Interactive Simulation of Contrast Fluid using Smoothed Particle Hydrodynamics

Andreas Grahn

January 1, 2008
Master's Thesis in Computing Science, 30 credits
Supervisor at CS-UmU: Kenneth Bodin
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

In a collaboration project between Umeå University and Stanford University a medical training simulator examining the gastro-intestinal tract is being developed. A real-time simulation of a viscous fluid known as contrast fluid is needed as well as the interaction between the contrast fluid and the colon wall.

In this thesis we examine existing methods for simulating fluids and deformable materials. We also implement a prototype and evaluate the possibilities of such a medical simulator. The prototype implements a variant of smoothed particle hydrodynamics together with a triangle mesh as a boundary condition. The solution simulates viscous fluid inside a complex triangle mesh and performs surface extraction based on screen space meshes. A illumination model is implemented on the GPU to perform shading effects such as fresnel reflection and refraction.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Department of Diagnostic Oral Radiology and VRlab at Umeå University have developed an educational virtual reality simulator that enables efficient training in oral radiology. The simulator has recently been commercialized in an off-spring company called Qbion AB.

In a collaboration project with Stanford University, this simulator paradigm is to be extended to other areas of diagnostic radiology, specifically examination of the gastrointestinal tract. In this project there is a need for visual and interactive simulation of contrast fluid interacting with the colon.

Chapter 2

Problem Description

Fluoroscopy is a medical imaging technique for obtaining real-time images of the internal structures of a patient through the use of a fluoroscope. A patient is placed between an x-ray source and a fluorescent screen. As x-rays pass through the patient, they are attenuated by varying amounts as they interact with different internal structures of the body, casting a shadow of the structures on the fluorescent screen. Fluoroscopy is used in investigations of the gastrointestinal tract where barium or contrast fluid often is used to intensify the contrasts in the x-ray image.

Contrast fluid is a thick, milkshake-like drink consisting of barium which is a dry, white, chalky, metallic powder mixed with water. The fluid is an x-ray absorber and appears white in an x-ray image. When swallowed, the barium coats the inside walls of the tract where the size and shape of the organs becomes visible in the x-ray image. Contrast fluid is used to diagnose structural or functional abnormalities such as cancers, tumors and polyps.

The objective is to develop a particle simulation method that simulates a viscous fluid with the colon as a boundary condition. The simulated system should also be visualized. The simulation method should be a variant of smoothed particle hydrodynamics (SPH). As a first approximation the colon should be considered as a fixed mesh or vortex field in which the fluid flows, and thereafter models for also simulating a deformable colon should be investigated.

First, a functional definition of the system is defined, and thereafter a literature study and decisions which methods to use. Algorithmic methods will be required for the collision detection (particle-particle and particle-boundary), for computing interactions and for rendering of the system. In addition methods might be required that optimize the data describing the boundary conditions. Thereafter a prototype will be implemented and evaluated against the functional description.

2.1 Goals

The purpose of this thesis is to investigate whether or not it is possible to implement a prototype that corresponds to the problem description. The functional goals for the

Figure 2.1: Fluoroscopic image of the gastrointestinal tract filled with contrast fluid.

prototype that should be fulfilled can be seen in Table 2.1.

Goal	Description
Interactive	The user should be able to interact with the simulation.
Viscous Fluid	Simulation of a viscous fluid.
Boundary Conditions	Fluid should be contained within a boundary.
Fluid Visualization	Visualization of the fluid surface.

Table 2.1: Functional goals for the prototype.

The boundary condition goal can be divided into two parts. First, a static colon should be implemented and evaluated. The second part is to investigate the possibility of having a deformable colon as a boundary condition for the fluid. The implementation of the second part is not required although it should be considered when designing the system and implementing the prototype.

Chapter 3

Computational Fluid Dynamics

Over the last decade, physics-based, yet interactive simulation and renderering of natural phenomena such as fluids has been an active research area in computer graphics. Fluids add substantially to the richness of a virtual world due to their ability to assume arbitrary shapes and to show complex behavior. A fluid is defined as a substance that cannot support shear stress in static equilibrium or more intuitively, a substance that flows because it cannot resist deformation. In order to keep fluids in place, they have to be kept in tanks or other containers. That is the reason why the interaction of fluids with solid boundaries plays a major role in fluid simulations and in this thesis.

3.1 History

Computational fluid dynamics (CFD) has a long history. In the 19:th century Claude Navier and George Stokes formulated the famous Navier-Stokes equations that describe the the dynamics of fluids. Besides the Navier-Stokes equation which describes conservation of momentum, two additional equations namely a continuity equation describing mass conservation and a state equation describing enery conservation are needed to simulate fluids. Since those equations are known and computers can solve them numerically, a large number of methods have been proposed in the area of CFD.

During the last decades, special purpose fluid simulation techniques have been developed in the field of computer graphics. Reeves [39] introduced particle systems as a technique for modeling fuzzy objects in 1983. Since then both the particle-based Lagrangian approach and the grid-based Eulerian approach have been used to simulate fluids. Desbrun and Cani [33] and Tonnesen [44] use particles to animate soft objects. Particles has also been used to animate surfaces, to control implicit surfaces and to animate lava flows. However, the Eulerian or grid-based approach is still more popular as for the simulation of fluids in general [40], water [37] [7] [23], soft objects [8] and melting effects [12].

3.2 Smoothed Particle Hydrodynamics

In astrophysical fluid dynamics problems frequently involve changes in spatial, temporal and density scales over many order of magnitude, thus adaptivity is an essential ingredient which is absent in an grid-based approach. An alternative to a grid-based approach is to represent fluid quantities by a set of moving interpolation points which follow the fluid motion. Since each point carries a fixed mass, the interpolation points are referred to as particles and the derivatives are evaluated by interpolation over neighbouring particles. Smoothed particle hydrodynamics (SPH) is a particle method (also known as a mesh-less method) introduced by Lucy [31], Gingold and Monaghan [38] to simulate the flow of interstellar gas. The advantage of SPH over grid-based approaches is that adaptivity is a built-in feature. The Lagrangian nature of the method means that changes in density and flow morphology are automatically accounted for. This also makes SPH very efficient in that computation is concentrated on regions of high density and not on empty regions. Another advantage is that no portions of the fluid can be lost from the simulation, unlike grid-based methods where fluid which has left the grid cannot return. One of the disadvantages of SPH is the additional computational cost of constructing the neighbour list for each particle. This is discussed in Section 7.2. Although SPH was developed for astrophysical problems, the method is general enough to be used in any kind of fluid simulation.

As mentioned earlier, SPH is an interpolation method for quantities defined at discrete particle locations. A scalar quantity A is interpolated at location $\mathbf{r}$ by a weighted sum of contributions from all particles:

$$A_s(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \tag{3.1}$$

The equation iterates over particles j, m_j is the mass of particle j, $\mathbf{r}_j$ its position, ρ_j the density and A_j the field quantity at $\mathbf{r}_j$. The function $W(\mathbf{r}, h)$ is called the smoothing kernel. The radius h is usually finite, which means that $W = 0$ when $|\mathbf{r} - \mathbf{r}_j| > h$. Different types of smoothing kernels are discussed in Section 3.5.

There exist SPH methods with varying and adaptive smoothing length and even fluid simulations with mixed fluids [17] where particles can have different mass and viscosity. However, this will not be discussed in this thesis. The kernel function should be both even and normalized satisfying the following conditions:

$$W(\mathbf{r}, h) = W(-\mathbf{r}, h) \tag{3.2}$$

$$\int W(\mathbf{r}, h) d\mathbf{r} = 1 \tag{3.3}$$

Through substitution in Equation 3.1 we get for density at location $\mathbf{r}$:

$$\rho_s(\mathbf{r}) = \sum_j m_j \frac{\rho_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h) \tag{3.4}$$

Derivatives are often used in fluid equations. These are easily evaluated when using SPH. The gradient of A is simply:

$$\nabla A_s(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h) \tag{3.5}$$

The laplacian of A becomes:

$$\nabla^2 A_s(\mathbf{r}) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r} - \mathbf{r}_j, h) \tag{3.6}$$

3.3 The Euler Form of the Navier-Stokes Equation

Isothermal fluids are described by a velocity field v, a density field ρ and a pressure field p. The evolution of these quantities over time is given by two equations, Equation 3.7 (also called the continuity equation) assures the conservation of mass and Equation 3.8 the conservation of momentum.

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{3.7}$$

$$\rho(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v}\nabla\mathbf{v}) = -\nabla p + F + \mu\nabla^2\mathbf{v} \tag{3.8}$$

Equation 3.8 is the most general form of the Navier-Stokes equation for incompressible fluids where F is an external force field and μ the viscosity of the fluid.

3.4 Modeling Fluids with Particles

By using a particle method for simulating fluids both Equation 3.7 and 3.8 are simplified further. Because the number of particles are constant and each particle has a constant mass, mass conservation is automatically guaranteed. The divergence of the velocity field is zero if we ignore temperature effects for the incompressible viscous fluid. We can therefore exclude Equation 3.7. However, we need to introduce a new equation 3.13 that preserves the fluid volume. Since the particles move with the fluid, the substantial derivative of the velocity field is simply the time derivative of the velocity. This means that we can remove the term $\mathbf{v}\nabla\mathbf{v}$ from the left hand side of Equation 3.8 and rewrite it as:

$$\frac{\partial \mathbf{v}}{\partial t} = -\frac{\nabla p}{\rho} + \frac{\mathbf{F}}{\rho} + \frac{\mu\nabla^2\mathbf{v}}{\rho} \tag{3.9}$$

Where $-\nabla p/\rho$ is modelling pressure, $\mu\nabla^2\mathbf{v}/\rho$ is modelling viscosity and $\mathbf{F}/\rho$ is external forces. The acceleration of a particle can thus be descibed as:

$$\mathbf{a}_i = \frac{d\mathbf{v}_i}{dt} = \frac{\mathbf{F}_i}{\rho_i} \tag{3.10}$$

We can then describe each term in Equation 3.9 by applying the SPH rule. The pressure term $-\nabla p/\rho$ yields:

$$f_i^{pressure} = -\sum_j m_i m_j \frac{p_j}{\rho_i \rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{3.11}$$

However, this force term is not symmetrical as can be seen when only two particles interact. As Newton's third law states, to every action there is an equal and opposite reaction. Different ways of making Equation 3.11 symmetrical has been proposed in literature but we will use the following pressure force:

$$f_i^{pressure} = -\sum_j m_i m_j \left(\frac{p_i}{\rho_i^2} + \frac{p_j}{\rho_j^2} \right) \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{3.12}$$

Before we can compute the pressure force we need to evaluate the pressure field. This can be done by using a modified version of the ideal gas state equation suggested by Debrun [33]:

$$p = k(\rho - \rho_0) \tag{3.13}$$

where ρ_0 is the rest density and k is determined by the speed of sound in the material. The offset in Equation 3.13 influences the gradient of a field smoothed by SPH and makes the simulation numerically more stable. We continue to the viscosity term $\mu \nabla^2 \mathbf{v}/\rho$ and proceed with similar fashion as for the pressure term. This also yields an asymmetric force:

$$f_i^{viscosity} = \mu \sum_j m_i m_j \left(\frac{\mathbf{v}}{\rho_i \rho_j} \right) \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{3.14}$$

and to balance the force Müller [34] suggest using the following viscosity force:

$$f_i^{viscosity} = \mu \sum_j m_i m_j \left(\frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_i \rho_j} \right) \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{3.15}$$

Equation 3.14 was simply symmetrized in a natural way by using velocity differences. The viscosity forces are only dependent on velocity differences and not on absolute velocities. Equation 3.15 describes how particle i is accelerated in the direction of the relative speed of it's neighbouring particles.

Although the Navier-Stokes equation mentioned earlier can model fluid dynamics, its still a simplified version which does not include surface tension. Surface tension can be describes as a force that attracts fluid molecules towards each other. Inside the fluid these intermolecular forces are equal in all direction and balance each other out. On the fluid surface, the tension force act in the direction of the surface normal and tends to minimize the curvature of the fluid surface. In this thesis we model surface tension forces based on ideas of Morris [36] but first we need to find the fluid surface. This is done by using an additional field quantity which is 1 at particle locations and 0 everywhere else. This is called color field in literature and for the smoothed color field we get:

$$c_s(\mathbf{r}) = \sum_j m_j \frac{1}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h) \tag{3.16}$$

The gradient of this color field yields the surface normal of the fluid pointing into the fluid.

$$\mathbf{n} = \nabla c_s \tag{3.17}$$

The curvature of the fluid surface can then be described with:

$$\kappa = \frac{-\nabla^2 c_s}{|\mathbf{n}|} \tag{3.18}$$

When putting it all together we get:

$$f_i^{surface} = \frac{m_i}{\rho_i} \sigma \kappa \mathbf{n} = -\frac{m_i}{\rho_i} \sigma \nabla^2 c_s \frac{\mathbf{n}}{|\mathbf{n}|} \tag{3.19}$$

where σ is the surface tension coefficient. The sum of all computed forces becomes the total force acting on particle i:

$$f_i^{total} = f_i^{pressure} + f_i^{viscosity} + f_i^{surface} + f_i^{external} \tag{3.20}$$

3.5 Smoothing Kernel

To achieve stability, accuracy and speed of the SPH, the kernel choice is important. The smoothing kernel should satisfy the requirements mentioned earlier in Section 3.2 that it should be both even and normalized. With this in mind Müller [34] designed the following kernel:

$$W_{poly6}(\mathbf{r}, h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & otherwise \end{cases} \tag{3.21}$$

which is of second order interpolation error. In addition, the kernels that are zero with vanishing derivatives at the boundary tends to bring stability. This kernel is also computationally pleasent since r is kept squared and the distance computation can be evaluated without computing the squared root which is an computational expensive task for the CPU.

If one were to use W_{poly6} in the pressure computations, particles would group up and build clusters under high pressure. This is because the gradient of the kernel approaches zero at the center and causes the repulsion force between two interacting particles to become zero which is not wanted. Instead Desbrun [33] designed a spiky kernel for pressure computations with a non-vanishing gradient near the center that generates the wanted repulsion forces.

$$W_{spiky}(\mathbf{r},h) = \frac{15}{\pi h^6}\begin{cases}(h-r)^3 & 0 \le r \le h \\ 0 & otherwise\end{cases} \tag{3.22}$$

$$\nabla W_{spiky}(\mathbf{r},h) = \frac{45}{\pi h^6}\begin{cases}\left(\frac{h^2+r^2}{r} - 2h\right)\mathbf{r} & 0 \le r \le h \\ 0 & otherwise\end{cases} \tag{3.23}$$

Viscosity is a measure of the resistance of a fluid to deform under shear stress. This phenomena causes the fluid's kinetic energy to turn into heat due to friction and reduces the relative velocity between two particles. Using either W_{poly6} kernel or W_{spiky} kernel for viscosity computation would cause an increase in velocities between two particles. This is because the laplacian of both kernels can become negative and result in forces that increase their relative velocity. Müller [34] suggest using the following kernel for viscosity compuation:

$$W_{viscosity}(\mathbf{r},h) = \frac{15}{2\pi h^3}\begin{cases}-\frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \le r \le h \\ 0 & otherwise\end{cases} \tag{3.24}$$

whose laplacian is positive everywhere:

$$\nabla^2 W_{viscosity}(\mathbf{r},h) = \frac{45}{\pi h^6}\begin{cases}(h-r) & 0 \le r \le h \\ 0 & otherwise\end{cases} \tag{3.25}$$

Chapter 4

Collision Detection

In this section we'll discuss the advantages and drawbacks of different techniques to detect collisions between objects in a physical simulation. In physics-based simulation, collision detection is often the bottle-neck, since a collision query needs to be perfomed every time step to determine contacting and colliding objects. Since there can exist a number of objects and each object is capable of producing a collision with any other object, collision detection is clearly of $O(n^2)$ complexity. Hubbard [26] introduced the concepts of broad-phase and narrow-phase which reduces the computational load by first performing a coarse test (broad-phase) to prune unnecessary pair test and then perform pairwise testing (narrow-phase) on the remaining set of objects.

