1-1-2008

# Computer simulation of developing erosion profiles including interference effects

Nastaran Shafiei
*Ryerson University*

# COMPUTER SIMULATION OF DEVELOPING EROSION PROFILES INCLUDING INTERFERENCE EFFECTS

by

Nastaran Shafiei, BSc, Amirkabir University of Technology,
Tehran, 2004

A thesis
presented to Ryerson University
in partial fulfillment of the requirement for the degree of
Master of Applied Science
in the Program of Electrical and Computer Engineering.
Toronto, Ontario, Canada, 2008

## Author's Declaration

I hereby declare that I am the sole author of this thesis.
I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.
Signature

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.
Signature

## Instructions on Borrowers

Ryerson University requires the signatures of all persons using or photocopying this thesis. Please sign below, and give address and date.

**Computer Simulation of Developing Erosion Profiles Including Interference effects, Nastaran Shafiei, MASc, Electrical and Computer Engineering, Ryerson University, Toronto, 2008**

# Abstract

A computer simulation was developed to predict the evolution of abrasive jet machined surfaces in unmasked substrates. Interference effects were included in the model by tracking individual particles and their collisions. The model was capable of investigating the effect of various parameters such as launch frequency, abrasive particle size and material and substrate material on the shape and size of the erosion profile, as a function of time. The model is also able to examine the effect of these parameters and the instantaneous shape of the profile on the particle interference patterns. The model was verified, for the case of a flat non eroding surface, against an existing computer simulation of particle interference effects. The predictions of the simulation were also tested against experimentally measured erosion profiles, with good agreement.

# Acknowledgments

I would like to thank Dr. M. Papini for his valuable support and advice during the last two years. Without his assistance and patience this work was impossible. I also wish to thank Dr. A. Sadeghian, for his valuable support, and insights. Special thanks are also extended to my parents for their never-ending love and support.

# Contents

# List of Figures

# List of Tables

# Nomenclature

| | |
|---|---|
| $d$ | Nozzle stand-off-distance |
| $d_s$ | Depth of the Substrate |
| $D$ | Constant related to properties of target material and erosive powder |
| $E$ | Erosion rate |
| $e_{pp}$ | Coefficient of restitution for particle-particle collisions |
| $e_{ps}$ | Coefficient of restitution for particle-surface collisions |
| $f$ | Friction coefficient |
| $f_n$ | Launch frequency |
| $f(\theta)$ | Function that describes the impact angle dependence of erosion rate |
| $I$ | Particle moment of inertia |
| $k$ | Velocity exponent of the erosion rate |
| $l_e$ | Edge length of environment cells |
| $L$ | Edge length of surface cells |
| $m$ | Particle mass |
| $P_n\ P_t\ P_{t'}$ | Impulse in $n$, $t$, and $t^{'}$ directions |
| $r$ | The radial distance on the target surface from the nozzle centerline |
| $r_n$ | Nozzle radius |
| $r_p$ | Particle radius |
| $r_c$ | Cutoff radius |
| $V_n\ V_t\ V_{t'}$ | Rebound particle velocities in the $n$, $t$, and $t^{'}$ directions |

| | |
|---|---|
| $v_n\ v_t\ v_{t'}$ | Incident particle velocities in the $n$, $t$, and $t^{'}$ directions |
| $V_{max}$ | Maximum particle velocity |
| $v_{noz}$ | Nozzle velocity |
| $\alpha$ | Nozzle orientation angle |
| $\beta$ | The focus coefficient |
| $\mu$ | Impulse ratio for particle-surface collisions |
| $\mu_t\ \mu_{t'}$ | Impulse ratio for $t$ and $t^{'}$ directions |
| $\mu_c$ | Critical impulse ratio for particle-surface collisions |
| $\mu_{tc}\ \mu_{t'c}$ | Critical Impulse ratio for $t$ and $t^{'}$ directions |
| $\Omega_n\ \Omega_t\ \Omega_{t'}$ | Rebound particle angular velocities in the $n$, $t$, and $t^{'}$ directions |
| $\omega_n\ \omega_t\ \omega_{t'}$ | Incident particle angular velocities in the $n$, $t$, and $t^{'}$ directions |
| $\rho_p$ | Density of abrasive material |
| $\rho_s$ | Density of target material |

# Chapter 1

# Introduction

Abrasive jet machining (AJM) is the process of eroding a target surface material due to the impact of a stream of fine-grained abrasive particles at high velocities. The particles are mixed with a compressed carrier gas, usually air, the gas/particle mixture is passed through a nozzle and the resulting particle jet is directed onto a target surface. The target material removal occurs due to mechanical processes such as cutting, ploughing, or fracture. This process is illustrated in Figure 1.1.

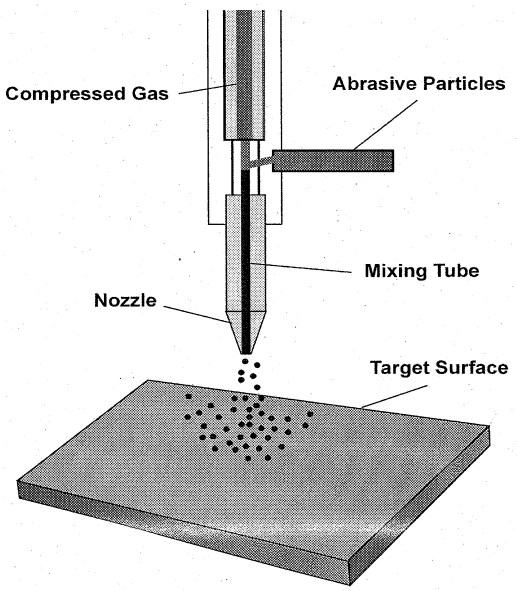



Figure 1.1: Abrasive Jet Machining Concept

AJM has been found to be highly suited for etching, polishing and cutting operations, and has a wide variety of industrial applications. One of AJM's most important applications is the powder blasting of glass and Si components for fabrication of device features (microchannels, microholes, etc) used in the microelectronic, microfluidic, and optoelectronic industries. It also has applications for deburring; i.e. the process of removing surface burrs and protuberances created during manufacturing processes. Finally, abrasive jets have also found use in dental operations, where their use instead of dental drills may make local anesthesia unnecessary due to their ability to cut small and shallow holes without significantly raising the temperature of the tooth.

## 1.1 Motivations

In most applications of AJM, the depth and shape of the machined surface are of importance. Highly accurate AJM process can be performed by controlling various parameters such as the distance between the nozzle tip and surface, called *stand-off distance*, particle size, particle velocity, nozzle diameter, etc. Many researchers have experimentally investigated the effects of these different machining parameters on the shape and size of the erosion profile in the AJM process [1, 2, 3].

Verma and Lal [1] conducted an experimental study of the AJM process. Their experimental results show the effects of stand-off distance, particle size, mixture ratio and carrier fluid pressure on material removal rate (MRR), erosion depth and the diameter of eroded holes. Venkatesh et al. [2] studied the abrasive jet machining process under different conditions and reported that the eroded surface has an elliptical bell mouthed shape. For deburring applications, Balasubramaniam et al. [3] reported the significant effects of stand-off distance on the deburred edge radius. They also investigated the effects of the nozzle diameter on the diameter of the hole generated by the jet.

Some analytical models have also been proposed for the shape and size of eroded features resulting from AJM [4, 5, 6, 7]. Slikkerveer et al. [4] proposed an analytical model for the development of powder-blasted structures in brittle materials that are locally covered with a

pattern of erosion resistant masks, having openings defining where the micro-features should be machined. In these models, the interactions between the particles and the mask edges, and any inter-particle collisions are ignored, and the mask is considered to be infinitely thin. According to the model, for deeper features, the particles rebounding from the side of the hole provide extra erosion in the center of the profile, which is called a "second strike" effect. However, using this model, the predicted machined profile is different from the experimental results, and the difference increases for the deeper profiles.

Balasubramaniam et al. [5] also proposed an analytical model to predict the shape of an abrasive jet machined surface. This model also ignores the collisions between reflected and incoming particles. In addition, the model also ignores the effect of angle of impact on the erosion, and therefore the surface slope is predicted to not affect the material removal rate (MRR). This is inconsistent with a large body of experimental measurements which show that materials have a strong dependence of erosion rate on the incident angle. The model, however, does show the effects of changes in particle size, stand-off distance and jet center line and peripheral velocity on both the shape of the eroded profile and the MRR.

Ten Thije Boonkkamp and Jansen [6] studied the erosion of a glass plate which is covered locally by an erosion resistant mask, and proposed a mathematical model for the time dependent development of hole and channel profiles made using AJM. The model ignores the effects of rebounding particles from the sides of the holes and channels, and also ignores interference effects between incoming and rebounding particles. Moreover, the shadow effect of the mask is only treated in a semi-empirical manner; i.e. the decrease in the particle flux near the edges of the mask is assumed to be linear, when in reality it depends on the kinematics of the particles as they rebound from the mask edge. The computed profile, while predicting the general shape of the profiles reasonably, unfortunately does not match experiments well. In particular, the model predicts unrealistically sharp transitions between the channel center and the channel walls, and the depth of the profile does not match experiments.

Achtsnick et al. [7] have also proposed an analytical model for AJM, and implemented it for two different nozzle configurations. In their analytical model, each impingement of

a particle with the surface is considered to be perpendicular to the surface, so that the continually changing slope of the surface cannot change the material removal rate. The model also ignores inter-particle collisions, and particle rotations. Their study does, however, demonstrate that the nozzle configuration can affect the erosion and thus the resulting blasted surface profile.

## 1.2 Objectives

The objective of this research is to construct a computer simulation which is capable of predicting the time dependent erosion profile resulting from AJM, under a wide variety of conditions. The simulation will track the movement of individual particles as they are launched from the nozzle, impact the surface, and rebound. It will thus address the shortcomings of previous models by including a number of commonly neglected factors such as particle scattering from feature edges, and the interference effects due to collisions between incoming and rebounding particles. The simulation will begin with an initially flat surface of a particular material, which is exposed to a jet of incident particles. The target surface can be one that erodes in either a 'brittle' or 'ductile' fashion (see Section 2.1). The formation of the eroded surface profile will then be predicted as a function of exposure time. The parameters that will be included in the model as inputs to the simulation can be listed as follows,

- Nozzle radius
- Nozzle stand-off distance
- Nozzle velocity
- Launch frequency
- Particle size
- Particle initial velocity

- Particle density
- Particle-Particle coefficient of restitution
- Particle-surface coefficient of restitution
- Surface density
- Friction coefficient in particle-surface collisions

## 1.3 Outline of the thesis

The rest of the thesis is organized as follows: Chapter 1 presents proposed mechanisms leading to erosion of a surface due to the impact of solid particles, and the different approaches to model these mechanisms. In Chapter 3 the assumptions of the model, collisions mechanics, and the criteria used to remove the target material are described. This chapter also presents all the algorithms used to develop the simulation. Chapter 4 discusses the performance of the simulation and presents methods used to verify the simulation. This chapter also includes a comparison between simulated results and experimentally obtained results and limitations of the model. Finally chapter 5 presents the conclusions and discusses the possible future work.

# Chapter 2

# Background

This chapter will present existing literature related to the mechanisms leading to erosion of a surface by an incident jet of particles, and the different surface advancement algorithms needed for modeling these mechanisms.

## 2.1 Erosion

The repetitive impacts of solid particles on a surface results in material removal. The behavior of an eroding material under solid particle impact can be classified into two main categories: ductile and brittle erosion. In ductile erosive systems, significant plastic deformation occurs surround the particle impact sites, as the target material is removed. On the other hand, the target material in brittle erosive systems experience little deformation before fracture. Several mechanisms of material removal due to the impact of abrasive particles have been proposed. For example, Finnie [8] has reported that for ductile erosive systems, the material removal is due to the cutting or ploughing action of erosive particles, while for brittle erosive systems the material removal is due to the intersection of cracks which radiate out from the contact point.

A description of the solid particle erosion mechanisms was first proposed by Bitter [9]. According to his analysis, if, during the collision, the yield strength of material is exceeded, plastic deformation occurs at the region near the impact point. Due to the multiple impacts, a plastically deformed surface layer is formed. The yield strength at the surface of the

material increases with the degree of plastic deformation, due to work hardening effects. Once the yield strength becomes equal to the strength of the material, no further plastic deformation occurs. At this point, the surface becomes hard and brittle, and parts of it can be removed by subsequent impacts.

For erosion of ductile materials impacted obliquely by spherical particles, Hutchings and Winter [10] assumed that shearing of the surface layers results in lips forming around the impact crater. According to their theory, these lips can be removed from the surface in subsequent impacts by advancing the ruptures at the base of the lips. They suggested that, for the cases in which the leading face of an angular particle makes a small angle with the target surface, this same spherical particle mechanism applies. They found that in work-hardened metal the deformation energy is intensely focused in the surface layers and fragile lips are created, while in annealed metal, the deformation energy extends over a large volume of the material. As a result, the target material is more easily removed in work-hardened metals.

The erosion of the surface depends on many factors such as the impact angle, velocity, shape and size of the incident particles, the number of particles hitting the surface, and the mechanical properties of the target material and the particles [8]. Regardless of the material removal mechanism, the material removal for a given target, particle and jet combination is usually expressed as a measured erosion rate, $E$. $E$ is defined as the ratio of the removed mass of the target material to the mass of the particles impacting the target:

$$E = \frac{\rho_t V_t^r}{\rho_p V_p} \tag{2.1}$$

where $\rho_t$ is the density of target material, $V_t^r$ is the volume of target material removed, $\rho_p$ is the density of particles, and $V_p$ is the volume of particles hitting the surface in the time in which the target material is removed. $E$ is usually measured in terms of a particle incident velocity, $v_p$, and the impact angle $\theta$ with respect to a surface normal [4].

$$E = Dv_p^k f(\theta) \tag{2.2}$$

where $f(\theta)$ is an experimentally determined function that describes the impact angle dependence of erosion rate, and both $D$ and $k$ are experimentally measured constants for a particular combination of target, particle and jet parameters; they depend on particle size, shape and material properties, as well as target material properties. In the ductile erosive system, the maximum material removal rate usually occurs at an impact angle between $20^o - 30^o$ [11]. For brittle erosive systems, it has been demonstrated that the erosion rate is dependent only on the normal component of incident velocity. Such erosive systems thus exhibit a maximum erosion rate at normal impact [11, 12].

## 2.2 Surface Advancement Algorithms

A number of different algorithms have been proposed to model surface advancement problems [13, 14, 15, 16]. The aim of these algorithms is to determine the surface or profile of the material as a function of the time of exposure of the surface to a damaging medium (e.g. acid etches). This section will thus briefly describe the most widely used methods to model surface advancement: string, ray-tracing, level set, and cellular methods.

### 2.2.1 String Algorithm

The string algorithm is based on the string model proposed by Jewett et al. [13] for modeling cross-sectional profiles of lines in lithographic fabrication. This approach uses a string of points connected together by line segments to approximate the surface. The line segments are roughly equal in length to gain a good resolution and avoid computational difficulties. The surface propagates by moving each point along vectors perpendicular to the local surface. Since this algorithm only monitors the etched area instead of the whole volume of material, it is faster than an algorithm based on cell removal. However, a significant problem that exists with this algorithm is the forming of loops in the string of points. [17]. Loops must be

removed before they expand and intersect, otherwise, the computational time and memory usage are highly increased. A forming loop can also result in the incorrect calculation of the normal to the surface which determines the direction of etching. This algorithm can thus be used, but with the computational expense of implementing a mechanism to remove the loops every few time steps.

### 2.2.2 Ray-Tracing Algorithm

The ray-tracing algorithm is based on the Hagouel's ray model [14] for X-ray lithography fabrication. In this algorithm, an etch ray is defined as a vector perpendicular to the border of developed and undeveloped region. The etch rays are refracted at the boundaries of regions with different etch rates. The algorithm starts with a set of points on the surface which are propagated along rays. Initially the rays for each point are perpendicular to the surface. After each step, the ray vectors are recomputed using the differential ray equation (Eq.(1) in [17]). The advantage of the ray algorithm is that calculating ray vectors is independent of the local etched surface, so it is not influenced by incorrect surfaces. The weakness of this algorithm is that it only traces points and does not keep track of connections between points. Thus, when the surface advances and end points of the rays become separated in space, the etched area cannot be easily determined.

### 2.2.3 Level Set Algorithm

Level set methods have been proposed by Osher and Sethian [15] for a wide variety of surface evolution problems. These numerical algorithms are based on a Hamilton-Jacobi equation for a propagating level set function. The main idea behind the method is the formulation of equations of motion for propagating fronts which are moving with a curvature dependent speed. Given an initial position for an interface $\Gamma$, the propagating interface is set as the zero level set of a higher dimensional function $\phi$ [18].

$$\phi(x, t = 0) = d \tag{2.3}$$

where $d$ is the distance from a point $x \in R^n$ to $\Gamma$, and takes positive values inside the closed region restricted by $\Gamma$, negative values outside of this region, and zero in the boundary of the region. This region is thus the set of points in the plane for which the level set method $\phi$ is positive or zero. The general form of the level set method representing a hypersurface $\Gamma$ can be described as,

$$\Gamma(t) = \{x|\phi(x,t) = 0\} \tag{2.4}$$

For the zero-level set moving with speed $V$ in the same direction as its normal, an evolution equation can be represented by a Hamilton-Jacobi equation as,

$$\phi_t = V|\nabla\phi| \tag{2.5}$$

One of the most important advantage of these methods is that they are able to precisely determine the geometric properties of the surface. The disadvantage of the level set techniques is that setting appropriate velocities for advancing the level set function can be very complicated.

### 2.2.4 Cellular Algorithm

Cellular algorithms are based on the cell-removal model originally proposed by Dill [16] for the exposure and development of a photoresist. The model is based on dividing the material to be etched into small cells. Each cell is distinguished as etched or unetched, and the surface is determined by the boundary between the etched and unetched cells. In Dill's algorithm [16], the etching starts from the top layer which is in contact with the developer. The developer removes cells based on a given etch rate and the number of sides of the cell exposed to the developer. After removing a given cell, new cells are exposed to the developer and begin etching.

To reduce the memory requirement, in the cellular algorithm proposed by Scheckler et al.

[19], the cell information is dynamically allocated. The cells are stored in a three dimensional array, and each cell includes information indicating if it is full or empty. The cells nearest to the surface are allocated more information: the average etch rate, the percentage of the cell volume removed so far, and the pointer to the cell. After removing each cell, the newly uncovered cells are allocated new information.

Strasser et al. [20] proposed a cellular algorithm which uses a different method to propagate the surface. This model is proposed for surface evolution in topography simulation and is based on two basic morphological operations, *Minkowski* subtraction and *Minkowski* addition. These operations, which are derived from image processing techniques, change the image with respect to a given geometric shape called a *structuring element.* The spatial dimensions of the structuring element determine the manner in which the surface advances. The local etch or deposition rate is used to calculate the spatial dimensions of the structuring element. The surface advances by moving the structuring element along the surface boundary. Depending on the type of the simulation, etch or deposition, material cells contained inside the structuring element can be removed or added. Cells are stored in an array and are characterized as etched or unetched. This algorithm also implements a dynamic allocation of information; a link list is used to dynamically store array addresses and etch or deposition rate information of the surface cells.

Cellular algorithms are robust and relatively easy to implement. They can also easily deal with arbitrary structures and inherently avoid the looping problem which occurs in some of the other surface advancement methods [13]. The disadvantage of cellular algorithms is that they require significant amounts of physical memory and computational time. These algorithms are also subject to the lack of accuracy in representing curved surfaces by a series of rectangular cells. The accuracy can be increased by using smaller cells, but this increases the memory usage and computational time.

In the present model a cellular algorithm was used to model the surface advancement. As there is a trade-off between efficiency and accuracy, to provide an accurate model, a number of different techniques were used to improve the run time and memory requirements. The

algorithm uses an efficient hierarchical data structure to store the volume of the material, and surface cells are dynamically created (see Section 3.10). Also to improve the accuracy in representing the surface, cells can be represented by their partially eroded depth.

# Chapter 3

# Methodology

This chapter will presents the assumptions used in the simulation, the details of the collisions mechanics, and the criteria used to remove the material. It will also describe all the algorithms used in the model. The algorithms are implemented in the Java programming language and the source code can be found in Appendix A.

## 3.1 General Description and Assumptions

The model is meant to simulate the developing erosion profile due to the impact of a jet of small spherical abrasive particles. The nozzle of radius, $r_n$, placed at a stand-off distance, $d$, can move in a line at a constant velocity, $v_{noz}$, and launches a jet of particles at a launch frequency, $f_n$. The launched particles have a radius, $r_p$, a density, $\rho_p$, and a velocity, $v_p$, and impact an initially flat target having a density, $\rho_s$, depth, $d_s$. Target material removal by mechanical erosion is assumed to occur as a result of the impact of the particle jet (Figure 3.1).

Particles are assumed spherical, and all of the same size and density. The effects of any external forces, such as drag and gravitational effects, are neglected; therefore, particles are assumed to be moving on straight paths from the nozzle to the surface. The nozzle launches the particles, one at a time, at an angle having a spatial distribution which is a combination of the Weibull and uniform distributions, with a velocity that decreases linearly across the jet, from the nozzle centerline (see Section 3.2). The model also neglects particle fracture; i.e.

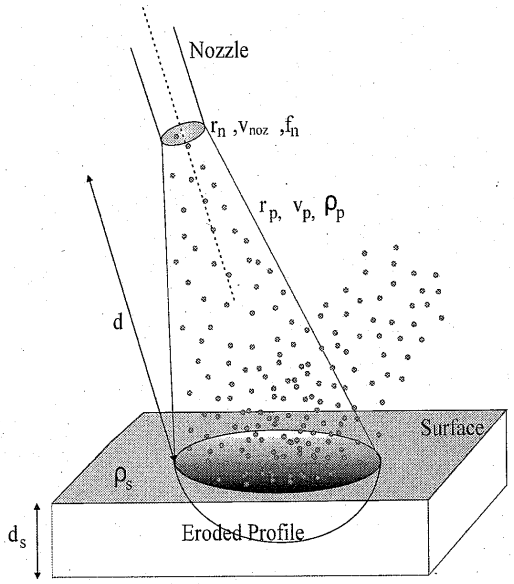



Figure 3.1: Unmasked profile

the possibility that the particles break during and after impacts. Collisions between any two particles at one time are allowed, so that changes in flux to the surface due to inter-particle collisions are considered. Inter-particle, and particle to surface collisions are treated using a coefficient of restitution approach [21] (see Section 3.3). The effect of friction is included in particle-surface collisions, but is neglected in inter-particle collisions, an assumption used in previous simulations of particle interference effects, with good success [22].

## 3.2 Particle Spatial and Velocity Distribution Across Jet

Measurements [23] at a low incident particle flux (i.e. so that inter-particle collisions have no effect on the distribution) show that, for a typical abrasive jet micromachining setup, the probability of a particle arriving to the flat surface at a distance between $r$ and $r + dr$ from the nozzle centerline is

$$P(r)dr = \frac{2\beta^2}{d^2} r e^{-\beta^2 \frac{r}{d}^2} dr \tag{3.1}$$

where $d$ is the nozzle to surface stand off distance (Figure 3.2), and $\beta$ is an experimentally determined dimensionless constant which depends on the nozzle configuration, particle shape, size, velocity, etc. The $\beta$ is commonly called the 'focus coefficient', and for more focused streams , its value increases. Using simple geometry, $r$ can be expressed as,

$$r = d\ tan(\theta) \tag{3.2}$$

where $\theta$ is the angle between the nozzle centerline and a particle trajectory (Figure 3.2). The derivative of Eq. (3.2) with respect to $\theta$ is,

$$\frac{dr}{d\theta} = dsec^2(\theta) \tag{3.3}$$

Since particles are assumed to travel in straight paths from the nozzle to the surface,

$$P(r)dr = P(\theta)d\theta \tag{3.4}$$

Substituting Eqs. (3.2) and (3.3) into Eq. (3.4) gives,

$$P(\theta)d\theta = \frac{2\beta^2 sin(\theta)}{cos^3(\theta)} re^{-\beta^2 tan^2(\theta)} d\theta \qquad 0 \leq \theta \leq \frac{\pi}{2} \tag{3.5}$$

which is the probability that a particle is launched from the nozzle on a trajectory between $\theta$ and $\theta + d\theta$. Since the jet of particles is symmetrical with respect to the nozzle centerline, the probability that a particle is launched at any direction around the nozzle centerline is the same for all directions,

$$P(\Phi) = \frac{1}{2\Pi} \tag{3.6}$$

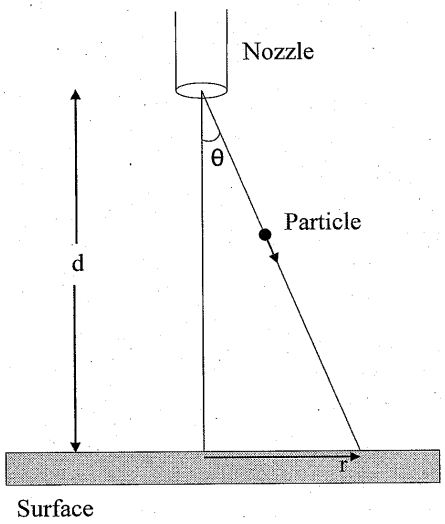



Figure 3.2: Trajectory of a particle

with angle $\Phi$ defined in Figure 3.3.

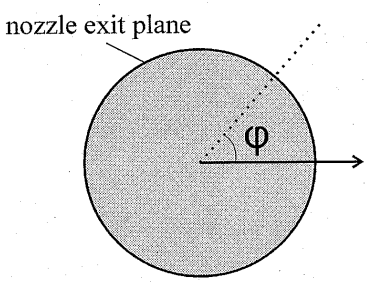



Figure 3.3: The nozzle exit plane and the launching angle.

The distribution function given in Eq. (3.5) assumes a nozzle that launches particles from a point source. To avoid problems when the surface is relatively close to the nozzle, or when the nozzle is relatively large, a method was developed to approximate the point source nozzle by one of a finite size. To be able to launch particles from a finite size nozzle and still have the particles arrive to the surface at a radial location, $r$, described by the Weibull distribution, Eq. (3.1), a cutoff radius , $r_c$, at which it is highly unlikely any particle arrives,

must be set. From the cumulative Weibull distribution, the probability that particles arrive to the surface at a distance $r$ which $r < r_c$ is given as,

$$D(r_c) = 1 - e^{-(\frac{\beta r_c}{d})^2} \tag{3.7}$$

Assuming that $D(r_c) = 0.99999$ (i.e. that 99.999% of the particles are launched within the cutoff radius $r_c$),

$$r_c = \frac{d\sqrt{-4ln(1 - 0.99999)}}{2\beta} \tag{3.8}$$

The position on the nozzle exit plane from which particles are launched, $r_l$, is proportional to the radial position at which they must arrive to the surface, $r$, in order to satisfy the Weibull distribution (Eq. 3.1). In this scheme, particles arriving to the surface at the radial position, $r_c$, are assumed to have been launched at the outermost portion of the nozzle cross section, i.e. at $r_n$. This can be generalized as:

$$r_l = \frac{r_n}{r_c} r \tag{3.9}$$

From Figure 3.4, the angle at which particles are launched is,

$$\theta = tan^{-1}(\frac{r - r_l}{d}) \tag{3.10}$$

This approach thus slightly changes the positions from which particles are launched.

According to the measurements for a 0.76 $mm$ nozzle launching 25 micron diameter particles, the velocity distribution as a function of radial distance, $r$, from nozzle centerline to the surface is [23],

$$V(r) = V_{max}(1 - 4.92\frac{r}{h}) \tag{3.11}$$

where $V_{max}$ is a maximum particle velocity which is observed on the nozzle centerline ($r = 0$).

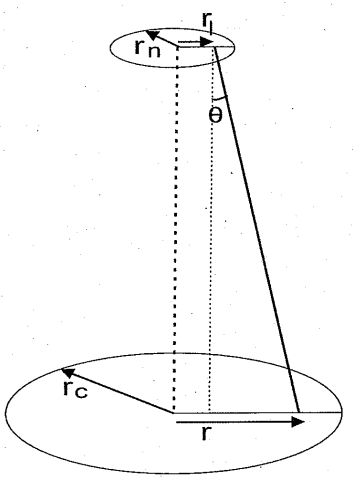



Figure 3.4: Finite size nozzle

## 3.3 Particle Scattering for Inter-Particle Collisions

The present model assumes that inter-particle collisions occur between only two particles at a time - i.e. three or more particles simultaneously impacting at the same location is not allowed. In the rare occurrences where three or more particles are found to collide at the same point and time, a sequence of collisions involving two particles is used instead. The analysis of the inter-particle collisions used in the present work is based on the coefficient of restitution approach first proposed by Brach [21].

The velocity of the colliding particles can be decomposed into the normal component which is parallel to the line joining the centers of the two colliding particles, and tangential component which is perpendicular to this line (see Figure 3.5). As mentioned in section 3.1, the effect of friction between particles is neglected. Hence, the impulse of the colliding

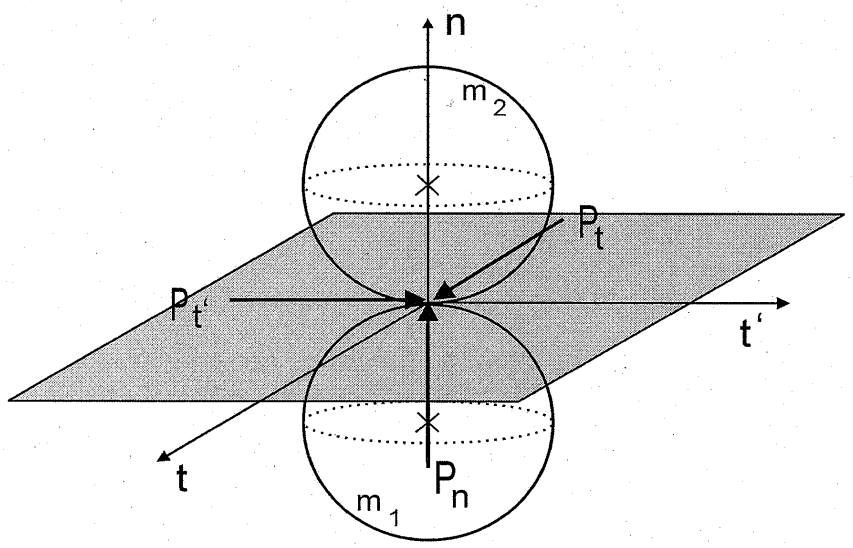



Figure 3.5: Diagram of the two spherical rigid body collision showing the dimensions and coordinates

particles is normal to the surface of collision, and the tangential components of the particle velocity do not change during the collision. By applying the law of conservation of momentum,

$$m_1 v_{n1} + m_2 v_{n2} = m_1 V_{n1} + m_2 V_{n2} \tag{3.12}$$

where $v_{n1}$ and $v_{n2}$ are the normal components of the incident velocities, and $V_{n1}$ and $V_{n2}$ are the normal components of the final velocities. By the definition of the coefficient of restitution, $e_{pp}$,

$$e_{pp} = -\frac{V_{n2} - V_{n1}}{v_{n2} - v_{n1}} \tag{3.13}$$

Solving Eqs. (3.12) and (3.13), gives the normal components of the final velocities, as follows:

$$V_{n1} = \frac{m_2 \left[ v_{n2} + e_{pp}(v_{n2} - v_{n1}) \right] + m_1 v_{n1}}{m_1 + m_2} \tag{3.14}$$

$$V_{n2} = \frac{m_1 \left[ v_{n1} - e_{pp}(v_{n2} - v_{n1}) \right] + m_2 v_{n2}}{m_1 + m_2} \tag{3.15}$$

## 3.4 Particle Scattering for Particle-Surface Collisions

The frictional impact of a particle against the target surface is also assumed to follow the coefficient of restitution approach first proposed by Brach [21]. The development here is similar to that presented by Ciampini et al. [22] and follows the two-dimensional impact model proposed by Brach [24]. Figure 3.6 shows a sphere colliding with a flat plane. The $n$ axis is normal to the plane, and the $t$ and $t'$ directions are tangential to the plane. $P_n$, $P_t$ and $P_{t'}$ are the components of impulse in the three directions, $n$, $t$, and $t'$, respectively. The impulse is regarded as the change in momentum of an object to which a resisting force is applied during the collision.

