University of West Bohemia

Faculty of Applied Sciences
Department of Computer Science and Engineering

Diploma Thesis

The Use of Voronoi Diagram of Spheres for Granular Models

Pilsen, 2012

Tatyana Peshkova

I hereby declare that this diploma thesis is completely my own work and that I used only the cited sources.

Pilsen, July 30, 2012 Tatyana Peshkova

Acknowledgements

This thesis could not have been written without Prof. Dr. Ing. Ivana Kolingerová, who has been a great tutor and supervisor. I especially would like to thank her for her help, support and experience, which she shared with me so freely.

My study in the University of West Bohemia would not have been possible without Prof. RNDr. Stanislav Míka, CSc. and his international student exchange program. Many thanks to Prof. Ing.Václav Skala, CSc. for his support, guidance and generous welcome at the Department of Computer Science and Engineering.

Lastly I wish to thank the members of my family and my husband, who all have been so understanding, incredibly supportive and have been with me throughout the years I dedicated to my studies.

Abstract

A lot of geometric objects in computer simulations of physical systems and computer graphics are approximated by a set of spheres. The Voronoi diagram has been widely used to solve applied problems in many of those disciplines due to its properties. One of the general uses of the Voronoi approach in physics is disordered packing of balls and models of liquid and gasses. The Voronoi-Delaunay technique is a useful tool for analysis of voids, e.g., empty spaces between atoms, where the Voronoi network represents the navigation map.

The primary objective of the diploma thesis has been to experiment with the Voronoi diagrams of spheres (balls) in granular models. As a proposed solution it was suggested to test the Voronoi diagrams in sphere packing problem and granular motion simulation. The main application uses the library of Voronoi diagram of balls implemented by Ing. M. Maňák and Smoothed Particle Hydrodynamics open source to solve the physics issue.

The work has been divided into two main parts, a survey of the Voronoi diagram of spheres, introduction into its geometry primitives and topology. The second part proposes two types of the diagram application, sphere packing and dynamic fluid simulation. The main application is implemented in C♯ programming language since the tested dynamic library was written in C♯ , the physics library has been added as a DLL library.
All experimental results were obtained on an Intel(R) Core(TM) i5 CPU, 240GHz, 4GB RAM using C♯, Windows 7.

Contents

List of Figures **vi**

1 Introduction **1**

2 Euclidean Voronoi Diagrams of 3D Balls **3**
2.1 Definition . . . 3
2.1.1 Properties of the Euclidean Weighted Voronoi Diagram in R^d . . . 5
2.1.2 Geometrical and Topological Properties . . . 6
2.2 Quasi Triangulation . . . 6
2.2.1 Simplex, Simplicial Complex, Triangulation . . . 7
2.2.2 Quasi Triangulation . . . 7
2.3 Geometry and Topology Representation . . . 9
2.4 Algorithms of VD(S) Construction . . . 11
2.4.1 Edge Tracing . . . 11
2.4.2 Region Expansion . . . 11

3 Application of Voronoi Diagram of 3D Balls **13**

4 Voronoi Diagram of 3D Balls Library **19**
4.1 M. Maňák's Voronoi Diagram of Spheres Library . . . 19
4.1.1 Example . . . 20
4.1.2 Other properties of VDS . . . 21

5 Smoothed Particle Hydrodynamics **24**
5.1 FLUIDS v.2 . . . 24
5.1.1 Example . . . 25

6 Granular models **28**
6.1 Sphere Packing . . . 28
6.1.1 The outline of packing spheres in a container . . . 28
6.2 Granular simulation . . . 35
6.2.1 Proposed Application of VDS in Granular Simulation . . . 35

6.2.2 Plane, Cylinder 36
6.3 Application 41

7 Results **43**
7.1 Sphere Packing 43
7.1.1 Cube 43
7.1.2 Cylinder 48
7.1.3 Triangle Mesh 52
7.2 Granular Fluid Simulation 57

8 Conclusion **59**

Abbreviations and Notation **60**

References **61**

List of Figures

2.1 2D additively Voronoi diagram 4
2.2 3D Voronoi region 4
2.3 The Euclidean bisector 5
2.4 Geometry and topology interpretations 6
2.5 Voronoi diagram and its corresponding quasi-triangulation 8
2.6 Anomalies 8
2.7 Data structure for topology representation of a Quasi triangulation 10
2.8 Geometry and Topology 10
2.9 Region expansion algorithm[6] 11

3.1 The Voronoi - Delaunay approach for the free volume analysis of a packing of balls in a cylindrical container 16
3.2 Computing Voronoi cell 17
3.3 Computing Voronoi cell 18

4.1 VDs library 19

5.1 FLUIDS v.2 - A Fast, Open Source, Fluid Simulator 24

6.1 Illustration of candidates to insert into the object 29
6.2 Packing into a container 30
6.3 An initial diagram 31
6.4 Bunny partition 33
6.5 2D Voronoi based medial axis 33
6.6 Bunny mesh 34
6.7 Illustration of balls behaviour 35
6.8 Granules falling on plane 38
6.9 Granules motion 38
6.10 Granular motion 38
6.11 Balls intersection 39
6.12 Segment triangle intersection 39
6.13 Application 41
6.14 The scheme of application 42

7.1 Filling the cube 44
7.2 The V-algorithm diagram 44

7.3 Cube with identical balls 45
7.4 The chart of V and VE algorithms 46
7.5 Cube packing by using V- and VE- algorithms 47
7.6 Packing spheres into a cube 48
7.7 Packing spheres into a cube 48
7.8 Cylinder filling 49
7.9 Cylinder filling 50
7.10 Cylinder. VE-algorithm. 51
7.11 Packing spheres in a cylinder 52
7.13 Bunny 53
7.14 Bunny 53
7.15 Kitten 54
7.16 Bunny 55
7.17 Bunny 55
7.18 Bunny 56
7.19 Fluid motion in a cylinder. 57
7.20 Cup (triangle mesh) 58

Chapter 1

Introduction

A lot of geometric objects in areas such as computer simulation of physical systems, mechanical engineering, computer graphics, are approximated by spheres. The Voronoi diagram has been widely used to solve applied problems in many of those disciplines due to its natural descriptive and manipulative properties[1]. One of the uses of the Voronoi approach in physics is disordered packing of balls and models of liquid and gasses. The Voronoi-Delaunay technique is a useful tool for analysis of voids, e.g. empty spaces between atoms, where the Voronoi network represents the navigation map.

The primary objective of the present work has been to experiment with the use of the Voronoi diagrams of spheres (balls) in granular models. As a proposed solution it was suggested to test the Voronoi diagrams in sphere packing problem and granular motion simulation. The main application uses the library of Voronoi diagram of balls implemented by M. Maňák[2] and Smoothed Particle Hydrodynamics open source[3] to solve the physics issue.

Overview of the work

In this section we take a tour of the things discussed in the following chapters of the present work.

Chapter 2 - *Euclidean Voronoi Diagrams of 3D Balls* - explains what the Voronoi diagram of 3D balls is, describes the geometrical and topological properties. This section also contains the description of dual representation of Voronoi diagrams of spheres - *Quasi Triangulation*, its topological representation and a brief overview of two fundamental algorithms of Voronoi diagrams construction, i.e. *Edge tracing* and *Region expansion* algorithms.

Chapter 3 - *Application of Voronoi Diagrams of 3D Balls* - contains the known solutions of packing of balls. The first part of the chapter contains the description of M. Gavrilova's approach of free volume analysis of packing of balls in a cylinder. The second part describes another algorithm by C. H. Rycroft, he uses a different technique of applying a cutting plane.

Chapter 4 - *Voronoi Diagram of 3D Balls Library* - introduces the Voronoi library implemented by M. Maňák and contains a brief report of the properties of the library.

Chapter 5 - *Smoothed Particle Hydrodynamics* - since the issue of physics is a complex problem, which involves a lot of time and understanding, it was decided to use an external source for granular motion. Hence, this section shortly presents the instructions for the use of SPH.

Chapter 6 - *Granular models* - explains the algorithms for packing spheres in geometrical (cube, cylinder) and triangle models. The given chapter also discusses the problem of granular motions. The last section briefly introduces the implemented application and a a pipeline of processing.

Chapter 7 -*Results* - contains achieved results, time measurements and reveals the situations when the Voronoi diagram of spheres was successful or failed.

Chapter 9 - *Conclusion* - this chapter contains the conclusion of the work done.

Chapter 2

Euclidean Voronoi Diagrams of 3D Balls

The Voronoi diagrams for spheres or 3D balls[1] are also known as *Voronoi diagrams of 3D balls*[2]. This chapter defines the Voronoi diagram of balls, describes geometrical and topological properties, contains a brief overview of duality between Voronoi diagrams of spheres and *Quasi Triangulation*[9], geometrical and topological representation and description of two fundamental algorithms of the Voronoi diagram construction such as *Edge tracing* and *Region expansion*.

2.1 Definition

Definition 2.1.1. Let $S = \{s_1, s_2, ..., s_n\}$ be a set of generators (spheres) for a Voronoi diagram where $s_i = (c_i, r_i)$ is a three dimensional sphere with a center $c_i = (x_i, y_i, z_i)$ and a radius r_i. No sphere can be completely contained inside another sphere, but intersections between spheres are allowed. Then the additively weighted distance of a point $p \in \mathbb{R}^3$ to a sphere $s_i \in S$ is

$$d_{aw}(p, s_i) = \|p - c_i\| - r_i \tag{2.1}$$

The additively weighted *Voronoi region* of $s_i \in S$ is the set of points

$$VR(s_i) = \{p \in \mathbb{R}^3 : \forall s_j \in S, s_j \neq s_i, d_{aw}(p, s_i) \leq d_{aw}(p, s_j)\} \tag{2.2}$$

The additively weighted *Voronoi diagram* of S is the set of regions

$$VD(S) = \{VR(s_1), VR(s_2), ..., VR(s_n)\} \tag{2.3}$$

Figure 2.1 illustrates the 2D additively weighted diagram, Figure 2.2 demonstrates the 3D Voronoi region. The definition assumes non-negative radii, since adding a constant

[1]The input set is a set of spheres or balls.

[2]The input is a set of weighted points.

value to all radii does not change the diagram. There are situations when the function $d_{aw}(p, s_i)$ has negative values, which cannot be a solution in this case. That is why the another function can be used:

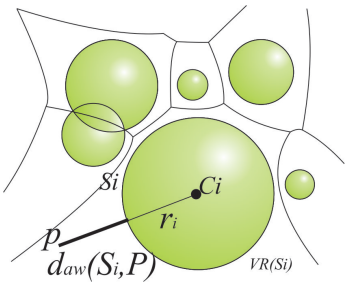

Figure 2.1: 2D additively Voronoi diagram - Illustration of the aw-distance, taken from [11]

$$u(p, s_i) = 1 + max(d_{aw}(p, s_i), 0) + \frac{min(d_{aw}(p, s_i), 0)}{r_{max}} \tag{2.4}$$

$i \in \{1, 2, ..., n\}$, $p \in \mathbb{R}^3$, $r_{max} = max\{r_i | i \in \{1, 2, ..., n\}\}$ and $d_{aw}(p, s_i) \in [-r_{max}, \infty)$. The $u(p, s_i)$ function converts all negative values of $d_{aw}(p, s_i)$ from $[-r_{max}, 0]$ to $[0, 1]$ and shifts all positive values by one.

In the case when all balls degenerate to points (all radii are zero), the distance function is the standard Euclidean distance, hence the resulting diagram is the ordinary Voronoi diagram.

The following definition of the Voronoi diagram will be useful in the context of a dual structure (*quasi triangulation*, see section 2.2).

Definition 2.1.2. Let $VD(S)$ denote as the Voronoi diagram of the sphere set S. Then $VD(S) = (V^v, E^v, F^v, C^v)$, where $V^v = \{v_1^v, v_2^v, ...\}$, $E^v = \{e_1^v, e_2^v, ...\}$, $F^v = \{f_1^v, f_2^v, ...\}$ and $C^v = \{c_1^v, c_2^v, ...\}$ denote the set of Voronoi vertices, Voronoi edges, Voronoi faces, Voronoi regions (cells), respectively.

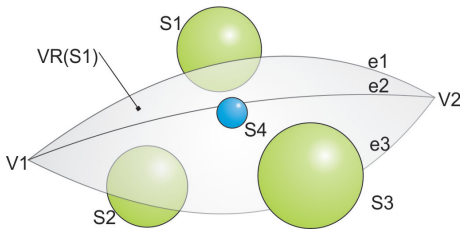

Figure 2.2: 3D Voronoi region - $VR(s_1)$- Voronoi region of a sphere s_1, e_1, e_2, e_3 - edges, v_1, v_2- vertices, taken from [11]

2.1.1 Properties of the Euclidean Weighted Voronoi Diagram in R^d

Property 2.1.1. The *bisector* $B(P, Q)$ of an Euclidean Weighted Voronoi diagram is one half of the hyperboloid with poles P and Q and parameter $l = |r_p - r_q|$ (see Figure 2.3) [19].

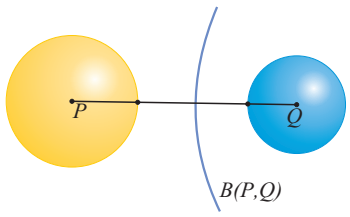

Figure 2.3: The Euclidean bisector - .

Note. If $r_p = r_q$ then the bisector is a hyperplane (i.e. a hyperboloid with parameter $l = 0$).

Property 2.1.2. Each Euclidean region $VR(S)$ is a connected region in R^d space, containing the whole sphere S. The region is a star-shaped relative to the center S.

Note: The Euclidean region is not necessarily convex. The boundary of the Euclidean region is formed by bisectors, which in general are hyperboloids, then the concave side of such a bisector belongs to a non-convex Euclidean region.

Definition 2.1.3. The Voronoi vertex v is the common intersection of exactly $d + 1$ Euclidean regions.

Definition 2.1.4. A *sphere* $S = \{c, r\}$ inscribed among $d + 1$ spheres $S_1, S_2, ..., S_{d+1}$ in the Euclidean metric is a sphere with a center $c = (c_1, c_2, ..., c_d)$ and a radius r, such that $r = dist(c, P_1) = dist(c, P_2) = ... = dist(c, P_{d+1})$.

Definition 2.1.5. An inscribed sphere S is called *empty* in the Euclidean metric if no sphere from S intersects its interior.

Property 2.1.3. (The nearest neighbour property) If $Q \in S$ is the nearest neighbour of $P \in S$ then the Euclidean regions $VR(Q)$ and $VR(P)$ has a common facet.

2.1.2 Geometrical and Topological Properties

Using definitions and properties from previous section the following geometrical and topological properties for 3D have been derived. Each Voronoi region in Voronoi diagrams of spheres is star-shaped with a respect to the center of the corresponding spherical generator. The boundary of region is constructed by faces and the boundary of faces consists of edges, while edges are defined by vertices. A Voronoi face is defined by two immediately neighbouring generators and represented as a part of plane or a hyperboloid. When a Voronoi face intersects another Voronoi face, the Voronoi edge is formed, which is a conic section[12]. Respectively, when Voronoi edges intersect, the Voronoi vertex is defined. A Voronoi vertex is a center of an empty sphere tangent to all four nearby balls, see Figure 2.4. Kim [12] denotes that the degree of a Voronoi vertex is four, since it is formed by four generators. A Voronoi edge is formed by the intersection of three Voronoi faces, thus only three generators define an edge.

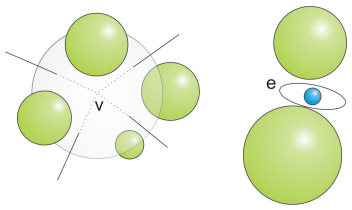

Figure 2.4: Geometry and topology interpretations - on the left side - a vertex *v* as a center of an empty sphere, on the right side - an elliptic edge without any vertices

Some regions, faces or edges could be unbounded (continue to infinity), what usually leads to problems of the diagram representation. The other problem of a construction of the diagram could be elliptic edges without any vertices, see Figure 2.4 (right illustration).

2.2 Quasi Triangulation

The duality between Voronoi diagrams and triangulations is a very important property. For example, the dual for Voronoi diagram is Delaunay triangulation. Similarly to the ordinary Voronoi diagram, a dual representation for additively weighted Voronoi diagrams exists. A duality maps diagram elements to simplices. In 1993 A. Mirzaian[20] described in his work a 2D case of duality for Voronoi diagrams, and defined the dual as a multigraph, where the edge is represented as a common face of two neighbouring Voronoi cells. Two neighbouring cells can have more faces that lead to more edges in a graph. In 2006, Kim [9] studied the duality, that dual structure was called *quasi triangulation*. However, the dual does not have to be a valid triangulation due to *anomalies*. Based on the properties of a quasi-triangulation a data structure was proposed, called an interworld data structure IWDS, eIWDS. The quasi-triangulation is defined from a Voronoi diagram of balls via a duality operator described below.

2.2.1 Simplex, Simplicial Complex, Triangulation

Boissonnat and Yvinec[4] presented in their approach the concept of a complex.

Definition 2.2.1. A simplex of dimension $k \leq d$ in d-dimensional Euclidean space is the convex hull of k+1 affinelly independent points.

Often 0-simplices are called points, 1-simplices are line segments, 2-simplices are triangles and 3-simplices are tetrahedrons.

Definition 2.2.2. A simplicial complex is a finite collection of simplices $\mathcal{K}$ such that $\sigma \in \mathcal{K}$ and $\tau \leq \sigma$ implies $\tau \in \mathcal{K}$, and $\sigma, \sigma_0 \in \mathcal{K}$ implies $\sigma \cap \sigma_0$ is either empty or a face of both.