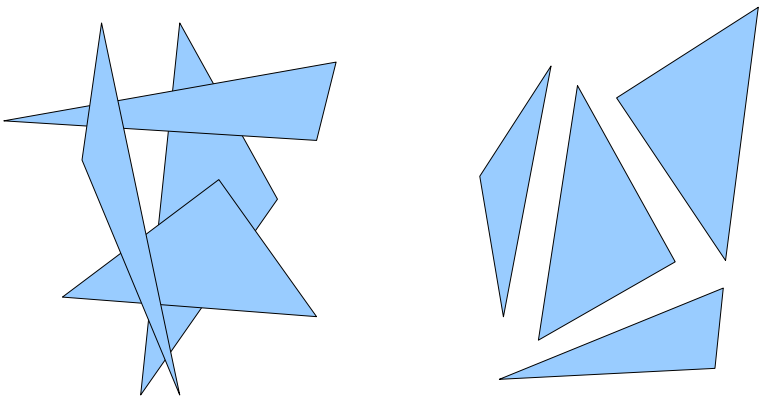

Figure 4.1: The amount of geometric coherence in a set of triangles. On the left, little coherence and on the right, much coherence.

We will now discuss different data structures that can be used to prune pairwise primitive test and accelerate collision detection in complex environments. An important concept in this context is geometric coherence which can be described as the amount of separability of the set of objects. Figure 4.1 shows two sets of triangles, the left with little geometric coherence and the right with much geometric coherence. The geometric coherence is important because it enables use to quickly reject pairs of objects based on the region of space they occupy. Spatial data structures such as regular grids and bounding volumes hierarchies can be used to capture the geometric coherence. These

data structures can be divided into two categories, namely space partitioning and model partitioning.

4.1 Space Partitioning

In this section we describe the most commonly used space partitioning methods in the interactive simulation community. For each method we first explain the method and then briefly evaluate the method from a programmer's point of view.

4.1.1 Regular Grid

A regular grid (or uniform grid or voxel grid in some literature) divides the space into a number of uniform rectangular cells. A regular grid is represented by an axis-aligned bounding box (AABB) enclosing all objects and a three-dimensional array of cells of size $N_x \times N_y \times N_z$, where N_i is the number of cells along each axis. Given a point $\mathbf{p} = [x, y, z]^T$ inside the AABB the indices of the cell containing the point are

$$i_i = (\mathbf{p}_i - \beta_i)\frac{N_i}{\gamma_i} \quad , i = x, y, z \tag{4.1}$$

where β is the base vertex of the AABB and γ the extent along each axis. Each cell contains a list of objects that intersects that cell. When adding, moving or removing an object we'll need to update the list in all cells that the objects covers. This is similar to scan conversion in computer graphics. A more simple approach is to update the cells which are covered by the object's AABB.

The benefits of using a regular grid is its simplicity and fast cell access. The storage can however become huge if the resolution of the grid is high. The best case scenario for the regular grids technique is when the objects are of similar size, evenly distributed and the number of cells is linear with the number of objects. Then, adding and moving an object can be done in constant time. However, the biggest drawback, which is common to all bucketing techniques, is that the performance is greatly dependent of the density of objects in the space being uniform. If the objects are clustered together, many objects will occupy the same cell which results in a lot of intersection testing. The hard task is to choose the best resolution for the grid, too few cells and the cells get very crowded and thus a lot of intersection tests will be performed among objects in the same cell. Too many cells results in a greater storage consumption and many cells will contain reference to the same object. Regular grids work best with geometrically coherent sets of objects of uniform density and size but few models satisfy these conditions, hence adaptive partitioning schemes, using recursive space partitioning, often yield a better result.

4.1.2 Spatial Hashed Grid

Instead of indexing directly into a large matrix like regular grids we can hash the coordinates into a hash key used to index an array of buckets. Cells intersecting with objects are kept in the hash table and this makes it more efficient to traverse the cells, since only cells of interest are kept in the hash table. A set of N buckets are stored as an array and each bucket contains a list of potential intersecting objects. The bucket

index of a point $\mathbf{p}$ in 3D-space with coordinates $[x, y, z]^T$ can be found by performing the following two steps:

1 Discretisize coordinates.
$[i, j, k] = [p_x/l, p_y/l, p_z/l]$, where l is the grid cell size.

2 Compute bucket index.
$i_{bucket} = hash(i, j, k) \mod N$, where $hash(i, j, k) = i \cdot p_0 + j \cdot p_1 + k \cdot p_2$ and each p_i is a large prime number.

The size of the hash table influences the performance of the algorithm. Large hash tables reduces the risk of mapping different 3D coordinates to the same bucket therefore it is often wise to start with a large hash table size and experiment with the size until satisfied. But too large size introduces a slight decrease in performance due to larger memory consumption. In Section 7.2 and 7.3 we apply this technique for particle neighbour search and collision detection between particles and a triangle mesh.

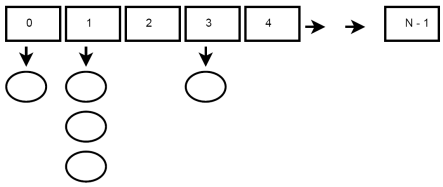

Figure 4.2: A list of buckets containing references to objects.

4.1.3 Distance Field

Consider an implicit function describing a convex shape of a sphere. Given a point $\mathbf{p} = [x, y, z]^T$ in 3D-space, we can determine if the point is inside the sphere, outside or on the surface of the sphere by examining the calculated value of $f(\mathbf{p})$.

$$f(\mathbf{p}) = x^2 + y^2 + z^2 - 1 \tag{4.2}$$

If $f(\mathbf{p}) < 0$ then the point is inside the sphere. If $f(\mathbf{p}) > 0$ then the point is outside the sphere and if $f(\mathbf{p}) = 0$ then the point is on the sphere surface. This is beneficial since it gives us a way to determine a collision in constant time. A distance field $\theta(..)$ works in a similar way, given a arbitrary point $\mathbf{p} = [x, y, z]^T$ the value $\theta(p)$ gives the signed distance to the closest point on the surface of an object. If $\mathbf{p}$ is outside the object, the value is positive. If $\mathbf{p}$ is inside the object, the value is negative and if the value is zero then the point is on the surface. The contact normal can be found by using the gradient at the given point. However, not all complex models can be described with such an implicit function. Instead the object is sampled into a 3D grid where each node in the grid stores the signed distance to the object and the face normal. The computation of a distance field can be done in many ways and is often computed before entering the simulation loop.

The surface normal is an important tool for determining whether a given point is inside the object or not. This is done by finding a point on the surface and taking the inner product of the surface normal and the point. However, triangle meshes are not a

smooth surface and does not have normals defined everywhere on the surface (vertices and edges). In [28] they use angle weighted psuedo-normals to capture these properties and compute a smooth surface. Distance fields is an algorithm known to be well suited for collision queries between a particle system and a complex object since each collision query is of $O(1)$ complexity. It is not suited for deformable objects with dynamically changing topology since the 3D grid needs to be updated in each time step. Another drawback when using distance fields is the large memory footprint.

4.1.4 Recursive Space Partitioning

Recursive or hierarchical space partitioning is done by recursively subdivide cells into sub-cells. The benefit of using this approach is the fact that it is adaptive to local densities of objects. Regions in space where objects are clustered the space is partitioned into several smaller cells, while in regions where there are hardly any objects the cells are kept large. The most common methods mentioned in literature are octree and k-d tree. Binary space partition (BSP) is another method which is the most general of all recursive space partitioning structures. Both the octree and k-d tree structure is represented as a hierarchical tree where each node in the tree corresponds to a region in space. The root node contains the whole space, while internal nodes represents a subdivided region where each smaller region corresponds to the nodes children. The leaf nodes contains the objects. The difference between the two is the way they split a node. Each node in the octree approach is split along the three coordinate axes into eight octants. This is illustrated in Figure 4.3. An internal node in a k-d tree splits its corresponding region into two regions along an arbitrary coordinate axis.

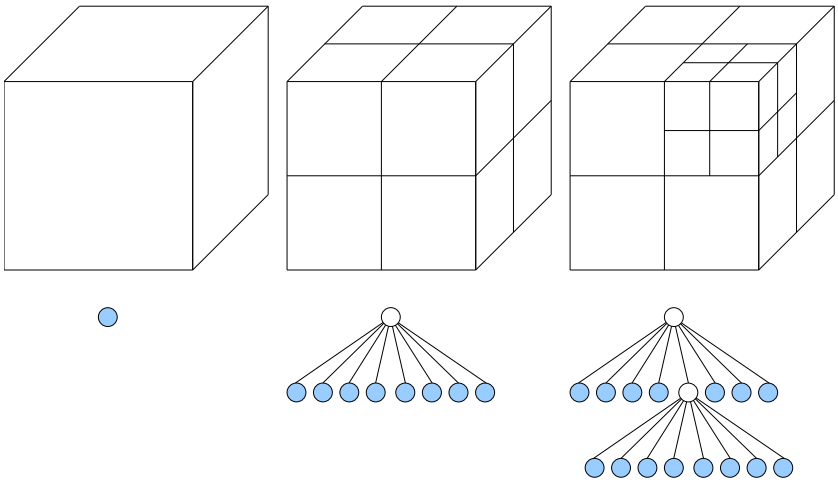

Figure 4.3: Octree structure for hierarchical partitioning of space into rectangular cells.

As mentioned earlier the most general technique in hierarchical space partitioning is BSP trees. In a BSP tree, a node can split its convex region into two regions separated by a freely oriented partitioning plane. Rephrased, a BSP tree can be seen as a variable split k-d tree but without the restriction that the partitioning planes are orthogonal to the coordinate axes. The importance of choosing the best partitioning plane is vital to reduce the size of the tree as well as the performance of the queries performed on the

tree. Usually the best split to achieve a good average-case query time is done by choosing a partitioning plane that splits the convex region into two regions of equal volume.

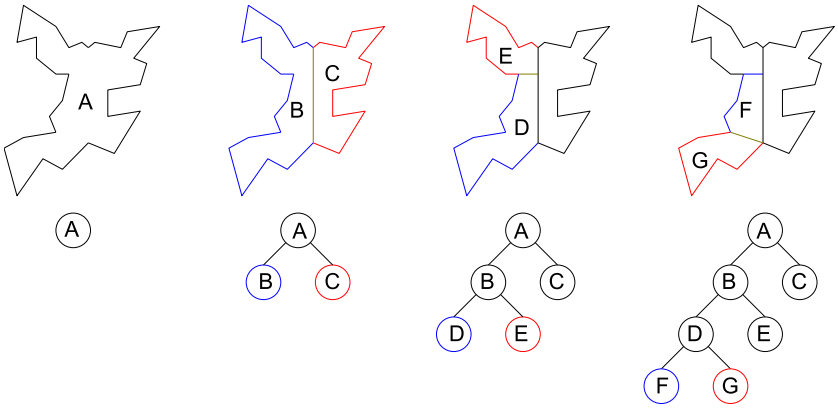

Figure 4.4: 2D example of constructing a BSP tree.

A drawback of space partitioning methods is the fact that an object can be on the boundary between two different cells causing the structure size to become greater than the number of objects. This can also cause multiple references to the same object which results in an increase of intersection tests. A solution to this would be to store objects in a search structure where each object is referred to only once. In hierarchical space partitioning structures this can be achieved by allowing objects to be stored in all nodes, not only in the leaf nodes. Each object is maintained in the node which corresponds to the smallest enclosing region of the object. However, intersection testing can become very complex since intersecting objects may be maintained in different levels in the tree.

4.2 Model Partitioning

Model partitioning is often a better choice than space partitioning and we can avoid the drawbacks such as multiple references to the same object. The basic principle of model partitioning is to divide a set of objects into geometrically coherent subsets. Each of these subsets are enclosed by a tight fitting bounding volume. Objects can quickly be rejected if their bounding volumes do not overlap.

A bounding volume can be any primitive shape, often represented by spheres or axis-aligned bounding boxes (AABB) but more complex bounding volumes exist such as oriented bounding boxes (OBB) and discrete-orientation polytopes (k-DOPs). The choice of bounding volume is important, it should be fit as tight as possible, overlap tests between bounding volumes should be computationally cheap, consume small amount of storage and the cost of computing the bounding volume for a given model should be low.

The most widely used bounding volume is AABB and even though they take up more storage (6 floats) than spheres (4 floats), they are computationally less expensive when testing for intersections. In Table 4.1 the most commonly used bounding volumes are compared.

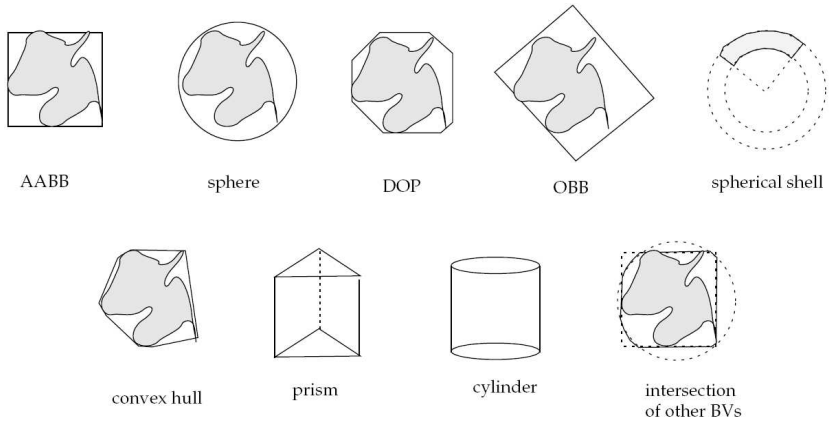

Figure 4.5: Different bounding volumes used in model partitioning.

Volume	Fit	Test (operations)	Storage (scalars)
Sphere	poor	11	4
AABB	poor	≤ 6	6
k-DOP	fair	$\leq 2k$	$2k$
OBB	good	≤ 200	15

Table 4.1: Bounding-volume comparison.

4.2.1 Bounding Volume Hierarchies

A bounding volume hierarchy (BVH) is a tree structure in which primitives are stored in the leaves. Each node maintains a bounding volume enclosing all the bounding volumes of the children nodes beneath it. There are basically two ways to construct a bounding volume hierarchy tree, top-down and bottom-up. There exist a third way, called incremental methods but it is not widely used in collision detection and will not be discussed. The top-down approach creates a bounding volume that is fitted and covers all primitives. This collection is then divided into two or more groups of primitives, each with a new fitted bounding volume. This continues until each group or node only contains a single reference to a primitive. Performing a bad split results in an unbalanced tree and a performance drop when traversing the tree for overlapping bounding volumes. Without clever search structures, the partitioning of n primitives takes $O(n)$ time. If fitting time also takes $O(n)$ time for n primitives, then the BVH building process has the same time relation as quicksort, namely $O(nlogn)$. Klosowski [10] mention picking a splitting plane orthogonal to any of the coordinate axes upon the following criteria:

Min Sum: Choose the axis that minimizes the sum of the volumes of the two resulting children.

Min Max: Choose the axis that minimizes the larger of the volumes of the two resulting children.

Longest Side: Choose the axis along which the bounding volume is longest.

In the bottom-up approach we first start by fitting bounding volumes for all primitives and then merge the leaf nodes and continue this process until the root is reached. After each merge a new bounding volume is created to fit the underlying bounding volumes. Given n primitives, then $n-1$ merges are required to build the BVH. As seen, the merge operation definitely decides the construction time of the BVH. Existing merging methods are far from producing the best BVH, usually a BVH is hand modelled by a 3D modeler. However, one can use mesh connectivity to guide the bottom-up construction. This is a dominant strategy when applying BVH to deformable objects and a performance increase is wanted [10].

An intersection test between two models, each represented by bounding volume hierarchy is done by recursively testing pairs of nodes. If the bounding volumes of the nodes do not overlap, then no collision has occurred. If both nodes are leaf nodes, then their primitives are tested for intersection. If one of the nodes are a leaf and the other one is an internal node then the node is tested for intersection with each of the other nodes children. If both nodes are internal nodes, then the node with a smaller bounding volume is tested against the larger bounding volume.

The benefits of using bounding volume hierarchies are the fast query times and the linear space requirements. If a model contains n triangles, we only keep one reference to each triangle and the tree will only need a $O(n)$ storage. The major drawback of bounding volume hierarchies is the computational cost of producing it and maintaining it during model changes and that is why they're generally used for complex rigid bodies.