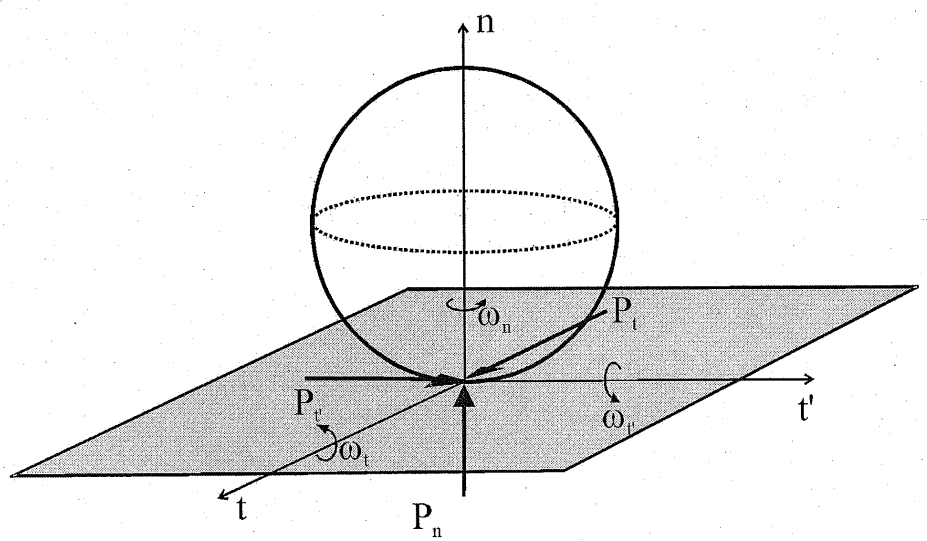



Figure 3.6: Diagram of spherical rigid body impact on the flat surface showing the dimensions and coordinates [22]

Brach [24] discusses three different cases for a rigid body impact on a flat massive surface: sliding exists during the whole of the contact, sliding stops at the rebound, and sliding stops prior to the rebound. In the latter case, after sliding, rolling begins. The appropriate equations of motion for the case of sliding during the whole of the contact period are,

$$V_n = -e_{ps} v_n \tag{3.16}$$

$$\mu_t V_n - V_t = \mu_t v_n - v_t \tag{3.17}$$

$$\mu_{t'} V_n - V_{t'} = \mu_{t'} v_n - v_{t'} \tag{3.18}$$

$$mrV_t + I\Omega_{t'} = mrv_t + I\omega_{t'} \tag{3.19}$$

$$-mrV_{t'} + I\Omega_t = -mrv_{t'} + I\omega_t \tag{3.20}$$

where $V_n$, $V_t$ and $V_{t'}$ are the components of the rebound velocity, and $v_n$, $v_t$ and $v_{t'}$ are the components of the incident velocity. $e_{ps}$ is the coefficient of restitution for particle-surface collisions. $\mu_t$ and $\mu_{t'}$ are the kinetic coefficients or impulse ratios. $\Omega_t$ and $\Omega_{t'}$ are the components of the rebound angular velocity, and $\omega_t$ and $\omega_{t'}$ are the components of the incident angular velocity. $m$, $r$, and $I$ are the sphere mass, radius and mass moment of inertia, respectively. Eq. (3.16) is simply the definition of the coefficient of restitution, and Eq. (3.17) and Eq. (3.18) are derived from the definitions of kinetic coefficients; i.e. the ratio of the tangential impulse components to the normal impulse components.

$$\mu_t = \frac{P_t}{P_n} = \frac{V_t - v_t}{V_n - v_n} \tag{3.21}$$

$$\mu_{t'} = \frac{P_{t'}}{P_n} = \frac{V_{t'} - v_{t'}}{V_n - v_n} \tag{3.22}$$

Eqs. (3.19) and Eq. (3.20) express the conservation of angular momentum about the contact point. Solving the Eqs. (3.17)-(3.20) for the rebound velocity components for the case of sliding during the whole of the contact,

$$V_n = -e_{ps} v_n \tag{3.23}$$

$$V_t = -\mu_t v_n (1 + e_{ps}) + v_t \tag{3.24}$$

$$V_t = -\mu_{t'} v_n (1 + e_{ps}) + v_{t'} \tag{3.25}$$

$$\Omega_t = \omega_t - \frac{5}{2r} \mu_{t'} v_n (1 + e_{ps}) \tag{3.26}$$

$$\Omega_{t'} = \omega_{t'} - \frac{5}{2r}\mu_t v_n(1+e_{ps}) \tag{3.27}$$

For the cases in which sliding ends at or prior to the rebound, Eqs. (3.17) and (3.18) must be replaced with Eqs. (3.28) and (3.29):

$$V_t - r\Omega = 0 \tag{3.28}$$

$$V_{t'} - r\Omega = 0 \tag{3.29}$$

Solving the Eqs. (3.16), (3.28), (3.29), (3.19) and (3.20) for the rebound velocity components for the case that sliding ends at or prior to the separation and sticking occurs,

$$V_n = -e_{ps}v_n \tag{3.30}$$

$$V_t = \frac{5}{7}v_t + \frac{2}{7}r\omega_{t'} \tag{3.31}$$

$$V_{t'} = \frac{5}{7}v_{t'} - \frac{2}{7}r\omega_t \tag{3.32}$$

$$\Omega_t = -\frac{5}{7r}v_{t'} + \frac{2}{7}\omega_t \tag{3.33}$$

$$\Omega_{t'} = -\frac{5}{7r}v_t + \frac{2}{7}\omega_{t'} \tag{3.34}$$

Brach [24] shows that there is a limiting or critical value of impulse ratios which maximizes the energy loss. The critical condition occurs at a point that sliding ends and sticking and rolling begins. At this point, the solution equations for sliding and rolling are the same. The critical impulse ratios can be obtained by equating the Eqs. (3.24) and (3.31), and Eqs. (3.25) and (3.32).

$$\mu_{tc} = \frac{2}{7}\frac{[v_t - r\omega_{t'}]}{v_n(1+e_{ps})} \tag{3.35}$$

$$\mu_{t'c} = \frac{2}{7}\frac{[v_{t'} + r\omega_t]}{v_n(1+e_{ps})} \tag{3.36}$$

It should be noted that with $\mu_t = \mu_{tc}$ and $\mu_{t'} = \mu_{t'c}$, the set of sliding Eqs. (3.16)-(3.20) gives the same equations as the rolling equations, Eqs. (3.30)-(3.34). Thus, only the set of Eqs. (3.16)-(3.20) are sufficient for all the cases, as long as the critical impulse ratios are used for the kinematic coefficients when sticking occurs.

It is assumed that all collisions occur with a mechanism of tangential impulse which is due to simple dry-friction. In this case, the impulse ratios are not independent [21] and can be expressed as,

$$\mu_t = \mu cos(\eta) \tag{3.37}$$

$$\mu_{t'} = \mu sin(\eta) \tag{3.38}$$

and the resultant impulse ratio, $\mu$ is

$$\mu = \sqrt{\mu_t^2 + \mu_{t'}^2} \tag{3.39}$$

In the case of dry-friction, the impulse direction is parallel to the motion direction and acting in the direction opposing that of the motion (Figure 3.7) [22]. Accordingly, the resultant

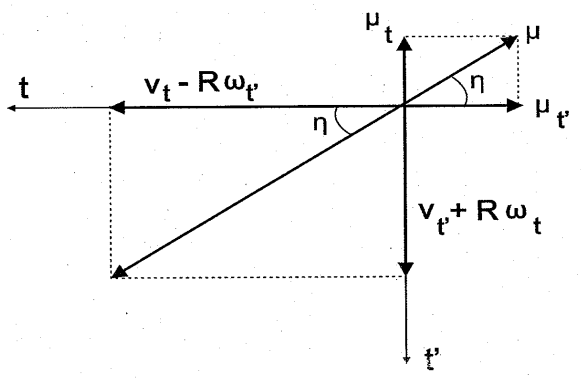



Figure 3.7: Directions of velocity components, and resulting impulse ratios, $\mu_i$, in tangential directions $t$ and $t'$

critical impulse ratio is

$$\mu_c = \sqrt{\mu_{tc}^2 + \mu_{t'c}^2} \tag{3.40}$$

In reference [24], to interpret the impact data, the friction coefficient, $f$, is compared to the critical impulse ratio, and two cases arise. The first one is when $f < |\mu_c|$. In this case, the sphere slides during the whole of the impact, and Eqs. (3.16)-(3.20) are solved with $\mu_t = f cos(\eta)$ and $\mu_{t'} = f sin(\eta)$. The second case is when $f \geq |\mu_c|$. In this case, sliding ends at or prior to the rebound, and Eqs. (3.16)-(3.20) are solved with $\mu_t = \mu_{tc}$ and $\mu_{t'} = \mu_{t'c}$. In both cases $\mu_t$ and $\mu_{t'}$ have the same signs as $\mu_{tc}$ and $\mu_{t'c}$, respectively (Figure 3.7).

## 3.5 Criteria for Removing Surface Cells

As described in Section 2.1, Eq. (2.1) is used to define the erosion rate. Assuming that a unit area, $A_t$, of the target material is exposed to the incoming particles for a time period of $\Delta t$, Eq. (2.1) can be written as,

$$E = \frac{\rho_t A_t \Delta l}{m_p} \tag{3.41}$$

where $\Delta l$ is a depth to which the area, $A_t$, is eroded, and $m_p$ is the mass of particles impacting the area, $A_t$, in the time period of $\Delta t$. By equating Eq. (3.41) to (2.2), the depth to which the area, $A_t$, is eroded in time $\Delta t$ can be expressed as,

$$\Delta l = \frac{D v_p^k f(\theta) m_p}{\rho_t A_t} \tag{3.42}$$

If in the $\Delta t$ time period, the area, $A_t$, is impacted by $n$ particles, each with its own $v_p$ and $\theta$,

$$\Delta l = \frac{D}{\rho_t A_t} \sum_{i=1}^{n} [m_{p_i} v_{p_i}^k f(\theta_i)] \tag{3.43}$$

$m_{p_i}$, $v_{p_i}$ and $\theta_i$, respectively, are the mass, incident velocity and impact angle of the $i^{th}$ impacting particle. As explained in Section 2.2, the surface will be represented by a series of interconnected volumetric cells. To predict the number of cells removed, $N_c$, in time period of $\Delta t$, it is assumed that $A_t$ is the exposed area of one cell having the height of $h$ (see Figure 3.8).

$$\Delta l = N_c h \tag{3.44}$$

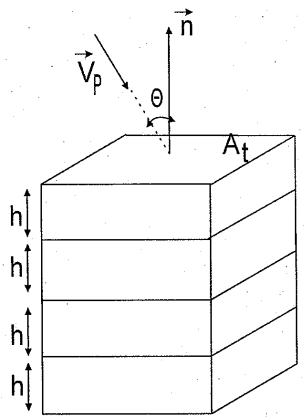



Figure 3.8: The stack of cells having surface area, $A_t$, and height, $h$, impacted by a particle with velocity, $\vec{v_p}$, at the angle, $\theta$, to the surface normal, $\vec{n}$

Therefore, using Eqs. (3.43) and (3.44), the number of cells removed in a given time period when $n$ particles are launched is:

$$N_c = \frac{D}{h\rho_t A_t} \sum_{i=1}^{n} [m_{p_i} v_{p_i}^k f(\theta_i)] \tag{3.45}$$

Eq. (3.45) can be applied for the cases that particles impact the top surfaces of cells only. However, in general,both the top and the side of cells can be impacted by particles. Following the same approach, for the cases that particles hit the side surfaces of cells, the number of cells removed from the side can be obtained as follows

$$N_c = \frac{D}{s\rho_t A_s} \sum_{i=1}^{n} [m_{p_i} v_{p_i}^k f(\theta_i)] \tag{3.46}$$

where $s$ is the width of the side surface of a cell, $A_s$ is the side area of the cell exposed to the incoming particles, and $\theta_i$ is the angle between the velocity of the $i^{th}$ particle and the normal to the side surface of a cell (see Figure 3.9). To apply a single equation for impingement on

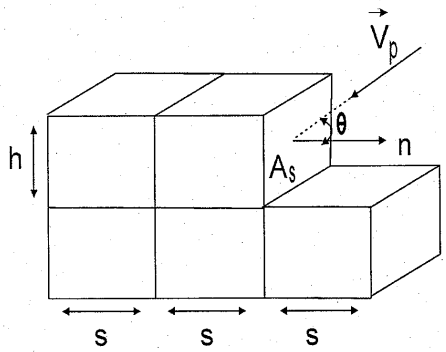



Figure 3.9: A particle with velocity, $\vec{v_{ps}}$, impacting the side area, $A_s$, of a cell at the angle, $\theta$, to the side surface normal, $\vec{n}$.

any face of the cell, $h$ and $A_t$ in Eq. (3.45), and both $s$ and $A_s$ in Eq. (3.46) can be replaced by the volume of a cubic cell, $V_c$.

$$N_c = \frac{D}{\rho_t V_c} \sum_{i=1}^{n} [m_{p_i} v_{p_i}^k f(\theta_i)] \tag{3.47}$$

To establish a criteria for a given cell being removed, Eq. (3.47) is set equal 1. Thus, a cell is removed when,

$$\frac{D}{\rho_t V_c} \sum_{i=1}^{n} [m_{p_i} v_{p_i}^k f(\theta_i)] \geq 1 \tag{3.48}$$

Since in brittle erosive systems, the erosion rate is dependent only on the normal component

of incident velocity, in such systems, $v_{p_i}$ and $f(\theta_i)$ in Eq. (3.48), are replaced by the normal component of the incident velocity [12], so that

$$\frac{D}{\rho_t V_c} \sum_{i=1}^{n} [m_{p_i} (v_{p_i} cos(\theta_i))^k] \geq 1 \tag{3.49}$$

For ductile erosive systems, Eq. (3.48) can be written as, [25]

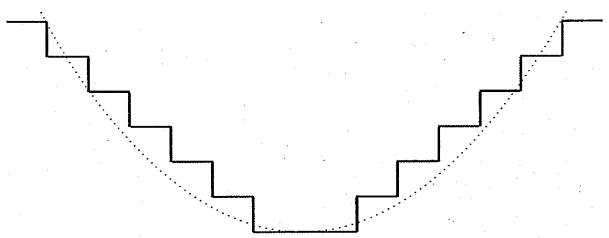
$$\frac{D}{\rho_t V_c} \sum_{i=1}^{n} [m_{p_i} v_{p_i}^k cos(\theta_i)^{n_1} (1 + Hv(1 - cos(\theta_i)))^{n_2}] \geq 1 \tag{3.50}$$

where $n_1$ and $n_2$ are experimentally determined constants, and $Hv$(GPa) is the hardness of the material.

For the cases that particles hit the edges or vertices of a cell, the normal to the surface is estimated using the methods described in Section 3.6.

## 3.6 Calculation of the Normal to the Surface

A significant problem with the cellular model which is used to implement the surface is facet formation. The surface consists of many small faces at 90 degree angles to one another. The two dimensional view of the surface which appears as a series of straight lines can be seen in Figure 3.6. Particles can, in general, hit either the vertical or horizontal faces, or the



corners which join the surface cells. Since the amount of material removed from the surface

highly depends on the impact angle of the abrasive particles, estimating the accurate normal to the surface is of great importance.

To obtain realistic results, the edges joining cells are treated as being part of a plane. For simplicity, collisions on the vertices (i.e. where three surfaces join) of cells are also assumed to be on a plane as well. Finding this plane requires knowledge of how the cells are connected to each other. To obtain this, the neighbors of the cells must be traced. To save computational time, instead of tracing the hierarchical data structure used to store cells (Section 3.10.2), the model uses a two dimensional array which stores the depth location of the surface cells. After estimating the surface plane, to assign damaging incident energy to the cell with the hit edge, the material removed from the cell is calculated; depending on the erosive system type either Eq. (3.49) or Eq. (3.50) is used by taking $\theta_i$ as an angle between the velocity of the $i_{th}$ particle and a normal to the estimated surface plane.

To estimate a plane, the algorithm searches for four different patterns. The first pattern is demonstrated in Figure 3.10, where it is assumed that an incoming particle has struck the edge $e$. The algorithm starts searching in a direction, $u$, which is outward and perpendicular to the side face containing the impacted edge. It should be noted that the cell adjacent to the impacted cell in the direction $u$ has been removed; otherwise the particle would not hit the edge $e$. If the second cell in this direction has not been removed yet, it is likely that the surface is globally flat, and the edge is treated as a surface parallel to the top face of the cell.

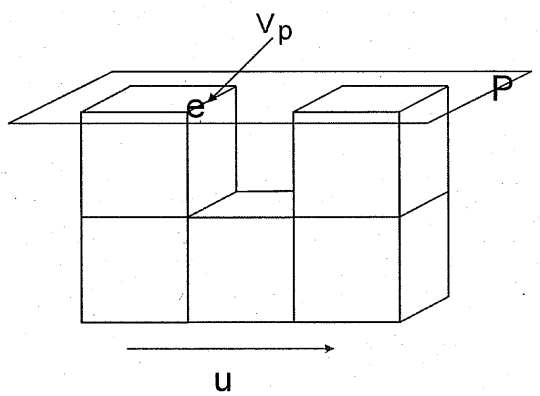



Figure 3.10: The first pattern searched by the algorithm to estimate the surface plane, $P$, when a particle strikes the edge $e$ at a velocity $v_p$.

If the first pattern is not recognized, the algorithm looks for a second pattern, depicted in Figure 3.11. In this case, the algorithm checks for the cell, $c$, below the cell adjacent the impacted edge in the direction $u$. If such a cell exists, the algorithm starts tracing from $c$ in the direction $u$. The tracing stops when the algorithm reaches a removed cell or the cell that has a neighbor on top of it. The surface plane is parallel to both the line connecting the centers of the impacted cell and the last traced cell, and an edge of the impacted cell which is normal to this line (Figure 3.11).

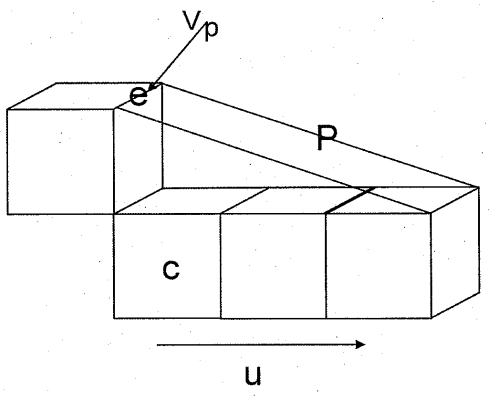



Figure 3.11: The second pattern searched by the algorithm to estimate the surface plane, $P$, when a particle strikes the edge $e$ at a velocity $v_p$.

If none of the previous patterns are recognized, the algorithm looks for two other patterns, shown in Figures 3.12 and 3.13. In both of these cases, the algorithm starts tracing from the impacted cell in the downward direction. The tracing stops when the algorithm reaches a removed cell or a cell that has a neighbor, $c$, in the direction $u$. For the case that a removed cell has been found at the end of the trace, the third pattern is recognized, and the impact is treated as an impact on the face of the cell in the direction $u$, as shown in Figure 3.12. For the case that the algorithm found the neighbor c, the fourth pattern is recognized, and the surface plane is the one parallel to both the line connecting the centers of the impacted cell and $c$, and an edge of the impacted cell which is normal to this line (Figure 3.13).

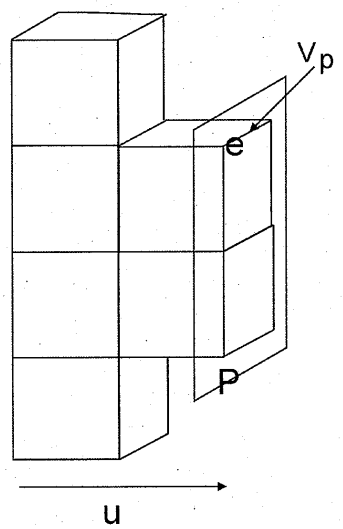



Figure 3.12: The third pattern searched by the algorithm to estimate the surface plane, $P$, when a particle strikes the edge $e$ at a velocity $v_p$.

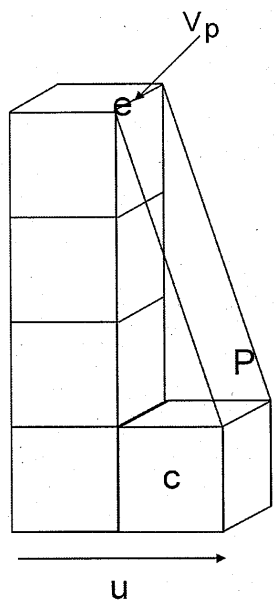



Figure 3.13: The fourth pattern searched by the algorithm to estimate the surface plane, $P$, when a particle strikes the edge $e$ at a velocity $v_p$.

## 3.7 General Description of the Algorithm

In general, events can be described as any occurrences of interest which cause a change in the state of the system. Events handled by the present simulation include the launching of a new particle, the collision between two particles, the collision between a particle and the surface, etc. One way to simulate such a system is using a time-step approach. In this approach, the system advances over the fixed time interval, $\Delta t$. At the end of each step, the system is examined to detect the events that have occurred during the step. For example, to detect the inter-particle collisions, one can check if the distance between the center of any two particles is less than or equal to the sum of their radii. This is also called exhaustive simulation since the system is examined at every time-step, regardless of whether any event occurred during the time-step. One problem with this approach is that it is possible to miss the events that occur during the time interval but leave no evidence at the end of it. For example, if two particles fully pass through each other over a time step, their collision can not be detected by this approach. Decreasing the time step reduces the probability of missing events, but increases the computational time. Another problem with a time-step approach is how to handle the particles that overlap with each other, or the target surface. To overcome this, one could move back the whole system to the maximum time at which there is not any overlapping, but this significantly increases the computational time.

A different approach to simulate the system is an event-driven one. In this approach, instead of fixed time intervals, the system advances from event to event. At the end of each step, the system is explored, and the upcoming event with the nearest time to the current is detected. The system then advances to the time of the predicted event, and the event is handled and executed. This approach has the advantages that no events are missed, and it also avoids the problem of overlapping particles. However, the disadvantage of this approach is that it requires a large memory usage, in order to store all the events. Also, the scheduling overhead makes the propagation of the system slow; i.e. it requires traversing the list of possible future events to find the one with the smallest time.

The present simulation is based on the event-driven approach. All the particles in the

system are given initial velocities and positions and travel in a straight line; therefore, their future behavior is predictable. There are four events that are handled by the model: Launching a new particle, a collision between two particles, a collision between a particle and the surface, and transfer; i.e. the space holding all the particles is divided into smaller space, when a particle enters a new space (see Section 3.9). For each type of event, a class is implemented which provides all the methods needed to predict and handle the event.

The *Nozzle* class is used to predict and handle particle launching events. The *EventPP-Collision* and *EventPSCollision* classes are used to predict and handle the collisions between two particles and the collisions between a particle and the surface, respectively. The *Event-Transfer* is used to predict and handle transfers. The information of predicted events are stored in a queue called the *event queue.* The event queue contains one element for each particle in the system, called the *particle node*, and one element for storing the time of launching the next particle, called the *launch node.* At the beginning of the simulation, there are no particles in the system and the event queue has only one element which is the launch node.

The general schematic representation of the algorithm can be seen from the flowchart of Figure 3.14. The simulation develops by performing the following steps:

S1. Select the next event in the event queue.

S2. Advance the time of the system.

S3. Handle the event.

S4. Update the event queue.

S5. Return to step S1.

To perform step S1, the algorithm searches the event queue and finds the earliest event. Step S2 consists of changing the time of the simulation to the time associated with the event. The time of the event is always greater than or equal to the time of the system. The equality is for handling the exactly simultaneous events which are very unlikely.

In step S3, depending on the type of the event, the appropriate handler is invoked and updates the state of the system. If the event is the launch of a new particle, a particle is created by instantiating from the *Particle* class and is given an initial position and velocity in the nozzle. If the event is a collision between two particles, their velocities are updated using the method described in Section 3.3. If the event is a collision between a particle and the surface, the velocity of the particle is changed, using the method described in Section 3.4. If it is necessary, the algorithm also removes a cell from the substrate, using the method explained in Section 3.5, and updates the shape of the surface (Section 3.10.2). Handling of the transfer requires changing the space containing the particle (Section 3.9).

After performing the step S3, as a result of the updating of the system to reflect the earlier event, the event queue might contain some events that will not occur. In step S4, the algorithm identifies the elements containing such invalid events and updates their information. For example, particles involved in an inter-particle or a particle-surface collision are given new trajectories, therefore their information stored in the event queue is not valid any more. For these events, step S4 consists of predicting the next events of these particles and updating their associated elements in the event queue. For the particles impacting the surface, the algorithm also determines the particles that have such particles as a partner for their next inter-particle collision, and updates their next inter-particle collision. If the event that was just handled is the launch of a new particle, in step S4, the event queue needs to be extended. In this case, the algorithm predicts the next events of the new particle, saves them in a new particle node associated with the particle, and adds the element to the event queue. Also, the algorithm computes the time of launching the next particle, and the launch node has its data updated. Updating the event queue after handling the transfer is explained in Section 3.9.

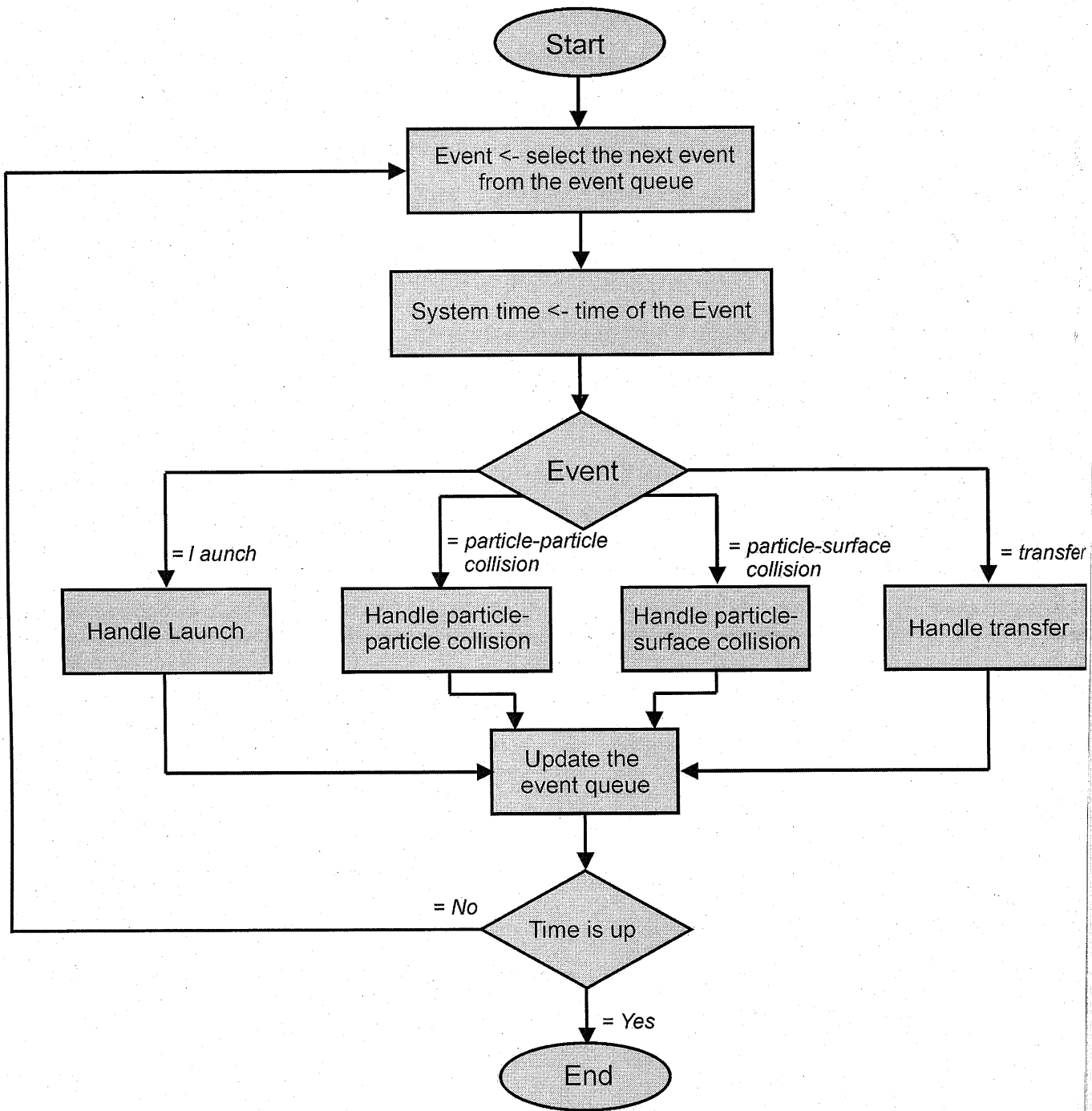



Figure 3.14: Flow chart of the general event-driven algorithm

## 3.8 Implementing the Event Queue

A heap is a complete binary tree in which every element has a key value. Depending on the ordering of the keys, a heap can be called either a max-heap or a min-heap. The present simulation uses a min-heap data structure to save the list of the computed times of particles events in the event queue. In a min-heap, a key of each element is smaller than any of its descendants if they exist, and thus the element with the smallest key is the root of the heap.

The event queue is implemented in the *EventHeap* class. This class provides methods to perform the heap operations, such as inserting a new element, deleting an element, and updating the position of an element. The nodes of the event queue, the particle nodes and the launch node, are instantiated from the *EventNode* class. For $n$ particles inside the system, the event queue has exactly $n + 1$ nodes. The key value of a particle node is the time of its earliest event, and the key value of the launch node is the launch time of the next particle. *EventNode* has a field of type *Event* which is instantiated for the particle nodes and is set to the null for the launch node. The *Event* class is used to store the information of the particle's next events. It has three inner classes: *PPCollisionStorage*, *PSCollisionStorage*, and *transferStorage*. The *PPCollisionStorage* class is used to store the information of the next inter-particle collision of the particle. The *PSCollisionStorage* class is used to store the information of the next particle-surface collision of the particle. Finally, the *transferStorage* class is used to store the information of the next transfer of the particle. For each particle entering the system, the algorithm computes the times of its next particle-particle collision event, particle-surface collision event, and transfer event, and initializes the instances of *PPCollisionStorage*, *PSCollisionStorage*, and *TransferStorage*.

The smallest computed time is used to adjust the key value of the particle node, then the particle node is added to the event queue. For the $n$ particles inside the system, the cost of inserting an element into the event queue is $O(log(n))$. For each particle leaving the system, its particle node is removed from the event queue. The cost of deleting an element from heap is $O(log(n))$. To find the next event, the algorithm simply selects the root of the event queue which costs $O(1)$. When the key value of an element in the event queue is changed,

its position in the event heap is updated costing $O(log(n))$.

## 3.9 Inter-particle Collision Detection

The inter-particle collision detection algorithm used in the model is based on that proposed by Alder and Wainwright [26] and developed and analyzed by Sigurgeirsson et al. [27]. As described in Section 3.1, particles are assumed to be spherical and move in straight paths in the absence of collisions. Consider two particles $P1$ and $P2$. Their positions at time $t$ can be described as,

$$x_1(t) = x_{01} + v_1 t \tag{3.51}$$

$$x_2(t) = x_{02} + v_2 t \tag{3.52}$$

where $x_{01}$ and $x_{02} \in R^3$ are the positions of $P1$ and $P2$ at time 0, and $v1$ and $v_2 \in R^3$ are their velocities. If the distance between the center of the two spherical particles is less than or equal to the sum of their radii, then a collision has occurred. If $P1$ and $P2$ are not overlapping at time 0, the following condition must be applied for them to collide at time $t$.

$$|x_1(t) - x_2(2)| = R \tag{3.53}$$

where $R$ is sum of their radii. Squaring the both sides of the Eq. (3.53),

$$\|\Delta v\|^2 t^2 + 2(\Delta v.\Delta x)t + \|\Delta x\|^2 = R^2 \tag{3.54}$$

If the solution of Eq. (3.54) for time, t, has only one positive solution, the solution is the time of the next collision, and if it has two positive solutions the smaller one is the time of the next collision. If it does not have any positive solution, then the two particles will not collide.

At any time, a finite number of particles, $n$, are inside the system. One way to detect the next particle-particle collision in the system is to compute the time of the next collision between any two particles, using Eq. (3.54), and select the nearest time. In this way $n(n+1)/2$ collision tests must be performed, and particle-particle detection requires $\Theta(n^2)$ calculations. Alder and Wainwright [26] indicate that storing the calculated collision times causes a great saving in computational time. In this manner, since after each collision the trajectories of only two particles are changed, the inter-particle collision test must be performed only on pairs including one or two of the particles involved in the collision. Therefore, the number of tests reduces to $(2n - 3)$. However, computing the very first collision time still requires the tests on every pair. The issue of how to store the list of computed times is addressed by Sigurgeirsson et al. [27]. They suggest using a heap data structure, the method which is used in the current simulation as well. The implementation of the heap is explained in Section 3.8.

