Karlsruhe Institute of Technology

Unified Physics Simulation of Variable Sized Particles Generated on Demand

Masterarbeit-Ausarbeitung von

Chao Jia

An der Fakultät für Informatik
Institut für Visualisierung und Datenanalyse,
Lehrstuhl für Computergrafik

September 16, 2017

Erstgutachter:	Prof. Dr.-Ing. Carsten Dachsbacher
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch
Betreuender Mitarbeiter:	Emanuel Schrade
Bearbeitungszeitraum:	17. März 2017– 16. September 2017

Contents

1. Introduction

Animation of many visually interesting physical phenomena such as rigid bodies, deformable objects (e.g. rubber, flesh), shells (e.g. cloth, paper), rods (e.g. strands, hair), gases, and liquids is of significant importance for movie special effects, commercials, computer games and other interactive systems. It is very difficult to model these phenomena by hand given the numerous degrees of freedom, and that the human visual system is capable of easily spotting any physical irregularities caused by modeling errors.

Physics-based animation can help to relieve artists of innumerable tedious work and allow them to concentrate on creative things such as characters, story and aesthetics. Simulated physics is usually cheaper, safer and free of some limitations of directly capturing real motion, e.g. "impossible" scenarios, fictional objects and exotic materials.

1.1 Motivation

Physical simulation has been well studied in computational science to reproduce the real-world experiments, gain novel insights into physical systems and verify theoretical assumptions. To keep the simulation as accurate as possible, many methods have been developed and some of them are successfully introduced into computer graphics.

Significant breakthroughs have been made in many key areas of physics-based simulation of deformable objects such as object modeling, fracture, plasticity, cloth simulation, stable fluid simulation, time integration strategies, discretization and numerical solutions of PDEs, multi-resolution modeling etc. Apart from the issue of instability, many of these techniques are rather sophisticated, because the simulation of deformable objects is an inter-discipline involving Newtonian dynamics, continuum mechanics, numerical computation, differential geometry, vector analysis, approximation theory and computer graphics [Nealen et al., 2006]. Therefore, real-time performance can hardly be achieved.

Generally the interactivity, visual plausibility and controllability are the most important factors for applications in real-time computer graphics, while the physical accuracy is of secondary interest, and in some cases even irrelevant. To serve this purpose, position-based approach has been proposed by Müller et al. [2007]. It is simple, unconditionally stable and capable of creating plausible visual effects, therefore has gained popularity recently.

The programmable Graphics Processor Unit or GPU is specialized for compute-intensive, highly parallel computation. Driven by the insatiable market demand, GPU has evolved into a highly parallel, multithreaded, manycore processor with strong floating-point capability and very high memory bandwidth [NVIDIA, 2017a].

By exploiting the tremendous computational horsepower of GPUs, Macklin et al. [2014] presented a unified dynamic framework based on position-based dynamics (PBD) for real-time visual effects, which can simulate scenes of tens of thousands particles connected by constraints for different kinds of natural phenomena including gases, liquids, deformable solids, rigid bodies and cloth with two-way interactions.

In this master project we based our work on the unified dynamic framework, and focused on utilizing GPU to make further improvement and to remove some limitations of the unified solver.

1.2 Structure

In **Chapter** 2, we give an overview of related work in position based dynamics and important background concepts.

In **Chapter** 3, we detail the formulation of the algorithms used in the simulation framework, and some concrete constraints for common phenomena.

In **Chapter** 4, we focus on the algorithms related to rigid bodies, including the constraint for restoring the shape, collision handling, and particle generation for rigid bodies.

In **Chapter** 5, we cover a few methods for accelerating the collision detection in our solver.

In **Chapter** 6, we describe our implementation and present the performance and visual results of the implementation.

In **Chapter** 7, we conclude our work based on the results, along with a brief discussion of the limitations of the methods we used in our project and areas for future work.

2. Related Work

2.1 Mass-Spring System

The mass-spring system is one of the most simple physically-based models, thus the most likely to achieve real-time performance. In a mass-spring system, deformable bodies are approximated by a set of masses linked by springs in a fixed topology. It is easy to implement, highly parallelizable, and involves few computations [Desbrun et al., 1999].

A simple and intuitive integration scheme to animate the mass-spring system is the *explicit Euler method* defined by

$$\begin{aligned} \mathbf{v}_i^{n+1} &= \mathbf{v}_i^n + \mathbf{F}_i^n \frac{\mathrm{d}t}{m}, \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^n \mathrm{d}t \end{aligned} \tag{2.1}$$

where $\mathbf{v}_i$ is the velocity of the mass point i of mass m at position $\mathbf{x}_i$, and $\mathbf{F}_i$ is the internal force due to spring acting on it. Although the implementation of the explicit Euler method is simple, the system can achieve stability only if the time step is very small, as being noted by Desbrun et al. [1999]. Therefore they used the follwing *implicit Euler integration*

$$\begin{aligned} \mathbf{v}_i^{n+1} &= \mathbf{v}_i^n + \mathbf{F}_i^{n+1} \frac{\mathrm{d}t}{m} \\ \mathbf{x}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^{n+1} \mathrm{d}t \end{aligned} \tag{2.2}$$

in their mass-spring system to simulate deformable objects. $\mathbf{F}^{n+1}$ is computed using the first order approximation

$$\mathbf{F}^{n+1} = \mathbf{F}^n + \frac{\partial \mathbf{F}}{\partial \mathbf{x}} \Delta^{n+1}\mathbf{x}, \tag{2.3}$$

where $\Delta^{n+1}\mathbf{x} = \mathbf{x}^{n+1} - \mathbf{x}^n = (\mathbf{v}^n + \Delta^{n+1}\mathbf{v})\mathrm{d}t$.

The implicit scheme will not give rise to instabilities even for large time steps. However, for very large time steps or high stiffness, the simulation behaves implausible. To eliminate large stretch, they used a straightforward post-step modification of mass points, i.e. the *position* of one point will be moved directly whenever the other of the two mass points is constrained at a given position and the spring exceeds a predefined normalized threshold.

There are some significant drawbacks of the mass-spring method ([Bender et al., 2015]):

- The behavior of the object depends on concrete setup of the spring network;
- The optimal parameters for the springs are difficult to be determined to prevent severe penetration and meanwhile to guarantee the stability [Jakobsen, 2001];
- Volumetric effects such as volume conservation cannot be simulated directly by mass-spring systems.

All of these problems can be solved by taking the entire volume of a solid into consideration.

2.2 Position Based Dynamics

Jakobsen [2001] proposed a simple, fast and stable method to simulate cloth for games and interactive use, which works immediately on positions instead of first updating the velocities and then the positions. He chose a velocity-less representation for the particle system, which stores the current position $\mathbf{x}_i^n$ and its previous position $\mathbf{x}_i^{n-1}$ for each particle i, instead of the position and velocity of each particle. To update the positions, he used *Verlet integration*:

$$\mathbf{x}_i^{n+1} = 2\mathbf{x}_i^n - \mathbf{x}_i^{n-1} + \frac{\mathbf{F}_i^n}{m}\mathrm{d}t^2 \tag{2.4}$$

Verlet integration is not always accurate and may lead to dissipation of the system, but it is fast and stable, because the position is directly computed from the force, rather than via the velocity. Collision and contact handling are resolved by projection, i.e. offending points are directly projected out of the obstacle. *Projection* means the movement of an offending point should be as little as possible, usually in the direction perpendicular to the collision surface. Concurrent constraints are solved by relaxation, namely all local constraints are consecutively solved in each of the solver iterations. The system will converge to a global configuration in which all constraints are satisfied at the same time if the conditions are physically consistent. He focused mainly on distance constraints, and only gave a simplified example for rigid bodies without a systematic method.

Müller et al. [2005] introduced a geometrically motivated mesh-less deformation model which significantly simplifies the preprocessing stage compared to modal analysis models which commonly require the incorporation of additional auxiliary object representations. In contrast to standard shape matching approaches which are primarily concerned with establishing the correct correspondences between two shape representations, the problem in their deformation model is to find least squares optimal rigid transformations between the current state and the rest state of the object with the correspondences being a priori known. The points are moved towards certain goal positions calculated from the optimal transformations.

By integrating different kinds of constraints into a unified solver, Müller et al. [2007] formulated the general position based approach, namely the *Position-Based Dynamics (PBD)*. They gave a general strategy for constraint projection, and proved it to be able to preserve linear and angular momenta for internal constraints, which is an important feature in animation because any error here would directly affect the motion in an awkward way [Desbrun et al., 1999], i.e. ghost forces will be introduced which act like external forces dragging and rotating the object. They implemented cloth simulation and two-way interactions with rigid bodies using the Position-Based Dynamics framework.

Deul et al. [2016] extended position-based dynamics to rigid bodies. They introduced three additional parameters for rigid bodies – orientation, angular velocity and inertia tensor apart from translational motion parameters – position, velocity and mass. They used a simple and intuitive concept of connectors [Witkin et al., 1990] to formulate constraints, so that generic constraints can be formulated without knowledge about the body itself [Bender et al., 2015]. Similar to the PBD framework presented by Müller et al. [2007], collision constraints are generated from scratch at each time step. The collision between two rigid bodies is resolved by moving the bodies in opposite direction along the intersection normal. Alternatively, shape matching constraints can be formulated for rigid bodies by setting their stiffness to one [Müller et al., 2005].

2.3 Fluid Simulation

Fluid simulation is a common building block of a variety of natural phenomena such as water, smoke, cloud and fire, and thus one of the most intriguing problems in computer graphics.

In fluid dynamics it is common to assume real fluids such as water and air to be incompressible and homogeneous. A fluid is incompressible if the volume of any subregion of the fluid is constant over time. The density ρ of a homogeneous fluid is constant in space. The state of fluids can be described by the *incompressible Navier–Stokes equations*:

$$\frac{\partial \mathbf{u}}{\partial t} = -(\mathbf{u} \cdot \nabla)\mathbf{u} - \frac{1}{\rho}\nabla p + \nu \nabla^2 \mathbf{u} + \mathbf{F}, \tag{2.5}$$

$$\nabla \cdot \mathbf{u} = 0, \tag{2.6}$$

where $\mathbf{u}(\mathbf{x}, t)$ is the velocity field, $p(\mathbf{x}, t)$ is the scalar pressure field, ν is the kinematic viscosity, $\mathbf{F}(\mathbf{x}, t)$ is the external forces that act on the fluid, and $\mathbf{x}$ is the spatial coordinates [Harris, 2005].

One approach to numerically solve the Navier–Stokes equations is to discretize the region through which the fluid flows into computational grid cells fixed in space throughout the simulation, known as *Eulerian scheme*, which is suitable for simulating fluids using the graphics pipeline, where the grid cells can be directly mapped to voxels of a 3D texture [Crane et al., 2007]. However, Eulerian scheme can be very memory intensive to animate arbitrary fluid motion. In contrast to Eulerian scheme, *Lagrangian schemes* discretize the fluids into particles, and the particles move with the fluids during simulation.

It was believed that physically-based fluid simulation was too expensive for real-time applications, because explicit integration schemes only allow very small time-steps. Stam [1999] presented the first implicit integration scheme to solve the Navier-Stokes equations for physically-based fluid simulation using Eulerian approach and achieved plausible visual effect in real-time and stability for large time-steps. His method is not accurate enough for engineering applications due to "numerical dissipation", i.e. the flow tends to dampen more rapidly than natural fluids, which, however, makes it easy to control in computer graphical applications.

Müller et al. [2003] proposed a particle-based Lagrangian method based on *Smoothed Particles Hydrodynamics (SPH)* [Lucy, 1977] to simulate fluids with free surfaces. SPH is an interpolation method to approximate the value of a scalar field A at particle i positioned at $\mathbf{x}_i$ using its local neighborhood with radius h. According to SPH,

$$A(\mathbf{x}_i) = \sum_j m_j \frac{A(\mathbf{x}_j)}{\rho_j} W(\mathbf{r}_{ij}, h) \tag{2.7}$$

Where m_j is the mass of particle j, ρ_j is the density, $\mathbf{r}_{ij} = \mathbf{x}_i - \mathbf{x}_j$ and $W(\mathbf{r}_{ij}, h)$ is the *smoothing kernel* with core radius h. To obtain a good approximation, the kernel must be normalized such that

$$\int W(\mathbf{x} - \mathbf{x}', h) \mathrm{d}\mathbf{x}' = 1, \tag{2.8}$$

where $\mathrm{d}\mathbf{x}'$ is a differential volume element, or in the discrete form (Bodin et al. [2012])

$$\sum_j m_j \frac{1}{\rho_j} W(\mathbf{r}_{ij}, h) = 1. \tag{2.9}$$

Derivatives of the field quantities appeared in fluid equations can be conveniently evaluated using SPH since derivatives only affect the smoothing kernel. The gradient of A is

$$\nabla A(\mathbf{x}_i) = \sum_j m_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r}_{ij}, h) \tag{2.10}$$

and the Laplacian of A is

$$\nabla A^2(\mathbf{x}_i) = \sum_j m_j \frac{A_j}{\rho_j} \nabla^2 W(\mathbf{r}_{ij}, h) \tag{2.11}$$

The viscosity and pressure force fields are derived directly from the Navier-Stokes equation using SPH and surface tension forces are modeled explicitly based on the ideas of Morris [2000].

Instead of using Navier-Stokes equations, Bodin et al. [2012] formulated a many-body constraint to maintain the constant mass density ρ_0 for incompressible fluid simulation. The density at each particle can be estimated using SPH. The constraint for particle i is

$$C_i(\mathbf{x}_1, \ldots, \mathbf{x}_n) = \frac{\rho_i}{\rho_0} - 1 = \frac{1}{\rho_0} \sum_j m_j W(\mathbf{r}_{ij}, h) - 1 \tag{2.12}$$

Compared to standard SPH, their constraint-based fluid is nearly incompressible and results in a much more stable and realistic simulation. By enforcing the density constraint into PBD framework, Macklin and Müller [2013] proposed position-based fluids.

2.4 Unified Solver

As aforementioned, based on PBD, Macklin et al. [2014] presented a unified solver using particle-based representation. Contact and collisions are treated in a unified manner through the use of particles connected by constraints as the fundamental building block, thus the framework is flexible enough to model gases, liquids, deformable solids, rigid bodies and cloth with two-way interactions.

They presented a novel friction constraint for their framework, which can effectively prevent granular piles from quickly collapsing. Fluid-solid coupling is also implemented in their unified solver elegantly by including solid particles in the fluid density estimation and treating fluid particles as solid particles during collision resolution.

Rigid bodies are also represented using particles, therefore the mesh-less shape matching is a more appropriate method for rigid body simulation. Particles for rigid bodies are created by performing solid voxelization of a closed triangle mesh and placing particles at occupied cells. *Signed distance field (SDF)* is used to determine the direction and magnitude of the position corrections for rigid bodies during collision resolution. Generally solid voxelization and exact construction of signed distance field are very compute-intensive, which makes the initialization very slow for scenes containing large amounts of complex meshes with many faces.

The unified dynamics framework imposes the restriction of fixed radius of the particles in the same scene, so that the collision detection can be efficiently performed using a uniform hash grid. However, under some circumstances this may lead to an inefficient representation of the scene. For example, given a certain amount of water, the particles for the water should not be too large, otherwise the simulation may be unrealistic, as has been shown in the demo included in the release package of NVidia Flex library (Fig. 2.1). However, it is usually sufficient to use a coarser representation for rigid bodies. Particles inside a stiff object which strongly resists deformation can be much larger than those lying on the surface.

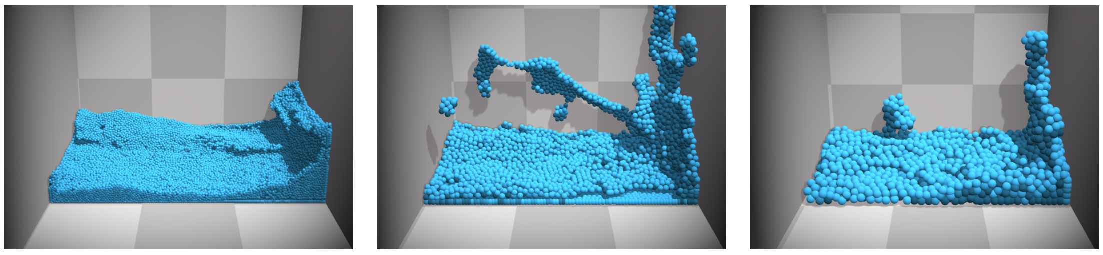

(a) radius = 5 cm (b) radius = 10 cm (c) radius = 15 cm

Fig. 2.1: Three dam break scenes. The particles are created by sampling an $1m \times 2m \times 1m$ box of water with different sampling rate at the initialization, resulting in particles of different sizes for different scenes. All of these three screenshots are captured at $80th$ frame. It is evident that lower sampling rate leads to an implausible fluid simulation.

In this project we integrated the GPU implementations of voxelization and SDF construction into the unified framework, which can largely reduce the time expense on the initialization of scenes containing complex meshes, and avoid large amounts of data transfer between the host and device. We also removed the requirement of fixed particle size through the use of hierarchical acceleration structures.

2.5 Voxelization

For most applications, voxelization is a performance-critical part, thus several algorithms have been proposed to voxelize triangle meshes using GPU rasterization pipeline in order to increase parallelism.

Fang and Chen [2000] proposed a slice-wise surface voxelization, in which a 3D texture object is split into multiple 2D slices using two shifted clip planes. The voxels are found using OpenGL wireframe mode. They implemented the solid voxelization by rendering the object from one direction and flipping the inside/outside state of all affected voxels. Instead of slicing, depth peeling can be used to avoid voxelization of the "empty" space, thereby improving the efficiency [Li et al., 2003]. By processing all slices in one pass, Eisemann and Décoret [2008] achieved high performance in their solid voxelization method.

Due to the point sampling provided by conventional rasterization, all these above mentioned voxelization methods may easily miss thin structures and suffer from gaps. Schwarz and Seidel [2010] proposed data-parallel algorithms for both surface and solid voxelization using GPU as a general massively parallel compute device instead of building on the standard graphics pipeline. They also proposed a novel method to perform sparse solid voxelization directly into a sparse hierarchical structure, which is more space-efficient.

2.6 Distance Transform (DT)

The distance transform, also known as distance field, is a useful construction in areas of Computer Vision, Physics and Computer Graphics. The distance transform is closely related to the Voronoi diagram, where the actual closest site to each point is of interest. Formally,

Definition 2.6.1. A d-dimensional binary image is a function I from the elements of an $n_1 \times n_2 \times \cdots \times n_d$ array to $\{0, 1\}$. The total number of the elements is $N = n_1 \times n_2 \times \cdots \times n_d$. The elements are called pixels when $d = 2$ and voxels when $k \geq 3$. Voxels of value 0 and 1 are called background voxels and foreground or feature voxels (FVs), respectively. For a given distance metric, the *distance transform (DT)* of an image I is an assignment to each voxel x of the distance between x and the *closest feature voxel (CFV)* in I [Maurer et al., 2003]. *Feature voxels (FVs)* are also termed *sites* by Cao et al. [2010].

Definition 2.6.2. the *Voronoi diagram* $\mathcal{V}_\mathcal{S}$ of a set of *Voronoi sites* $\mathcal{S} = \{\mathbf{s}_i\}$ consists of a set of disjoint *Voronoi cells* $\mathcal{V}_\mathcal{S} = \{\mathcal{C}_{\mathbf{s}_i}\}$, $i = 1, 2, \ldots, N_\mathcal{S}$. The Voronoi cell $\mathcal{C}_{\mathbf{s}_i}$ is the set of all points whose closest site is $\mathbf{s}_i$ together with the cell boundaries formed by points equidistant from $\mathbf{s}_i$ and one or more other Voronoi sites. The Voronoi site $\mathbf{s}_i$ is also known as the *Voronoi center* of $\mathcal{C}_{\mathbf{s}_i}$ [Maurer et al., 2003].

Chamfer distance transform is a family of two-pass algorithms to compute distance transform, in which the first pass propagates the distance information from left top to right bottom, and the second pass propagates the distance information from right bottom to left top of the image (Fig. 2.2) [Grevera, 2007].

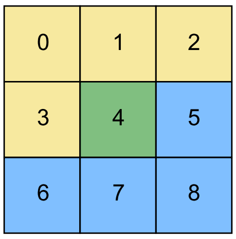


Fig. 2.2: Chamfer distance transform at pixel 4 using a window of size 3×3. In the first pass, the distance information of left and top neighbors (pixel $0 - 3$) are used to update the distance information at pixel 4, and in the second pass, the right and bottom neighbors (pixel $5 - 8$) propagate their distance information to pixel 4. The pixels involved in each pass roughly form a "chamfered edge".

Rosenfeld and Pfaltz [1966] presented a method to perform chamfer distance transform to an image, and created distance skeleton for the distance transform. Thereafter the accuracy of the chamfer distance transform has been improved, and alternative algorithms such as vector distance transform, fast marching method and level sets have been introduced. Although earlier work concentrated mainly on two dimensional image processing, three and higher dimensional distance transform has also been researched as the concept of distance transform found their use in other areas such as correcting the topology of meshes, collision detection, modeling, manipulation and visualization of objects for Computer Graphics [Jones et al., 2006].

A substantial amount of effort has been devoted to both approximate DT and exact DT. Most approximate DT algorithms are based on vector propagation which stores a vector pointed to the candidate site for each pixel or voxel in the image, and these vectors are propagated according to a predefined pattern (*vector template*) [Danielsson, 1980].

The *fast marching method (FMM)* is a numerical algorithm for computing the arrival time of a front propagating in the direction normal to itself by solving the Eikonal equation

$$\|\nabla T\| = \frac{1}{F}, \tag{2.13}$$

where $F \geq 0$ is the speed of the front, and T is the arrival time of the front. It can be used to approximate the distance transform if the speed of the front is set to 1. The time complexity of FMM is $O(N \log N)$ [Jones et al., 2006].

Hoff III et al. [1999] presented an approach to compute Generalized Voronoi Diagrams and the approximate distance transform in two and three dimensions using interpolation-based polygon rasterization hardware. They rendered a three dimensional distance mesh for each Voronoi site. The closest site and the distance to that site for each sample point are computed using polygon scan-conversion and Z-buffer depth comparison. Their method suffers from different kinds of errors:

- Meshing error caused by approximation of the distance functions using distance mesh;
- Tessellation error due to approximating curved primitive using linear primitives;
- Resolution error introduced by coarse sampling of the cell grid by the raster;
- Hardware precision error due to fixed-precision arithmetic.

Maurer et al. [2003] presented a sequential algorithm of running time $O(N)$ for exact DT of binary images in arbitrary dimensions using dimensionality reduction and partial Voronoi diagram construction. Specifically, the D-dimensional DT is computed by intersecting the $(D-1)$-dimensional Voronoi diagram whose sites are the feature voxels with the Dth dimension of the image. The intersection can be performed using the $(D-1)$-dimensional DT.

Cao et al. [2010] proposed a work-efficient parallel algorithm of complexity $O(N)$ to compute the exact distance transform for 2D and higher dimensional images using GPU. They partitioned the input image into bands to increase the parallelism, and used an efficient merging to combine the sub-results. Their method outperforms all GPU-based approximate DT algorithm in 2D and 3D.

2.7 Acceleration Structures

For particles of uniform size, collision detection and neighbor finding can be accelerated using a uniform gird which subdivide the simulation space into grid of uniformly sized cells (usually the same as the size of particles, namely double the radius) [Green, 2008]. A uniform hash grid can be efficiently implemented on the GPU and provide quick collision detection by restricting tests to the particles inside 27 grid cells for each particle. Due to limited memory, the size of the hash grid is typically not large enough to cover the whole scene, thus the particles outside the hash grid will be mapped back into the grid during collision test, also known as wrapping, which is usually realized by the modulo operation.

Typically there are two choices for the size of the grid cells when using a uniform hash grid in a scene containing differently sized particles:

(a) The size is chosen to be the same as the diameter of the largest particle. In this case, the number of the grid cells to be tested is still 27 for each particle. However, the number of the tests will increase drastically since the grid cells could be very large. In the extreme, the complexity is roughly $O(n^2)$ if the size of the largest particle is of the same order of magnitude as the hash grid;

(b) The size is chosen to be the same as the diameter of the smallest particle. In this case, a larger particle i will be mapped into multiple cells, and each particle (partially) mapped into any of these cells has to be tested for particle i. The test between two large particles may be executed more than once due to this kind of "one-to-many" mapping. Another issue is related to the limited size of the hash grid. For a particle of comparable size to the hash grid, almost all the particles in the whole scene have to be tested even if this large particle is outside the hash grid.

Therefore, a hierarchical acceleration structure is more suitable for scenes consisting of particles of different sizes.

2.7.1 Construction

Most research on bounding volume hierarchy (BVH) construction focuses purely on serial construction algorithms, which can hardly achieve real-time performance. A number of methods has been proposed for constructing BVHs, octrees and k-d trees using general-purpose GPU computing, some of which aim to maximize the quality of resulting trees using surface area heuristic (SAH) [Danilewski et al., 2010], while others prefer rapid construction to the tree quality [Garanzha et al., 2011; Lauterbach et al., 2009; Pantaleoni and Luebke, 2010].