Definition 2.2.3. A complex $\mathcal{K}$ is homogeneously d-dimensional if and only if any lower-dimensional simplex in $\mathcal{K}$ constitutes a face of some d-dimensional simplex in $\mathcal{K}$. Then d-triangulation is a homogeneously d-dimensional complex, which is connected and is without singular faces.

2.2.2 Quasi Triangulation

Definition 2.2.4. Let a *quasi – triangulation* QT(S) for a sphere set S in three dimensions be defined as $QT(S) = \{V^Q, E^Q, F^Q, C^Q\}$, where $V^Q = \{v_1^Q, v_2^Q, ..., v_n^Q\}$, $E^Q = \{e_1^Q, e_2^Q, ...\}$, $F_Q = \{f_1^Q, f_2^Q, ...\}$, $C^Q = \{c_1^Q, c_2^Q\}$ denote the sets of vertices, edges, faces, regions in the quasi triangulation, respectively[1] [9].

Let us denote **D** as a dual operator. Then **D** maps a Voronoi diagram VD(s) to a quasi triangulation QT(S):

- $\mathbf{D} : c^v \rightarrow v^Q$: maps a Voronoi cell to a dual quasi vertex. The vertex v^Q corresponds to the center c of a generator ball s corresponding to a Voronoi cell c^v.
- $\mathbf{D} : f^v \rightarrow e^Q$: maps a Voronoi face to a dual quasi edge. The edge e^Q is a line segment bounded by v_i^Q and v_j^Q, where the balls s_i, s_j define the Voronoi face f^V.
- $\mathbf{D} : e^v \rightarrow f^Q$: maps a Voronoi edge to a dual quasi face. The f^Q is a triangle whose vertices are v_j^Q, v_i^Q, v_k^Q, where the balls s_j, s_i, s_k define the Voronoi edge e^V.

[1] there is one-to-one correspondence between a vertex v_1^Q and a generator sphere s_i

- **D** : $v^v \rightarrow c^Q$: maps a Voronoi vertex to a dual quasi cell. The c^Q is a topological tetrahedron bounded by four vertices v^Q, where the v^Q corresponds to the four balls defining the Voronoi vertex v^v.

Figure 2.5 illustrates a Voronoi diagram of seven balls: two big balls (a_1, a_2), one small ball (a_3) placed between two big balls and four balls with the identical radii (a_4, a_5, a_6, a_7). Let us consider a Voronoi face defined by two big balls. The Voronoi face is bounded by Voronoi edges defined by two big balls and one of the four balls (e.g., a_1, a_2 and a_5). A Voronoi region of a small ball is bounded only by two Voronoi faces, each of which is defined by the small ball a_3 and one of the two big balls a_1, a_2. This situation implies that the small ball defines a hole in the face defined by two big balls. The boundary of the hole is defined by the big balls and the small ball. Hence, there is no edge connection. Thus, the quasi-face formed by balls a_1, a_2 and a_3 is not a part of any tetrahedral quasi-cell, which means that there can be anomalies in the triangulation.

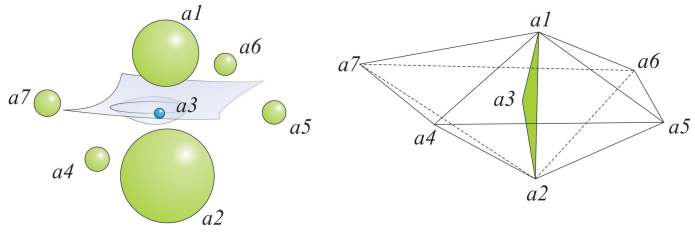

Figure 2.5: Voronoi diagram and its corresponding quasi-triangulation - taken from [11]

The dual structure QT(S), which is called quasi-triangulation, has some drawbacks, it is not always a valid triangulation in Euclidean space. Those anomalies are caused by elliptic Voronoi edge (degeneracy), which is defined without any Voronoi vertex. Hence, quasi-faces in QT(S) being not a part of any quasi-cell. Another anomaly is a hole in Voronoi faces(singuarity), which in dual spaces occurs between two quasi-cells, sharing a quasi-edge (singular). This anomaly tends to break the definition of triangulation. And the last anomaly corresponds to doubled Voronoi vertices (multiplicity), when in dual space two quasi-cells share more than one quasi-face, see Figure 2.6.

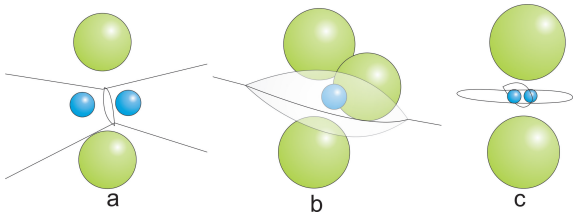

Figure 2.6: Anomalies - (a), (b) - example of multiplicity anomaly. (a) Two common triangular faces between two small-tetrahedra, and (b) three common triangular faces between two small-tetrahedra. (c) example of singularity anomaly.

Voronoi diagram is represented as face-connected, while quasi triangulation is edge-connected. Voronoi faces can have topological holes, thus the Voronoi diagram is edge-

disconnected, what leads to QT face-disconnection. Considering this situation of face-connectedness, the whole QT can be divided into face-connected component, called *worlds*.

The disconnection causes problems, that is why it is important to make some connectivity between them. The connectivity is realized via quasi-edges. There is only one quasi-edge, which connects two worlds - *gate edge*. A gate edge is an intersection of two neighbouring worlds, which result is a pair of quasi-vertices[1].

A world, which contains other worlds is called a *big world*, otherwise it is a *small world*. Big and small are relative terms, i.e., a small world can be big for some other world. The hierarchy creates a tree connected by gates. A gate always connects one big world with all small worlds, which are directly beneath it.

2.3 Geometry and Topology Representation

First structure to store VD data was a radial edge structure *RED*. This structure has been developed for storing non-manifold models[10], see Figure 2.7. The idea of RED was to represent all elements in VD (e.g. edges, vertices, faces, regions or cells) and two supplementary entities - Lf for holding the holes in faces and Fr for dealing with the adjacency of faces to an edge. Nevertheless, using of this structure was not memory efficient since the VD are not so general.

In 2006 *quasi triangulation* and *interworld data structure* were suggested, known as *IWDS* or *eIWDS*. The idea of this structure is based on Delaunay triangulation. Each quasi cell references to its four quasi vertices and four neighboring cells, which share a common quasi-face. Degeneracy is solved as a special case of a quasi-cell. Each quasi-vertex references one of its incident quasi-cell, remaining incident quasi-cells can be obtained by a topology traversal algorithm. Quasi-faces and quasi edges are not represented explicitly, thus the gates have to be represented explicitly, see Figure 2.8(left scheme). A gate references $1 + m_{SM}$ quasi-cells (1 - big world, m_{SM} - small worlds belonged to a given gate). A gate references two vertices creating the corresponding edge.

[1] there can be more than one quasi edge defined for on a same pair of quasi-vertices

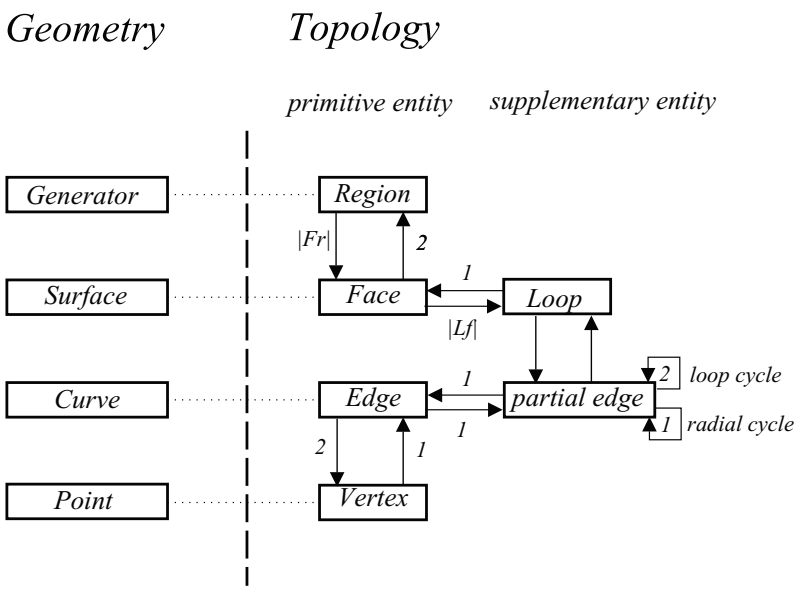

Figure 2.7: Data structure for topology representation of a Quasi triangulation - IWDS representation on left the side and eIWDS representation on the right side

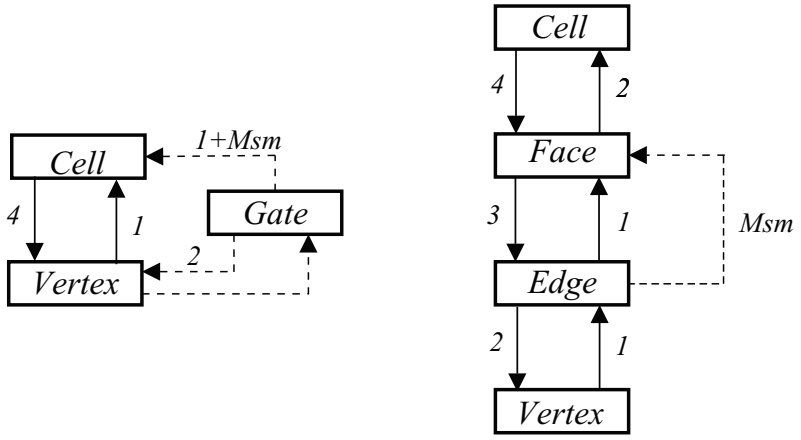

Figure 2.8: Geometry and Topology - Topological representation of Euclidean Voronoi diagram of spheres, Fr - number of faces bounding the corresponding region, Lf - number of loops in the corresponding face.

IWDS is useful in storing quasi triangulation. Its implicit representation of quasi edges and quasi faces saves memory.

When it is required to have an explicit representation during the processing then this is the case of eIWDS (gate is represented implicitly), see Figure 2.8 (right scheme).

2.4 Algorithms of VD(S) Construction

This section briefly describes the existing algorithms of the Voronoi diagram of spheres construction.

2.4.1 Edge Tracing

One of the techniques of a construction of the Voronoi diagram is *edge tracing*[2, 12]. The main idea of the edge tracing is quite simple. The algorithm first locates a true Voronoi vertex v_0 by computing an empty tangent sphere defined by four appropriate nearby balls. The edges e_0, e_1, e_2 and e_3 are emanated from a given Voronoi vertex v_0, and then pushed into a *Edge-stack*. In other words, the starting vertex for all of these edges is v_0. After popping the edge from the stack, the algorithm finds the end vertex of the popped edge. The end vertex can be found by calculating an empty sphere tangent to four balls (three balls, which define the edge and one of the $(n-3)$ candidate balls). When an empty sphere is found, its center can be announced as the end vertex of the popped edge. Once the end vertex is found, it is possible to define another three new edges emanating from this new vertex. Thus, this new edge starts from the new vertex (which was just computed) and pushed into *Edge-stack*. The algorithm continues until the stack is empty.

The *Edge tracing* algorithm runs in $O(mn)$ time in the worst case, where m is the number of edges and n is the number of balls. The number of algorithm iterations is $O(m)$ since it is required to trace each edge. For each edge it is necessary to check all n candidates, to find a valid tangent sphere.

Luchnnikov[7] introduced an algorithm based on the computation of the trajectory of an empty sphere moving inside the system and variable in size. Medvedev[8] published another algorithm for construction of Voronoi S-network using a dual representation. M. Maňák presented algorithm for fast discovery of Voronoi vertices in the construction of Voronoi diagram of 3D balls[11] using Delaunay triangulation.

2.4.2 Region Expansion

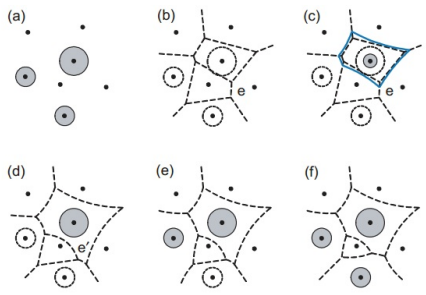

Figure 2.9: Region expansion algorithm[6] - Scheme of region expansion in 2D.

Kim[6] presents a region expansion algorithm for construction of the Voronoi diagram. The proposed algorithm first constructs the Voronoi diagram of the centers of spheres (Voronoi diagram of points). Consider the centers as a shrunk spheres, the algorithm gets each sphere one after another and expands to its full size. While a sphere expands, the Voronoi region corresponding to the given sphere expands simultaneously, see Figure 2.9. By repeating this procedure for all spheres, the correct topology of the Voronoi diagram is obtained. The Voronoi diagram here is represented as a variation of radial edge data structure since the diagram is a cell structure, one of the common nonmanifold models, bounded by faces where a face may have an arbitrary number of vertices and it can even have some topological holes inside[6]. Author also mentioned that edge-disconnectedness is no more a special case.

The computation required to expand a Voronoi region (starting from a center point to a complete ball) takes $O(n^2 \log n)$ time in the worst case. Lets note that there can be $O(n^2)$ number of edges in the Voronoi diagram of spheres and sorting all events is necessary in the algorithm. Therefore, the construction of the whole Voronoi diagram using this algorithm takes $O(n^3 \log n)$ time in the worst case.

Chapter 3

Application of Voronoi Diagram of 3D Balls

The Voronoi diagrams are well-known in mathematics and computer graphics due to their properties, and are widely used in solving physical problems. Originally, the Voronoi-Delaunay approach in physics was applied to study the structure of disordered packing of balls. Another application is empty spaces detection between atoms, where the Voronoi diagram represents a navigation map. This section provides a brief overview of the application of additively weighted Voronoi diagrams.

Free volume analysis of a packing of balls in a cylinder

V.A. Luchnikov and M.L. Gavrilova introduce in [7] the Voronoi-Delaunay approach for the free volume analysis of a packing of balls in a cylinder. They used the generalized Voronoi diagram as a data structure. They solve two problems: efficient construction of the Voronoi diagram inside a cylindrical boundary and the analysis of the Voronoi network to study a distribution of empty spheres in the system.

The algorithm is based on the idea of the *empty sphere*. Let us assume that the empty sphere is moving inside the system, so it touches at least three objects at any moment of time, thus in that case the center of the sphere is moving along the edge of the 3D Voronoi diagram/network. The distance from any point in the space to any object is expressed by an explicit function d, and the trajectory of the center of the Delaunay empty sphere can be computed numerically by performing a series of small shifts along the edge. The direction of the shift v is calculated :

$$(\nabla d_i \cdot v) = (\nabla d_j \cdot v) = (\nabla d_k \cdot v) \tag{3.1}$$

where i, j, k are the indices of the objects touched by the sphere. For a cylindrical wall (for simplicity authors assume a vertical cylinder) the following distance equation is used:

$$d_c = R_c - \sqrt{(x^2 + y^2)} \tag{3.2}$$

where R_c is the radius of a cylinder, x, y are the coordinates of a point inside the cylinder, providing the origin on the axis of the cylinder. Distance function d_c is a differentiable function of the coordinates, except for the axis of the cylinder[1]. The main advantage of

this algorithm is its simplicity and it can be used for constructing the Voronoi network for the system of any convex non-spherical objects, which do not have an explicit formula to compute the coordinates of the Voronoi vertex.

The goal is to inscribe a sphere between cylinder boundary and three not-intersecting balls inside a (vertical) cylinder. Let us denote the sphere with the smallest radius as (x_4, y_4, z_4, r_4), and choose the origin at the center of this sphere. Then shrink all balls by radius r_4 and increase the cylinder radius by r_4. Hence, we obtain the set of equations representing the condition when the sphere (x, y, z, r) touches all three spheres and the cylinder.

$$\begin{aligned}(x - x_1)^2 + (y - y_1)^2 &= (r - r_1)^2 \\ (x - x_2)^2 + (y - y_2)^2 + (z - z_2)^2 &= (r - r_2)^2 \\ (x - x_3)^2 + (y - y_3)^2 + (z - z_3)^2 &= (r - r_3)^2 \\ x^2 + y^2 + z^2 &= r^2\end{aligned} \tag{3.3}$$

where (x_1, y_1, r_1) are the coordinates of the axis of the cylinder and its radius, and (x_i, y_i, z_i, r_i), $i = 2, 3$ are the coordinates and radii of two remaining spheres. Then we subtract the last equation from the first three and obtain the following system:

$$\begin{aligned}2xx_1 + 2yy_1 - 2rr_1 &= x_1^2 + y_1^2 - r_1^2 - z_1^2 \\ 2xx_2 + 2yy_2 + 2rr_2 &= x_2^2 + y_2^2 + z_2^2 - r_2^2 - 2zz_2 \\ 2xx_3 + 2yy_3 + 2rr_3 &= x_3^2 + y_3^2 + z_3^2 - r_3^2 - 2zz_3 \\ x^2 + y^2 + z^2 &= r^2\end{aligned} \tag{3.4}$$

The first three equations from (3.4) are used to express (x, y, r) through z:

$$\mathbf{A}\begin{bmatrix}x \\ y \\ r\end{bmatrix} = \mathbf{b} \tag{3.5}$$

$$\mathbf{A} = \begin{bmatrix}x_1 & y_1 & -r_1 \\ x_2 & y_2 & z_2 \\ x_3 & y_2 & z_3\end{bmatrix}$$

$$\mathbf{b} = \begin{bmatrix}x_1^2 + y_1^2 - r_1^2 - z^2 \\ x_2^2 + y_2^2 + z_2^2 r_2^2 - 2zz_2 \\ x_3^2 + y_3^2 + z_3^2 r_3^2 - 2zz_3\end{bmatrix}$$

Let's denote $l_1 = x_1^2 + y_1^2 - r_1^2$, $l_2 = x_2^2 + y_2^2 + z_2^2 r_2^2$ and $l_3 = x_3^2 + y_3^2 + z_3^2 r_3^2$.