Alder and Wainwright [26] noted that distant particles are very likely to change their direction before they collide. They suggest dividing up the cubic space holding all particles into smaller cubes, called *cells* [27], and detecting collisions only between particles in neighboring cells. Implementing this idea requires detecting and handling transfers between cells. The present simulation uses a three dimensional array to implement the environment containing all the particles, as suggested by Sigurgeirsson et al. [27]. Each element of the array represents a cell of a grid, and is instantiated from the *Cell* class. The *Cell* class has a list of particles, called *members*, which are assigned to it. In the algorithm proposed by Alder and Wainwright [26], each particle is assigned to a unique cell which contains the center of the particle, while in the implemented algorithm, each particle can be assigned to more than one cell. The particles are assigned to the cells that contain them. Also, particles are added to the members of cells in which they are about to enter. Particles that are assigned to the same cells are called *neighbours*.

To predict the next collision time for a particle, instead of calculating the collision times between the particle and all the particles in the neighboring cells, only the collision times

between the particle and its neighbors are calculated. Using this method the algorithm looks for inter-particle collisions in smaller spaces, speeding up the particle-particle detection procedure (see Figures 3.15(a)-3.15(c)). However, this increase in speed is at the expense of implementing a mechanism to monitor the cells in which particles are about to enter.

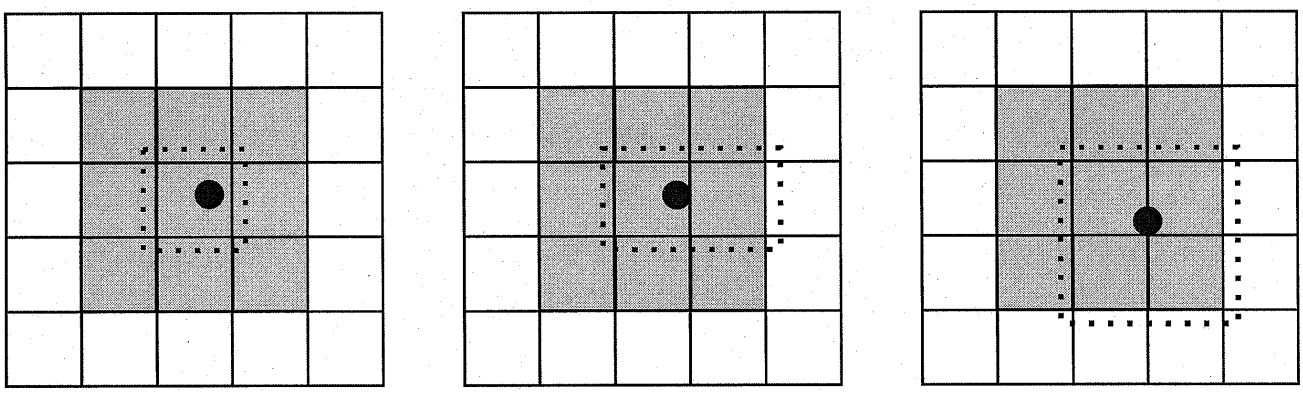



Figure 3.15: Three different examples are demonstrated which compare the inter-particle collision detection approach used in the present simulation with the one proposed in [26]. According to the Alder and Wainwright model, to predict the next collision of the black particle, collision times between this particle and particles whose centers are within the gray cells are calculated [26]. According to the present model, to predict the next collision of the black particle, collision times between this particle and particles whose centers are within the dotted region are calculated

For each detected inter-particle collision, the two involved particles are referred to as *partners* of each other. For each neighbor, a computed time is compared to both the smallest computed time so far and the collision time between the neighbor and its current partner, if any, and is accepted if it is smaller than both. After computing all collision times, if a particle finds a partner, their next collision times are adjusted. If the new partner has an old partner, its next collision time is invalid, and the algorithm finds the next collision time between this particle and all of its neighbors, except for its old partner. For each particle-

particle collision detection, the algorithm stops when a particle cannot find any partner, or the new partner does not have an old partner.

To predict the next transfer for particles, the algorithm computes the intersection of the trajectory of a particle and the faces of a cube and selects the earliest one. To handle the transfer, the algorithm moves the particle to the closest position at which its surface is tangent to the face of the new cells. Then the particle is removed from the members of old cells that do not contain it anymore, and added to the members of new cells; i.e. this ensures that before the particle enters any cell its inter-particle collision within that cell is detected. The algorithm then detects the next inter-particle collision of the particle and updates its element in the event queue. For the cases where there is no cell adjacent to the exit face, the particle is leaving the boundary of the system. In this case, the algorithm removes the particle from the members of its old cells and removes its associated element from the event queue. After each transfer, the next collision for the particle's old partner, if any, is invalid and must be updated. For a particle that is still in the system after handling the transfer event, since its trajectory has not been changed, there is no need to update its particle-surface collision.

As material is removed from the initially flat surface, the space holding particles extends. To allow particles to flow into the newly eroded areas and allow collisions to occur within these areas, the cells which are adjacent to the surface are not cubic. Instead they are cuboids with the same width and height as the other cells but their depth goes all the way down to the end of the substrate.

### 3.9.1 Optimal Cell Size

As the size of the environment cells decreases, the number of transfers increases, and a smaller number of particles needs to be examined in order to detect inter-particle collisions at each event. On the other hand, increasing the cell size decreases the number of transfers and requires more pairs of particles examined at each event. Since the cell size affects the performance of the simulation, it is important to find an optimal cell size.

To see how the choice of cell size affects the performance of the simulation, different values of launched frequency were set, from 200,000 to 2,000,000 in increments of 200,000. For each value of launch frequency , the simulation runs were conducted with a varying environment cell size, while holding all the parameters at the values given in Table 3.1. Figure 3.16 shows how the run time of the simulation with a launch frequency of 1,000,000 particles/s changes with the environment cell size. As can be seen, for $f_n = 100000$, the environment cells with an edge length around 0.8 $mm$ give the best performance. A similar approach was used to determine the optimal cell size for other launch frequencies. The linear regression on optimal values gives the optimal edge length, $l_e$, in terms of launch frequency, $f_n$, as,

$$l_e = -3.266666 \times 10^{-7} f_n + 1.15 \tag{3.55}$$

Figure 3.17 shows the optimal edge lengths at each launch frequency and the fitted line.

| input | value | input | value |
|---|---|---|---|
| $v_{noz}$ | 0.0 $mm/s$ | $r_n$ | 0.38 $mm$ |
| $\Gamma$ | 0 | $L$ | 14 $\mu m$ |
| $d$ | 20 $mm$ | $\beta$ | 15 |
| $r_p$ | 25 $\mu m$ | $\rho_p$ | 4000 $kg/m^3$ |
| $V_{max}$ | 162 $m/s$ | $\rho_s$ | 2200 $kg/m^3$ |
| $K$ | 1.43 | $D$ | $6.3 \times 10^{-6}$ |
| $e_{pp}$ | 1 | $e_{ps}$ | 0.5 |
| $f$ | 0.0 | | |

Table 3.1: The inputs to the simulation used to estimate an optimal environment cell size.

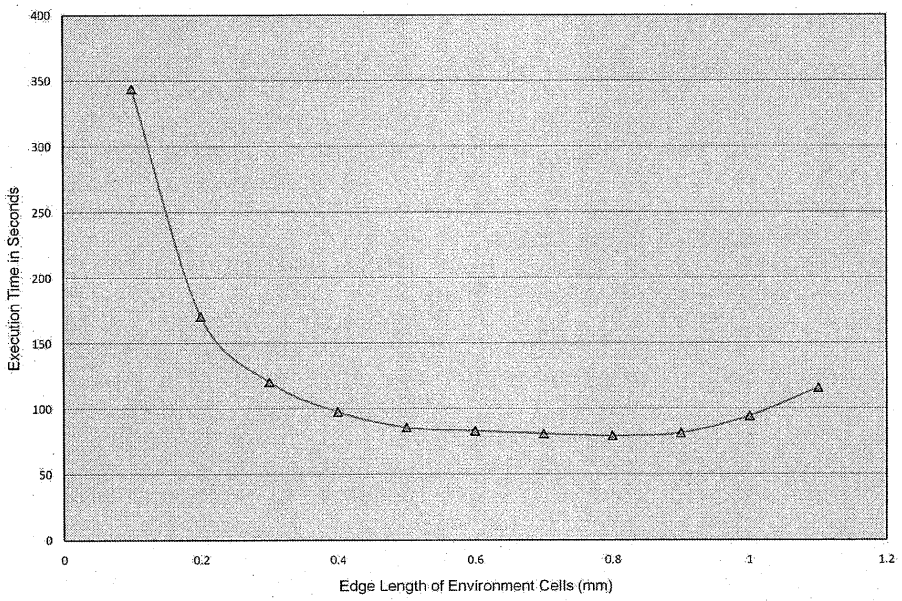



Figure 3.16: Execution times for different environment cell sizes using $f_n = 1000000$, with all other parameters at the values given in Table 3.1

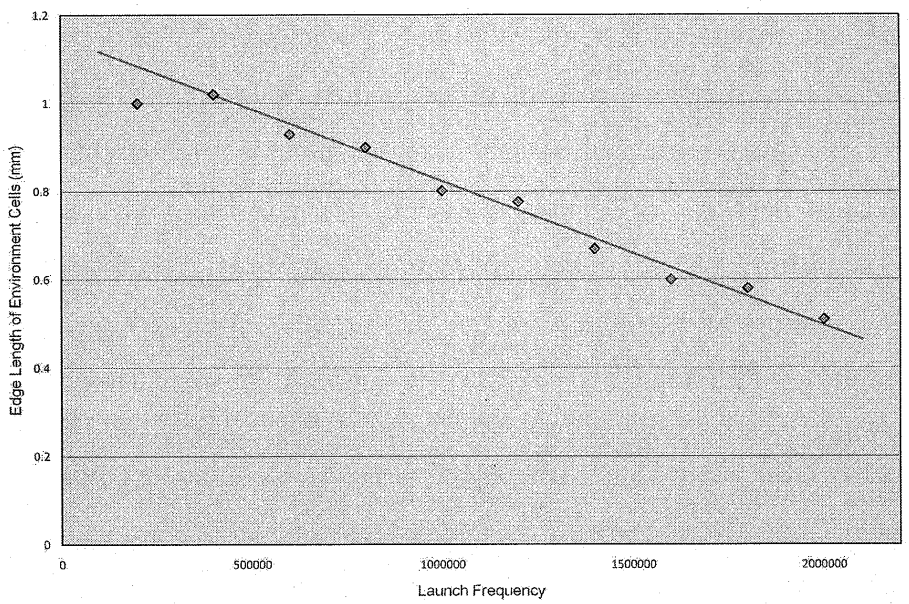



Figure 3.17: Optimal cell sizes for different launch frequencies using the input parameters from Table 3.1 and the fitted line.

## 3.10 Modeling the Surface Advancement

As mentioned in Section 2.2.4, the present model uses a cell-removal algorithm to model the surface advancement. For the cellular algorithms implemented in the present simulation, a substrate was represented by a cube which was divided into very small cubes of equal size, henceforth referred to as *cells*. Collision detection between a spherical particle and the cellular surface requires a large number of geometrical intersection tests; i.e. checking whether the sphere will touch any of the polygons used to model the surface. The algorithm cannot stop once it detects a single collision; it has to find all possible collisions and choose the nearest one. To reduce the number of tests, a hierarchical representation of the substrate is generated. The present simulation uses a hierarchical data structure, called a *cellular octree*(section 3.10.2), which is based on the *region octree* data structure [28].

### 3.10.1 Octree Data Structure

The term octree [28] refers to a class of hierarchical data structures which are based on a recursive subdivision of a space. The simplest octree is called a region octree which is based on the successive subdivision of a bounded volume into eight octants of equal size [28]. To represent an object by a region octree, the object is defined by a three-dimensional array of 1's and 0's, a 1 indicating the unit cube is contained in the object, and a 0 indicating the unit cube is outside the object; i.e. a 1 represents the presence of the object and a 0 represents the absence of the object. The subdivision process is represented by a tree in which non-leaf nodes have exactly eight children and a root node represents the entire object. In general, a node needs only to be subdivided into octants if it does not consist entirely of 1's or entirely of 0's. The subdivision process starts from the entire array and continues until cubes are gained that either contain only 1's or only 0's; these cubes represent the leaf nodes of the tree, and are fully within or outside the object. The leaf nodes representing cubes within the object are called black, the leaf nodes representing cubes outside the object are called white, and non-leaf nodes are called gray. Figures 3.18(a)-(c) show a three-dimensional object, its octree block decomposition, and its tree representation, respectively [28].

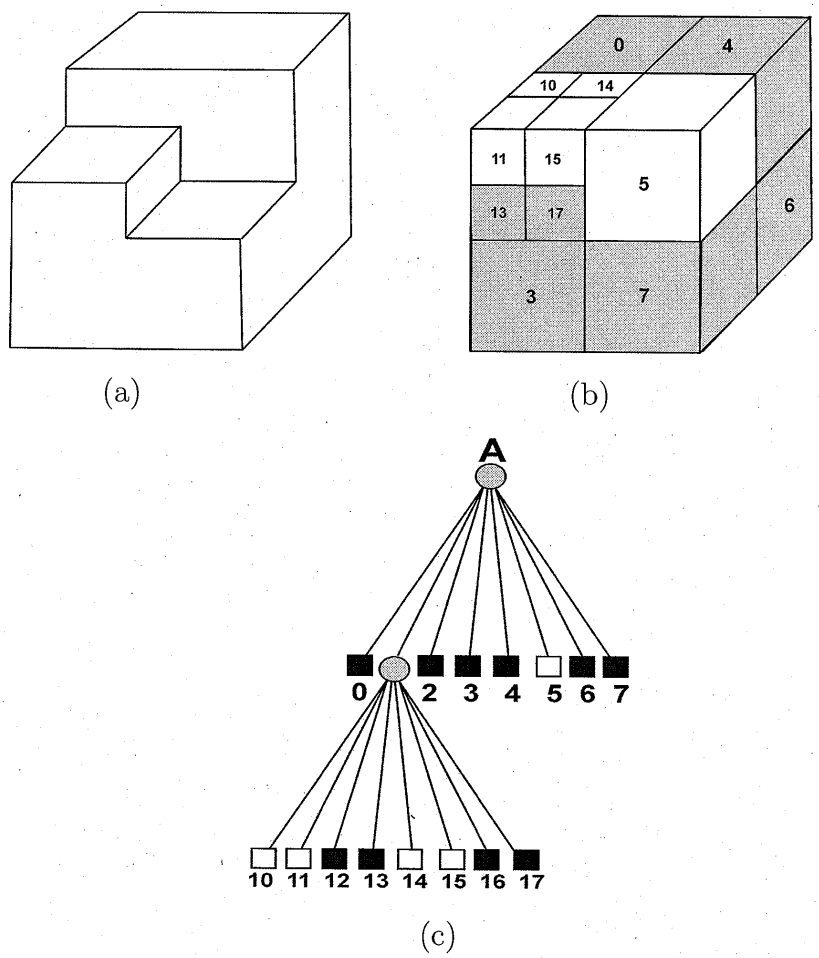



Figure 3.18: (a) A three dimensional object, (b) its octree block decomposition, (c) its tree representation

### 3.10.2 Surface Advancement Algorithm

The cellular octree is defined based on the region octree, and it is used to model the target substrate. The model implements the tree representation of the cellular octree using a top-down approach. The root of the tree represents the entire substrate. The leaf nodes are called white, black, or unit, and they do not need further subdivision. White nodes correspond to the cubes that are fully outside the substrate; i.e. they are no longer part of the substrate. Black nodes correspond to the cubes that are fully within the substrate and have not been

hit by any particle; i.e. they represent the presence of the substrate. Unit nodes correspond to the cubes that are fully inside the substrate, are the same size as surface cells, and have been hit by at least one particle. The corresponding cubes of unit nodes are referred as unit cubes and they represent the cells of the surface. A maximum level of subdivision of the cellular octree is determined by the size of the surface cells which is fixed beforehand (i.e. as an input to the simulation). The non-leaf nodes are called gray or cellular. The gray nodes correspond to the cube that are partially inside the substrate. The cellular nodes correspond to the cube that are fully within the substrate, have been hit by at least one particle, and are larger than the surface cells.

All the nodes of the tree are instantiated from the *OctNode* class. This class is used to store the information of the corresponding cubes of nodes, such as position, index which determines the label of the node with respect to the parent (see Figure 3.19), along with references to its octants. The *OctNode* class has an array of eight elements, called *child*, which is used to store the references to the octants, and for the leaf nodes, since they do not need further subdivision, it is set to null. There is also a field in *OctNode* class called *loss*. This field is only used for unit nodes, and depending on the type of the erosive system, brittle or ductile, it stores the value obtained by Eqs. (3.49) or (3.50). The unit nodes whose loss value is equal or greater than 1 must be removed.

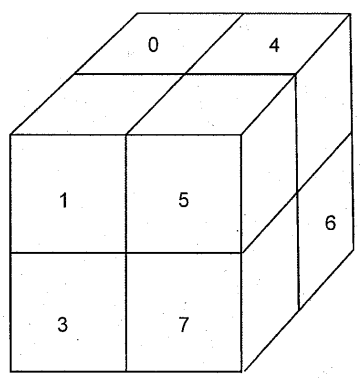



Figure 3.19: Labeling convention of octants in a cellular octree

Constructing the substrate by creating all of its cells is a costly process in terms of time and memory. On the other hand, the spatial distribution of particles within the jet (Eq. 3.5), implies that there are many more particles near the nozzle centerline than at the periphery of the jet. Thus, in reality, it is likely that there are parts of the surface on the periphery of the jet of colliding particles that are rarely hit by any particles. To avoid creating unnecessary cells in these areas, the algorithm allows for the decomposition to occur only on parts of the surface that are actually hit by particles.

For simplicity, it is assumed that an initially cubic substrate with a flat surface is exposed to the particle jet. At the start of the simulation, the tree has only a root which is black and represents the entire substrate. In general, after each particle-surface collision, the algorithm determines the leaf node, and if the leaf node is not white, it decomposes the corresponding cube of the leaf node until the unit cube containing the collision point is obtained. Then the collision is handled. Once the leaf node containing the collision point is determined, three possible cases arise. The first case is when the leaf node is unit. In this case, there is no need for further subdivision and the collision is handled. The second case is when the leaf node is black. In this case, the status of the node is changed to unit or cellular depending on whether its corresponding cube is the same size as the unit cubes or larger, respectively. If the node has been changed to a unit node, no further subdivision is necessary and the collision is handled. If the node has been changed to a cellular node, the algorithm divides the cube into octants, and sets the status of the octants to black. Then the algorithm repeats the decomposition process from the octant containing the collision point until it reaches the unit cube containing the collision point, and then the collision is handled. The third case is when the leaf node is white. In this case, the target cell has been removed (due to impacts from some other particles) sometime after the particle-surface collision was detected for this particle. These types of events are called *unsuccessful collisions*. Unsuccessful collisions are not handled and just simply ignored by the algorithm.

As an example, consider Figure 3.20. Figures 3.20(a) and (b), respectively, show a black leaf node and its corresponding cube which will be hit by a particle at the point shown by

the red cross. After the collision, the black leaf node turns to a cellular node whose tree representation and corresponding cube are shown in Figures 3.20(c) and 3.20(d).

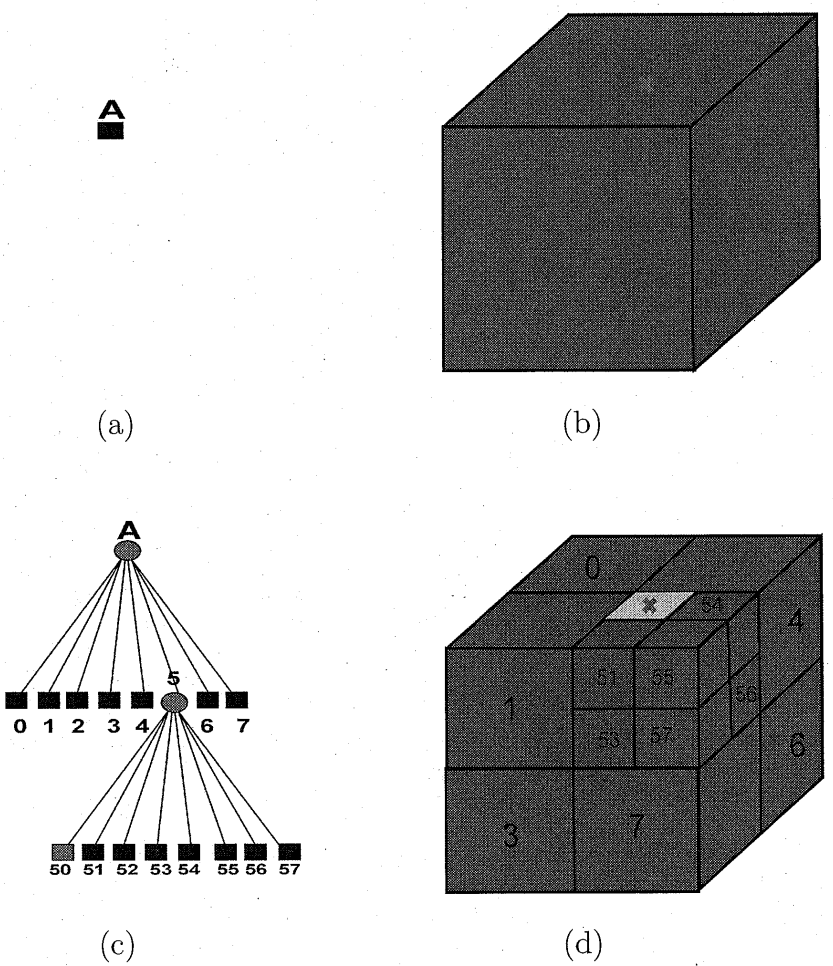



Figure 3.20: (a) A black leaf node, (b) its corresponding cube. (c) The tree representation of the node after the collision, (d) its block decomposition.

To handle the collision, the velocity of the particle involved in the collision is changed, the value of the *loss* variable of the unit cell containing the collision point is updated and the cell is removed if one of the conditions given by Eq. (3.49) or (3.50), depending on the erosive system, is satisfied. Removing a cell is performed by changing the status of the corresponding unit node to white. Once the unit cube is removed, the status of its corresponding node is

changed to white, and the statuses of all of its ancestors are changed to gray.

When a cell is removed, there may still be some particles that have the cell as the target for their next particle-surface collision. So their detected particle-surface collisions are not valid anymore. Finding these particles requires checking the target cells for all particles colliding with the surface at each cell removal, which is a costly process. To avoid going through this process, when a cell is removed, the algorithm does not try to find such particles. Before handling each particle-surface collision, the algorithm checks for the target cube containing the collision point and if its status is white, a unsuccessful collision occurred and it will thus not be handled. Since these particles may go on to hit another cell later, in the step of updating the event queue, the algorithm predicts their next particle-surface collisions. Because they are not given the new trajectories, there is no need to update their transfers and inter-particle collisions.

The size of the cellular octree is reduced by merging groups of eight siblings of the white color. To perform the merging process, the eight siblings are removed, the status of their parent is set to white, and its child array is set to null. Once a unit cell is removed, the algorithm performs a check for a possible merge. If a merge occurs, it checks for another possible merge in a higher level of the tree. [28]

The pseudo-code of the surface advancement algorithm is presented below.

Notations:

- *SurfaceAdvancement(particle, collisionPoint)*: obtains the unit cube that contain the *collisionPoint* and then handles the collision between the *particle* and the unit cube.
- *Status(node)*: returns the status of a node.
- *LeafNode(point)*: returns the leaf node whose corresponding cube contains *point*.
- *CubeSize(Node)*: returns the size of the corresponding cube of the *Node*.
- *decompose(Node)*: creates eight octants of the *Node*; the decomposition of a node is

performed by eight times instantiating from *OctNode* class and setting the elements of the child array of the node to these instances.

- *Octant(i,node)*: returns the $i^{th}$ octant of a node.

**Algorithm 1** Surface Advancement Algorithm

**procedure** SurfaceAdvancement(particle, collisionPoint)

```
1.  leafNode ← LeafNode(collisionPoint)
2.  foundUnitNode ← false
3.  while (foundUnitNode=false)
4.      case Status(leafNode) of
5.          Unit:
6.              foundUnitNode ← true
7.              handleCollision(particle, leafNode)
8.          Black:
9.              if CubeSize(leafNode) = unitSize then
10.                 leafNode.setStatus(Unit)
11.                 foundUnitNode ← true
12.                 handleCollision(particle, leafNode)
13.             else
14.                 leafNode.setStatus(Cellular)
15.                 decompose(leafNode)
16.                 i ← 0
17.                 while (i < 8)
18.                     if Octant(i, node).contains(collisionPoint) then
19.                         leafNode ← Octant(i, node)
20.                         exit loop
21.                     i ← i + 1
22.                 end while
23.             end if
24.         White:
25.             foundUnitNode ← false
26.             exit
27.     end case
28. end while
29.
```

## 3.11 Particle-Surface Collision Detection

The particles that are far from the surface are likely to change their direction before they hit the surface. To avoid detecting particle-surface collisions for these particles, the algorithm detects particle-surface collisions only for the particles that are contained in the environment cells adjacent to the surface. The collision detection algorithm used in the model, requires intersection tests between trajectories of particles and corresponding cubes of the cellular octree. A particle is represented by a sphere having a certain radius, and whose center is located by cartesian coordinates. A correspoding cube of a node of the cellular octree is represented by a cube having a certain edge, and whose center is located by cartesian coordinates. Before describing the general algorithm, the approach used to detect collisions between a sphere and a cube will be presented.

### 3.11.1 Sphere-Cube Collision Detection Algorithm

Detecting collisions of objects with a sphere is relatively easy to calculate because of its symmetry. All the points on the surface of a sphere are the same distance from its center, so it is easy to determine whether or not an object will intersect with it. Therefore, for each cube, a bounding sphere is defined as a sphere whose surface contains all the vertices of the cube. The algorithm performs the intersection test between the sphere representing the particle, and the bounding sphere of the cube by calculating the following equation,

$$\|v_p\|^2 t^2 + 2(v_p.\Delta x)t + \|\Delta x\|^2 = R^2 \tag{3.56}$$

where $v_p$ is the particle velocity, $\Delta x$ is difference between the spheres centers, and $R$ is sum of their radii. If the equation has no positive solution, the sphere will not collide with the cube, and the algorithm does not go through the further calculation to detect the sphere-cube collision. Otherwise, if the equation has at least one solution there is a possibility that a collision occurs, and the algorithm checks for a collision between the cube and the sphere. A cube is bounded by six square faces. The collision detection algorithm checks the sphere

against the faces of the cube. Depending on the position of the sphere with respect to the cube, at most three faces are examined by the algorithm (i.e. if the sphere is on the left side of the cube, the collision with the right face is impossible). The problem now is reduced to finding the collision between the sphere and one of the square planes. When a collision is detected, the position where the sphere hits the cube and the time of the collision are needed to handle the collision. Three different cases are examined for each collision detection:

- The sphere collides inside the square plane.
- The sphere collides with one of the vertices of the square plane.
- The sphere collides with one of the edges of the square plane.

The case of the collision inside the square plane is checked first, and if such a collision is detected the algorithm skips the tests for edges and vertices. If the sphere does not collide with the inside of the square, collisions with the vertices and the edges are detected, and the earliest one is selected.

### 3.11.2 Particle-Surface Collision Detection Algorithm

The particle-surface collision detection is achieved by traversing the cellular octree in a top-down fashion applying a set of rules. The white nodes are ignored, as they represent the absence of the substrate. For each non-white node, the algorithm searches for the collision between the particle and the corresponding cube of the node, using the approach explained in section 3.11.1. If there are no collisions, the particle will not collide with the substrate bounded inside the cube, and the algorithm does not search for collision in the further subdivisions of the cube. If the particle is going to collide with the cube, two cases can arise. The first is when the node is black, cellular or unit. Since the collision point is contained inside the substrate, the collision is detected. The other case is when the node is gray. In this case, the collision may be invalid, depending on which area of the cube contains the collision point. To determine whether the collision is valid or not, the algorithm obtains the leaf node, whose corresponding cube contains the collision point. If the leaf node is black or

unit, the point is contained inside the substrate, and the collision is detected. If the node is white, the collision is not acceptable and the traversal of the cellular octree proceeds; i.e. the algorithm goes through the octants of the node, finds the collision against each octant, using the same process, and accepts the earliest one, if any.

The pseudo-code of the collision detection algorithm is presented below.

Notations:

- *Status(node)*: returns the status of a node.
- *Octant(i,node)*: returns the $i^{th}$ octant of a node.
- *DetectCubeCollision(node, particle)*: checks for a collision between a particle and the corresponding cube of a node and returns a variable that has two parts: *collision* which is *true* if the collision is detected, *contactPoint* at where the particle collides with the cube.
- *LeafNode(point)*: returns the leaf node whose corresponding cube contains *point*.

**Algorithm 2** Particle surface collision detection algorithm

```
procedure DetectCollision(node, particle)
1.  if Status(node) ≠ White then
2.      (collision, contactPoint) ← DetectCubeCollision(node, particle)
3.      if collision = true then
4.          case Status(node) of
5.              Black: foundCollision ← true
6.              Unit: foundCollision ← true
7.              Cellular: foundCollision ← true
8.              Gray:
9.                  leaf ← LeafNode(contactPoint)
10.                 if Status(leaf) ≠ White then
11.                     foundCollision ← true
12.                 else
13.                     i ← 0
14.                     while (i < 8)
15.                         DetectCollision(Octant(i, node), particle)
16.                         i ← i + 1
17.                     end while
18.                 end if
19.         end case
20.     end if
21. end if
```

## 3.12 Graphical User Interface

This section presents the graphical user interface (GUI) constructed for the simulation. The GUI enables the user to enter the input parameters and monitor the progress and the actual time of the simulation. The GUI has been implemented as a collection of buttons, labeled text input fields, and a progress bar. Figure 3.21 shows the GUI which includes four panels. The first panel, the Nozzle Specification, includes the input fields used to implement the nozzle which launches the particles. The second panel, the Particles Specification, includes input fields accepting the properties of the abrasive particle material. The third panel, Surface Specification, includes input fields accepting the properties of the target surface. The last panel, the Environment Specification, includes an input field used to determine the size of the environment. The bottom part of the GUI includes a text field accepting the time duration of the simulation which is used as a criteria to stop the simulation. It also includes an input field which determines the interval for creating the results of the simulation. At each specified interval, two Microsoft Excel files are created: CrossSection and 3DProfile. CrossSection includes the partially eroded depth of the surface cells at the mid cross-section of the surface. 3DProfile includes the partially eroded depth of all the surface cells which provides the three-dimensional view of the erosion profile. Table 3.2 gives a brief summary of the input parameters. The GUI has a progress bar implemented to convey the progress of the simulation in terms of time. By clicking on the button run, the simulation starts and a file containing the values of inputs is created. The user can stop the simulation at anytime by clicking on the exit button, and which time previously mentioned Excel files are also generated.