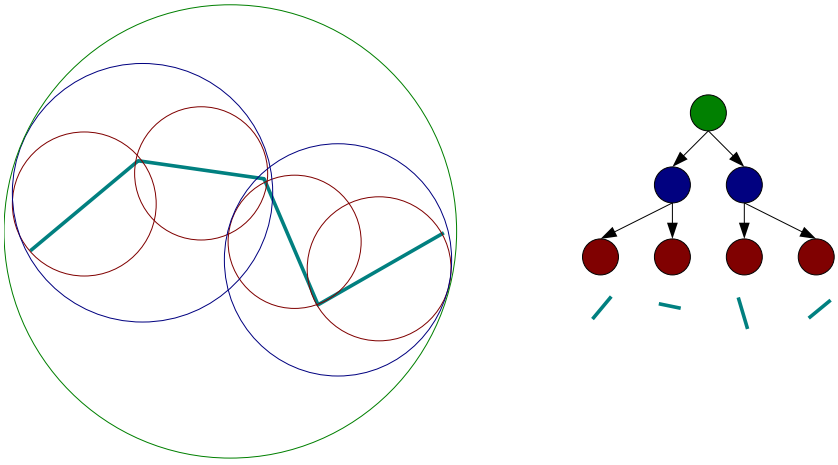

Figure 4.6: A BVH tree using sphere primitives is generated from a 2D shape. The red nodes store a reference to each line-segment and a computed bounding volume. Each of the blue nodes has a bounding volume encapsulating its children, in this case two red nodes. The bounding volume of the green root node covers the entire 2D shape.

4.3 Time is Continuous

In the real world, time is continuous but in the world of a computer simulation, time is discrete. The viewer might not notice this but if we would perform collision detection for these discretely sampled distances of time only, we might end up in detecting collision too late or not at all as seen in Figure 4.7. For instance, a bullet might pass a wall during a time step which would not result in a correct response. Using a higher sampling rate of time could help us detect collisions but this is not always an option in real-time simulations, since that increases computational load. Continuous collision detection (CCD) can solve this by performing intersection tests not only in object space but also in time space. This is done by extruding the 3-dimensional object in time and create a swept volume. However, performing CCD between two complex objects is not a trivial task and computationally too expensive for a real-time simulation with thousands of particles. CCD was considered but we came to conclude that it would become to expensive for this kind of simulation. Instead we have found that introducing a interaction radius to each particle increased the chances of detecting a collision. This resulted in no collision misses during the simulation.

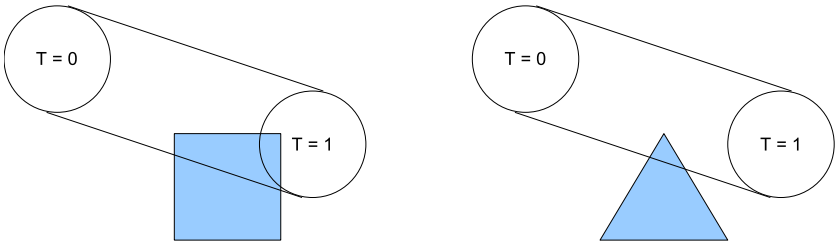

Figure 4.7: Problems when detecting collisions at discrete time steps. On the left we detect the collision too late and on the right we don't detect any collision at all.

4.4 Intersection Testing

In this section we discuss intersection tests between primitives. The intersection tests that we consider are particle and triangle, line and triangle and last sphere and triangle. Other trivial intersection tests such as AABB against another AABB and sphere against another sphere will not be discussed. The basic principle of all primitive intersection tests is that we try to reject the test as quick as possible. Another important aspect of intersection testing is that it should be computationally fast to perform the test, that is why the choice of algorithms is important.

4.4.1 Point and Triangle Test

A particle is a point in space with infinite small radius. The point does not have a volume and this makes it especially difficult to determine collision. This can be solved by storing two positions for the particle, the previous and the present. However, this now becomes another intersection test, namely line and triangle test which is discussed later.

Let say that, given a triangle t with normal $\mathbf{n}$ and a particle i with position $\mathbf{p}_i$ and velocity $\mathbf{v}_i$, we can then determine if the particle is on the correct side of the triangle by projecting the position onto the normal and look at the computed signed value.

$$s_i = \mathbf{n}(\mathbf{p}_i - \mathbf{t}_k) \tag{4.3}$$

where t_k is one of the triangles vertices. If $s_i = 0$ then the point is on the surface. If $s_i > 0$ then we don't have a collision. If $s_i < 0$ then we may have a collision and further checking is required to see if the point is inside the triangle area. We project the point $\mathbf{p}_i$ onto the triangle plane:

$$\mathbf{P}_i = \mathbf{p}_i - \mathbf{n}s_i \tag{4.4}$$

A point within the boundaries of a triangle can be described as:

$$\mathbf{p} = \lambda_0\mathbf{p}_0 + \lambda_1\mathbf{p}_1 + \lambda_2\mathbf{p}_2 \tag{4.5}$$

$$\lambda_0 + \lambda_1 + \lambda_2 = 1 \tag{4.6}$$

where λ_i are barycentric coordinates and $(\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2)$ are the vertices defining the triangle. We compute the barycentric coordinates for the projected point $\mathbf{P}_i$ and determine whether it's within the triangle boundaries. We can rewrite expression 4.5 to:

$$u(\mathbf{p}_1 - \mathbf{p}_0) + v(\mathbf{p}_2 - \mathbf{p}_0) = \mathbf{P} - \mathbf{p}_0 \tag{4.7}$$

$$u, v \geq 0 \tag{4.8}$$

$$u + v \leq 1 \tag{4.9}$$

Let say we place vertex $\mathbf{p}_0$ in origo and perform the following substitutions

$$\mathbf{b} = \mathbf{p}_1 - \mathbf{p}_0 \tag{4.10}$$

$$\mathbf{c} = \mathbf{p}_2 - \mathbf{p}_0 \tag{4.11}$$

$$\mathbf{p} = \mathbf{P} - \mathbf{p}_0 \tag{4.12}$$

and solving this for u gives us

$$ub_x + vc_x = p_x \tag{4.13}$$

$$v = \frac{p_x - ub_x}{c_x} \tag{4.14}$$

$$ub_y + \frac{p_x - ub_x}{c_x}c_y = p_y \tag{4.15}$$

$$ub_yc_x + p_xc_y - ub_xc_y = p_yc_x \tag{4.16}$$

$$ub_yc_x - ub_xc_y = p_yc_x - p_xc_y \tag{4.17}$$

$$u(b_yc_x - b_xc_y) = p_yc_x - p_xc_y \tag{4.18}$$

$$\tag{4.19}$$

$$u = \frac{p_yc_x - p_xc_y}{b_yc_x - b_xc_y} \tag{4.20}$$

$$v = \frac{p_yb_x - p_xb_y}{c_yb_x - c_xb_y} \tag{4.21}$$

We can now calculate u, v and check if $u, v \geq 0$ and $u + v \leq 1$. If the conditions are met we return the projected point $\mathbf{P}_i$ as the contact point together with the surface normal $\mathbf{n}$ and penetration depth s_i. However, there exist a problem using this method. If a complex triangle model is used, this method may not discover collisions in concave regions between triangles. The solution is discussed in the next section.

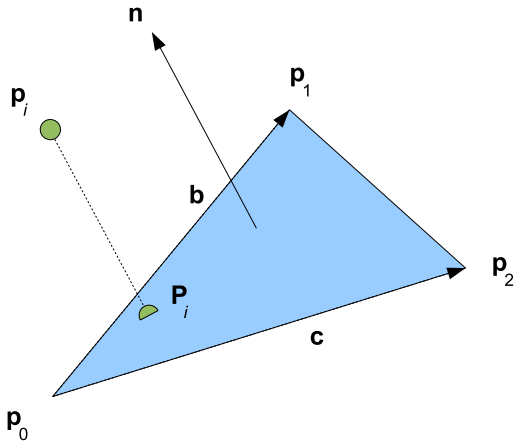

Figure 4.8: 3D example of a non-colliding point.

4.4.2 Line-Segment and Triangle Test

To avoid the drawbacks of the previous method we can introduce a previous and a present position and perform a line and triangle intersection test. By doing this we have created a simple continuous collision detection method since we are basically sampling a continuous curve through space and we will always detect if we intersect a static geometry. The idea for testing a line-segment against a triangle is very similar to the point-triangle method discussed earlier. We first find out if the line-segment intersects the triangle at all, if it doesn't we can exit early but if it does intersect we need to determine the intersection point. This intersection point on the triangle plane is then checked the same way as described in the previous section.

Let us assume that the triangle normal is given by:

$$\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)$$

where $\mathbf{p}_i$ for $i = 0, 1, 2$ are the vertices of the triangle. A line-segment is given by two points $\mathbf{l}_0$ and $\mathbf{l}_1$. We determine the distance from $\mathbf{l}_0$ to the triangle surface along the line-direction given by $\mathbf{l}_1 - \mathbf{l}_0$:

$$d = -\frac{(\mathbf{l}_0 - \mathbf{p}_0) \cdot \mathbf{n}}{(\mathbf{l}_1 - \mathbf{l}_0) \cdot \mathbf{n}} \tag{4.22}$$

The line-segment intersects the triangle if and only if $0 \leq d \leq |\mathbf{l}_1 - \mathbf{l}_0|$. If the condition is met, we determine the intersection point by inserting d into the parametric line equation:

$$\mathbf{P}_i = \mathbf{l}_0 + \mathbf{l}_1 \cdot d \tag{4.23}$$

We check if the intersection point $\mathbf{P}_i$ is on the triangle in a similar way as in the previous section by solving u and v and examining their values.

4.4.3 Sphere and Triangle Test

A sphere is represented with a point in space and a radius. This is the general way to represent particles as well, especially when simulating SPH. In this test we compute a point on the triangle closest to the sphere center. If the distance between the sphere center and the computed point on the triangle is less than the radius, we have a intersection.

Johnson's distance algorithm [9] computes the distance between two arbitrary convex objects. The following presented is a version of Johnson's distance algorithm with some incorporated optimizations. Let $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2$ be the vertices of a triangle. A point in the triangle is then given with barycentric coordinates by $\mathbf{p} = \lambda_0\mathbf{p}_0 + \lambda_1\mathbf{p}_1 + \lambda_2\mathbf{p}_2$ as described in Equation 4.5.

We assume that all coordinates are in the sphere's local coordinate system, that is, the sphere is centered at the origin and $\mathbf{p}_i$ is in fact $\mathbf{p_i} - \mathbf{c}$, where $\mathbf{c}$ is the center of the sphere. Let us first find the closest point on the triangle plane by dropping the constraint $\lambda_i > 0$. The point $\mathbf{x}$ is the closest point if and only if the vector from $\mathbf{x}$ to the origin is orthogonal to the triangle plane. This is also the case if and only if the vector is orthogonal to two of the triangle's edges. The choice of the two edges is arbitrary but let us use $\mathbf{p_0p_1}$ and $\mathbf{p_0p_2}$. The vector $\mathbf{x}$ is orthogonal to $\mathbf{p_0p_1}$ and $\mathbf{p_0p_2}$ if and only if

$$(\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{x} = 0 \tag{4.24}$$
$$(\mathbf{p_2} - \mathbf{p_0}) \cdot \mathbf{x} = 0 \tag{4.25}$$

So lets substitute $\mathbf{x} = \lambda_0\mathbf{p_0} + \lambda_1\mathbf{p_1} + \lambda_2\mathbf{p_2}$ in Equation 4.24 and 4.25 for both edges and discover the following linear system of equations:

$$\begin{bmatrix} 1 & 1 & 1 \\ (\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{p_0} & (\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{p_1} & (\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{p_2} \\ (\mathbf{p_2} - \mathbf{p_0}) \cdot \mathbf{p_0} & (\mathbf{p_2} - \mathbf{p_0}) \cdot \mathbf{p_1} & (\mathbf{p_2} - \mathbf{p_0}) \cdot \mathbf{p_2} \end{bmatrix} \begin{bmatrix} \lambda_0 \\ \lambda_1 \\ \lambda_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \tag{4.26}$$

If $\lambda_i > 0$ for $i = 0, 1, 2$ then the point is an internal point of the triangle, otherwise it is a point on the triangle's boundary, which could be one of its vertices or a point on one of the triangle's edges. To compute the closest point on an edge, say the edge of $\mathbf{p_0p_1}$, we solve the following linear system:

$$\begin{bmatrix} 1 & 1 \\ (\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{p_0} & (\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{p_1} \end{bmatrix} \begin{bmatrix} \mu_0 \\ \mu_1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \tag{4.27}$$

If $\mu_i > 0$ for $i = 0, 1$ then the point is an internal point of the edge. If $\mu_0 \leq 0$ then $\mathbf{p}_1$ is the closest point else if $\mu_1 \leq 0$ then $\mathbf{p}_0$ is the closest point. The remaining edges are tested in a similar way. Furthermore, if $\mathbf{p}_0$ is the closest point of both edge $\mathbf{p_0p_1}$ and $\mathbf{p_0p_2}$ then the point $\mathbf{p}_0$ is the closest point of the triangle.

The outline of the method is as follows. First, we compute the parameters μ_i for the edges of the triangle. If we find a vertex $\mathbf{p}_i$ that is the closest of its two connecting edges, we return this vertex as the closest point of the triangle. If not, the closest point must be on one the edges and we compute the parameters λ_i for the triangle. If $\lambda_i \leq 0$ for any $i = 0, 1, 2$ then the closest point is on one of the triangles edges, otherwise the closest point is an internal point with barycentric coordinates λ_i for $i = 0, 1, 2$.

We solve the linear system of equations by using Cramer's Rule. For edge $\mathbf{p_0p_1}$, let

$$\mathbf{A} = [\mathbf{a_0 a_1}] = \begin{bmatrix} 1 & 1 \\ (\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{p_0} & (\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{p_1} \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}.$$

This gives us the solution

$$\mu_0 = \frac{\det[\mathbf{ba_1}]}{\det[\mathbf{A}]} \quad \mu_1 = \frac{\det[\mathbf{a_0 b}]}{\det[\mathbf{A}]}$$

In the first step we only need the signs of the parameters and because it has been shown that $\det[\mathbf{A}]$ is positive for affine independent triangles [9] we only need to compute the numerator of u_i in order to determine the sign of each parameter. We define $\Delta_i^{p_0p_1}$ as the numerator of μ_i

$$\begin{aligned} \Delta_0^{p_0p_1} &= \det[\mathbf{ba_1}] = (\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{p_1} \\ \Delta_1^{p_0p_1} &= \det[\mathbf{a_0 b}] = -(\mathbf{p_1} - \mathbf{p_0}) \cdot \mathbf{p_0} \end{aligned}$$

We know that $\mu_0 + \mu_1 = 1$ therefore also that $\det[A] = \det[\mathbf{ba}_1] + \det[\mathbf{a}_0\mathbf{b}]$. Lets define $\Delta^{p_0p_1} = \det[A] = \Delta_0^{p_0p_1} + \Delta_1^{p_0p_1}$. We proceed in similar for the remaining edges $\mathbf{p_0p_2}$ and $\mathbf{p_1p_2}$:

$$\begin{aligned} \Delta_0^{p_0p_2} &= (\mathbf{p_2} - \mathbf{p_0}) \cdot \mathbf{p_2} \\ \Delta_1^{p_0p_2} &= -(\mathbf{p_2} - \mathbf{p_0}) \cdot \mathbf{p_0} \end{aligned}$$

$$\begin{aligned} \Delta_0^{p_1p_2} &= (\mathbf{p_2} - \mathbf{p_1}) \cdot \mathbf{p_2} \\ \Delta_1^{p_1p_2} &= -(\mathbf{p_2} - \mathbf{p_1}) \cdot \mathbf{p_1} \end{aligned}$$

We solve the equations for the triangle itself in similar way. The solution is given by

$$\lambda_i = \frac{\Delta_i^{p_0p_1p_2}}{\Delta^{p_0p_1p_2}}$$

where

$$\Delta^{p_0p_1p_2} = \Delta_0^{p_0p_1p_2} + \Delta_1^{p_0p_1p_2} + \Delta_2^{p_0p_1p_2}$$

$$\Delta_0^{p_0p_1p_2} = \det \begin{bmatrix} (p_1 - p_0) \cdot p_1 & (p_1 - p_0) \cdot p_2 \\ (p_2 - p_0) \cdot p_1 & (p_2 - p_0) \cdot p_2 \end{bmatrix}$$

$$\Delta_1^{p_0p_1p_2} = -\det \begin{bmatrix} (p_1 - p_0) \cdot p_0 & (p_1 - p_0) \cdot p_2 \\ (p_2 - p_0) \cdot p_0 & (p_2 - p_0) \cdot p_2 \end{bmatrix}$$

$$\Delta_2^{p_0p_1p_2} = \det \begin{bmatrix} (p_1 - p_0) \cdot p_0 & (p_1 - p_0) \cdot p_1 \\ (p_2 - p_0) \cdot p_0 & (p_2 - p_0) \cdot p_1 \end{bmatrix}$$

However, the determinants computed for the edges can be reused for the computation of the above determinants. We rewrite the expressions

$$\begin{aligned}\Delta_0^{p_0p_1p_2} &= \Delta_0^{p_0p_1}\Delta_1^{p_1p_2} + \Delta_2^{p_1p_2}((\mathbf{p_1}-\mathbf{p_0})\cdot\mathbf{p_2})\\ \Delta_1^{p_0p_1p_2} &= \Delta_1^{p_0p_1}\Delta_0^{p_0p_2} - \Delta_2^{p_0p_2}((\mathbf{p_1}-\mathbf{p_0})\cdot\mathbf{p_2})\\ \Delta_2^{p_0p_1p_2} &= \Delta_2^{p_0p_2}\Delta_0^{p_0p_1} - \Delta_1^{p_0p_1}((\mathbf{p_2}-\mathbf{p_0})\cdot\mathbf{p_1})\end{aligned}$$

Lets conclude the steps needed for testing the intersection of a triangle and a sphere centered at the origin:

1. We compute the determinants for the three edges of the triangle. If $\Delta_j^{p_ip_j} \leq 0$ and $\Delta_k^{p_ip_k}$ for any $i = 0, 1, 2$ and $j, k \neq i$, then we return the closest point $\mathbf{x} = \mathbf{p_i}$.
2. Otherwise, compute the determinants for the triangle. if $\Delta_i^{p_0p_1p_2} \leq 0$ for any $i = 0, 1, 2$, then return the closest point $\mathbf{x} = \mu_j\mathbf{p}_j + \mu_k\mathbf{p}_k$, where $j, k \neq i$ and $\mu_j = \Delta_j^{p_jp_k}/\Delta^{p_jp_k}$ and $\mu_k = \Delta_k^{p_jp_k}/\Delta^{p_jp_k}$.
3. Otherwise, return the closest point $\mathbf{x} = \lambda_0\mathbf{p}_0+\lambda_1\mathbf{p}_1+\lambda_2\mathbf{p}_2$, where $\lambda_i = \Delta_i^{p_0p_1p_2}/\Delta^{p_0p_1p_2}$.
4. The triangle intersects the sphere if and only if $\|\mathbf{x}\| \leq r$, where r is the radius of the sphere.