Lauterbach et al. [2009] presented a parallel algorithm for rapidly constructing BVH on manycore GPUs using a linear ordering derived from spatial Morton codes, by means of which the BVH construction problem is reduced to a simple sorting problem. The hierarchy is generated in a top-down fashion.

Zhou et al. [2011] constructed octrees using Morton codes in the context of surface reconstruction. They used a reverse level-order traversal of the octree to build the hierarchy, i.e. the nodes in depth $(D-1)$ are determined using parallel compaction operations on the nodes in depth D.

Karras [2012] presented a novel data-parallel algorithm for the construction of binary radix trees from Morton codes, and efficient methods to build BVHs, octrees, and k-d trees using binary radix trees. The parallel radix tree construction algorithm overcomes the sequential bottlenecks in the construction of various spatial acceleration structures.

2.7.2 Traversal

BVH traversal is usually implemented via a stack, which is straightforward and efficient. However, the memory and bandwidth requirement of maintaining a full stack could be huge if the traversal is carried out in a highly parallel fashion. To minimize the use of memory, many stack-less traversal algorithms have been proposed.

Hapala et al. [2011] presented an iterative traversal algorithm using a state machine to infer the next node to be visited. One problem with this algorithm is that some inner nodes are "accessed" twice, and that the traversal order heuristic (near/far child) may be executed twice.

Barringer and Akenine-Möller [2013] proposed low-overhead stack-less traversal algorithms for implicit binary trees and sparse trees. Their algorithms support dynamic descent direction without restarting. Áfra and Szirmay-Kalos [2014] proposed a similar algorithm

MBVH2 for binary BVH traversal which uses a bitstack in place of the descent level index used by Barringer and Akenine-Möller [2013]. In contrast to the algorithms proposed by Barringer and Akenine-Möller [2013], MBVH2 does not require the traversal to return to the root node before termination, therefore is slightly more efficient.

3. Position-Based Unified Particle Physics

Our work is based on the unified particle-based dynamics framework proposed by Macklin et al. [2014], where particles are the fundamental building blocks for all object types. The particle representation significantly reduces the collision types to be processed, simplifies the algorithms for contact generation, and can be efficiently implemented by exploiting the highly parallel processing capability of the GPU.

The core particle state consists of the following properties:

```
struct Particle
{
    float x[3]; // position
    float v[3]; // velocity
    float w; // multiplicative inverse of the mass
    float radius;
    int   phase;
};
```

Since the restriction of fixed particle size per scene is removed in our project, it is necessary to include the radius in the particle properties. The *phase identifier* is used to organize particles into groups so that the interaction between particles such as disabling collisions between particles of the same group can be conveniently controlled.

The particles are connected by constraints. *Constraints* are limitations imposed on the geometrical or kinematic configuration of a mechanical system. A constraint is *holonomic* if it imposes restrictions only on the geometrical configuration of the particles $\mathbf{x}_i$, and imposes no restriction on their time variations $\dot{\mathbf{x}}_i, \ddot{\mathbf{x}}_i$ [DiBenedetto, 2010].

Typically only holonomic constraints are used in PBD. A *bilateral constraint*, i.e. an equality constraint is defined by

$$C(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N) = 0, \tag{3.1}$$

and a *unilateral constraint*, namely an inequality constraint is defined by

$$C(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N) \geq 0, \tag{3.2}$$

where $\{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}$ are the particle positions.

3.1 Solver

The equation of motion of a particle is governed by Newton's second law:

$$\dot{\mathbf{v}}_i = \frac{1}{m_i}\mathbf{f}_i, \tag{3.3}$$

where $\mathbf{f}_i$ is the sum of forces acting on particle i, and the velocity $\mathbf{v}_i$ is the rate of change of the particle position $\mathbf{x}_i$:

$$\dot{\mathbf{x}}_i = \mathbf{v}_i. \tag{3.4}$$

The system to be simulated is second order in time, therefore both the positions and velocities of the particles need to be initialized before entering the simulation loop.

3.1.1 Time Integration

The first step of the simulation loop is to predict the velocities and positions of the particles. In the field of PBD the most popular integration scheme is the *symplectic Euler method*:

$$\begin{aligned} \mathbf{v}_i^{n+1} &= \mathbf{v}_i^n + \mathbf{F}_i\frac{\mathrm{d}t}{m_i} \\ \mathbf{p}_i^{n+1} &= \mathbf{x}_i^n + \mathbf{v}_i^{n+1}\mathrm{d}t \end{aligned}. \tag{3.5}$$

Symplectic Euler method is similar to Verlet integration(Eq. 2.4) in the sense that the velocity of the last step $\mathbf{v}_i^n$ is derived from the position change in the last step (Algorithm 3.3), although this is done implicitly in Verlet integration. However, more tuning options for PBD are given by explicitly storing the velocity, such as particle sleeping described in section 3.1.5.

In contrast to the explicit Euler method (Eq. 2.1), the velocity at $(n+1)$th time step is used for the prediction of the position vector $\mathbf{p}_i^{n+1}$.

3.1.2 The System to be Solved

After the prediction step described above, the positions of the particles should be adjusted so that all the constraints are still satisfied. The goal of the PBD solver is to find the position correction $\Delta\mathbf{x}_i$ for each particle i. For the following formula deduction and practical calculation we mostly follow the work of Macklin et al. [2014] and Bender et al. [2015].

Given q constraints, the system to be solved is

$$\begin{gathered} C_1(\mathbf{x}+\Delta\mathbf{x}) \succ 0, \\ C_2(\mathbf{x}+\Delta\mathbf{x}) \succ 0, \\ \vdots \\ C_q(\mathbf{x}+\Delta\mathbf{x}) \succ 0, \end{gathered} \tag{3.6}$$

where $\mathbf{x} = [\mathbf{x}_1^T,\ \mathbf{x}_2^T,\ \ldots,\ \mathbf{x}_N^T]^T$, $\Delta\mathbf{x} = [\Delta\mathbf{x}_1^T,\ \Delta\mathbf{x}_2^T,\ \ldots,\ \Delta\mathbf{x}_N^T]^T$. The symbol $\succ$ denotes "$=$" if C_i is a bilateral constraint and "$\geq$" if C_i is a unilateral constraint. The constraints can be approximated using the linearization of C around $\mathbf{x}$,

$$C_i(\mathbf{x}+\Delta\mathbf{x}) \approx C_i(\mathbf{x}) + \nabla C_i(\mathbf{x})\Delta\mathbf{x}. \tag{3.7}$$

Let $\mathbf{M} = diag(m_1, \dots, m_N)$ be the mass matrix, and $\mathbf{W} = diag(w_1, \dots, w_N)$ be the inverse of the mass matrix $\mathbf{M}$. Linear and angular momentum conservation requires $\Delta\mathbf{x}$ to be in the direction of $\nabla C_i(\mathbf{x})$ and weighted by $\mathbf{W}$, i.e.

$$\Delta\mathbf{x} = \mathbf{W}\nabla C_i(\mathbf{x})^T \lambda_i. \tag{3.8}$$

where λ_i is the *Lagrange multiplier*. For bilateral constraints, λ_i can be calculated by combining Eq. (3.7) and Eq. (3.8):

$$\lambda_i = -\frac{C_i(\mathbf{x})}{\nabla C_i(\mathbf{x})\mathbf{W}\nabla C_i(\mathbf{x})^T} \tag{3.9}$$

After a given number of iterations, the change in velocity is computed according to the total constraint delta

$$\Delta\mathbf{v}_i = \frac{\Delta\mathbf{x}_i}{\Delta t} \tag{3.10}$$

According to the Gauss principle of least constraints, the problem is equivalent to finding the minimum change in kinematic energy that satisfies the constraints, which can be formally stated as an optimization problem:

$$\text{minimize} \quad \frac{1}{2}\Delta\mathbf{v}^T\mathbf{M}\Delta\mathbf{v} \tag{3.11}$$
$$\text{subject to} \quad C_i(\mathbf{x} + \Delta\mathbf{x}) \succ 0, i = 1, \dots, q \tag{3.12}$$

According to Eq. (3.10), minimizing $\frac{1}{2}\Delta\mathbf{v}^T\mathbf{M}\Delta\mathbf{v}$ is equivalent to minimizing $\frac{1}{2}\Delta\mathbf{x}^T\mathbf{M}\Delta\mathbf{x}$. By applying the linearization in Eq. (3.7), and assuming all the constraints are bilateral, the problem becomes

$$\text{minimize} \quad \frac{1}{2}\Delta\mathbf{x}^T\mathbf{M}\Delta\mathbf{x} \tag{3.13}$$
$$\text{subject to} \quad \mathbf{C}(\mathbf{x}) + \nabla\mathbf{C}(\mathbf{x})\Delta\mathbf{x} = \mathbf{0}, \tag{3.14}$$

which is a quadratic programming problem. This problem can be solved by the method of Lagrange multipliers [Rao, 2009]. The Lagrange function can be written as

$$\mathcal{L}(\Delta\mathbf{x}, \boldsymbol{\lambda}) = \frac{1}{2}\Delta\mathbf{x}^T\mathbf{M}\Delta\mathbf{x} + (\nabla\mathbf{C}(\mathbf{x})\Delta\mathbf{x} + \mathbf{C}(\mathbf{x}))^T\boldsymbol{\lambda}, \tag{3.15}$$

where $\boldsymbol{\lambda} = [\lambda_1,\ \lambda_2,\ \dots,\ \lambda_q]^T$ is the vector of Lagrange multipliers. The optimum solutions can be found by solving the following equations (necessary conditions):

$$\frac{\partial\mathcal{L}}{\partial\Delta\mathbf{x}} = \mathbf{M}\Delta\mathbf{x} + \nabla\mathbf{C}^T(\mathbf{x})\boldsymbol{\lambda} = \mathbf{0}, \tag{3.16}$$
$$\frac{\partial\mathcal{L}}{\partial\boldsymbol{\lambda}} = \nabla\mathbf{C}(\mathbf{x})\Delta\mathbf{x} + \mathbf{C}(\mathbf{x}) = \mathbf{0}. \tag{3.17}$$

Eq. (3.8) and (3.9) can be immediately derived from the conditions (3.16) and (3.17).

3.1.3 Iterative Methods

From Eq. (3.16) we have

$$\Delta \mathbf{x} = \mathbf{M}^{-1} \nabla \mathbf{C}^T(\mathbf{x}) \boldsymbol{\lambda}. \tag{3.18}$$

Plugging Eq. (3.18) into Eq. (3.17) yields

$$\mathbf{A}\boldsymbol{\lambda} = [\nabla \mathbf{C}(\mathbf{x}) \mathbf{M}^{-1} \nabla \mathbf{C}^T(\mathbf{x})] \boldsymbol{\lambda} = -\mathbf{C}(\mathbf{x}). \tag{3.19}$$

Instead of constructing $\mathbf{A} = \nabla \mathbf{C}(\mathbf{x}) \mathbf{M}^{-1} \nabla \mathbf{C}^T(\mathbf{x})$ explicitly, and finding the exact solution to $\boldsymbol{\lambda}$, which could be prohibitively memory intensive, PBD processes each constraint separately by computing $\boldsymbol{\lambda}$ with Eq. (3.9) and updating $\mathbf{x}$ with Eq. (3.8) in each solver iteration. Generally multiple iterations are needed for the convergence of the solution.

There is a group of iterative methods based on matrix splitting to solve linear system iteratively [Allaire and Kaber, 2008]. Let $\mathbf{A}$ and $\mathbf{P}$ be nonsingular matrices, $\mathbf{A} = \mathbf{P} - \mathbf{N}$ is a splitting of $\mathbf{A}$. Given the initial solution $\mathbf{x}_0$, a splitting-based iterative method for solving the linear system $\mathbf{A}\mathbf{x} = \mathbf{b}$ is

$$\mathbf{x}^{k+1} = \mathbf{P}^{-1}\mathbf{N}\mathbf{x}^k + \mathbf{P}^{-1}\mathbf{b}, \quad \forall k \geq 1, \tag{3.20}$$

where the matrix $\mathbf{P}^{-1}\mathbf{N}$ is called an *iteration matrix* or *amplification matrix* of the iterative method.

Let matrix $\mathbf{A} = (a_{i,j})_{1\leq i,j \leq n}$, $\mathbf{D} = (d_{i,j})_{1\leq i,j \leq n}$, $\mathbf{E} = (e_{i,j})_{1\leq i,j \leq n}$, $\mathbf{F} = (f_{i,j})_{1\leq i,j \leq n}$, where $\mathbf{D}$ is the diagonal, $-\mathbf{E}$ is the lower part, and $-\mathbf{F}$ is the upper part of $\mathbf{A}$. Namely,

$$\begin{cases} d_{i,j} = a_{i,j}\delta_{i,j}; \\ e_{i,j} = -a_{i,j} \text{ if } i > j, \text{and 0 otherwise;} \\ f_{i,j} = -a_{i,j} \text{ if } i < j, \text{and 0 otherwise.} \end{cases} \tag{3.21}$$

The *Jacobi method* is defined by the splitting $\mathbf{P} = \mathbf{D}$, $\mathbf{N} = \mathbf{D} - \mathbf{A}$. The entries of x^{k+1} are computed from x^k:

$$x_i^{k+1} = \frac{1}{a_{i,i}} \left[-a_{i,1}x_1^k - \cdots - a_{i,i-1}x_{i-1}^k - a_{i,i+1}x_{i+1}^k - \cdots - a_{i,n}x_n^k + b_i \right], \tag{3.22}$$

The *Gauss-Seidel* method is defined by the splitting $\mathbf{P} = \mathbf{D} - \mathbf{E}$, $\mathbf{N} = \mathbf{F}$. The computation of x_i^{k+1} also uses the first $(i-1)$ entries of $\mathbf{x}^{k+1}$ which is already available:

$$x_i^{k+1} = \frac{1}{a_{i,i}} \left[-a_{i,1}x_1^{k+1} - \cdots - a_{i,i-1}x_{i-1}^{k+1} - a_{i,i+1}x_{i+1}^k - \cdots - a_{i,n}x_n^k + b_i \right]. \tag{3.23}$$

In PBD, Gauss-Seidel iteration updates the positions immediately after each constraint is processed. The Gauss-Seidel method is a linear relaxation method with slow global convergence. However, it is stable, easy to implement, and able to damp out jitter and instabilities [Bodin et al., 2012]. Despite the slow convergence, significant errors can be effectively reduced by the Gauss-Seidel method in the first few iterations [Bender et al., 2015].

Projected Gauss-Seidel (Algorithm 3.1) is used in the presence of unilateral constraints which turn the linear system into a *linear complementarity problem (LCP)*. The constraint is in effect skipped if $C_i(\mathbf{x}) \geq 0$ through the use of the projection operator “max” in Line 4 of Algorithm 3.1, hence the name “projected” [Lu, 2016].

Algorithm 3.1 Projected Gauss-Seidel

```
 1: $k \leftarrow 0$                                              ▷ set the iteration counter $k$ to 0
 2: repeat
 3:     for $i = 1, 2, \ldots, q$ do
 4:         $\lambda_i \leftarrow \max\left(0, -\frac{C_i(\mathbf{x})}{\nabla C_i(\mathbf{x})\mathbf{M}^{-1}\nabla C_i(\mathbf{x})^T}\right)$      ▷ Eq. (3.9)
 5:         $\Delta\mathbf{x} \leftarrow \mathbf{M}^{-1}\nabla C_i(\mathbf{x})^T\lambda_i$      ▷ Eq. (3.8)
 6:         $\mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}$
 7:     end for
 8:     $k \leftarrow k + 1$
 9: until $k = k_{max}$                              ▷ $k_{max}$: maximum number of iterations
```

Gauss-Seidel iteration is sequential due to Line 6 of Algorithm 3.1, which cannot be very efficiently implemented on the GPU. *Projected Jacobi* (Algorithm 3.2) can be used to increase parallelism, because the positions of the particles remain unchanged during each Jacobi iteration.

Algorithm 3.2 Projected Jacobi

```
 1: $k \leftarrow 0$                                              ▷ set the iteration counter $k$ to 0
 2: $\Delta\mathbf{x} \leftarrow \mathbf{0}$
 3: repeat
 4:     for $i = 1, 2, \ldots, q$ do
 5:         $\lambda_i \leftarrow \max\left(0, -\frac{C_i(\mathbf{x})}{\nabla C_i(\mathbf{x})\mathbf{M}^{-1}\nabla C_i(\mathbf{x})^T}\right)$      ▷ Eq. (3.9)
 6:         $\Delta\mathbf{x} \leftarrow \Delta\mathbf{x} + \mathbf{M}^{-1}\nabla C_i(\mathbf{x})^T\lambda_i$      ▷ Eq. (3.8)
 7:     end for
 8:     $\mathbf{x} \leftarrow \mathbf{x} + \Delta\mathbf{x}$
 9:     $k \leftarrow k + 1$
10: until $k = k_{max}$                              ▷ $k_{max}$: maximum number of iterations
```

3.1.4 Relaxation

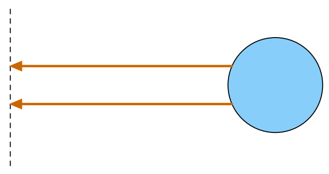


Fig. 3.1: A one dimensional particle i of mass m_i is constrained to lie at $x = 0$ by two identical distance constraints. At the beginning of one time step, the particle locates at position x'_i, which is away from its rest pose, thus both of the two constraints are violated.

Under some circumstances the constraints are ill-conditioned, which makes it impossible to solve the system using the Jacobi method. The problem can be explained with the example (Fig. 3.1) given by Macklin et al. [2014].

The constraint equations are:

$$\begin{aligned} C_1(x) &= x_i = 0, \\ C_2(x) &= x_i = 0. \end{aligned} \tag{3.24}$$

At the beginning of the first iteration, the Lagrange multiplier is $\lambda_1 = -\frac{x'_i}{m_i}$, and $\Delta x = -x'_i$. Because the Gauss-Seidel method will update $\mathbf{x}$ after the first constraint is processed, i.e. the particle is moved by $-x'_i$, thereby back to the origin, λ_2 for constraint $C_2(x)$ becomes 0, and no further position correction for particle i is needed.

However, with Jacobi method, $\lambda_2 = \lambda_1 = -\frac{x'_i}{m_i}$ for the first iteration, as the position $x = x'_i$ is the same for both constraints. After the first iteration, particle i will be moved to $x = -x'_i$, thereby neither of the constraints is satisfied. Similarly, during the next iteration particle i will be projected back to the position $x = x'_i$. In fact, the particle will oscillate all the time, and the correct solution will never be reached by Jacobi iterations.

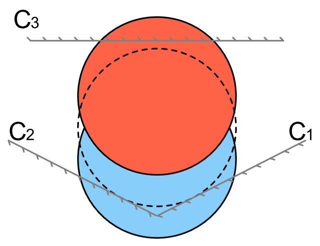


Fig. 3.2: The position of the particle j is constrained by three walls. At time step n, after the prediction step (subsection 3.1.1), the particle locates at the position $\mathbf{x}_j^0$ represented by the light blue circle. Although one Gauss-Seidel iteration will move the particle to the correct position $\mathbf{x}_j^{goal}$ represented by the dotted circle, where all three constraints are satisfied and no more iteration is needed, the first Jacobi iteration will project the particle to the position $\mathbf{x}_j^1$ represented by the orange circle, where constraint C_3 is violated. Therefore Jacobi method needs another iteration to fulfill all of these three constraints.

Similar constraints can result in very slow convergence, as illustrated in Fig. 3.2. The problem could be much more severe for the Jacobi method due to the slow local propagation of the error if the three walls in Fig. 3.2 are replaced by a pile of particles.

Macklin et al. [2014] applied under-relaxation based on constraint averaging [Bridson et al., 2002] to address the oscillation and slow convergence problem with the Jacobi method. Specifically, Line 8 of Algorithm 3.2 (projected Jacobi method) is replaced by

$$\mathbf{x} \leftarrow \mathbf{x} + \boldsymbol{\kappa} \circ \Delta\mathbf{x}, \tag{3.25}$$

where $\boldsymbol{\kappa} = [\frac{1}{n_1}, \frac{1}{n_2}, \ldots, \frac{1}{n_N}]^T$ is a vector of the multiplicative inverse of the number of constraints affecting each particle. The momentum may not be preserved when $n_i \neq n_j$ for two neighboring particles i and j which are connected by one collision constraint, but the visual artifacts are typically negligible.

In cases when the averaging in Eq. (3.25) is too aggressive, which leads to slow convergence, Macklin et al. [2014] introduced a user-tunable parameter ω to control the rate of *successive* *over-relaxation (SOR)*. The formula for position update becomes

$$\mathbf{x} \leftarrow \mathbf{x} + \omega \boldsymbol{\kappa} \circ \Delta \mathbf{x}. \tag{3.26}$$

Under most circumstances $1 \leq \omega \leq 2$. $\omega < 1$ corresponds to the under-relaxation, which is typically unnecessary since the under-relaxation provided by the constraint averaging is sufficient for most cases to avoid divergence, as being noted by Macklin et al. [2014]; $\omega > 2$ could be used for some scenes when necessary, however, higher values may cancel out the effect of the constraint averaging and result in an unstable system.

3.1.5 Stabilization

A common paradigm to achieve the best approximation of the exact solution using an iterative method is to break the iteration loop only when the error is smaller than a user-defined threshold or the maximum execution time is exceeded. In contrast, the PBD solver only runs a given number of iterations for each time step to achieve real-time performance. One problem with this approach is that some constraints may not be fulfilled at the beginning of the next time step, resulting in undesirable effect, such as unnatural movement (Fig. 3.3) and positional drifting.

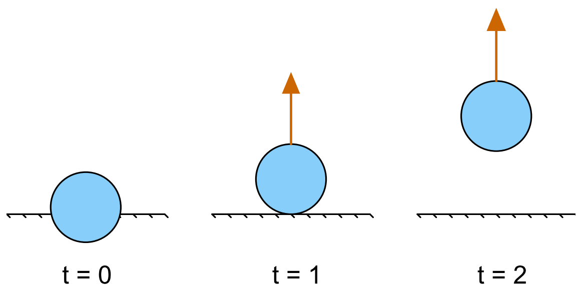


Fig. 3.3: An example given by Macklin et al. [2014]. At the beginning of $t = 0$, part of the particle with velocity 0 is trapped under the ground due to early break of the iteration loop. PBD projects it to the surface and updates its velocity accordingly at end of this time step. Due to the resulting velocity the particle seems to be shot out of the ground and continues to move upwards in the subsequent time steps.

Macklin et al. [2014] introduced a *pre-stabilization* pass to address this issue. After the prediction step (subsection 3.1.1) and the generation of the contact constraints, the pre-stabilization pass solves the contact constraints with the positions before prediction. That is, both the original positions $\mathbf{x}^n$ and the predicted positions $\mathbf{p}^{n+1}$ will be corrected by the pre-stabilization. Usually one or two iterations of pre-stabilization on contact constraints such as collision and friction is sufficient to avoid most visual artifacts.

Positional drifting can be solved by *particle sleeping* at the end of each time step, which freezes particles in place if their velocity becomes smaller than a user-defined threshold ϵ [Macklin et al., 2014]:

$$\mathbf{x}_i^{n+1} = \begin{cases} \mathbf{p}_i^{n+1}, & \left|\mathbf{p}_i^{n+1} - \mathbf{x}_i^n\right| > \epsilon, \\ \mathbf{x}_i^n, & \text{otherwise.} \end{cases} \tag{3.27}$$

We note that the velocity $\mathbf{v}_i$ shall not be affected by particle sleeping, otherwise the behavior of the system becomes unrealistic. For example, rigid bodies may balance in a non-physical pose.