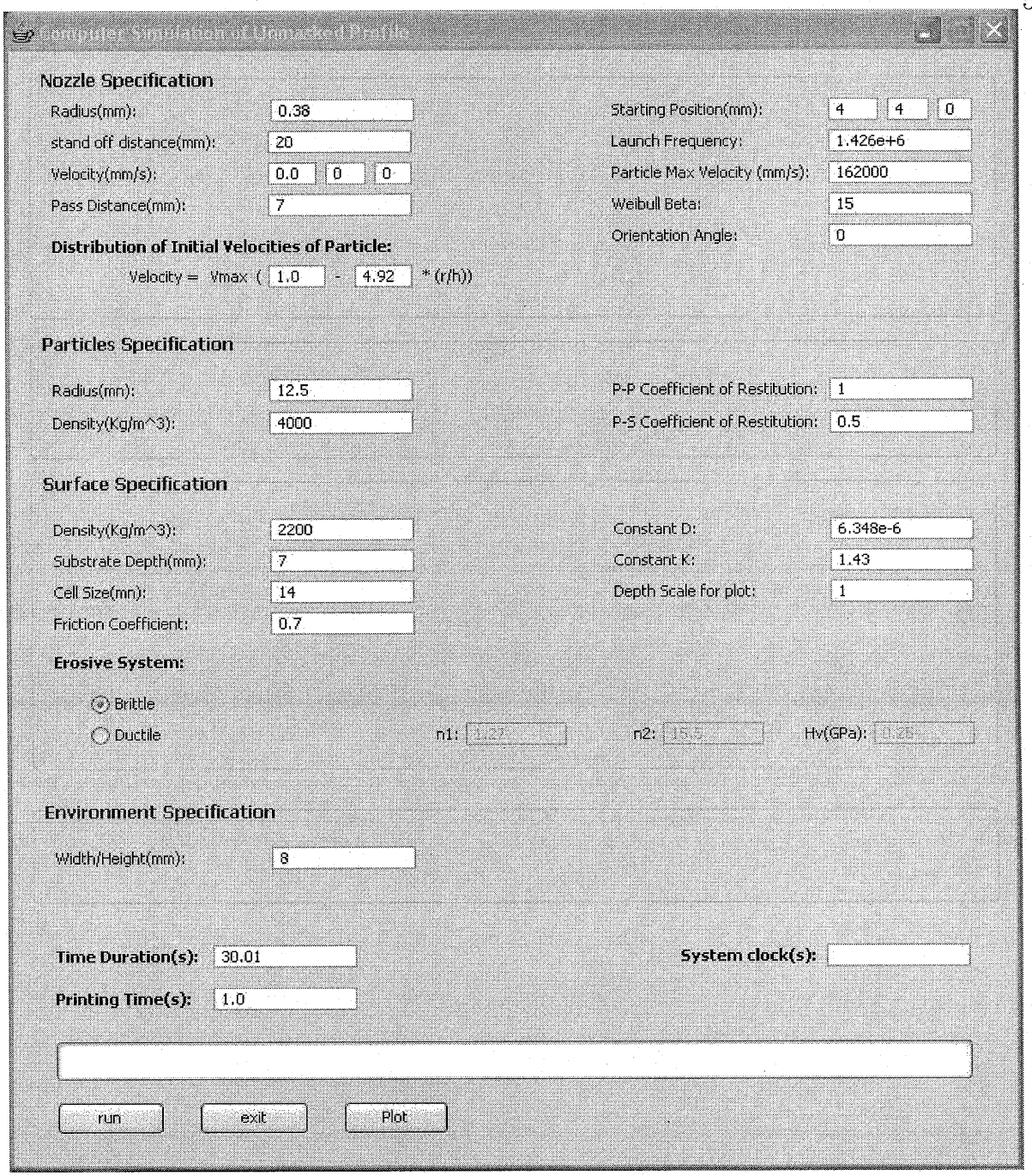



Figure 3.21: The graphical user interface used to enter the input parameters

| *Nozzle Specification Panel* | |
|---|---|
| *Radius* | the radius of the nozzle in $mm$ |
| *Stand off distacne* | the stand-off-distance of the nozzle in $mm$ |
| *Velocity* | the velocity of the nozzle in $mm/s$ |
| *PassDistance* | the distance scanned by the non-stationary nozzle to create channel in $mm$ |
| *Distribution of Initial Velocities of Particles* | the distribution of particles velocities at the nozzle exit plane |
| *Starting Position* | the initial position of the nozzle in $mm$ |
| *Launch Frequency* | the number of particles launched from the nozzle per second |
| *Particle Max Velocity* | the velocity of particles moving along the nozzle centerline |
| *Weibull Beta* | the focus coefficient |
| *Orientation Angle* | the angle by which the nozzle is rotated in counter-clockwise direction about axis x (Figure 3.22) |
| *Particles Specification Panel* | |
| *Radius* | the radius of particles in $\mu m$ |
| *Density* | the density of abrasive material in $kg/m^3$ |
| *P − P Coefficient of Restitution* | the coefficient of restitution for inter-particle collisions |
| *P − S Coefficient of Restitution* | the coefficient of restitution for particle-surface collisions |
| *Surface Specification Panel* | |
| *Density* | the density of the substrate material in $kg/m^3$ |
| *Substrate Depth* | the depth of the substrate in $mm$ |
| *Cell Size* | the edge of the surface cells in $\mu m$ |
| *Friction Coefficient* | the friction coefficient of the target surface |
| *Erosive System* | the type of the eroding material |
| $n_1$ and $n_2$ | the constants used in the cell removal criteria for ductile erosive systems |
| *Hv* | the hardness of the eroding material in $GPa$ |
| *Constant D* and *Constant K* | the constants used for erosion rate |
| *Environment Specification Panel* | |
| *Width/Height* | the width and height of the environment in $mm$ |
| *Time Duration* | the duration of the simulation in $s$ |
| *Printing Time* | the time interval at which the results of the simulation are created in $s$ |
| *System Clock* | the current time of the simulation in $s$ |

Table 3.2: The descriptions of the input text fields implemented in GUI

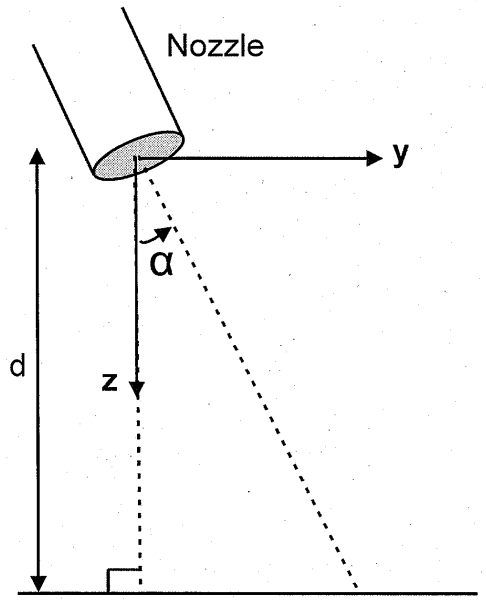



Figure 3.22: The angle $\alpha$ by which the nozzle is rotated in a counterclockwise direction about the x axis, in the y-z plane.

# Chapter 4

# Results and Discussion

## 4.1 Performance of the Simulation

The execution time of the simulation depends on the input parameters. One parameter that significantly affects the execution time is the edge length of the surface cells. To give an idea of the execution time, a simulation run was conducted with the input parameters given in Table 4.1. The environment holding all the particles was initially of size $8 \times 8 \times 20$ $mm^3$. Simulating 0.1 seconds took 24 seconds of execution time on an Intel 2.4 GHz quad processor system with 2030 MB of RAM. It should be noted that the simulation run does not necessarily run at this pace for the whole execution; it often slows down due to the increase in the number of cellular octree nodes, as more cells are removed from the substrate. This slows down the particle-surface collision detection, since a larger number of collision tests between particles and cubes associated with the nodes of the cellular octree must be performed. Moreover, as the surface erodes, the area holding the particles extends and more particles flow into the boundary of the system. Finally an increase in dynamically allocated memory increases the time associated with the Java garbage collection.

To give an idea of how much the simulation is slowed down, a run was conducted for a simulation time duration of 30 seconds. The execution times needed to simulate each time interval of 0.5 seconds were calculated, and are depicted in Figure 4.1. It is clearly evident that the simulation time for each interval increases as the simulation progresses.

| input | value | input | value |
|---|---|---|---|
| $v_{noz}$ | 0.0 $mm/s$ | $r_n$ | 0.38 $mm$ |
| $f_n$ | $1.2 \times 10^6$ | $\alpha$ | 0 |
| $d$ | 20 $mm$ | $\beta$ | 15 |
| $r_p$ | 25 $\mu m$ | $\rho_p$ | 4000 $kg/m^3$ |
| $V_{max}$ | 162 $m/s$ | $\rho_s$ | 2200 $kg/m^3$ |
| $K$ | 1.43 | $D$ | $6.3 \times 10^{-6}$ |
| $e_{pp}$ | 1 | $e_{ps}$ | 0.5 |
| $f$ | 0.0 | $L$ | 14 $\mu m$ |

Table 4.1: The inputs to the simulation

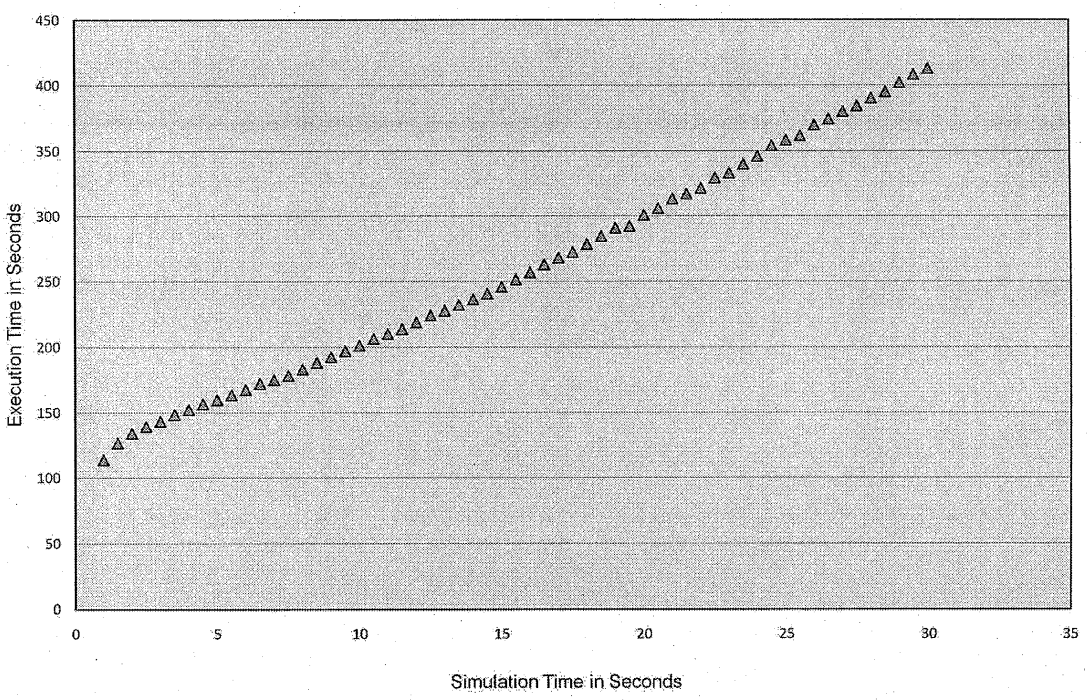



Figure 4.1: Execution times needed to simulate each time interval of 0.5 seconds.

## 4.2 Surface cell size

In cellular models, a more accurate shape of the surface can be obtained by decreasing the size of the surface cells. This is due to the facet formation problem of cellular algorithms described in Section 2.2.4. For example, in the simulation of the machining of a hole, the cellular simulation results in a hole opening which is polygonal, rather than circular. If smaller cells are used, the shape would of course be closer to circular, but, on the other hand, the use of small cells presents two problems. The first is the computational cost, and the second is that using small cells sometimes does not allow particles to do as much damage as they should because each impacting particle can only damage one cell. Thus, the maximum volume of material that can be removed by a particle impacting the surface is equal to the volume of a surface cell. For very small surface cells, it is possible that a particle transfers more energy than it is needed to remove the small target cell. In this case, some of its energy is wasted because the simulation does not provide any mechanism to transfer the remaining impact energy to the other cells. Since using too small of a cell size can result in an incorrect shape and depth of erosion profiles, the cell size should be chosen with caution.

To improve the quality of the plotted surface, the simulation does not plot only undamaged surface cells. Instead, the partially eroded depth of the damaged surface cells is used to plot the profile. This depth can be calculated as,

$$d_p = d + (N \times L) \tag{4.1}$$

where $d$ is the depth (i.e. the distance below the initial uneroded surface) of the cell, $N$ is the value calculated by Eqs. (3.49) or (3.50) for the particular surface cell in question, and $L$ is the edge length of the cell. Using this technique, to some extent, avoids the facet formation problem and gives a more accurate shape for the erosion profile.

Figure 4.2 compares the predicted cross-sections of runs conducted with two different surface cell sizes, cells of edge 14 $\mu m$ and 10 $\mu m$. As described earlier, the shallower profile

for the smaller cells is due to the fact that some of the energy of particles impacting the surface is wasted and is not transferred to the substrate.

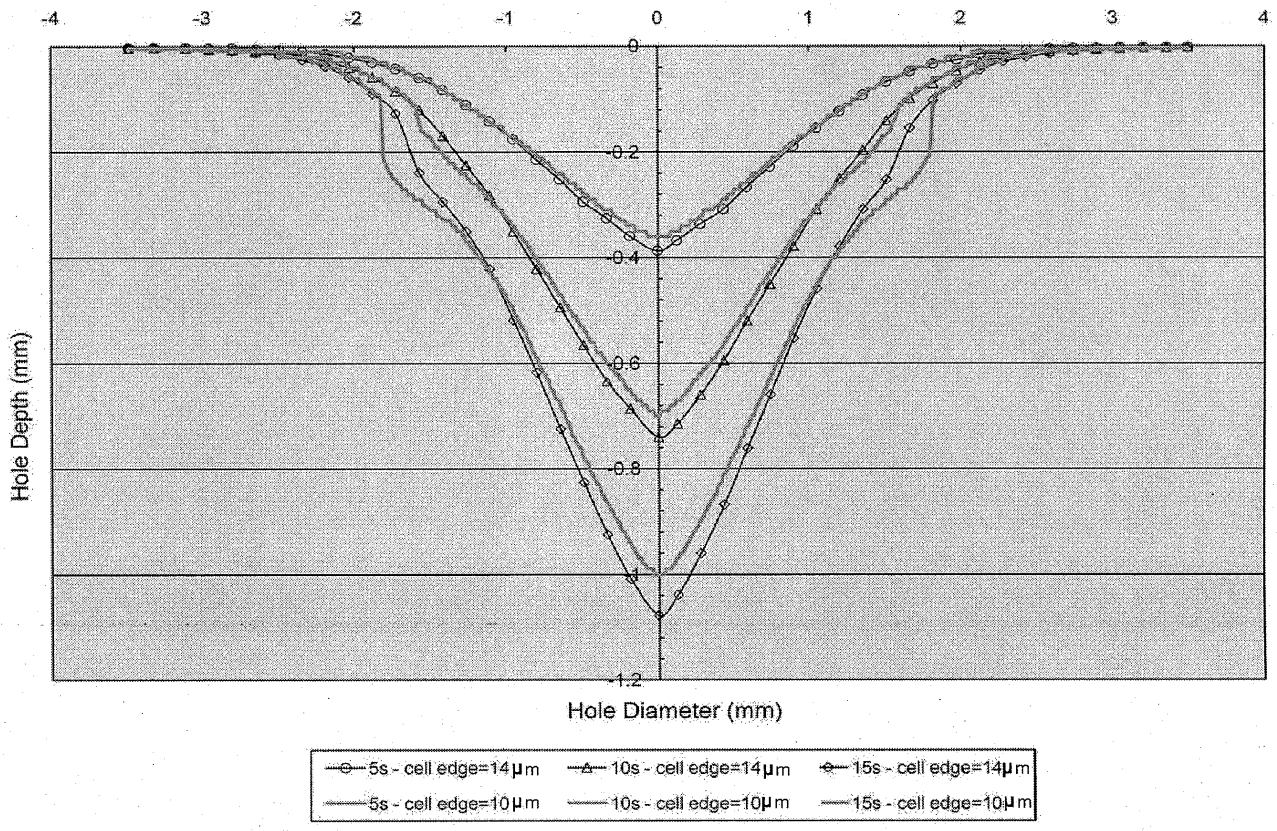



Figure 4.2: Comparison of predicted cross-sections using cells of edge 14 $\mu m$ and 10 $\mu m$, with all other parameters at the values given in Table 4.5.

To select a reasonable value for surface cell size, simulation runs were conducted by varying the value of the cell edge length and holding all the others parameters at the values given in Table 4.3. At each run, the volumes of the target material, $V_w$, associated with the total wasted impact energy after 1 second were calculated. Figure 4.3 demonstrates the effect of cell size on the value of $V_w$. Values between 13 and 15 $\mu m$ are considered reasonable, since they are the smallest values that do not cause considerable effects on the volume of material removed.

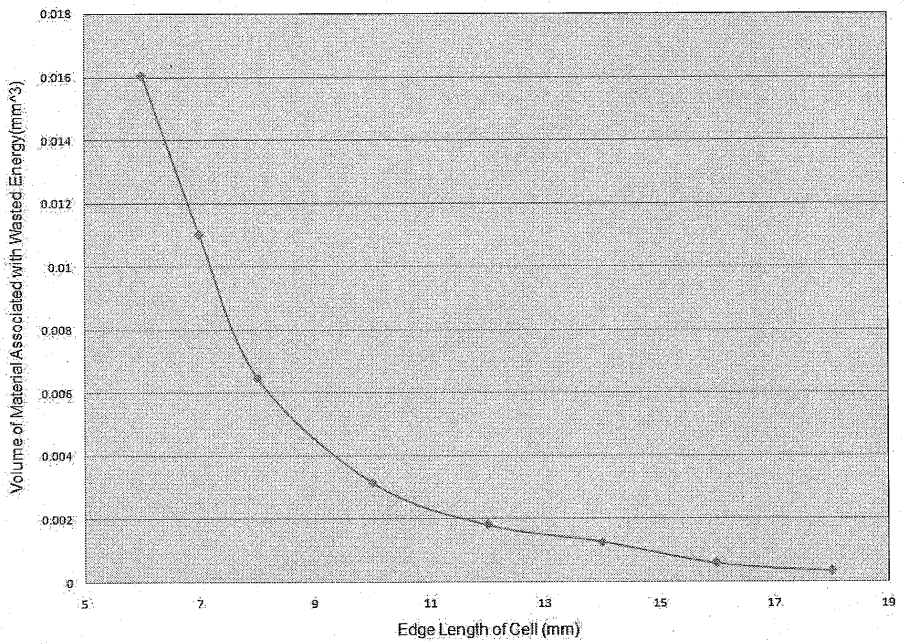



Figure 4.3: Effects of different surface cell sizes on the volume of material removed.

## 4.3 Basic Model Verification

This section describes two methods used to verify the computer model. To perform these verifications, the simulation was run with a surface that does not erode and thus stays flat.

### 4.3.1 Verification of Algorithm to Launch Particles in Weibull Distribution

As explained in Section 3.2, measurements [23] show that the probability of a particle impacting the flat surface at a radial distance between $r$ and $r + dr$ follows the probability distribution function given in Eq. 3.1. To prove that the simulation produces the same probability distribution, the radial locations for 1000000 particle impacts on the flat surface were computed. Then 100 equal radial intervals of size 0.06 $mm$ were determined. To obtain the probability of the simulation generating an impact within each interval, the number of radial

distances within the interval was divided by the total number of observations. These probabilities were compared to the ones calculated by Eq. 3.1 over each interval with $\beta = 15$ and $d = 20\ mm$. As can be seen in Figure 4.6, the comparison shows that the simulated results are highly matched with those obtained by the use of Eq. 3.1. It can thus be concluded that the present model is capable of properly launching particles according to realistic measured particle spatial distributions.

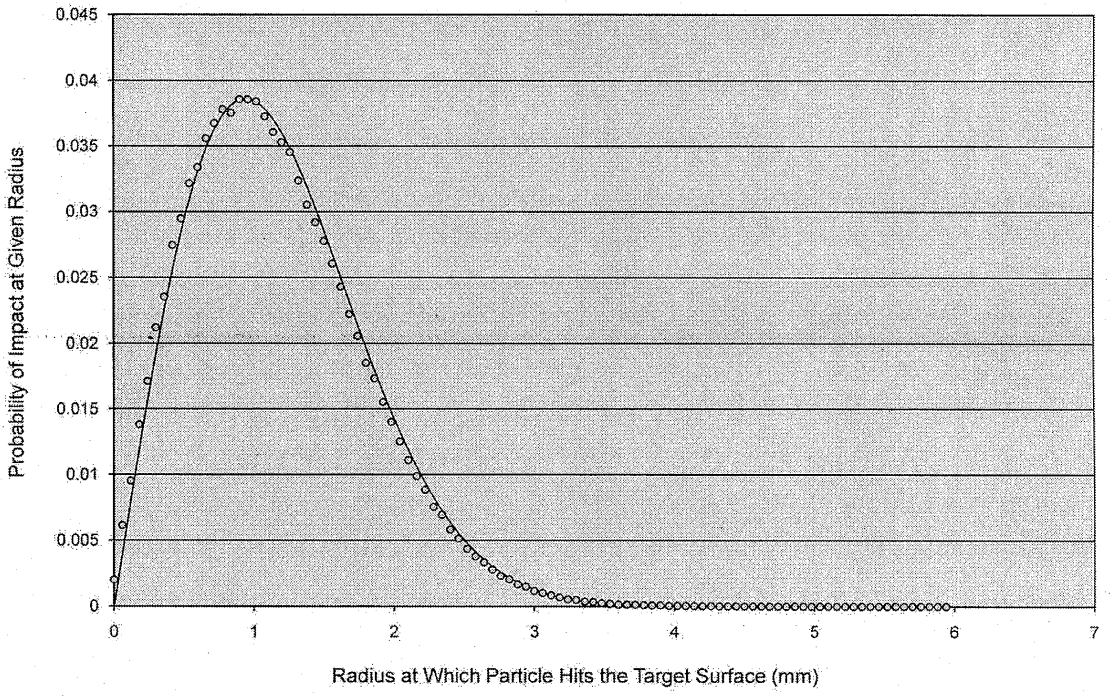



Figure 4.4: Verification of simulation launching algorithm for case of 0.76 $mm$ round nozzle having $\beta = 15$, with a stand-off-distance of 20 $mm$, and powder mass flow rate of 2.83 $g/min$. The solid line demonstrates theoretical results, and the circles demonstrate simulated results.

### 4.3.2 Comparison with Previous Computer Simulation for Non Eroding Flat Surface

Ciampini et al.'s model [29] implements a computer simulation for a non eroding surface which is capable of predicting interference-effects under a wide variety of input conditions. For different values of launch frequency, the percentages of particles arriving to the surface without experiencing any particle-particle collision were obtained for the present simulation, and compared against Ciampini et al.'s simulation. The simulations were run with different launch frequencies in the range of 500000 to 5000000 for a 0.1 second time period. The results were compared for two different nozzle stand-off-distances: 20 $mm$ and 10 $mm$. This comparison can be seen in Figures 4.5 and 4.6, which show good agreement between the two models. It can thus be concluded that the particle kinematics and collision detection algorithms implemented in the present simulation match those used by Ciampini et al.

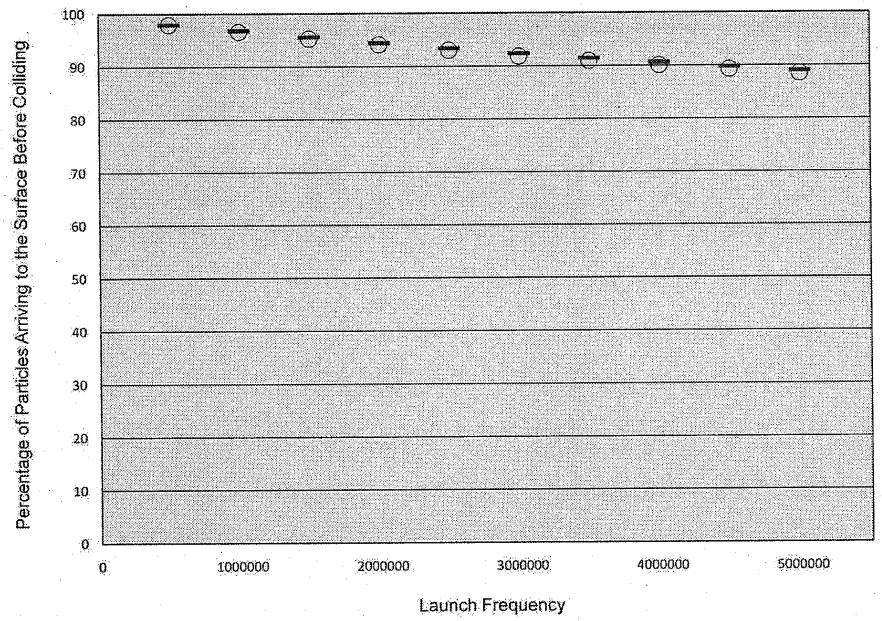



Figure 4.5: Comparison between the present model and Ciampini et al.'s model [29] using 25 $\mu m$ diameter aluminum oxide particles, and a point source round nozzle having $\beta = 15$ with a stand-off-distance of $d = 20$ $mm$. No surface erosion is included, so that the surface remains flat. Horizontal lines: results from the present model; circles: results from Ciampini et al.'s model

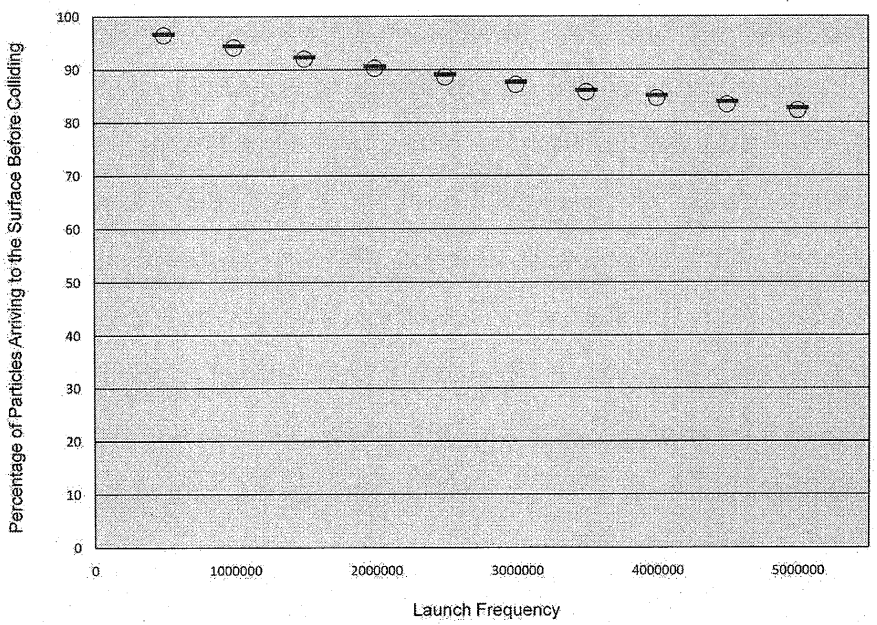



Figure 4.6: Comparison of present model with that of Ciampini et al. [29] using 25 $\mu m$ diameter aluminum oxide particles, and a point source round nozzle having $\beta = 15$ with a stand-off-distance of $d = 10$ $mm$. No surface erosion is included, so that the surface remains flat. Horizontal lines: results from the present model; circles: results from Ciampini et al.'s model

## 4.4 Experimental Verification

This section presents a comparison between simulated results and experimentally obtained results. To test the model for a brittle erosive system, a borosilicate glass target (5 $mm$ thick, Borofloat, Schott North America Inc., Elmsford, NY, USA) was used, and for a ductile erosive system, a polymethylmethacrylate (PMMA) target material was used. The experimental setup used to machine the holes and channels is exactly as that described in Ref. [23]. Holes were created by keeping the nozzle stationary, and straight channels were created by scanning the nozzle with a constant velocity for a different number of passes over the surface. In all experiments, surfaces were machined by blasting granular 25 $\mu m$ nominal diameter aluminum oxide. A constant blasting pressure of 200 $kPa$ was used. Experiments were performed using a round microabrasive nozzle of radius 0.38 $mm$. Channels and holes were cross-sectioned, and then were photographed and measured to obtain the cross sectional

profiles.

In all cases, the simulation was run under identical conditions to those used in the experiments. It should be noted that the experimental uncertainty for particle flow rates in a given experiment is estimated to be on the order of +/-5% [23]. For experiments on multiple days, this may be even higher, as much as 20% [30]. The launch frequencies used in simulation runs were adjusted within this range to improve the agreement between simulated and experimental results. However, in the case of the channel, since the nozzle is moving, more uncertainties can be expected, and in some cases, the mass flow rate is taken to be of slightly more deviation.

### 4.4.1 Glass Targets

It has been shown that the coefficient of restitution for inter-particle collisions, $e_{pp}$, has an extremely small effect on the predicted results (Section 4.5.3). Thus inter-particle collisions were assumed perfectly elastic and $e_{pp}$ was chosen as 1. Slikkerveer et al. [4] measured the coefficient of restitution between aluminum oxide particles and glass as a value between 0.2 and 0.5. For the borosilicate glass targets, a coefficient of restitution of 0.5 was chosen for particle-surface collisions. The friction coefficient, $f$, was taken as zero because surface friction has little effect on interference [22].

#### Low Flux

The results for the glass channel at low flux were compared with the measured data reported by Ghobeity et al. [23]. The nozzle was scanned with a constant velocity of 1 $mm/s$ for eight passes over the channel. The nozzle was placed at a 20 $mm$ distance from the glass plate with the centerline perpendicular to the surface. Particles were launched with the mass flow rate of 2.83 $g/min$. To test the model against the experiments in this case, the input parameters given in Table 4.2 were used. The constants $k$ and $D$, which are related to the erosive characteristics of the target material, were previously calculated from measurements of erosion rate using 25 micron aluminum oxide on Borofloat glass at the mass flow rate of 2.83 $g/min$ [23]. The model was run for 56 seconds to simulate eight passes of length

7 $mm$. This 7 $mm$ is approximately 1 $mm$ larger than the 'spot size' diameter at which 99.9999% of the particles hit the surface, so that the full jet of particles can assume to have passed a given cross section. The three-dimensional view of the predicted erosion profile after simulating eight passes can be seen in Figure 4.7. Figure 4.8 (taken from [23]) demonstrates the photograph of the cross-section of the actual surface after eight passes. Figure 4.9 shows the comparison of the measured and predicted cross-sections; a good agreement between the simulated and experimental results is demonstrated.

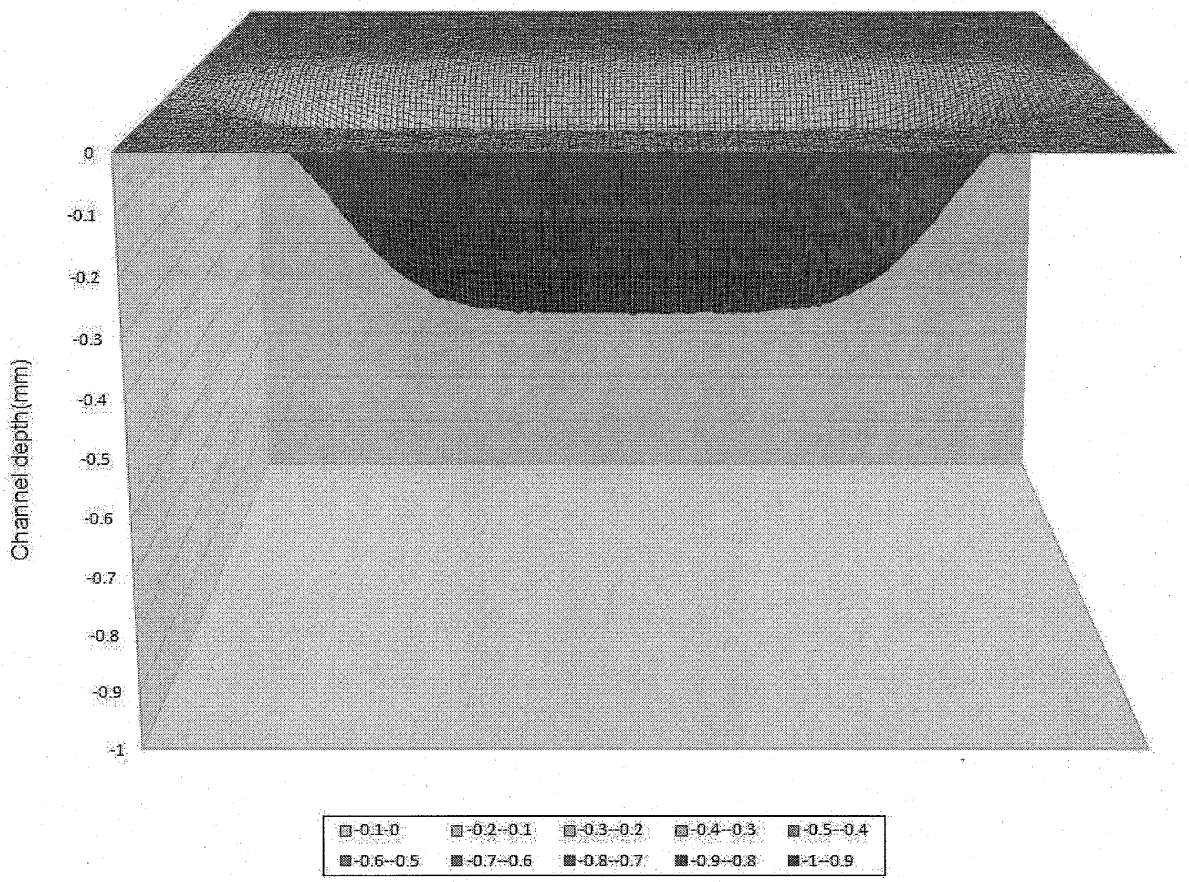



Figure 4.7: The predicted erosion profile of the borosilicate glass channel after eight passes on a surface of size $7.98 \times 11.9\ mm^2$ using the parameters given in Table 4.2.