Before entering the steps above, we can check if any of the triangle's vertices are contained in the sphere. If one of them are contained in the sphere, we return that vertex as the contact point. This is a very cheap process and can help us exit early. As we centered the sphere at the origin, it is important that we remember to translate the contact point back to world coordinates.

4.4.4 Virtual Boundary Particles

The interaction between a particle and a triangle mesh always depend on the distance between them. In Figure 4.9 several isosurfaces are generated from the distance field which is C^0 continous everywhere. This generates discontinous first derivatives in the distance field which in turn yields artifacts such as jittering of particles in concave regions. One way to remove the problem is to replace the minimum distance by a weighted sum similar to the technique described in Section 4.1.3. However, this can distort the distance field near triangle boundaries as seen in Figure 4.9.

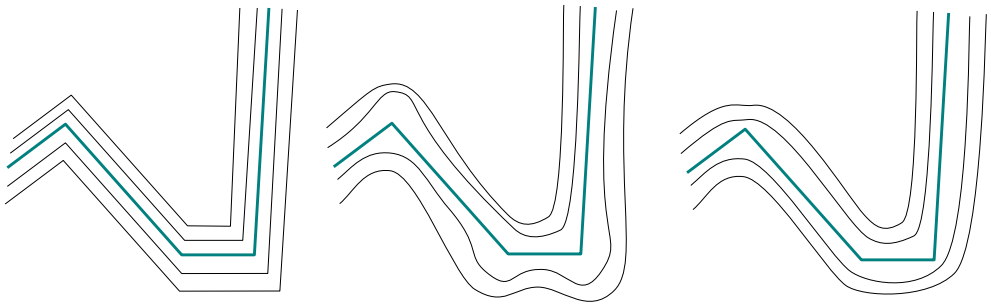

Figure 4.9: Left: Isosurface of the euclidian distance field of a linear curve with discontinous first derivative. Middle: Weighted sums yield a smooth surface with bulges. Right: Convolution yields a bulge-free smooth isosurface.

A way of generating a bulge-free surface around a skeleton S is to define a scalar function F_S as the convolution:

$$F_S(\mathbf{p}) = \int_{\mathbf{x}\in S} W(\mathbf{p}-\mathbf{x})d\mathbf{x} \tag{4.28}$$

The convolution sums up infinite small parts of the skeleton each properly weighted. In Figure 4.9, the bulge-free smooth isosurfces generated by the convolution can be seen. For skeletal elements other than points, the integral in Equation 4.28 yields complex comptutations. Monaghan [38], the founder of the SPH formalism used boundary particles on fixed borders and modeled the interaction between boundary particles and regular particles using a Lennard-Jones-like potential. This reduces the problem of triangle-particle interaction to particle-particle interaction. In [14], they distribute the boundary particles on a triangle mesh using the seven point rule, each weighted by Gauss quadrature rule [45].

The interaction potential of a particle $\mathbf{p}$ and triangle t can be described as:

$$U(\mathbf{p},t) = \int_{\mathbf{x}\in t} U(\mathbf{p}-\mathbf{x})d\mathbf{x} \approx A\sum_i w_i U(\mathbf{p}-\mathbf{x}_i) \tag{4.29}$$

where A is the area of the triangle, $\mathbf{x}_i$ the boundary particles with weights w_i. The force acting on particle $\mathbf{p}$ due to triangle t then becomes:

$$\mathbf{f}(\mathbf{p},t) = \int_{\mathbf{x}\in t} \tau(|\mathbf{p}-\mathbf{x}|)d\mathbf{x} \tag{4.30}$$

where τ is the traction function to model repulsion and adhesion. Mûller et al. [14] propose using the following traction function:

$$\tau(r) = \begin{cases} k\frac{(h-r)^4-(h-r_0)^2(h-r)^2}{h^2 r_0(2h-r_0)} & if r < h \\ 0 & otherwise \end{cases} \tag{4.31}$$

where h is the interaction radius of the particles and k controls the stiffness of the interaction. Equation 4.31 evaluates to zero when $r = r_0$ which is the preferred distance between particles and surface. This approximates to:

$$\mathbf{f}(\mathbf{p}) \approx \sum_i A_i \sum_j w_{ij}\tau(|\mathbf{p}-\mathbf{x}_{ij}|) \tag{4.32}$$

They also compute friction between boundary particles and fluid particles by including the boundary particles in the viscosity computation 3.4 of the SPH particles. The main advantage of using gaussian boundary particles is the smoothness of the force fields which is essential for stability in the simulation.

Chapter 5

Deformable Bodies

In this section we perform a small survey on deformable bodies and different methods that are active today. A deformable body is defined by its undeformed shape known as the equilibrium configuration and a set of material parameters that define how the body deforms under applied forces. When forces are applied, the body deforms and a point at location $\mathbf{p}$ is moved to a new location $\mathbf{x}(\mathbf{p})$, where $\mathbf{x}$ is a vector field. The deformation can also be described by the displacement vector field $\mathbf{u}(\mathbf{p}) = \mathbf{x}(\mathbf{p}) - \mathbf{p}$. The dimensionless elastic strain ϵ is then computed from $\mathbf{u}(\mathbf{p})$. The strain tensor is the geometrical expression describing a 3D deformation caused by the action of stress on the body. Many strain tensors exist, although Green's nonlinear strain tensor and Cauchy's linear strain tensor are probably the most commonly used [2]:

$$\epsilon_G = \frac{1}{2}\left(\nabla\mathbf{u} + [\nabla\mathbf{u}]^T + [\nabla\mathbf{u}]^T \nabla\mathbf{u}\right) \tag{5.1}$$

$$\epsilon_C = \frac{1}{2}\left(\nabla\mathbf{u} + [\nabla\mathbf{u}]^T\right) \tag{5.2}$$

where $\epsilon_G, \epsilon_C, \nabla\mathbf{u} \in \mathbb{R}^{3\times3}$. However, a linear strain can cause unacceptable volume distortions during large deformations. The internal stress tensor $\sigma \in \mathbb{R}^{3\times3}$ (measured as force per unit area) can then be computed using Hooke's linear material law:

$$\sigma = \mathbf{E} \cdot \epsilon \tag{5.3}$$

where $\mathbf{E}$ is a rank four tensor. For isotropic materials, the coefficient of $\mathbf{E}$ only depend on Young's modulus and Poisson's ratio which each describe the stiffness of a given material and how much thinner the material gets when it's streched. The forces are then computed in the following sequence:

$\mathbf{u}$	$\rightarrow$	$\nabla\mathbf{u}$	$\rightarrow$	ϵ	$\rightarrow$	σ	$\rightarrow$	$\mathbf{f}$
displacements		*derivatives*		*strains*		*stresses*		*forces*

Finite element method (FEM) is probably the most popular method for simulating deformable bodies. The body is discretized using an irregular mesh into a set of elements. Instead of solving the spatially continous function $\mathbf{x}(\mathbf{p}, t)$ at time t, we only solve for the nodes of the mesh. In [29] [24] [13] a simple form of the Finite Element Method is used for the simulation of deformable bodies, often referred to as the explicit Finite

Element Method which is quite easy to understand and implement compared to the original Finite Element Method. Many of the FEM based techniques produces realistic and visually convincing results but it's not designed for interactive and real-time use.

When using bodies composed of homogenous interior one can use the Boundary Element Method [27]. In this method the equation of motion is transformed into a surface integral and computations are only performed on the surface (boundary) of the elastic body instead of its volume. The three dimensional problems becomes two dimensional and substantionally increases the computation speed. However, topological changes such as fracturing becomes difficult to handle.

The simpliest and most intuitive of all deformable models is mass-spring systems [6] [32]. As the name implies, these models simply consist of point masses connected together by a network of massless springs. The springs are regularly spaced in a lattice. The force acting on mass i is then generated from the spring connecting the point masses i and j:

$$\mathbf{f}_i = k_s \left(|\mathbf{x}_{ij}| - l_{ij} \right) \frac{\mathbf{x}_{ij}}{|\mathbf{x}_{ij}|} \tag{5.4}$$

where $\mathbf{x}_{ij}$ is the difference in position $(\mathbf{x}_j - \mathbf{x}_i)$, k_s is the spring stiffness and l_{ij} is the spring's rest length. In order to simulate the dissipation of energy during deformation, viscoelastic springs are used to damp out relative motion. In addition to Equation 5.4 a viscous force is used:

$$\mathbf{f}_i = k_d \left(\mathbf{v}_j - \mathbf{v}_i \right) \tag{5.5}$$

where k_d is the spring's damping constant. Mass-spring system is basically a particle system with connectivity. In [21] a more general particle system is used. The deformation energies are derived from soft constraints that are to be maintained in the model. Given some constraint $\mathbf{C}(\mathbf{x}) = 0$, the energy is defined as $\frac{k_s}{2}\mathbf{C}^T(\mathbf{x})\mathbf{C}(\mathbf{x})$ with k_s defining the stiffness coefficient. These energies are minimized at the model's rest state and enforce the preservation of model distances, areas and volumes.

Particle systems tend to be more intuitive and simple to implement. They are computationally efficient and handles large deformations with ease. However, unlike FEM which are built on elasticity theory, mass-spring systems are not necessarily accurate and the behaviour is dependent on the mesh resolution and topology. The spring constants are often chosen arbitrary and one can say little about the material being modeled.

5.1 Point-based Animation

Point-based animation is a combination of mesh-free physics and point sampled surfaces [25]. In [15], a mesh-free continuum-mechanics-based framework is introduced. The framework can model the animation of elastic, plastic and melting objects. The model is sampled at a finite number of point locations without connectivity information and without the need of generating a volumetric mesh. The simulation quantities such as location, density, deformation, velocity, strain, stress, and body forces are carried by the physically simulated points called phyxels (physical elements). The forces acting

on a phyxel are computed by weighting the neighbouring phyxels similar to SPH described in Section 3.2. In case of stretching and compression or even fracturing [19], a resampling and relaxation scheme ensures that the surface is completely covered and uniformly sampled with surface elements used for rendering. This is done by splitting and merging surfels similar to a surface extraction technique covered later in this thesis.

The authors in [22] describe how to extend the framework to viscous material such as fluids by merging the solid mechanics equation with the Navier-Stokes equations. This allows the simulation of non-realistic materials such as elastic fluids [41]. The material properties such as stiffness, viscosity and plasticity can be defined per particle and coupling these properties to temperature, local freezing and melting of objects becomes possible. Müller et al. achieve interactive framerates with a surface model of 10.000 surfels and approximately 200 phyxels [15].

5.2 Meshless Deformations Based on Shape Matching

A different approach for animating deformable models is proposed in [16]. The idea is to replace energies by geometric constraints and forces by distances of current positions to goal positions. This is similar to [18] where position based dynamics is introduced. The typical instability problem such as the overshooting problem of explicit integration schemes is removed. The nodes of a volumetric mesh are treated as point masses and animated as a particle system without connectivity. At every time step, the original configuration of points is fitted to the actual point cloud using shape matching techniques for point clouds. The fitted rest shape yields goal positions for all point masses. Each point mass is then pulled towards its goal position while the velocity is computed from the displacement divided by the time step.

The performance of this approach depends on the number of points used for shape matching but the computational complexity scales linearly with the number of points. Performing shape matching on 1145 objects and $24,618$ points takes 0.12 milliseconds. However, this is without collision detection and visualization. In terms of object representation, memory efficiency and computational complexity and unconditional stability make the approach particular interesting in interactive applications.

5.3 Data Structures

To realistically process the interaction between deformable objects very efficient collision detection algorithms must be used. Bounding volume hierarchies have proven to be very efficient [43] and many different types of bounding volumes have been investigated, spheres, axis-aligned bounding boxes (AABB), oriented bounding boxes (OBB) and discrete oriented polytopes (k-DOPs). Initially BVH's was designed for rigid bodies without any dynamic deformation and usually a preprocessing step. In case of deformable bodies, the BVH tree must be updated on any topological changes, often every time step. They pose a computational burden and storage overhead for complex objects. In [43] Akenine-Möller evaluate suitable BVH's for deforming bodies that are pre-built and then updated efficiently during simulation using a hybrid update combining the

bottom-up and top-down strategy mentioned in Section 4.2.1.

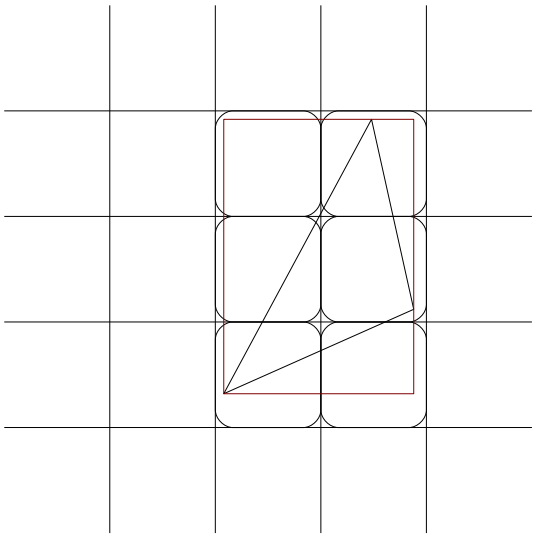

Figure 5.1: A hashed tetrahedron. Hash values are computed for all grid cells covered by the AABB of a tetrahedron. The tetrahedron is checked for intersection with all vertices found at these hash indices.

Müller discuss using optimized spatial hashing [20] for deformable objects whose approach both detects collisions and self-collisions. They perform hashing on tetrahedrons in two passes. In the first pass, all AABB's are updated and the vertices of the tetrahedrons are hashed into the hash table. In the second pass, the AABB representing a tetrahedron is discretized and hashed into the hash table, that is, all cells that the tetrahedron touches are hashed. Figure 5.1 illustrates this. If a vertex and a tetrahedron is mapped to the same hash index and the vertex is not a member of the tetrahedron, then a intersection test is performed. The algorithm is of complexity $O(n)$ and is linearly dependent on the number of primitives. They achieve interactive framerates and in Section 7.3 we implement a modified version of the method for our prototype.

Chapter 6

Surface Extraction and Visualization

No matter what fluid simulation method used, they still require the generation of a renderable free surface using a polygonization technique such as metaballs and marching cubes, point splatting, or carpet visualization. In this section we discuss different techniques used when reconstructing the surface of a fluid and visualizing it.

