, can be used as well.

APPENDIX B

Hair Simulation and Parallelism

Here we follow up our discussion on parallel simulation techniques by reviewing our work on Lagrangian hair simulation and the techniques used to parallelize this simulation.

B.1 Segment-Based Head Collision

In our previous work (Poyart and Faloutsos, 2010), we showed that high-quality physical simulation of hair can be achieved with a novel approach that allows hair strands to rest in layers on top of one another, even when the head is tilted in arbitrary directions. This was done in the following way: each hair strand starts at a root position in the head, and is composed of a sequence of particles connected by springs. Particles in each strand are indexed, starting at the root position. The index of the particle determines an offset for the collision with the head, such that particles further away from the root collide at a bigger distance, therefore resting on top of other hair strands below it. Figure B.1 illustrates this mechanism.

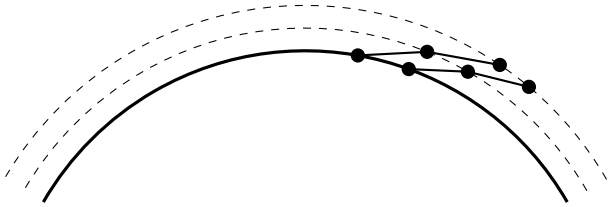

Figure B.1: *Segment based collision. The line segments between the dots are hair segments; the solid circle represents the head surface where the root nodes lie. The dashed circles are collision spheres for subsequent nodes of the hair.*

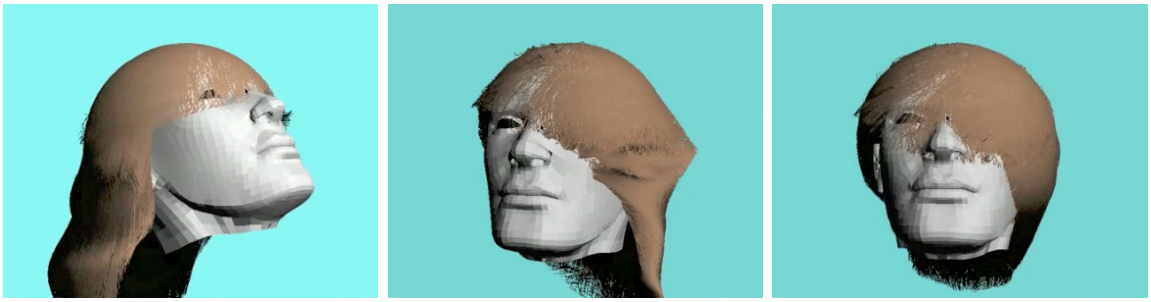

Figure B.2: *Hair simulation examples.*

This type of hair simulation is also Lagrangian and particle-based, with springs attached between the particles. Some of the aspects of particle-based fluid simulation can be applied to it, especially for collisions between hair and head as well as interactions between hair strands.

One particular problem with Lagrangian hair simulation is enforcing the constant length of hair strands, particularly when springs and masses are used. This is analogous to the incompressibility problem in SPH. We used stiff conditions to minimize hair stretchiness, not unlike the WCSPH method described in Section 2.2.3. We also implement an *error reduction parameter* (ERP) similar to the one implemented in Open Dynamics Engine (Smith, 2000). ERP is a scalar value between 0 and 1 that defines how much of the length error will be corrected at each time step. The correction is performed as a change in position of each node, along the length of the spring, towards the correct length. If ERP = 0, there is no correction. If ERP = 1, there is full correction—each node will be moved such that the length is fully corrected in one time step. Setting ERP too high, however, introduces damping, so it is set to a moderate value.

Some resulting images are shown in Figure B.2.