By solving the equation below we obtain:

$$x = \frac{1}{2|\mathbf{A}|}\begin{vmatrix} l_1 - z_2 & y_1 & -r_1 \\ l_2 - 2zz_2 & y_2 & r_2 \\ l_3 - 2zz_3 & y_3 & r_3 \end{vmatrix} \tag{3.6}$$

$$y = \frac{1}{2|\mathbf{A}|}\begin{vmatrix} x_1 & l_1 - z^2 & -r_1 \\ x_2 & l_2 - 2zz_2 & r_2 \\ x_3 & l_3 - 2zz_3 & r_3 \end{vmatrix}$$

$$r = \frac{1}{2|\mathbf{A}|}\begin{vmatrix} x_1 & y_1 & l_1 - z^2 \\ x_2 & y_2 & l_2 - 2zz_2 \\ x_3 & y_3 & l_3 - 2zz_3 \end{vmatrix}$$

where $|\mathbf{A}|$ is determinant of the matrix. If $|\mathbf{A}| = 0$ then it is the case of degeneracy, i.e., infinitely many spheres to inscribe. Then let us assume $|\mathbf{A}| \neq 0$ and substitute obtained equations (3.6) into the last equation of the system (3.4): $x^2 + y^2 + z^2 - r^2 = 0$ and the polynomial of 4th degree:

$$az^4 + bz^3 + cz^2 + dz + e = 0 \tag{3.7}$$

where:

$$\begin{aligned} a &= A_x^2 + A_y^2 - A_r^2 \\ b &= 2(A_xB_x + A_yB_y - A_rB_r) \\ c &= B_x^2 + 2A_xC_x + b_y^2 + 2A_yC_y + 4|\mathbf{A}|^2 - B_r^2 - 2A_rC_r \\ d &= 2(B_xC_x + b_yC_y - B_rC_r) \\ e &= C_x^2 + C_y^2 - C_r^2 \end{aligned} \tag{3.8}$$

and where:

$$A_x = \begin{vmatrix} -1 & y_1 & -r_1 \\ 0 & y_2 & r_2 \\ 0 & y_3 & r_3 \end{vmatrix}, B_x = -2\begin{vmatrix} 0 & y_1 & -r_1 \\ z_2 & y_2 & r_2 \\ z_3 & y_3 & r_3 \end{vmatrix}, C_x = \begin{vmatrix} l_1 & y_1 & -r_1 \\ l_2 & y_2 & r_2 \\ l_3 & y_3 & r_3 \end{vmatrix} \tag{3.9}$$

$$A_y = \begin{vmatrix} x_1 & -1 & -r_1 \\ x_2 & 0 & r_2 \\ x_3 & 0 & r_3 \end{vmatrix}, B_y = -2\begin{vmatrix} x_1 & 0 & -r_1 \\ x_2 & z_2 & r_2 \\ x_3 & z_3 & r_3 \end{vmatrix}, C_y = \begin{vmatrix} x_1 & l_1 & -r_1 \\ x_2 & l_2 & r_2 \\ x_3 & l_3 & r_3 \end{vmatrix}$$

$$A_r = \begin{vmatrix} x_1 & y_1 & -1 \\ x_2 & y_2 & 0 \\ x_3 & y_3 & 0 \end{vmatrix}, B_x = -2\begin{vmatrix} x_1 & y_1 & 0 \\ x_2 & y_2 & z_2 \\ x_3 & y_3 & z_3 \end{vmatrix}, C_x = \begin{vmatrix} x_2 & l_1 & l_1 \\ x_2 & l_2 & l_2 \\ x_3 & l_3 & l_3 \end{vmatrix} \tag{3.10}$$

Then the equation (3.7) is solved for z, the final answer[1] for the empty sphere has the

[1]The solutions with imaginary or negative r are non-physical and excluded.

following form:

$$x_f = \frac{(A_x z^2 + B_x z + C_x)}{2|\mathbf{A}|} + x_4 \tag{3.11}$$
$$y_f = \frac{(A_y z^2 + B_y z + C_y)}{2|\mathbf{A}|} + y_4$$
$$r_f = \frac{(A_r z^2 + B_r z + C_r)}{2|\mathbf{A}|} - r_4$$
$$z_f = z + z_4$$

The result of the algorithm is illustrated in Figure 3.1.

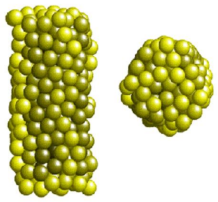

Figure 3.1: The Voronoi - Delaunay approach for the free volume analysis of a packing of balls in a cylindrical container - Model of 300 Lennard-Jones atoms obtained by Monte Carlo relaxation with the fixed diameter of a cylinder.

The Voronoi cell algorithm

Another solution of sphere packing is represented by Christopher H. Rycroft[5] in his Ph.D. thesis about multiscale modelling in granular flow. The author uses the method of cell-by-cell construction, which allows to handle non-standard boundary conditions, since the Voronoi cells can be individually tailored by applying additional plane cuts. The author suggests a mechanism for adding walls to the container class that creates additional plane and cuts during the cell computation.

In a 3D random packing, the Voronoi cell of a particular particle will be a convex irregular polyhedron, the faces of which are the perpendicular bisectors of its neighbours. It is expected that the Voronoi cell will be a simple polyhedron, while in general situation there are no limits of how the shape can be complicated.

Let us assume a single particle surrounded by a densely packed spherical shell of particles, all at some large radial separation R. The Voronoi cell will be approximately a sphere with radius R/2, with many facets due to the particles in the shell. If R increases, the number of facets increases too. Thus, the algorithm is targeted to handle the situations with an arbitrary convex polygon.

The approach suggests to compute the Voronoi cell for a particle with position $\mathbf{x}_p$ as follows[5]:

1. Surround the particle with an initial Voronoi cell consisting of the entire container volume.

2. Set $r_{test} = d/2$ (d- diameter of a bead (represented by a sphere)).

3. Find all the neighbouring particles with positions x in the spherical shell defined by $r_{test} < |\mathbf{x}_p - \mathbf{x}| < r_{test} + d$.

4. For each of these particles, cut the Voronoi cell by the plane which is the perpendicular bisector between $\mathbf{x}$ and $\mathbf{x}_p$.

5. Compute the maximal distance R of a vertex of the Voronoi cell to its center $\mathbf{x}_p$.

6. If $2R > r_{test}$, then increase r_{test} by d, and go back to step 3. If $2R \leq r_{test}$, the computation is complete.

By looping over concentric spherical shells, it is efficient to test those particles nearest $\mathbf{x}_p$ which will most likely create the facet of the Voronoi cell[5]. When we know that all particles which are left in the packing are further than $2R$ from the particle, then we can be sure that the cutting planes of those particles (which at the very least are a distance of R from $\mathbf{x}_p$) would not intersect the Voronoi cell, and thus do not need to be tested[5].

The challenging problem comes out from item 4, when it is required to represent an arbitrary convex polyhedron and the following recomputing based on cutting off an arbitrary plane. To solve this problem the author defines a structure to represent polyhedra: *a list of polyhedron vertices x_1, x_2, x_3, a table of edges (for each vertex store the three connected vertices) and relational table describing how vertices are connected back to each other.* This approach assumes that the vertices always have three edges, the case of four or more edges are treated as degenerated.

This paragraph will shortly describe the main idea of a plane cutting (Figure 3.2), the full and detailed approach could be found in [5], Chapter 4. The process starts by picking a vertex, and moving across the polyhedron to search for an edge which intersects the plane. If the plane intersects the cell, then one of the three neighbours to a vertex will be closer to the plane[1]. Once the intersected edge is found, the algorithm traces around intersected facets and find all intersected edges. On each intersected edge, a new point is created at the intersection point, and those intersection points create a new facet. Then all old vertices are removed. Figure 3.3 illustrates the stages of a cylinder packing by applying a method described above.

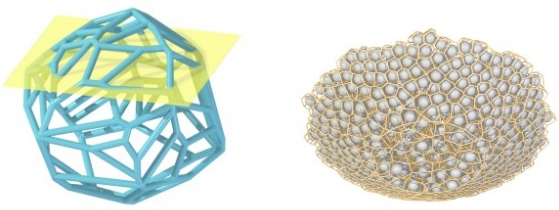

Figure 3.2: Computing Voronoi cell - (left) Irregular polyhedron cut by a plane, (right) An example result of the Voronoi cell algorithm, looking up at particles falling out of a conical funnel from below.

[1]The only time this fails is if the cell and the plane are disjoint

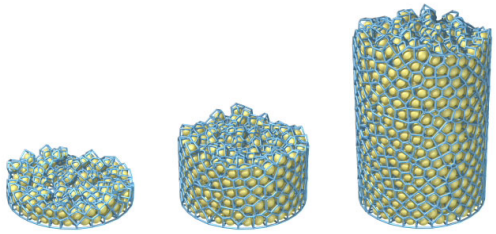

Figure 3.3: Computing Voronoi cell - Packing balls with the same radii in a cylinder

Chapter 4

Voronoi Diagram of 3D Balls Library

This section describes the tested VDS library and contains a short example showing how to compute the Voronoi diagram of 3D balls with concurrent explanation.

4.1 M. Maňák's Voronoi Diagram of Spheres Library

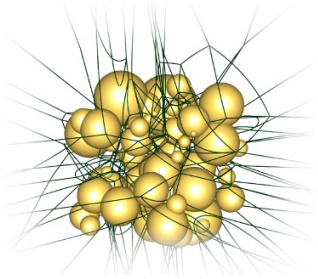

Figure 4.1: VDs library - The output diagram of a given set of balls, taken from VDS library documentation[2].

The author implemented the library, which constructs the Voronoi diagram of 3D balls as a single connected component, thus the diagram can be incomplete for some kind of data. Usually it is caused by the data, where the difference in the radii of balls is great. The example of the Voronoi diagram is illustrated in Figure 4.1.

The created diagram does not represent elliptic edges without any vertices. The library does not solve the concept of infinite vertex, i.e infinite edges have only one reference to the end vertex. That is why it is suggested to enclose the input set by four more balls, sufficiently distant from the input set, which forms an outer tetrahedron. This construction allows to avoid problems with edges going to infinity. The library has the following assumptions:

- there does not exist a sphere fully contained in another sphere
- every face is defined by exactly two spheres
- every edge is defined by exactly three spheres
- every vertex is defined by exactly four spheres

The library uses the edge-tracing algorithm, where the input is a set of generators and the output is a diagram. Diagram represents only Voronoi vertices and edges. Each vertex stores indices to its four defining generators and the position of a Voronoi vertex. Each edge contains references to its two end vertices and stores indices to its three defining generators.

The brute force edge tracing algorithm has the quadratic execution time. That is why M. Maňák developed a new approach for fast discovery of Voronoi Vertices with using a Delaunay triangulation of ball centers and presents a definition of *a feasible region*. The full algorithm description can be found in [2]. This approach works well when the maximal difference in balls radii is small.

The library is implemented in C♯ programming language as a black box object. To use this library it is required to set references to it.

4.1.1 Example

Before we start a discussion about a Voronoi diagram construction let us take a look at the balls data structure, see Code 4.1.

Code 4.1: Generator structure

```
1 Generator (double x, double y, double z, double radius)
2 Generator (Vector3d center, double radius)
```

where x, y, z are the coordinates, r is the radius and *ceter* is the center of the sphere.

At the beginning it is necessary to initialize some auxiliary, see Code 4.2

Code 4.2: The multiplication factor and list of balls

```
1 double factor = 4.0;
2 List<Generator> generators = new List<Generator>();
```

Code 4.2 represent the multiplication $factor$ for the construction of the outer tetrahedron and initialize the list of balls.

Code 4.3: Adding an outer tetrahedron.

```
1 Generator[] ouTetrahedron;
2 Generator.ComputeOuterTetrahedron(generators, factor, out
     ouTetrahedron);
3 generators.AddRange(outerTetra);
```

Code 4.3 illustrates how to add an outer tetrahedron to the input set of spheres to avoid the issue of infinity edges. Simply, the balls are enclosed into a bounding sphere, its radius is multiplied by a $factor$, and then this sphere is inscribed to the sides of the outer tetrahedron.

Code 4.4: Edge-tracing.

```
1 EdgeTracing algo = new EdgeTracing();
```

The next step is to create an object which implements the *edge-tracing* algorithm for a diagram construction, see Code 4.4.

The library allows to use two strategies of Vertex search, Code 4.5(line 1) suggests the slow quadratic algorithm and Code 4.5(line 3) performs the significantly faster Delaunay vertex search. Lines 5 set up the input balls, line 6 performs the diagram construction and line 7 get the result.

Code 4.5: Searching the Voronoi vertex

```
1 algo.VertexSearch = new DefaultVertexSearch();
2
3 algo.VertexSearch = new FeasibleRegionVertexSearch();
4
5 algo.DefineGenerators(generators);
6 algo.ConstructDiagram();
7 ETDiagram diagram = algo.ReleaseDiagram();
```

Optionally we can compute the geometry of Voronoi edges. A curve is represented as a Bézier quadratic curve, see Code 4.6.

Code 4.6: Computing the geometry of Voronoi edges

```
1 Factory geometryFactory = new Factory(diagram);
2 foreach (ETEdge edge in diagram.Edges)
3 edge.Curve = geometryFactory.ProduceGeometry(edge);
```

4.1.2 Other properties of VDS

The constructed diagram contains the information about generators, vertices, edges, facets and regions. To let these data call the following methods, see Code 4.7.

Code 4.7: ETDiagram properties

```
1 diagram.Generators  - return a set of generators
2 diagram.Vertices    - return an array of Voronoi vertices
3 diagram.Edges       - return an array of edges
4 diagram.Faces       - return an array of faces
5 diagram.Regions     - return an array of regions
```

Vertices of the diagram are stored in *ETVertex* structure, it defines the topology and the geometry in the edge tracing algorithm, see Code 4.8 (*position* stores the position coordinates, and *G*4 creates a group of indices of generators which define the Voronoi vertex).

Code 4.8: ETVertex structure

```
1 ETVertex (G4 gens, Vector3d position)
2 ETVertex (G4 gens)
3 ETVertex ()
```

The other very important structure is *ETEdge*, see Code 4.10 (*startVertex* and *endVertex* define the edge).

Code 4.9: ETEdge structure

```
1 ETEdge (ETVertex startVertex, ETVertex endVertex)
```

The ETEdge support the following methods, showed in Code 4.10

Code 4.10: ETEdge methods

```
1 edge.StartVertex          - return ETVertex that starts the edge
2 edge.EndVertex            - return ETVertex that finishes the edge
3 edge.QuadraticBezierCurve - return rational quadratic Bezier curve
4 edge.G3                   - return three gens defining the edge
5 edge.IncidentFaces        - return incident face
```

ETFace contains references to child edges (Code 4.11, line 1), to its two generators defining the face (Code 4.11, line 2) and its incident regions (Code 4.11, line 3).

Code 4.11: ETFace structure

```
1 face.Edges                       - return a set of edges
2 face.G2                          - return gens defining the face
3 face.IncidentRegions - return incident regions
```

In Code 4.11 (line 2) $G2$ contains indices of generators defining the face.

ETRegion: A region is defined by exactly one generator (Code 4.12, line 1). It has references to child faces (Code 4.12, line 2).

Code 4.12: ETRegion structure

```
1 region.G1    - return gens defining the region
2 region.Faces - return faces
```

The VDS library allows to construct the topology over edges (Code 4.13). The method runs in $O(n)$ time.

Code 4.13: Construction of a higher topology

```
1 diagram.CreateHigherTopology();
```

One of the most interesting method in the library is a neighbours search (Code 4.14), *genId* is a specified index of a generator and $foundGens$ is a list of indices of neighbouring balls.

Code 4.14: Neighbours search

```
1 diagram.FindNeighbors(int genId, ICollection<int> foundGens)
```

Since there is no higher topology than vertices and edges, this search runs in $O(M)$ time, where M is the number of all edges in the diagram. Note than M can be even $O(N^2)$.

Table 4.1 illustrates the comparison results of the nearest neighbour search by using a brute force algorithm and a Voronoi diagram of spheres (t[s] time in seconds). All results were obtained on Intel(R) Core(TM) i5 CPU, 240GHz, 4GB RAM, C♯.

count of balls	Brute force t[s]	VDS t[s]
4086	0.002	16
8171	0.005	49.73
16342	0.005	91.55

Table 4.1: Brute force versus Voronoi Diagram of spheres in nearest neighbour search.

Chapter 5

Smoothed Particle Hydrodynamics

In everyday life we usually do not think about the complexity of ordinary fluids (liquids, gases) behaviour, for example, ocean waves, turbulence, rising smoke. Those phenomena seem to be ordinary, but they represent a tough computational problem. Since *Smoothed Particle Hydrodynamics* used to solve this problem, it was decided to use an external library for my diploma thesis. I would like to thank Věra Skorkovská for her cooperation and help with the choosing of a SPH library. After research and studying the properties of different libraries the sph library from R.C. Hoetzlein[3] was chosen. The author suggests an easy-to-use open source *SPH* (Fluids v.2) library implemented in C++ programming language.