A surface can be defined as an implicit function

$$f(x, y, z) = f(\mathbf{p}) = 0 \tag{6.1}$$

where a point $\mathbf{p}$ is on the surface if $f(\mathbf{p}) = 0$. The surface normal is described by the gradient

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right) \tag{6.2}$$

These implicit surfaces are contours also known as isosurfaces through some scalar field in 3-dimensional space. The surface generation of a set of particles usually known as a point cloud is not a trivial task. The evaluation of the color field is the most commonly technique used to extract a surface from a SPH simulation.

6.1 Polygonization Techniques

There are several ways of visualizing an iso-surface on today's graphics hardware. In this section we'll discuss different polygonization techniques which can be employed to visualize an iso-surface. The common base for all polygonization techniques is that they construct a polygon mesh, each polygon consisting of three or more points. The triangle primitive is often used due to its simplicity and efficiency on today's graphics hardware.

6.1.1 Marching Cubes

The marching cubes algorithm generates an iso-surface and extracts a polygonal mesh from the 3-dimensional scalar field. The method traverses the scalar field and takes eight neighbor locations (creating an imaginary cube) and then determines the polygons needed to represent the isosurface passing through this cube. The resolution of the

sampled grid points affects the smoothness and resolution of the generated polygonal mesh. If any of the eight scalar values are greater than the iso-value then it is considered as inside the surface and outside the surface when less than the iso-value. This creates 15 unique cube configurations and a total of 256 ($2^8 = 256$) possible polygon configurations when these unique configurations are reflected and symmetrically rotated.

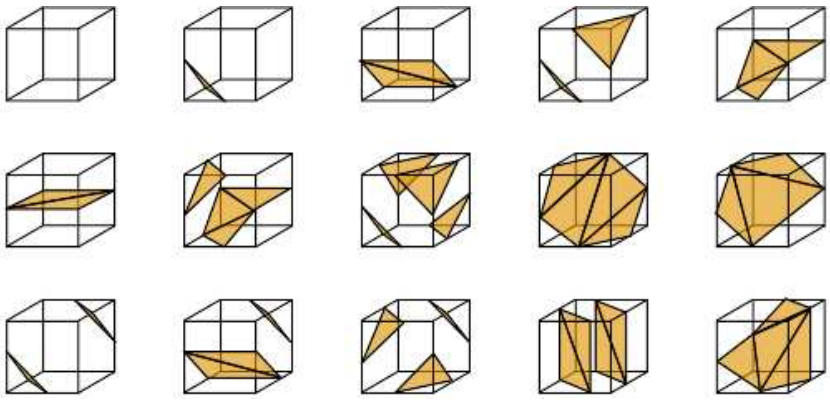

Figure 6.1: 15 unique cube configurations.

Each vertex of a generated polygon is then linearly interpolated between the two scalar values along the cube's edge. The gradient of the scalar field at one of the grid points is also the surface normal at that point. These can then be used to interpolate a normal for each generated vertex, which later can be used for shading computation when rendering the polygonal surface.

A drawback of the marching cubes algorithm is that small details of the surface requires a very fine or an adaptive discretization of the underlying space. Another noticeable artifact occurs in between different frames when a fluid drop moves between two different cubes. The fluid drop then changes shape every frame instead of maintaining a spherical shape over several frames. Although this method is not commonly used in interactive simulations due the computational cost there exist several implementations [3] where they use a low resolution grid achieving interactive frame rates. The latest graphics hardware from nVidia (8xxx series with geometry shader) can perform the marching cubes algorithm on the GPU (graphics processing unit) [4] which dramatically increases the performance.

6.1.2 Screen Space Meshes

Müller et al. [35] presented a new approach for the generation and rendering of surfaces defined by the boundary of a 3D point cloud. The method constructs a 2D screen space triangle mesh from a depth map using a modified marching squares technique and transforms the mesh into 3D world coordinates. The mesh can then be used to compute occlusion, reflection, refraction and other shading effects. The algorithm resolves parts of the fluid which is close to the camera with more triangles than distant parts, automatically yielding a view-dependent level of detail. One disadvantage of the algorithm can be found when computing shadows due to the fact that the method is

view-dependent and we have a mesh from the camera point of view. In order accurately calculate shadows another mesh needs to be computed from the lights point of view.

The basic algorithm consists of 7 steps. The first step involves setting up a depth map similar to point splatting of particles. A grid is constructed with size $N_x \times N_y$, where $N_x = W/h+1$, $N_y = H/h+1$ and h is the screen spacing or the resolution of the grid. All depth values in the grid are initialized to ∞. The algorithm iterates through all N particles twice. In the first pass, the depth values are set. In the second pass, additional depth values are generated where silhouettes cut the grid. Consider a particle with homogeneous coordinates $\mathbf{p} = [x, y, z, 1]^T$. These coordinates are transformed using projection matrix $\mathbf{P}$ to get:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w \end{bmatrix} = \mathbf{P} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{6.3}$$

The projected coordinates are then computed as:

$$\begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} = \begin{bmatrix} W\left(\frac{1}{2} + \frac{1}{2}x'/w\right) \\ H\left(\frac{1}{2} + \frac{1}{2}y'/w\right) \\ z' \end{bmatrix} \tag{6.4}$$

In OpenGL, perspective division yields canonical coordinates in the range $-1..1$ for all three coordinates. However, this would distort the depth value non-linearly so this is ignored as can be seen in the above computation. This way we have $x_p = [0 \dots W]$ and $y_p = [0 \dots H]$ while z_p is the non-distorted distance to the camera. We assume that the aspect ratio of the projection is chosen according to the viewport (i.e. W/H). In this case we have a single projected radii ($r_p = r_x = r_y$):

$$r_p = \frac{rW\sqrt{\mathbf{P}_{1,1}^2 + \mathbf{P}_{1,2}^2 + \mathbf{P}_{1,3}^2}}{w} \tag{6.5}$$

All depth values in the grid at (i, j) with $(ih - x_p)^2 + (jh - y_p)^2 \leq r_p^2$ are updated as:

$$z_{i,j} = \min(z_{i,j}, z_p - rh_{i,j}) \tag{6.6}$$

where

$$h_{i,j} = \sqrt{1 - \frac{(ih - x_p)^2 + (jh - y_p)^2}{r_p^2}} \tag{6.7}$$

The grid is now populated with depth values. In the second pass, the algorithm iterates through the particles a second time. This time around to detect silhouettes on edges which connect depth values further apart than z_{max} which is the depth connection threshold. All cuts from a particle p with position (x_p, y_p) and radius r_p are computed and considered as a silhouette node candidate. The cut is only stored if two conditions are satisfied:

1. The depth z_p of the particle is smaller than the average depth of the silhouette edge.

2. The location of the cut on the silhouette is further from the endpoint with the smaller depth value than a previously stored silhouette node for this edge.

The next step is to generate the vertices and triangles from the grid nodes and the silhouette nodes. Each grid node with an initialized depth value ($\neq \infty$) generates a vertex at its screen space location with the same depth value. Each silhouette edge with only one initialized node generates a silhouette vertex at the location of the cut with the depth of its silhouette node. This vertex will be on the outer silhouette. Each silhouette edge with both nodes initialized generates two additional vertices, both at the location of the silhouette node. The vertex associated with the endpoint with smaller depth value (the front vertex) gets the depth value from the silhouette node. The other generated vertex (the back vertex) gets its depth value extrapolated from its neighboring nodes as can be seen in Figure 6.2.

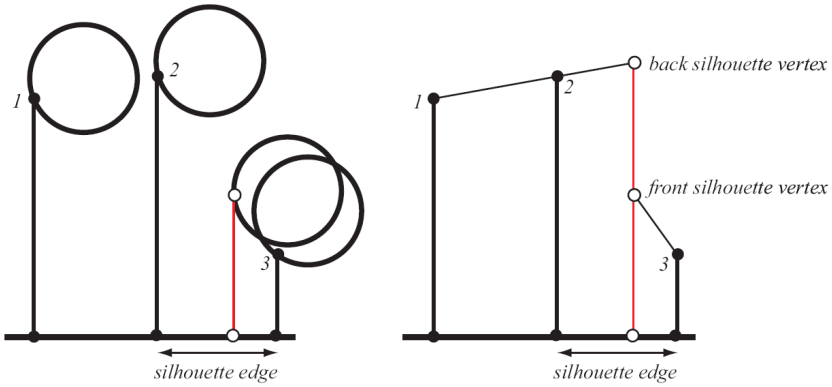

Figure 6.2: Left: Silhouette node created on a solhouette edge. Right: The back vertex gets its depth value extrapolated from vertex 1 and 2. [35]

The generation of triangles is similar to Marching Squares [30]. We traverse each cell in the grid one by one and generate triangles for them. Each of the cell's edges is either a silhouette edge or not. This leads to 16 unique cases that can occur when traversing the cells. We only generate a triangle when all three vertices exist.

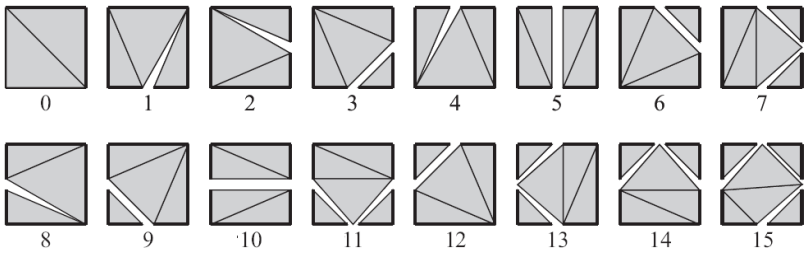

Figure 6.3: 16 unique square configurations with silhouette generation. [35]

At this point we have a generated mesh in screen space so the next step is to transform

the mesh back into world coordinates. This is done by inverting the transformation in Equation 6.3 and 6.4. The world coordinates can be computed using the inverse projection matrix $\mathbf{Q} = \mathbf{P}^{-1}$.

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} (-1 + 2x_p/W)w \\ (-1 + 2y_p/H)w \\ z_p \\ w \end{bmatrix} \tag{6.8}$$

where the projective divisor w can be computed as:

$$w = \frac{1 - \mathbf{Q}_{4,3} z_p}{\mathbf{Q}_{4,1}(-1 + 2x_p/W) + \mathbf{Q}_{4,2}(-1 + 2y_p/H) + \mathbf{Q}_{4,4}} \tag{6.9}$$

Once the mesh is transformed into world coordinates, we can compute vertex normals. The result so far produces a bumpy depth map, this is fixed by smoothing the depth map using a separable binomial filter before sampling the depth map. The filter is applied to all values $z_{i,j} \neq \infty$ in two passes, first horizontally and then vertically. Müller also propose a simple iterative scheme that replaces each vertex position with the average of its neighboring vertex positions. This silhouette smoothing causes the outer silhouette to shrink and make connected particles look more realistic as can be seen in Figure 6.4.

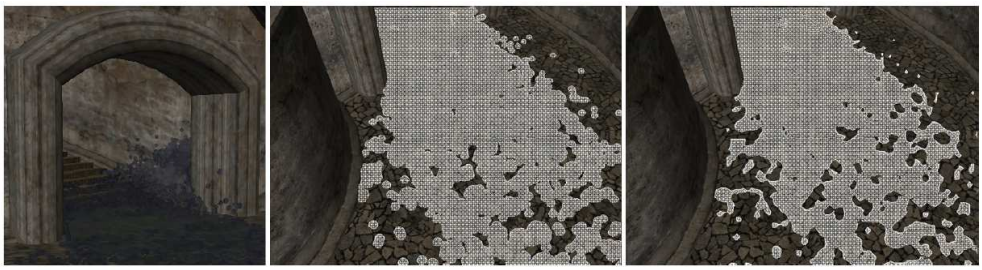

Figure 6.4: Dungeon scene illustrating the differences in silhouette smoothing. Without and with silhouette smoothing in the middle and right image respectively. [35]

Screen space meshes is a powerful yet simple way to construct and visualize surfaces with interactive framerates. It is also well suited to be implemented on the GPU although the CPU implementation described in Chapter 7.4.1 already allows generation of complex surfaces in real-time.

6.2 Point-based Techniques

Unlike other geometry-based visualization techniques, point-based rendering uses points instead of triangles to represent and render an isosurface. This section is divided into two parts, the generation of surface points and the rendering of these surface points known as point-splatting.

6.2.1 Surface Extraction

In order to extract an surface from a point cloud, we need a way to define a scalar field for a set of particles. The already defined color field in Section 3.2 can be used to define our isosurface. The particles belonging to the surface can be described as:

$$|\mathbf{n}_i| > l \tag{6.10}$$

where $\mathbf{n}_i$ is the value of the gradient of the color field at particle i and l is a threshold value. The advantage of using this approach is that we already have evaluated this when we computed the surface tension force in the SPH simulation. The same goes with the surface normal. However, other algorithms exists. In [5] the algorithm generates point samples in a grid covering the entire isosurface similar to the Marching Cubes algorithm [30]. These point samples are then projected onto to the isosurface using an approximate projection with a Newton-Raphson root-finding method.

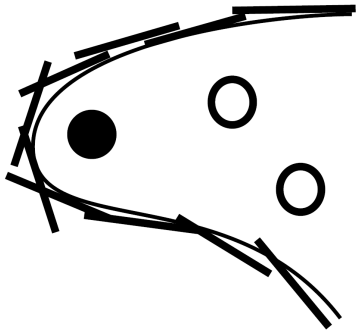

Figure 6.5: 2D example of a refined solution set.

The resulting surface points which are also known as surfels (surface elements) are usually too few and large holes will appear in the geometry. Therefore, surface refinement is often needed in order to increase the point cloud density. In a previous master's thesis [46] conducted at VRLab a surface extraction and refinement algorithm was designed. This surface extraction algorithm is based on the color field used in a SPH simulation. It's an iterative solution that initially creates a set of surfels and refines this set until no more surfels needs to be refined.

The basic idea of the algorithm is as follows and is similar to [5] in some ways. In each iteration, each surfel $p \in \mathbf{P}$, where $\mathbf{P}$ is a set of surfels is moved closer to the isosurface using one Newton-Raphson step:

$$\mathbf{x}_{i+1} = \mathbf{x}_n - \frac{f(\mathbf{x}_n)\nabla f(\mathbf{x}_n)}{|\nabla f(\mathbf{x}_n)|^2} \tag{6.11}$$

The function f is defined as

$$f(\mathbf{x}) = c_s(\mathbf{x}) - \kappa_{isocolor} \tag{6.12}$$

where $c_s(\mathbf{x})$ is the color field and $\kappa_{isocolor}$ a constant. The length of the step for each surfel is an approximation of the error in placement for that particular surfel. If

the approximated error is considered too great, the surfel is discarded and not kept for the next iteration. Once faulty surfels have been discarded, the set is divided into two sets, one set with surfels with good enough approximation and one set with surfels that need further refinement. The first set is added to the final solution while the second set is divided into four new surfels which are sent back into the iterative process for further refinement. This process continues until no further surfels need to be refined or a fixed number of iterations have been reached.

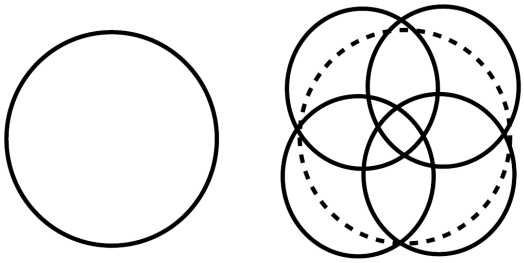

Figure 6.6: Illustrates a surfel split. Each surfel is divided into four new surfels.

6.2.2 Surface Shading

In order to render a set of points as a surface we use surfels as rendering primitives. A surfel is defined as a point in space, a normal describing the orientation of the surface at that point and a radius specifying the points influence on neighbouring surfels. Depending on what rendering algorithm used, a surfel area can either be circular, rectangular, quadratic or elliptical. The usual way to render a surfel is to send a quad consisting of four vertices to the GPU. However, the number of vertices can become huge. Instead a more efficient way is to sent one vertex per surfel together with a normal and a radius to the GPU. The splatting of the actual surfel is then done in shader program.

In [11] a high quality point-splatting technique is implemented on the GPU using multiple render targets (MRT) and a deferred shading approach. Deferred shading was usually non-interactive a few years ago but are now possible in real-time using a programmable GPU which also has the property of writing to several buffers in one rendering pass (MRT). The technique is used in a recently released computer-game named *S.T.A.L.K.E.R.* The scene geometry is sent once to the graphics card, storing per-pixel attributes into the G-buffer. The G-buffer is a set of render targets, storing the normal, depth, diffuse color and specular color of each pixel. These are then combined in a later rendering pass where per-pixel shading occurs. The advantage of deferred shading in comparison with traditional rendering algorithms is that the computationally worst case complexity is $O(N_{objects} + N_{lights})$ instead of $O(N_{objects} \times N_{lights})$.