3.1.6 The Simulation Loop

Summing up the methods presented in the previous subsections leads to Algorithm 3.3 for the simulation loop [Macklin et al., 2014]:

Algorithm 3.3 Simulation Loop

```
1: for all particles $i$ do
2:     $\mathbf{v}_i \leftarrow \mathbf{v}_i + \mathbf{F}_{ext}(\mathbf{x}_i)\Delta t$
3:     $\mathbf{p}_i \leftarrow \mathbf{x}_i + \mathbf{v}_i\Delta t$
4:     $m_i \leftarrow m_i e^{-kh(\mathbf{p}_i)}$    ▷ mass scaling for the fast convergence of stacks of rigid bodies
5: end for
6: for all particles $i$ do
7:     find neighboring particles $\mathcal{N}_i(\mathbf{p}_i)$
8:     generate solid contacts constraints
9: end for
10: $iter \leftarrow 0$
11: while $iter < k_{stabilize}$ do    ▷ $k_{stabilize}$: maximum iterations of pre-stabilization
12:     $\Delta\mathbf{x} \leftarrow \mathbf{0},\ \mathbf{n} \leftarrow \mathbf{0}$
13:     solve contact constraints for $\Delta\mathbf{x},\ \mathbf{n}$
14:     $\mathbf{x} \leftarrow \mathbf{x} + \boldsymbol{\kappa} \circ \Delta\mathbf{x}$    ▷ Eq. (3.25)
15:     $\mathbf{p} \leftarrow \mathbf{x} + \boldsymbol{\kappa} \circ \Delta\mathbf{x}$
16: end while
17: $iter \leftarrow 0$
18: while $iter < k_{solver}$ do    ▷ $k_{solver}$: maximum number of solver iterations
19:     $\Delta\mathbf{x} \leftarrow \mathbf{0},\ \mathbf{n} \leftarrow \mathbf{0}$
20:     solve contact constraints for $\Delta\mathbf{x},\ \mathbf{n}$
21:     $\mathbf{p} \leftarrow \mathbf{p} + \boldsymbol{\kappa} \circ \Delta\mathbf{x}$
22:     restore mass $m_i$
23:     for each constraint group $\mathcal{G}$ do
24:         $\Delta\mathbf{x} \leftarrow \mathbf{0},\ \mathbf{n} \leftarrow \mathbf{0}$
25:         solve all constraints in $\mathcal{G}$ for $\Delta\mathbf{x},\ \mathbf{n}$
26:         $\mathbf{p} \leftarrow \mathbf{p} + \boldsymbol{\kappa} \circ \Delta\mathbf{x}$
27:     end for
28: end while
29: for all particles $i$ do
30:     $\mathbf{v}_i \leftarrow \frac{1}{\Delta t}(\mathbf{p}_i - \mathbf{x}_i)$
31:     $\mathbf{x}_i \leftarrow \mathbf{p}_i$ or apply particle sleeping
32: end for
```

The mass scaling in Line 4 of Algorithm 3.3 is a parallel-friendly method to improve the stability of rigid stacks, which will be explained in the next chapter. The constraint groups (Line 23 of Algorithm 3.3) are built based on the types of the constraints which will be detailed in the next section.

3.2 Constraint Types

Different types of constraints can be incorporated into the PBD framework, making it capable of simulating a variety of natural objects such as rigid bodies, deformable bodies, cloth, granular materials and fluids. In the following we will introduce some specific constraints.

3.2.1 Distance Constraint

Distance constraint is one of the most simple yet non-linear constraints [Bender et al., 2015]. One distance constraint restricts the distance between two particles to be constant. Formally,

$$\mathbf{C}(\mathbf{x}_1, \mathbf{x}_2) = |\mathbf{x}_{21}| - d = |\mathbf{x}_1 - \mathbf{x}_2| - d = 0, \tag{3.28}$$

Where $\mathbf{x}_{21} = \mathbf{x}_1 - \mathbf{x}_2$.

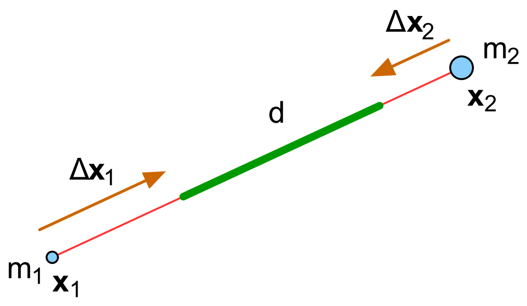


Fig. 3.4: Distance constraint $\mathbf{C}(\mathbf{x}_1, \mathbf{x}_2) = |\mathbf{x}_{21}| - d$. The corrections $\Delta\mathbf{x}_i$ are weighted by the multiplicative inverse of the particle mass $w_i = \frac{1}{m_i}$.

The derivative with respect to $\mathbf{x}_1$ is

$$\nabla_{\mathbf{x}_1}\mathbf{C}(\mathbf{x}_1, \mathbf{x}_2) = \mathbf{n} = \frac{\mathbf{x}_{21}}{|\mathbf{x}_{21}|}, \tag{3.29}$$

and the derivative with respect to $\mathbf{x}_2$ is

$$\nabla_{\mathbf{x}_2}\mathbf{C}(\mathbf{x}_1, \mathbf{x}_2) = -\mathbf{n} = -\frac{\mathbf{x}_{21}}{|\mathbf{x}_{21}|}. \tag{3.30}$$

According to Eq. (3.9), the Lagrange multiplier for distance constraint is computed to be

$$\lambda = -\frac{|\mathbf{x}_{21}| - d}{w_1 + w_2}. \tag{3.31}$$

The position corrections can be calculated with Eq. (3.8):

$$\begin{aligned}\Delta\mathbf{x}_1 &= -\frac{w_1}{w_1 + w_2}\left(|\mathbf{x}_{21}| - d\right)\mathbf{n},\\ \Delta\mathbf{x}_2 &= +\frac{w_2}{w_1 + w_2}\left(|\mathbf{x}_{21}| - d\right)\mathbf{n},\end{aligned} \tag{3.32}$$

which are exactly the same as the formula proposed by Jakobsen [2001], meaning that the distance constraints and the corresponding correction formula introduced by Jakobsen [2001] can be treated as a special case of the general constraint projection.

3.2.2 Contact Constraints

Generally two kinds of constraints are generated when two objects come into contact with each other – the collision constraint along the contact normal, and the friction constraint perpendicular to the contact normal.

Collisions are handled by first finding every pair of objects collide with each other and then generating one collision constraint for each pair. Objects which are close enough to each other could also be included, because contact and neighbor finding are only executed once at the beginning of each time step, and reused by all the solver iterations and pre-stabilization iterations at that time step (Algorithm 3.3). In each iteration only the collision constraints being violated are processed. Collision constraints between two objects may be formulated differently depending on the type of the objects.

To simulate friction, Müller et al. [2007] used damping forces which is vulnerable to other positional constraints, thus static friction can not be modeled [Bender et al., 2015]. Macklin et al. [2014] proposed a novel friction model which could be incorporated into the PBD framework as a part of positional constraints.

3.2.2.1 Environment Collisions

Although in the particle-based unified solver, most of the objects are represented as particles, it is more suitable to use a non-particle representation for some objects. For example, walls and grounds can be modeled as infinite planes and static shapes can be represented by triangle meshes.

An infinite plane could be described by an equation of the form $\mathbf{n} \bullet \mathbf{x} + d = 0$, with the unit vector $\mathbf{n}$ being the normal the plane, and d being the distance from the origin to the plane. The constraint for the collision between particle i and the plane $\mathbf{n} \bullet \mathbf{x} + d = 0$ can be formulated as follows:

$$C(\mathbf{x}_i) = \mathbf{n} \bullet \mathbf{x}_i + d - d_{\text{rest}} \geq 0, \tag{3.33}$$

where d_{rest} is the minimum distance from the particle to the plane. In this constraint only one particle $\mathbf{x}_i$ is involved, and the derivative with respect to this particle is

$$\nabla_{\mathbf{x}_i} \mathbf{C}(\mathbf{x}_i) = \mathbf{n}, \tag{3.34}$$

thus the Lagrange multiplier

$$\lambda = -\frac{\mathbf{n} \bullet \mathbf{x}_i + d - d_{\text{rest}}}{w_i}. \tag{3.35}$$

and the position correction

$$\Delta \mathbf{x}_i = -(\mathbf{n} \bullet \mathbf{x}_i + d - d_{\text{rest}})\mathbf{n}. \tag{3.36}$$

Collision between particles and shapes represented by triangle meshes can be handled by first finding a set of candidate contact triangles, and then each contact triangle can be treated as an infinite plane.

3.2.2.2 Particle Collisions

Linear constraints can also be generated for collisions between free particles by first introducing a contact plane and then treating the contact plane as an infinite plane [Bender et al., 2015]. *Free particles* refer to those particles which are not bounded by a shape matching constraint, namely they do not belong to any rigid body. A more robust method is to maintain the non-linear constraint

$$\mathbf{C}(\mathbf{x}_1, \mathbf{x}_2) = |\mathbf{x}_{21}| - (r_1 + r_2) = |\mathbf{x}_1 - \mathbf{x}_2| - (r_1 + r_2) \geq 0, \tag{3.37}$$

Where $\mathbf{x}_{21} = \mathbf{x}_1 - \mathbf{x}_2$, r_1 and r_2 are the radii of particle 1 and particle 2, respectively. The Lagrange multiplier can be calculated in a similar manner to the distance constraint described in Section 3.2.1. This constraint can be used to simulate granular-like materials [Macklin et al., 2014]. Collision between particles of rigid bodies will be discussed in the next chapter.

3.2.2.3 Friction

After the inter-penetration between two particles being resolved, Macklin et al. [2014] applied frictional position corrections based on the tangential relative displacement to both particles. The relative displacement

$$\Delta\mathbf{x}_\perp = \Big[(\mathbf{p}_j - \mathbf{x}_j) - (\mathbf{p}_i - \mathbf{x}_i)\Big] \perp \mathbf{n}, \tag{3.38}$$

where $\mathbf{p}_i$ and $\mathbf{p}_j$ are the prediction positions (Algorithm 3.3) which include all the position corrections applied to these two particles, $\mathbf{x}_i$ and $\mathbf{x}_j$ are the original particle positions at the beginning of the current time step, $\mathbf{n} = \frac{\mathbf{p}_j - \mathbf{p}_i}{|\mathbf{p}_j - \mathbf{p}_i|}$ is the contact normal, and $\perp$ is a projection operator. The scalar factor equivalent to the Lagrange multiplier

$$\lambda = -\frac{1}{w_i + w_j} \begin{cases} |\Delta\mathbf{x}_\perp|, & |\Delta\mathbf{x}_\perp| < \mu_s d \\ |\Delta\mathbf{x}_\perp| \cdot \min\left(\frac{\mu_k d}{|\Delta\mathbf{x}_\perp|}, 1\right), & \text{otherwise} \end{cases} \tag{3.39}$$

where d is the penetration depth, $0 \leq \mu_s, \mu_k \leq 1$ are the coefficients of static and kinematic friction, respectively. The position corrections

$$\begin{aligned} \Delta\mathbf{x}_i &= +w_i \lambda \frac{\Delta\mathbf{x}_\perp}{|\Delta\mathbf{x}_\perp|}, \\ \Delta\mathbf{x}_j &= -w_j \lambda \frac{\Delta\mathbf{x}_\perp}{|\Delta\mathbf{x}_\perp|}. \end{aligned} \tag{3.40}$$

The friction model is applicable for all kinds of objects. In cases when one of the two colliding objects is a static object such as ground, its mass is set to infinity (i.e. zero weight). By doing so the total position correction will be applied to the movable object. The contact normal is calculated based on the shape of the two colliding objects.

3.2.3 Density Constraint

As aforementioned, Macklin and Müller [2013] proposed position-based fluids (PBF) based on the density constraint

$$C_i(\mathbf{x}_1, \dots, \mathbf{x}_n) = \frac{\rho_i}{\rho_0} - 1 = \frac{1}{\rho_0} \sum_{j \in \mathcal{M}(\mathbf{x}_i)} m_j W(\mathbf{x}_i - \mathbf{x}_j, h) - 1 = 0, \tag{3.41}$$

where $\mathcal{M}(\mathbf{x}_i) = \mathcal{N}_i(\mathbf{x}_i) \cup \{i\}$, and $\mathcal{N}_i(\mathbf{x}_i)$ is a set of local neighbors of particle i inside radius h of the SPH kernel. Apart from being normalized, the SPH kernel should also be even [Müller et al., 2003], i.e. $W(\mathbf{x}_i - \mathbf{x}_j, h) = W(\mathbf{x}_j - \mathbf{x}_i, h)$, thereby $\nabla_{\mathbf{x}_i} W(\mathbf{x}_i - \mathbf{x}_i, h)$ is undefined, thus $\nabla_{\mathbf{x}_i} C_i$ is calculated indirectly from its neighbors by taking advantage of the following symmetries

$$j \in \mathcal{N}_i(\mathbf{x}_i) \Longleftrightarrow i \in \mathcal{N}_j(\mathbf{x}_j), \tag{3.42}$$

$$W(\mathbf{x}_i - \mathbf{x}_j, h) = W(\mathbf{x}_j - \mathbf{x}_i, h) \Longrightarrow \nabla_{\mathbf{x}_i} W(\mathbf{x}_i - \mathbf{x}_j, h) = -\nabla_{\mathbf{x}_i} W(\mathbf{x}_j - \mathbf{x}_i, h). \tag{3.43}$$

The gradient of the constraint function in Eq. (3.41) with respect to particle k is given by

$$\nabla_{\mathbf{x}_k} C_i = \frac{1}{\rho_0} \begin{cases} m_k \sum\limits_{j \in \mathcal{N}_i(\mathbf{x}_i)} \nabla_{\mathbf{x}_k} W(\mathbf{x}_i - \mathbf{x}_j, h) & \text{if } k = i \\ -m_k \nabla_{\mathbf{x}_k} W(\mathbf{x}_i - \mathbf{x}_k, h) & \text{otherwise} \end{cases} \tag{3.44}$$

The Lagrange multiplier

$$\lambda_i = -\frac{C_i(\mathbf{x}_1, \dots, \mathbf{x}_n)}{\sum\limits_{k \in \mathcal{M}(\mathbf{x}_i)} w_k |\nabla_{\mathbf{x}_k} C_i|^2}. \tag{3.45}$$

To avoid the instability caused by the infinitesimal denominator of Eq. (3.45) due to the vanishing gradient at the smoothing kernel boundary, Macklin and Müller [2013] introduced a relaxation parameter ε into the formula (3.44). The Lagrange multiplier is now

$$\lambda_i = -\frac{C_i(\mathbf{x}_1, \dots, \mathbf{x}_n)}{\varepsilon + \sum\limits_{k \in \mathcal{M}(\mathbf{x}_i)} w_k |\nabla_{\mathbf{x}_k} C_i|^2}. \tag{3.46}$$

The position correction for particle i can be calculated as follows:

$$\begin{aligned} \Delta \mathbf{x}_i = \frac{1}{\rho_0} \Bigg[& \left(-w_i \cdot m_i \sum_{j \in \mathcal{N}_i(\mathbf{x}_i)} \nabla_{\mathbf{x}_i} W(\mathbf{x}_j - \mathbf{x}_i, h) \lambda_j \right) + \\ & \left(w_i \cdot m_i \cdot \lambda_i \sum_{j \in \mathcal{N}_i(\mathbf{x}_i)} \nabla_{\mathbf{x}_i} W(\mathbf{x}_i - \mathbf{x}_j, h) \right) \Bigg] \\ = \frac{1}{\rho_0} & \sum_{j \in \mathcal{M}(\mathbf{x}_i)} (\lambda_i + \lambda_j) \nabla_{\mathbf{x}_i} W(\mathbf{x}_i - \mathbf{x}_j, h) \end{aligned} \tag{3.47}$$

Macklin and Müller [2013] also introduced an artificial pressure to address particle clustering problem caused by negative pressures when a particle has only a few neighbors and the rest density is unable to be reached, similar to the approach proposed by Monaghan [2000].

The corrective pressure is specified in terms of the smoothing kernel:

$$s_{\text{corr}} = -k \left(\frac{W(\mathbf{x}_i - \mathbf{x}_j, h)}{W(\Delta\mathbf{q}, h)} \right)^n , \tag{3.48}$$

where $\Delta\mathbf{q}$ is a fixed point inside the smoothing kernel radius, k and n are positive constants. After applying the corrective term, the position correction becomes

$$\Delta\mathbf{x}_i = \frac{1}{\rho_0} \sum_{j \in \mathcal{M}(\mathbf{x}_i)} (\lambda_i + \lambda_j + s_{\text{corr}}) \nabla_{\mathbf{x}_i} W(\mathbf{x}_i - \mathbf{x}_j, h) \tag{3.49}$$

To simulate two-way fluid solid coupling, we follow the approach proposed by Macklin et al. [2014]. When a fluid particle collides with a solid particle, both particles are treated as solid particles, and constraints are generated and processed as described in Section 3.2.2. Conversely, the solid particles also contribute to the fluid density estimation

$$\rho_i = \sum_{j \in \mathcal{M}_f(\mathbf{x}_i)} m_j W(\mathbf{x}_i - \mathbf{x}_j, h) + \sum_{j \in \mathcal{M}_s(\mathbf{x}_i)} s_j m_j W(\mathbf{x}_i - \mathbf{x}_j, h), \tag{3.50}$$

where $\mathcal{M}_f(\mathbf{x}_i)$ is a set of fluid particle neighbors including particle i, $\mathcal{M}_s(\mathbf{x}_i)$ is a set of solid particle neighbors, s_j is a per-particle scaling factor depending on the sampling rate of the solid particles and the configuration of the solid bodies.

Because each particle with radius r represents the substance inside a cube with edge length $2r$, the mass of the particle is calculated as follows:

$$m_i = \rho_i \cdot V = \rho_i \cdot (2r_i)^3, \tag{3.51}$$

where ρ_i is the density specified for particle i at the initialization, and r_i is the radius of particle i.

In our project, the solid particles also participate in the estimation of the gradient of the density constraint.

4. Rigid Bodies

The shape matching method proposed by Müller et al. [2005] is a mesh-less approach intended to simulate visually plausible elastic and plastic deformations. However, by setting the stiffness to one, a deformable object becomes intolerant of deformations of any degree, thus turns in effect to a rigid body. The shape matching constraint can be easily incorporated into the PBD framework, and it is very suitable for the particle representation of rigid bodies.

The collision between particles with the same phase-identifier (i.e. they belong to the same shape, cf. Line 7 of the particle structure at the beginning of Chapter 3) will be resolved implicitly, thus additional contact constraints are no longer necessary for them.

4.1 Shape Matching

After the prediction step (Section 3.1.1), the relative positions of the particles inside one body might be changed drastically, which could no longer represent the original shape of the body. The objective is to find an appropriate goal position for each particle inside the body so that the geometric shape of the body is retained and meanwhile the movement, i.e. the rotation and translation of the body is plausible.

The optimal rotation and translation for the body are found by shape matching. Formally, the shape matching problem in the particle-based solver can be stated as follows:

Definition 4.1.1. Given two sets of particles $\bar{\mathcal{X}} = \{\bar{\mathbf{x}}_1, \dots, \bar{\mathbf{x}}_n\}$ and $\mathcal{X} = \{\mathbf{x}_1, \dots, \mathbf{x}_n\}$ and their correspondences

$$\bar{\mathbf{x}}_i \longleftrightarrow \mathbf{x}_i, \quad \forall i \in \{1, \dots, n\}, \tag{4.1}$$

The goal of the shape matching is to find the rotation matrix $\mathbf{R}$ and the translational vectors $\bar{\mathbf{c}}$ and $\mathbf{c}$ which minimize

$$f(\mathbf{R}, \bar{\mathbf{c}}, \mathbf{c}) = \sum_{i=1}^{n} \mathrm{w}_i \left(\mathbf{R}(\bar{\mathbf{x}}_i - \bar{\mathbf{c}}) - (\mathbf{x}_i - \mathbf{c})\right)^2, \tag{4.2}$$

where w_i is the weight of particle i (not to be confused with the multiplicative inverse of mass, which is represented by w_i). one natural choice for the weights is the physical mass m_i.

The optimal translation vectors are given by

$$\bar{\mathbf{c}} = \frac{\sum_{i=1}^{n} \mathrm{w}_i \bar{\mathbf{x}}_i}{\sum_{i=1}^{n} \mathrm{w}_i}, \quad \mathbf{c} = \frac{\sum_{i=1}^{n} \mathrm{w}_i \mathbf{x}_i}{\sum_{i=1}^{n} \mathrm{w}_i}. \tag{4.3}$$

If $\mathrm{w}_i = m_i$, $\bar{\mathbf{c}}$ and $\mathbf{c}$ are the centers of mass of the initial shape $\bar{\mathcal{X}}$ and the current shape $\mathcal{X}$ respectively. Let

$$\bar{\mathbf{q}}_i = \bar{\mathbf{x}}_i - \bar{\mathbf{c}}, \quad \mathbf{q}_i = \mathbf{x}_i - \mathbf{c}, \quad \forall i \in \{1, \ldots, n\}, \tag{4.4}$$

be the *relative positions* of the initial shape $\bar{\mathcal{X}}$ and the current shape $\mathcal{X}$, and relax the problem of finding the optimal rotation $\mathbf{R}$ to finding the optimal linear transformation $\mathbf{A}$, the term to be minimized in Eq. (4.2) becomes

$$f(\mathbf{A}) = \sum_{i=1}^{n} \mathrm{w}_i \left(\mathbf{A}\bar{\mathbf{q}}_i - \mathbf{q}_i\right)^2. \tag{4.5}$$

Setting the derivative of $f(\mathbf{A})$ with respect to each element in $\mathbf{A}$ to **zero** yields the optimal linear transformation

$$\mathbf{A} = \underbrace{\left(\sum_{i=1}^{n} \mathrm{w}_i \mathbf{q}_i \bar{\mathbf{q}}_i^T\right)}_{\mathbf{A}_{q\bar{q}}} \underbrace{\left(\sum_{i=1}^{n} \mathrm{w}_i \bar{\mathbf{q}}_i \bar{\mathbf{q}}_i^T\right)^{-1}}_{\mathbf{A}_{\bar{q}\bar{q}}} = \mathbf{A}_{q\bar{q}} \mathbf{A}_{\bar{q}\bar{q}}. \tag{4.6}$$

$\mathbf{A}_{\bar{q}\bar{q}}$ is a symmetric matrix which contains no rotation, thus the optimal rotation $\mathbf{R}$ can be extract from the *covariance matrix* $\mathbf{A}_{q\bar{q}}$ of the body $\mathcal{X}$. The *goal position*

$$\mathbf{g}_i = \mathbf{R}(\bar{\mathbf{x}}_i - \bar{\mathbf{c}}) + \mathbf{c}, \quad \forall i \in \{1, \ldots, n\}, \tag{4.7}$$

and the position correction for body $\mathcal{X}$ is given by

$$\Delta \mathbf{x}_i = \alpha(\mathbf{g}_i - \mathbf{x}_i), \quad \forall i \in \{1, \ldots, n\}, \tag{4.8}$$

where α is a user-specified stiffness to restrain the deformation of the body $\mathcal{X}$. For rigid bodies, $\alpha = 1$.

4.1.1 Extraction of the Rotational Part

The problem of extracting the rotational part $\mathbf{R}$ from matrix $\mathbf{A}$ can be formally defined as follows [Müller et al., 2016]:

Definition 4.1.2. Given an arbitrary 3×3 matrix $\mathbf{A}$, find an orthonormal 3×3 matrix $\mathbf{R}$ with $\det(\mathbf{R}) = 1$ which minimizes the *Frobenius-norm* [Björck, 2015]

$$F(\mathbf{R}) = \|\mathbf{A} - \mathbf{R}\| = \sqrt{\sum_{i=1}^{3}\sum_{j=1}^{3} |a_{ij} - r_{ij}|^2}. \tag{4.9}$$

4.1.1.1 Gram-Schmidt Orthogonalization

Gram-Schmidt orthogonalization is one of the simplest methods to extract the rotational part $\mathbf{R}$ from matrix $\mathbf{A}$ [Müller et al., 2016]. Let $\mathbf{R} = [\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3]$, and $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3]$, where $\mathbf{r}_i$ and $\mathbf{a}_i$ are the ith columns of matrix $\mathbf{R}$ and $\mathbf{A}$, respectively, then $\mathbf{R}$ can be constructed as follows:

$$\begin{aligned} \tilde{\mathbf{r}}_1 &= \mathbf{a}_1, & \mathbf{r}_1 &= \frac{\tilde{\mathbf{r}}_1}{\|\tilde{\mathbf{r}}_1\|}, \\ \tilde{\mathbf{r}}_2 &= \mathbf{a}_2 - \frac{\tilde{\mathbf{r}}_1 \bullet \mathbf{a}_2}{\tilde{\mathbf{r}}_1 \bullet \tilde{\mathbf{r}}_1}\tilde{\mathbf{r}}_1, & \mathbf{r}_2 &= \frac{\tilde{\mathbf{r}}_2}{\|\tilde{\mathbf{r}}_2\|}, \\ \tilde{\mathbf{r}}_3 &= \mathbf{r}_1 \times \mathbf{r}_2, & \mathbf{r}_3 &= \frac{\tilde{\mathbf{r}}_3}{\|\tilde{\mathbf{r}}_3\|}. \end{aligned} \tag{4.10}$$

As noted by Müller et al. [2016], the result depends on the order in which the columns of $\mathbf{A}$ are processed, thereby introducing a bias which causes “ghost forces”.