5.1 FLUIDS v.2

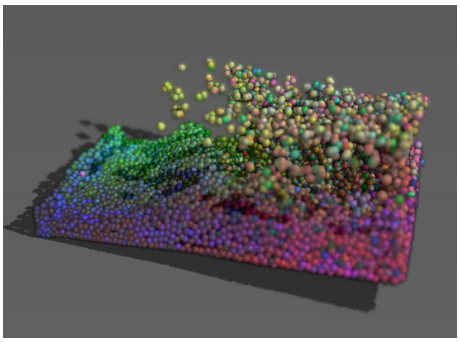

Figure 5.1: FLUIDS v.2 - A Fast, Open Source, Fluid Simulator - A fluid simulation screenshot

The library uses a *Smoothed Particle Hydrodynamics* for achieving a real time effect. The *SPH* method utilizes a novel and innovative form of local field approximation to differentials and functions making it possible to solve a differential field equation using an intuitive and extensible particle method. The core of the method is to use a smoothing kernel, i.e. function with certain properties, which are used to approximate value of the field function from the surrounding particles. The *SPH* method is also known as a *mesh - free* method. *Mesh-free* methods are generally growing in popularity since they are both computationally and numerically efficient and also easy to implement and use.

The source code itself was not aimed to be used as an external library, it was targeted on the users who want to expand or rewrite the code, see the original scene generated by Fluids v.2 in Figure 5.1. The application also suggests two versions of implementations, CPU and GPU (tested with Visual Studio 2008), however, we used only a CPU version since the VDS library was not oriented for dynamic computations. This library is available for Windows and Linux users. Because the tested VDS was written in a C♯ programming language, it was decided to convert the source code of the SPH into a dynamic library.

The SPH provides some basic examples, which could be in future modified as necessary. It is possible to set parameters for a required fluid motion, e.g., gravitation direction, bounds, viscosity, particle mass, rest density, emit position and its angle, smooth radius, etc. The SPH itself contains its own tool for a scene visualization, however, it was not used in my C♯ application. Basically, the SPH computes the whole particle motion: acceleration, control velocity limiting and boundary condition.

The full approach description can be found in[**?**],
http://www.rchoetzlein.com/eng/graphics/fluids.htm.

5.1.1 Example

The communication between sph and main application is provided by the code showed in Code 5.1.

Code 5.1: Export and import of SPH

```
extern "C"__declspec(dllexport) method() // for export from c++

[DllImport("CppDll.dll")]  // access to dll from c#
public static extern Method();
```

Initialization of SPH starts with the method, see Code 5.2.

Code 5.2: Initialization

```
C_Initialize(BFLUID, nmax);
```

BFLUID is a constant, which defines the type of particle or fluid motion and *nmax* defines the number of particles (balls) in fluid. C_Initialize() calls SPH_Setup() in c++ sph library with the following parameters, see Code 5.3.

Code 5.3: Initialization parameters

```
m_Param [ SPH_SIMSCALE ] = 0.004; // unit size
m_Param [ SPH_VISC ] = 0.0001; // pascal-second (Pa.s) = 1 kg m^-1 s
    ^-1
m_Param [ SPH_RESTDENSITY ] = 600.0; // kg / m^3
m_Param [ SPH_PMASS ] = 0.205; // kg
m_Param [ SPH_PRADIUS ] = 0.04; // m
m_Param [ SPH_PDIST ] = 0.0059; // m
m_Param [ SPH_SMOOTHRADIUS ] = 0.01; // m
```

The second method which has to be called is showed in Code 5.4

Code 5.4: Create SPH with nmax numbers of particles (balls).

```
CreateSph(nmax);
```

Code 5.4 is responsible for computing a smoothing kernel, grid setup and inserting particles into fluid. After all initialization procedures were performed, the fluid motion can be computed, this is performed by methods in Code 5.5.

Code 5.5: Pointer on fluid data

```
G_GetStart(); // hold pointer to fluid data
C_Run(); // provides the main computation of a fluid motion(insert
    particels, force grid, pressure grid, calculate acceleration, new
    position of a particle, test boundary conditions).
```

Since the SPH has their own structures for a fluid, it was necessary to implement them in the main application, see Code 5.6.

Code 5.6: Fluid and GeomBuf structures

```
[StructLayout(LayoutKind.Sequential)] //Fluid
        public struct {
            public Vector3 pos;
            public uint clr;
            public int next;
            public Vector3 vel;
            public Vector3 vel_eval;
            public short age;
            public float pressure;
            public float density;
            public Vector3 sph_force;
            public (Vector3 pos, uint clr, int next, Vector3 vel,
            Vector3 vel_eval, short age, float pressure, float
                density,
            Vector3 sph_force)
            {
                this.pos = pos;
                this.clr = clr;
                this.next = next;
                this.vel = vel;
                this.vel_eval = vel_eval;
                this.age = age;
                this.pressure = pressure;
                this.density = density;
                this.sph_force = sph_force;
            }
       }

[StructLayout(LayoutKind.Sequential)] //GeomBuf
        public unsafe class
        {
            public byte dtype;
            public uint num;
            public uint max;
            public int size;
            public ushort stride;
            public sbyte* data;
            public GeomBuf()
            {
```

```
39                  dtype = 0;
40                  num = 0;
41                  max = 0;
42                  stride = 0;
43                  data = (sbyte*)0;
44              }
45          }
```

Fluid stores fluid data, and *GeomBuf* is responsible for accessing to fluid data, this operation is showed below, see Code 5.7.

Code 5.7: Example of accessing to fluid

```
1  GeomBuf mBuf = bufWrapper(0); // wrapper: pointer to structure
2  int count = C_NumPoints();  // return number of particles
3  sbyte* end = mBuf.data + count * mBuf.stride;
4  sbyte* n;
5  for (n = mBuf.data; n < end; n += mBuf.stride) //go through each
       particle in fluid
6  {
7          ...
8          f = (Fluid*)n;
9          f->pos;
10         f->vel;
11         //TODO
12         ...
13 }
```

Chapter 6

Granular models

6.1 Sphere Packing

Packing problems are a class of optimization problems in mathematics or computer graphics, which involves attempting to pack objects together (usually inside a container), as densely as it is possible. Generally, Voronoi-Delaunay approach is applied to models with an open boundary conditions, e.g., molecules. However, in most physical problems the boundary plays the determinative role.

This chapter proposes to use the Voronoi diagram of 3D balls in disordered packing of 3D balls (of different and equal radii) in a container, since the Voronoi network is a helpful tool for analysis of voids, empty spaces between spherical particles, where the Voronoi diagram plays the role of a navigation map.

This approach is dealing with packing of spheres in models like a cube, a cylinder and the objects constructed from a set of triangles. This section will describe the algorithms used in a sphere packing problem.

I often leave out the word "Voronoi" from the terms, e.g., I will use "diagram", "edge", "vertex" instead of "Voronoi diagram", "Voronoi edge", "Voronoi vertex", respectively.

6.1.1 The outline of packing spheres in a container

Let us denote $g_i = \{g_1, g_2, ...\}$ a set of generators (spheres or balls), $v_i = \{v_1, v_2, ...\}$ Voronoi vertices, $e_i = \{e_1, e_2, ...\}$ Voronoi edges and $c_i = \{c_1, c_2, ...\}$ a set of *candidates*, a set of *empty spheres*, which could be inserted into a container.

The goal of my thesis is to pack different containers by spheres using the Voronoi Diagram of 3D Balls Library, see Chapter 4. The Voronoi diagram (constructed by Voronoi Diagram of 3D Balls Library) plays the role of the map of insertions of new balls. The algorithm uses the idea of *empty sphere* with a center in a *Voronoi vertex*. The balls are inserted one by one and a new diagram is constructed. If the diagram was not built, the ball is rejected because we need to have a valid diagram for detecting other voids for inserting new balls.

Figure 6.1 (left) illustrates the principle of using the vertices as potential spots for insertions, where $g_1 - g_7$ are generators, $v_1 - v_7$ are the Voronoi vertices, $c_1 - c_7$ are candidates, i.e., possible positions for inserting of new spheres. Figure 6.1 (right)

illustrates the use of vertices and edges of the diagram, $e_1 - e_5$ are middle points of edges and $c_1 - c_3$ are candidates.

As the edges of Voronoi diagrams of spheres could be presented as a Bézier curve, defined by three control points, we can use the definition of a quadratic Bézier curve to find a point on a curve the center of which is the center of an empty sphere. A quadratic Bézier curve is the path traced by the function $B(w)$,

$$B(w) = (1-w)^2 P_0 + 2(1-w)wP_1 + w^2 P_2, \quad w \in [0,1] \tag{6.1}$$

where P_0, P_1, P_2 are control points and w is a weight of a ball.

This method generates much more candidates than using only vertices.

The radius of the empty sphere is calculated with respect to the nearby balls, defining a Voronoi vertex/edge. Since the intersections between spheres are prohibited, the resulting radius will be the distance from the center of a Voronoi vertex or point on a Voronoi edge to the center of the generator minus its radius.

$$r_{cadidate} = \sqrt{(x_v - x_g)^2 + (y_v - y_g)^2 + (z_v - z_g)^2} - r_g \tag{6.2}$$

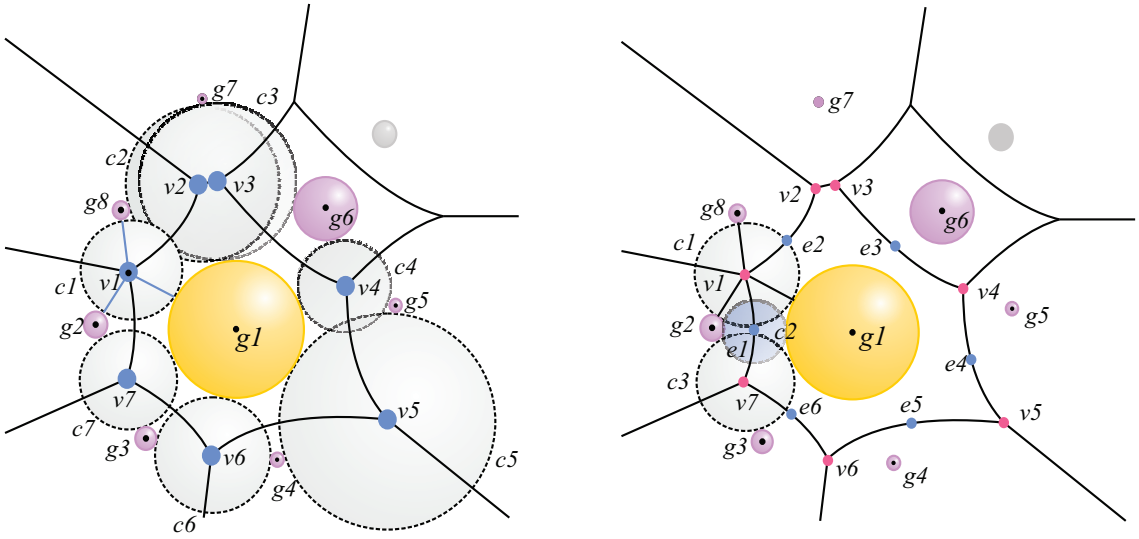

Figure 6.1: Illustration of candidates to insert into the object - 2D diagram: on the left side is the scheme of using Voronoi vertices, on the right side is the example of using Voronoi edges, where we denote a point on a curve as a possible position of empty spheres.

Packing spheres makes sense when we are packing them into containers, e.g. box, cylinder, triangle mesh models. Thus, we have to perform additional computations for boundaries, one of the solution of a packing of spheres in a cylinder was proposed by M. Gavrilova[1], described in a previous section.

The approach of this thesis uses the Voronoi Diagram of 3D Balls Library which constructs the diagram, bounded by four generators, which create a tetrahedron. This option allows us to avoid the problems with infinities in edges. Hence, the Voronoi

diagram is not strictly constructed inside a container, the edges and vertices could intersect the container.

After the radius $r_{candidate}$ has been found, we calculate the distance from the Voronoi vertex to bounds, then the smallest distance is picked and used as a radius of a new sphere. This operation processes all vertices which are inside the container, other vertices are not taken into consideration, see Figure 6.2 (left), on the left side the 2D slice of a cylinder is illustrated, where only Voronoi vertices are used, v_8 and v_9 does not satisfy the condition of bounds, thus they are not used as possible positions of new spheres. Figure 6.2 (right) illustrates the cube, where the edges and vertices are used as potential spots for inserting new spheres, but again vertices v_8, v_9 and points on edges $e_8 - e_{16}$ do not satisfy the boundary conditions, thus they are not in a set of candidates.

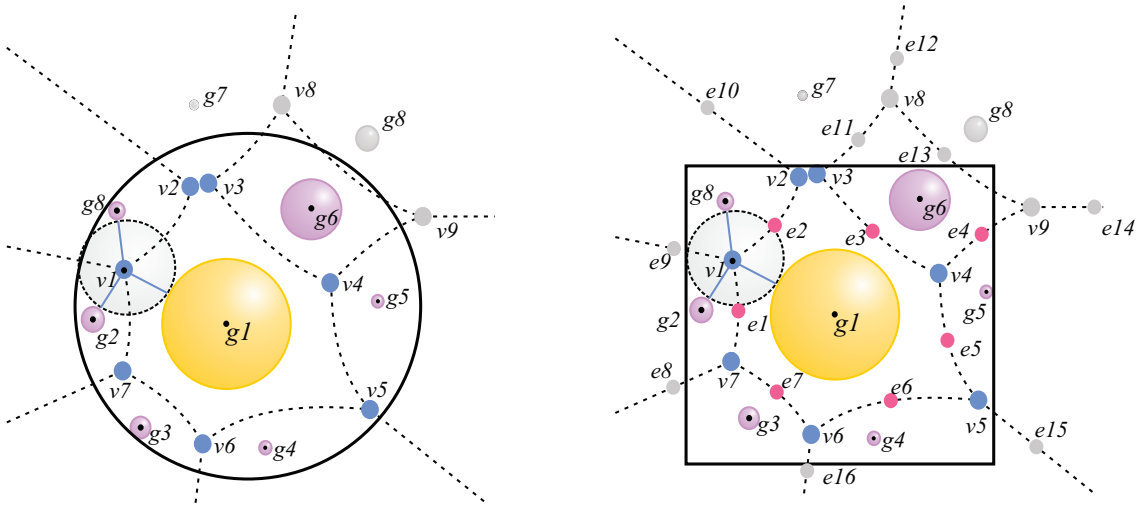

Figure 6.2: Packing into a container - 2D scheme, from left to right: cylinder, cube

If the order of inserts is important (in case of different balls packing), i.e., large spheres have to be inserted first and the smallest last, then it is required to sort the *candidates* with the radii consideration. In case we are packing a container with equal spheres, the calculations of radius are unnecessary since the size of sphere is already defined.

To perform the packing, we need an initial Voronoi diagram, which provides us with start vertices or points on edges as new positions for *candidates*. Cube and cylinder have automatically calculated generators, which define the start of the Voronoi diagram, see Figure 6.3.

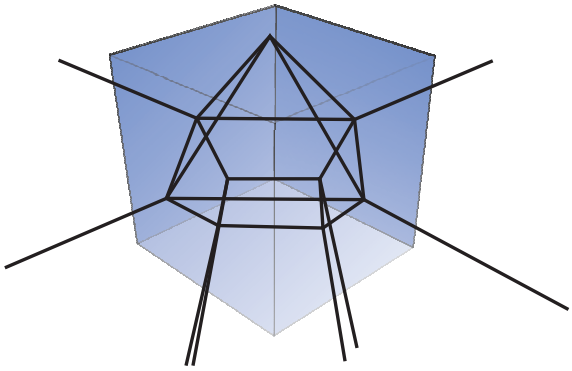

Figure 6.3: An initial diagram - Cube example

Algorithm 6.1.1 and Algorithm 6.7 illustrates the pseudocode of packing containers by equal and different balls respectively.

Algorithm 6.1.1 Equal radii

Input: a geometrical model (cube, cylinder)

Output: a container packed by equal balls, the Voronoi diagram of an expanded set of balls

- Define start generators for a given model (for construction the start diagram).
- Initialize the start diagram and find all Voronoi vertices v_i inside a container.

while count of generators changed **do**

- Find new candidates c

Process each candidate c_i (a vertex v_i with a given radius r).

for each c_i **do** Insert c_i into a set of generators and construct the diagram.

if diagram is not constructed **then** c_i is rejected (see Chapter 2).

else

if there is any intersection with other balls from the set g **then** the candidate c_i is rejected.

else it is accepted.

end if

end if

end for

end while

Algorithm 6.1.2 Different radii

Input: a geometrical model (cube, cylinder)

Output: a container packed by different balls, the Voronoi diagram of spheres

• Define start generators for a given model (for construction the start diagram).

• Initialize the start diagram and find all Voronoi vertices v_i inside a container.

while count of generators changed **do**

• Find new candidates c

for each c_i **do**

• Calculate the radii ($r_i, i \in \{1, 2, ...\}$) with respect to defining generators and boundaries. Filter by radius (in case if we defined the smallest and largest balls).

end for

• Sort candidates with respect to the radii sizes (from larger to smaller).

for each c_i **do** Insert c_i into a set of generators and construct the diagram.

if diagram is not constructed **then** c_i is rejected (see Chapter 2).

else

if there is any intersection with other balls from the set g **then** the candidate c_i is rejected.

else it is accepted.

end if

end if

end for

end while

Triangle mesh model

Packing spheres into triangle mesh objects requires more precise definition of initial generators. If the edges or vertices are constructed outside our model they do not have a value for us. These points are rejected in the preprocessing part (tests if points are inside a model). For more tangled objects (or objects with a hole inside) is preferably to define the start Voronoi diagram with a larger number of generators or have a uniform distribution of generators. This helps to avoid a poor filling performance.

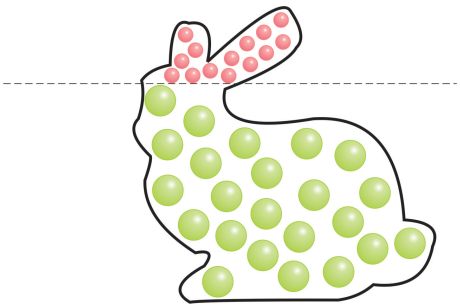

Figure 6.4: Bunny partition - Divide the bunny into two parts: the top part is more complex, it contains more initial generators, the bottom part is simpler and there is no need to have it so precise.