The first step in rendering a surfel is called *Visibility Pass*. In this step all surfels are rendered to the depth buffer giving us the front-most of all the surfels. Back-face culling is also used to remove surfels facing away from the viewer. The depth buffer is then used in the next step called *Attribute Pass*. We render all surfels once again with the depth buffer shifted by ϵ, only this time all surfels that fail the depth test will

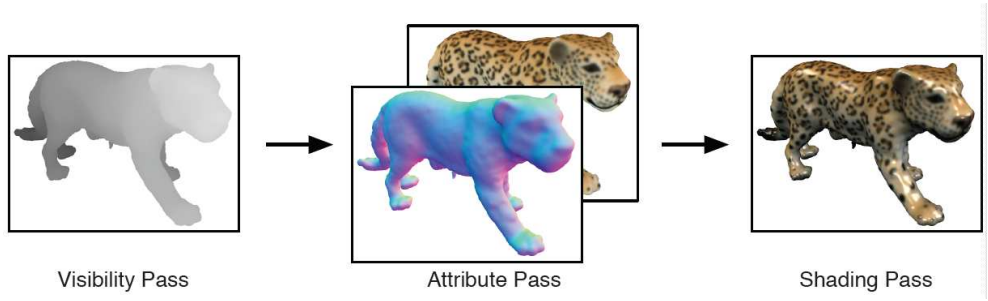

Figure 6.7: Visibility pass (left), attribute pass (middle) and shading pass (right).

be excluded. We use additive blending to accumulate the normals $\mathbf{n}_i$ and colors c_i on a per-pixel level. In the last step known as the *Shading Pass* a large screen-size quad is rendered and when each pixel is rasterized the previously stored normal, color and depth information is used to compute per-pixel shading.

Chapter 7

Implementation

In this chapter implementation issues will be addressed. In parallell to the implementation of this simulation, a collision library is being developed at VRLab. This library is based on the work of a previous thesis. The library is far from complete and did not contain support for particle-trimesh collision although it did contain a BVH data structure for trimeshes. The non-existing parts were implemented and integrated into the library which now supports particle-trimesh and sphere-trimesh collisions using BVH and spatial hashed grids. The existing particle system was also replaced to support more general particle data and to increase performance. The library is to become a part of *Colosseum 3D* which is a framework for developing interactive 3D simulations. More information about the *Colosseum 3D* framework can be found on their website [1].

7.1 System Overview

Before going further into specific implementation details, we'll briefly discuss the system design. As mentioned earlier, the program is being developed on top of a collision library. The library consists of four main modules, *Dynamics*, *Collision*, *Math* and *Util*. The program itself communicates with the library through the *Dynamics* module, which in turn calls the underlying layers performing collision queries and time integration of the objects in the simulation.

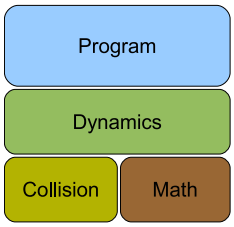

Figure 7.1: The figure illustrates the program overview. The program communicates with the underlying collision library through the dynamics module.

Each body in the simulation can have a set of operators that operate on the body. For example, a object can have a force operator that in each time step of the simulation loop causes the object to fall in the negative y-axis simulating gravity. A SPH operator

is created and used to compute the interaction forces between particles in a particle system. Time integration is another operator that is applied to the particle system determining the particles new positions. This creates a very modular and flexible solution. The other part of the program is the visualization part which consists of surface reconstruction and rendering. Figure 7.2 shows the program flow.

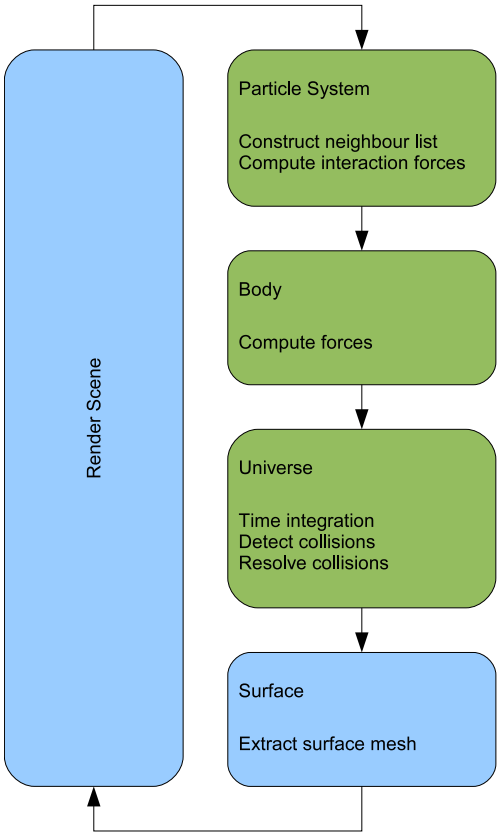

Figure 7.2: A flow chart showing the different parts and steps of the program.

7.2 Smoothed Particle Hydrodynamics

We choose to simulate the contrast fluid using SPH described in Section 3.2. The particle system is subjected to a SPH operator in each time step of the simulation loop. The operator computes the accumulated force acting on each particle. We use spatial hashed grids described in Section 4.1.2 to find the neighbouring particles. The algorithm uses a cell size of h, where h is the interaction radius of a particle and the smoothing length of the kernel function.

In the first pass, each particle is hashed into a hash map. In the second pass, each particle's cell and adjacent cells are examined when searching for neighbouring particles.

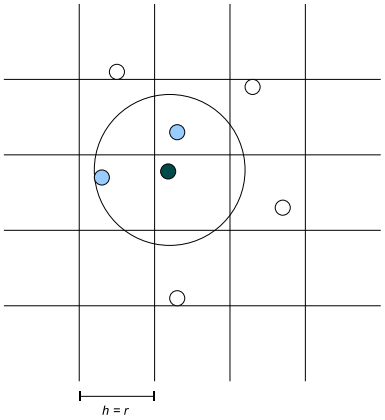

Figure 7.3: A grid with cell resolution $h = r$ occupied by particles. r is the interaction radius of a particle.

The total number of examined cells is 27. However, since interaction pairs are symmetrical it is enough to check 13 of the 26 neighbouring cells.

For stability and speed we use the Leap-Frog time integration scheme when resolving the particle positions:

$$\mathbf{v}_{t+\frac{1}{2}\Delta t} = \mathbf{v}_{t-\frac{1}{2}\Delta t} + \Delta t \cdot \mathbf{a}_t \tag{7.1}$$

$$\mathbf{x}_{t+\Delta t} = \mathbf{x}_t + \Delta t \cdot \mathbf{v}_{t+\frac{1}{2}\Delta t} \tag{7.2}$$

7.3 Collsision Detection

The aim on this thesis was to implement an effective collision detection scheme between a particle system and a triangle mesh. The complex models used in the simulations are described with vertices and triangles. Two different broad-phase methods have been implemented and evaluated. The first method uses spatial hashed grid described in Section 4.1.2 as data structure for the triangle mesh. We use a similar approach to Müller in [20] and hash each cell within the triangle's AABB using the same hash function used in the SPH neighbour search. However, using this approach introduces an extra step when checking a particle against the triangle mesh. In addition to check the cell occupied by the particle, we need to check the 26 surrounding cells as well.

Instead we extend the triangle's AABB by one cell size in each dimension and compute hash indices for each of the cells as can be seen in Figure 7.4. When a particle needs to be checked against the triangle mesh, we only examine the cell occupied by the particle. We don't need to check the surrounding cells of the particle because this step was already considered when hashing the triangle's extended AABB. The actual hashing of the triangle mesh is done before entering the simulation loop since we're only considering a static triangle mesh.

The second method uses a top-down BVH described in Section 4.2.1 to represent the triangles of the mesh. As mention earlier, we use a static triangle mesh and no rebuild-

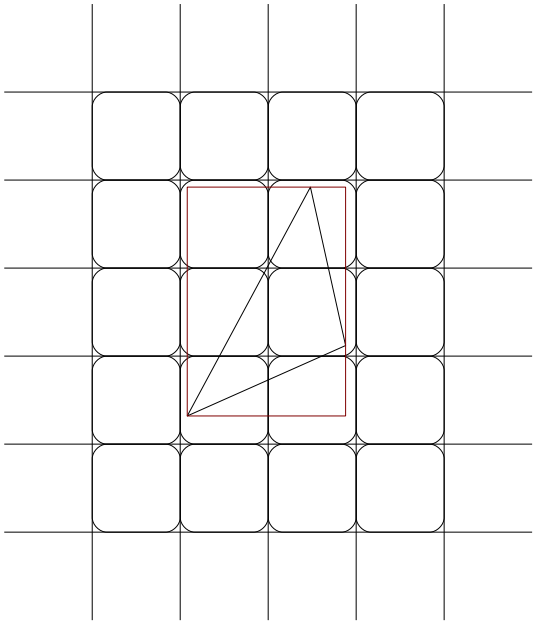

Figure 7.4: Not only the cells covered by the AABB are hashed but also the surrounding cells for efficiency.

ing of the BVH is needed in each simulation step. The BVH is generated once before entering the simulation loop. For each particle, we traverse the BVH in a top-down manner starting at the root node. A quick AABB-AABB intersection test is performed between each node and if they intersect we continue down the branch of the tree. If a leaf node is reached, we may have a potential particle-triangle collision and further checking is needed.

The narrow-phase test implements the sphere-triangle intersection test described in Section 4.4.3. If intersections are detected a callback routine is called to handle the collision responses. We implement a simple yet effective way to resolve collisions. We reflect the particle velocity $\mathbf{v}_i$ about the contact normal $\mathbf{n}$:

$$\mathbf{v}_{i+1} = \mathbf{v}_i - \mathbf{n}(1+\epsilon)(\mathbf{v}_i \cdot \mathbf{n}) \tag{7.3}$$

$$\mathbf{v}_{i+1} = \mathbf{v}_i + \frac{\mathbf{J}}{m} \tag{7.4}$$

$$\mathbf{J} = -\mathbf{v}_n(1+\epsilon)m + \mathbf{v}_t(max\left[0, 1-\mu\frac{|\mathbf{v}_n|}{|\mathbf{v}_t|}\right]-1)m \tag{7.5}$$

where ϵ is the restitution coefficiant of the material. We also project the particle position back onto the contact surface if the particle is still colliding after one time step.

$$\mathbf{p}_{i+1} = \mathbf{p}_i + \mathbf{n}d \tag{7.6}$$

where d is the contact penetration depth. This gives us plausable result although artifacts may occur in concave regions where a particle can get stuck when using a large

time step. In Figure 7.5 such an artifact is visualized in 2D. The collision between the particle and triangle t_1 is resolved causing another collision between the particle and triangle t_2. This causes a jittering effect when running the simulation. The solution may be to use an iterative solver that during a number of iterations puts the particle into a legal state but this can become quite expensive when dealing with a large number of particles. Smaller time steps or introducing boundary particles as described in Section 4.4.4 would be another solution.

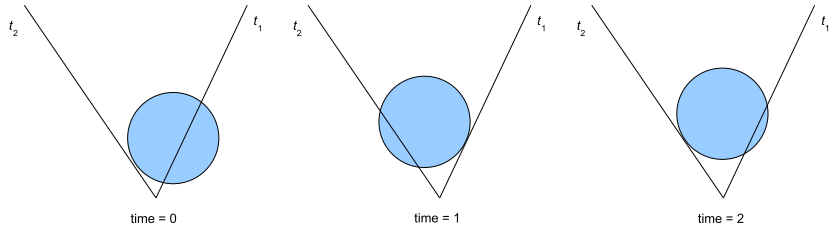

Figure 7.5: Following a stuck particle during three time steps.

7.4 Visualization

In Chapter 2 we discussed the visual aspects of contrast fluid in fluoroscopy. Since the fluid itself is very much nontranslucent and appears as white in an black and white 2D medical image, we only render the reconstructed fluid surface without shading effects. The only thing that affects the nontranslucency of the contrast fluid is the thickness of the fluid at a given pixel. A way of visualizing this would be to reconstruct two surfaces, one back-surface and one front-surface. These are then rendered separately into two depth maps. The value for a given pixel is then given by the difference of the two depth values from the depth maps. However, the main problem solved is the surface reconstruction of a point cloud. To give a nice visual appearance of the fluid surface, we've incorporated shading effects such as refraction and reflection.

7.4.1 Surface Reconstruction

The surface refinement algorithm discussed in Section 6.2.1 has already been evaluated at VRLab and other methods were examined. We implement Müller's screen space meshes mentioned in section 6.1.2. The implementation includes the reconstruction of the fluid surface but not the extensions mentioned such as depth smoothing and silhouette smoothing. The visual result can be seen in Figure 8.2.

7.4.2 Particle Trails

The contrast fluid used in fluoroscopy has the same property as dye, it sticks on a surface. We've implemented a way of simulating this called particle trails. When a collision between a triangle and particle occurs, the collision library gathers all information needed to resolve the collision. In addition to this, it also includes a reference to the triangle. Assume that the texture coordinates $\mathbf{u}_i \in \mathbb{R}^2$ for each vertex $\mathbf{v}_i \in \mathbb{R}^3$ in the triangle can be extracted from this information, then the texture coordinate $\mathbf{u}_p$ for the

contact point **p** can be computed. This texture coordinate can then be used to render a texture splat onto a texture.

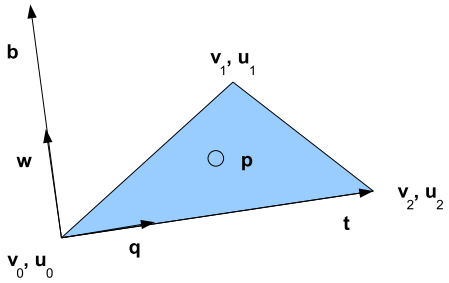

Figure 7.6: Illustration of tangent space. The triangle normal is pointing out of the paper towards the reader.

A TBN (tangent, bi-normal, normal) matrix is used to transform coordinates from world space to tangent space. There is no need to include the normal in the computations since the contact point is already determined to be on the triangle plane. We only construct the tangent and bi-normal:

$$\mathbf{t} = \mathbf{v}_2 - \mathbf{v}_0 \tag{7.7}$$

$$\mathbf{b} = \mathbf{n} \times \mathbf{t} \tag{7.8}$$

where **n** is the triangle normal. We then construct two orthogonal vectors in tangent space called **q** and **w** using Gram-Schmidt orthogonalization:

$$\mathbf{q} = \mathbf{u}_2 - \mathbf{u}_0 \tag{7.9}$$

$$\mathbf{w} = (\mathbf{u}_1 - \mathbf{u}_0) - (\mathbf{u}_1 - \mathbf{u}_0) \cdot \left(\frac{\mathbf{q}}{|\mathbf{q}|} \cdot \mathbf{q} \right) \tag{7.10}$$

where both **q** and **w** is normalized. The texture coordinates for the contact point **p** is then computed as:

$$\mathbf{u_p} = \mathbf{u_0} + \mathbf{q} \cdot \frac{\mathbf{p} \cdot \mathbf{t}}{|\mathbf{t}|} + \mathbf{w} \cdot \frac{\mathbf{p} \cdot \mathbf{b}}{|\mathbf{b}|} \tag{7.11}$$

Every time a collision occurs, the texture coordinate for the contact point is computed and stored in a list to be used in the rendering stage. Before rendering the final scene, a large textured quad is rendered. The texture used on the quad is the same texture used by the model. Each texture coordinate stored in the list is sent as a 2D vertex to the graphics pipeline. A shader renders a colored circle on the texture at the given texture coordinate. The finished texture is then detached from the frame buffer object and used as a texture when rendering the model. The list of texture coordinates is then cleared and ready to be filled in the next time frame. The only additional computational cost when using this method is the cost of computing the texture coordinate for each contact point and the extra rendering pass that renders the splats onto the texture using frame buffer objects in OpenGL. This approach is better than decal methods where each

contact point is represented with a quad placed on the surface and rendered with a small depth offset. In order to see splats that occured several time steps back in time, one would need to keep all quads and this would result in a large number of vertices needed to be sent to the GPU. The only disadvantage is that when a texture appears stretched on the surface, the splat will also appear streched instead of a symmetrical circle. However, this is not a problem if the model is correctly textured.