4.1.1.2 Polar Decomposition

Perhaps polar decomposition is the most popular approach in computer graphics, based on which Müller et al. [2005] extracted the rotational part $\mathbf{R}$ from the covariance matrix $\mathbf{A}$ of a body. The polar decomposition of $\mathbf{A}$ is defined by

$$\mathbf{A} = \mathbf{R}\underbrace{\sqrt{\mathbf{A}^T\mathbf{A}}}_{\text{symmetric}}. \tag{4.11}$$

Let $\mathbf{B} = \mathbf{A}^T\mathbf{A}$ and $\mathbf{S} = \sqrt{\mathbf{B}}$, the rotational part

$$\mathbf{R} = \mathbf{A}\left(\sqrt{\mathbf{A}^T\mathbf{A}}\right)^{-1} = \mathbf{A}\left(\sqrt{\mathbf{B}}\right)^{-1} = \mathbf{A}\mathbf{S}^{-1}. \tag{4.12}$$

To compute $\mathbf{S}$, a diagonal matrix $\mathbf{D}$ will be found such that $\mathbf{B} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1}$, where $\mathbf{V}$ is a matrix with the three eigenvectors of $\mathbf{B}$ as columns. Then

$$\mathbf{S} = \mathbf{V}\sqrt{\mathbf{D}}\mathbf{V}^{-1}. \tag{4.13}$$

The correctness can be immediately verified by the fact that

$$\mathbf{S}^2 = (\mathbf{V}\sqrt{\mathbf{D}}\mathbf{V}^{-1})^2 = \mathbf{V}\sqrt{\mathbf{D}}(\mathbf{V}^{-1}\mathbf{V})\sqrt{\mathbf{D}}\mathbf{V}^{-1} = \mathbf{V}\mathbf{D}\mathbf{V}^{-1} = \mathbf{B}. \tag{4.14}$$

The diagonal matrix $\mathbf{D}$ and the eigenvectors of $\mathbf{B}$ can be found by a few iterations of Jacobi rotations. Since $\mathbf{S}$ is a 3×3 matrix, the inverse of $\mathbf{S}$ can be easily found.

4.1.1.3 Yet Another Method

Müller et al. [2016] introduced a novel method which is much simpler and more effective. Instead of directly solving for the optimal orthonormal matrix $\mathbf{R}$, they solved the problem through the use of an update rule

$$\mathbf{R} \longleftarrow \exp(\boldsymbol{\omega})\mathbf{R}, \tag{4.15}$$

with $\tilde{\mathbf{R}}$ being an initial approximation which is already available, and the term $\exp(\boldsymbol{\omega})$ is an exponential map, i.e. a rotation matrix with axis $\dfrac{\boldsymbol{\omega}}{|\boldsymbol{\omega}|}$ and angle $|\boldsymbol{\omega}|$.

For simulations, a reasonable choice for the initial approximation $\tilde{\mathbf{R}}$ could be the solution of the last time step. The ideal choice for the direction of $\boldsymbol{\omega}$ is the one about which the Frobenius-norm $F(\tilde{\mathbf{R}})$ decreases the most.

Let $\tilde{\mathbf{R}} = [\tilde{\mathbf{r}}_1, \tilde{\mathbf{r}}_2, \tilde{\mathbf{r}}_3]$, and $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \mathbf{a}_3]$, where $\tilde{\mathbf{r}}_i$ and $\mathbf{a}_i$ are the ith columns of matrix $\tilde{\mathbf{R}}$ and $\mathbf{A}$, respectively. Let

$$U_F = \frac{1}{2}F^2(\tilde{\mathbf{R}}) = \frac{1}{2}\sum_{i=1}^{3}(\tilde{\mathbf{r}}_i - \mathbf{a}_i)^2 \tag{4.16}$$

be the potential acting on the rigid body represented by the rotation $\tilde{\mathbf{R}}$, which produces the forces acting at the tips of the axes of $\tilde{\mathbf{R}}$

$$\mathbf{f}_i = -\frac{\partial U_F}{\partial \tilde{\mathbf{r}}_i} = \mathbf{a}_i - \tilde{\mathbf{r}}_i, \quad i \in \{1, 2, 3\} \tag{4.17}$$

The total torque acting on $\tilde{\mathbf{R}}$ due to these forces is

$$\boldsymbol{\tau} = \sum_{i=1}^{3}\tilde{\mathbf{r}}_i \times (\mathbf{a}_i - \tilde{\mathbf{r}}_i) = \sum_{i=1}^{3}\tilde{\mathbf{r}}_i \times \mathbf{a}_i. \tag{4.18}$$

To minimize the Frobenius-norm, the direction of $\boldsymbol{\omega}$ should be the same as $\boldsymbol{\tau}$. Let ϕ be the angle between the vectors $\tilde{\mathbf{r}}_i$ and $\mathbf{a}_i$. After being rotated by $\boldsymbol{\omega}$, $\mathbf{r}_i$ and $\mathbf{a}_i$ should be aligned to make $\boldsymbol{\tau} = \mathbf{0}$, which can be achieved by

$$\boldsymbol{\omega} = \frac{\tilde{\mathbf{r}}_i \times \mathbf{a}_i}{\tilde{\mathbf{r}}_i \bullet \mathbf{a}_i} \Longrightarrow |\boldsymbol{\omega}| = \frac{|\tilde{\mathbf{r}}_i||\mathbf{a}_i|\sin\phi}{|\tilde{\mathbf{r}}_i||\mathbf{a}_i|\cos\phi} = \tan\phi \approx \phi. \tag{4.19}$$

Therefore, given the initial approximation $\mathbf{R} = \tilde{\mathbf{R}}$, the final iterative formula is given by

$$\mathbf{R} \longleftarrow \exp\left(\frac{\sum\limits_{i=1}^{3}\mathbf{r}_i \times \mathbf{a}_i}{\left|\sum\limits_{i=1}^{3}\mathbf{r}_i \bullet \mathbf{a}_i\right| + \varepsilon}\right)\mathbf{R}, \tag{4.20}$$

where ε is a very small positive constant. The absolute operator in the denominator is used to prevent the flip of the torque direction.

Despite the simple form, this method is very efficient for simulations where the rotation in the last step is already available, and can robustly handle a rank deficient matrix $\mathbf{A}$.

4.1.2 An Example

The concept of shape matching will be clarified through a simple two-dimensional example in Fig. 4.1, where the rigid body $\mathcal{X} = \{\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3\}$ consists of four particles.

It is to be noted that Fig. 4.1 is not just an schematic representation of the idea of shape matching; instead, the positions of the particles are exactly calculated using the shape matching algorithm, with the z-axis being the rotation axis.

In Fig. 4.1,

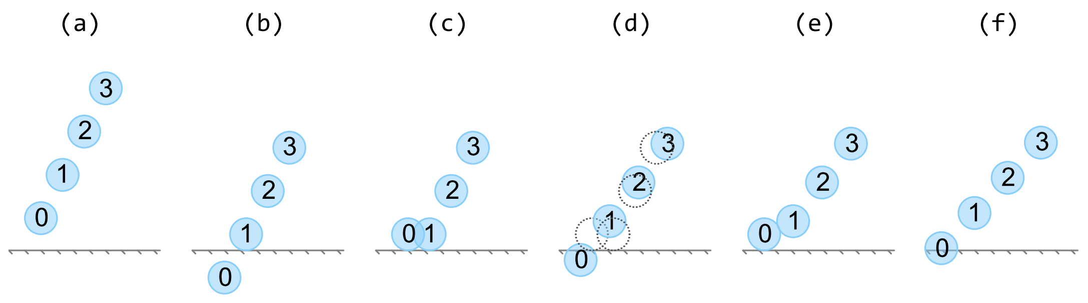


Fig. 4.1: A two-dimensional shape matching example

(a) At the beginning of the simulation, the rigid body $\mathcal{X}$ is falling towards the ground with high velocity;
(b) After the prediction step, $\mathcal{X}$ comes into penetrating contact with the ground;
(c) During contact handling of the first solver iteration, particle 0 of $\mathcal{X}$ is projected onto the ground;
(d) After shape matching being applied to $\mathcal{X}$, the shape is restored; the new configuration of $\mathcal{X}$ can be seen as a rotated and translated version of the configuration right after the prediction step (Fig. 4.1 b), which mitigates the inter-penetration. The dotted circles indicate the configuration before shape matching (Fig. 4.1 c);
(e) A second iteration is necessary since the penetration has not been completely resolved by last iteration. After contact handling of the second iteration, particle 0 is projected onto the ground again;
(f) The shape matching restores the shape of $\mathcal{X}$, but particle 0 is still slightly penetrating the ground, thus additional iterations remain necessary.

One problem with the shape matching algorithm is the inherent under-relaxation due to the minimum mean squared error (MMSE) estimator for the optimal linear transformation (Eq. 4.5 and Eq. 4.6). For example, in Fig. 4.1 d), particle 0 is projected under the ground by the shape matching constraint.

Indeed, to minimize the term in Eq. (4.5), the positions of the four particles have to be adjusted accordingly. However, the weight is too small for particle 0 to prevent the error caused by the other three particles from being distributed to it, which leads to another penetration and eventually slow convergence.

One natural thought is to adjust the weights of the rigid body particles for shape matching based on the position correction during the process of the contact constraints. Specifically, larger displacement corresponds to heavier weight, which makes it harder for non-penetrating particles to cancel out the corrections by contact handling.

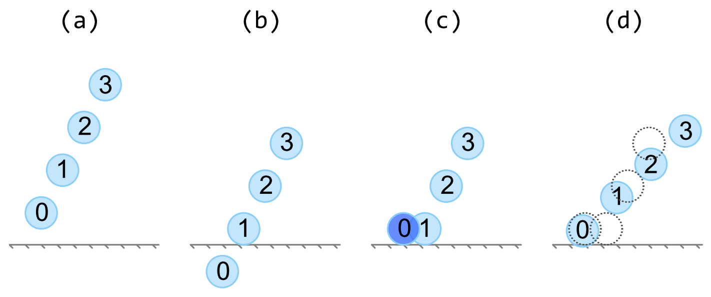


Fig. 4.2: Shape matching with adaptive weighting

In Fig. 4.2 c), the weight of particle 0 is increased to 32 (an experimentally determined value) during the handling of the shape matching constraint, while the weights for the other three particles remains to be 1. With only one solver iteration, the shape is restored, meanwhile the penetration is almost completely resolved (Fig. 4.2 d). In fact, particle 0 is barely moved by the shape matching constraint. The convergence could be largely accelerated for large shapes consisting of a huge amount of particles.

Compared to the fixed weighting in Fig. 4.1, the adaptive weighting in Fig. 4.2 moves the rotation center closer to particle 0, resulting in a more realistic visual effect when the friction between the rigid body and other objects is strong. However, overweighted particles might cause implausible rotations when the friction is supposed to be very weak. Therefore, instead of blindly changing the weights in the simulation of complex scenes, a more apposite method which takes all kinds of conditions into consideration remains to be found.

Another problem with the shape matching algorithm arises when dealing with deformable objects. Deformable objects exhibit higher stiffness as the number of the solver iterations increases, because the particles are pulled closer to their goal positions by the shape matching constraint in each iteration. This problem can be solved either by solving different types of constraints with different frequencies instead of a global frequency for all constraints, as suggested by Macklin et al. [2014], or by applying the shape matching constraint only when the difference between the current position and the goal position of a particle exceeds some threshold derived from the stiffness parameter α.

Besides, the degree of the deformation decreases with smaller time steps as less penetration can be tolerated by the system. Müller et al. [2005] solved this problem by setting $\alpha = h/\tau$, where h is the time step size and τ is a time constant.

4.2 Collision Handling

The method introduced in Section 3.2.2.2 for solving the collision between two free particles such as granular-like materials cannot stop two rigid bodies from being interlocked once tunneling occurs (Fig. 4.3).

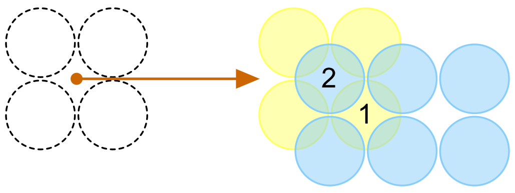


Fig. 4.3: An example given by Macklin et al. [2014]. In this example, one rigid body penetrates another rigid body due to high velocity. Both particle 1 and particle 2 will not be projected out of the other shape if the collision is handled in the same way as two free particles, which only takes local collisions between single particles into consideration. In fact, the sum of the position corrections due to the four neighbors is $\mathbf{0}$, thus causes interlock.

To resolve this problem, Macklin et al. [2014] used signed distance field (SDF) to determine the direction and magnitude of the position corrections. the signed distance at the points inside the rigid body are defined as negative values, and outside the rigid body as positive values. Therefore, the gradients point outside the rigid body. The magnitude of the SDF for each particle is the distance from the center of the particle to the surface of the rigid body.

By sampling the SDF magnitude φ and the SDF gradient $\nabla\varphi$ onto each particle belonging to a rigid shape, Macklin et al. [2014] are able to generate the contact constraints between particles of different shapes in the same way as free particles, and to process these constraints similarly:

$$\begin{aligned}\Delta\mathbf{x}_i &= -\frac{w_i}{w_i + w_j}(d \cdot \mathbf{n}_{ij}),\\ \Delta\mathbf{x}_j &= +\frac{w_j}{w_i + w_j}(d \cdot \mathbf{n}_{ij}),\end{aligned} \tag{4.21}$$

where $d = \min(|\varphi_i|, |\varphi_j|)$, and

$$\mathbf{n}_{ij} = \begin{cases} \nabla\varphi_i & \text{if } |\varphi_i| < |\varphi_j|,\\ -\nabla\varphi_j & \text{otherwise.}\end{cases} \tag{4.22}$$

However, due to the strong under-relaxation introduced by the shape matching constraints (Section 4.1.2), we note that letting $d = \max(|\varphi_i|, |\varphi_j|)$ leads to a faster convergence. The gradient is still chosen according to Eq. (4.22). In cases when $\nabla\varphi$ is undefined for some particles inside a shape, an arbitrary unit vector is used.

The under-sampling of the SDF near the surface of a shape leads to discontinuous penetration depth and jitter. To address this issue, Macklin et al. [2014] modified the contact normal when one of two colliding particles lies on the boundary of a shape (i.e. $|\varphi| <$ boundary particle radius) as follows:

$$\mathbf{n}^*_{ij} = \begin{cases} \mathbf{x}_{ij} - 2(\mathbf{x}_{ij} \bullet \mathbf{n}_{ij})\mathbf{n}_{ij} & \text{if } \mathbf{x}_{ij} \bullet \mathbf{n}_{ij} < 0,\\ \mathbf{x}_{ij} & \text{otherwise,}\end{cases} \tag{4.23}$$

and $d = r_i + r_j - |\mathbf{x}_i - \mathbf{x}_j|$, where $\mathbf{x}_{ij} = -\dfrac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|}$.

When only one particle i of two colliding particles belongs to a rigid body and the other one j is a free particle, we use $d = |\varphi_i|$ and the contact normal

$$\mathbf{n}_{ij} = -\frac{\mathbf{x}_i - \mathbf{x}_j}{|\mathbf{x}_i - \mathbf{x}_j|}. \tag{4.24}$$

If i is a boundary particle, the modified contact normal is calculated according to Eq. (4.23), and $d = r_i + r_j - |\mathbf{x}_i - \mathbf{x}_j|$.

4.2.1 Mass Scaling

Large piles of rigid bodies converge very slow when iterative methods are used because the local propagation of the position corrections by contact constraints is very slow (e.g. one particle per iteration for Jacobi method), thus many iterations are needed to stop the oscillation of the rigid piles, making them visually plausible.

Noticing that heavier particles are more resistant to the projection by collision constraints, Macklin et al. [2014] modified the mass of the particles temporarily (Line 4 of Algorithm 3.3) such that particles with low potential energy are heavier than the particles with high potential energy to achieve faster convergence. The scaling factor are determined by the following equation:

$$s_i(\mathbf{x}_i) = e^{-kh(\mathbf{x}_i)}, \tag{4.25}$$

where k can be considered as the speed of error propagation. $h(\mathbf{x}_i)$ is the height function which is inversely proportional to the potential energy. When the piles are maintained by the gravity, which is almost always the case, the height (i.e. the y-coordinate) of the particle i can be used. The modified mass is given by $m_i^* = s_i m_i$. After the contact handling, the mass are restored (Line 22 of Algorithm 3.3).

This method can also effectively prevent granular piles from fast collapsing due to the additional kinematic energy introduced by the slow local propagation of the position corrections by contact constraints.

4.3 Solid Voxelization

To generate particles from a triangle mesh, the first step is to perform a solid voxelization of the triangle mesh. To exploit the huge computational capacity of the GPU, we implemented the triangle-parallel solid voxelization algorithm proposed by Schwarz and Seidel [2010].

One basic operation of their algorithm is to test the overlap between the projection of each triangle face and the projection of the center of each voxel on one coordinate plane (i.e. xy-plane, yz-plane or zx-plane), for which they utilized the edge function.

4.3.1 The Edge Function

The *edge function* is a linear function to classify points on a two dimensional plane divided by a line into three regions: the points to the left of the line, the points to the right of the line, and the points on the line [Pineda, 1988].

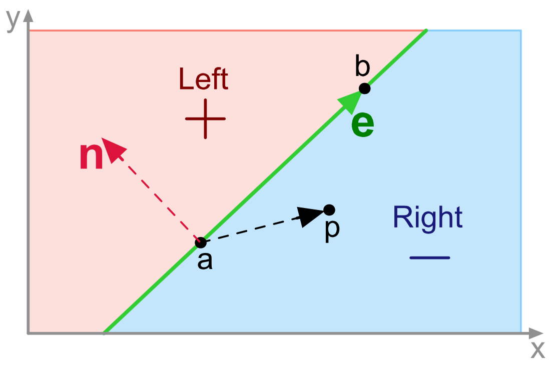


Fig. 4.4: edge function

In Fig. 4.4, the line $\mathcal{L}$ is defined as

$$\mathbf{x}_a + d \cdot \mathbf{e} = \mathbf{x}_a + d \cdot [\mathbf{e}_x, \mathbf{e}_y]^T = \mathbf{x}_a + d \cdot (\mathbf{x}_b - \mathbf{x}_a), \quad -\infty < d < \infty, \tag{4.26}$$

where a and b are two arbitrary points on the line $\mathcal{L}$, and ray $\mathcal{R}$ is the positive part (i.e. $d > 0$) of line $\mathcal{L}$. The left side of the line is the region swept by ray $\mathcal{R}$ when rotating ray $\mathcal{R}$ counterclockwise about its initial point a by π radians, and the right side of the line is the region swept by ray $\mathcal{R}$ when rotating ray $\mathcal{R}$ clockwise about its initial point a by π radians.

Vector $\mathbf{n} = [-\mathbf{e}_y, \mathbf{e}_x]^T$ is normal to edge $\mathbf{e}$. Vector $\mathbf{p} = \mathbf{x}_p - \mathbf{x}_a$, where p is an arbitrary point on the xy-plane. The edge function of line $\mathcal{L}$ is defined by

$$f_{\mathcal{L}}(\mathbf{x}_p) = \mathbf{p} \bullet \mathbf{n} = \mathbf{x}_p \bullet \mathbf{n} - \mathbf{x}_a \bullet \mathbf{n}. \tag{4.27}$$

Let α be the angle between $\mathbf{p}$ and $\mathbf{n}$, an interesting property of the edge function is

$$f_{\mathcal{L}}(\mathbf{x}_p) \begin{cases} > 0 \text{ if } p \text{ lies on the left side of line } \mathcal{L}, \text{ i.e. } |\alpha| < \dfrac{\pi}{2} \\ = 0 \text{ if } p \text{ lies on the line } \mathcal{L}, \text{ i.e. } |\alpha| = \dfrac{\pi}{2} \\ < 0 \text{ if } p \text{ lies on the right side of line } \mathcal{L}, \text{ i.e. } |\alpha| > \dfrac{\pi}{2} \end{cases} \tag{4.28}$$

4.3.2 Overlap Test

Let $\triangle \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ be the projection of a triangle face $\triangle \mathbf{x}_0\mathbf{x}_1\mathbf{x}_2$ with face normal $\mathbf{n} = [\mathbf{n}^x, \mathbf{n}^y, \mathbf{n}^z]^T$ on the yz-plane, as illustrated in Fig. 4.5. The edges of $\triangle \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$ are defined by

$$\mathbf{e}_i = \mathbf{v}_{(i+1) \bmod 3} - \mathbf{v}_i = [\mathbf{e}_i^y, \mathbf{e}_i^z]^T, \quad i \in \{0, 1, 2\}, \tag{4.29}$$

and the normal for each edge is defined by

$$\mathbf{n}_{\mathbf{e}_i} = [\mathbf{n}_{\mathbf{e}_i}^y, \mathbf{n}_{\mathbf{e}_i}^z] = [-\mathbf{e}_i^z, \mathbf{e}_i^y]^T \cdot \begin{Bmatrix} 1, & \mathbf{n}_x \geq 0 \\ -1, & \mathbf{n}_x < 0 \end{Bmatrix}, \quad i \in \{0, 1, 2\}, \tag{4.30}$$

which ensures that all of these normals always point to the inside of the triangle $\triangle \mathbf{v}_0\mathbf{v}_1\mathbf{v}_2$, regardless of the orientation of the triangle face $\triangle \mathbf{x}_0\mathbf{x}_1\mathbf{x}_2$.

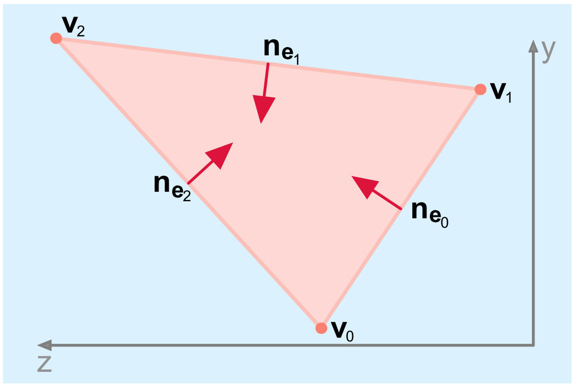

$\mathbf{v}_2$
$\mathbf{n}_{\mathbf{e}_1}$
$\mathbf{v}_1$
y
$\mathbf{n}_{\mathbf{e}_2}$
$\mathbf{n}_{\mathbf{e}_0}$
$\mathbf{v}_0$
z

Fig. 4.5: The projection of a mesh triangle on the yz-plane.

Let $d_{\mathbf{e}_i} = -\mathbf{v}_i \bullet \mathbf{n}_{\mathbf{e}_i}$ for each edge $\mathbf{e}_i$, and $\mathbf{p} = [\mathbf{p}^y, \mathbf{p}^z]^T$ be the projection of the center of a voxel $\mathbf{q}$ on the yz-plane, the overlap test for triangle face $\triangle \mathbf{x}_0\mathbf{x}_1\mathbf{x}_2$ and voxel center $\mathbf{q}$ is defined by

$$f_{\triangle \mathbf{x}_0\mathbf{x}_1\mathbf{x}_2}(\mathbf{p}) = \bigwedge (\mathbf{p} \bullet \mathbf{n}_{\mathbf{e}_i} + d_{\mathbf{e}_i} \geq 0), \quad \forall i \in \{0, 1, 2\}. \tag{4.31}$$

$f_{\triangle \mathbf{x}_0\mathbf{x}_1\mathbf{x}_2}(\mathbf{p}) = \textit{true}$ indicates an overlap.

4.3.3 The Algorithm

The first step of voxelizing a mesh is to uniformly scale and translate the mesh so that it fits exactly into an $n_x \times n_y \times n_z$ volume of voxels defined by the two corner points $\mathbf{b}_{min} = [0, 0, 0]^T$ and $\mathbf{b}_{max} = [n_x, n_y, n_z]^T$, where each voxel is represented by one bit and initialized with 0 meaning that the voxel is not occupied by the mesh. By doing so the numerical error caused by the limited precision of the floating point number can be avoided. The edge length of each voxel is exactly 1, which simplifies the calculation to some extent.

Then each triangle face is projected on the yz-plane, and the AABB will be calculated, from which the yz range of the covered voxels can be derived.

If the range is non-empty, the projection of the center $\mathbf{p}_{(j,k)}$ of each contained voxel column (j, k) on the yz-plane will be tested against the projection of the triangle face on the yz-plane using the function in Eq. (4.31).

Each $\mathbf{p}_{(j,k)}$ which passes the overlap test (i.e. the function returns *true*) will be projected along the x-axis to the point $\mathbf{p}_{(j,k),\triangle}$ on the triangle's plane. Let

$$n_x = \left\lfloor \mathbf{p}^x_{(j,k),\triangle} + \frac{1}{2} \right\rfloor - 1, \tag{4.32}$$

where $\mathbf{p}^x_{(j,k),\triangle}$ is the x-coordinate of the point $\mathbf{p}_{(j,k),\triangle}$, then for each voxel (i, j, k), where $0 \le i \le n_x$, the corresponding bit will be flipped. The flip operation is performed by the atomic XOR function. For a voxel occupied by the mesh, the number of the flip operation is odd, thus the final value of the corresponding bit is 1. It is to be noted that for a meaningful voxelization using this algorithm, the mesh has to be closed.