A model of the Stanford Bunny is a good example, it consists of a pretty bulky body, a head and small long ears. Filling of the main body and the head is not complicated, but the ears involve more attention. Using a uniform distribution of start generators leads to a situation, where the large number of existing vertices and edges are out of the model. The details with higher complexity cannot be filled properly. To avoid this problem of ”empty ears” in a given example, I suggest to divide the model into several parts with respect to its complexity. More complicated parts (ears) should be defined by more initial generators, see Figure 6.4. Why more generators? The idea is similar to building the medial axes inside the object (see Figure 6.5), to achieve as much vertices inside a mesh as possible.

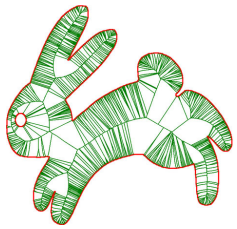

Figure 6.5: 2D Voronoi based medial axis - taken from [18]

Since the construction of the Voronoi diagram of spheres is not adapted to a dynamic computation, I denote the parts inclined to be troublesome in packing problems and define the start generators more precise. The rest of the object which can be formed easily by

a smaller number of start generators and does not need pointless increasing of number of initial balls, because it leads to a slower computation, see Figure 6.6.

In that case the initial generators can be chosen up from the set of vertices, which forms the triangle mesh, e.g., every N*th* vertex for ears and every M*th* for the body are chosen. An even distribution of initial generators to partial components (e.g., an ear) results in better packing.

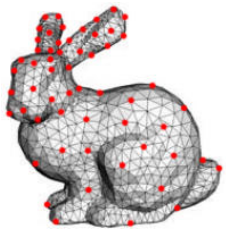

Figure 6.6: Bunny mesh - Bunny mesh with initial generators

The radius of the empty sphere is calculated with respect to its defining generators. Boundaries in that case are more complicated, since it is represented as a set of triangle faces. The ray-triangle intersection test is performed. When the intersection point of a ray and a triangle mesh is found, then the distance from Voronoi vertex to intersection point is tested. In case the distance is smaller than the previous smallest radius, then it is is saved as a new radius. The ray is moving around X, Y and Z axes with a given angle.

A Voronoi edge is defined by three generators and start(end) vertices are defined by four generators. Using this property of VDS reduces the computation of radii, since there is no need to control the whole set of spheres.

It is expected that packing containers by different balls will perform faster, since the empty space is filled by the sphere with the maximal radius which could be inscribed. Inserting larger balls will not greatly increase the number of candidates as it happens with equal spheres.

6.2 Granular simulation

Granular animations are very popular in computer graphics. Physically-based simulation algorithms are usually preferred over manually modelling behaviour. Granular interactions with other materials or with itself can be very complex.

In my case I used both libraries *Voronoi Diagram of 3D Balls* and *SPH*. SPH library provides simple boundaries, but additional methods were implemented to simulate a bounce from cube walls, cylinder coat and bounce from a nearby granule or triangle face.

6.2.1 Proposed Application of VDS in Granular Simulation

The motion of particles is based on fluid forces. The *SPH (Fluid v.2)* supports only a simple bounding box to limit the particle flow. In this approach the particle movement is performed in different bounding models, that is why the code had to be modified for correct interaction of faces of models with the particles. The motion is effected by forces of gravity and by faces with respect to a particle radius.

The *FLUID v.2* uses a uniform grid spatial division to speed up the calculations of the particle forces. The size is given a double value of the smoothing radius of a particle, thus the grid resolution is calculated with respect to the size of the scene. This work for particles of the same radii size.

The use of the VDS simplifies the neighbours search, since the diagram stores information about vertices, edges and generators. The drawback of using the VDS is that sometimes it may cause an error in motion. The ball can move only in case the diagram was successfully constructed (it happens in a rare cases, see Chapter 2), otherwise the ball stays on a previous place, thus there are situations, when one ball is suddenly frozen, while others continue in motion, observing such an event looks really odd.

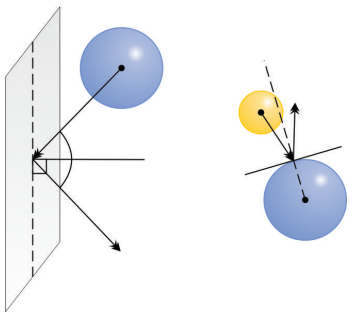

Figure 6.7: Illustration of balls behaviour - Bouncing

At the beginning all granules have the same direction of motion (along the *Oy* axis), but during the move the direction could change. That happens when the ball collide with other balls or with the boundaries of the model. Using the law of reflection the angle of impact is equal to the angle of rebound, see Figure 6.7.

The diagram could be constructed several times during one iteration. The application saves the last version of the diagram for the situations when the diagram was not successfully constructed. This little option saves the time, when the diagram is defined by a large number of generators.

6.2.2 Plane, Cylinder

Fluid simulation on a plane, the simplest one due to absence of boundaries except the plane on which the ball set is falling on. Let us assume $b_i = \{b_1, b_2, ...\}$ as a set of balls in a fluid. Using the VDS library the appropriate diagram for a given set of balls b_i is constructed. The particle motion is performed by SPH (Fluid v.2) library. The VDS helps to find neighbours and calculate a new position of a particle in case of balls collision.

Algorithm 6.2.1, Algorithm 6.2.2 illustrates the pseudocode of particles motion with respect to the plane p and a container, respectively.

Algorithm 6.2.1 Plane

Input: a set of particles (balls), cylinder

Output: new positions of particles in space with respect to a given plane

for each $b_i \in$ Fluid **do**

if b_i is above the plane p **then**

• get the new position of a ball (calculated by SPH, see Chapter 5, Code 5.7) and construct the diagram.

if the diagram is constructed **then** find neighbours and detect intersections

if there are intersections with neighbours **then** change the velocity direction[1].

else the new position is accepted.

end if

else the new position is rejected.

end if

else calculate the point of intersection of b_i trajectory with the given plane and change the new position to the intersection position (with respect to the particle radius).

end if

end for

Figure 6.8 illustrates motion on a plane.

[1]The value of a new direction depends on a normal vector of the intersected ball, then the vector of

Algorithm 6.2.2 Cylinder

Input: a set of particles (balls), plane

Output: new positions of particles in space with respect to a given container

for each $b_i \in$ Fluid **do**

if the line given by b_i's actual and previous positions (calculated by SPH, see Chapter 5, Code 5.7) does not intersect the container **then**

- get the actual position of a ball and construct the diagram.

if the diagram is constructed **then** find neighbours and detect intersections

if there are intersections with neighbours **then** change the velocity direction and compute new position (described below).

else the new position is accepted.

end if

else the new position is rejected.

end if

else calculate the point of intersection of b_i trajectory with the given container and change the new position to the intersection position (with respect to the particle radius) and calculate the new velocity direction.

end if

end for

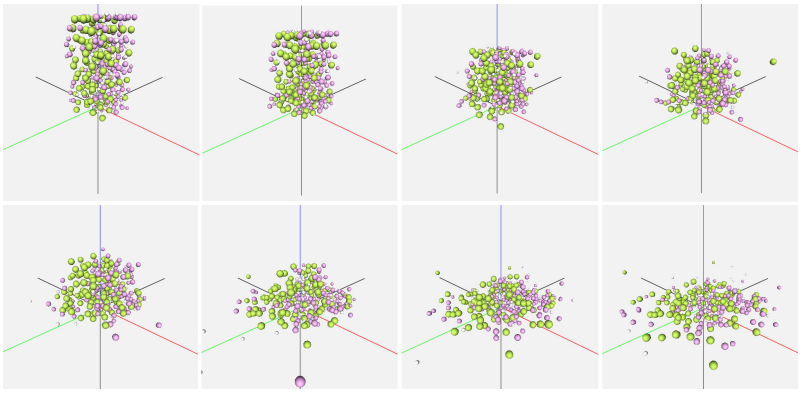

Figure 6.8: Granules falling on plane - Simulation of falling on a plane

Figures 6.9-6.10 illustrates the Algorithm 6.2.2.

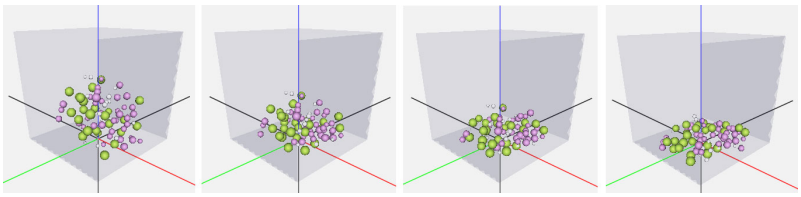

Figure 6.9: Granules motion - Simulation of falling particles in a cube

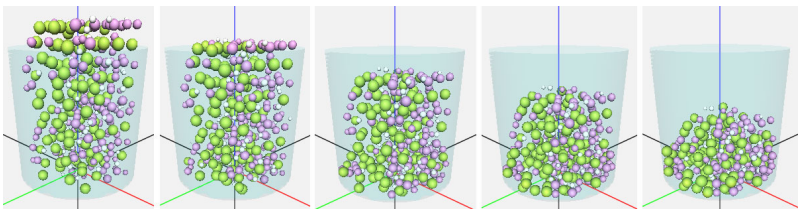

Figure 6.10: Granular motion - Simulation of falling particles in a cylinder

The idea of motion does not allow ball intersections. The SPH library assume balls with the same radii of balls, hence a code was implemented to find a point on a ball trajectory when the balls touch and inflicts the change of the velocity direction. Let us assume we have a situation similar to the scheme illustrated in Figure 6.11.

rebound is calculated using the following equation: $R = V - 2 * (V \cdot N) * N$, where R is the reflecting vector, N is the normal vector to sphere and V is the velocity vector.

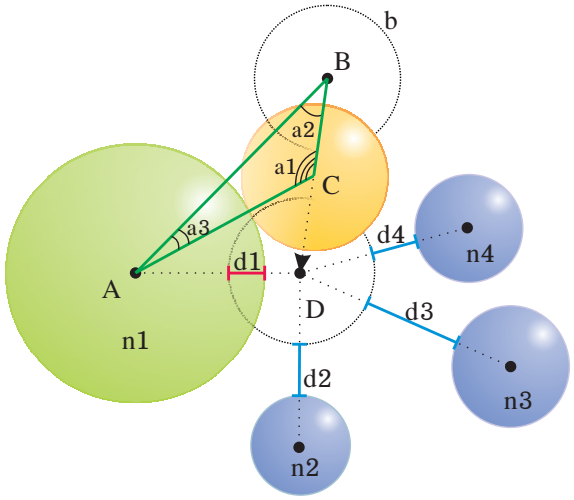

Figure 6.11: Balls intersection - Computation of a new position.

Let us suppose that a new position of a ball b is located at the point D. Before we accept a new position, we check intersections with the neighbour balls (n_1, n_2, n_3, n_4), if the distance $dist_{diff} < 0$ (Eq. 6.3) then there is an intersection between b and n_i.

$$dist_{diff} = (b_x - n_{ix})^2 + (b_y - n_{iy})^2 + (b_z - n_{iz})^2 - b_r - n_{ir} \tag{6.3}$$

Using the Sine and Cosine theorems we obtain the $|BC|$ distance. Knowing the $|BC|$ and $|BD|$ we can calculate the coordinates of C, a new correct position at which no intersection occurs.

Triangle Mesh

For testing motions in more complex objects it was suggested to use a triangle mesh since it is the most ubiquitous rendering primitives in use and most of non-trivial objects are represented by a set of triangles. Let assume we have a triangle mesh T, then check each particle p in a fluid F with respect to the triangle mesh in the following manner:

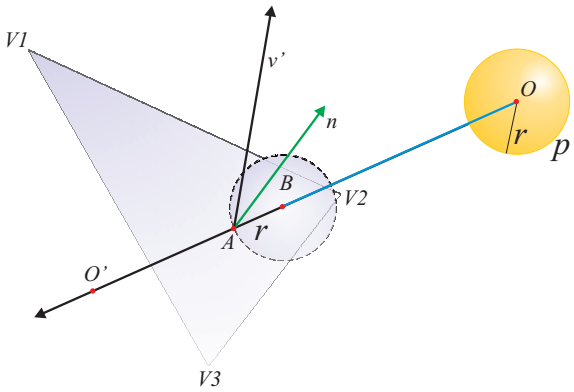

Figure 6.12: Segment triangle intersection - Computing a new position of a particle flow.

Algorithm 6.2.3 Triangle mesh

Input: a set of particles (balls), a triangle mesh

Output: new positions of particles in space with respect to a given triangle mesh

• Check the line going through the positions (O, O') of a particle, and test for intersection of this line with a triangle $t_i(V_1, V_2, V_3) \in T$.

if there is no intersection between a segment OO' and a triangle t_i **then** then construct a new Voronoi diagram with a new position at O'.

if the diagram is not constructed **then** the new position is rejected (the diagram was not changed, because the point stayed at the old position).

else check intersections between nearby balls

if there are intersections **then** recalculate the new position and the velocity direction and check the validity of a newly constructed diagram again, if it fails then use the previous position, otherwise the new position is accepted.

else the new position is accepted.

end if

end if

else

• get a point of intersection A and calculate a new position with the respect to particle radius r since our particle is not degenerated to a point. This is performed by moving the center of a particle along OA and stops at B, where the distance $|AB| = r$.

• Construct the diagram

if the diagram was constructed **then** and check intersections between nearby balls.

if there are intersections **then** recalculate the new position and the velocity direction and check the validity of a newly constructed diagram again, if it fails then use the previous position, otherwise the new position is accepted.

else the new position is accepted. Use the normal in the intersection point A and calculate a new velocity direction v' using the reflection low.

end if

else use the previous position (the diagram was not changed).

end if

end if

6.3 Application

The fundamental stones of the application(see Figure 6.13) are external VDS and SPH libraries, whose properties were used in the proposed solution to solve the issue of sphere packing in a container and granular simulations. As a native programming language C♯ was chosen since the tested VDS library is implemented in it. The SPH was originally implemented as an open source C++ application, hence it was changed and the appropriate interface for communication was proposed as a dynamic (*.dll) library and integrated into the main application. As a graphics platform OpenGL(+TaoFramework) was chosen, the application also uses Microsoft Windows platform.

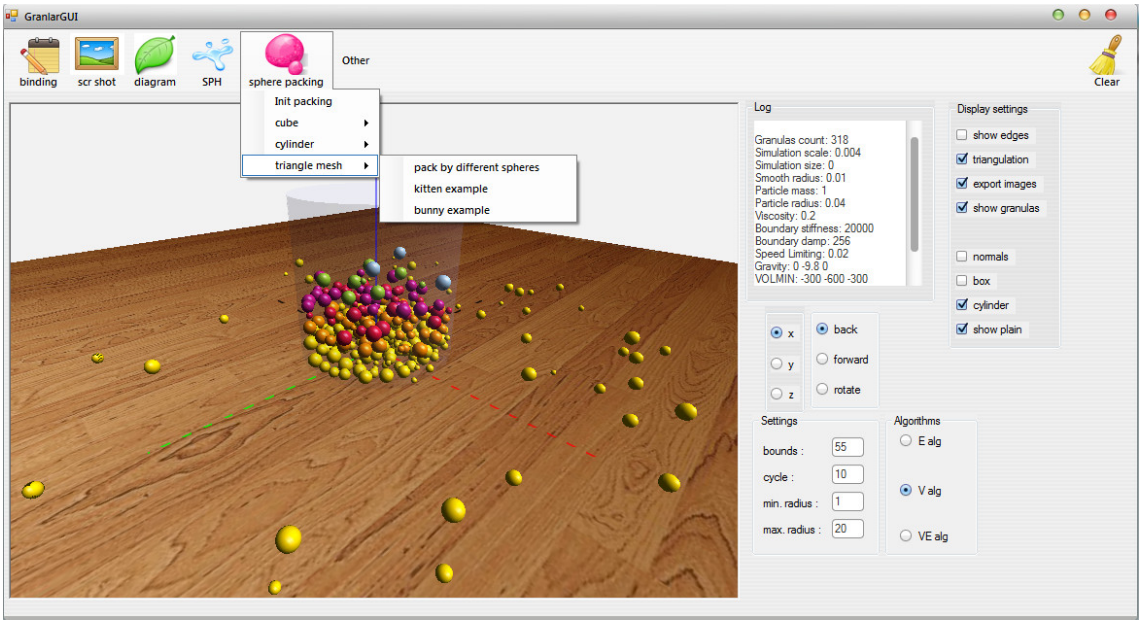

Figure 6.13: Application - screenshot

Scheme captured in Fugure 6.14 illustrates the outline of packing containers and fluid motions. Packing spheres unit performs a preprocessing before running the algorithm. Initialization depends on the type of the model to pack. The algorithm runs until there are new balls to insert into a container. Granular motion does not need a preprocessing since the number of balls stays constant.