7.4.3 Surface Rendering

One advantage of the surface reconstruction method used is that we have a 3D triangle mesh that with ease can be sent to the standard graphics pipeline and already implemented shaders can be used with having to implement new ones. We can achieve per-pixel shading effects such as diffuse and specular shading, refraction and fresnel reflection.

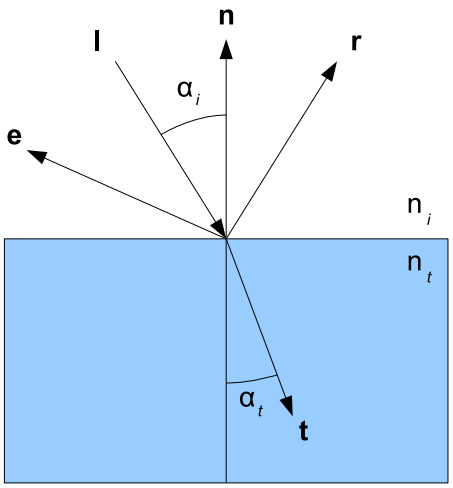

Figure 7.7: A simple reflection and refraction light model. **l** is the incoming light, **e** is the viewing direction, **n** is the surface normal, **r** is the reflected light and **t** is the transmitted light.

The diffuse term is computed by:

$$I_{diffuse} = L_{diffuse} \cdot M_{diffuse} \cdot (\mathbf{n} \cdot \mathbf{l}) \tag{7.12}$$

where $L_{diffuse}$ and $M_{diffuse}$ are the light's diffuse color and material's diffuse coefficient respectively. The specular component which describes the shininess of the surface is computed by:

$$\mathbf{r} = 2\mathbf{n}(\mathbf{l} \cdot \mathbf{n}) - \mathbf{l} \tag{7.13}$$

$$I_{specular} = L_{specular} \cdot M_{specular} \cdot (\mathbf{r} \cdot \mathbf{e})^s \tag{7.14}$$

where s is the shininess value, $L_{specular}$ is the light's specular intensity and $M_{specular}$ is the material's specular coefficient. This yields a surface more similar to clay and contrast fluid than a transparent fluid like water.

When rendering a transparent surface we need to consider refraction and reflection. When light passes through a boundary between two different media such as air and water, the angle of incidence α_i and refraction α_t differ. This relationship can be described by Snell's Law:

$$n_i \sin(\alpha_i) = n_t \sin(\alpha_t) \tag{7.15}$$

where n_i and n_t are the indices of refraction. The amount of reflection when observing water depends on the viewing angle. This is known as the Fresnel reflection which describes the probability that a photon is reflected at the surface. We implement an approximation of the fresnel reflection:

$$R_{fresnel} = (1 - (\mathbf{e} \cdot \mathbf{n}))^k \tag{7.16}$$

We perform the following steps when rendering a scene with reflection, refraction and fresnel effect. We render the surface into the stencil buffer, stencil values are set to 1 if pixel is written. We then render the rest of the scene using the previous stencil buffer, only accepting pixels where the stencil value is 1. The color values are rendered into a texture. This texture is then used in a shader program to retrieve the refraction color. The reflection color for a given pixel is sampled from a cube map. The final color of a pixel is then blended:

$$I_{final} = I_{refraction} \cdot R_{fresnel} + I_{reflection} \cdot (1 - R_{fresnel})$$

Chapter 8

Results

In this chapter the results of the implemented simulation will be discussed. The presented results were achieved running on a Intel Core2 1.8Ghz system with 1024MB of memory and a nVidia 7600 GS 256MB graphics card. At first, no code optimization was used when compiling the source code and plausable results were achieved. Enabling SSE and SSE2 code optimization resulted in a 20% speed gain.

All of the required functionality listed in Section 2.1 were fulfilled. The deformable colon was not implemented but some potential methods were examined in Chapter 5.

8.1 Performance

The results Müller present in [34] show that 2200 particles can be simulated running at 20 frames per second. They visualize the fluid using a point splatting technique. In comparison with our implementation we achieve quite similar results on a similar system, however Müller does not include collision against a triangle mesh. Some optimizations can be introduced when rendering the fluid surface. Our implemention renders the triangles and vertices using several gl funtions calls such as glBegin(...). Replacing these with vertex buffers can result in a performance increase when uploading the mesh data to the GPU. This can also be avoided by implementing the surface generation on the GPU thus balancing CPU-GPU load and increasing the performance of the simulation.

In Figure 8.1 a performance chart is presented showing the differences between spatial hashed grids and BVH. The results obtained verify that spatial hashed grids would be a better candidate than BVH when representing a static trimesh. We can see that spatial hashed grids perform slightly better than BVH as data structure for the trimesh.

8.2 Visual

The visual result from the method described in Section 6.1.2 can be seen in Figure 8.2, 8.4 and 8.5. The visual comparison with other point cloud rendering techniques shows that the method used for this simulation was satisfactory. The easiest way to visualize the contrast fluid is to render the surface as white without any shading.

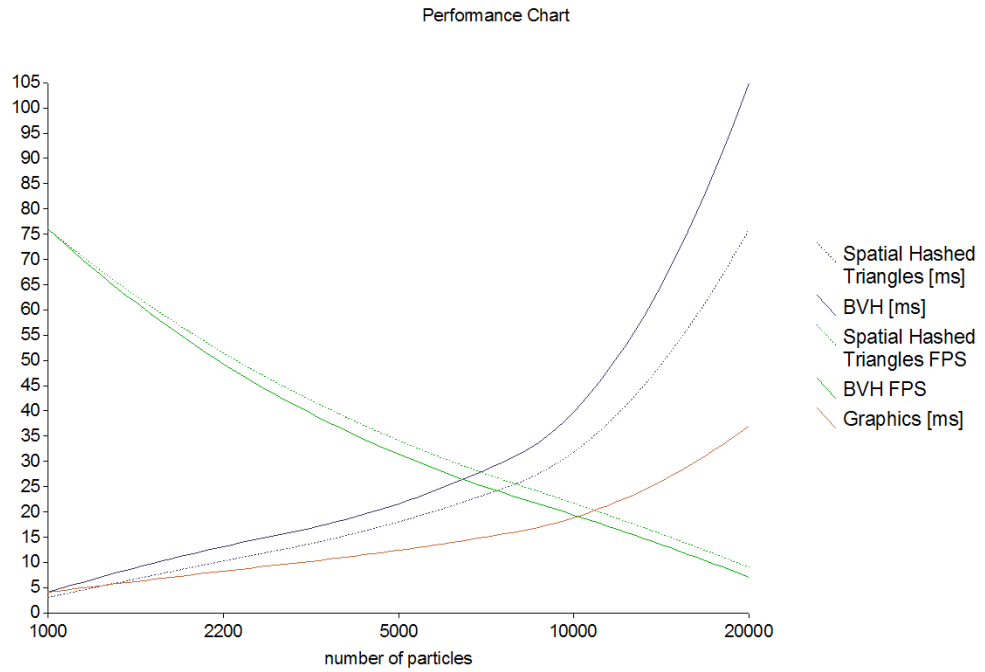

Figure 8.1: A performance chart showing the performance differences between BVH structure and spatial hashed grids. The model used contained 500 triangles and a maximum of 40 neighbours per particle was allowed. The update time measured in milliseconds clearly decreases as the number of particles used increases, directly affecting the number of frames per second.

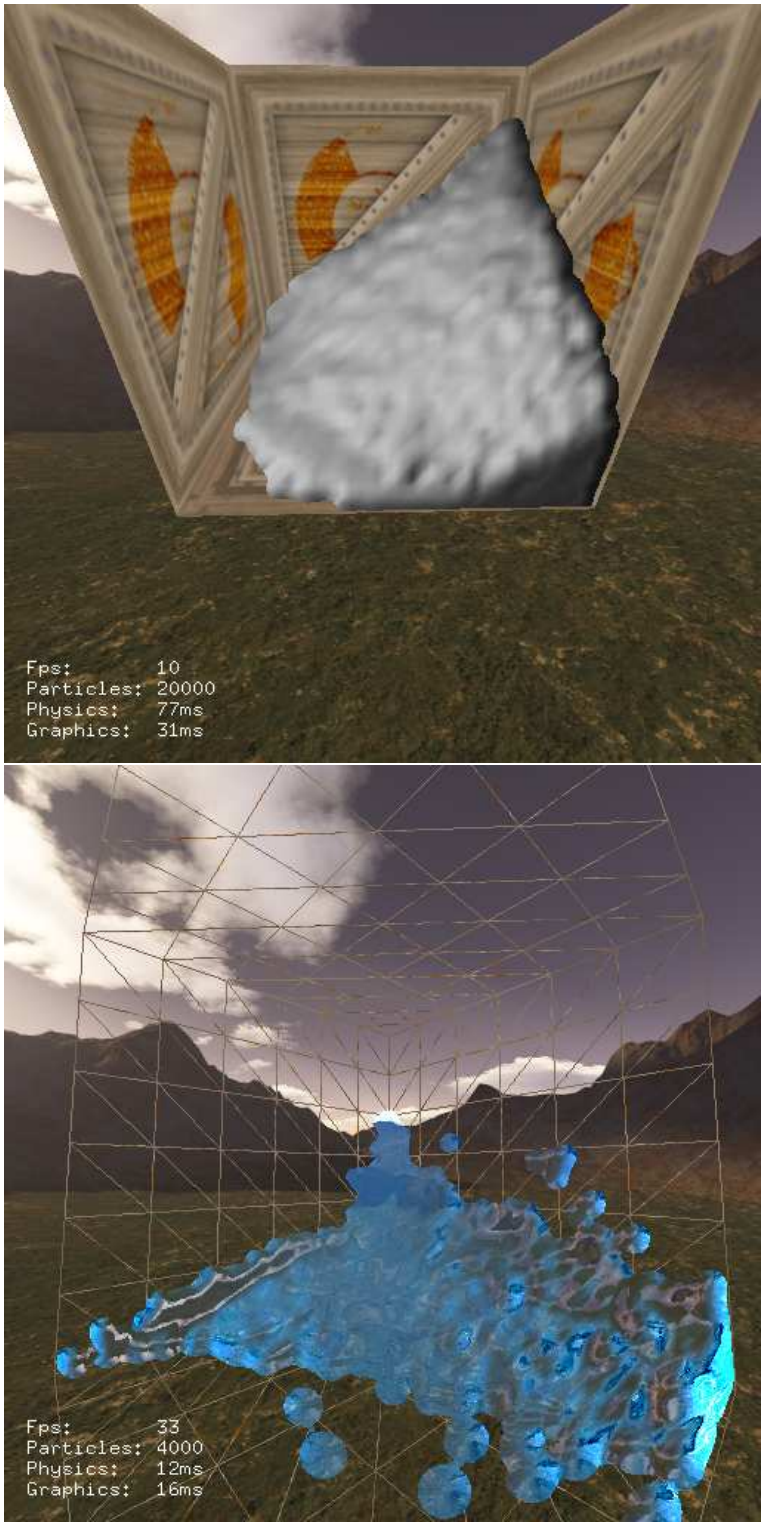

Figure 8.2: Top: The extracted surface is rendered using a simple light model to better show the structure of the surface. Bottom: The extracted surface is rendered with fresnel reflection and refraction.

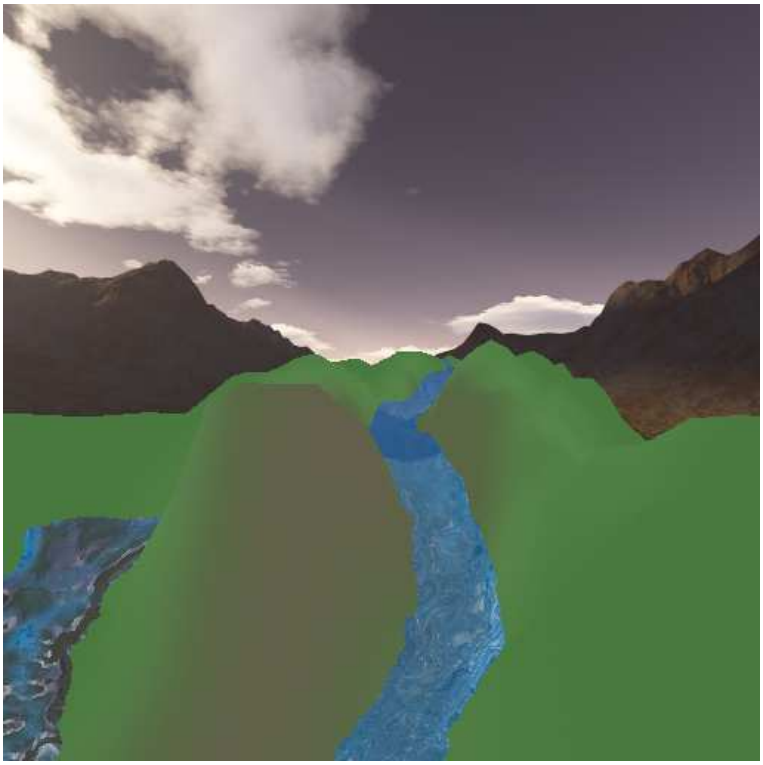

Figure 8.3: Screen capture from a river simulation of 3000 particles on a triangle landscape.

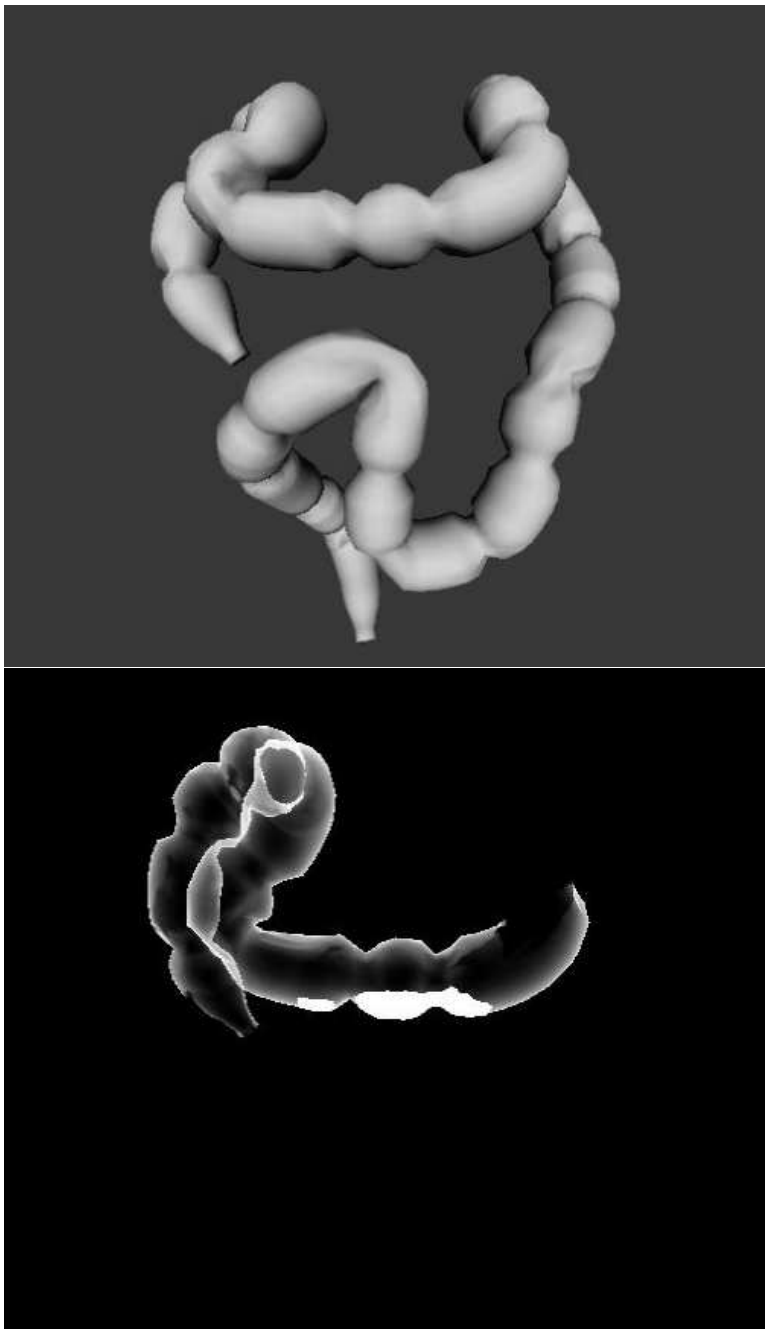

Figure 8.4: Two screen captures from a simulation of 4096 particles inside a colon consisting of 8040 triangles. The upper image shows the colon while the lower image shows an x-ray image of the colon partially filled with contrast fluid.