One problem with the overlap test $f_{\triangle \mathbf{x}_0\mathbf{x}_1\mathbf{x}_2}(\mathbf{p})$ in Eq. (4.31) is that if an edge or a vertex shared by two triangle faces overlaps the same voxel center $\mathbf{q}$, both tests will return true, which leads to an incorrect conclusion about the voxel. To avoid this problem, Schwarz and Seidel [2010] only took the top edges ($\mathbf{n}^y_{\mathbf{e}_i} > 0$) and the right edges ($\mathbf{n}^y_{\mathbf{e}_i} = 0 \wedge \mathbf{n}^z_{\mathbf{e}_i} < 0$) into consideration. Concretely, they defined the term

$$f_{\mathbf{e}_i} = \left\{ \begin{array}{l} \epsilon, \ \mathbf{n}^y_{\mathbf{e}_i} > 0 \bigvee \left(\mathbf{n}^y_{\mathbf{e}_i} = 0 \bigwedge \mathbf{n}^z_{\mathbf{e}_i} < 0 \right) \\ 0, \ \text{otherwise} \end{array} \right\}, \quad i \in \{0, 1, 2\}, \tag{4.33}$$

where ϵ is the machine epsilon for the chosen floating point type, and incorporated it into Eq. (4.31). The modified overlap test

$$f^*_{\triangle \mathbf{x}_0\mathbf{x}_1\mathbf{x}_2}(\mathbf{p}) = \bigwedge \left((\mathbf{p} \bullet \mathbf{n}_{\mathbf{e}_i} + d_{\mathbf{e}_i}) + f_{\mathbf{e}_i} > 0 \right), \quad \forall i \in \{0, 1, 2\}. \tag{4.34}$$

The parallelism is realized by assigning each triangle to one CUDA thread. To potentially increase the parallelism, we choose the projection plane adaptively. Specifically, let $\mathbf{d} = [\mathbf{d}^x, \mathbf{d}^y, \mathbf{d}^z]^T$ be the dimensions of the AABB of the mesh to be voxelized, then the ij-plane will be chosen as the projection plane such that the area of the projection of the AABB will be maximized, namely

$$ij = \arg\max_{ij}(\mathbf{d}^i \cdot \mathbf{d}^j), \quad ij \in \{xy, yz, zx\}. \tag{4.35}$$

When the zx-plane is chosen, the first coordinate of the projection $\mathbf{v}_k$ of the vertex k at $\mathbf{x}_k$ is the z-coordinate of $\mathbf{x}_k$, i.e. $\mathbf{v}_k = [\mathbf{x}_k^z, \mathbf{x}_k^x]^T$.

Schwarz and Seidel [2010] also proposed a tile-based solid voxelization algorithm to mitigate the potential under-utilization caused by the triangle-level parallelism. However, the 3D models in this project are generally not very complicated. In fact, the most complicated one we used – the Stanford dragon model contains no more than 48k triangles. The dimension of the volume is usually less than 128^3 and the voxelization is only performed once during initialization, therefore we believe that tile-based optimization is unnecessary and might be less efficient than the triangle-parallel algorithm [Schwarz and Seidel, 2010].

Their sparse solid voxelization which performs the voxelization directly into a sparse hierarchical structure is very space-efficient and seems very decent for the purpose of generating particles of different sizes. Unfortunately the resulting data structure does not directly fit into the algorithm we used for the construction of the SDF, and this kind of voxelization may lead to severe under-sampling near the boundary of the mesh. Therefore, instead of using the sparse voxelization method, we generate differently sized particles by merging voxels based on the magnitude of their signed distances.

4.4 SDF Construction

The SDF of an $n_1 \times n_2 \times n_3$ volume of voxels is constructed through distance transform, for which we implemented a simplified version of the parallel banding algorithm proposed by Cao et al. [2010], namely a row- or column-parallel algorithm without partitioning the volume into small bands and then merging the results, as the dimensions of the volume are generally small (less than 128^3).

4.4.1 General Approach

The fundamental idea for the general approach for exact distance transform is traced back to Kolountzakis and Kutulakos [1992], and further extended by Hayashi et al. [1998], Lee et al. [2003], Maurer et al. [2003] and Cao et al. [2010]. Here we mainly follow the notation and formulation proposed by Cao et al. [2010], start with the two dimensional case, and then extend the same idea to three dimensions.

Before more detailed description we would like to precisely define some terms heavily used in the following discussion:

Definition 4.4.1. *Row* j of a 2D image consists of all the pixels with the same y-coordinate j; *column* i of a 2D image consists of all the pixels with the same x-coordinate i; *row* (j, k) of a 3D volume consists of all the voxels with the same y-coordinate j and z-coordinate k; *column* (i, k) of a 3D volume consists of all the voxels with the same x-coordinate i and z-coordinate k; *aisle* (i, j) of a 3D volume consists of all the voxels with the same x-coordinate i and the same y-coordinate j; *slice* k of a 3D volume consists of all the voxels with the same z-coordinate k.

4.4.1.1 Two Dimensional Distance Transform

Given a 2D binary image of size $N = n_1 \times n_2$, a site $\mathbf{s}$ is termed a *2D proximate site* of column i if the Voronoi region of $\mathbf{s}$ intersects the pixels in column i. Let $\mathbf{s}_{i,j}^{1D}$ be the closest site to the pixel (i, j) in row j ($\mathbf{s}_{i,j}^{1D} = \text{NULL}$ if there is no site in row j), $\mathcal{S}_i^{2D} = \{\mathbf{s}_{i,j}^{1D} : \mathbf{s}_{i,j}^{1D} \neq \text{NULL}, 0 \leq j \leq n_2 - 1\}$ be the collection of such closest sites for all pixels in column i, $\mathcal{P}_i^{2D}$ be the set of the proximate sites of column i, and the site with smaller coordinate be the closest site in cases when a pixel is equidistant from two sites, then the following three propositions hold (cf. Fig. 4.6):

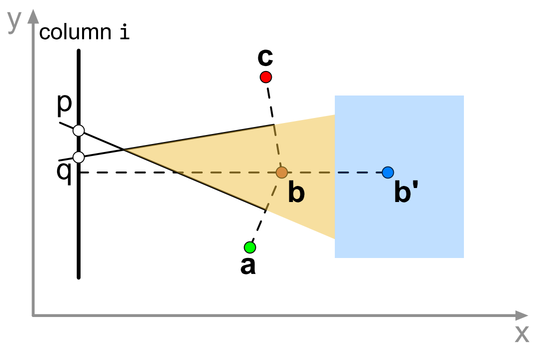


Fig. 4.6: Given four sites **a**, **b**, **b′**, and **c**, the perpendicular bisector of segment $\overline{\mathbf{ab}}$ intersects column i at p, and the perpendicular bisector of segment $\overline{\mathbf{bc}}$ intersects column i at q.

Proposition 4.4.2. *Let* $\mathbf{b}$ *at* (i_1, j) *and* $\mathbf{b}'$ *at* (i_2, j) *be two sites in the same row* j*. The Voronoi region of* $\mathbf{b}'$ *cannot intersect column* i *if* $|i_1 - i| < |i_2 - i|$*.*

Proposition 4.4.3. *Let* $\mathbf{a}$ *at* (i_1, j_1)*,* $\mathbf{b}$ *at* (i_2, j_2) *and* $\mathbf{c}$ *at* (i_3, j_3) *be three sites with* $j_1 < j_2 < j_3$*. Assuming that the perpendicular bisector of segment* $\overline{\mathbf{ab}}$ *intersects column* i *at* $p(i, u)$*, and the perpendicular bisector of segment* $\overline{\mathbf{bc}}$ *intersects column* i *at* $q(i, v)$*, then the Voronoi region of* $\mathbf{b}$ *cannot intersect column* i *if* $u > v$*, therefore* $\mathbf{b} \notin \mathcal{P}_i^{2D}$*. In this case, it is said that* a *and* c dominate b*.*

Proposition 4.4.4. *Let* $q'(i, v)$ *and* $p'(i, u)$ *be two pixels in column* i *such that* $u > v$*,* $\mathbf{a}$ *at* (i_1, j_1) *and* $\mathbf{c}$ *at* (i_3, j_3) *be the closest sites to* q' *and* p' *respectively, then* $j_1 \leq j_3$*.*

Proposition 4.4.2 indicates that there is no more than one site along a row that can potentially be a proximate site of column i. Namely, $\mathcal{P}_i^{2D} \subseteq \mathcal{S}_i^{2D}$, thus $\left|\mathcal{P}_i^{2D}\right| \leq n_2$.

Proposition 4.4.4 implies that the Voronoi regions of the proximate sites of column i appear in exactly the same order as when sorting the approximate sites by their y-coordinates.

With these three propositions, the exact distance transform for a 2D image can be performed as follows:

(1) For each pixel (i, j), compute $\mathbf{s}_{i,j}^{1D}$ (1D Voronoi diagram per row);
(2) Compute the set $\mathcal{P}_i^{2D}$ for each column i using $\mathcal{S}_i^{2D}$;
(3) For each pixel (i, j), compute the closest site using $\mathcal{P}_i^{2D}$.

4.4.1.2 Extension to Three Dimensions

Given a volume of size $N = n_1 \times n_2 \times n_3$, a site $\mathbf{s}$ is termed a *3D proximate site* of aisle (i, j) if the Voronoi region of $\mathbf{s}$ intersects the voxels in aisle (i, j). Let $\mathbf{s}_{i,j,k}^{2D}$ be the closest site to the voxel (i, j, k) on slice k ($\mathbf{s}_{i,j,k}^{2D} = \text{NULL}$ if there is no site on slice k), $\mathcal{S}_{i,j}^{3D} = \{\mathbf{s}_{i,j,k}^{2D} : \mathbf{s}_{i,j,k}^{2D} \neq \text{NULL}, 0 \leq k \leq n_3 - 1\}$ be the collection of such closest sites for all voxels in aisle (i, j), $\mathcal{P}_{i,j}^{3D}$ be the set of the proximate sites of aisle (i, j), and the site with smaller coordinate be the closest site in cases when a voxel is equidistant from two sites, then all of these three propositions above still hold if each "row" of a 2D image is replaced by each "slice" of a 3D volume, and each "column" of the 2D image is replaced by each "aisle" of the volume. Precisely,

Proposition 4.4.5. *Let* $\mathbf{b}$ *at* (i_1, j_1, k) *and* $\mathbf{b}'$ *at* (i_2, j_2, k) *be two sites in the same slice* k*. The Voronoi region of* $\mathbf{b}'$ *cannot intersect aisle* (i, j) *if* $\|(i_1 - i, j_1 - j)\| < \|(i_2 - i, j_2 - j)\|$*.*

Proposition 4.4.6. *Let* $\mathbf{a}$ *at* (i_1, j_1, k_1)*,* $\mathbf{b}$ *at* (i_2, j_2, k_2) *and* $\mathbf{c}$ *at* (i_3, j_3, k_3) *be three sites with* $k_1 < k_2 < k_3$*. Assuming that the perpendicular bisecting plane of segment* $\overline{\mathbf{ab}}$ *intersects aisle* (i, j) *at* $p(i, j, u)$*, and the perpendicular bisecting plane of segment* $\overline{\mathbf{bc}}$ *intersects aisle* (i, j) *at* $q(i, j, v)$*, then the Voronoi region of* $\mathbf{b}$ *cannot intersect aisle* (i, j) *if* $u > v$*, therefore* $\mathbf{b} \notin \mathcal{P}^{3D}_{i,j}$*. In this case, it is said that a and c* dominate *b.*

Proposition 4.4.7. *Let* $q'(i, j, v)$ *and* $p'(i, j, u)$ *be two voxels in aisle* (i, j) *such that* $u > v$*,* $\mathbf{a}$ *at* (i_1, j_1, k_1) *and* $\mathbf{c}$ *at* (i_3, j_3, k_3) *be the closest sites to* q' *and* p' *respectively, then* $k_1 \le k_3$*.*

Given all the propositions above, the exact distance transform for a 3D volume can be done in the following five steps:

(1) For each voxel (i, j, k) on each slice k, compute $\mathbf{s}^{1D}_{i,j,k}$ (1D Voronoi diagram per row);
(2) Compute the set $\mathcal{P}^{2D}_{i,k}$ for each column (i, k) on each slice k using $\mathcal{S}^{2D}_{i,k}$;
(3) For each voxel (i, j, k), compute $\mathbf{s}^{2D}_{i,j,k}$ using $\mathcal{P}^{2D}_{i,k}$ (2D Voronoi diagram for each slice);
(4) Compute the set $\mathcal{P}^{3D}_{i,j}$ for each aisle (i, j) of the volume using $\mathcal{S}^{3D}_{i,j}$;
(5) For each voxel (i, j, k), compute the closest site using $\mathcal{P}^{3D}_{i,j}$.

4.4.2 The Algorithm

To compute the distance transform of a volume of size $N = n_1 \times n_2 \times n_3$, an array of the same size is allocated, with each element at (i, j, k) being a 3D vector where the index of the closest site to voxel $p(i, j, k)$ will be stored.

Let $\mathcal{N}_c(p(i, j, k)) = \{q(i + d_i, j + d_j, k + d_k) : \ |q| \neq |p|, \ -1 \le d_i, d_j, d_k \le 1\}$ be the set of neighbors of voxel $p(i, j, k)$ whose bits are different from p. During the initialization (Algorithm 4.1), the sites (i.e. the voxels at the boundary of the mesh) will be found, and the magnitude of the SDF at a site $p(i, j, k)$ is determined during the initialization, which is half of the distance between the center of p and the center of the closest voxel $q \in \mathcal{N}_c(p)$ whose bit is different from p. During the calculation of the SDF, each voxel is assumed to be a unit cube, and will be scaled according to the actual size of the mesh during the generation of particles.

Algorithm 4.1 Distance Transform Step 0: Initialization

```
1: for each voxel $p(i,j,k)$ do
2:     if $\mathcal{N}_c(p) = \emptyset$ then
3:         $\mathbf{s}_{i,j,k} \leftarrow (-\infty, -\infty, -\infty)$        ▷ set the initial closest site to an impossible value
4:         $|\phi_p| \leftarrow \infty$                              ▷ $|\phi_p|$ is the magnitude of the SDF at $p(i,j,k)$
5:     else
6:         $\mathbf{s}_{i,j,k} \leftarrow (i,j,k)$                                          ▷ $p$ is a site
7:         $(i_q, j_q, k_q) \leftarrow \arg\min_{i_q,j_q,k_q} \|(i,j,k) - (i_q,j_q,k_q)\|, \quad \forall q(i_q,j_q,k_q) \in \mathcal{N}_c(p)$
8:         $|\phi_p| \leftarrow \frac{1}{2}\|(i,j,k) - (i_q,j_q,k_q)\|$
9:     end if
10: end for
```

In step 1, for each voxel $p(i, j, k)$, the closest site $\mathbf{s}^{1D}_{i,j,k}$ in the same row (j, k) will be determined, which is equivalent to constructing the Voronoi diagram of all the sites in row (j, k) with only the voxels in the same row being considered (cf. definition 2.6.2). We use a two-pass sweeping to find the closest site to each voxel $p(i, j, k)$ of each row (j, k) [Schneider et al., 2009]. In the first pass, the information of the closest site is propagated from left to right, and in the second pass from right to left (Algorithm 4.2).

For step 2, we use the sequential implementation proposed by Cao et al. [2010] for each column (i, k). To determine $\mathcal{P}^{2D}_{i,k}$, all the sites $\mathbf{s}^{1D}_{i,j,k}$ in $\mathcal{S}^{2D}_{i,k}$ are scanned in a bottom-up

Algorithm 4.2 Distance Transform Step 1: Voronoi Diagram in 1D

```
1: for each row $(j, k)$ do
2:     $\mathbf{s}_{curr} \leftarrow (-\infty, -\infty, -\infty)$
3:     for $i = 0, 1, \ldots, n_1 - 1$ do
4:         if $\|\mathbf{s}_{curr} - (i, j, k)\| < \|\mathbf{s}_{i,j,k} - (i, j, k)\|$ then
5:             $\mathbf{s}_{i,j,k} \leftarrow \mathbf{s}_{curr}$
6:         else
7:             $\mathbf{s}_{curr} \leftarrow \mathbf{s}_{i,j,k}$
8:         end if
9:     end for
10:
11:     $\mathbf{s}_{curr} \leftarrow (-\infty, -\infty, -\infty)$
12:     for $i = n_1 - 1, n_1 - 2, \ldots, 1, 0$ do
13:         if $\|\mathbf{s}_{curr} - (i, j, k)\| < \|\mathbf{s}_{i,j,k} - (i, j, k)\|$ then
14:             $\mathbf{s}_{i,j,k} \leftarrow \mathbf{s}_{curr}$
15:         else
16:             $\mathbf{s}_{curr} \leftarrow \mathbf{s}_{i,j,k}$
17:         end if
18:     end for
19: end for
```

fashion, from $j = 0$ to $j = n_2 - 1$, and meanwhile a stack of sites which are potentially proximate sites is maintained. When a new site $\mathbf{c}^{1D}_{i,j,k}$ in $\mathcal{S}^{2D}_{i,k}$ is reached, the top of the stack $\mathbf{b}^{1D}_{i,j,k}$ will be repeatedly popped out until it is no longer dominated by $\mathbf{c}^{1D}_{i,j,k}$ and the second top of the stack $\mathbf{a}^{1D}_{i,j,k}$ (cf. proposition 4.4.3). Then $\mathbf{c}^{1D}_{i,j,k}$ is pushed into the stack. It is to be noted that the stack grows upwards, with the index of the bottom being 0. At the end of the process, the stack contains $\mathcal{P}^{2D}_{i,k}$ (algorithm 4.3).

At the beginning of step 3, for each column (i, k), an index $I_{i,k}$ is set to 0. Starting from j = 0, for each voxel $p(i, j, k)$, $I_{i,k}$ will be repeatedly increased by one as long as the distance between the voxel p and the site at $I_{i,k}$ of the stack $\mathcal{P}^{2D}_{i,k}$ is larger than the distance between the voxel p and the site at $I_{i,k} + 1$. When the loop terminates, the site at $I_{i,k}$ is the closest site to voxel p on slice k due to proposition 4.4.4. Then the next voxel $p'(i, j + 1, k)$ is processed (Algorithm 4.4).

Step 4 (Algorithm 4.5) and step 5 (Algorithm 4.6) are proceeded in a similar way to step 2 and step 3, respectively.

After the closest site for each voxel is determined, the magnitude of the SDF at voxel $p(i, j, k)$ can be calculated as the distance between the center of p and the center of the closest site $\mathbf{s}_{i,j,k}$ plus the magnitude of the SDF at site $\mathbf{s}_{i,j,k}$ (Algorithm 4.7). The signed distance is negative for a voxel p occupied by the mesh, and positive otherwise. The gradient of the SDF at each voxel is calculated from the signed distance of its six direct neighbors. To ensure the accuracy at the boundary of the mesh, we extend the size of the volume by one in each direction ($\pm x$, $\pm y$, and $\pm z$).

4.4.3 Voxel Merging

To generate particles of different sizes and meanwhile keep a fine-grained sampling near the boundary of the mesh, we take an approach similar to the calculation of mipmaps, yet the voxels near the boundary of the mesh will not be merged.

Let $V^l(i, j, k)$ be the voxel of volume V^l at (i, j, k) with signed distance $\phi^l(i, j, k)$ and $\left|V^l\right|$

Algorithm 4.3 Distance Transform Step 2: Proximate Sites in 2D

```
 1: for each column $(i, k)$ do
 2:     $T_{i,k} \leftarrow -1$                    ▷ index of the top of stack $P_{i,k}$
 3:     for $j = 0, 1, \ldots, n_2 - 1$ do
 4:         $\mathbf{c} \leftarrow \mathbf{s}_{i,j,k}$
 5:         while $T_{i,k} > 0$ do
 6:             $\mathbf{b} \leftarrow P_{i,k}[T_{i,k}]$
 7:             $\mathbf{a} \leftarrow P_{i,k}[T_{i,k} - 1]$
 8:             if a and c dominate b then
 9:                 $T_{i,k} \leftarrow T_{i,k} - 1$
10:                 pop b out of $P_{i,k}$
11:             else
12:                 break
13:             end if
14:         end while
15:         push c to $P_{i,k}$
16:         $T_{i,k} \leftarrow T_{i,k} + 1$
17:     end for
18: end for
```

be the number of voxels in V^l whose bits are 1, where l is the mipmap level with $V^0 = V$, then the size of V^{l+1}

$$N^{l+1} = n_1^{l+1} \times n_2^{l+1} \times n_3^{l+1} = \left\lfloor \frac{n_1^l}{2} \right\rfloor \times \left\lfloor \frac{n_2^l}{2} \right\rfloor \times \left\lfloor \frac{n_3^l}{2} \right\rfloor \tag{4.36}$$

Let $\mathcal{S}_{i,j,k}^l = \{V^{l-1}(2i + d_i, 2j + d_j, 2k + d_k) : \ 0 \le d_i, d_j, d_k \le 1\}$ be the set of the 8 voxels covered by voxel $V^l(i, j, k)$. During merging its bit will be set to 1 if the signed distances of all the 8 voxels are smaller than a user-specified threshold ϕ_b. Let $V^{l-1}(x, y, z)$ be the voxel with the largest signed distance in $\mathcal{S}_{i,j,k}^l$, then the signed distance at voxel $V^l(i, j, k)$ after the merge

$$\phi^l(i, j, k) = \phi^{l-1}(x, y, z) - \sqrt{3} \cdot 2^{l-2}, \tag{4.37}$$

where $\sqrt{3} \cdot 2^{l-2}$ is the distance between the center of voxel $V^l(i, j, k)$ and the center of voxel $V^{l-1}(x, y, z)$. The gradient of the SDF $\nabla\phi^l(i, j, k)$ will be the same as voxel $V^{l-1}(x, y, z)$ (Algorithm 4.8). The merging process can be completely disabled by setting ϕ_b to $-\infty$.

Finally, particles are placed at the occupied voxels. The radii and actual positions of the particles are derived from the initial position of the mesh, the scale of the mesh and the sizes of the voxels.

Algorithm 4.4 Distance Transform Step 3: Voronoi Diagram in 2D

```
1:  for each column $(i,k)$ do
2:      $I_{i,k} \leftarrow 0$
3:      for $j = 0, 1, \ldots, n_2 - 1$ do
4:          while $I_{i,k} < T_{i,k}$ do                ▷ $T_{i,k}$ is the index of the top of stack $P_{i,k}$
5:              if $\|(i,j,k) - P_{i,k}[I_{i,k}]\| \leq \|(i,j,k) - P_{i,k}[I_{i,k}+1]\|$ then
6:                  $\mathbf{s}_{i,j,k} \leftarrow P_{i,k}[I_{i,k}]$
7:                  break
8:              else
9:                  $I_{i,k} \leftarrow I_{i,k} + 1$
10:             end if
11:         end while
12:         if $I_{i,k} = T_{i,k}$ then
13:             $\mathbf{s}_{i,j,k} \leftarrow P_{i,k}[I_{i,k}]$
14:         end if
15:     end for
16: end for
```

Algorithm 4.5 Distance Transform Step 4: Proximate Sites in 3D

```
1:  for each aisle $(i,j)$ do
2:      $T_{i,j} \leftarrow -1$                         ▷ index of the top of stack $P_{i,j}$
3:      for $k = 0, 1, \ldots, n_3 - 1$ do
4:          $\mathbf{c} \leftarrow \mathbf{s}_{i,j,k}$
5:          while $T_{i,j} > 0$ do
6:              $\mathbf{b} \leftarrow P_{i,j}[T_{i,j}]$
7:              $\mathbf{a} \leftarrow P_{i,j}[T_{i,j} - 1]$
8:              if a and c dominate b then
9:                  $T_{i,j} \leftarrow T_{i,j} - 1$
10:                 pop b out of $P_{i,j}$
11:             else
12:                 break
13:             end if
14:         end while
15:         push c to $P_{i,j}$
16:         $T_{i,j} \leftarrow T_{i,j} + 1$
17:     end for
18: end for
```

Algorithm 4.6 Distance Transform Step 5: Voronoi Diagram in 3D

```
 1: for each aisle $(i, j)$ do
 2:     $I_{i,j} \leftarrow 0$
 3:     for $k = 0, 1, \ldots, n_3 - 1$ do
 4:         while $I_{i,j} < T_{i,j}$ do                ▷ $T_{i,j}$ is the index of the top of stack $P_{i,j}$
 5:             if $\|(i, j, k) - P_{i,j}[I_{i,j}]\| \leq \|(i, j, k) - P_{i,j}[I_{i,j} + 1]\|$ then
 6:                 $\mathbf{s}_{i,j,k} \leftarrow P_{i,j}[I_{i,j}]$
 7:                 break
 8:             else
 9:                 $I_{i,j} \leftarrow I_{i,j} + 1$
10:             end if
11:         end while
12:         if $I_{i,j} = T_{i,j}$ then
13:             $\mathbf{s}_{i,j,k} \leftarrow P_{i,j}[I_{i,j}]$
14:         end if
15:     end for
16: end for
```

Algorithm 4.7 Signed Distance

```
1: for each voxel $p(i, j, k)$ do
2:     $d \leftarrow 0$
3:     if $p$ is not a site then        ▷ $|\phi|$ has been calculated for each site in Algorithm 4.1
4:         $d \leftarrow \|(i, j, k) - \mathbf{s}_{i,j,k}\|$                 ▷ $\mathbf{s}$ is the closest site to $p$
5:     end if
6:     $\phi_p \leftarrow (1 - 2|p|)\left(d + \left|\phi_{\mathbf{s}_{i,j,k}}\right|\right)$          ▷ $|p|$ is the value of the voxel bit
7: end for
```

Algorithm 4.8 Voxel Merging

```
 1: $l \leftarrow 1$
 2: while $\left|V^{l-1}\right| \geq 2^3$ do
 3:     for each voxel $V^l(i, j, k)$ do
 4:         if $\bigwedge\left(\phi^{l-1}(x, y, z) < \phi_b\right), \ \forall V^{l-1}(x, y, z) \in \mathcal{S}^l_{i,j,k}$ then
 5:             $\left|V^l(i, j, k)\right| \leftarrow 1$       ▷ $\left|V^l(i, j, k)\right|$ represents the bit of voxel $V^l(i, j, k)$
 6:             $\left|V^{l-1}(x, y, z)\right| \leftarrow 0, \quad \forall V^{l-1}(x, y, z) \in \mathcal{S}^l_{i,j,k}$
 7:             $V^{l-1}(x, y, z) \leftarrow \arg\max \phi^{l-1}(x, y, z), \quad \forall V^{l-1}(x, y, z) \in \mathcal{S}^l_{i,j,k}$
 8:             $\phi^l(i, j, k) \leftarrow \phi^{l-1}(x, y, z) - \sqrt{3} \cdot 2^{l-2}$
 9:             $\nabla\phi^l(i, j, k) \leftarrow \nabla\phi^{l-1}(x, y, z)$
10:         end if
11:     end for
12:     $l \leftarrow l + 1$
13: end while
```

5. Acceleration Structures

In our project, the purpose of using acceleration structures is twofold: 1) efficient collision detection and neighbor finding for each time step of the simulation; 2) searching for the skin particles (see below) for the mesh.