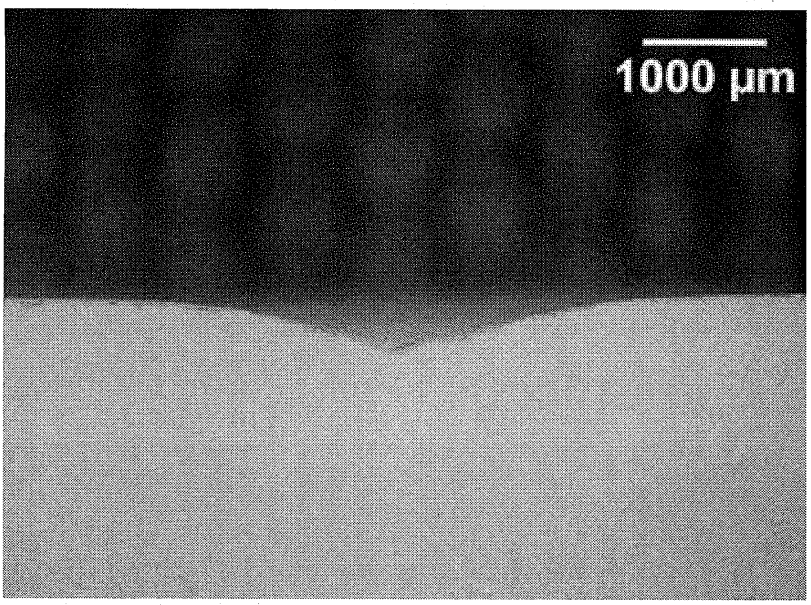



Figure 4.8: Cross-section of unmasked channel in borosilicate glass after eight passes of the nozzle [23].

| input | value | input | value |
|---|---|---|---|
| $v_{noz}$ | $1\ mm/s$ | $r_n$ | $0.38\ mm$ |
| $f_n$ | $1.23 \times 10^6$ | $\alpha$ | 0 |
| $d$ | $20\ mm$ | $\beta$ | 15 |
| $r_p$ | $25\ \mu m$ | $\rho_p$ | $4000\ kg/m^3$ |
| $V_{max}$ | $162\ m/s$ | $\rho_s$ | $2200\ kg/m^3$ |
| $K$ | 1.43 | $D$ | $6.3 \times 10^{-6}$ |
| $e_{pp}$ | 1 | $e_{ps}$ | 0.5 |
| $f$ | 0.0 | $L$ | $14\ \mu m$ |

Table 4.2: The inputs to the simulation for the case of glass channels at low flux.

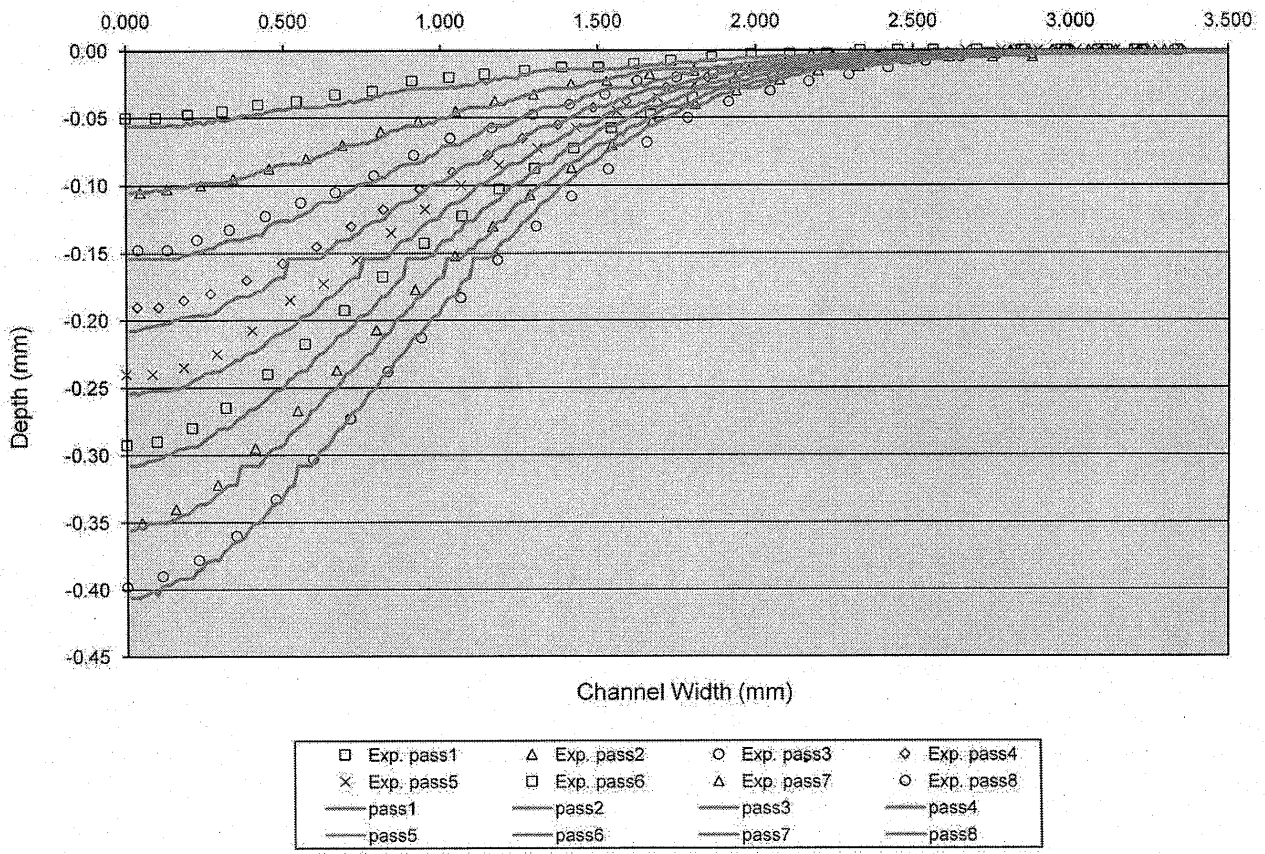



Figure 4.9: The comparison of predicted cross sections of borosilicate glass channels against the measured data, at low flux, using the input parameters from Table 4.2. Solid lines indicate predictions of the present model, and symbols represent experimental values.

For the case of glass holes machined at low flux, the predicted cross sections were compared with the experimental data reported by Ghobeity et al. [31]. The nozzle stand-off distance was set to 20 $mm$, and the nozzle centerline was perpendicular to the surface. 25 $\mu m$ aluminum oxide was launched at a mass flow rate of 2.83 $g/min$. The input parameters to the computer model used to simulate this case can be seen in Table 4.3. The three-dimensional view of the predicted erosion profile after 30 $s$ is demonstrated in Figure 4.10. Figure 4.11 compares the cross sections of the predicted and the measured erosion profiles of glass holes at low flux, and shows a reasonably good agreement. It should be noted that the

inconsistent material removal per unit time at the center of the hole, indicates significant experimental scatter. Such repeatability problems arise because it is difficult to actually measure the developing profile as a function of time. Therefore, the experimental profiles shown in Figure 4.8 are actually for 6 different holes, each sectioned separately to obtain the profile. Moreover, sectioning of a hole cross section relies on the ability of the experimenter to section perfectly in the center of the hole to obtain the deepest profile. This is often difficult to do. Thus, experimental scatter likely accounts for most of the discrepancy between model and experiment. Repeatability problems of this type are discussed in detail in Ref. [30].

| input | value | input | value |
|---|---|---|---|
| $v_{noz}$ | $0.0\ mm/s$ | $r_n$ | $0.38\ mm$ |
| $f_n$ | $1.4973 \times 10^6$ | $\alpha$ | 0 |
| $d$ | $20\ mm$ | $\beta$ | 15 |
| $r_p$ | $25\ \mu m$ | $\rho_p$ | $4000\ kg/m^3$ |
| $V_{max}$ | $162\ m/s$ | $\rho_s$ | $2200\ kg/m^3$ |
| $K$ | 1.43 | $D$ | $6.3 \times 10^{-6}$ |
| $e_{pp}$ | 1 | $e_{ps}$ | 0.5 |
| $f$ | 0.0 | $L$ | $14\ \mu m$ |

Table 4.3: The inputs to the simulation for the case of glass holes at low flux.

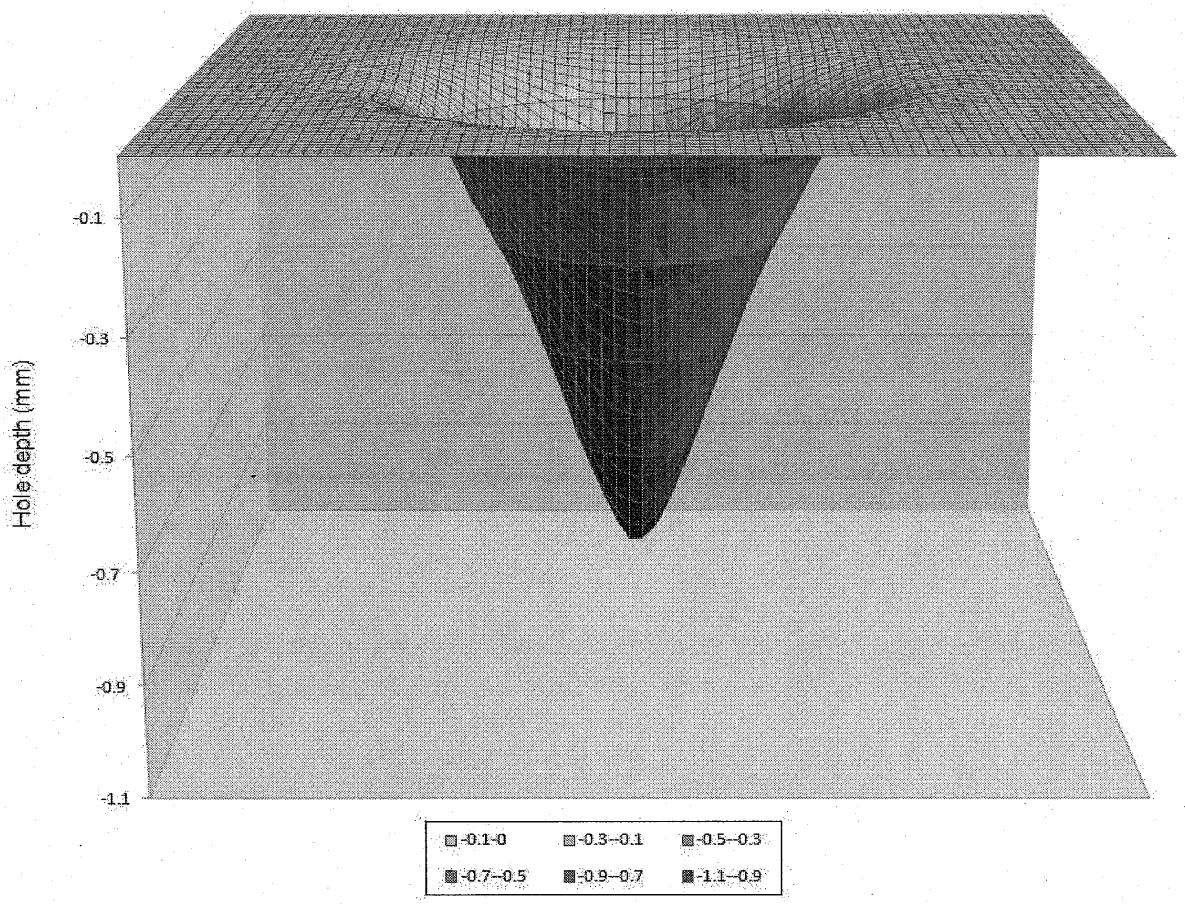



Figure 4.10: The predicted erosion profile of the borosilicate glass hole after 30 $s$ on a surface of size $7.952 \times 7.952\ mm^2$ using the parameters given in Table 4.3.

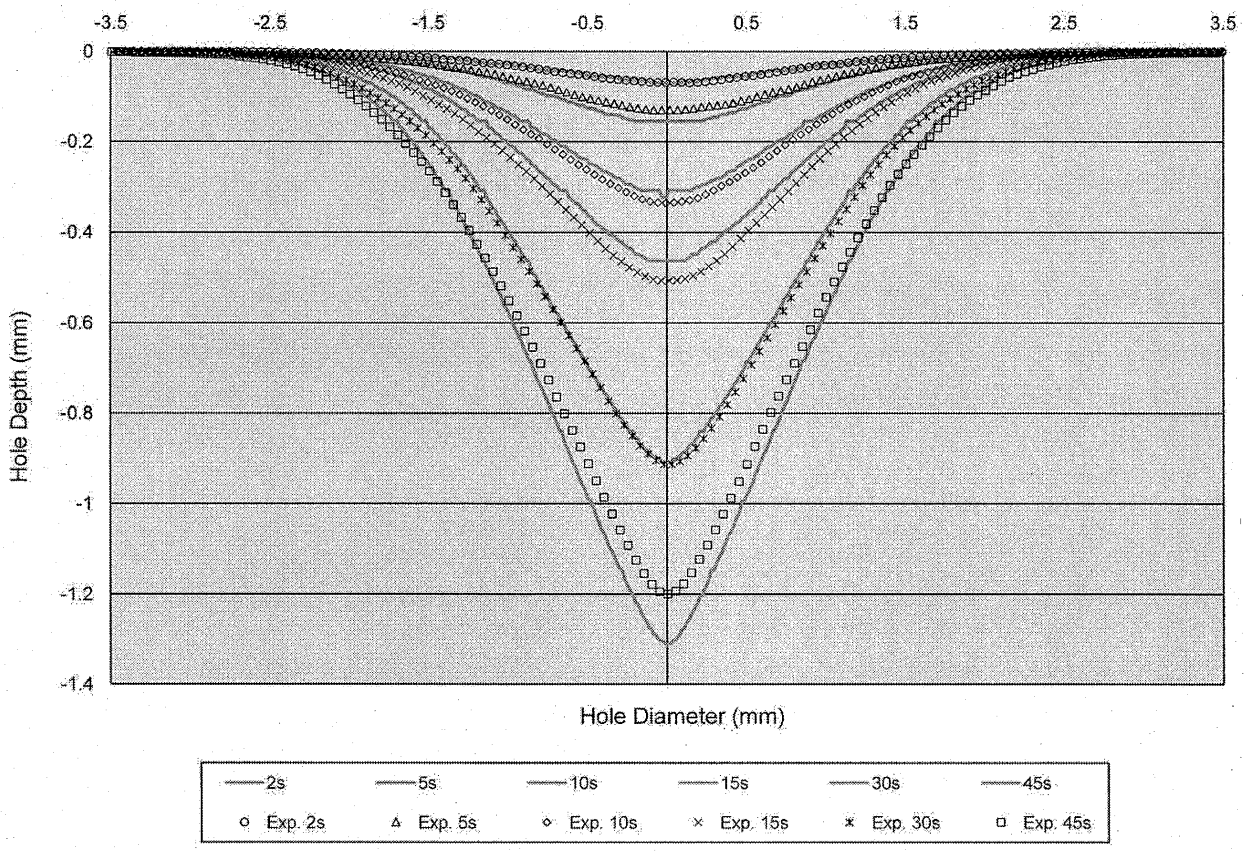



Figure 4.11: Comparison of predicted hole cross sections against the measured data for the borosilicate glass target at low flux using the inputs given in Table 4.3. Solid lines indicate predictions of the present model, and symbols represent experimental values.

Experiments using a nozzle held oblique to the surface such that the nozzle centerline was at an angle of $45^o$ to the surface, were also performed. The nozzle stand-off-distance was set to 20 $mm$ along the nozzle centerline resulting in a vertical nozzle exit plane to target surface distance of 14.14 $mm$. The abrasive was launched at a mass flow rate of 2.83 $g/min$. The simulation run was conducted using the input parameters given in Table 4.4. Figure 4.12 demonstrates the non axi-symmetric three-dimensional view of the predicted erosion profile after 25 $s$. Figure 4.13 shows the comparison between the predicted results and measured results; the simulation results are in very good agreement with experimental ones. When the nozzle is inclined, the abrasive jet is spread out over a larger surface area

than when it is held normal to the surface. As expected, the run conducted with a nozzle held oblique gives shallower holes which have a wider opening. This can be seen from Figure 4.14 which compares the cross-sections of glass holes created by keeping the nozzle oblique with the cross-sections created by a nozzle with the centerline perpendicular to the surface.

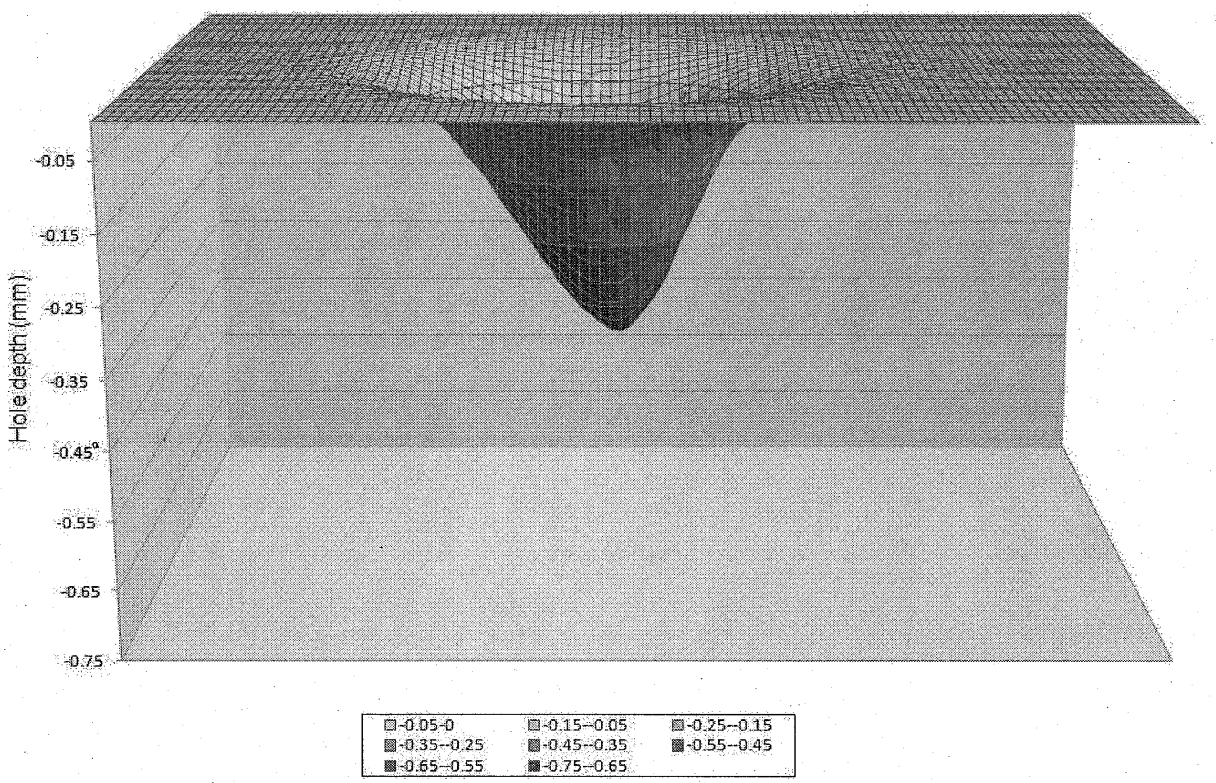



Figure 4.12: The predicted erosion profile of the borosilicate glass hole created by a nozzle held at 45 degrees to the surface, after 25 $s$, on a surface of size $7.504 \times 11.2\ mm^2$ using the parameters given in Table 4.4.

| input | value | input | value |
|---|---|---|---|
| $v_{noz}$ | $0.0\ mm/s$ | $r_n$ | $0.38\ mm$ |
| $f_n$ | $1.462 \times 10^6$ | $\alpha$ | $45^o$ |
| $d$ | $14.14\ mm$ | $\beta$ | 15 |
| $r_p$ | $25\ \mu m$ | $\rho_p$ | $4000\ kg/m^3$ |
| $V_{max}$ | $162\ m/s$ | $\rho_s$ | $2200\ kg/m^3$ |
| $K$ | 1.43 | $D$ | $6.3 \times 10^{-6}$ |
| $e_{pp}$ | 1 | $e_{ps}$ | 0.5 |
| $f$ | 0.0 | $L$ | $14\ \mu m$ |

Table 4.4: The inputs to the simulation for the case of glass holes at low flux using oblique stationary nozzle.

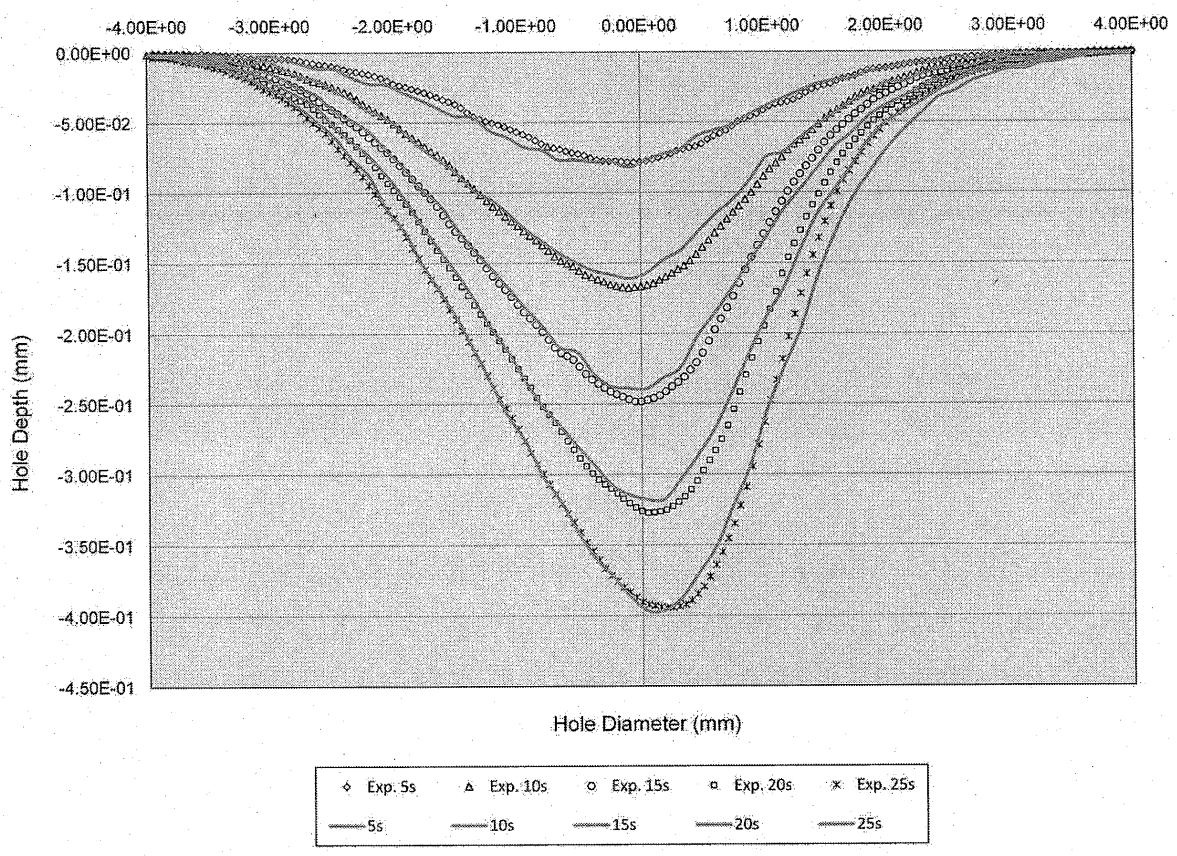



Figure 4.13: Comparison of predicted hole cross sections against the measured data on borosilicate glass at low flux. Nozzle held at 45 degrees to the surface, and the model inputs are from Table 4.4. Solid lines indicate predictions of the present model, and symbols represent experimental values.

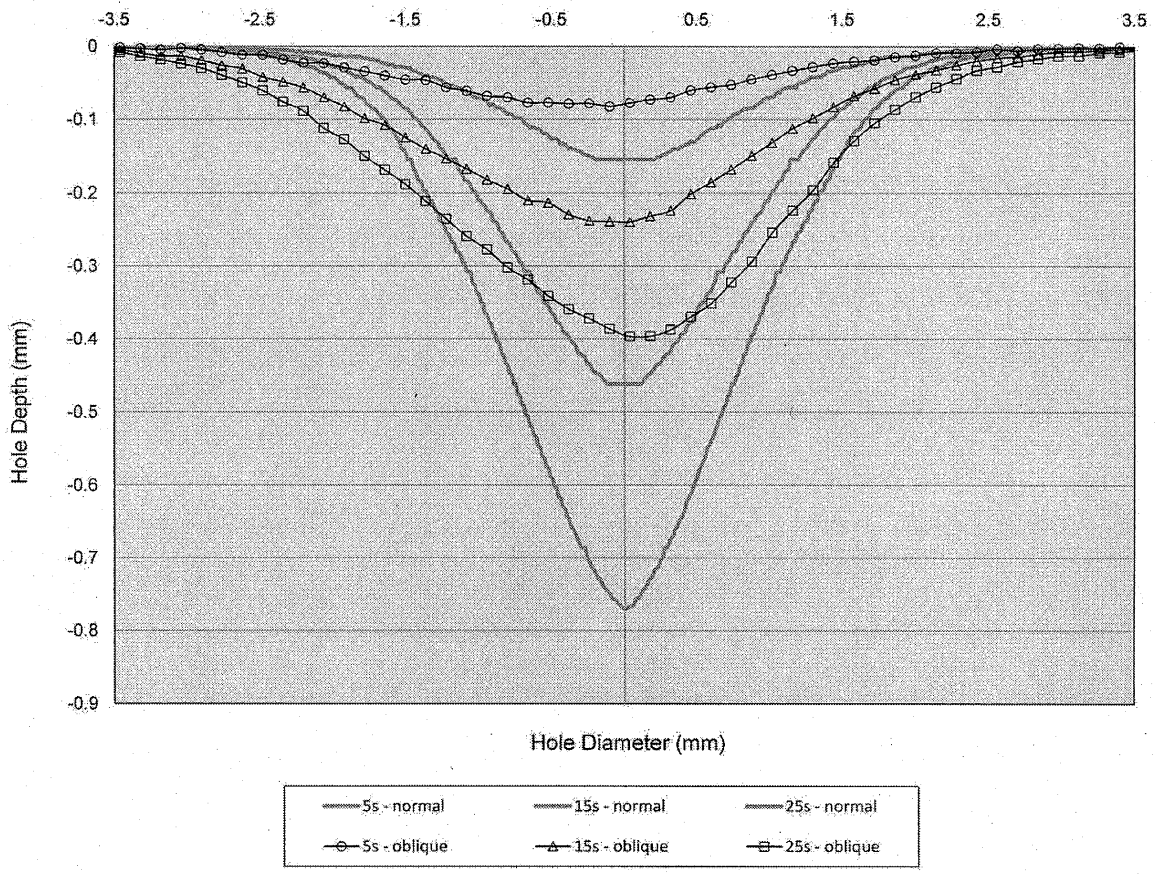



Figure 4.14: Comparison of predicted hole cross sections with the nozzle centerline normal to the surface and the ones using the nozzle held oblique with the inputs from Tables 4.3 and 4.4, respectively.

### Intermediate and High Flux

The high incident particle flux cases represent a very important test for the present simulation because it can account for inter-particle collisions, which are expected to be significant only at high flux. High fluxes are also expected to raise the mass loading in the jet, and thus lower the particle velocity [32]. Unfortunately, it was not possible to measure the particle velocity at such high fluxes using presently available equipment. Thus, the input parameter, $V_{max}$, the velocity on the nozzle centerline, was estimated by comparing the predicted cross-sections using different values of $V_{max}$.

To create glass holes at intermediate flux, the experimental mass flow rate was increased to 9.31 $g/min$. The nozzle was placed at 20 $mm$ stand-off distance with the centerline perpendicular to the surface. In this case, $V_{max}$ was taken as 140 $m/s$. Inputs to the model used to simulate this case are given in Table 4.5. Figure 4.16 demonstrates that the choice of the 140 $m/s$ max velocity gave results which were in good agreement with experimentally measured profiles. The model predicted that approximately 85% of the particles arrived to the surface without undergoing an inter-particle collision. As expected, this value is less than that seen for the low flux case (i.e 95%), indicating a moderate particle interference effect.

Compared to the input parameters of the glass hole at low flux given in Table 4.3, in this case, the $f_n$ increased by a factor of 3.169 and $V_{max}$ decreased by a factor of 1.157. Comparing the depth of the predicted profiles in Figure 4.16 with Figure 4.11 indicates that increase in depth of the profiles does not follow the same proportion (i.e. for example comparing the hole depth after 15 $s$, the depth increased by a factor of 2.33 rather than $\frac{3.169}{1.157}$ (= 2.738). This non proportionality is caused by the changes in interference effects due to the increase in launch frequency.

| input | value | input | value |
|---|---|---|---|
| $v_{noz}$ | 0.0 $mm/s$ | $r_n$ | 0.38 $mm$ |
| $f_n$ | $4.745 \times 10^6$ | $\alpha$ | 0 |
| $d$ | 20 $mm$ | $\beta$ | 15 |
| $r_p$ | 25 $\mu m$ | $\rho_p$ | 4000 $kg/m^3$ |
| $V_{max}$ | 140 $m/s$ | $\rho_s$ | 2200 $kg/m^3$ |
| $K$ | 1.43 | $D$ | $6.3 \times 10^{-6}$ |
| $e_{pp}$ | 1 | $e_{ps}$ | 0.5 |
| $f$ | 0.0 | $L$ | 14 $\mu m$ |

Table 4.5: The inputs to the simulation for the case of glass holes at intermediate flux.

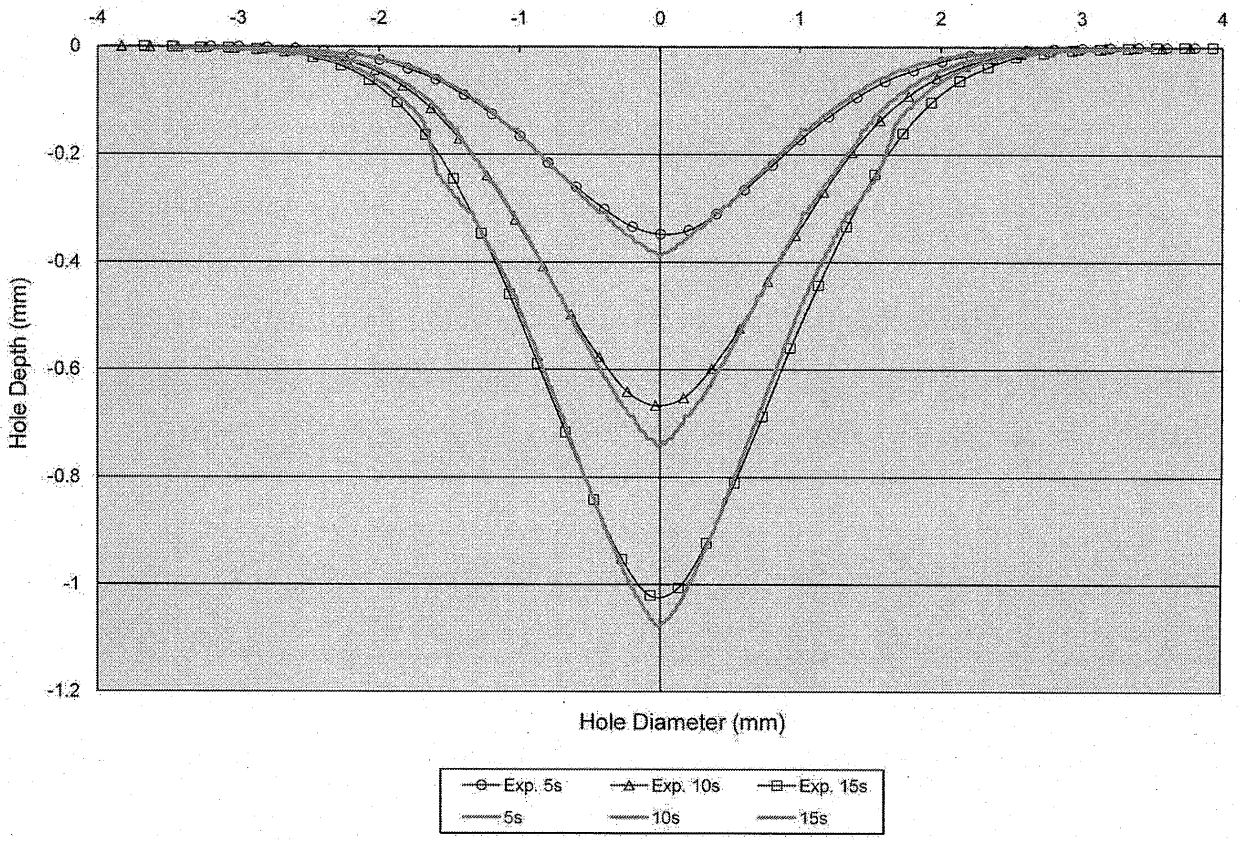



Figure 4.15: Comparison of predicted cross sections of holes against the measured data for a borosilicate glass target at intermediate flux using the inputs given in Table 4.5. Solid lines indicate predictions of the present model, and symbols represent experimental values.