B.2 Hair-Hair Interaction and Parallelism

In our implementation, we did not have hair-hair interaction. An approach to solving that would be to approximate hair-hair interaction by the use of potential fields, like the density and pressure fields in SPH. Forces on the attracting region of the Lennard-Jones potential

can keep bundles of hair together, and repulsing forces at further distances can give volume to the hair. In the work on continuum simulation of hair by McAdams et al. (2009), hair is immersed in a fluid (it is indeed immersed in air in the real world) and simulating this fluid allows for complex contact and collision effects to appear. An interesting extension would be to use both particle-based fluids and hair, and keep the simulation entirely in the Lagrangian domain.

Due to the absence of hair-hair interaction, each hair strand is treated independently, so parallelism is not difficult. We were able to simulate three heads full of hair, animated simultaneously in real-time. Introducing hair-hair interaction makes the problem more difficult, but one way to address it is as follows: A 2D grid structure can be defined at the surface of the head. For sufficiently short hair, each hair strand only interacts with other hair strands in its vicinity, so the position of the root of each hair can define its cell in the grid. The size of the cells can be tuned to be a function of the length of the hair. When processing each cell, only the data on the neighboring cells need be accessed. Parallelism and efficiency can be increased. As a future avenue of research, a customized GPU architecture can also be devised, in which each thread block has access to the local memory of its 8 neighbors, and we can benefit from it to accelerate hair and other simulations.

Bibliography

ACR and NEMA (1985). ACR-NEMA digital imaging and communications standard. In *NEMA Standards Publication*, volume 300. National Electrical Manufacturers Association, Washington, DC, USA. 18

Adamson, B., Bormann, C., Handley, M., and Macker, J. (2009). NACK-oriented reliable multicast (NORM) transport protocol. RFC 5740. 58

Bares, W. H. and Lester, J. C. (1997). Cinematographic user models for automated realtime camera control in dynamic 3D environments. In *Proceedings of the Sixth International Conference on User Modeling*, pages 215–226. Springer. 16

Becker, M. and Teschner, M. (2007). Weakly compressible SPH for free surface flows. In *Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '07, pages 209–217, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 10

Blinn, J. F. (1982). A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256. 50

Chen, Y. and Medioni, G. (1992). Object modelling by registration of multiple range images. *Image Vision Comput.*, 10(3):145–155. 13

Christianson, D. B., Anderson, S. E., He, L.-W., Salesin, D. H., Weld, D. S., and Cohen, M. F. (1996). Declarative camera control for automatic cinematography. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence, Volume 1*, AAAI '96, pages 148–155. AAAI Press. 16

Deering, S. (1989). Host Extensions for IP Multicasting. RFC 1112. 58

Desbrun, M. and Gascuel, M.-P. (1996). Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proceedings of the Eurographics Workshop on Computer*

Animation and Simulation '96, pages 61–76, New York, NY, USA. Springer-Verlag New York, Inc. 9

Dymond, J. H. and Malhotra, R. (1988). The Tait equation: 100 years on. *International Journal of Thermophysics*, 9(6):941–951. 11

Enright, D., Marschner, S., and Fedkiw, R. (2002). Animation and rendering of complex water surfaces. *ACM Transactions on Graphics*, 21(3):736–744. 11

Fava, A., Fava, E., and Bertozzi, M. (1999). MPIPOV: A parallel implementation of POV-Ray based on MPI. In *in: Proc. Euro PVM/MPI'99, Lecture Notes in Computer Science*, pages 426–433. Springer-Verlag. 50

Friedman, S. (1994). Large scale urban visualization. Technical Report, UCLA Department of Architecture. 16

Gingold, R. A. and Monaghan, J. J. (1977). Smoothed particle hydrodynamics – Theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 181:375–389. 6

Goswami, P., Schlegel, P., Solenthaler, B., and Pajarola, R. (2010). Interactive SPH simulation and rendering on the GPU. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 55–64, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 12, 67

Guo, J., Yang, C., Han, J., Tang, J., Zheng, M., Liao, X., and Yuan, Z. (2013). Particle-based cardiac rhythm simulation. In *25th Chinese Control and Decision Conference (CCDC)*, pages 2072–2075. 6