For a rigid body, the new vertex positions of the mesh can be determined by the translation and the rotation of the rigid body calculated during the shape matching. For a deformable object, however, the new vertex positions cannot be calculated this way to achieve the visual effect of deformation. Instead, the new position of each vertex is derived from a few particles (namely *skin particles*) closest to it during the initialization, which needs a k-nearest neighbors (k-NN) search for each vertex.

For a deformable body consisting of n particles, the mesh of which consisting of m vertices, the complexity of brute force search is $O(m \cdot n)$. Brute force search could be very slow for large rigid bodies and complicated meshes. Therefore, we use a uniform hash grid to accelerate the k-NN search.

As aforementioned, a uniform hash grid is not suitable for the collision detection when simulating scenes consisting of differently sized particles, therefore we use the BVH in our solver to accelerate the collision detection.

5.1 Uniform Hash Grid

5.1.1 Two Construction Methods

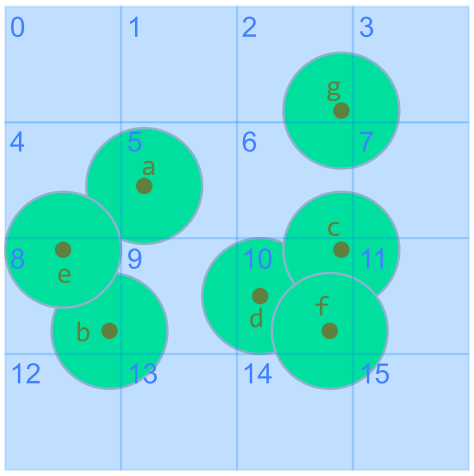
Green [2008] proposed two different methods for the construction of a hash grid: 1) a relatively simple method which requires the GPU to support atomic operations; 2) a more complex but efficient method using fast radix sort.

In both methods, the edge length of the cells is set to be the same as the diameter of the uniformly sized particles. During collision detection, for each particle i mapped into cell j, all the particles mapped into cell i and all the neighbor cells have to be tested. This corresponds to 9 cells in a 2D hash grid and 27 cells in a 3D hash grid.

The first step is the same for both methods, in which each thread i computes the index of the cell j into which particle i is mapped, then

(a) In the method which relies on atomic operations, each thread i atomically increases the counter of the cell j by one. With `atomicAdd`, multiple threads can update the counter of the same cell simultaneously without conflicts. The return value k of the function `atomicAdd` is exactly the index of the particle ID in the array preallocated to cell j which stores all the IDs of the particles mapped into cell j, and thread i will set the kth element of the array to the ID of particle i (table 1 in Fig. 5.1);

(b) In the method based on sorting, the pair (j, i) is set to be the ith element of a list of size N, where N is the number of the particles, resulting in an unsorted list. The list will be sorted by the cell indices. Finally, for each cell, its offset in the list will be recorded. For an empty cell, the offset is set to an impossible value such as ∞ (table 2 in Fig. 5.1). The particles mapped into cell j can be efficiently found by the offset of j.



hash grid

cell index	count	particle ID
0	0	
1	0	
2	1	g
3	0	
4	0	
5	1	a
6	0	
7	0	
8	2	b, e
9	0	
10	3	c, d, f
11	0	
12	0	
13	0	
14	0	
15	0	

table 1

unsorted	sorted	index	offset
(5 a)	(2 g)	0	
(8 b)	(5 a)	1	
(10 c)	(8 b)	2	0
(10 d)	(8 e)	3	
(8 e)	(10 c)	4	
(10 f)	(10 d)	5	1
(2 g)	(10 f)	6	
		7	
		8	2
		9	
		10	4
		11	
		12	
		13	
		14	
		15	

table 2

Fig. 5.1: Table 1 and Table 2 correspond to the data structures maintained by the method using atomic operations and the method using radix sort to construct the 2D hash grid on the left, respectively.

There are two problems with the method using atomic operations:

(1) The array for storing particle IDs has to be preallocated on the GPU, which should be large enough to avoid missing collisions when the particles are not uniformly distributed. Thus a huge amount of memory is required;

(2) The write operations will be serialized if multiple threads update the counter of the same cell simultaneously, which causes performance degradation.

On the other hand, the radix sort used in the sorting-based method can be efficiently implemented on the GPU [Le Grand, 2007; Satish et al., 2009]. The total memory requirement of the sorting-based method in number of 32-bit integer is $n + n + d_1 d_2 d_3 = 2n + d_1 d_2 d_3$ when using a hash grid of size $d_1 \times d_2 \times d_3$, namely n `int32_ts` for particle indices, n `int32_ts` for hash values of the particles, and $d_1 d_2 d_3$ `int32_ts` to record the offsets of the grid cells.

To sum up, compared to the sorting-based method, the atomic-based method is slower and more sensitive to the distribution of the particles, as noted by Green [2008].

5.1.2 k-Nearest Neighbors Search

For the special case of finding skin particles (Fig. 5.2), it is guaranteed there will be no more than one particle being mapped into the same cell if the edge length of the grid cells is set to be the same as the diameter of the boundary particles of the mesh (cf. Section 4.4.3), thus an atomic operation is unnecessary when using the atomic-based method.

After the mapping between particles and cells of the hash grid has been established, the k nearest particles for each vertex can be found iteratively by increasing the search radius r by one in each iteration, starting with $r = 0$ (i.e. the cell into which the vertex is mapped).

The weight `w` of particle i in the determination of the position of vertex v is calculated based on a monotonically decreasing function of the distance between particle i and vertex v, subject to the condition that `w` is positive and normalized such that the sum of the weights of the k particles neighboring to vertex v is one.

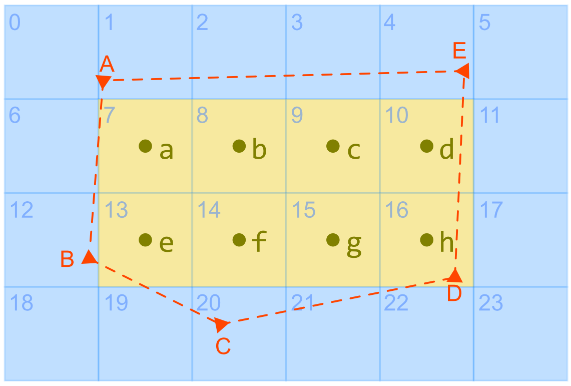


Fig. 5.2: After the voxelization, particles $a - h$ (only the centers are drawn) are generated for pentagon $ABCDE$. The nearest k particles for each vertex can be efficiently found through the use of a hash grid.

5.2 BVH Construction on the GPU

To achieve real-time collision detection, we implemented the algorithm presented by Karras [2012], in which the BVH is constructed from a hierarchical representation of the Morton codes of the particles derived from their positions.

5.2.1 Morton Code

Given the unit interval $\mathcal{I} = [0, 1]$ and the unit square $\mathcal{S} = [0, 1]^d$, a *space-filling curve* is a curve $f_{\mathcal{S}}(\mathcal{I})$ which establishes the mapping $f : \mathcal{I} \to \mathcal{S}$. One simple mapping for $d = 2$ is defined by

$$f(t) = f(0_2.b_1b_2b_3b_4b_5b_6\dots) = \begin{pmatrix} s_x \\ s_y \end{pmatrix} = \begin{pmatrix} 0_2.b_2b_4b_6\dots \\ 0_2.b_1b_3b_5\dots \end{pmatrix} \tag{5.1}$$

and the inverse mapping is defined by

$$g\begin{pmatrix} s_x \\ s_y \end{pmatrix} = g\begin{pmatrix} 0_2.b_2b_4b_6\dots \\ 0_2.b_1b_3b_5\dots \end{pmatrix} = 0_2.b_1b_2b_3b_4b_5b_6\dots, \tag{5.2}$$

where $0_2.b_1b_2b_3b_4b_5b_6\dots$ and $\binom{0_2.b_2b_4b_6\dots}{0_2.b_1b_3b_5\dots}$ are the binary representations of $t \in \mathcal{I}$ and $\binom{s_x}{s_y} \in \mathcal{S}$, respectively. This mapping is known as *Morton order*, and t is the *Morton code* of $\binom{s_x}{s_y}$. All image points that share the same sequence of binary digits lie in a common subsquare of side length 2^{-n}, where n is the length of the shared sequence. The sequentialisation of the subsquares generates *Z-curve* numbering pattern (Fig. 5.3), therefore Morton order is also known as *Z-order* [Bader, 2012].

For infinite sets, neither Z-order mapping nor the inverse mapping is bijective, as shown through an example given by Bader [2012]:

$$f\left(\frac{1}{2}\right) = f(0_2.1) = \begin{pmatrix} 0_2.0 \\ 0_2.1 \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} \tag{5.3}$$

and

$$f\left(\frac{1}{6}\right) = f(0_2.0010101\ldots) = \begin{pmatrix} 0_2.00000\ldots \\ 0_2.01111\ldots \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix}. \tag{5.4}$$

However, by restricting the spatial subdivision to a specific level, and that $0 \leq \|s_x\|_b - \|s_y\|_b \leq 1$, where $\|s_x\|_b$ and $\|s_y\|_b$ are the number of bits of the binary representations of s_x and s_y respectively, both mappings are bijective. In three dimensions, it is required that $0 \leq \|s_x\|_b - \|s_y\|_b \leq 1$ and $0 \leq \|s_y\|_b - \|s_z\|_b \leq 1$. For convenience, each cell of the subdivided space is usually defined as a unit square with edge length equal to 1.

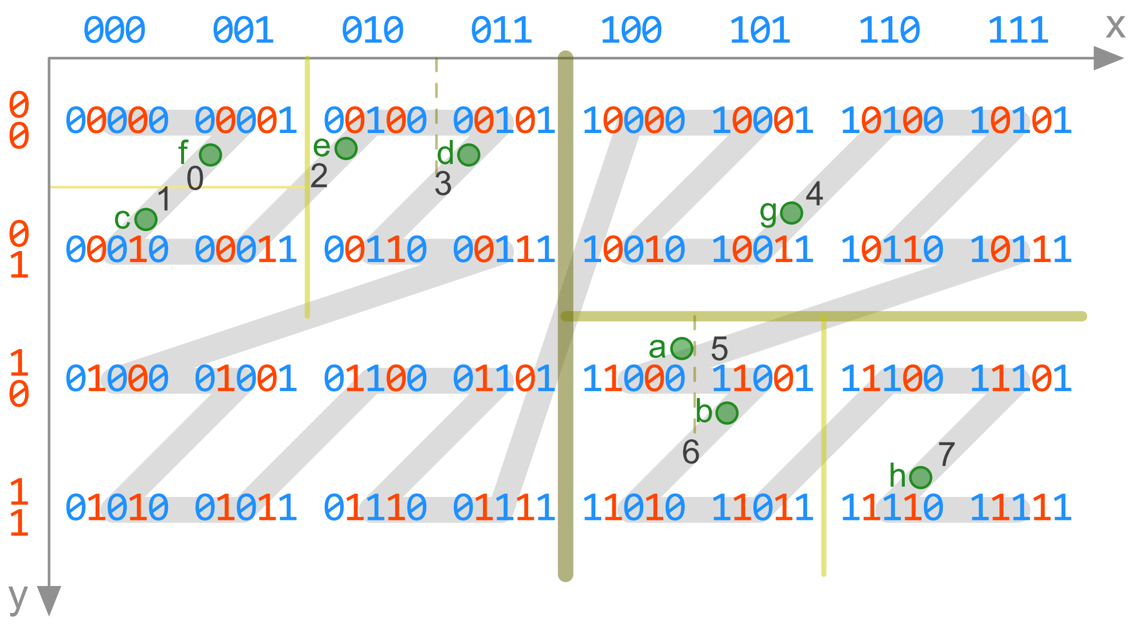


Fig. 5.3: The Morton codes in 2D, with 3 bits for x-coordinate and 2 bits for y-coordinate. There is an array of particles $a - h$ in the 2D space, and these particles are sorted based on their Morton codes. The black colored numbers are the indices of the particles in the sorted array.

For collision detection between particles, the coordinate of the cell into which particle i is mapped

$$\mathbf{x}_i^b = \begin{pmatrix} \mathbf{x}_{i,x}^b \\ \mathbf{x}_{i,y}^b \\ \mathbf{x}_{i,z}^b \end{pmatrix} = \left\lfloor \frac{\mathbf{x}_i - \mathbf{b}_{min}}{d} \right\rfloor, \tag{5.5}$$

where $\mathbf{x}_i$ is the coordinate of the particle center, $\mathbf{b}_{min}$ is the corner point of the scene space with the minimum coordinate and d is the diameter of the smallest particle. The

Morton code for particle i can be constructed by interleaving the successive bits of $\mathbf{x}_{i,x}^b$, $\mathbf{x}_{i,y}^b$ and $\mathbf{x}_{i,z}^b$, from the least significant bit (LSB) to the most significant bit (MSB), from x-coordinate to z-coordinate. Lauterbach et al. [2009] used the barycenter of each triangle primitive as the representative point for scenes containing triangle meshes.

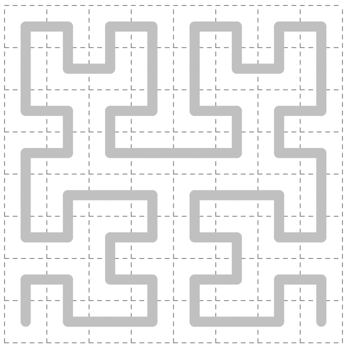


Fig. 5.4: The Hilbert curve at the third iteration, in which the space is divided into $2^3 \times 2^3$ subsquares.

In contrast to more complicated space-filling curves such as *Hilbert curve* (Fig. 5.4), which only connect neighboring grid cells, Z-curve is obviously not continuous. For example, there is a big jump between the cell with the Morton code `01111` and the cell with the Morton code `10000` in Fig. 5.3. However, the Morton code for each point can be calculated directly from its coordinate, while the constructions of other space-filling curves are more expensive [Lauterbach et al., 2009].

5.2.2 Binary Radix Trees

Given a set of n keys $k_0, k_1, \ldots, k_{n-1}$ represented as bit strings, a binary radix tree is a hierarchical representation of their common prefixes, in which the internal nodes represent the common prefixes of different length, with the keys being the leaf nodes. Each internal node of the binary radix tree contains exactly two children. Let N be the number of all the nodes in the tree, the number of the nodes except the root is $2(N-n)$, which should be equal to $(N-1)$, thus $N = 2n-1$, and the number of the internal nodes is $(n-1)$.

If the keys are Morton codes, the internal nodes correspond to the subsquares of different sizes in the $D-$dimensional space filled by the Morton curve. Given an array of n points whose Morton codes are the keys of a binary radix tree, each pair of two consecutive points in the sorted array based on their Morton codes is split by a $(D-1)$-dimensional hyperplane. The subsquare of some internal node is also split by the hyperplane into two subregions, which correspond to the two children of that internal node (cf. Fig. 5.3 and Fig. 5.5). Since there are $(n-1)$ hyperplanes, the number of internal nodes is also $(n-1)$.

Assuming that the keys are already sorted, the keys covered by each node can be represented as a linear range $[i, j]$. Following the notation proposed by Karras [2012], let $\delta(i, j)$ be the length of the longest common prefix between keys k_i and k_j, then

$$\delta(i', j') \geq \delta(i, j), \quad \forall i', j' \in [i, j], \tag{5.6}$$

thus the prefix corresponding to a given node can be exclusively determined by comparing its first key and last key. Let $\gamma \in [i, j-1]$ be the index of the last key whose $(\delta(i, j) + 1)$-th bit is 0, then the node will be partitioned into two children represented by the subranges $[i, \gamma]$ and $[\gamma+1, j]$, respectively. Further partitions for the resulting children

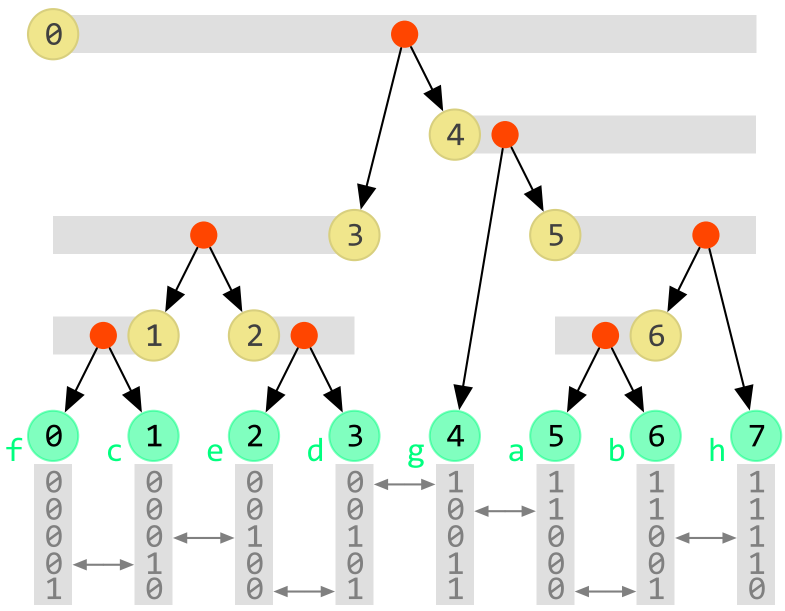


Fig. 5.5: An example of the binary radix tree given by Karras [2012]. The keys (i.e. the leaves) correspond to the Morton codes of the particles in Fig. 5.3.

can be proceeded similarly. γ is termed *split position* by Karras [2012]. Each split position corresponds to a hyperplane in the space filled by the Morton curve.

For example, in Fig. 5.5, the root corresponds to the range $[0, 7]$ with $\delta(0, 7) = 0$. Since key k_3 is the last key whose first bit is 0 and the next key k_4 starts with 1, $\gamma = 3$ for the root node, which results in the subregion $[0, 3]$ for internal node 3 and $[4, 7]$ for internal node 4. Because $\delta(0, 3) = 2$ and the last key with the first 3 bits being 0 is k_1, $\gamma = 1$ for the internal node 3, leading to the subregions $[0, 1]$ and $[2, 3]$, which correspond to internal nodes 1 and 2, respectively. These split positions correspond to the yellow colored split lines in Fig. 5.3.

The height h of the resulting tree is bounded by the length l of the Morton code, i.e. $h \leq l + 1$. In the extreme, if each Morton code mapped to a coordinate in the space filled by the Morton curve is used as a key for the construction of the tree, l times of splitting are needed, resulting in a tree with height $l + 1$.

5.2.3 Construction of Binary Radix Trees

A binary radix tree can be recursively constructed by finding the split position and creating the child nodes for each node, starting from the root. This approach is inherently sequential as the coverage of one node depends on its ancestors. By assigning indices to the internal nodes in a way that their children can be found without earlier results, Karras [2012] proposed a highly parallel algorithm for constructing binary radix trees.

They stored the leaf nodes and the internal nodes in two separate arrays L and I, such that the root is located at I_0, and the children of each internal node which covers the interval $[i, j]$ are placed according to the split position γ of that node. The left child is located at I_γ with the coverage $[i, \gamma]$ if it covers more than one key, namely $\min(i, j) < \gamma$, otherwise it is a leaf located at L_γ. Similarly, the right child is located at $I_{\gamma+1}$ with the coverage $[\gamma + 1, j]$ if $\max(i, j) > \gamma + 1$, otherwise it is a leaf located at $L_{\gamma+1}$ (Fig. 5.5). Evidently, the index of every internal node coincides with either its first or last key.

The other end of the range for each internal node can be efficiently found by looking at the nearby keys. Specifically, for an internal node I_i, the first step is to determine the "direction" d of its range by looking at the neighboring keys k_{i-1}, k_i, and k_{i+1}, where $d = +1$ for a range beginning at i and $d = -1$ for a range ending at i. Due to the fact that each internal node covers at least two keys, k_i and k_{i+d} are definitely covered by I_i. On the other hand, k_{i-d} has to be covered by the sibling node I_{i-d}, as siblings are always located next to each other in array I.

Algorithm 5.1 Construction of a Binary Radix Tree [Karras, 2012]

```
 1: for each internal node with index $i \in [0, n-2]$ in parallel do
 2:     $d \leftarrow \text{sign}(\delta(i, i+1), \delta(i, i-1))$
 3:     $\delta_{min} \leftarrow \delta(i, i-d)$
 4:     $l_{max} \leftarrow 2$
 5:     while $\delta(i, i + l_{max} \cdot d) > \delta_{min}$ do
 6:         $l_{max} \leftarrow l_{max} \cdot 2$
 7:     end while
 8:
 9:     $l \leftarrow 0$
10:     for $t = l_{max}/2, l_{max}/4, \ldots, 1$ do          ▷ binary search for $j$
11:         if $\delta(i, i + (l+t) \cdot d) > \delta_{min}$ then
12:             $l \leftarrow l + t$
13:         end if
14:     end for
15:     $j \leftarrow i + l \cdot d$
16:
17:     $s \leftarrow 0$
18:     for $t = \lceil l/2 \rceil, \lceil l/4 \rceil, \ldots, 1$ do          ▷ binary search for $\gamma$
19:         if $\delta(i, i + (s+t) \cdot d) > \delta(i, j)$ then
20:             $s \leftarrow s + t$
21:         end if
22:     end for
23:     $\gamma \leftarrow i + s \cdot d + \min(d, 0)$
24:
25:     if $\min(i, j) = \gamma$ then
26:         left $\leftarrow L_\gamma$
27:     else
28:         left $\leftarrow I_\gamma$
29:     end if
30:     if $\max(i, j) = \gamma + 1$ then
31:         right $\leftarrow L_{\gamma+1}$
32:     else
33:         right $\leftarrow I_{\gamma+1}$
34:     end if
35:     $I_i \leftarrow (\text{left}, \text{right})$
36: end for
```

By definition, the keys covered by I_i share a common prefix different from the one shared by the keys covered by the sibling node of I_i, which gives the lower bound $\delta_{min} = \delta(i, i-d)$ for the length of the prefix. For each key k_j covered by node I_i, $\delta(i, j) > \delta_{min}$. Therefore the direction can be determined by

$$d = \text{sign}(\delta(i, i+1), \delta(i, i-1)). \tag{5.7}$$

Due to the same reasoning, the other end of the range can be found by searching for the largest l such that $\delta(i, i + ld) > \delta_{min}$. Let q be the minimum positive integer such that

$$\delta(i, i + 2^q d) = \delta(i, i + l_{max} d) < \delta_{min} = \delta(i, i - d), \tag{5.8}$$

then l can be found using binary search (Line 9 – 14 of Algorithm 5.1) in the range $[0, l_{max} - 1]$, and the other end $j = i + ld$.

The split position γ can also be determined by performing a similar binary search (Line 17 – 22 of Algorithm 5.1) for the largest $s \in [0, l - 1]$ which satisfies $\delta(i, i + sd) > \delta(i, j)$. If $d = +1$, $\gamma = i + sd$ corresponding to the largest index of the keys covered by the left child. If $d = -1$, $\gamma = i + sd - 1$ to account for the inverted indexing.

Finally, given i, j and γ, the locations of the children of I_i can be easily found, including the indices and the pointers to the corresponding arrays. For the BVH construction, the parent pointers for the two children should also be recorded at this step.

Providing that the length of the keys is fixed, $\delta(i, j)$ can be efficiently calculated by counting the leading zero bits of the logical XOR between the two keys, which can be realized by the integer intrinsic functions in CUDA: `_clz()` for 32-bit keys and `_clzll()` for 64-bit keys.