To create glass holes at a very high flux, the mass flow rate was increased to 16.7 $g/min$. The nozzle was placed at 20 $mm$ stand-off distance with the centerline perpendicular to the surface. To model this case, $V_{max}$ was chosen as 120 $m/s$, a linear interpolation based on the velocities for the low (162 $m/s$, 2.8 $g/min$) and intermediate fluxes (140 $m/s$, 9.31 $g/min$). The other inputs were set to the values given in Table 4.6. The comparison between the cross-sections of the predicted and measured erosion profiles can be seen from Figure 4.16 which is satisfactory, but not as good as the results observed for the lower fluxes.

The model predicted that approximately 77% of the particles arrived at the surface without undergoing an inter-particle collision. As expected, this value is more than that

seen for the intermediate and low flux cases, indicating a significant interference effect.

It is likely that the discrepancy between experiment and simulation is due to the estimate of particle velocity used. To understand this further, the particle velocity needs to be measured properly and used as an input in the model.

Comparing the depth of the predicted profiles in Figure 4.16 with the ones in Figure 4.11 indicates that the depth of the profile in the high flux case did not change with the same proportion that $f_n$ increased and $V_{max}$ decreased. This implies that increases in the launch frequency produce different interference patterns. In the cases with higher launch frequency, a higher proportion of incoming particles are blocked by the rebounding particles.

| input | value | input | value |
|---|---|---|---|
| $v_{noz}$ | 0.0 $mm/s$ | $r_n$ | 0.38 $mm$ |
| $f_n$ | $8.414 \times 10^6$ | $\alpha$ | 0 |
| $d$ | 20 $mm$ | $\beta$ | 15 |
| $r_p$ | 25 $\mu m$ | $\rho_p$ | 4000 $kg/m^3$ |
| $V_{max}$ | 120 $m/s$ | $\rho_s$ | 2200 $kg/m^3$ |
| $K$ | 1.43 | $D$ | $6.3 \times 10^{-6}$ |
| $e_{pp}$ | 1 | $e_{ps}$ | 0.5 |
| $f$ | 0.0 | $L$ | 13 $\mu m$ |

Table 4.6: The inputs to the simulation for the case of glass holes at high flux.

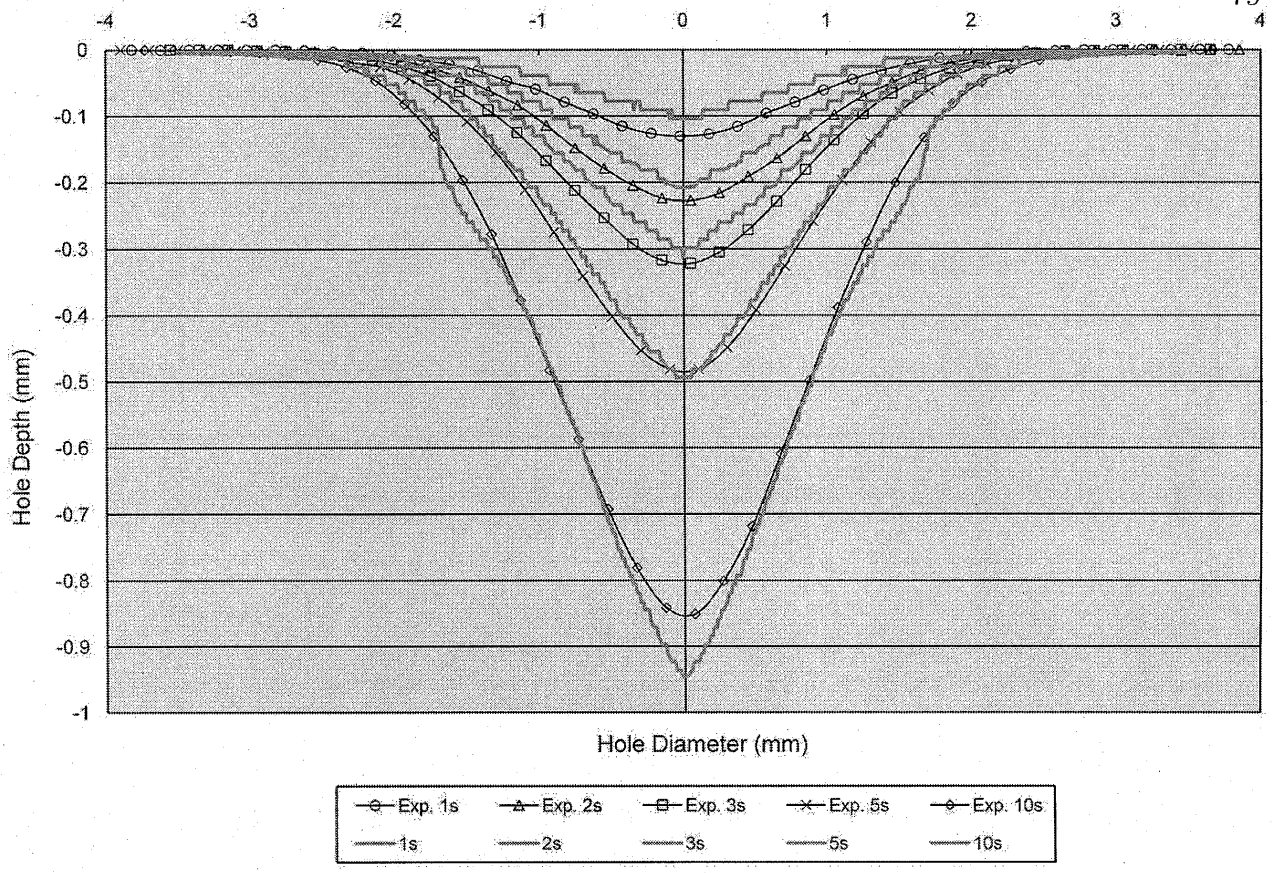



Figure 4.16: Comparison of predicted cross sections of holes against the measured data for the borosilicate glass target at high flux using the inputs given in Table 4.6. Solid lines indicate predictions of the present model, and symbols represent experimental values.

### 4.4.2 PMMA surfaces

For the PMMA targets, the coefficient of restitution for inter-particle collisions, $e_{pp}$, was also chosen as 1 since this has an extremely small effect on the predicted results (Section 4.5.3). The coefficient of restitution for particle-surface collisions, $e_{ps}$, was chosen as 0.5. Because surface friction has little effect on interference [22], it was taken as 0. To create channels in PMMA targets, the same experimental setup that was used with the glass channels (Section 4.3.1) was used, and the nozzle was scanned with the constant velocity of 0.25 $mm/s$ for seven passes over the surface. The nozzle was placed at 20 $mm$ distance to the PMMA plate with a centerline perpendicular to the target. Particles were launched with a mass flow rate

of 2.83 $g/min$. The inputs to the simulation for the case of the PMMA channels can be seen from Table 4.7. The simulation was run for 196 seconds for seven passes of length 7 $mm$. Figure 4.17 shows the comparison of the measured and predicted cross-sections which are in a good agreement.

Comparing to the glass channel at low flux, the erosion rate for the PMMA material is significantly slower. In the PMMA target, machining each pass took 28 $s$, while for glass channels at low flux , having a similar depth, each pass took 7 $s$. As expected, since a plastically deformed surface layer is formed later in ductile erosive systems, the material removal rate is slower than of the brittle erosive systems.

| input | value | input | value |
|---|---|---|---|
| $v_{noz}$ | 0.25 $mm/s$ | $r_n$ | 0.38 $mm$ |
| $f_n$ | $1.426 \times 10^6$ | $\alpha$ | 0 |
| $d$ | 20 $mm$ | $\beta$ | 15 |
| $r_p$ | 25 $\mu m$ | $\rho_p$ | 4000 $kg/m^3$ |
| $V_{max}$ | 162 $m/s$ | $\rho_s$ | 1190 $kg/m^3$ |
| $K$ | 2 | $D$ | $5.731 \times 10^{-8}$ |
| $e_{pp}$ | 1 | $e_{ps}$ | 0.5 |
| $f$ | 0.0 | $Hv$ | 0.25 $GPa$ |
| $n_1$ | 1.27 | $n_2$ | 15.5 |
| $L$ | 14 $\mu m$ | | |

Table 4.7: The inputs to the simulation for the case of PMMA channels at low flux.

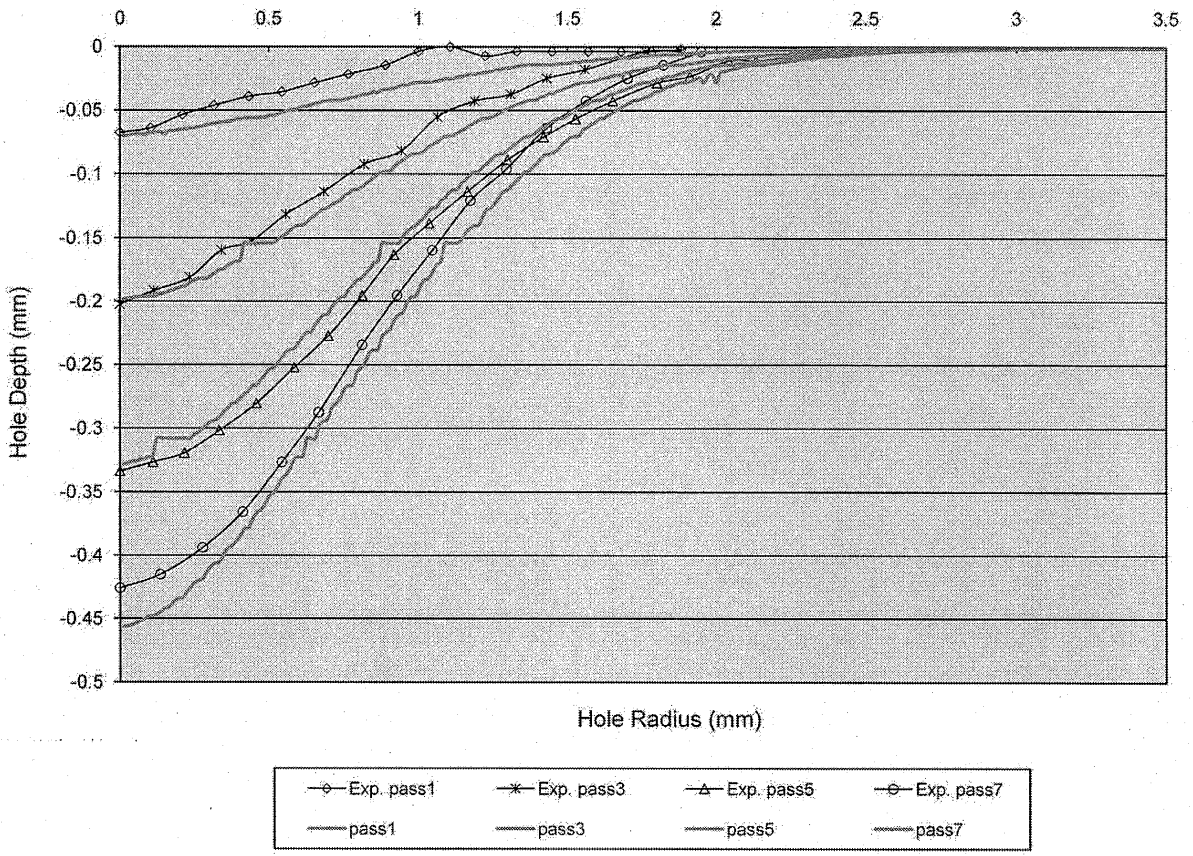



Figure 4.17: Comparison of predicted cross sections of PMMA channels against the measured data at low flux using the inputs given in Table 4.7. Solid lines indicate predictions of the present model, and symbols represent experimental values.

The Table 4.8 shows the percentages of errors associated with each case. The data for this table were measured for the deepest points in channels and holes. Usually the largest errors between predicted cross-sections and experimentally measured data occur at these points.

| target | $v_{noz}$ | flux | $\alpha$ | error |
|---|---|---|---|---|
| glass | 1 $mm/s$ | low | $90^o$ | 5.06% |
| glass | 0 | low | $90^o$ | 7.33% |
| glass | 0 | low | $45^o$ | 3.2% |
| glass | 0 | intermediate | $90^o$ | 9.41% |
| glass | 0 | high | $90^o$ | 9.14% |
| PMMA | 0.25 $mm/s$ | low | $90^o$ | 3.65% |

Table 4.8: The errors associated with each case.

## 4.5 Parametric Study

This section presents the effects of different parameters on the depth and shape of predicted erosion profiles, using the present model for glass holes at an intermediate flux. Simulation runs were conducted by varying the value of one parameter and holding all the others parameters at the values from Table 4.5.

### 4.5.1 Friction Coefficient

Figure 4.18 compares the predicted cross-sections for different values of friction coefficient, $f = 0$ and $f = 0.5$. There is little difference in the volume of removed material as $f$ is varied; slightly more cells are removed for the frictionless case. In a real application, an increase in the friction results in more energy being wasted transforming the incident linear velocity into a rotational one (for cases where the particles stick), rather than damaging. In the present case, however, the model only considers the energy loss for particle rebound kinematics purposes. When a particle impacts a surface with friction, it loses some of its energy and the tangential velocity components reduce. This slow moving rebounding particle (i.e. slower than if there was no friction) has a greater chance of being hit by an incoming particle than if there were no friction. Moreover, if it is hit by an incoming particle and does strike the surface again, it will be at a lower velocity than if there were no friction. Since particles that arrive at the surface with a particular initial velocity do the same amount of

damage regardless of friction, the difference between the results is likely mostly caused by the loss in energy due to incident particles hitting the slow moving rebounding particles.

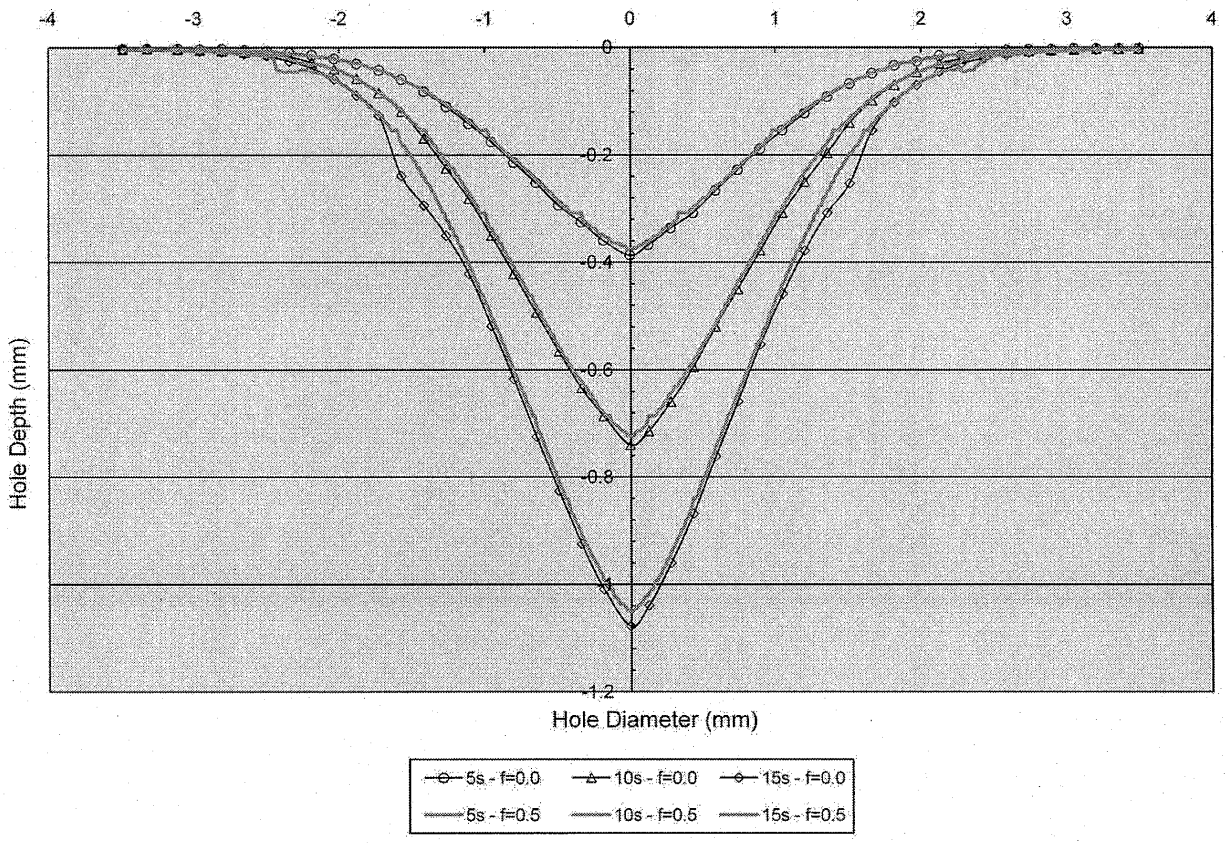



Figure 4.18: The comparison of predicted cross-sections using $f = 0$ and $f = 0.5$, with all other parameters at the values given in Table 4.5.

### 4.5.2 Coefficient of Restitution for Particle-surface Collisions

The kinetic energy lost by a particle impacting the surface can be expressed in terms of $e_{ps}$. The smaller the value of $e_{ps}$, the lower will be the rebounding velocities of the particles. Similar to the case of an increase in friction, a decrease in $e_{ps}$ will cause a lower incident energy for particles that hit the surface more than once. The slower moving particles rebounding from the surface also stay for a longer time in the path of the incoming particles, thus

increasing the frequency of inter-particle collisions, and increasing the probability that the rebounding particles hit the surface multiple times. Figures 4.19 and 4.20 compare the total number of inter-particle collisions and particle-surface collisions for simulation runs with $e_{ps} = 0.2$ and $e_{ps} = 1$, respectively; the system with lower $e_{ps}$ experienced more inter-particle and particle-surface collisions, as expected. The corresponding predicted cross-sections are compared in Figure 4.21. A decrease in $e_{ps}$, resulted in more cells being removed from the target substrate. This is due to the increase in the number of particles striking the surface multiple times.

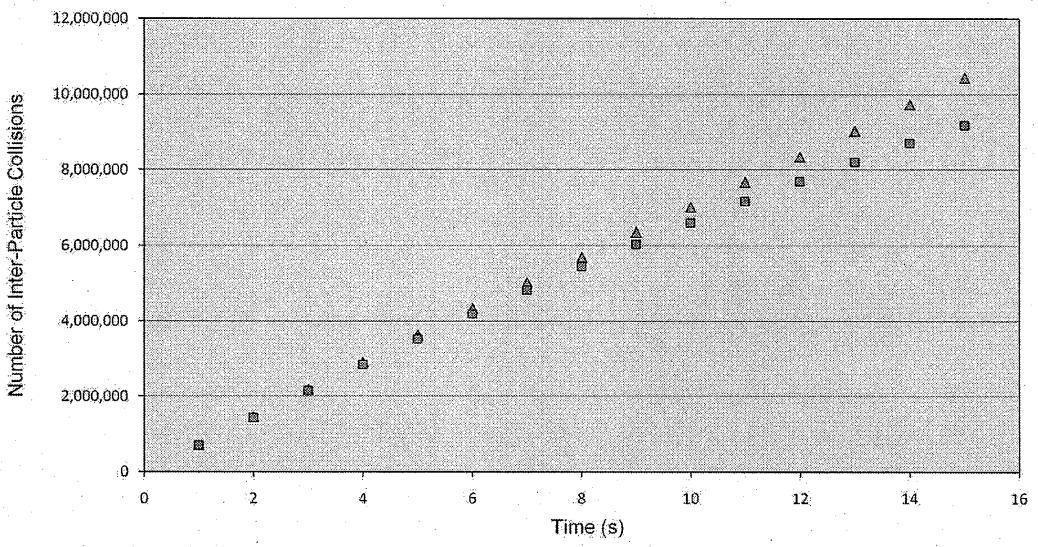



Figure 4.19: Comparison of the number of inter-particle collisions for $e_{ps} = 0.2$ and $e_{ps} = 1$, with all other parameters at the values given in Table 4.5. Triangles: results for a run conducted with $e_{ps} = 0.2$; squares: results for a run conducted with $e_{ps} = 1$

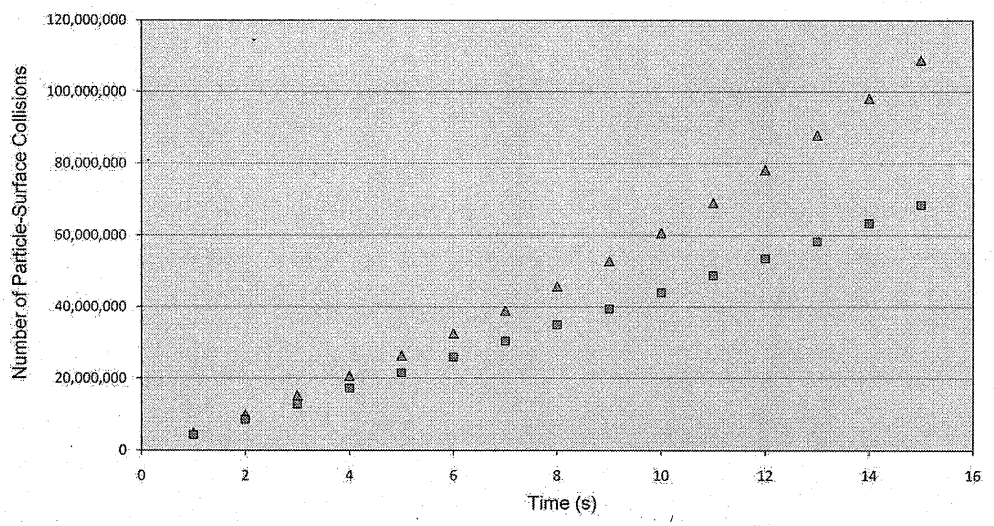



Figure 4.20: Comparison of the number of particle-surface collisions for $e_{ps} = 0.2$ and $e_{ps} = 1$, with all other parameters at the values given in Table 4.5. Triangles: results for a run conducted with $e_{ps} = 0.2$; squares: results for a run conducted with $e_{ps} = 1$

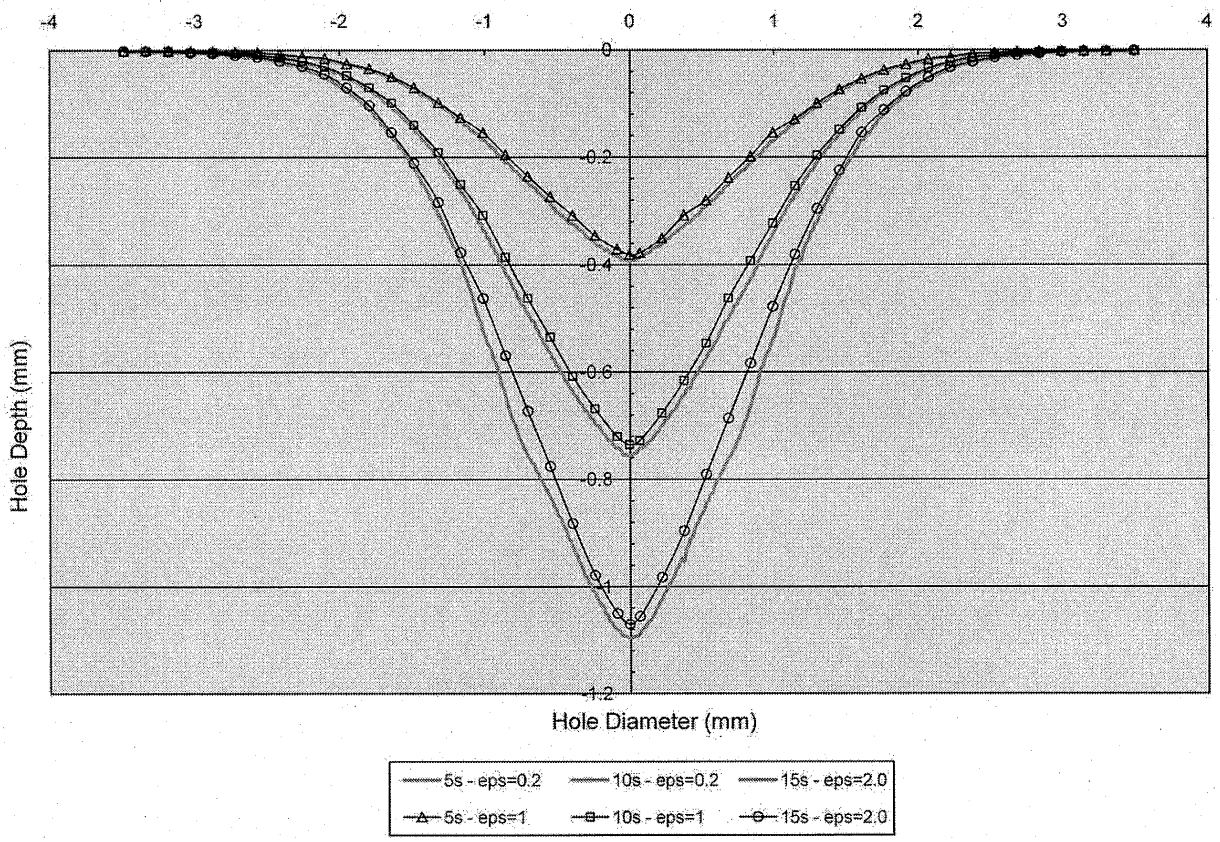



Figure 4.21: Comparison of predicted cross-sections using $e_{ps} = 0.5$ and $e_{ps} = 0$, with all other parameters at the values given in Table 4.5.

### 4.5.3 Coefficient of Restitution for Inter-particle Collisions

Figures 4.22 and 4.23 compare the total number of inter-particle collisions and particle-surface collisions for simulation runs with $e_{pp} = 0.2$ and $e_{pp} = 1$, respectively. In the run conducted with $e_{pp} = 0.2$, particles rebounding from inter-particle collisions lose some of their kinetic energy. This results in a decrease in their velocity, and it thus takes more time for them to leave the system. Similar to what was found previously for varying $f$ and $e_{ps}$, these slower particles tend to undergo more inter-particle collisions, which can result in more particle-surface collisions. However, the differences are extremely small, and this is reflected in Figure 4.24, which compares the predicted cross-sections for the two values of

$e_{pp}$. As expected, the depths are essentially independent of $e_{pp}$, since the number of surface collisions is not greatly changed (Figure 4.23).

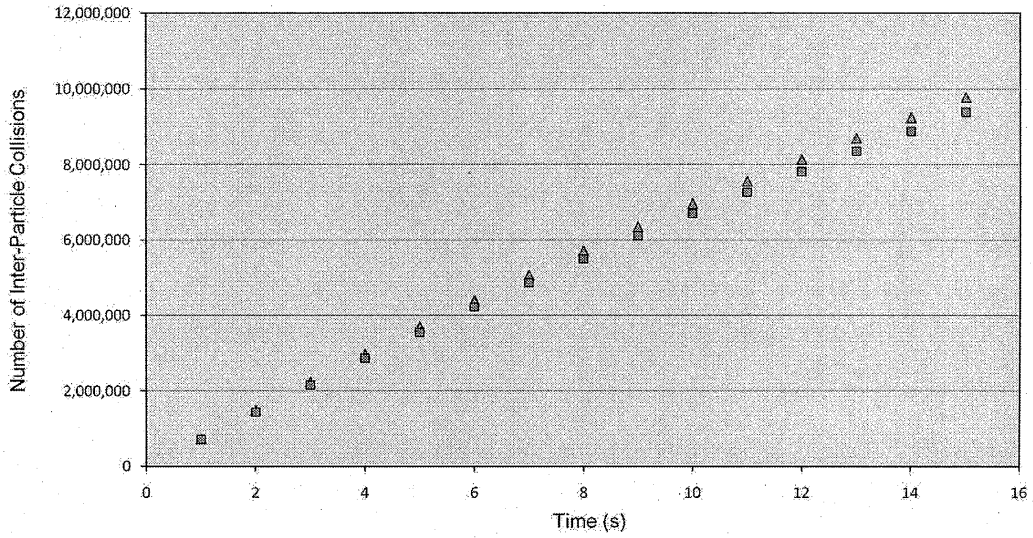



Figure 4.22: Comparison of the number of inter-particle collisions for $e_{pp} = 0.2$ and $e_{pp} = 1$, with all other parameters at the values given in Table 4.5. Triangles: results for a run conducted with $e_{pp} = 0.2$; squares: results for a run conducted with $e_{pp} = 1$

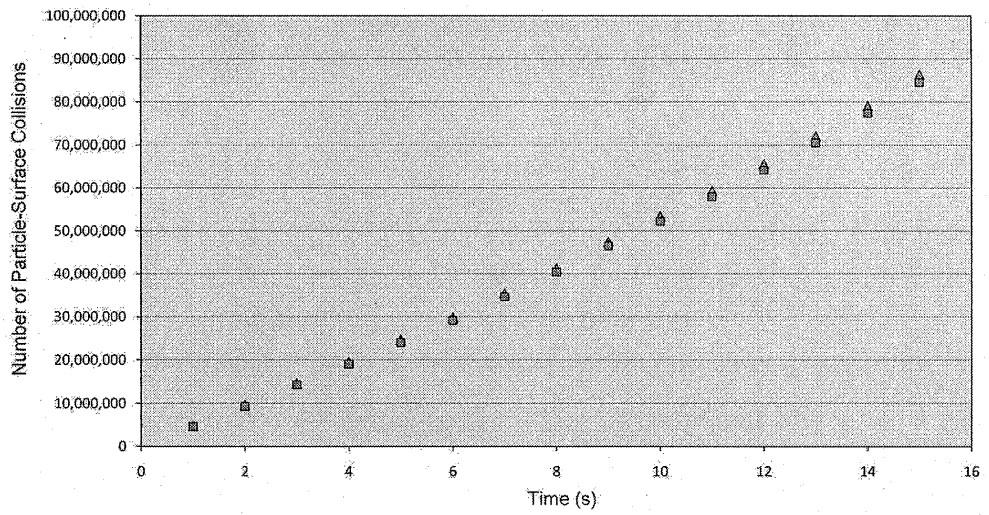



Figure 4.23: Comparison of the number of particle-surface collisions for $e_{pp} = 0.2$ and $e_{pp} = 1$, with all other parameters at the values given in Table 4.5. Triangles: results for a run conducted with $e_{pp} = 0.2$; squares: results for a run conducted with $e_{pp} = 1$

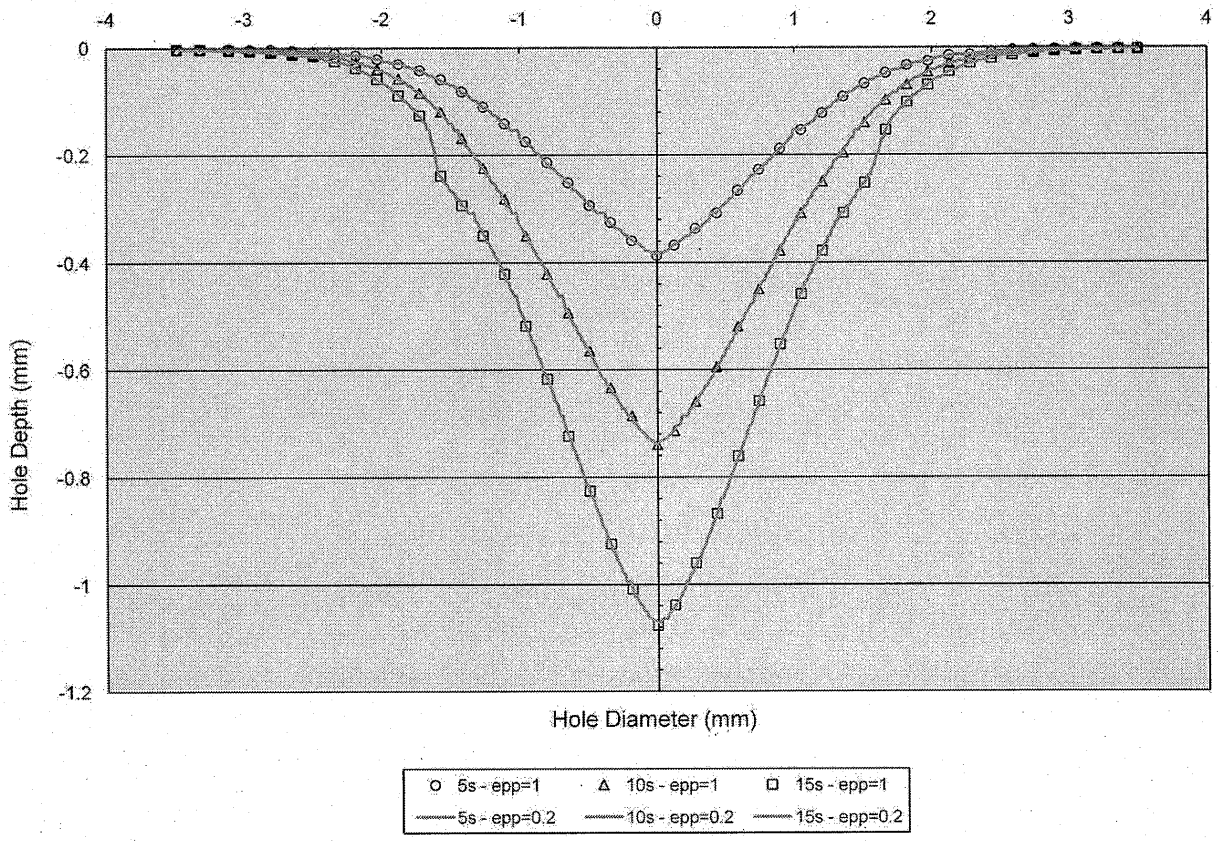



Figure 4.24: Comparison of predicted cross-sections using $e_{pp} = 1$ and $e_{pp} = 0.2$, with all other parameters at the values given in Table 4.5.