He, L.-W., Cohen, M. F., and Salesin, D. H. (1996). The virtual cinematographer: A paradigm for automatic real-time camera control and directing. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 217–224, New York, NY, USA. ACM. 16

Hecht, F., Lee, Y. J., Shewchuk, J. R., and O'Brien, J. F. (2012). Updated sparse Cholesky factors for corotational elastodynamics. *ACM Transactions on Graphics*, 31(5):X:1–13. Presented at SIGGRAPH 2012. 43

Hennessy, J. L. and Patterson, D. A. (2006). *Computer Architecture: A Quantitative Approach*. Elsevier, fourth edition. 2

Hong, T. M., Magnenat-Thalmann, N., and Thalmann, D. (1988). A general algorithm for 3-D shape interpolation in a facet-based representation. In *Proceedings of Graphics Interface '88*, GI '88, pages 229–235, Toronto, Ontario, Canada. Canadian Man-Computer Communications Society. 14

Ihmsen, M., Akinci, N., Becker, M., and Teschner, M. (2011). A parallel SPH implementation on multi-core CPUs. *Computer Graphics Forum*, 30(1):99–112. 67

Ihmsen, M., Orthmann, J., Solenthaler, B., Kolb, A., and Teschner, M. (2014). SPH fluids in computer graphics. In Lefebvre, S. and Spagnuolo, M., editors, *Eurographics 2014 – State of the Art Reports*. The Eurographics Association. 12

Jackins, C. L. and Tanimoto, S. L. (1980). Oct-trees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(3):249–270. 52

Jepson, W., Liggett, R., and Friedman, S. (1995). An environment for real-time urban simulation. In *Proceedings of the Symposium on Interactive 3D Graphics*, I3D '95, pages 165–ff., New York, NY, USA. ACM. 16, 70

Jepson, W., Liggett, R., and Friedman, S. (1996). Virtual modeling of urban environments. *Presence: Teleoperators and Virtual Environments*, pages 72–86. 16

Kanai, T., Suzuki, H., and Kimura, F. (1997). 3D geometric metamorphosis based on harmonic map. In *Computer Graphics and Applications, 1997. Proceedings., The Fifth Pacific Conference on*, pages 97–104. 14

Kass, M., Witkin, A., and Terzopoulos, D. (1988). Snakes: Active contour models. *International Journal of Computer Vision*, 1(4):321–331. 15, 89, 93

Keall, P. (2004). 4-dimensional computed tomography imaging and treatment planning. *Seminars in Radiation Oncology*, 14(1):81–90. High-Precision Radiation Therapy of Moving Targets. 13

Kitware, Inc. (2015). The visualization toolkit. http://www.vtk.org/. Accessed 2015-11-27. 18

Kovács, S. J., Mcqueen, D. M., and Peskin, C. S. (2001). Modelling cardiac fluid dynamics and diastolic function. *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, 359(1783):1299–1314. 13

Krog, O. E. and Elster, A. C. (2012). Fast GPU-based fluid simulations using SPH. In Jónasson, K., editor, *Applied Parallel and Scientific Computing: 10th International Conference, PARA 2010, Reykjavík, Iceland, June 6-9, 2010, Revised Selected Papers, Part II*, pages 98–109. Springer Berlin Heidelberg, Berlin, Heidelberg. 12

Lal, P. S., Unnikrishnan, A., and Jacob, K. P. (1998). Parallel implementation of octtree generation algorithm. In *Image Processing, 1998. ICIP 98. Proceedings. 1998 International Conference on*, pages 1005–1009 vol.3. 52

Lazarus, F. and Verroust, A. (1998). Three-dimensional metamorphosis: A survey. *The Visual Computer*, 14(8):373–389. 14

Lee, A. W. F., Dobkin, D., Sweldens, W., and Schröder, P. (1999). Multiresolution mesh morphing. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 343–350, New York, NY, USA. ACM Press/Addison-Wesley Publishing Co. 14