5.2.4 BVH Construction

Karras [2012] presented methods for constructing BVHs, octrees and k-d trees from binary radix trees. Given a set of particles, the BVH can be constructed in four steps:

(1) Calculate the Morton codes based on the positions of the particle centers;
(2) Sort the indices of the particles based on their Morton codes;
(3) Construct a binary radix tree;
(4) Calculate the bounding volumes for all the internal nodes.

The construction of binary radix trees relies on the keys being unique. However, it is possible that duplicated Morton codes are generated in step 1 when two particles are so close to each other that the quantized coordinates for both of them are identical. Karras [2012] solved this problem by using the indices i and j as fallback of the two identical keys k_i and k_j when evaluating $\delta(i, j)$.

In step 4, each thread starts from one leaf node and walks up the tree using parent pointers recorded during the construction of the binary radix tree, and atomically increases the counter of the node being visited using `atomicAdd`. The thread terminates immediately if the return value of `atomicAdd` is 0. Otherwise, the thread will calculate the bounding volume for that node. Due to the bottom-up approach, the bounding volumes of the children of node i must haven been computed before node i gets visited.

An intuitive and simple choice for the bounding volumes of particles is a sphere. However, we found that for scenes containing many large rigid bodies, AABBs result in better performance.

5.3 BVH Traversal on the GPU

5.3.1 Traversal Using Stacks

The most popular method to traverse a tree is to use a stack. Because shared memory on a CUDA device is generally not large enough for a typical scene consisting of tens of thousands particles, we implemented the stack on global memory. To reduce the global memory access, which is generally expensive for GPU devices, we copy the indices of the nodes into global memory only when necessary (Algorithm 5.2).

Algorithm 5.2 BVH Traversal Using Stack

```
1: $N \leftarrow i_{\text{root}}$    ▷ set the top of the stack $\mathcal{S}$ to the index of root, cf. Section 5.2.2
2: $T \leftarrow 0$    ▷ $T$ is the index of the top of the stack
3: while $T \geq 0$ do
4:     $T \leftarrow T - 1$    ▷ pop the top element of the stack
5:     if $N$ is a leaf node then
6:         process leaf
7:     else
8:         $C_L \leftarrow$ index of the left child of $N$
9:         $C_R \leftarrow$ index of the right child of $N$
10:        test $C_L$ and $C_R$
11:        if both accepted then
12:            find the near child $C_{\text{near}}$ and the far child $C_{\text{far}}$,    ▷ $C_{\text{near}}, C_{\text{far}} \in \{C_L, C_R\}$
13:            $T \leftarrow T + 1$
14:            $\mathcal{S}[T] \leftarrow C_{\text{far}}$    ▷ copy the index of the far child to global memory
15:            $T \leftarrow T + 1$
16:            $N \leftarrow C_{\text{near}}$    ▷ set the top of the stack $\mathcal{S}$ to the near child
17:        else if only $C_i$ accepted then    ▷ $C_i \in \{C_L, C_R\}$
18:            $T \leftarrow T + 1$
19:            $N \leftarrow C_i$
20:        else if $T \geq 0$ then
21:            $N \leftarrow \mathcal{S}[T]$
22:        end if
23:    end if
24: end while
```

5.3.2 Stack-less Traversal

The parallel implementation of the stack-based traversal needs to maintain a full stack for each particle, thus the storage and bandwidth costs could be very high. Hoping that traversal without stack could possibly increase the performance, we also implemented the sparse stack-less traversal algorithm 5.3 proposed by Barringer and Akenine-Möller [2013] and a similar but more efficient algorithm 5.4, namely MBVH2 proposed by Áfra and Szirmay-Kalos [2014].

Both algorithms use an integer variable `bitstack` to keep track of the traversal level. The variable `bitstack` should be at least h bits long, where h is the height of the tree. In our project, we use a 64-bit integer for `bitstack`. However, `bitstack` can be efficiently implemented for higher trees. To ensure that the height of the BVH will not exceed 64 and that the space covered by the Morton codes is sufficient for typical scenes, we used 15 bits for each component of the quantized coordinates, thus a 64-bit integer is large enough for each Morton code. The maximum height of the BVH will be 46 when there are no duplicated keys, and will not exceed 64 as long as the total number of the particles is less than $2^{32} = 4\,294\,967\,296$, because the indices of the particles are used when they have the same Morton code, which still results in a binary radix tree. In fact, given n particles, and let m be the number of their unique Morton codes, the maximum height of the binary radix tree can be achieved when $(n - m + 1)$ particles have the same Morton code, and the resulting height is $h = 1 + \log_2 m + \log_2(n - m + 1) \leq -1 + 2\log_2(n + 1)$. Therefore, to keep $h \leq 64$, the maximum number of particles is $n = \lfloor\sqrt{2^{65}} - 1\rfloor$.

Both stack-less traversal algorithms heavily rely on the siblings of the nodes. Therefore we also construct an array of sibling pointers during the construction of the BVH. The extra expense of building such an array is very low, since the sibling of node i is going to be accessed anyway to compute the bounding volume of the parent of node i, and the array of the Morton codes ($n \times$ 64-bit) can be reused for storing the sibling indices ($(2n-1) \times$ 32-bit) as the Morton codes are no longer in use after the construction of the binary radix tree (cf. Section 5.2.4), where n is the number of particles. Table 5.1 shows the memory requirement for stack-less BVH traversal and construction using bounding spheres and using AABBs.

	Indices	**Morton codes**	**parents**	**children**	**Bounding Volumes**	**Total**
Bounding spheres	n	$2n$	$2n-1$	$2(n-1)$	$4(n-1)$	$11n-7$
AABBs	n	$2n$	$2n-1$	$2(n-1)$	$6(n-1)$	$13n-9$

Table 5.1: Memory requirement for stack-less traversal in number of 32-bit Integers.

MBVH2 terminates the traversal immediately when `bitstack` becomes 0 (Line 25 of algorithm 5.4), whereas the sparse traversal exits the loop only when it returns to the root node (line 28 of Algorithm 5.3), and the bitwise operation is typically more efficient than arithmetic operation (such as the increment operator in Line 17 of algorithm 5.3). Therefore, MBVH2 is slightly more efficient than the sparse traversal algorithm.

Algorithm 5.3 Sparse Stack-less Traversal [Barringer and Akenine-Möller, 2013]

```
 1: bitstack ← 0
 2: $N \leftarrow i_{\text{root}}$
 3: **repeat**
 4:     **if** $N$ is a leaf node **then**
 5:         process leaf
 6:     **else**
 7:         $C_L \leftarrow$ index of the left child of $N$
 8:         $C_R \leftarrow$ index of the right child of $N$
 9:         test $C_L$ and $C_R$
10:         **if** any accepted **then**
11:             bitstack ← bitstack $\ll 1$
12:             **if** both accepted **then**
13:                 find the near child $C_{\text{near}}$ and the far child $C_{\text{far}}$   ▷ $C_{\text{near}}, C_{\text{far}} \in \{C_L, C_R\}$
14:                 $N \leftarrow C_{\text{near}}$
15:             **else if** only $C_i$ accepted **then**   ▷ $C_i \in \{C_L, C_R\}$
16:                 $N \leftarrow C_i$
17:                 bitstack ← bitstack + 1
18:             **end if**
19:             **continue**
20:         **end if**
21:     **end if**
22:     bitstack ← bitstack + 1
23:     **while** bitstack $\wedge 1 = 0$ **do**   ▷ $\wedge$: bitwise AND
24:         $N \leftarrow parent(N)$
25:         bitstack ← bitstack $\gg 1$
26:     **end while**
27:     $N \leftarrow sibling(N)$
28: **until** $N = i_{\text{root}}$
```

Algorithm 5.4 MBVH2 Stack-less Traversal [Áfra and Szirmay-Kalos, 2014]

```
 1: bitstack ← 0
 2: $N \leftarrow i_{\text{root}}$
 3: loop
 4:     if $N$ is a leaf node then
 5:         process leaf
 6:     else
 7:         $C_L \leftarrow$ index of the left child of $N$
 8:         $C_R \leftarrow$ index of the right child of $N$
 9:         test $C_L$ and $C_R$
10:         if any accepted then
11:             bitstack ← bitstack $\ll$ 1
12:             if both accepted then
13:                 find the near child $C_{\text{near}}$ and the far child $C_{\text{far}}$   ▷ $C_{\text{near}}, C_{\text{far}} \in \{C_L, C_R\}$
14:                 $N \leftarrow C_{\text{near}}$
15:                 bitstack ← bitstack $\vee$ 1                                  ▷ $\vee$: bitwise OR
16:             else if only $C_i$ accepted then                                ▷ $C_i \in \{C_L, C_R\}$
17:                 $N \leftarrow C_i$
18:             end if
19:             continue
20:         end if
21:     end if
22:
23:     while bitstack $\wedge$ 1 = 0 do                                          ▷ $\wedge$: bitwise AND
24:         if bitstack = 0 then
25:             return
26:         end if
27:         $N \leftarrow parent(N)$
28:         bitstack ← bitstack $\gg$ 1
29:     end while
30:     $N \leftarrow sibling(N)$
31:     bitstack ← bitstack $\oplus$ 1                                            ▷ $\oplus$: bitwise XOR
32: end loop
```

5.3.3 Further Optimization

We note that the traversal algorithms 5.2 – 5.4 actually do some unnecessary test by starting from the root node of the BVH, because the bounding volumes of all the ancestors of particle p unconditionally collide with particle p. To avoid such extra work, we start the traversal from the leaf node, i.e. the particles.

Algorithm 5.5 Bottom-Up Traversal

```
1: for each particle $p$ do
2:     $R \leftarrow sibling(p)$
3:     while $R \neq$ root do
4:         traverse the subtree rooted at $R$ with one of the algorithms 5.2 – 5.4
5:         $P \leftarrow parent(R)$
6:         $R \leftarrow sibling(P)$                                 ▷ $sibling(\text{root}) = \text{root}$
7:     end while
8: end for
```

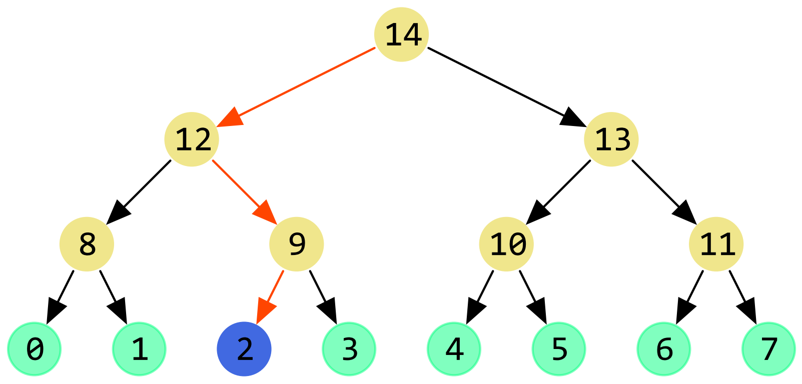


Fig. 5.6: A binary tree

For example, in Fig. 5.6, both of the internal nodes 12 and 9 will be tested for particle 2 when using the top-down algorithms 5.2 – 5.4. However, these tests will be directly skipped by the bottom-up traversal algorithm 5.5. Precisely, only particle 3, the subtrees rooted at internal nodes 8 and 13 will be tested, thereby saving $O(\log n)$ tests for each particle, where n is the number of the particles.

Noticing that the solver will not generate contact constraints between the particles of the same rigid body, we optimize the collision detection for rigid body particles by comparing the phases (cf. the structure of particle attributes at the beginning of Chapter 3) of each internal node I_i and the particle L_j being inspected. The collision test will be immediately skipped if particle L_j belongs to some rigid body and I_i has the same phase as L_j. The phases of internal nodes can be efficiently determined during the BVH construction (cf. Section 5.2.4). Specifically, if the two children of an internal node I_i have the same phase, the phase of node I_i is set to be the same as its children. Otherwise, the phase of I_i is set to an impossible value. In this way, all the particles covered by the internal node I_i and particle L_j belong to the same rigid body if the phase of particle L_j and the phase of node I_i are the same. We did not allocate extra memory for the phases of internal nodes. Instead, we used the array of the counters for constraint averaging to store the phases of internal nodes temporarily (Eq. 3.25 in Section 3.1.4).

6. Implementation and Results

We implemented the solver using CUDA 8.0 on GeForce GTX 1060 (Laptop), installed in a notebook with 2.6 GHz Intel Core i7-6700HQ CPU running Windows 10. All the solver data are stored in one dimensional arrays on the GPU allocated in linear memory using `cudaMalloc` except for the structure of the solver parameters, which is declared in constant memory space. For voxelization and SDF construction on the GPU, we also used one dimensional arrays in global memory.

For linear transformations of the data, filling arrays of non-char types and sorting, we used the data parallel primitives provided by the thrust library, which is a C++ template library for CUDA based on the Standard Template Library and included in the CUDA toolkit [NVIDIA, 2017b].

6.1 Rigid Bodies

For the SDF construction, We merged step 2 and step 3 into one kernel, and implemented step 4 and step 5 using the same kernel code with different parameters, namely changing the processing direction from "column" to "aisle" (Section 4.4.2).

For comparison, we also implemented the fast marching method (FMM, cf. Section 2.6) for the SDF construction on the CPU. Voxelization on the CPU is implemented with ray marching, in which the intersection of rays against triangles is accelerated by the BVH implemented on the CPU.

Model	#triangles	Grid size	#voxels	GPU	CPU
Box	12	32^3	32 768	1.51	0.71
		96^3	884 736	10.20	8.73
Stanford bunny	5 002	32^3	6 620	0.56	7.24
		96^3	177 695	0.53	25.79
Armadillo	30 000	32^3	2 250	0.58	43.56
		96^3	60 714	0.55	62.13
Stanford dragon	47 794	32^3	1 854	0.64	64.33
		96^3	49 921	0.63	90.14

Table 6.1: Running time (in *ms*) for voxelization on the GPU and CPU.

The results of the solid voxelization are shown in Table 6.1. For simple meshes consisting of only a few triangles, CPU implementation is more efficient. However, as the number of triangles increases, the BVH construction and traversal become more expensive, thus the time expense increases rapidly. In contrast, the GPU implementation can fully utilize the GPU computing power when the number of triangles gets large. Larger grid sizes result in slightly better performance, because there are less voxels shared by multiple threads, which can mitigate the serialization due to atomic operations.

Model	**Grid size**	**#voxels**	**CPU**	**GPU**	**Merge**	**#voxels** (merged)
Box	32^3	32 768	48.01	2.45	1.67	10 963 (-66.5%)
	96^3	884 736	1686.41	19.71	4.09	107 967 (-87.8%)
Stanford bunny	32^3	6 620	47.32	2.48	1.35	4 023 (-39.2%)
	96^3	177 695	2057.06	16.94	3.35	46 606 (-73.8%)
Armadillo	32^3	2 250	48.12	2.53	1.58	1 977 (-12.1%)
	96^3	60 714	2063.54	15.98	3.28	29 025 (-52.2%)
Stanford dragon	32^3	1 854	49.77	2.20	1.23	1 805 (-2.6%)
	96^3	49 921	2044.61	15.13	3.96	29 201 (-41.5%)

Table 6.2: Running time (in *ms*) for SDF construction on the CPU, SDF construction and voxel merging on the GPU. The last column shows the number of voxels after merging and the percentage reduction.

The results of the SDF construction are shown in Table 6.2. It is evident that the GPU implementation is far more efficient than the CPU implementation. The number of voxels can be greatly reduced by merging the voxels in most cases. Voxel merging does not work well on the dragon model with grid size 32^3 due to the fact that the model is very thin, thereby resulting in many voxels near the boundary of the mesh, which should not be merged.

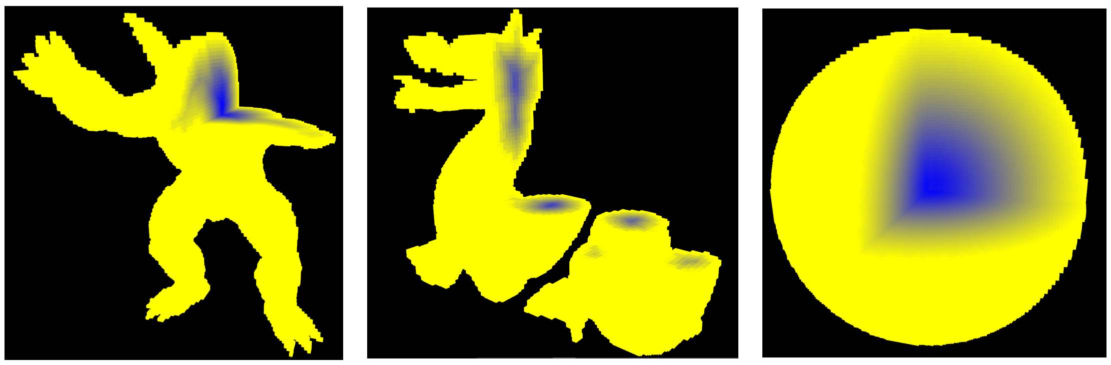
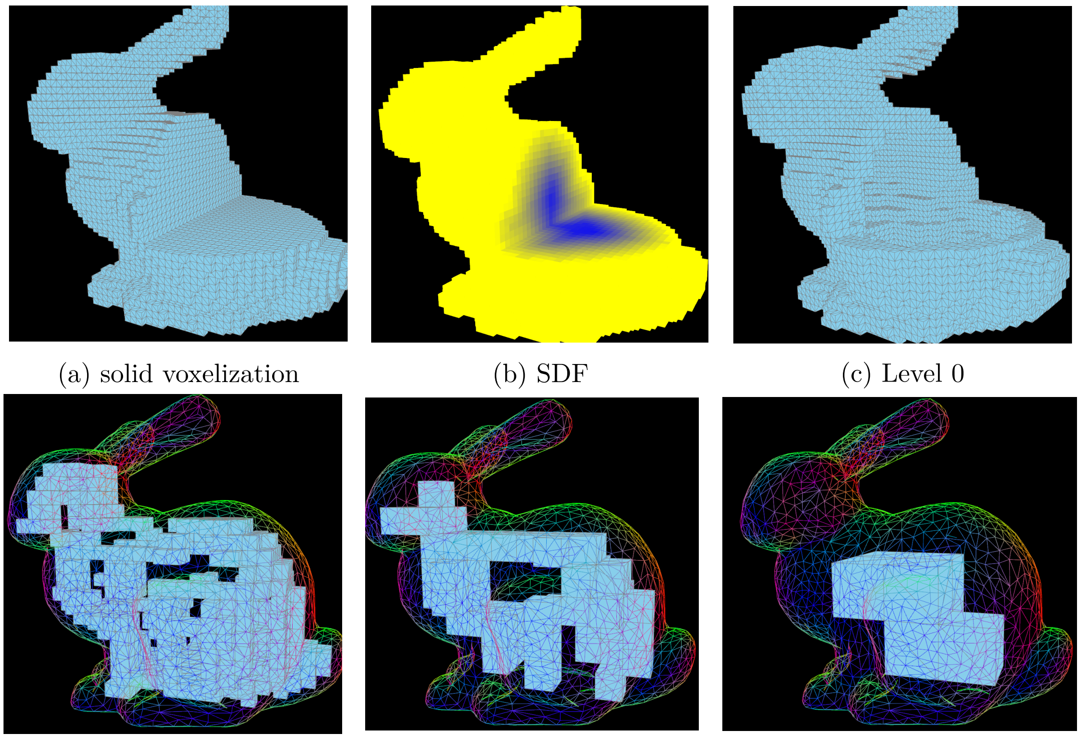


Fig. 6.1: SDF of three models. Some voxels are removed for better visualization. The magnitude of the signed distance of blue-colored voxels is larger than yellow-colored ones.

During shape matching (Section 4.1), to calculate the translations and the sum of particle weights for rigid bodies, the reduction primitives of the thrust library can be directly used in the kernel, and each thread processes one rigid body. However, this naïve implementation may lead to underutilization of the parallel cores.

To increase parallelism, we used the primitive `DeviceSegmentedReduce` of the CUB library, which can take advantage of the dynamic parallelism supported by CUDA devices of compute capability 3.5 and higher to achieve better performance. With dynamic parallelism, a CUDA kernel is able to create and synchronize with new work directly on the GPU, which reduces the need to transfer execution control and data between host and device [NVIDIA, 2017a].



(a) solid voxelization (b) SDF (c) Level 0

(d) Level 1 (e) Level 2 (f) Level 3

Fig. 6.2: Voxelization and SDF of the Stanford bunny. The image on the top left is the voxelization before merging, and the image next to it is the corresponding SDF, with part of the voxels removed for better visualization. After merging, four levels 0 − 3 are generated.

For comparison we measured two scenes consisting of 216K particles. In the first scene, there are 10^3 boxes and each of them consisting of 6^3 particles; In the second scene, there are 5^3 boxes and each of them consisting of 12^3 particles. Table 6.3 shows the results. Although the two implementations are almost equally efficient for the first scene, the naïve implementation becomes significantly slower when the rigid bodies get larger. In contrast, with CUB the performance is even slightly better for the second scene because fewer rigid bodies need fewer reductions.

	Scene 1 ($10^3 \times 6^3$)	**Scene 2** ($5^3 \times 12^3$)
Naïve implementation	1.07	3.14
Implementation with CUB	1.02	0.99

Table 6.3: Running time (in *ms*) for shape matching on the GPU. Scene 1 consists of 1000 boxes, each of which consists of 216 particles; Scene 2 consists of 125 boxes, each of which consists of 1728 particles.

6.2 Collision Detection

Since collision test is the most performance-critical part, for which we replaced the hash grid with the BVH, it is very important to *fairly* compare these methods. To achieve a fair comparison for scenes of rigid bodies, we tested our solver for the scene consisting of 5^3 boxes, each of which consisting of 6^3 particles (Fig. 6.3). The box shape is chosen because it is relatively stable, thus more suitable for testing the performance of these methods compared to other shapes such as Stanford bunnies, which can easily be separated apart from each other by the collision constraints. The boxes are voxelized into 12^3 particles, which seems to be unnecessarily high. However, a coarser voxelization is not apposite for other shapes due to under-sampling on the boundary. To test scenes consisting of granular-like particles, we simply replaced the shape matching constraints for the rigid bodies with collision constraints. Therefore, both scenes consist of 216K particles. Large amounts of particles also improve the accuracy of the running time measurement, and ensure the scalability of these methods. We also applied voxel merging to the rigid bodies, which leads to another test scene for rigid bodies consisting of 153K particles. For comparison, we implemented the hash grid of size 64^3 and of size 128^3.

The array of neighbor indices (Line 7 of Algorithm 3.3 in Section 3.1.6) for each particle should be preallocated on the GPU. For typical scenes, an array of length 32 for each particle is usually large enough to achieve plausible simulations. Larger arrays would result in unnecessary waste of memory, whereas smaller arrays might lead to visual artifacts. Therefore we used an array of length $32 \cdot n$ for these test scenes, where n is the number of particles.

The radius of the boundary particles is 75 mm, thus the edge length of a box is 1.8 m. The box piles are located at $h = 3.6$ m with zero velocity at the beginning of the simulations, and fall down towards the ground under gravity. The number of time steps is set to 2, which means the collision test will be executed twice for each frame. We averaged the measured running time on every 100 frames, and only recorded the measurement for the first 800 frames, because all the scenes become almost stable after 800 frames.

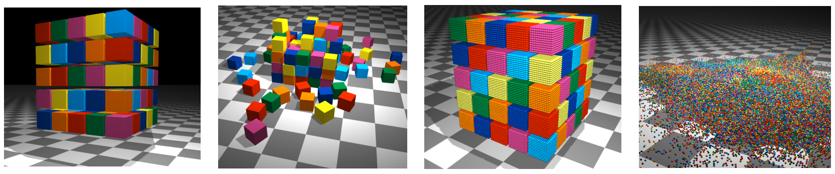


Fig. 6.3: The two images on the left are the initial state and final state of the test scene for rigid bodies respectively, and the two images on the right are the initial state and final state of the test scene for granular-like particles respectively.

BVH construction consists of three parts: sorting, binary radix tree construction and calculation of bounding volumes (bounding spheres or AABBs). As shown in Table 6.4, the BVH construction is relatively fast. The bottleneck of the simulation largely lies on the collision detection, as shown in Table 6.5 – Table 6.7.

	Sorting	**Binary Radix Tree**	**Bounding Spheres**	**AABBs**	**Total**
153K	1.62	0.18	0.28	0.23	2.08 (2.03)
216K	2.03	0.22	0.36	0.31	2.61 (2.56)

Table 6.4: Average running time (in ms) for BVH construction in two scenes consisting of 153K particles and 216K particles, respectively. In the last column, the number in parenthesis is the running time for BVH construction with AABBs.

From the beginning of the simulation to the 400th frame, the performance of both hash grids is better, because for each collision, the complexity is $O(\log n)$ using BVH, where n is the total number of particles, and the cost of using hash grid is constant. After many boxes reach the ground, the performance of the hash grid of size 64^3 becomes the worst, because most of the particles are mapped into the lower part of the hash grid. Meanwhile, the BVH becomes more efficient, due to the fact that there is less overlap between bounding volumes (Table 6.5 and Table 6.6). This degradation problem with hash grids is more severe for scenes containing granular-like particles (Table 6.7).