## 4.6 Limitations

A number of factors may affect the reliability of the simulated results. These are briefly outlined as follows:

1. The model ignores fluid effects. Ignoring the fluid effects implies that the particles are moving on straight paths. However, in reality, the gas flow can affect the particle rebound velocity [4]. To avoid such effects, the stand-off-distance should be large enough to have a negligible effect of gas on rebound velocities. Appropriate guidelines

can be found in Ref. [4].

2. The model ignores gravitational effects. Gravity causes the slow moving particles to deviate from the linear trajectories more than the faster moving particles. The error associated with this assumption increases for the cases that incident stream velocity is low or $e_{ps}$ and $e_{pp}$ are close to zero and a large amount of energy loss occurs. However, since the particle velocities in realistic abrasive jets are so high, and the standoff distances so small, it is unlikely that gravity has a significant effect on the results.

3. The model assumes that particles are spherical. Since real powders are more angular, the assumption of spherical particles affects the particle collision kinematics. Two typical behaviors were observed experimentally by Hutchings [33] during the impact of square particles. The same behaviors were observed during the impact of diamond-shape particles in the experiments performed by Dhar et al. [34]. Particles either rotate forward or backward, and in most cases multiple impacts were observed. Dhar et al. [34] have reported that forward rotating particles rebounds with a very high rotational velocity and significant amount of the linear kinetic energy of these particles is transferred to the rotational energy. They also reported that in the case of backward rotating particles, increase in rebound rotational energy is significantly lower. This assumption significantly simplified the prediction of future behavior of particles. However, it resulted in overestimating the linear kinetic energy of particles due to collisions affects the rebounding characteristic and thus the interference patterns. The error associated with this assumption is reduced for the case of more rounded powders.

4. The model assumes no friction for inter-particle collisions. Including friction in inter-particle collisions is at the expense of solving fifteen equations for each collision [21], an unacceptable computational penalty. Certainly, the fundamental assumption that the angular particles behave as spheres most likely introduces more error than the omission of friction in inter-particle collisions.

5. In the high flux case, launching a few particles simultaneously shows the similar in-

terference effects with launching one particle at a time at the same overall flux. But if many particles are launched simultaneously, they travel in bursts and have different associated interference patterns. [22]

6. The model neglects contact mechanics. It is assumed that particles are not deformed during collisions and time duration of impacts equals zero. This assumption reduces the probability of simultaneous collisions of more than two particles, which is an extremely complicated problem.

# Chapter 5

# Conclusions and Recommendations

## 5.1 Summary

In the present study, a computer simulation was developed having a capability of predicting the shape and size of erosion profiles in the abrasive jet machining process. The model allows for the variation of a wide range of material and process parameters and, for the first time, allows particle to particle collisions to occur. To the author's knowledge, no such model exists in the literature.

The implementation of the computer model is based on the event-driven approach in which the system advances from event to event, and after handling each event, the next upcoming event is predicted. The information of the predicted events are saved in an event queue which is ordered with respect to the event occurrence times. To speed up the particle-particle collision detection, the space holding the particles was divided into small parts, and inter-particle collisions were detected between particles belonging to the same area. To model the surface advancement, the cellular algorithm was used, which is based on dividing the target surface into very small cells. The cellular octree data structure which was defined based on the octree data structure was used to model the target surface. To predict the next particle-surface collision, particles were tested against the nodes of the cellular octree, which were cubes presenting the volume of the substrate, in a top-down manner. To handle inter-particle collisions and particle-surface collisions, the model uses a simplified coefficient of restitution model.

Since the model is able to track individual particles and their collisions until they leave the system, interference effects are included in the model. The model is also capable of examining the effects of material and process parameters on the shape and size of the erosion profile, and interference patterns.

The predictions of the present model were in quite good agreement with a previously developed computer simulation for non eroding surfaces, which is capable of predicting interference-effects. The predicted eroded profiles as a function of time, for different levels of flux and for two different erosive systems were compared with experimentally measured results. The good agreement with the measured data demonstrates the promise of the model in predicting the developing erosion profile for abrasive jet micromachining operations.

## 5.2 Future Work

The present study only considered the AJM process on unmasked targets. In many applications, masked surfaces are exposed to the stream of particles. The present model could be modified in the future so that the shape and depth of the erosion profile for targets covered locally by an erosion resistant mask, can be predicted. This would allow, for the first time, a prediction of abrasive jet micromachined features that includes particle to particle and particle to mask interactions.

It should be noted that in most of the cases masks cannot provide prefect resistance against the erosion , and will themselves erode. It would thus be desirable to use the presently developed algorithms to also study an eroding mask. This is particularly interesting, since the edges of the mask become rounded as they erode, and this will likely change the rebounding characteristics of particles impacting the edges. This interaction between mask edge erosion and resulting eroded target material has never before been studied, and the present model provides the basic algorithms needed to construct such a complete model.

# Appendix A

# Computer Simulation Source Code

This appendix presents the implementations of the simulation classes in the Java programming language.

**Plane.java:** The Plane class provides functionality to deal with planar surfaces.

```
package Objects3D;

public class Plane {
    /**
     * The array used to store the coefficients of the plane equation.
     */
    private double[] equation;

    /**
     * A point on the plane.
     */
    private Vector3D origin;

    /**
     * Normal to the plane.
     */
    private Vector3D normal;

    /**
     * Creates a new instance of Plane
     *
     * @param origin a point on the plane.
     * @param normal a normal to the plane.
     */
    public Plane(Vector3D origin, Vector3D normal)
    {
        this.normal = new Vector3D(normal);
        this.origin = origin;
        equation = new double[4];
        getEquation()[0]=normal.getX();
```

```
        getEquation()[1]=normal.getY();
        getEquation()[2]=normal.getZ();
        getEquation()[3]=-(VectorOperation.dotProduct(normal,origin));
    }

    /**
     * Checks if the plane is front facing to the given vector.
     *
     * @param v a vector.
     *
     * @return true if the plane is front facing to the given vector; false otherwise.
     */
    public boolean isFrontFacingTo(Vector3D v)
    {
        double dot = VectorOperation.dotProduct(getNormal(), v);
        return(dot<=0);
    }

    /**
     * Calculates the signed distance of the given point from this plane.
     */
    public double signedDistanceTo(Vector3D point)
    {
        return(VectorOperation.dotProduct(point, getNormal())+getEquation()[3]);
    }

    public double[] getEquation() {
        return equation;
    }

    public Vector3D getOrigin() {
        return origin;
    }

    public Vector3D getNormal() {
        return normal;
    }
}
```

**Vector3D.java:** The Vector3D class provides functionality for three-dimentional vector quantities.

```
package Objects3D;

public class Vector3D {
    /**
     * The first coordinate of the vector.
     */
    public double X;

    /**
     * The second coordinate of the vector.
     */
    public double Y;

    /**
     * The third coordinate of the vector.
     */
    public double Z;

    /**
     * Creates a new instance of Point
     */
    public Vector3D() {
    }

    /**
     * Creates a new instance of Point and sets its coordinate to the given values.
     *
     * @param x the value of the first coordinate.
     * @param y the value of the second coordinate.
     * @param z the value of the third coordinate.
     */
    public Vector3D(double x, double y, double z) {
        this.setX(x);
        this.setY(y);
        this.setZ(z);
        return;
    }

    /**
     * Creates a new instance of Point and sets its coordinate to the coordinate values of
       the given vector.
     *
     * @param C a vector.
     */
    public Vector3D(Vector3D C) {
        this.setX(C.X);
        this.setY(C.Y);
        this.setZ(C.Z);
        return;
    }

    /**
     * Sets the coordinates of this vector to the given values.
     *
     * @param x the value of the x-coordinate.
     * @param y the value of the y-coordinate.
     * @param z the value of the z-coordinate.
```

```
     */
    public void setPoint(double x, double y, double z) {
        this.setX(x);
        this.setY(y);
        this.setZ(z);
        return;
    }

    /**
     * Sets the coordinates of this vector to the coordinates values of the given vector.
     *
     * @param v a vector
     */
    public void setPoint(Vector3D v) {
        this.setX(v.getX());
        this.setY(v.getY());
        this.setZ(v.getZ());
        return;
    }

    /**
     * Adds the coordinates of this vector with the coordinates of the given vector.
     *
     * @param v a vector
     */
    public void addWith(Vector3D v) {
       this.X += v.getX();
       this.Y += v.getY();
       this.Z += v.getZ();
   }

    /**
     * Adds the coordinates of this vector with the given value.
     */
    public void addWith(double c) {
        this.X += c;
        this.Y += c;
        this.Z += c;
    }

    /**
      * Multiplies the coordinates of this vector by the given value.
      */
    public void multiplyBy(double c) {
        this.X *= c;
        this.Y *= c;
        this.Z *= c;
    }

    /**
      * Divides the coordinates of this vector by the given value.
      */
    public void divideBy(double c) {
        this.X /= c;
        this.Y /= c;
        this.Z /= c;
    }

    /**
     * Calculates the norm (length) of this vector.
     *
     * @return the norm of this vector.
     */
    public double getNorm() {
```

```
    return (Math.sqrt(X*X+Y*Y+Z*Z));
}

/**
 * Calculates the square norm of this vector.
 *
 * @return the square norm of this vector.
 */
public double getSquareNorm() {
    return (X*X+Y*Y+Z*Z);
}

/**
 * Normalizes this vector to obtain a unit vector.
 *
 * @return this vector which has been normilized; a vector in the same direction but
     with length 1.
 */
public Vector3D normalize() {
    double norm = this.getNorm();
    this.setPoint(this.X/norm, this.Y/norm, this.Z/norm);
    return this;
}

/**
 * Moves this vector for the given time interval with the given velocity.
 *
 * @param V the velocity vector.
 * @param time the time interval.
 */
public void movePoint(Vector3D V, double time){
    this.setPoint(V.X*time+this.X, V.Y*time+this.Y, V.Z*time+this.Z);
}

/**
 * Checks if the content of this vector is equal to the content of the given vector.
 *
 * @param v a vector
 *
 * @return true if the content of this vector is equal to the content of the given
     vector; false otherwise.
 */
public boolean isEqualTo(Vector3D v){
   return( (this.X==v.getX()) && (this.Y==v.getY()) && (this.Z==v.getZ()) );
}

/**
* Returns the x-coordinate of this vector.
*/
public double getX() {
    return X;
}

/**
 * Sets the x-coordinate of this vector to the given value.
 */
public void setX(double X) {
    this.X = X;
}

/**
* Returns the y-coordinate of this vector.
*/
public double getY() {
```

```
        return Y;
    }

    /**
     * Sets the y-coordinate of this vector to the given value.
     */
    public void setY(double Y) {
        this.Y = Y;
    }

    /**
    * Returns the z-coordinate of this vector.
    */
    public double getZ() {
        return Z;
    }

    /**
     * Sets the z-coordinate of this vector to the given value.
     */
    public void setZ(double Z) {
        this.Z = Z;
    }

     /**
     * Returns a string representation of this vector.
     *
     * @return a string representation of this vector.
     */
     public String toString() {
          return("["+X+", "+Y+", "+Z+"]");
     }
}
```

**VectorOperation.java:** The VectorOperation class provides Operations on vector quantities.

```
package Objects3D;

import OcTreeADT.CellularOcTree;

import java.io.FileOutputStream;

import java.io.IOException;

import mySimulation.SimulationManager;

public class VectorOperation {
    /**
     * Creates a vector which is equal to the sum of the two given vectors.
     *
     * @param v1 a vector.
     * @param v2 a vector.
     *
     * @return a vector which is equal to the sum of the two given vectors.
     */
    public static Vector3D add(Vector3D v1, Vector3D v2){
        return(new Vector3D(v1.getX()+v2.getX(), v1.getY()+v2.getY(), v1.getZ()+v2.getZ()))
            ;
    }

    /**
     * Creates a vector which is equal to the sum of the three given vectors.
     *
     * @param v1 a vector.
     * @param v2 a vector.
     * @param v3 a vector.
     *
     * @return a vector which is equal to the sum of the three given vectors.
     */
    public static Vector3D add(Vector3D v1, Vector3D v2, Vector3D v3){
        double x = v1.getX() + v2.getX() + v3.getX();
        double y = v1.getY() + v2.getY() + v3.getY();
        double z = v1.getZ() + v2.getZ() + v3.getZ();
        return(new Vector3D(x,y,z));
    }

    /**
     * Creates a vector which is equal to the subtraction of the two given vectors.
     *
     * @param v1 a vector.
     * @param v2 a vector.
     *
     * @return a vector which is equal to the subtraction of the two given vectors.
     */
    public static Vector3D subtract(Vector3D v1, Vector3D v2){
        return(new Vector3D(v1.getX()-v2.getX(), v1.getY()-v2.getY(), v1.getZ()-v2.getZ()))
            ;
    }

    /**
     * Creates a vector which is equal to the given vector added by the given value.
     *
     * @param v a vector.
```

```
     * @param c a real value.
     *
     * @return a vector which is equal to the given vector added by the given value.
     */
    public static Vector3D add(Vector3D v, double c){
        return(new Vector3D(v.getX()+c, v.getY()+c, v.getZ()+c));
    }

    /**
     * Creates a vector which is equal to the multiplication of the two given vectors.
     *
     * @param v1 a vector.
     * @param v2 a vector.
     *
     * @return a vector which is equal to the multiplication of the two given vectors.
     */
    public static Vector3D multiply(Vector3D v1, Vector3D v2){
        return(new Vector3D(v1.getX() * v2.getX(), v1.getY() * v2.getY(), v1.getZ() * v2.
            getZ()));
    }

    /**
     * Creates a vector which is equal to the given vector multiplied by the given value.
     *
     * @param v a vector.
     * @param c a real value.
     *
     * @return a vector which is equal to the given vector multiplied by the given value.
     */
    public static Vector3D multiply(Vector3D v, double c){
        return( new Vector3D(v.getX()*c, v.getY()*c, v.getZ()*c));
    }

    /**
     * Creates a vector which is equal to the given vector divided by the given value.
     *
     * @param v a vector.
     * @param c a real value.
     *
     * @return a vector which is equal to the given vector divided by the given value.
     */
    public static Vector3D divide(Vector3D v, double c){
        return( new Vector3D(v.getX()/c, v.getY()/c, v.getZ()/c));
    }

    /**
     * Caculates the dot product of the two given vectors.
     *
     * @param v1 a vector.
     * @param v2 a vector.
     *
     * @return the dot product of the two given vectors.
     */
    public static double dotProduct(Vector3D v1, Vector3D v2){
        return (v1.getX()*v2.getX() + v1.getY()*v2.getY() + v1.getZ()*v2.getZ());
    }

    /**
     * Caculates the cross product of the two given vectors.
     *
     * @param v1 a vector.
     * @param v2 a vector.
     *
     * @return the cross product of the two given vectors.
```

```
     */
    public static Vector3D crossProduct(Vector3D v1, Vector3D v2){
        return(new Vector3D( (v1.getY()*v2.getZ())-(v1.getZ()*v2.getY()),
                (v1.getZ()*v2.getX())-(v1.getX()*v2.getZ()),
                (v1.getX()*v2.getY())-(v1.getY()*v2.getX()) ));
    }

    /**
     * Creates a vector which is equal to the normalized vector of the given vector.
     *
     * @param v a vector.
     * @return a vector which is equal to the normalized vector of the given vector.
     */
    public static Vector3D normalize(Vector3D v){
        double size = v.getNorm();
        return(new Vector3D(v.getX()/size, v.getY()/size, v.getZ()/size));
    }

    /**
     * Prints an error message.
     *
     * @param S1 a string representing the error.
     * @param S2 a string representing the place of the error.
     */
     public static void errMassage(String S1, String S2)
    {
        System.out.println("\n***************************************************");
        System.out.println("Error ocurred inside "+S2);
        System.out.println(S1);
        System.out.println("***************************************************\n");
    }

    /**
     * Prints a critical error message.
     *
     * @param S1 a string representing the error.
     * @param S2 a string representing the place of the error.
     */
    public static void errMassageExit(String S1, String S2)
    {
        System.out.println("\n****************critical error********************");
        System.out.println("Error ocurred inside "+S2);
        System.out.println(S1);
        System.out.println("Exiting from the Program");
        System.out.println("***************************************************\n");
    }
}
```

**CellularOcTree.java:** CellularOcTree is defined based on the region octree, and it is used to model the substrate using the top-down approach.

```
package OcTreeADT;

import Objects3D.Vector3D;

import Objects3D.VectorOperation;

import com.sun.org.apache.xml.internal.utils.SystemIDResolver;

import mySimulation.Particle; import SystemEnvironment.Cell;

import SystemEvents.EventPSCollision.*;

import SystemEvents.EventSphereCubeCollision.CollisionPacket;

import SystemEvents.*;

import mySimulation.SimulationManager;

import java.util.*;

import javax.swing.*;

import OcTreeADT.OctNode.Type;

public class CellularOcTree {
    /**
     * The direction of the neighboring cells with respect to the cell.
     * <p>
     * L: left, R: right, D: Down, U: Up, B: Back, F: Front
     */
    public static enum Dir {L,R,D,U,B,F,LD,LU,LB,LF,RD,RU,RB,RF,DB,DF,UB,UF,LDB,LDF,LUB,LUF
        ,RDB,RDF,RUB,RUF, none}

    /**
     * The density of the target surface.
     */
    private double surfaceDensity;

    /**
     * The firction coefficient of the target surface.
     */
    private double friction;

    /**
     * An experimentally measured constant to calculate erosion rate.
     */
    private double D;

    /**
     * An experimentally measured constant to calculate erosion rate.
     */
    private double K;

    /**
     * The variable determining the type of erosive system. It is set to true for brittle
        erosivr systems
     * and set to false for ductile erosive systems.
```

```
 */
private boolean brittle;

/**
 * A experimentally determined constant used for dutile erosive systems to calculate
     the volume of the material removed from the substrate.
 */
private double n1;

/**
 * A experimentally determined constant used for dutile erosive systems to calculate
     the volume of the material removed from the substrate.
 */
private double n2;

/**
 * The hardness of the target material used in the case of dutile erosive systems.
 */
private double Hv;

/**
 * The depth of the substrate material.
 */
private double substrateDepth;


/**
 * The edge of the cube representing the substrate.
 */
protected double worldSize;

/**
 * The edge of the surface cells.
 */
private static double cellSize;

/**
 * The half edge of the surface cells.
 */
private static double minHalfSize;

/**
 * The value used to determins the surface cells.
 */
public static double testCELL;

/**
 * The depth of the cellular octree.
 */
long totalDepth = 0;

/**
 * The root of the cellular octree.
 */
private OctNode ocRoot;

/**
 * Creates a new instance of CellularOcTree.
 *
 * @param worldSize the edge of the cubic substrate.
 * @param envDepth the depth of the environment.
 * @param surfaceDensity the density of the target material.
 * @param friction the friction coefficient of the surface.
 * @param D a contant used to calculate the erosion rate.
```

```
     * @param K a contant used to calculate the erosion rate.
     * @param brittle a flag which is true for the case of brittle erosive systems and
         false in the case of ductile erosive systems.
     * @param n1 an experimentally determined constant used for dutile erosive systems to
         calculate the volume of the material removed from the substrate.
     * @param n2 an experimentally determined constant used for dutile erosive systems to
         calculate the volume of the material removed from the substrate.
     * @param Hv the hardness of the target material used in the case of dutile erosive
         systems.
     * @param substrateDepth
     */
    public CellularOcTree(double worldSize, double envDepth, double surfaceDensity, double
        friction, double D, double K,
            boolean brittle, double n1, double n2, double Hv, double substrateDepth)
    {
        this.worldSize = worldSize;
        this.minHalfSize = this.cellSize/2.0;
        this.testCELL = getMinHalfSize() + (getMinHalfSize()/2.0);
        double worldHalfSize = worldSize/2.0;
        ocRoot = new OctNode(this, null, worldHalfSize, worldHalfSize,
                worldHalfSize+envDepth, Type.BLACK, -1);
        this.surfaceDensity = surfaceDensity;
        this.friction = friction;
        this.D = D;
        this.K = K;
        this.brittle = brittle;
        this.n1 = n1;
        this.n2 = n2;
        this.Hv = Hv;
        this.substrateDepth = substrateDepth;
    }

    /**
     * An auxiliary variable used by the SurfaceAdvancement method.
     */
    static int methodCounter;

    /**
     * Decomposes the substrate to obtain the target cell and models the surface
         advancement.
     *
     * @param particle  the particle impacting the surface.
     * @param target  the cell collided by the particle.
     * @return true if successful ps-collision occurs and false if unsuccessful ps-
         collision occurs.
     */
    public boolean SurfaceAdvancement(Particle particle,OctNode target) {
        methodCounter++;
        int i;
        OctNode targetCell;
        Vector3D normal = null;
        Vector3D contactPoint = particle.getEvent().getPSCollision().getContactPoint();

        if(target==null)
            targetCell = particle.getEvent().getPSCollision().getTargetBox();
        else
            targetCell = target;

        if(targetCell.getHalfSize()>this.testCELL && !targetCell.containsPoint(contactPoint
            ))
            targetCell=this.ocRoot;

        if(targetCell.getOcType()==Type.WHITE)
            return false;
```

```
if(targetCell.getOcType()==Type.UNDEFINED)
    Objects3D.VectorOperation.errMassageExit("Cell status in undefined","Octree/
        insertWhiteBox()");

while(targetCell.getOcType()==Type.GRAY || (targetCell.getOcType()==Type.CELL &&
    targetCell.getHalfSize()>=this.testCELL))
    targetCell = targetCell.getOctantContainsPoint(contactPoint);

/**
 * When the loop stops the nodes are decomposed to the point that the cell
 * is reached.
 */
if(targetCell.getOcType()!=Type.WHITE)
{
    while(targetCell.getHalfSize()>=this.testCELL)
    {
        if(targetCell.getOcType()==Type.BLACK)
        {

            // that is the first time that cell is hitted by a particle so its
                status is changed to CELL
            targetCell.setOcType(Type.CELL);

            for(i=0; i<8; i++)
            {
                if(targetCell.getChild()[i]!=null)
                    Objects3D.VectorOperation.errMassageExit("targetCell.getChild()
                        [ "+ i +"]!=null! for black box: \n"+ targetCell,
                            "OcTree/applyPSCollision()");

                targetCell.getChild()[i] = new OctNode(this, targetCell, i, Type.
                    BLACK);
            }
            targetCell = targetCell.getOctantContainsPoint(contactPoint);
        }
        else
            Objects3D.VectorOperation.errMassageExit("at this point box must be
                black!!!","Octree/insertWhiteBox()");
    }
}


if(targetCell.getOcType()==Type.BLACK || targetCell.getOcType()==Type.CELL)
    EventPSCollision.setParticleTargetBox(targetCell);
else if(targetCell.getOcType()==Type.WHITE)
{
    if(methodCounter==1)
    {
        if(fittingIntersectionPoint_WhiteBox(contactPoint,targetCell))
            return(SurfaceAdvancement(particle,null));
        else
            return false;
    }
    else if(methodCounter==2)
        return false;
}
else
    Objects3D.VectorOperation.errMassage("The cell should not be gray!!! ","OcTree/
        insertWhiteBoxVersion2()");

// merging the white cells
while(targetCell.parent.merging())
    targetCell = targetCell.parent;
```

```
        return true;
    }

    /**
     * This method is implemented to due the precisions of real numbers.
     * It invoked when the collision point, P, is not contained in the area of the cell and
         shifts the collision point.
     *
     * @param P the particle impacting the surface.
     * @param cell the cell collided by the particle.
     * @return true if the point has been shifted; false otherwise.
     */
    private boolean fittingIntersectionPoint_WhiteBox(Vector3D P, OctNode cell) {
        int u = 0;
        boolean shifted = false;
        if(Math.abs(P.getX()-(cell.getPosition().getX()-cell.getHalfSize()))<=0.0000001)
        {
            P.setX(P.getX()-this.getMinHalfSize());
            shifted = true;
        }
        else if(Math.abs(P.getX()-(cell.getPosition().getX()+cell.getHalfSize()))
            <=0.0000001)
        {
            P.setX(P.getX()+this.getMinHalfSize());
            shifted = true;
        }

        if(Math.abs(P.getY()-(cell.getPosition().getY()-cell.getHalfSize()))<=0.0000001)
        {
            P.setY(P.getY()-this.getMinHalfSize());
            shifted = true;
        }
        else if(Math.abs(P.getY()-(cell.getPosition().getY()+cell.getHalfSize()))
            <=0.0000001)
        {
            P.setY(P.getY()+this.getMinHalfSize());
            shifted = true;
        }
        if(P.getZ()>=(SimulationManager.AJM_Environment.getDepth()+this.getMinHalfSize())
            &&
                Math.abs(P.getZ()-(cell.getPosition().getZ()-cell.getHalfSize()))
                    <=0.0000001)
        {
            P.setZ(P.getZ()-this.getMinHalfSize());
            shifted = true;
        }
        else if(P.getZ()>=(SimulationManager.AJM_Environment.getDepth()+this.getMinHalfSize
            ()) &&
                    Math.abs(P.getZ()-(cell.getPosition().getZ()+cell.getHalfSize()))
                        <=0.0000001)
        {
            P.setZ(P.getZ()+this.getMinHalfSize());
            shifted = true;
        }
        return shifted;
    }

    /**
     * Recieves a GRAY cube and detects the collision between the particle, P, and the cube
         .
     *
     * @param infoPack a storage used to save the information of detected ps-collision.
     * @param P the particle.
     * @param grayCube The node of the cellular tree with the GRAY status.
```

```
 */
private void predicPSCollisionGRAYBox(CollisionPacket infoPack, Particle P, OctNode
    grayCube) {
    CollisionPacket temp = new CollisionPacket(P);

    // if the collision point is contained inside the substrate the method returns
    if(isAcceptablePoint(infoPack.targetBox, infoPack.intersectionPoint))
        return;
    else
        infoPack.reset();

    if(grayCube.getOcType()!=Type.GRAY)
        Objects3D.VectorOperation.errMassageExit("is not gray!!! grayBox: " + grayCube
            ,"OcTree/predicPSCollisionGRAYBox");

    for(int i=0; i<8; i++)
    {
        if(grayCube.getChild()[i]==null)
            Objects3D.VectorOperation.errMassageExit("child is nullllll---grayBox: " +
                grayCube ,"OcTree/predicPSCollisionGRAYBox");

        else if(grayCube.getChild()[i].getOcType()==Type.BLACK || grayCube.getChild()[i
            ].getOcType()==Type.CELL)
            EventSphereCubeCollision.detectSphereCubeCollision(infoPack,grayCube.
                getChild()[i], P);

        else if(grayCube.getChild()[i].getOcType()==Type.GRAY)
        {
            temp.reset();
            if(EventSphereCubeCollision.detectSphereCubeCollision(temp,grayCube.
                getChild()[i], P))
            {
                this.predicPSCollisionGRAYBox(temp, P, grayCube.getChild()[i]);
                if(temp.foundCollision && temp.nearestTime<infoPack.nearestTime)
                    infoPack.set(temp);
            }
        }
    }
}

/**
 * Moves the particle backward and invokes the predict_PSCollision method to
 * detect collision between the particle and target surface.
 *
 * @param P the particle.
 * @return false if any unusual situation occures; true otherwise.
 */
public boolean predictPSCollision(Particle P)
{
    double t = (0.5)*P.getRadius()/(P.getLinearVelocity().getNorm());
    mySimulation.Counters.predictPSColl++;
    /**
     * move particle, P, back for a distance equal to half of its radius. This is done
         due to the
     * problems of working with real number. Because the particle, P, might be
         overlapped inside
     * the substrate(for a high precision).
     */
    P.updatingParticlePosition(-t);
    boolean result = predict_PSCollision(P);
    if(result)
        P.updatingParticlePosition(t);
    mySimulation.Counters.predictPSColl--;
    return result;
```

```
}

/**
 * Detects the collision between particle and substrate.
 *
 * @param P the particle.
 * @return false if any unusual situation occures; true otherwise.
 */
public boolean predict_PSCollision(Particle P)
{
    CollisionPacket temp = new CollisionPacket(P);
    CollisionPacket infoPack = new CollisionPacket(P);
    OctNode Box = this.ocRoot;
    boolean result=true;

    // for the case that the particle, P, is above the cube containing the substrate
    if(!Box.embedded(P))
    {
        /**
         * check to see if particle is colliding with the cube containing the whole
             substrate,
         * if not, obviously particle is not colliding the substrate.
         */
        EventSphereCubeCollision.detectSphereCubeCollision(infoPack, Box, P);
        if(!infoPack.foundCollision)
            return true;

        // if the target box is BLACK or CELL the target point is found
        if(Box.getOcType()==Type.BLACK || Box.getOcType()==Type.CELL)
        {
            P.getEvent().getPSCollision().setInfo(Box, P.getParticleTime()+infoPack.
                nearestTime, infoPack.intersectionPoint,
                    infoPack.planeNormal,infoPack.element, infoPack.v1, infoPack.v2);
            return true;
        }

        // if the Box is GRAY, the collision test is check againt its octant
        else if(Box.getOcType()==Type.GRAY)
        {
            this.predicPSCollisionGRAYBox(infoPack,P,Box);

            if(infoPack.foundCollision)
            {
                mySimulation.Counters.foundPSCollEmbed++;
                P.getEvent().getPSCollision().setInfo(infoPack.targetBox, P.
                    getParticleTime()+infoPack.nearestTime, infoPack.intersectionPoint,
                        infoPack.planeNormal,infoPack.element, infoPack.v1, infoPack.v2
                            );
            }
            else
            {
                P.setUnusualSituation(true, "PSDetection-notEmbedded-faild");
                return false;
            }
            return true;
        }
        // catch error
        else if(Box.getOcType()==Type.WHITE)
        {
            Objects3D.VectorOperation.errMassageExit("the Box is WHITE: "+Box,
                    "Octree/predictPSCollision()");
        }
        return true;
    }
```

```
// for the case that the particle, P, is embedded inside the cube containing the
    substrate
else
{
    List list = new LinkedList();
   // List embeddedCellList = new LinkedList();
    whiteBoxCount = 0;
    embeddedCellsCount = 0;

    // making list out of the boxes that do not have overlap with particle, P
    this.makingListOutOfContainingBox(P,Box,list,infoPack);

    // if there is collision between particle, P, and a embedded Box.
    if(infoPack.foundCollision)
        return true;

    /**
     * This part implemented to catch errors due to the problem of working with
       real
     * numbers. It checks if the particle is stucked inside the substrate.
     */
    if(embeddedCellsCount>0)
    {
        if(whiteBoxCount==0)
            {

                if(P.getZ()-P.getRadius()>SimulationManager.AJM_Environment.
                    getDepth())
                {
                    P.setUnusualSituation(true,"embedded.size>0_but_whiteBoxCount
                        ==0");
                    return false;
                }
            }
    }

    OctNode curr;
    // detect the collision from the embedded Box and their octants
    for(int i=0;i<list.size();i++)
    {
        curr = (OctNode)list.get(i);

        if(curr.getOcType()==Type.BLACK || curr.getOcType()==Type.CELL)
            EventSphereCubeCollision.detectSphereCubeCollision(infoPack, curr, P);

        else if(curr.getOcType()==Type.GRAY)
        {
            temp.reset();
            if(EventSphereCubeCollision.detectSphereCubeCollision(temp, curr, P))
            {
                this.predicPSCollisionGRAYBox(temp,P,curr);
                if(temp.foundCollision && temp.nearestTime<infoPack.nearestTime)
                    infoPack.set(temp);
            }
        }
    }

    // if collision was found the Event instance of the particle is set
    if(infoPack.foundCollision)
    {
        mySimulation.Counters.foundPSCollEmbed++;
        P.getEvent().getPSCollision().setInfo(infoPack.targetBox, P.getParticleTime
            ()+infoPack.nearestTime, infoPack.intersectionPoint,
                infoPack.planeNormal,infoPack.element, infoPack.v1, infoPack.v2);
```

```
        }
        return true;
    }
}

/**
 * Determines if the detected collision point is contained in the substrate.
 *
 * @param grayCube the gray cube.
 * @param point detected collision point.
 * @return true if the point is contained in the gray cube; false otherwise.
 */
public boolean isAcceptablePoint(OctNode grayCube, Vector3D point){
    while(grayCube.getChild()[0]!=null)
        grayCube = grayCube.getOctantContainsPoint(point);

    if(grayCube.getOcType()==Type.GRAY || grayCube.getChild()[0]!=null)
        Objects3D.VectorOperation.errMassageExit("wrong found box!!! " + grayCube,"
            Octree/checkPointColorInGrayBox");

    if(grayCube.getOcType()==Type.BLACK || grayCube.getOcType()==Type.CELL)
        return true;

    return false;
}

/**
 * An auxiliary variable used to detect errors for ps-collisions.
 */
public int whiteBoxCount=0;

/**
 * An auxiliary variable used to detect errors for ps-collisions.
 */
public int embeddedCellsCount = 0;

/**
 * Makes a list of cubes that have potential to be impacted by the particle.
 *
 * @param P the partilce.
 * @param Cube a cube corresponding to a node of the cellular octree.
 * @param list a list to be filled up by cubes.
 * @param infoPack a storage used to save the information of detected ps-collision.
 */
private void makingListOutOfContainingBox(Particle P, OctNode Cube, List list,
    CollisionPacket infoPack)
{
    if(!Cube.embedded(P))
    {
        /**
         * if Box does not contain the Particle P and is NOT white, Box is added to
         * the list; but if it is WHITE the method just returns.
         */
        if(Cube.getOcType()!=Type.WHITE)
        {
            if(EventSphereCubeCollision.IsPossibilityForCollision(Cube,P))
                list.add(Cube);
        }
        else
            whiteBoxCount++;
        return;
    }
    else
    {
```

```
        if(Cube.getOcType()==Type.BLACK || Cube.getOcType()==Type.CELL)// && P.
            getLastVelocity().getZ()>0)
        {
            EventSphereCubeCollision.detectSphereCubeCollision(infoPack, Cube, P);
            if(infoPack.foundCollision)
            {
                P.getEvent().getPSCollision().setInfo(infoPack.targetBox, P.
                    getParticleTime(), infoPack.intersectionPoint,
                        infoPack.planeNormal,infoPack.element, infoPack.v1, infoPack.v2
                            );
                return;
            }
            embeddedCellsCount++;
            return;
        }
        if(Cube.getOcType()==Type.WHITE)
            whiteBoxCount++;
        else if(Cube.getChild()[0]==null)
            System.err.println("it shouldnt be null!" + Cube);
        else if(Cube.getChild()[0]!=null || Cube.getOcType()!=Type.WHITE)
        {
            for(int i=0;i<8;i++)
                this.makingListOutOfContainingBox(P,Cube.getChild()[i], list,infoPack);
        }
        return;
    }
}

 /**
  * Obtains the smallest cube coressponding to a node of the cellular octree containing
      the point, P.
  *
  * @param P a point.
  * @return the smallest cube containing P.
  */
 public OctNode getTargetBox(Vector3D P){
     OctNode Box = this.ocRoot;
     while(Box!=null && Box.getChild()!=null && Box.getChild()[0]!=null)
        Box = Box.getOctantContainsPoint(P);
     return Box;
 }

 /**
  * An auxiliary variable used to detect errors for ps-collisions.
  */
  public static int testCount = 0;

  /**
   * The ID of the last particle that impacted the target surface.
   */
  public static int prevID=-1;

/**
 * Invokes the SurfaceAdvancement() and do some tests to catch and handle possible
     errors.
 *
 * @param P the particle impacting the surface.
 * @return true if no error detected; false otherwise.
 */
 public boolean removeCellFromOctree(Particle P){
    methodCounter=0;
    if(prevID==P.getID())
        testCount++;
    else
```

```
        {
            testCount=0;
            prevID = P.getID();
        }
        if(testCount>1000)
        {
            P.setUnusualSituation(true,"repeatedCollDetection-more-than100");
            return false;
        }
        boolean result = SurfaceAdvancement(P,null);
        return(result);
    }

    public OctNode getOcRoot() {
        return ocRoot;
    }

     /**
     * Returns a string representation of the cellular octree root.
     *
     * @return a string representation of the cellular octree root.
     */
    public String toString(){
        return("ocRoot: " + this.ocRoot);
    }

    public static double getCellSize() {
        return cellSize;
    }

    public static void setCellSize(double aCellSize) {
        cellSize = aCellSize;
    }

    public static double getMinHalfSize() {
        return minHalfSize;
    }

    public double getSurfaceDensity() {
        return surfaceDensity;
    }

    public double getFriction() {
        return friction;
    }

    public double getD() {
        return D;
    }

    public double getK() {
        return K;
    }

    public boolean isBrittle() {
        return brittle;
    }

    public double getN1() {
        return n1;
    }

    public double getN2() {
        return n2;
```

```
    }

    public double getHv() {
        return Hv;
    }

    public double getSubstrateDepth() {
        return substrateDepth;
    }
}
```