Leng, S., Tang, J., Zambelli, J., Nett, B., Tolakanahalli, R., and Chen, G.-H. (2008). High temporal resolution and streak-free four-dimensional cone-beam computed tomography. *Physics in Medicine and Biology*, 53(20):5653–5673. 13

Liggett, R., Friedman, S., and Jepson, W. (1995). Interactive design/decision making in a virtual urban world: Visual simulation and GIS. In *15th Annual ESRI User Conference*, pages 22–26. 16

Lindstrom, P. (2014). Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics*, 20(12):2674–2683. 64

Lorensen, W. E. and Cline, H. E. (1987). Marching cubes: A high resolution 3D surface construction algorithm. In *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '87, pages 163–169, New York, NY, USA. ACM. 19

Lucy, L. B. (1977). A numerical approach to the testing of the fission hypothesis. *Astronomical Journal*, 82:1013–1024. 6

McAdams, A., Selle, A., Ward, K., Sifakis, E., and Teran, J. (2009). Detail preserving continuum simulation of straight hair. In *ACM SIGGRAPH 2009*, SIGGRAPH '09, pages 62:1–62:6, New York, NY, USA. ACM. 96

McAdams, A., Sifakis, E., and Teran, J. (2010). A parallel multigrid Poisson solver for fluids simulation on large grids. In *Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '10, pages 65–74, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 6, 43

McInerney, T. and Terzopoulos, D. (1997). Medical image segmentation using topologically adaptable surfaces. In *CVRMed-MRCAS'97: First Joint Conference Computer Vision, Virtual Reality and Robotics in Medicine and Medical Robotics and Computer-Assisted Surgery, Grenoble, France, March 19–22, 1997 Proceedings*, pages 23–32. Springer Berlin Heidelberg, Berlin, Heidelberg. 15

Morton, G. M. (1966). A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottawa, Ontario, Canada. 67

Müller, M., Charypar, D., and Gross, M. (2003). Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 7, 9, 10, 34, 41

Nie, X., Chen, L., and Xiang, T. (2015). Real-time incompressible fluid simulation on the GPU. *Int. J. Computer Games Technology*, 2015:2:2–2:2. 12

Nishimura, H., Hirai, M., Kawai, T., Kawata, T., Shirkawa, I., and Omura, K. (1985). Object modeling by distribution function and a method of image generation. *Electronics Communication Conference*, pages 718–725. 50

Persistence of Vision Pty. Ltd. (2004). Persistence of vision raytracer. http://www.povray.org/. 50

Peskin, C. and McQueen, D. (1996). Fluid dynamics of the heart and its valves. In Othmer, H., Adler, F., Lewis, M., and Dallon, J., editors, *Case Studies in Mathematical Modeling*, pages 309–337. Prentice Hall. 13

Peskin, C. S. (1972). Flow patterns around heart valves: A numerical method. *Journal of Computational Physics*, 10(2):252–271. 12

Peskin, C. S. (1977). Numerical analysis of blood flow in the heart. *Journal of Computational Physics*, 25(3):220–252. 12

Peskin, C. S. and McQueen, D. M. (1989). A three-dimensional computational method for blood flow in the heart I: Immersed elastic fibers in a viscous incompressible fluid. *Journal of Computational Physics*, 81(2):372–405. 12

Pham, T. and Van Vliet, L. (2005). Separable bilateral filtering for fast video preprocessing. In *Multimedia and Expo, 2005. ICME 2005. IEEE International Conference on*, pages 4–pp. IEEE. 49

Pixmeo SARL (2015). OsiriX sample datasets. http://www.osirix-viewer.com/datasets/. Accessed 2015-11-27. 18

Poyart, E. and Faloutsos, P. (2010). Real-time hair simulation with segment-based head collision. In Boulic, R., Chrysanthou, Y., and Komura, T., editors, *Motion in Games*, volume 6459 of *Lecture Notes in Computer Science*, pages 386–397. Springer Berlin / Heidelberg. 94