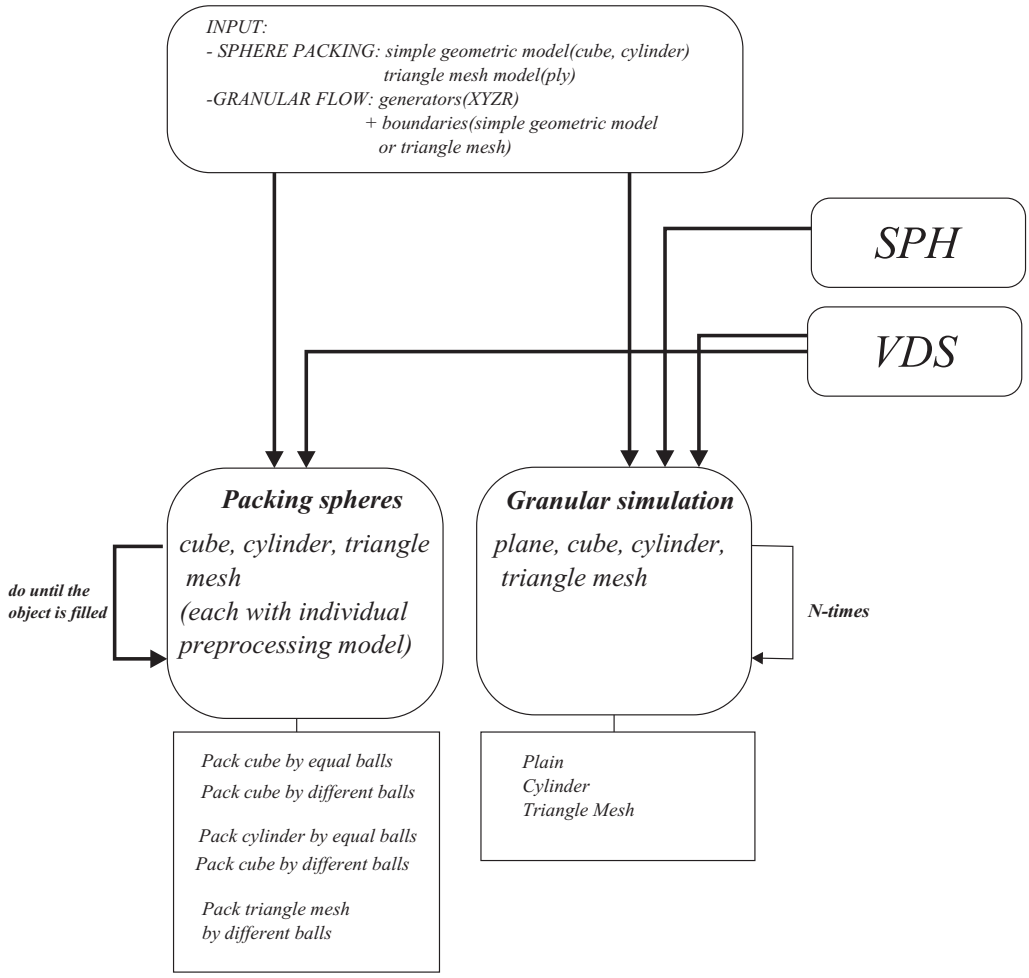

Figure 6.14: The scheme of application - The pipeline of processing of spheres

The testing application allows to read the input data in a format $*.data$[1] and the triangle mesh models in a format $*.ply$. In case of packing spheres into simple geometric containers no input is required.

However, most of the computations could not be visualized in real time, that is why it is recommended to use the option of saving of scenes (default state is ”turn on”). Other options are to turn on/off the visualization of diagram itself, balls, or the boundaries in case if we would like to have a preview of only specific elements on a scene.

The use of the main application is easy and intuitive. There were implemented some prepared examples for tests. In case of packing it is necessary to pick the model (cube/cylinder/ or import a triangle mesh), set the value of the minimal (maximal) radius, the radius of a cylinder or the size of a side of a cube. As for fluid simulation it is also necessary to import particles and set the model (plane/cylinder or import a triangle mesh). All results saves into *results* directory.

[1] $*.data$ store *XYZR* information, where x, y, z are coordinates and r is a radius.

Chapter 7

Results

The goal of my diploma thesis was to test the ”Voronoi Diagram of 3D Balls Library” and this chapter presents the results I have achieved during the process. The VDS library has been tested on two types of data, the sets of balls with the same sizes and different sizes of radii, both examples are discussed in this section.

All experimental results were obtained on an Intel(R) Core(TM) i5 CPU, 240GHz, 4GB RAM using C♯. All radii and container sizes are presented in dimensionless units.

7.1 Sphere Packing

The first idea was to use only vertices of the diagram as candidates to insert new balls, but in cases when the vertices are outside of a container it was suggested to use Voronoi edges as guidelines for new spheres, which increases the number of candidates. The drawback is that this option slows down the computation since the number of edges grows significantly. Sometimes in the cases when the difference between radii is large it becomes troublesome for Voronoi diagram construction (see *anomalies*, Chapter 2).

The diagram is generated at least N-times, where N-is the number of possible candidates. After each iteration the N value is increased since new candidates were accepted.

Let us denote algorithms with using Voronoi vertices, Voronoi edges and Voronoi vertices as V-algorithm, VE-algorithm, respectively.

7.1.1 Cube

One of the examples of application of VD is filling a cube by spheres. The packing was tested on vertices and vertices with edges. The bigger is the smallest radius the more imprecisely the object is filled, see Figure 7.1.

Figure 7.2 illustrates how many Voronoi vertices were generated in VDS (blue), how many of them were rejected in preprocessing (orange), how many candidates we actually had to test (yellow) and how many spheres with Voronoi vertices were actually accepted (green) in a given iteration.

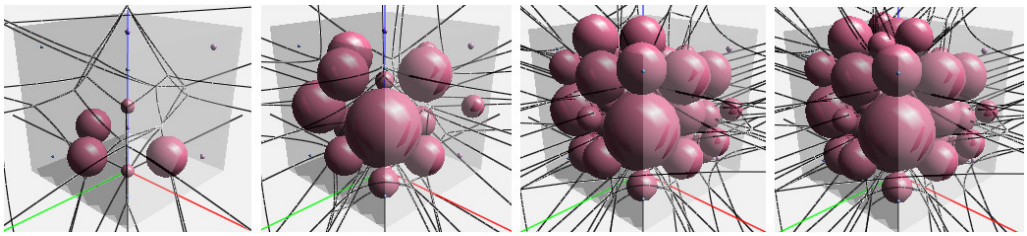

Figure 7.1: Filling the cube - The smallest allowed radius is 7. Iterations: 2, 4, 9, 12.

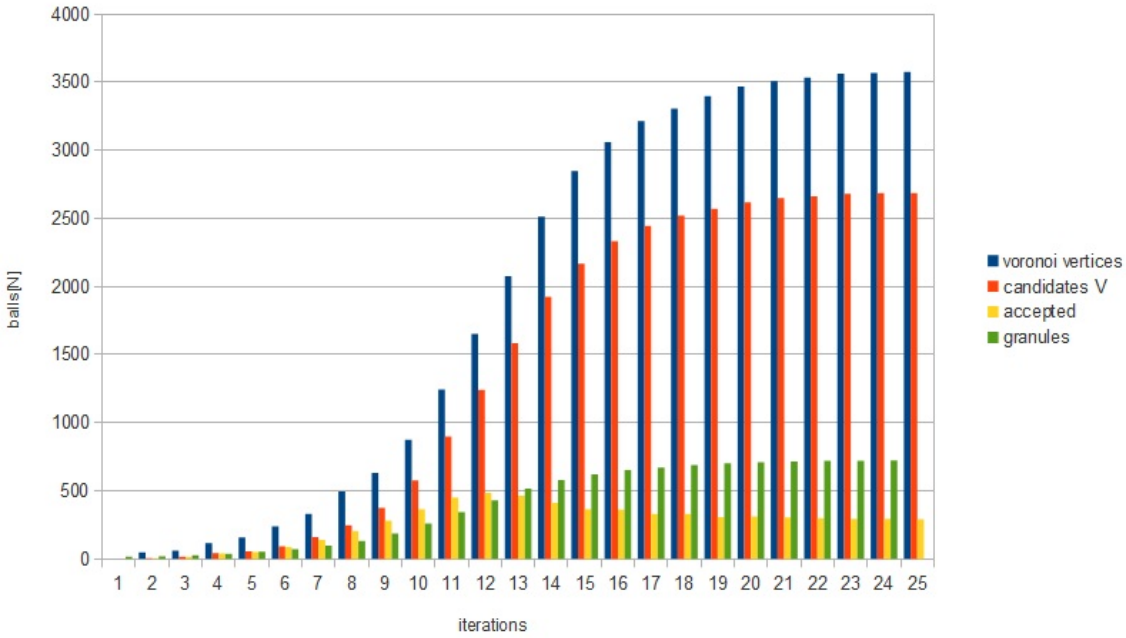

Figure 7.2: The V-algorithm diagram - The diagram of Voronoi vertices acceptance.

Figure 7.3 is a result of packing of the same balls in a cube, in this example it is tested if a Voronoi vertex center is inside a container without radius consideration. Hence if the size of the ball is small enough due to the size of a container, the packing performs great. The set of the same balls does not tend to have anomalies, thus most of candidates are accepted, but they could be rejected with regard to intersections with other balls in the set.

Figure 7.3 illustrates the process of a cube packing by identical balls (iterations: 2,4,5,6,7,8,9,11). The yellow balls show the candidates, four candidates on each edge. The red small balls represent the start generators.

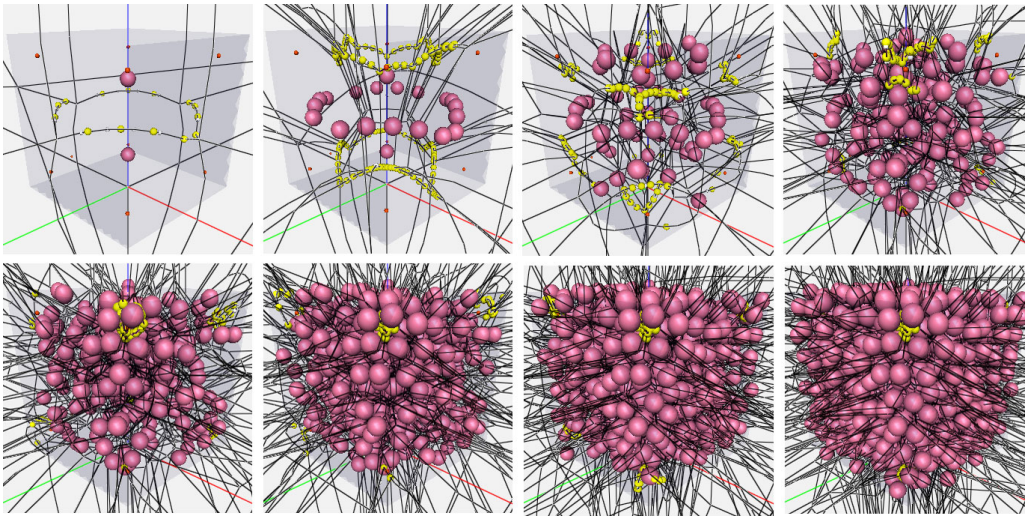

Figure 7.3: Cube with identical balls - (in lines, left to right) The radius of each ball is r = 2, the box size is 30x30x30, EV - algorithm.

Different Radii

The next paragraph analyses the V- and VE-algorithms. Using the vertices in a packing is faster, but if it is desired to achieve a higher quality, it is suggested to use edges in addition to vertices. However in some situations VE-algorithm generates less balls in a container, which can be caused by anomalies, see Chapter 2.

In Table 7.1 the results of V-algorithm packing are shown, where $ball_2$, $ball_3$ are balls with minimal radii 2, 3, respectively. The size of a cube is given as $side \times side \times side$, the time t_2, t_3 of packing cubes by balls of radii 2, 3 is presented in seconds, respectively. Cubes were packed by using a V-algorithm, it is obvious that a smaller minimal radius guarantees more densely packing, but the execution time increases significantly.

size of a cube	$ball_2$	t_2	$ball_3$	t_3
$20 \times 20 \times 20$	39	3.168	23	0.995
$30 \times 30 \times 30$	64	14.819	48	6.645
$40 \times 40 \times 40$	131	73.569	68	36.839
$50 \times 50 \times 50$	163	157.030	110	65.256
$60 \times 60 \times 60$	314	1135.712	224	417.513
$70 \times 70 \times 70$	492	4073.949	138	328.245

Table 7.1: Time execution of packing spheres into a cube, where spheres have different radii, V-algorithm

Table 7.2 contains results of using VE-algorithm (notationas are the same as in the previous table). The packing algorithm uses edges in addition to vertices. The results illustrated in Table 7.2 shows that the VE-algorithm of packing (where the minimal radius is 3) is better than V-algorithm in Table 7.1.

size of a cube	$ball_2$	t_2	$ball_3$	t_3
$20 \times 20 \times 20$	49	5.580	19	0.560
$30 \times 30 \times 30$	108	82.699	43	4.112
$40 \times 40 \times 40$	248	600.971	64	7.167
$50 \times 50 \times 50$	62	45.539	81	22.375
$60 \times 60 \times 60$	99	168.742	71	23.8623
$70 \times 70 \times 70$	463	9458.011	160	131.678

Table 7.2: Time execution of packing spheres into a cube, where spheres have different radii, VE-algorithm

Figure 7.4 illustrates V and VE algorithms. It is obvious that at the beginning the VE-algorithm is better than V-algorithm, but with increasing of volume the difference becomes negligible.

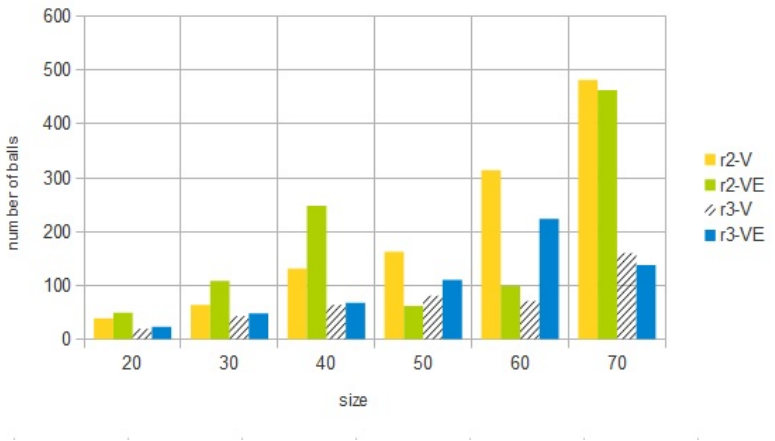

Figure 7.4: The chart of V and VE algorithms - Comparison of V with VE. $r2-V$ - a count of balls packed by using V-algorithms, where the smallest allowed radius of a ball is 2, $r3-V$ - a count of balls packed by using V-algorithm, where the smallest allowed radius of a ball is 3, $r2-VE$, $r3-VE$ - counts of balls packed by using VE-algorithm, where the smallest radii of a ball are 2, 3, respectively.

Figure 7.5 illustrates a cube packing using V-algorithm (Figure 7.5(a,c)) and VE-algorithm (Figure 7.5(b,d)). The minimal radii in both algorithms are equal, but the difference in performed results is obvious. Thus, the final result strongly depends not only on a chosen algorithm, but on ball positions in a container, which affects the construction of diagrams.

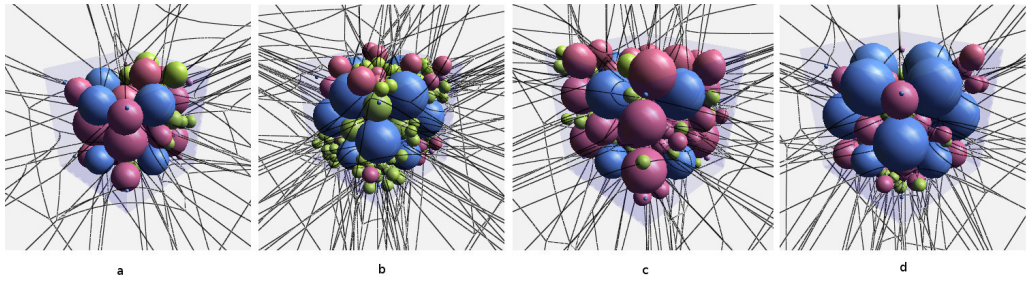

Figure 7.5: Cube packing by using V- and VE- algorithms - (a)- V-algorithm, 71 balls. (c) - V-algorithm, 160 balls. (b) - VE-algorithm, 224 balls, (d) - VE-algorithm, 138 balls. (a), (b) - the box size is 60×60×60, (c), (d) - the box size is 70×70×70. The top part of boxes is unbounded.

Equal Radii

The packing by the same balls is tested only on V-algorithm, since it meets the requirements and it is unnecessary to complicate the computations by increasing the number of candidates as equal balls do not generate anomalies, with the exception when there are more than four balls on a plane.

The following Table 7.3 shows the achieved results, it is obvious that the smaller the ball the more precise packing is, see Figure 7.6 with the ball radius = 3.

size of a box	$ball_3$	t_3	$ball_4$	t_4
$20 \times 20 \times 20$	26	1.227	19	0.514
$30 \times 30 \times 30$	63	24.43	36	3.354
$40 \times 40 \times 40$	160	437.658	63	27.556
$50 \times 50 \times 50$	339	1941.351	122	281.807
$60 \times 60 \times 60$	606	100031.398	227	1065.883

Table 7.3: Time execution of packing spheres into a box, balls of the same radii, V - algorithm. The size of a box is given as $side \times side \times side$, $balls_3$, $balls_4$ - are sets of balls, where the smallest allowed radii are 3, 4, respectively. t_3, t_4 - execution time in seconds for $ball_3$, $ball_4$.

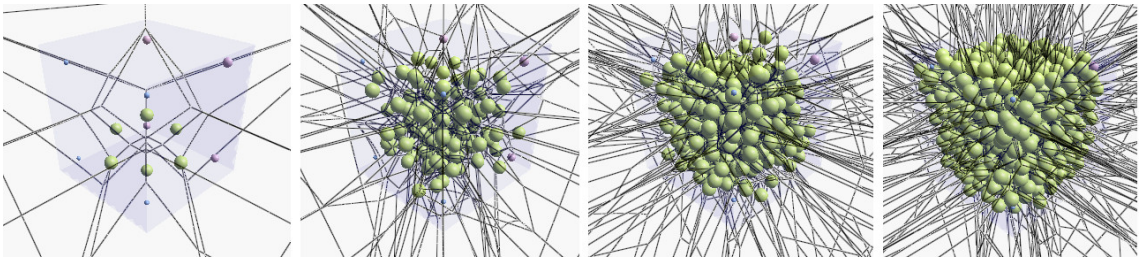

Figure 7.6: Packing spheres into a cube - Illustration of a cube packing by stages, radius of a ball is 3. In lines, from left to right, iterations: 2, 5, 7, 13.