	BVH	MBVH2 - 153K	Bot.Up - 153K	Stack Sp. - 153K	AABB - 153K	Grid	64^3 Hash	128^3 Hash
1	1.79	5.99	3.24	2.25	1.24	1.70	0.74	0.70
2	2.12	8.42	5.94	5.03	1.91	1.68	0.82	0.79
3	2.12	8.36	6.33	5.92	2.44	1.67	1.19	1.01
4	2.16	8.05	5.79	5.42	2.26	1.80	2.05	1.28
5	2.11	7.81	5.55	5.00	2.17	1.86	2.82	1.30
6	2.11	7.54	5.29	4.96	2.16	1.76	3.03	1.29
7	2.10	7.44	5.23	4.96	2.12	1.77	3.06	1.28
8	2.12	7.45	5.27	4.94	2.12	1.75	3.04	1.20

Table 6.5: Running time (in *ms*) for different traversal methods. Column **BVH** and Column **Grid** give the construction time of BVH and hash grid, respectively. The BVH is tested for the scene of rigid boxes with differently sized particles generated using voxel merging. The hash grids are tested for the scene of rigid boxes with uniformly sized particles, since it does not support particles of different sizes. Column **MBVH2-153K** is the running time for MBVH2 (Algorithm 5.4); Column **Bot.Up-153K** is the running time for the bottom-up algorithm 5.5 in combination with MBVH2; Column **Stack Sp.-153K** is the running time for the bottom-up algorithm 5.5 in combination with stack-based traversal. So far these methods use exclusively bounding spheres for the BVH. Column **AABB-153K** is the running time for the bottom-up algorithm 5.5 in combination with stack-based traversal, but using AABBs for the BVH. All the BVH traversal methods are optimized using the phase identifiers of internal nodes except the MBVH2 top-down traversal (Section 5.3.3).

We also note that Column **AABB-153K** (Table 6.5) is only marginally better than Column **AABB-216K** (Table 6.6), compared to the 63K particles reduced by voxel merging. The degradation is possibly caused by warp divergence in the scene consisting of differently sized particles where two threads inside the same warp are more likely to take different execution flows than in a scene consisting of uniformly sized particles.

As for memory requirement in number of `int32_ts`, compared to a hash grid, the stack-less traversal with BVH using AABBs is more memory-efficient when $13n - 9 < 2n + d^3$, i.e. when the number of the particles $n < (d^3+9)/11$, where d^3 is the hash grid size and n is the number of particles (cf. Section 5.1.1 and Section 5.3.2). For $d = 64$, stack-less traversal is more memory-efficient when $n \leq 23\,832$ and for $d = 128$, $n \leq 190\,651$. However, in the test scenes for rigid bodies, 63K ($0.29 \cdot n$) particles are reduced by voxel merging, meaning that $0.29 \cdot 32n = 9.30n$ decrease in the length of the array of neighbor indices. Taking the attributes of the rigid body particles into consideration, including positions ($3n$), predicted positions ($3n$), velocities ($3n$), masses (n), phases (n), radii (n), SDF ($4n$), initial positions ($3n$), position corrections ($3n$), counters for constraint averaging (n), weights for shape matching (n), initial ($3n$) and current ($3n$) relative positions (Chapter 3 and Section 4.1), the total decrease in memory requirement is $0.29 \cdot 30n + 9.30n = 18n$. The total change in memory requirement due to stack-less AABB traversal and voxel merging is $0.71 \cdot (13n - 9) - 18n = -8.77n - 6.39$, whereas the total change due to a hash grid of size d^3 is $2n + d^3$. Therefore, voxel merging followed by BVH stack-less traversal for rigid bodies is more memory-efficient.

6.3 Simulations

We use the bottom-up algorithm 5.5 in combination with the stack-based traversal for collision detection accelerated by the BVH using AABBs as bounding volumes, since it is

	MBVH2 - 216K	**Bot.Up - 216K**	**Stack Sp. - 216K**	**AABB - 216K**	**64^3 Hash**	**128^3 Hash**
1	9.96	5.15	5.99	1.95	0.74	0.70
2	12.55	7.04	7.62	2.72	0.82	0.79
3	12.67	7.67	7.31	2.49	1.19	1.01
4	12.41	7.12	6.87	2.37	2.05	1.28
5	11.94	6.80	6.73	2.35	2.82	1.30
6	11.60	6.62	6.81	2.33	3.03	1.29
7	11.53	6.60	6.78	2.29	3.06	1.28
8	11.50	6.69	6.79	2.29	3.04	1.20

Table 6.6: Running time (in *ms*) for different traversal methods. The only difference from Table 6.5 is that the BVH is tested for the scenes of rigid boxes with uniformly sized particles generated using voxel merging. Compared to Table 6.5, the bottom-up algorithm using bounding spheres in combination with stack-less MBVH2 (Column **Bot.Up-216K**) is occasionally more efficient than in combination with stack-based traversal (Column **Stack Sp.-216K**). This degradation is probably caused by deeper stacks due to the 63K more particles.

	Stack Sp. - 216K	**AABB - 216K**	**64^3 Hash**	**128^3 Hash**
1	7.44	5.81	1.54	1.07
2	14.18	8.01	6.18	1.10
3	14.99	9.01	11.16	2.74
4	15.63	9.69	14.48	3.60
5	16.37	10.47	17.12	4.26
6	16.84	11.40	18.12	4.75
7	16.51	12.02	18.46	4.98
8	15.96	12.10	18.71	5.03

Table 6.7: Running time (in *ms*) for different traversal methods in simulating the scene consisting of 216K granular-like particles generated by replacing the shape matching constraints for the rigid bodies with collision constraints. In this test, AABBs (Column **AABB-216K**) still outperform bounding spheres (Column **Stack Sp.-216K**).

the most efficient method to deal with differently sized particles.

For scenes consisting of rigid bodies, typically one time step of 16.7 *ms* per frame (corresponding to 60 fps, which is the maximum refresh rate of most monitors), 6 solver iterations and 2 pre-stabilization iterations (Section 3.1.5) per time step are enough. However, for large granular piles, more solver iterations and pre-stabilization iterations are needed to achieve satisfactory results (Fig. 6.4).

In contrast, according to our experiments, fluid simulations need 2 sub-steps per frame, with each being 8.33 *ms*, 2 solver iterations and 2 pre-stabilization iterations per sub-step to avoid large density estimation errors, otherwise individual fluid particles might be push out of the fluids with very large velocities. The length of the neighbor index array for each particle should be at least 48 in order to keep the density estimation accurate enough to avoid unrealistic behavior of the fluid particles. A typical smooth kernel radius is 4 times of the fluid particle radius. Because of the complexity of the BVH traversal and the fluid constraint, the frame rates drop to ca. 40 fps with only 37.5K fluid particles.

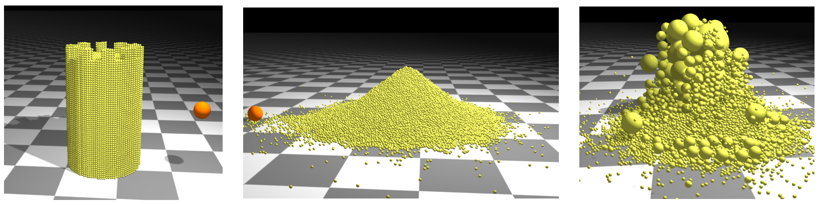


Fig. 6.4: One large particle collides with a granular pile consisting of 64 934 particles. Despite the 16 solver iterations and 16 pre-stabilization iterations per time step needed to keep the pile stable at the beginning of the simulation, real-time performance (60 fps) can still be achieved. In the image on the right, the differently sized particles are generated from merged voxels.

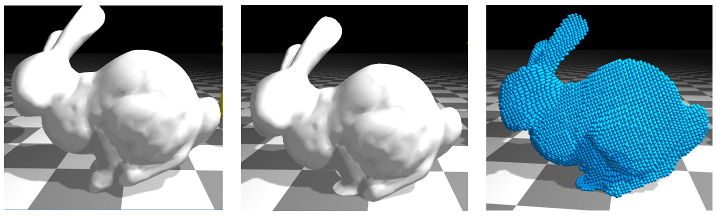


Fig. 6.5: A deformable bunny. By using a large grid size (60^3) for the voxelization, the deformation can be realistically simulated.

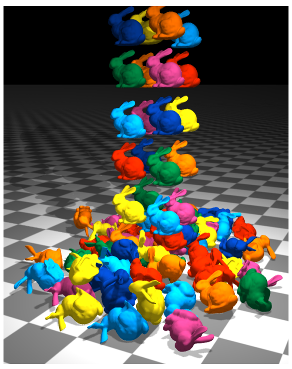


Fig. 6.6: 80 bunnies are falling towards the ground. Each bunny is voxelized with the grid size 20^3. The scene consists of 128 480 particles. With voxel merging, the number of the particles is reduced to 104 960. Under both circumstances, 60 fps can be achieved.

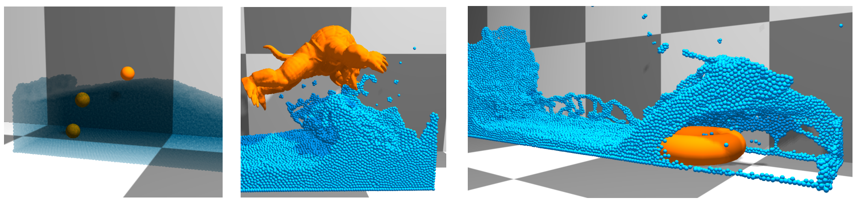


Fig. 6.7: In the image on the left, three large solid particles of different densities interact with fluid particles; The image in the middle shows that Armadillo is thrown up in the air by the fluid particles due to its small density; In the image on the right, the fluid is torn apart by a heavy rigid body.

7. Conclusion

In this project we have successfully reduced the time expense on the initialization of scenes by implementing and optimizing algorithms for fast voxelization and SDF constructions on the GPU. We implemented and compared several algorithms for constructing and traversing BVHs on the GPU, which enables us to simulate scenes containing tens of thousands differently sized particles in real-time. Merging particles inside an object allows us to use smaller particles on the surface of the object for a more detailed simulation in interactive applications, meanwhile the total amount of particles are even less. Our methods are preferable when the available memory is very limited.

7.1 Limitations

Nevertheless, there are still some limitations of these methods. The solid voxelization (Section 4.3.3) may still suffer from numerical errors for regular meshes such as Cartesian grids, because the voxel centers are very likely to be projected onto the edges of two parallel triangles (Fig. 7.1). One possible solution might be to apply larger scale to the mesh than the one used in Section 4.3.3, since this problem does not arise for the simple box mesh consisting of 12 triangles.

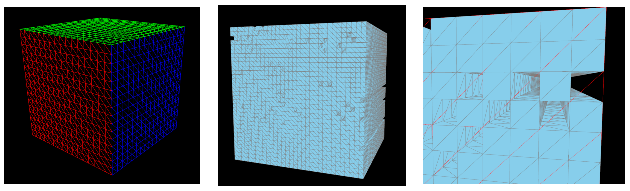
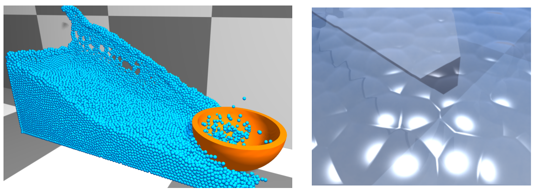


Fig. 7.1: The solid voxelization is applied to a Cartesian grid. Some voxels are erroneously classified. The red lines in the image on the right are the edges of the mesh triangles.

Neighbor finding for fluid particles could be very slow when using the BVH due to the requirement of smaller time steps and a relatively large SPH kernel radius.

One problem with the solver is the slow convergence caused by the shape matching constraint due to the inherent under-relaxation (Section 4.1.2) when one rigid body consists of a large amount of particles. Besides, the shape matching constraint is only able to simulate small deviations from the initial shape. Region-based shape matching [Bender et al., 2015; Müller et al., 2005] can be used for larger deformations.

Another inherent problem with all the PBD-based simulations is tunneling due to high velocities of the particles (Fig. 7.2a). Smaller time steps can mitigate this problem, but the real-time performance might not be reached this way. For fluid rendering, the screen space techniques [Green, 2010] are very popular for real-time applications. However, it could be problematic under some circumstances (Fig. 7.2b).



(a) Tunneling (b) Fluid surface

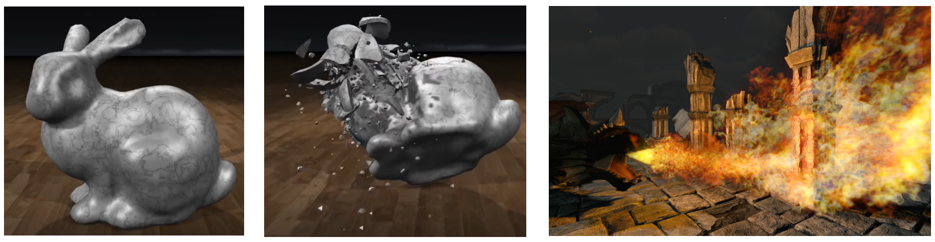
Fig. 7.2: The image on the left shows the tunneling between fluid particles and the bowl due to its thin-shell structure; The image on the right shows the fluid surface rendered in screen space by the NVidia Flex demo. The shape of the particles is still visible when the camera comes close to the surface.

7.2 Future Work

In the future, we would like to integrate fracture effect (Fig. 7.3b), namely the dynamic destruction of objects into the particle-based framework using the fast GPU voxelization and SDF construction algorithms. An apposite method for adaptively weighting the particles in shape matching would be desirable to achieve faster convergence (Section 4.1.2).

As has been described in the previous chapter, the hash grid becomes less efficient when the distribution of the particles concentrates on a 2D plane. This problem could possibly be mitigated by shaping the hash grid according to the distribution of the particles. One simple way is to change the size of the grid based on the AABB of the particles, which can be efficiently determined by two reductions.

In neighbor finding for fluid particles, it could be possible to directly use the Morton curve when simulating scenes consisting of differently sized particles, thereby reducing the cost of the BVH traversal. However, one should be very careful when using this approach, because the behavior of fluids is very sensitive to density estimation errors. For fluid rendering, we would like to reconstruct the fluid surface using the method proposed by Zhou et al. [2011]. Although the behavior of fire is also governed by the fluid constraint, the rendering of fire could be very challenging, and most of the current implementations take an Eulerian approach (Section 2.3). It is very intriguing to integrate the fire simulation into the unified particle-based solver.



(a) (b) (c)

Fig. 7.3: The two images on the left show the fracture simulation by Müller et al. [2013]; The image on the right shows the fire simulation using NVidia Flameworks System which takes an Eulerian approach [Green and Chentanez, 2014].

Index

Bibliography

A. T. Áfra and L. Szirmay-Kalos, "Stackless multi-bvh traversal for cpu, mic and gpu ray tracing," in *Computer Graphics Forum*, vol. 33, no. 1. Wiley Online Library, 2014, pp. 129–140.

G. Allaire and S. M. Kaber, *Numerical linear algebra.* Springer, 2008, vol. 55.

M. Bader, *Space-filling curves: an introduction with applications in scientific computing.* Springer Science & Business Media, 2012, vol. 9.

R. Barringer and T. Akenine-Möller, "Dynamic stackless binary tree traversal," *Journal of Computer Graphics Techniques*, vol. 2, no. 1, pp. 38–49, 2013.

J. Bender, M. Müller, and M. Macklin, "Position-based simulation methods in computer graphics." in *Eurographics (Tutorials)*, 2015.

Å. Björck, *Numerical methods in matrix computations.* Springer, 2015.

K. Bodin, C. Lacoursiere, and M. Servin, "Constraint fluids," *IEEE Transactions on Visualization and Computer Graphics*, vol. 18, no. 3, pp. 516–526, 2012.

R. Bridson, R. Fedkiw, and J. Anderson, "Robust treatment of collisions, contact and friction for cloth animation," *ACM Transactions on Graphics (ToG)*, vol. 21, no. 3, pp. 594–603, 2002.

T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, "Parallel banding algorithm to compute exact distance transform with the gpu," in *Proceedings of the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games.* ACM, 2010, pp. 83–90.

K. Crane, I. Llamas, and S. Tariq, "Real-time simulation and rendering of 3d fluids," *GPU gems*, vol. 3, no. 1, 2007.

P.-E. Danielsson, "Euclidean distance mapping," *Computer Graphics and image processing*, vol. 14, no. 3, pp. 227–248, 1980.

P. Danilewski, S. Popov, and P. Slusallek, "Binned sah kd-tree construction on a gpu," *Saarland University*, pp. 1–15, 2010.

M. Desbrun, P. Schröder, and A. Barr, "Interactive animation of structured deformable objects," in *Graphics Interface*, vol. 99, no. 5, 1999, p. 10.

C. Deul, P. Charrier, and J. Bender, "Position-based rigid-body dynamics," *Computer Animation and Virtual Worlds*, vol. 27, no. 2, pp. 103–112, 2016.

E. DiBenedetto, *Classical Mechanics: Theory and Mathematical Modeling*, ser. Cornerstones. Birkhäuser Boston, 2010.

E. Eisemann and X. Décoret, "Single-pass gpu solid voxelization for real-time applications," in *Proceedings of graphics interface 2008.* Canadian Information Processing Society, 2008, pp. 73–80.

S. Fang and H. Chen, “Hardware accelerated voxelization,” *Computers & Graphics*, vol. 24, no. 3, pp. 433–442, 2000.

K. Garanzha, J. Pantaleoni, and D. McAllister, “Simpler and faster hlbvh with work queues,” in *Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics.* ACM, 2011, pp. 59–64.

S. Green, “Cuda particles,” *NVIDIA whitepaper*, vol. 2, no. 3.2, p. 1, 2008.

——, “Screen space fluid rendering for games,” in *Proceedings for the Game Developers Conference*, 2010.

S. Green and N. Chentanez, “Smoke and mirrors: Advanced volumetric effects for games,” 2014.

G. J. Grevera, “Distance transform algorithms and their implementation and evaluation,” in *Deformable Models.* Springer, 2007, pp. 33–60.

M. Hapala, T. Davidovič, I. Wald, V. Havran, and P. Slusallek, “Efficient stack-less bvh traversal for ray tracing,” in *Proceedings of the 27th Spring Conference on Computer Graphics.* ACM, 2011, pp. 7–12.

M. J. Harris, “Fast fluid dynamics simulation on the gpu.” in *SIGGRAPH Courses*, 2005, p. 220.

T. Hayashi, K. Nakano, and S. Olariu, “Optimal parallel algorithms for finding proximate points, with applications,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 12, pp. 1153–1166, 1998.

K. E. Hoff III, J. Keyser, M. Lin, D. Manocha, and T. Culver, “Fast computation of generalized voronoi diagrams using graphics hardware,” in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques.* ACM Press/Addison-Wesley Publishing Co., 1999, pp. 277–286.

T. Jakobsen, “Advanced character physics,” in *Game Developers Conference*, vol. 3, 2001.

M. W. Jones, J. A. Baerentzen, and M. Sramek, “3d distance fields: A survey of techniques and applications,” *IEEE Transactions on visualization and Computer Graphics*, vol. 12, no. 4, pp. 581–599, 2006.

T. Karras, “Maximizing parallelism in the construction of bvhs, octrees, and k-d trees,” in *Proceedings of the Fourth ACM SIGGRAPH/Eurographics conference on High-Performance Graphics.* Eurographics Association, 2012, pp. 33–37.

M. N. Kolountzakis and K. N. Kutulakos, “Fast computation of the euclidian distance maps for binary images,” *Information processing letters*, vol. 43, no. 4, pp. 181–184, 1992.

C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, and D. Manocha, “Fast bvh construction on gpus,” in *Computer Graphics Forum*, vol. 28, no. 2. Wiley Online Library, 2009, pp. 375–384.

S. Le Grand, “Broad-phase collision detection with cuda,” *GPU gems*, vol. 3, pp. 697–721, 2007.

Y.-H. Lee, S.-J. Horng, and J. Seltzer, “Parallel computation of the euclidean distance transform on a three-dimensional image array,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 3, pp. 203–212, 2003.

W. Li, Z. Fan, X. Wei, and A. Kaufman, “Gpu-based flow simulation with complex boundaries,” *GPU Gems*, vol. 2, pp. 747–764, 2003.

Y. Lu, “A framework for comparison of methods for solving complementarity problems that arise in multibody dynamics,” Ph.D. dissertation, Rensselaer Polytechnic Institute, 2016.

L. B. Lucy, “A numerical approach to the testing of the fission hypothesis,” *The astronomical journal*, vol. 82, pp. 1013–1024, 1977.

M. Macklin and M. Müller, “Position based fluids,” *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, p. 104, 2013.

M. Macklin, M. Müller, N. Chentanez, and T.-Y. Kim, “Unified particle physics for real-time applications,” *ACM Transactions on Graphics (TOG)*, vol. 33, no. 4, p. 153, 2014.

C. R. Maurer, R. Qi, and V. Raghavan, “A linear time algorithm for computing exact euclidean distance transforms of binary images in arbitrary dimensions,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 25, no. 2, pp. 265–270, 2003.

J. J. Monaghan, “Sph without a tensile instability,” *Journal of Computational Physics*, vol. 159, no. 2, pp. 290–311, 2000.

J. P. Morris, “Simulating surface tension with smoothed particle hydrodynamics,” *International journal for numerical methods in fluids*, vol. 33, no. 3, pp. 333–353, 2000.

M. Müller, D. Charypar, and M. Gross, “Particle-based fluid simulation for interactive applications,” in *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation.* Eurographics Association, 2003, pp. 154–159.

M. Müller, B. Heidelberger, M. Teschner, and M. Gross, “Meshless deformations based on shape matching,” *ACM transactions on graphics (TOG)*, vol. 24, no. 3, pp. 471–478, 2005.

M. Müller, B. Heidelberger, M. Hennix, and J. Ratcliff, “Position based dynamics,” *Journal of Visual Communication and Image Representation*, vol. 18, no. 2, pp. 109–118, 2007.

M. Müller, N. Chentanez, and T.-Y. Kim, “Real time dynamic fracture with volumetric approximate convex decompositions,” *ACM Transactions on Graphics (TOG)*, vol. 32, no. 4, p. 115, 2013.

M. Müller, J. Bender, N. Chentanez, and M. Macklin, “A robust method to extract the rotational part of deformations,” in *Proceedings of the 9th International Conference on Motion in Games.* ACM, 2016, pp. 55–60.

A. Nealen, M. Müller, R. Keiser, E. Boxerman, and M. Carlson, “Physically based deformable models in computer graphics,” in *Computer graphics forum*, vol. 25, no. 4. Wiley Online Library, 2006, pp. 809–836.

NVIDIA, “Cuda c programming guide,” *NVIDIA, January*, 2017.

——, “Thrust quick start guide,” 2017.

J. Pantaleoni and D. Luebke, “Hlbvh: hierarchical lbvh construction for real-time ray tracing of dynamic geometry,” in *Proceedings of the Conference on High Performance Graphics.* Eurographics Association, 2010, pp. 87–95.

J. Pineda, “A parallel algorithm for polygon rasterization,” in *ACM SIGGRAPH Computer Graphics*, vol. 22, no. 4. ACM, 1988, pp. 17–20.

S. Rao, *Engineering optimization: theory and practice.* John Wiley & Sons, 2009.

A. Rosenfeld and J. L. Pfaltz, “Sequential operations in digital picture processing,” *Journal of the ACM (JACM)*, vol. 13, no. 4, pp. 471–494, 1966.

N. Satish, M. Harris, and M. Garland, “Designing efficient sorting algorithms for manycore gpus,” in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on.* IEEE, 2009, pp. 1–10.

J. Schneider, M. Kraus, and R. Westermann, “Gpu-based real-time discrete euclidean distance transforms with precise error bounds.” in *VISAPP (1)*, 2009, pp. 435–442.

M. Schwarz and H.-P. Seidel, “Fast parallel surface and solid voxelization on gpus,” in *ACM Transactions on Graphics (TOG)*, vol. 29, no. 6. ACM, 2010, p. 179.

J. Stam, “Stable fluids,” in *Proceedings of the 26th annual conference on Computer graphics and interactive techniques.* ACM Press/Addison-Wesley Publishing Co., 1999, pp. 121–128.

A. Witkin, M. Gleicher, and W. Welch, *Interactive dynamics.* ACM, 1990, vol. 24, no. 2.

K. Zhou, M. Gong, X. Huang, and B. Guo, “Data-parallel octrees for surface reconstruction,” *IEEE Transactions on Visualization and Computer Graphics*, vol. 17, no. 5, pp. 669–681, 2011.

Erklärung

Ich versichere, dass ich die Arbeit selbstständig verfasst habe und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe. Die Arbeit wurde in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt und von dieser als Teil einer Prüfungsleistung angenommen.

Karlsruhe, den September 16, 2017

(Chao Jia)