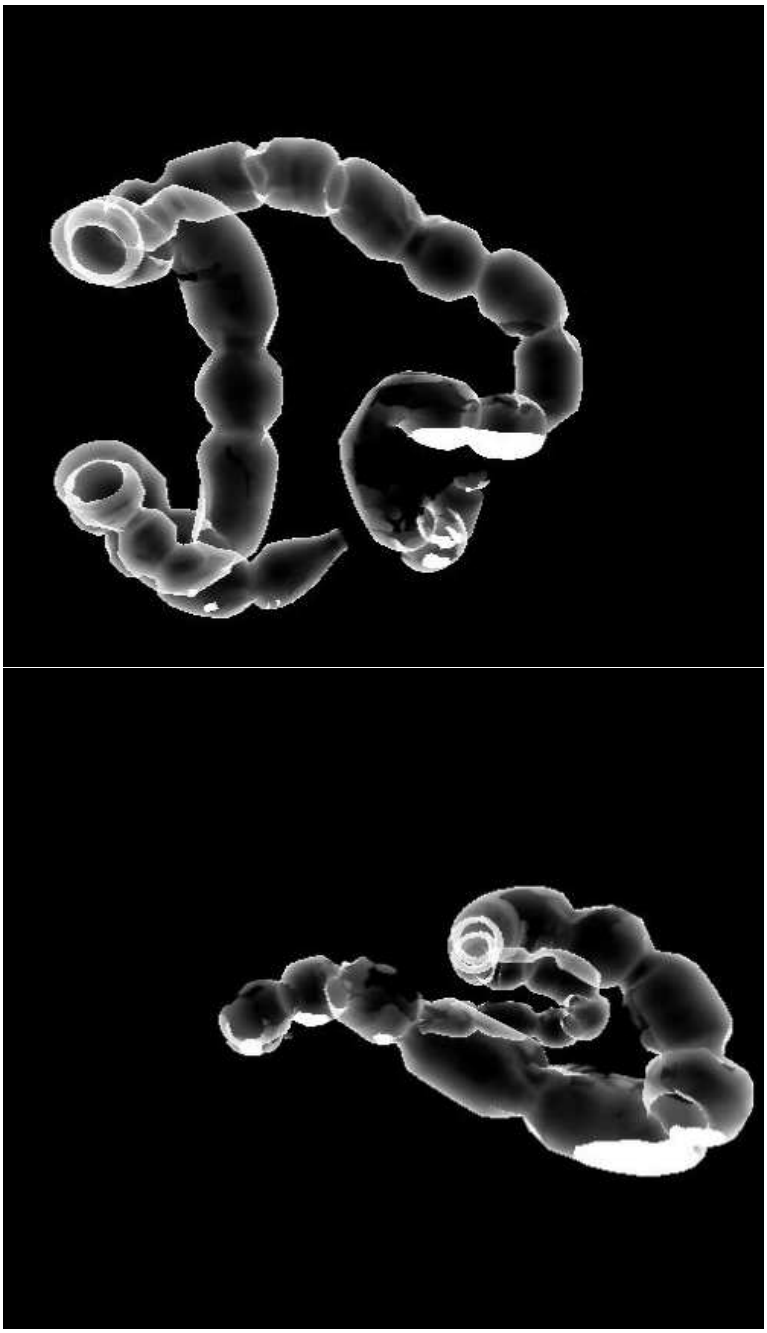

Figure 8.5: Two other screenshots from the same simulation as in Figure 8.4.

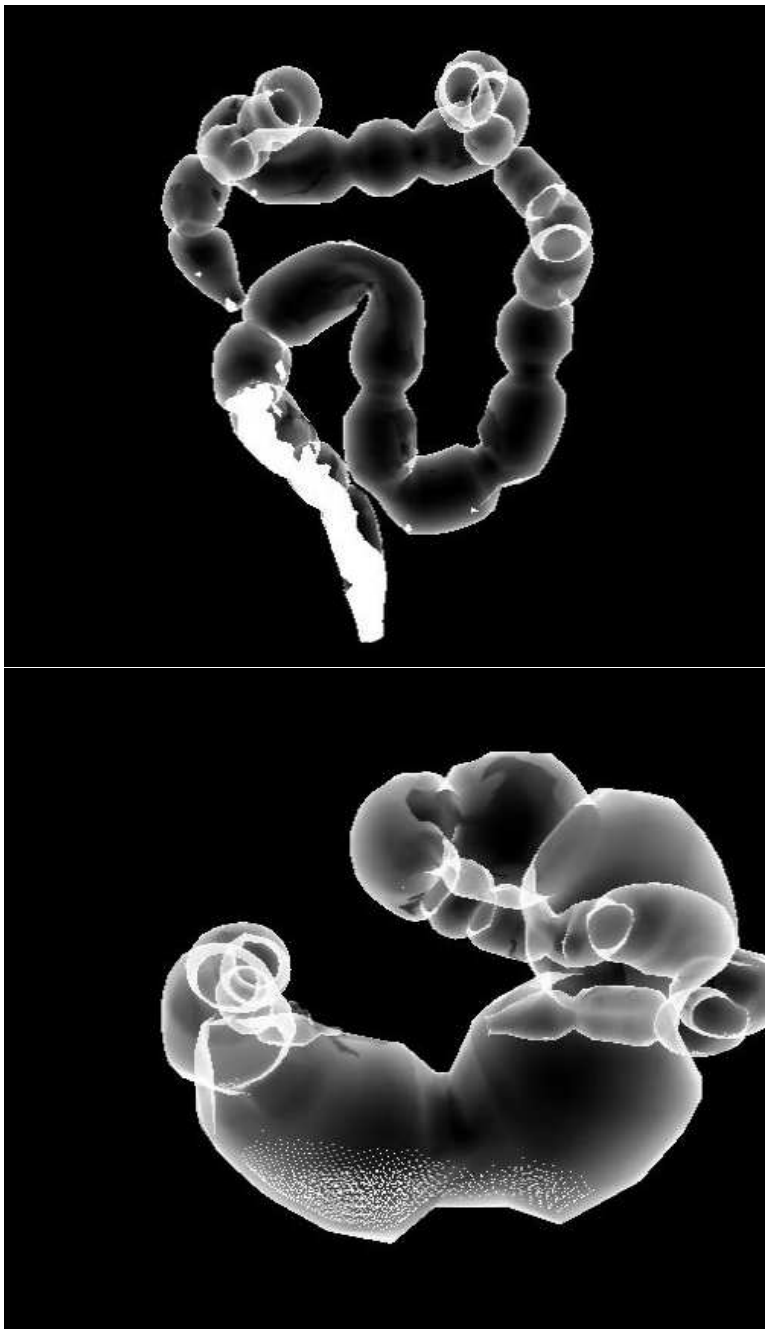

Figure 8.6: The last two screen captures from the simulation mentioned in Figure 8.4. The lower image shows the particles used to simulate the contrast fluid.

Chapter 9

Conclusions and Future Work

We have shown that SPH is a good candidate for simulating contrast fluid. Collision detection using a spatial hashed grid has been proven to be a better candidate than BVH's for static triangle meshes. However, the implemented system is limited by the number of triangles in the model. The original model of the colon consisted of $40,000$ triangles which is a large amount of data. Using this model would store too many triangles in each hash bucket and a lot of intersection tests would be needed. Increasing the number of hash buckets would consume more memory and little gain in speed. The construction of a BVH of this high resolution model gives us a huge tree and traversing this tree with a system of thousands of particles would not be possible in an interactive simulation.

We have found that the importance of a detailed model of the colon overweight a deformable colon. To preserve the detail of the high resolution model we suggest using 3D signed distance fields. Although this would limit the use to a rigid colon only, it would preserve details better and increase the simulation speed since each collision query between a particle and the distance field is of $O(1)$ complexity.

Harada et al. [42] found a way to perform inter-particle collision detection on the GPU and has shown interesting results. They've implemented regular grids described in Section 4.1.1 on the GPU and can perform neighbour search and force computations of $16,386$ particles in 14.8ms on a high-end computer. They perform collision detection with the environment by storing a 3D signed distance field in a 3D texture. One drawback of the method is that each cell in the regular grid is limited to contain a maximum of four particles due to the texture format (one particle per color channel) on today's graphics card. Although this could be solved by performing two or more passes, storing the regular grid in more than one texture, it would double the number of vertex texture fetches and the simulation would become slower.

Early implementations confirms that it would be faster to perform neighbour search using the GPU other than the CPU. Collision detection against the triangle mesh still needs to be performed on the CPU and therefore requires a read-back from the GPU. This is a bottle-neck on today's graphics card and should be avoided. Maybe gaussian boundary particles described in Section 4.4.4 can be implemented on the GPU. By coupling these boundary particles to triangles, the entire simulation could be run on the

GPU and the read-back could be avoided. However, the surface area of the colon requires approximately 12.000 particles to be completely covered when separated by their radius and this is only one layer. A minimum of three layers are needed to ensure that no particles escape.

Maybe it would be better to procedurally generate the colon from a parametric cubic spline, where each control point also has a scalar describing the colon radius. This could also benefit the students using the simulator since no colon are alike and each run of the simulation would generate a different colon. Performing collision detection against the parametric cubic spline could maybe even be implemented on the GPU but this needs to be investigated further. Still the problem remains of visualizing the coating of the colon walls.

Furthermore, the prototype needs to be integrated with a volume renderer that can visualize x-ray images from MRI (magnetic-resonance-imaging) and CT (computer-tomography) scans. The possibilities of a deformable triangle mesh seems to be a few years ahead. As the computational power of today's multi-core CPU's and GPU's increases and new algorithms are discovered it will become possible to simulate a coupled particle system of thousands of particles interacting with a deformable mesh while renderering an accurate x-ray image in real-time.

Chapter 10

Acknowledgements

I would like to thank my supervisor Kenneth Bodin at VRlab, Umeå University for his input and thoughts. I would also like to thank Anders Backman at VRlab, Umeå University for helping me with the rendering part. Other people I would like to thank is Jan Ahlqvist, Tore Nilsson and Magnus Johansson at the Department of Diagnostic Oral Radiology for this interesting project and the opportunity to visit Stanford University and meet great people.

I would also like to thank Mattias Linde for the great cooperation on the physics library used in both our projects.

References

[1] D. Sjölie A. Backman, K. Bodin. Colosseum 3d. http://www.vrlab.umu.se/research/colosseum3d/.

[2] M. Müller A. Nealen. Physically based deformable models in computer graphics. *Computer Graphics Forum*, 2005.

[3] T. Amada. Real-time particle-based fluid simulation. http://www.ss.iij4u.or.jp/ amada/fluid/ (visited 2007-04-13.

[4] C. Crassin. Opengl geometry shader marching cubes. http://www.icare3d.org/content/view/50/9/ (visited 2007-08-02).

[5] K.I. Joy C.S. Co, B. Hamann. Iso-splatting: A point-based alternative to isosurface visualization. *Proceedings of the 11th Pacific Conference on Computer Graphics and Applications*, 2003.

[6] A. Witkin D. Baraff. Large steps in cloth simulation. *Proceedings of SIGGRAPH 1998*, 1998.

[7] R. Fedkiw D. Enright, S. Marschner. Animation and rendering of complex water surfaces. *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, 2002.

[8] R. Lobb D. Nixon. A fluid-based soft-object model. *IEEE Computer Graphics and Applications*, 2002.

[9] S.S. Keerthi E.G. Gilbert, D.W. Johnson. A fast procedure for computing the distance between complex objects in three-dimensional space. *IEEE Journal of Robotics and Automation*, 1988.

[10] K. Erleben et al. *Physics-Based Animation*. 2005.

[11] M. Botsch et al. High-quality surface splatting on today's gpus. *Eurographics Symposium on Point-Based Graphics*, 2005.

[12] M. Carlson et al. Melting and flowing. *Proceedings of the ACM SIGGRAPH symposium on Computer animation*, 2002.

[13] M. Müller et al. Stable real-time deformations. *Proceedings of the 2002 ACM SIGGRAPH*, 2002.

[14] M. Müller et al. Interaction of fluids with deformable solids. *Journal of Computer Animation and Virtual Worlds*, 2004.

[15] M. Müller et al. Point based animation of elastic, plastic and melting objects. *Proceedings of the 2004 ACM SIGGRAPH*, 2004.

[16] M. Müller et al. Meshless deformations based on shape matching. *Proceedings of SIGGRAPH'05*, 2005.

[17] M. Müller et al. Particle-based fluid-fluid interaction. *Proceedings of SIGGRAPH'05 Symposium on Computer Animation*, 2005.

[18] M. Müller et al. Position based dynamics. *3rd Workshop in Virtual Reality Interactions and Physical Simulation*, 2006.

[19] M. Pauly et al. Meshless animation of fracturing solids. *Proceedings of ACM Siggraph 2005*, 2005.

[20] M. Teschner et al. Optimized spatial hashing for collision detection of deformable objects. *Proc. Vision, Modeling, Visualization*, 2003.

[21] M. Teschner et al. A versatile and robust model for geometrically complex deformable solids. *Eurographics 2004*, 2004.

[22] R. Keiser et al. A unified lagrangian approach to solid-fluid animation. *Proceedings of the Eurographics Symposium on Point-Based Graphics 2005*, 2005.

[23] T. Takahashi et al. The simulation of fluid-rigid body interaction. *ACM Siggraph Sketches & Applications*, 2002.

[24] M. Desbrun G. Debunne. Dynamic real-time deformations using space & time adaptive sampling. *Computer Graphics Proceedings*, 2001.

[25] M. Zwicker H. Pfister. Surfels: Surface elements as rendering primitives. *Siggraph 2000*, 2000.

[26] P.M. Hubbard. Collision detection for interactive graphics applications. *IEEE Transactions on Visualization and Computer Graphics 1*, 1995.

[27] P. Hunter. Fem/bem notes. http://www.bioeng.auckland.ac.nz/cmiss/fembemnotes/fembemnotes.pdf, 2005.

[28] H. Aanaes J.A. Baerentzen. Signed distance computation using the angle weighted pseudo-normal. *IEEE Transactions on Visualization and Computer Graphics*, 2005.

[29] J.K. Hodgins J.F. O'Brien. Graphical modeling and animation of brittle fracture. *Proceedings of SIGGRAPH 1999*, 1999.

[30] W.E. Lorensen. Marching cubes: A high resolution 3d surface construction algorithm. *Siggraph 1987*, 1987.

[31] L.B. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 1977.

[32] A.H. Barr M. Desbrun, P. Schrder. Interactive animation of structured deformable objects. *Graphics Interface '99*, 1999.

[33] M.P. Cani M. Desbrun. Smoothed particles: A new paradigm for animating highly deformable bodies. *Computer Animation and Simulation '96*, 1996.

[34] M. Gross M. Müller, D. Charypar. Particle-based fluid simulation for interactive applications. *Proceedings of ACM SIGGRAPH Symposium on Computer Animation*, 2003.

[35] S. Duthaler M. Müller, S. Schirm. Screen space meshes. *Eurographics/ ACM SIGGRAPH Symposium on Computer Animation 2007*, 2007.

[36] J.P. Morris. Simulating surface tension with smoothed particle hydrodynamics. *International Journal for Numerical Methods in Fluids*, 2000.

[37] R. Fedkiw N. Foster. Practical animation of liquids. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001.

[38] J.J. Monaghan R.A. Gingold. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Monthly Notices of the Royal Astronomical Society*, 1977.

[39] W.T. Reeves. Particle systems - a technique for modeling aclass of fuzzy objects. *ACM Transactions on Graphics 2*, 1983.

[40] J. Stam. Stable fluids. *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, 1999.

[41] J.F. O'Brien T. Goktekin, A.W. Bargteil. A method for animating viscoelastic fluids. *Proceedings of ACM Siggraph 2004*, 2004.

[42] Y. Kawaguchi T. Harada, S. Koshizuka. Smoothed particle hydrodynamics on gpu. *Siggraph 2007*, 2007.

[43] T. Akenine-Möller T. Larsson. Collision detection for continuously deforming bodies. *Eurographics 2001*, 2001.

[44] D. Tonnesen. *Dynamically Coupled Particle Systems for Geometric Modeling, Reconstruction and Animation.* PhD thesis, University of Toronto, 1998.

[45] Unknown. Gaussian quadrature. http://en.wikipedia.org/wiki/Gaussian_quadrature.

[46] Marcus Vesterlund. Simulation and rendering of a viscous fluid using smoothed particle hydrodynamics. Master's thesis, Umeå University, Umeå, Sweden, 2004.