The difference in packing by identical and different balls is the time of execution and the count of balls in a container. Tables 7.1, 7.2, 7.3 illustrates that the packing by different balls is significantly faster (the minimal radius in packing by the same balls is 3, 4 and 2, 3 in packing by different balls). The packing by equal balls inserts the balls of the same size even if the detected empty space endure a larger ball, while the void in packing by different balls is filled by the largest ball, which can be inscribed between the spheres. Hence, the diagram of different balls is less complicated, less vertices, less edges, thus, less candidates to test. The packing by identical spheres usually tends to perform good results with a right definition of a ball diameter. And last but not least, even if the count of balls in a container in case of different packing is less than the count of balls in equal packing, it does not mean that the quality is lower, see Figure 7.7.

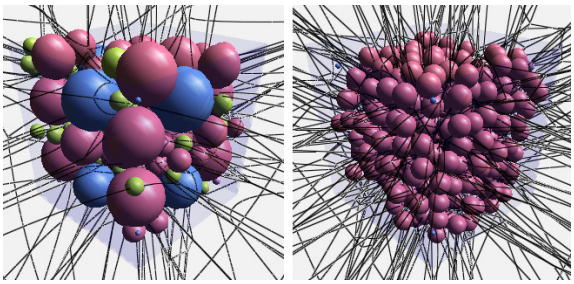

Figure 7.7: Packing spheres into a cube - in lines, from left to right, the left cube is packed by 160 different balls, the right cube is packed by 385 identical balls (with exception of initial generators (blue))

7.1.2 Cylinder

Another example is a cylinder model. Cylinder packing is very similar to cube packing, the difference is a boundary test. Figure 7.8 illustrates packing by equal balls, where the candidate centers are tested against boundaries, the results seem to be pleasing, the cylinder is fully packed.

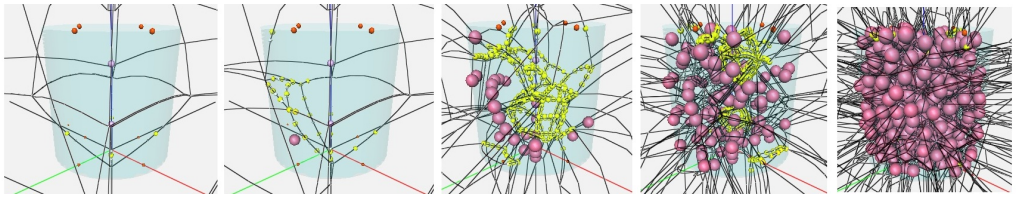

Figure 7.8: Cylinder filling - r_{ball} = 4, $r_{cylinder}$ = 40, height = 40, 853 balls. The yellow set of balls represents the candidates, the purple set of spheres represents the spherical particles.

Different Radii

The V-algorithm yields to the VE-algorithm (with the same start generators), because most of vertices in the preprocessing part were rejected. The results of tests are illustrated in Tables 7.4-7.5, where $cylinder_d$ is a diameter of a cylinder, $ball_1$, $ball_2$, $ball_3$ - are sets of balls, where the minimal allowed radii are 1, 2, 3, respectively, t_1, t_2, t_3 - is the appropriate execution time for sets of balls with $ball_1$, $ball_2$, $ball_3$ smallest radii, respectively.

The V- and VE- algorithms were also tested on a cylinder, but in this case the VE-algorithm seems to be better, see Table(7.4- 7.5). Figure 7.9 presents the diagram of the dependence of balls counts in a cylinder on the algorithm.

$cylinder_d$	$ball_1$	t_1	$ball_2$	t_2	$ball_3$	t_3
20	28	6.080	32	8.350	14	0.144
30	35	10.550	29	5.512	26	0.3.961
40	40	11.754	33	9.001	30	5.600
50	35	9.134	34	8.002	34	7.411
60	54	58.884	48	42.873	34	10.612
70	87	253.360	77	190.152	62	83.111
80	87	211.808	54	60.650	48	28.300
90	75	129.934	46	37240	38	15.107

Table 7.4: Time execution of packing spheres into a cylinder, balls of different radii, VE - algorithm

$cylinder_d$	$ball_1$	t_1	$ball_2$	t_2	$ball_3$	t_3
20	25	1.270	28	1.804	14	0.144
30	23	0.980	21	0.865	17	0.532
40	21	0.930	21	0.923	21	0.904
50	23	1.132	22	0.919	22	0.877
60	28	1.360	26	1.322	26	1.343
70	57	17.430	26	1.325	26	1.362
80	48	10.786	27	1.354	26	1.382
90	31	1.926	28	1.378	27	1.324

Table 7.5: Time execution of packing spheres into a cylinder, balls of different radii, V - algorithm

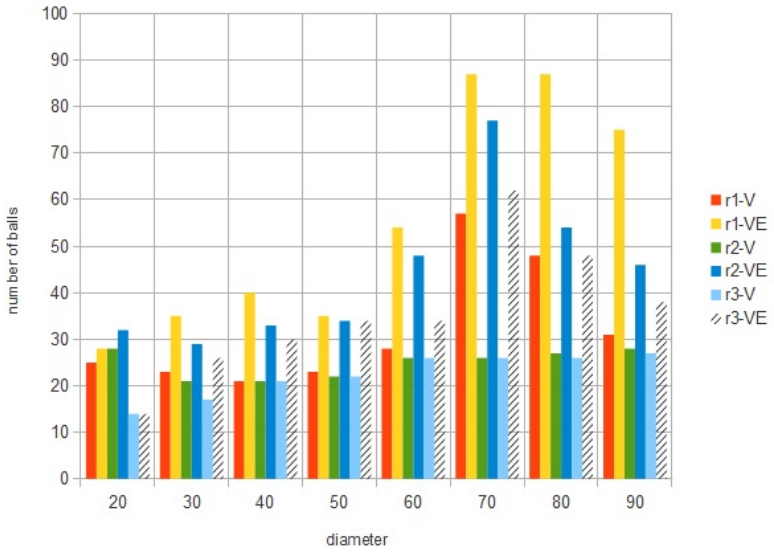

Figure 7.9: Cylinder filling - Comparison of V- and VE-algorithms. $r1-V$, $r2-V$, $r3-V$ - application of the V-algorithm on the sets of balls, where the minimal radius is 1, 2, 3, respectively. $r1-VE$, $r2-VE$, $r3-VE$ - application of the VE-algorithm on the sets of balls, where the minimal radius is 1, 2, 3, respectively.

Figure 7.9 illustrates that both algorithms have almost equal results for smaller radii of a cylinder (e.g., for diamaters 20, 30, 40, 50), but with diameter increasing the difference starts to be significant.

Figure 7.10 illustrates the VE-algorithm in packing into a cylinder with radius 30 and height 60. The cylinder is packed by 161 different balls.

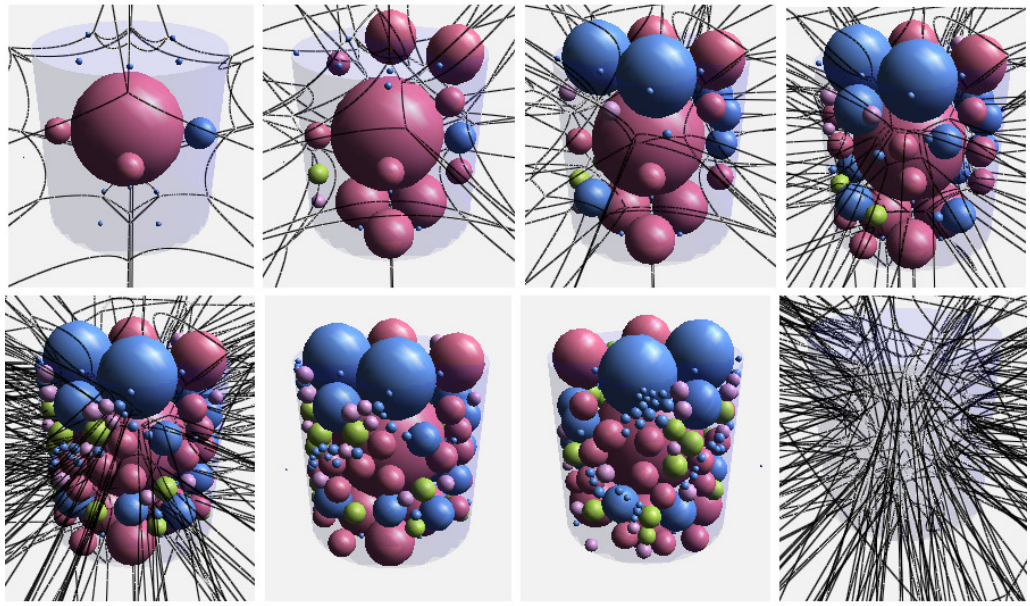

Figure 7.10: Cylinder. VE-algorithm. - In lines, from left to right, from top to bottom. iterations: 2, 4, 5, 6, 7, 7 (without diagram), 7 (the opposite side of a cylinder), 7 (the Voronoi diagram of spheres).

Equal Radii

The results of packing equal balls into a cylinder are presented in Table 7.6 ($cylinder_d$ is a diameter of a cylinder, $balls_3$, $balls_4$ are sets of balls, where the minimal radii are 3, 4, respectively. t_3, t_4 - is an appropriate execution time).

$cylinder_d$	$balls_3$	t_3	$balls_4$	t_4
20	14	0.153	14	0.152
30	64	32.251	14	0.152
40	128	171.101	68	36.864
50	123	53.425	117	156.733
60	492	6952.4936	86	18.454

Table 7.6: Time execution of packing spheres into a cylinder, balls of same radii, V - algorithm

Figure 7.11 captures the packing by identical spheres (with diameter 6), the final result is comparable to result illustrated in Figure 7.10. Both cylinders have unbounded top parts. Figure 7.11 has more densely packing than Figure 7.10 (this can be solved by defining a smaller minimal allowed radius).

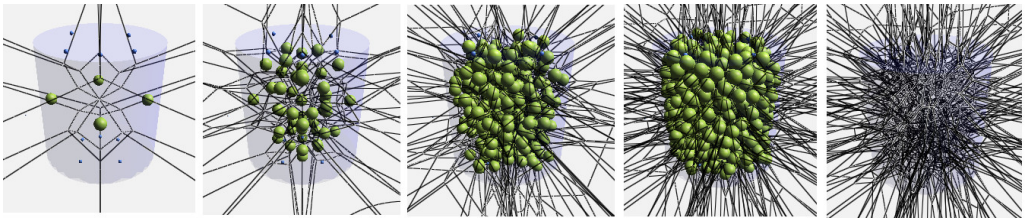

Figure 7.11: Packing spheres in a cylinder - Packing spheres in a cylinder, the radius is 3, 492 balls. Iterations: 2, 4, 6, 13, 13 (the appropriate Voronoi diagram)

The results can be compared with M. Gavrilova's approach [1] (briefly described in Chapter 3), the computations were implemented in Fortran 77 and tested on Sun Ultra 5 Workstation, and the results can be seen in Figure 3.1, Chapter 3. Their cylinder contains 300 Lennard-Jones atoms obtained by Monte Carlo relaxation with the fixed diameter of a cylinder. Our simulation of a similar cylinder showed in Figure 7.12 contains 494 balls. Our approach has more chaotic packing of spheres in comparison with approach from [1], Figure 3.1.

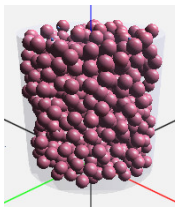

Figure 7.12: - Packing spheres in a cylinder

7.1.3 Triangle Mesh

Most of complex models are represented by triangle meshes, thus it is handy if we can pack more dense objects than cubes or cylinders. A given example is a little tricky, since it requires to create the initial VDS more accurate to avoid voids, i.e., minor or curved details are defined by a greater number of generators.

Figure 7.13 illustrates the results of packing by equal balls and it depends on a count of initial generators of the diagram. Figure 7.13(a) has random generators thus the result is not cheering, since the head part is almost empty (most of candidates were rejected in the preprocessing part). Figure 7.13(b) contains more initial generators (uniform distribution), but still the ear part is not fully filled.Figure 7.13(c) has more initial generators than the other two and apparently this result is the best one. It was decided to initialize ear part more accurately as it is possible and pack by smaller balls than the rest of a body. This makes the final result more smoother and softer.

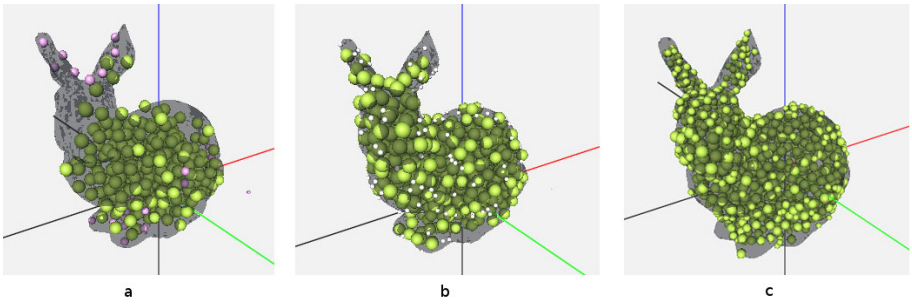

Figure 7.13: Bunny - (a) - random generators (the violet set of balls represents the initial generators). (b) - uniform distribution of initial generators (white balls represent the initial generators). (c) - the ear part contains more initial generators than the rest of the bunny (initial generators has a zero radius)

.

Figure 7.14 illustrates the packing of a bunny by equal balls stage by stage, the set of red balls represents the initial generators.

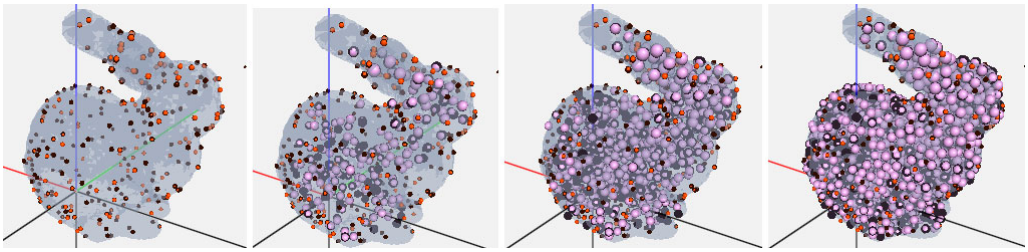

Figure 7.14: Bunny - Filling the bunny step by step. Iterations: 1, 2, 3, 6.

Packing by different spheres is shown in Figure 7.15 (with a corresponding VDS), the idea is the same as in previous examples, to insert the largest possible ball into a void. The initial digram was created by a subset of vertices of triangles, thus the distribution of generators is close to be even (every 21th vertex from the triangle mesh was taken as a start generator). At first iteration the largest founded balls which fitted in the model were added. The generated kitten contains 605 balls, the smallest ball is 1 and the largest ball was limited to 50.

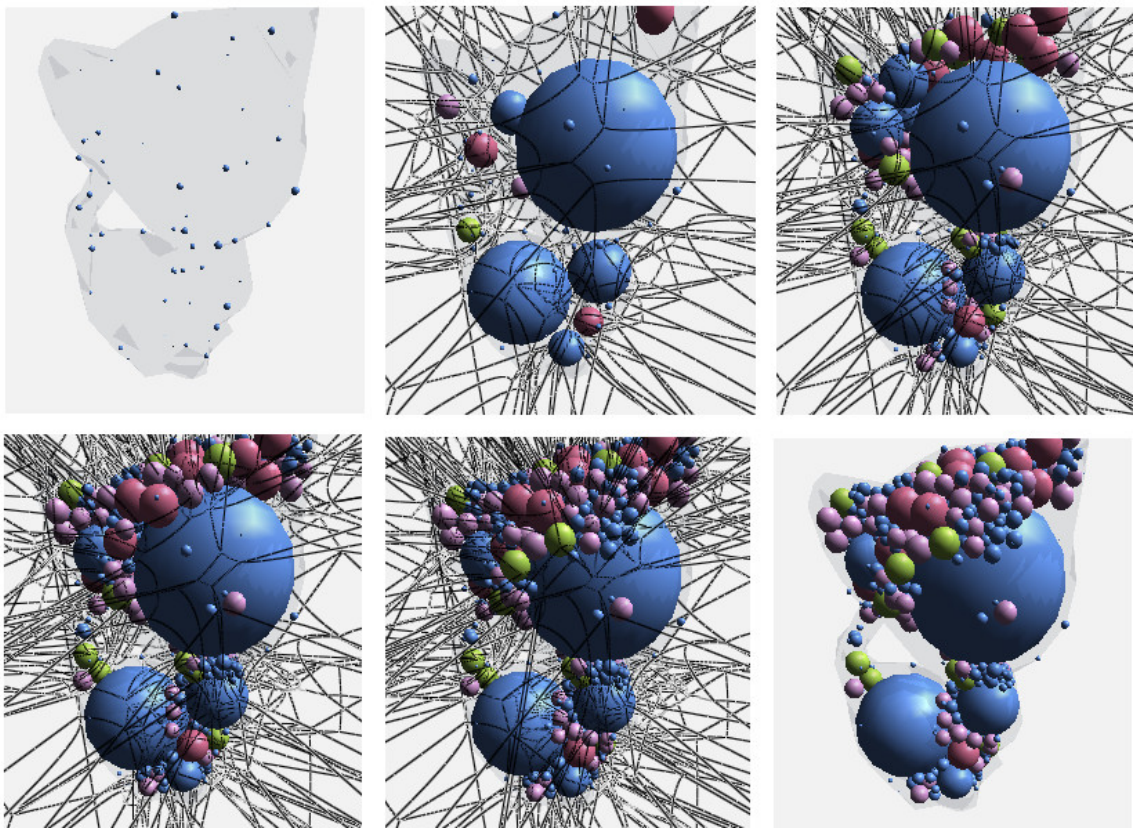

Figure 7.15: Kitten - In lines, from left to right, from top to bottom. Filling the kitten model step by step, the minimal radius =1, the maximal radius is 50. Iterations: 1, 2, 5, 7, 9, 17.