Poyart, E., Snyder, L., Friedman, S., and Faloutsos, P. (2011). VSim: Real-time visualization of 3D digital humanities content for education and collaboration. In *Proceedings of the 12th International Conference on Virtual Reality, Archaeology and Cultural Heritage*, VAST'11, pages 129–135, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association. 68

Ratnasamy, S., Ermolinskiy, A., and Shenker, S. (2006). Revisiting IP multicast. In *Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '06, pages 15–26, New York, NY, USA. ACM. 58

Rusinkiewicz, S. and Levoy, M. (2001). Efficient variants of the ICP algorithm. In *3-D Digital Imaging and Modeling*, pages 145–152. IEEE. 14

Schechter, H. and Bridson, R. (2012). Ghost SPH for animating water. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2012)*, 31(4). 41

Schroeder, W., Martin, K., and Lorensen, B. (2006). *The visualization toolkit: An object-oriented approach to 3D graphics*. Kitware. 18

Schroeder, W. J., Zarge, J. A., and Lorensen, W. E. (1992). Decimation of triangle meshes. In *Computer Graphics (SIGGRAPH '92 Proceedings*, pages 65–70. 22

Smith, R. (2000). The Open Dynamics Engine. http://www.ode.org/. Accessed 2016-08-21. 95

Solenthaler, B. and Pajarola, R. (2009). Predictive-corrective incompressible SPH. In *ACM SIGGRAPH 2009*, SIGGRAPH '09, pages 40:1–40:6, New York, NY, USA. ACM. 11

Speakman, T., Crowcroft, J., Gemmell, J., Farinacci, D., Lin, S., Leshchiner, D., Luby, M., Montgomery, T., Rizzo, L., Tweedly, A., Bhaskar, N., Edmonstone, R., Sumanasekera, R., and Vicisano, L. (2001). PGM reliable transport protocol specification. RFC 3208. 58

Spilker, C. (1989). The ACR-NEMA digital imaging and communications standard: A nontechnical description. *Journal of Digital Imaging*, 2(3):127–131. 18

Staten, M. L., Owen, S. J., Shontz, S. M., Salinger, A. G., and Coffey, T. S. (2012). A comparison of mesh morphing methods for 3D shape optimization. In Quadros, W. R., editor, *Proceedings of the 20th International Meshing Roundtable*, pages 293–311, Berlin, Heidelberg. Springer Berlin Heidelberg. 15

Tu, X. and Terzopoulos, D. (1994). Artificial fishes: Physics, locomotion, perception, behavior. In *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '94, pages 43–50, New York, NY, USA. ACM. 43

Turner, R., Balaguer, F., Gobbetti, E., and Thalmann, D. (1991). Physically-based interactive camera motion control using 3D input devices. In *Scientific Visualization of Physical Phenomena (Proceedings of CG International 91*, pages 135–145. Springer. 16

van der Laan, W. J., Green, S., and Sainz, M. (2009). Screen space fluid rendering with curvature flow. In *Proceedings of the 2009 Symposium on Interactive 3D graphics and games*, I3D '09, pages 91–98, New York, NY, USA. ACM. 50

Vedam, S. S., Keall, P. J., Kini, V. R., Mostafavi, H., Shukla, H. P., and Mohan, R. (2003). Acquiring a four-dimensional computed tomography dataset using an external respiratory signal. *Physics in Medicine and Biology*, 48(1):45. 13

Ware, C. and Osborne, S. (1990). Exploration and virtual camera control in virtual three dimensional environments. In *Proceedings of the Symposium on Interactive 3D Graphics*, I3D '90, pages 175–183, New York, NY, USA. ACM. 70

Weaver, T. and Xiao, Z. (2016). Fluid simulation by the smoothed particle hydrodynamics method: A survey. In *11th International Conference on Computer Graphics Theory and Application (GRAPP 2016)*, Rome, Italy. 12

Zeleznik, R. and Forsberg, A. (1999). UniCam – 2D gestural camera controls for 3D environments. In *Proceedings of the Symposium on Interactive 3D Graphics*, I3D '99, pages 169–173, New York, NY, USA. ACM. 16