Figure 7.16 illustrates the distribution of initial generators and the following packing of the model. The minimal radius was set to 0.9, the maximal radius was limited to 50. The smaller the minimal radius, the better the result is. The bigger inscribed sphere into a void, the faster packing process is, since the number of vertices and edges in the diagram does not increase so extensively. But it is important not to forget that a big ball in rare cases could still cause voids (see *anomalies*, Chapter 2).

The packing into the Stanford Bunny is shown in Figure 7.16, the big parts are filled by big balls. The top part of the model contains more initial generators, to define the ear part more properly.

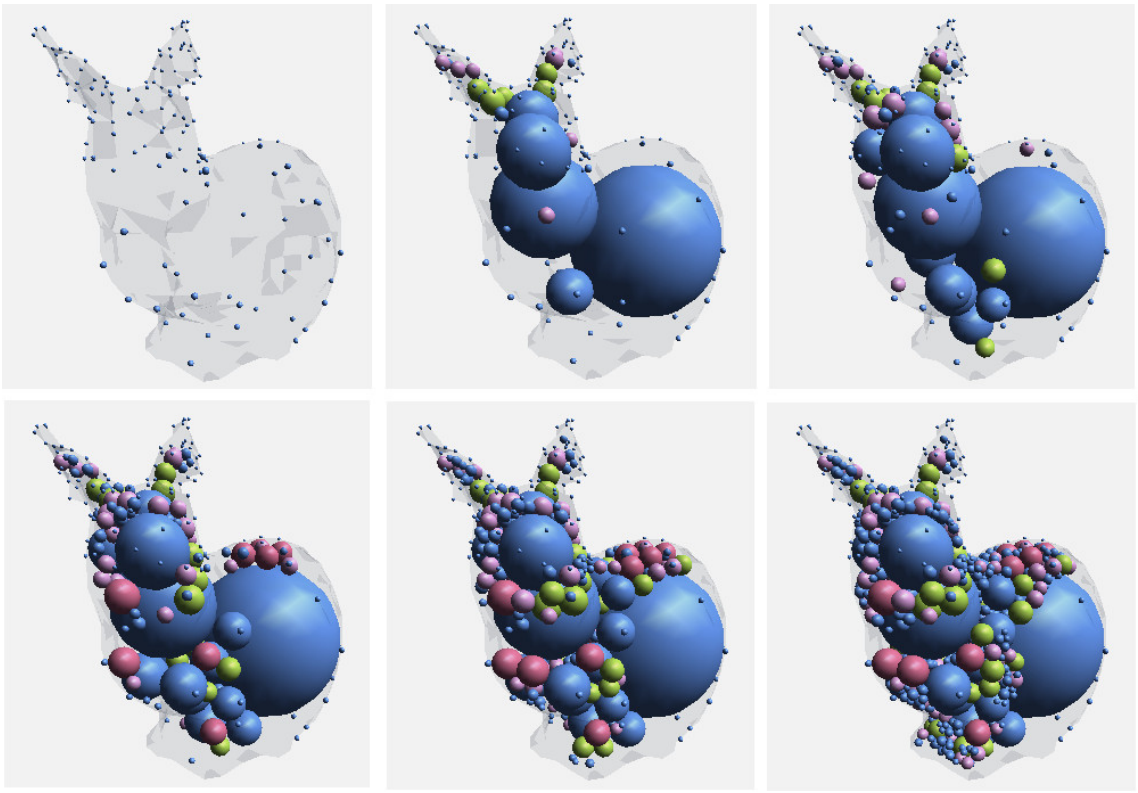

Figure 7.16: Bunny - Packing by different balls

Figure 7.17(a) the bunny has a uniform distribution of start generators, Figure 7.17(b) the top part of the model is defined by a larger number of start generators than the rest of the body. The difference between the two models is great, Figure 7.17(a) has completely empty ears and a lot of empty spaces in the bottom part.

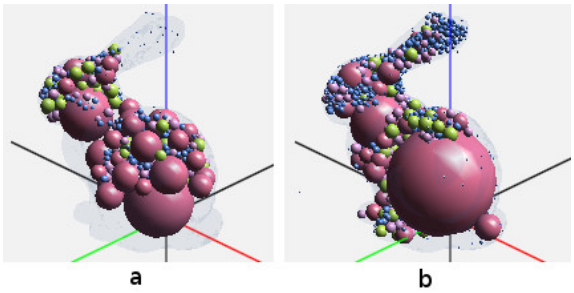

Figure 7.17: Bunny - Packing by different balls

The uniform distribution of generators in a model for creating the initial diagram does not work well on all types of triangle mesh models, e.g, Figure 7.15 captures a better packing than Figure 7.17(a). Hence, in cases that have complex models it is better to divide them into several parts and define the start generators with dependency on the complexity of a given part.

The achieved results can be compared with another method of packing spheres for arbitrary objects [21]. The idea is to start with the largest sphere that fits in the object

and iteratively insert new spheres, under the constraints that they must not intersect already existing spheres and be completely contained inside the object, the full approach is described in [21]. The Algorithm was implemented using CUDA (and a NVIDIA GTX480 graphics card), Figure 7.18 shows the result of packing a bunny. The cover volume is approximating to 94-95%.

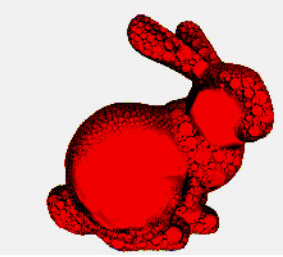

Figure 7.18: Bunny - Packing by different balls, taken from [21]

The bottleneck of the algorithm is that we calculate the Voronoi diagram of spheres for each new ball from the beginning, maybe it could work faster in case of incremental option in the VDS construction or it might be found another way of application of the VDS in packing problems, e.g., as it was suggested in [21] to use only some parts of the diagram.

Others examples of packing spheres in a container are saved on a CD, in *results/spack* directory.

7.2 Granular Fluid Simulation

The simulation is strongly depended on the Voronoi diagram, if the diagram is not constructed, the new position is rejected. In very rare cases errors in the VDS construction occurs (anomalies, see Chapter 2), thus leads to a situation when the tested ball stops its motion for a couple of iterations while others balls continue to move. The ball can start to move when the structure of the diagram has been modified (the positions of other balls due to the stopped ball has changed).

Also it could be problematic to touch the bottom of a cylinder/box or plane mathematically correctly, because three more balls on a plane or more than four balls with one inscribed sphere do not allow to construct the diagram. One of the solution is to add a very small number to a position of a ball, or in case of motion simulations shift the particle by a short distance between the partial iterations, that allows to use a previous position in case of failure of the diagram construction.

While the particle size in majority of fluid simulations is negligible and has a constant value, our particle has a weight (radius), which plays the primary role in computations. The external SPH library also counts with the same sized particles, thus it was necessary to implement an additional part of code, which tests intersections between nearby balls and the following adjustment of a particle position and a velocity direction.

Several simulations were created to illustrate the results of applying the VDS library. Figure 7.19 illustrates a fluid motion of 156 particles in a cylinder. The scene is very simple but shows the interaction between balls. The iteration between balls in a fluid does not have a attenuation, thus the motion looks a little chaotic.

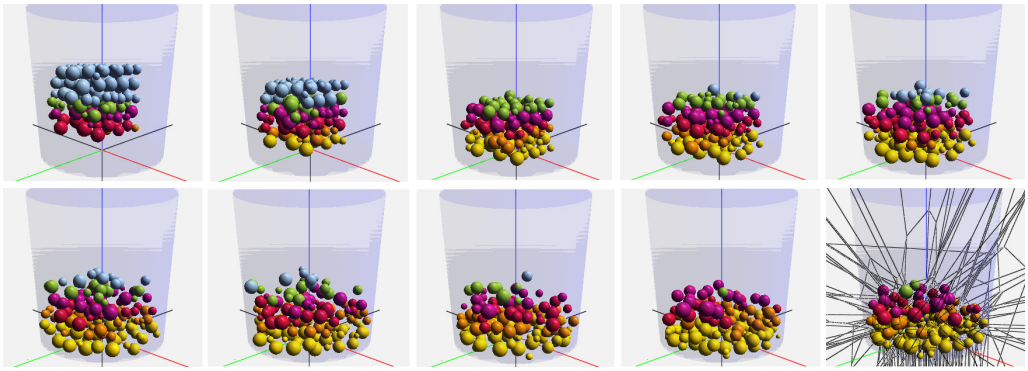

Figure 7.19: Fluid motion in a cylinder. - in lines left to right, top to bottom.

Figure 7.20 captures a fluid motion in a triangle mesh.

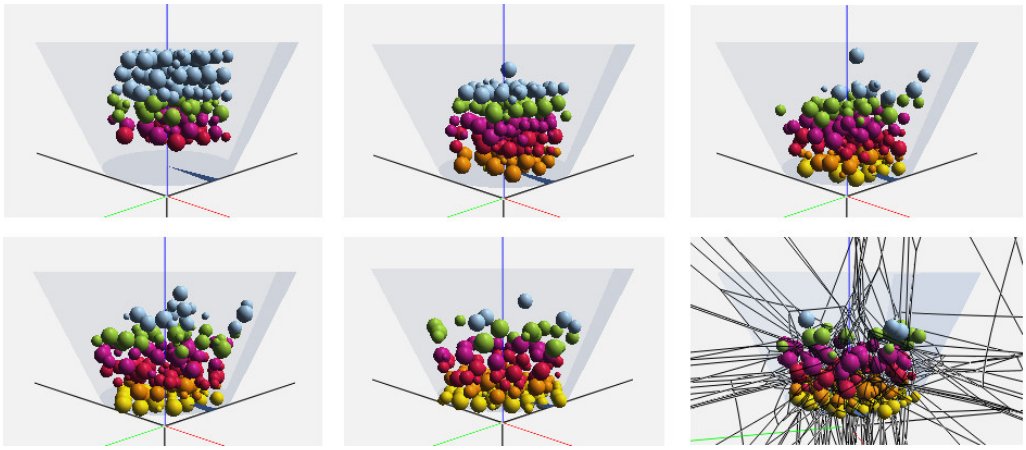

Figure 7.20: Cup (triangle mesh) - in lines left to right, top to bottom.

The results are compared with the original SPH (Fluid v.2). Unfortunately our simulation is much slower, since SPH was implemented in C++ (CPU, supports also GPU (CUDA)) and our simulation is coded in C♯. But the advantage of our method is that the set of input balls allows to have different radii and does not intersect nearby balls. The construction of the Voronoi diagram in each iteration performs well and rarely fails.

Other animations are saved on a CD, in *results/animations* directory.

Chapter 8

Conclusion

The primary objective of this work was to test the Voronoi Diagram of 3D Balls Library, to find an application of diagrams in a granular material. Half of my work was focused on container packing by balls, where the Voronoi diagram plays a role of a map. The Voronoi vertices and points on Voronoi edges were propounded as potential spots for new spheres. The advantage of the VDS was a fast neighbour search (i.e., the diagram stores the whole information about neighbouring balls), where the set of tested balls was significantly reduced. The VDS was tested on two types of data, sets of equal and different size of balls. The packing by different size balls performed quite well. The drawback was that sometimes the big difference in balls sizes caused situations when the diagram could not be constructed. Another tricky situation occurred in case of three or more balls on the same plane or the situation of four balls with one inscribed sphere. The other problem were large balls, they tended to cause voids, which sometimes could be solved by using Voronoi edges in addition as a guideline for new spheres (but it significantly increases the number of candidates). Packing by the same sized balls worked out great, the result depends only on a radii of a ball, but in this situation the idea of 3D balls had no purpose, since it could be reduced to a problem of points when all balls have the same weight. Overall the VDS library is a very useful tool in packing problems or free volume analysis (e.g., detection of voids and neighbour search in static scenes).

As for granular simulations the VDS was used for a fast nearby balls search, but in practise it showed up that for dynamic calculations when the diagram has to be recomputed with each motion of a particle in a fluid is not very effective. Unfortunately the simulation for a large number of particles was expected to be time-consuming because the main application and VDS library were implemented in *C♯* programming language. The simulation also needs to be optimized to achieve better performance. Preferably for practical use it is better to rewrite the program code in C/C++.

Abbreviations and Notation

- The following abbreviations are met throughout the diploma thesis:

VD	—	Voronoi diagram
VDS	—	Voronoi diagram of spheres
VV	—	Voronoi vertex
VE	—	Voronoi edge
VF	—	Voronoi face
SPH	—	Smooth Particle Hydrodynamics
GUI	—	Graphical User Interface

References

[1] V. A. Luchnikov, M. L. Gavrilova, N. N. Medvedev, V. P. Voloshin. The Voronoi−Delaunay approach for the free volume analysis of a packing of balls in a cylindrical container, Institute of Chemical Kinetics and Combustion, Novosibirsk, 2002. 1, 13, 29, 52

[2] M. Maňák and I. Kolingerova. Fast discovery of Voronoi vertices in the construction of Voronoi diagram of 3D balls. IEEE Computer Society, International Symposium on Voronoi Diagrams in Science and Engineering, 2010. 1, 11, 19, 20

[3] R. Hoetzlein. Fluids v.2 - a fast, open source, uid simulator. http://www.rchoetzlein.com/eng/, 2009. Online; Accessed: 15/5/2012. 1, 24

[4]] J.-D. Boissonnat, M. Yvinec. Algorithmic geometry. Cambridge: Cambridge University Press; 1998. 7

[5] C. H. Rycroft. Multiscale Modeling in Granular Flow, Ph.D. Thesis, 2007. 16, 17

[6] D. Kim, D-S. Kim. Region-expansion for the Voronoi diagram of 3D spheres. Hanyang University, 2005. vi, 11, 12

[7] V.A. Luchnikov, N.N. Medvedev, L. Oger, J.-P. Troadec. The VoronoiDelaunay analysis of voids in system of nonspherical particles. Phys. Rev. E. 59 (6)72057212, 1999. 11, 13

[8] N. N. Medvedev, V. P. Voloshin, V. A. Luchnikov, M. L. Gavrilova. An Algorithm for Three-Dimensional Voronoi S-Network, Wiley InterScience, 2006. 11

[9] D.-S. Kim, D. Kim, Y. Cho, K. Sugihara. Quasi-triangulation and interworld data structure in three dimensions. Computer-Aided Design, 38(7):808819, 2006. 3, 6, 7

[10] Y. Cho, D. Kim, and D.-S. Kim. Topology representation for the Voronoi diagram of 3D spheres. International Journal of CAD/CAM, 5(1):5968, 2005. 9

[11] M. Maňák Computational Geometry Applied to Modeling and Visualization of Proteins, Technical Report No. DCSE/TR-2010-5 May, 2010. 4, 8, 11

[12] D.-S. Kim, Y. Cho, D. Kim. Euclidean Voronoi diagram of 3D balls and its computation via tracing edges. Computer-Aided Design, 37:14121424, 2005. 6, 11

[13] M. L. Gavrilova. Proximity and Applications in General Metrics. Ph.D. Thesis: The University of Calgary, Dept. of Computer Science, Calgary, AB, Canada, 1998.

[14] M. L. Gavrilova, J. Rokne. Updating the topology of the dynamic Voronoi diagram for spheres in Euclidean d-dimensional space. Comput Aided Geometric Des, 2003.

[15] R. Wang, K. Zhou, J. Snyder, X. Liu, H. Bao, Q. Peng, and B. Guo. Variational sphere set approximation for solid objects. The Visual Computer, 22(9):612621, 2006.

[16] M. Vesterlund. Simulation and Rendering of a Viscous Fluid using Smoothed Particle Hydrodynamics, 2004.

[17] M. Desbrun, M. P. Cani. Smoothed particles: A new paradigm for animating highly deformable bodies. In Computer Animation and Simulation 96 (Proceeding of EG Workshop on Animation and Simulation), 1996.

[18] A. Schulz, F. Ganacim and L. Cruz On the Evaluation of the Voronoi-Based Medial Axis. http://ganacim.impa.br/gc/maxis/maxis.html. Online, Accessed: 15/5/2012. 33

[19] A. Okabe, B. Boots and K. Sugihara. Spatial Tessellations. Concepts and Applications of Voronoi Diagrams. J. Wiley and Sons, Chichester, New York, Brisbane, Toronto and Singapore, 1992. 5

[20] A. Mirzaian. Minimum weight Euclidean matching and weighted relative neighborhood graphs. In WADS 93: Proceedings of the Third Workshop on Algorithms and Data Structures, London, UK, 1993. 6

[21] R. Weller, G. Zachmann. ProtoSphere: A GPU-Assisted Prototype Guided Sphere Packing Algorithm for Arbitrary Objects, Clausthal University, Germany, 2009.
55, 56

Evidenční list

Souhlasím s tím, aby moje diplomová práce byla půjčována k prezenčnímu studiu v Univerzitní knihovně ZČU v Plzni.

Datum: Podpis:

Uživatel stvrzuje svým čitelným podpisem, že tuto diplomovou práci použil ke studijním účelům a prohlašuje, že ji uvede mezi použitými prameny.

Jméno	**Fakulta/katedra**	**Datum**	**Podpis**