# OctNode.java: OctNode implements the nodes of the cellular octree.

```
package OcTreeADT;

import Objects3D.Vector3D;

import Objects3D.VectorOperation;

import mySimulation.Particle;

import SystemEvents.EventPSCollision;

import mySimulation.SimulationManager;

import SystemEvents.EventSphereCubeCollision;

public class OctNode {
    /**
     * The status of nodes.
     */
    public enum Type { GRAY, BLACK, WHITE, CELL, UNDEFINED }

    /**
     * The varaible that store the status of the node.
     */
    private Type ocType;

    /**
     * The parent of the node.
     */
    public OctNode parent = null;

    /**
     * The array that stores the octants of the node.
     */
    private OctNode[] child;

    /**
     * The label of the node with respect to its parent.
     */
    public int index;

    /**
     * The depth of the node in the cellular octree.
     */
    public int depth;

    /**
     * The variable that determines the removed volume of the cell corresponding to this
        node.
     */
    private double loss = 0;

    /**
     * The center of the node.
     */
    private Vector3D position;

    /**
     * The half edge of the cube corresponding to this node.
     */
```

```
private double halfSize;

/**
 * The diameter of the bounding sphere.
 */
public double diameter;

/**
 * The number of impacts on the cell corresponding to this node.
 */
public double numOfColls=0;

/**
 * Creates a new instance of OctNode
 *
 * @param tree the cellular octree.
 * @param p the parent of the new node.
 * @param x the x-coordinate of the position of the cube corresponding to this node.
 * @param y the y-coordinate of the position of the cube corresponding to this node.
 * @param z the z-coordinate of the position of the cube corresponding to this node.
 * @param ocType the status of the new node.
 * @param index the label of the new node with respect to its parent.
 */
public OctNode(CellularOcTree tree, OctNode p, double x, double y, double z, Type
    ocType, int index) {
    parent = p;
    setPosition(new Vector3D(x, y, z));
    if(p!=null)
    {
        setHalfSize(p.getHalfSize() / 2);
        depth = p.depth++;
    }
    else
    {
        setHalfSize(tree.worldSize/2);
        depth = 0;
    }
    // right triangle: a^2 = b^2 + c^2;
    double b = Math.sqrt(getHalfSize()*getHalfSize()*2.0);
    diameter = Math.sqrt( b*b + getHalfSize()*getHalfSize() );
    this.setOcType(ocType);
    child = new OctNode[8];

    for (int j = 0; j < 8; j++)
        if (getChild()[j] != null){
            getChild()[j] = null;
        }
    this.setIndex(index);
}

/**
 * Creates a new instance of OctNode
 *
 * @param tree the cellular octree.
 * @param p the parent of the new node.
 * @param index the label of the new node with respect to its parent.
 * @param ocType the status of the new node.
 */
public OctNode(CellularOcTree tree, OctNode p, int index, Type ocType) {
    parent = p;
    Vector3D ppp = new Vector3D(this.getChildCenter(p, index));
    setPosition(ppp);
    setHalfSize(p.getHalfSize() / 2.0);
    depth = p.depth + 1;
```

```
        double b = Math.sqrt(getHalfSize()*getHalfSize()*2.0);
        diameter = Math.sqrt( b*b + getHalfSize()*getHalfSize() );
        this.setOcType(ocType);
        child = new OctNode[8];

        for (int j = 0; j < 8; j++)
            if (getChild()[j] != null){
                getChild()[j] = null;
            }
        this.setIndex(index);
    }

    /**
     * Calculates the center of a child using the center of its parent.
     *
     * @param parent the parent of the child.
     * @param child the label of the child.
     * @return center of the child node.
     */
    public Vector3D getChildCenter(OctNode parent, int child)
    {
        double quarterSize = parent.getHalfSize()/2.0;
        double x = parent.getPosition().getX(),
               y = parent.getPosition().getY(),
               z = parent.getPosition().getZ();

        if(child==0 || child==2 || child==4 || child==6)
            x -= quarterSize;
        else
            x += quarterSize;

        if(child==0 || child==1 || child==2 || child==3)
            y -= quarterSize;
        else
            y += quarterSize;

        if(child==0 || child==1 || child==4 || child==5)
            z -= quarterSize;
        else
            z += quarterSize;

        return(new Vector3D(x, y, z));
    }

    /**
     * Checks if the cube corresponding to this node contains a given point (with small
       precision).
     *
     * @param P a given point.
     * @return true if the cube contains the point; false otherwise.
     */
    public boolean containsPoint(Vector3D P)
    {
        double pr = SimulationManager.precision*100.0;
        double h = this.getHalfSize()+pr;
        return( P.getX()<=(this.getPosition().getX()+h) && P.getX()>=(this.getPosition().
            getX()-h) &&
                P.getY()<=(this.getPosition().getY()+h) && P.getY()>=(this.getPosition().
                    getY()-h) &&
                P.getZ()<=(this.getPosition().getZ()+h) && P.getZ()>=(this.getPosition().
                    getZ()-h) );
    }

    /**
```

```
 * Checks if the cube corresponding to this node contains a given point.
 *
 * @param P a point.
 * @return true if the cube contains the point; false otherwise.
 */
public boolean embedded(Particle P) {
    double X=P.getPosition().getX(), Y=P.getPosition().getY(), Z=P.getPosition().getZ()
        ;
    double a,b,c,d;

    /**
     * checking if the Particle and Box(this) overlapped in Z axis
     * checking if line a-b and c-d overlapping(a<=c)
     */
    if(Z-P.getRadius()<=this.position.getZ()-this.getHalfSize())
    {
        a=Z-P.getRadius();
        b=Z+P.getRadius();
        c=this.position.getZ()-this.getHalfSize();
    }
    else
    {
        a=this.position.getZ()-this.getHalfSize();
        b=this.position.getZ()+this.getHalfSize();
        c=Z-P.getRadius();
    }
    if(c>=b-0.0000000001)
        return false;

    // checking if the Particle and this octNode overlapped in X axis
    if(X-P.getRadius()<=this.position.getX()-this.getHalfSize())
    {
        a=X-P.getRadius();
        b=X+P.getRadius();
        c=this.position.getX()-this.getHalfSize();
    }
    else
    {
        a=this.position.getX()-this.getHalfSize();
        b=this.position.getX()+this.getHalfSize();
        c=X-P.getRadius();
    }
    if(c>=b-0.0000000001)
        return false;

    // checking if the Particle and this octNode overlapped in Y axis
    if(Y-P.getRadius()<=this.position.getY()-this.getHalfSize())
    {
        a=Y-P.getRadius();
        b=Y+P.getRadius();
        c=this.position.getY()-this.getHalfSize();
    }
    else
    {
        a=this.position.getY()-this.getHalfSize();
        b=this.position.getY()+this.getHalfSize();
        c=Y-P.getRadius();
    }
    if(c>=b-0.0000000001)
        return false;

    return true;
}
```

```
/**
 * Finds the octant of this node containing the given point.
 * <p>
 * Note: upper, left and back boundaries are closed, and lower, right and front
     boundaries are opened
 *
 * @param P a given point
 * @return the octant containing the given point
 */
public OctNode getOctantContainsPoint(Vector3D P) {
    if(this.getChild()[0]==null)
        Objects3D.VectorOperation.errMassageExit("Box info: " + this,"OcCell/
            getOctantContainsPoint()");

    // numbers represent the index of the octant
    // 0, 2, 4, 6
    if( P.getX()<getPosition().getX())
    {
        // 0, 2
        if(P.getY()<getPosition().getY())
        {
            // 0
            if(P.getZ()<getPosition().getZ())
                return getChild()[0];
            // 2
            else
                return getChild()[2];
        }
        // 4, 6
        else
        {
            // 4
            if(P.getZ()<getPosition().getZ())
                return getChild()[4];
            // 6
            else
                return getChild()[6];
        }
    }
    // 1, 3, 5, 7
    else
    {
        // 1, 3
        if(P.getY()<getPosition().getY())
        {
            // 1
            if(P.getZ()<getPosition().getZ())
                return getChild()[1];
            // 3
            else
                return getChild()[3];
        }
        // 5, 7
        else
        {
            // 5
            if(P.getZ()<getPosition().getZ())
                return getChild()[5];
            // 7
            else
                return getChild()[7];
        }
    }
}
```

```
/**
 * Merges the group of eight siblings of the white color.
 *
 * @return true if merging has occurred; false otherwise.
 */
public boolean merging() {
    int i;

    /**
     * check if all the children are white. If not inside for loop the FALSE will be
     * returned by the function
     */
    for(i=0;i<8;i++)
        if(this.child[i].ocType!=Type.WHITE)
            return false;

    // all the children were WHITE, so they will be removed and merged to the one WHITE
        cube.
    for(i=0;i<8;i++)
        this.child[i]=null;
    this.ocType = Type.WHITE;
    return true;
}

public Type getOcType() {
    return ocType;
}

public void setOcType(Type ocType) {
    this.ocType = ocType;
}

public OctNode[] getChild() {
    return child;
}

public void setChild(int i, OctNode child) {
    this.child[i] = child;
}

public Vector3D getPosition() {
    return position;
}

public void setPosition(Vector3D position) {
    this.position = position;
}

public int getIndex() {
    return index;
}

public void setIndex(int index) {
    this.index = index;
}

public double getLoss() {
    return loss;
}

public void setLoss(double loss) {
    this.loss = loss;
}
```

```
    public double getHalfSize() {
        return halfSize;
    }

    public void setHalfSize(double halfSize) {
        this.halfSize = halfSize;
    }

    /**
     * Returns a string representation of the ranges occupied by the cube corresponding to
         this node in three axes.
     *
     * @return the string representing the range of the cube.
     */
    public String getRange(){
       return ("\n  x: " + (position.getX()-this.getHalfSize())+"-"+(position.getX()+this.
           getHalfSize())+
               "\n  y: " + (position.getY()-this.getHalfSize())+"-"+(position.getY()+this.
                   getHalfSize())+
               "\n  z: " + (position.getZ()-this.getHalfSize())+"-"+(position.getZ()+this.
                   getHalfSize())+ "\n");
    }

    /**
     * Returns a string representation of this node.
     *
     * @return a string representation of this node.
     */
    public String toString() {
        int i;
        String s="\n Position: "+this.getPosition()+ "\n Color: " +  this.ocType + "\n
            Index: " +
                this.getIndex() + "\n Depht: " + this.depth + "\n Halfsize: " + this.
                    getHalfSize() + "\n num Of PSColl: " + this.numOfColls+
                "\n loss: " + this.loss +"\n Range:" + this.getRange();
        OctNode p = this.parent;
        s = s + "\n parents(bot->top)";
        s = s + this.getIndex();
        while(p!= null && p.index>-1){
            s = s + p.index;
            p = p.parent;
        }
        return s;
    }
}
```

**Cell.java:** Cell implements the elements of the array representing the cuboid space holding all the particles.

```
package SystemEnvironment;

import OcTreeADT.CellularOcTree;

import Objects3D.Vector3D;

import java.util.*;

import mySimulation.*;

public class Cell {
    /**
     * The first index of the cell in the array representing the space holding particles.
     */
    public int index1;

    /**
     * The second index of the cell in the array representing the space holding particles.
     */
    public int index2;

    /**
     * The third index of the cell in the array representing the space holding particles.
     */
    public int index3;

    /**
     * The size of the cell.
     */
    private Vector3D size;

    /**
     * The top/left/back corner of the cell.
     */
    private Vector3D position;

    /**
     * The list of particles assigned to this cell.
     */
    private List Members;

    /**
     * Creates a new instance of Cell.
     *
     * @param size the size of the cell.
     * @param position  the top/left/back corner of the cell.
     * @param I1 the first index of the cell.
     * @param I2 the second index of the cell.
     * @param I3 the third index of the cell.
     */
    public Cell(Vector3D size, Vector3D position, int I1, int I2, int I3)
    {
        this.size = new Vector3D(size);
        this.position = new Vector3D(position);
        Members = new LinkedList();
        index1 = I1;
        index2 = I2;
```

```
    index3 = I3;
}

/**
 * Checks if the cell either fully or partially contains the given particle.
 *
 * @param P a particle.
 * @return  true if particle fully or partially is contained inside this cell; false
     otherwise.
 */
public boolean contains(Particle P)
{
    if( position.getX()<P.getX() && position.getX()+size.getX()>P.getX() )
    {
        if( position.getY()<P.getY() && position.getY()+size.getY()>P.getY())
        {
            if( position.getZ()<P.getZ() && position.getZ()+size.getZ()>P.getZ())
                return true;
        }
    }
    return false;
}

/**
 * Checkes if particle is the member of this cell.
 *
 * @param P a particle.
 * @return true if particle is the member of this cell; false otherwise.
 */
public boolean isItMember(Particle P){
    return Members.contains(P);
   }

/**
 * Adds the given particle to the Members list.
 *
 * @param P a particle.
 */
 public void addMemberTo(Particle P){
    Members.add(P);
    return;
   }

 /**
 * Deletes particle from thr Members list
 * @param P a particle.
 */
 public void deleteMemberFrom(Particle P){
    Members.remove(P);
    return;
 }

 public List getMembers(){
    return Members;
}

 public Vector3D getCubePosition(){
    return this.position;
   }

 public Vector3D getSize(){
    return this.size;
}
```

```
    /**
    * Returns a string representation of this cell.
    *
    * @return a string representation of this cell.
    */
    public String toString(){
       return("\n   index:    ["+index1+", "+index2+", " + index3 +"]" +
              "\n   poision: " + position +
              "\n   size: " + size );
      }
}
```

**Environment.java:** The Environment class represents the whole space holding particles. The cuboid space holding all particles is divided up into smaller cubes called cells.

```
package SystemEnvironment;

import Objects3D.Vector3D;

import Objects3D.VectorOperation;

import java.lang.Math; import mySimulation.*;

import SystemEvents.Event.enteredCubeType;

import SystemEvents.Event.*;

import SystemEvents.*;

import OcTreeADT.CellularOcTree;

public class Environment {
    /**
     * The width of the cuboid space holding particles.
     */
    private double width;

    /**
     * The height of the cuboid space holding particles.
     */
    private double height;

    /**
     * The depth of the cuboid space holding particles.
     */
    private double depth;

    /**
     * The edge of the non-marginal environment cells.
     */
    private double cellSize;

    /**
     * The size of the non-marginal cells.
     */
    private Vector3D stdCubeSize;

    /**
     * The number of cells of the environment along x, y, and z-axes.
     */
    public static Vector3D numOfCubes;

    /**
     * The array whose element are the cells of the environment.
     */
    private Cell[][][] Space;

    /**
     * Creates a new instance of Environment.
     * A 3D-array is created to hold particles flowing in the boundary of the system.
     *
     * @param envWidth the width of the environment.
```

```
 * @param envHeight the height of the environment
 * @param envDepth the depth of the environment
 * @param cellSize the edge length of the non-marginal cubic cells of the environment.
 */
public Environment(double envWidth, double envHeight, double envDepth, double cellSize)
{
    int roundW, roundH, roundD;
    double exactW, exactH, exactD;
    int i, j, k;
    double xSize=0, ySize=0, zSize=0;
    double xPosition, yPosition, zPosition;
    double leftWidth, leftHeight, leftDepth;
    Vector3D cubeSize = new Vector3D();
    Vector3D cubePosition = new Vector3D();

    this.width = envWidth;
    this.height = envHeight;
    this.depth = envDepth;
    this.cellSize = cellSize;
    stdCubeSize = new Vector3D(cellSize, cellSize, cellSize);

    exactW = envWidth/stdCubeSize.getX();
    exactH = envHeight/stdCubeSize.getY();
    exactD = envDepth/stdCubeSize.getZ();

    roundW = (int)Math.ceil(exactW);
    roundH = (int)Math.ceil(exactH);
    roundD = (int)Math.ceil(exactD);

    leftWidth  =  envWidth  - (double)(roundW-1)*(stdCubeSize.getX());
    leftHeight =  envHeight - (double)(roundH-1)*(stdCubeSize.getY());
    leftDepth  =  envDepth  - (double)(roundD-1)*(stdCubeSize.getZ());

    if( leftWidth<0 || leftHeight<0 || leftDepth<0 )
        Objects3D.VectorOperation.errMassageExit("␣leftWidth<0␣||␣leftHeight<0␣||␣
            leftDepth<0","Environment");

    Space = new Cell[roundW][roundH][roundD];

    xPosition = 0;
    yPosition = 0;
    zPosition = 0;

    for(i=0; i<roundW; i++)
    {
        for(j=0; j<roundH; j++)
        {
            for(k=0; k<roundD; k++)
            {
                if(i == (roundW-1))
                    xSize = leftWidth;
                else
                    xSize = stdCubeSize.getX();

                if(j == (roundH-1))
                    ySize = leftHeight;
                else
                    ySize = stdCubeSize.getY();

                if(k == (roundD-1))
                    zSize = leftDepth + SimulationManager.AJM_Substrate.
                        getSubstrateDepth();
                else
                    zSize = stdCubeSize.getZ();
```

```
                if(k == (roundD-1))
                    cubeSize.setPoint(xSize, ySize, zSize+SimulationManager.
                        AJM_Substrate.getSubstrateDepth());
                else
                    cubeSize.setPoint(xSize, ySize, zSize);

                cubePosition.setPoint(xPosition, yPosition, zPosition);

                zPosition = zPosition + zSize;
                Space[i][j][k] = new Cell(cubeSize, cubePosition, i, j, k);
            }
            yPosition = yPosition + ySize;
            zPosition = 0;
        }
        xPosition = xPosition + xSize;
        yPosition = 0;
    }
    numOfCubes = new Vector3D(roundW, roundH, roundD);
}

/**
 * Findes the environment cell containing a given point.
 *
 * @param P a point.
 * @return the cell containing the given point.
 */
public Cell findCube(Vector3D P)
{
    int Xc, Yc, Zc;
    double Xp=P.getX(), Yp=P.getY(), Zp=P.getZ();
    Xc = Yc = Zc = 0;

    if( (Math.ceil(Xp/stdCubeSize.getX())==Xp/stdCubeSize.getX()) && (Xp!=this.getWidth
        ()) ) Xc=1;
    if( (Math.ceil(Yp/stdCubeSize.getY())==Yp/stdCubeSize.getY()) && (Yp!=this.
        getHeight()) ) Yc=1;
    if( (Math.ceil(Zp/stdCubeSize.getZ())==Zp/stdCubeSize.getZ()) && (Zp!=this.getDepth
        ()) ) Zc=1;

    Xc = Xc+(int)Math.ceil(Xp/stdCubeSize.getX())-1;
    Yc = Yc+(int)Math.ceil(Yp/stdCubeSize.getY())-1;
    Zc = Zc+(int)Math.ceil(Zp/stdCubeSize.getZ())-1;

    if(Zc==numOfCubes.getZ())
        Zc--;
    if(Zc<0)
        Zc=0;
    if(Xc>=numOfCubes.getX() || Yc>=numOfCubes.getY() || Zc>=numOfCubes.getZ() || Xc<0
        || Yc<0 || Zc<0 )
        return null;

    return Space[Xc][Yc][Zc];
}

/**
 * Adds a new particle into a cell of the environment.
 *
 * @param P a new particle.
 * @return true if no error occurrs; false otherwise.
 */
public Cell addNewParticle(Particle P)
{
    Cell cube;
```

```
        cube = findCube(P.getPosition());
        if(cube!=null)
        {
            cube.addMemberTo(P);
            P.setCurrentSpace(cube);
        }
        return cube;
    }

    /**
     * Returns the cell with given indices in the array representing the environment.
     *
     * @param i1 the first index.
     * @param i2 the second index.
     * @param i3 the third index.
     * @return  the cell with the given indices.
     */
    public Cell getSpaceElement(int i1, int i2, int i3){
        if(i3==this.numOfCubes.getZ())
            i3--;
        return Space[i1][i2][i3];
    }

    /**
     * Calculates the total number of members of all the environment cells.
     * @return the total number of members of all the environment cells
     */
    public static double getTotalMembersOfCubes()
    {
        double totnumMem = 0.0;
        int i,j,k;
        for(i=0; i<numOfCubes.getX(); i++)
        {
            for(j=0; j<numOfCubes.getY(); j++)
            {
                for(k=0; k<numOfCubes.getZ(); k++)
                    totnumMem += SimulationManager.AJM_Environment.getSpaceElement(i,j,k).
                        getMembers().size();
            }
        }
        return totnumMem;
    }

    public Vector3D getNumOfCubes() {
        return numOfCubes;
    }

    public double getWidth() {
        return width;
    }

    public double getHeight() {
        return height;
    }

    public double getDepth() {
        return depth;
    }
}
```

**Event.java:** The Event class is created for each particle. This class is used to store the information of the particle's next events.

It has three inner classes implemented:

1. The PPCollisionStorage class is used to store the information of the next inter-particle collision of the particle.

2. The PSCollisionStorage class is used to store the information of the next particle-surface collision of the particle.

3. The transferStorage class is used to store the information of the next transfer of the particle.

```
package SystemEvents;

import OcTreeADT.OctNode;

import OcTreeADT.CellularOcTree;

import Objects3D.Vector3D;

import Objects3D.VectorOperation;

import mySimulation.*;

import SystemEnvironment.Cell;

import Objects3D.Vector3D;

import SystemEvents.EventSphereCubeCollision.CollisionPacket;

import SystemEvents.EventSphereCubeCollision.Element;

public class Event {
    /**
     * The type of the events.
     */
    public enum EventType {PPCollisionType, PSCollisionType, transferType, launchType,
        undefinedType}

    /**
     * The directions that particles can take to enter a new environment cell.
     */
    public enum enteredCubeType {back, backLeft, backTop, backTopLeft, left, top, topLeft, front,
        bottom, right, itself, undefinedCube}

    /**
     * The particle associated with this instance of the Event class.
     */
    private Particle ownerParticle;
```

```
/**
 * the variable that stores the information of the next pp-collision of the particle.
 */
private PPCollisionStorage PPCollision;

 /**
 * the variable that stores the information of the next ps-collision of the particle.
 */
private PSCollisionStorage PSCollision;

 /**
 * the variable that stores the information of the next transfer of the particle
 */
private transferStorage Transfer;

/**
 * Creates a new instance of Event for a given particle.
 *
 * @param P the ownerParticle.
 */
public Event(Particle P)
{
    setOwnerParticle(P);
    PPCollision = this. new PPCollisionStorage();
    PSCollision = this. new PSCollisionStorage();
    Transfer = this. new transferStorage();
}

/**
 * Creates a new instance of Event for ownerParticle and fills up the new event using
     the information of the given event, e.
 *
 * @param e an event.
 */
public Event(Event e)
{
    setOwnerParticle(e.getOwnerParticle());
    PPCollision = e.getPPCollision();
    PSCollision = e.getPSCollision();
    Transfer = e.getTransfer();
}

/**
 * The PPCollisionStorage class is used to store the information of the next inter-
     particle collision of the particle.
 */
public class PPCollisionStorage
{
    /**
     * The particle invloved in the next detected pp-collison of the ownerParticle.
     */
    public Particle InvolvingParticle;

    /**
     * The time of the next pp-collision of the ownerParticle.
     */
    public double PPCollTime;

    /**
     * Creates a new instance of PPCollisionStorage.
     */
    public PPCollisionStorage()
        {
```

```
            PPCollTime = Double.POSITIVE_INFINITY;
            InvolvingParticle = null;
        }

    /**
     * Resets the information of this storage to default values.
     */
    public void resetInfo() {
        PPCollTime = Double.POSITIVE_INFINITY;
        InvolvingParticle = null;
    }

    /**
     * Assigns new values to the variables of this storage.
     *
     * @param p the new particle invloved in the next detected pp-collison of the
       ownerParticle.
     * @param t the time of the next pp-collision for the ownerParticle.
     */
    public void setInfo(Particle p, double t)
    {
       PPCollTime = t;
       InvolvingParticle = p;
    }

    public double getMinTime(){
        return PPCollTime;
    }

    public Particle getInvolvingParticle(){
        return InvolvingParticle;
    }

     /**
     * Returns a string representation of this PPCollisionStorage.
     *
     * @return a string representation of this PPCollisionStorage.
     */
    public String toString(){
        return ("PPCollisionEvent: " + "\n1. time:" +PPCollTime +
                "\n2. involvingParticle position:" + InvolvingParticle.getPosition());
    }
}

public class PSCollisionStorage
{
    /**
     * The time of the next ps-collision of ownerParticle.
     */
    private double minTime;

    /**
     * The collision point for the next ps-collision of ownerParticle.
     */
    private Vector3D contactPoint;

    /**
     * The target cube correspoding to a node of the cellular octree of the next ps-
       collision.
     */
    private OctNode targetBox;

    /**
     * The normal to the local surface at the predicted collision point.
```

```
 */
private Vector3D planeNormal;

/**
 * The element at which ownerParticle hits the target cell in its next ps-collision
 */
private Element element;

/**
 * For the case of impacting on edge or vertices, this variables stores an ends of
     the edge or vertex containing the detected collision point.
 */
private Vector3D v1;

/**
 * For the case of impacting on edge, this variables stores an ends of the edge
     containing the detected collision point.
 */
private Vector3D v2;

/**
 * Creates a new instance of PSCollisionStorage.
 */
public PSCollisionStorage(){
    setMinTime(Double.POSITIVE_INFINITY);
    this.setContactPoint(null);
    this.setPlaneNormal(null);
    element = Element.none;
    v1 = null;
    v2 = null;
}

/**
 * Assigns new values to the variables of this storages.
 *
 * @param targetBox the target cube correspoding to a node of the cellular octree
     of the next ps-collision.
 * @param minTime the time of the next ps-collision.
 * @param contactPoint the collision point of the next ps-collision.
 * @param planeNormal the normal to the local surface for the next ps-collison.
 * @param element the element at which ownerParticle hits the target cell in its
     next ps-collision.
 * @param v1 an end of the edge or vertex containing next ps-collision point. For
     the cases that particle hits the square plane this variable is set to null.
 * @param v2 an end of the edge containing next ps-collision point. For the cases
     that particle hits the square plane or one of the vertices this variable is
     set to null.
 */
public void setInfo(OctNode targetBox, double minTime, Vector3D contactPoint,
    Vector3D planeNormal, Element element, Vector3D v1, Vector3D v2)
{
    this.setTargetBox(targetBox);
    this.setMinTime(minTime);
    this.setContactPoint(contactPoint);
    this.setPlaneNormal(planeNormal);
    this.element = element;
    this.v1 = v1;
    this.v2 = v2;
}

/**
  * Resets the information of this storage to default values.
  */
```

```
public void resetInfo(){
    minTime = Double.POSITIVE_INFINITY;
    contactPoint = null;
    targetBox = null;
    planeNormal = null;
    element = Element.none;
    v1 = null;
    v2 = null;
}

public double getMinTime() {
    return minTime;
}

public void setMinTime(double minTime) {
    this.minTime = minTime;
}

public Vector3D getContactPoint() {
    return contactPoint;
}

public void setContactPoint(Vector3D contactPoint) {
    this.contactPoint = contactPoint;
}

public OctNode getTargetBox() {
    return targetBox;
}

public void setTargetBox(OctNode targetBox) {
    this.targetBox = targetBox;
}

public Vector3D getPlaneNormal() {
    return planeNormal;
}

public void setPlaneNormal(Vector3D planeNormal) {
    this.planeNormal = planeNormal;
}

public Element getElement() {
    return element;
}

public void setElement(Element element) {
    this.element = element;
}

public Vector3D getV1() {
    return v1;
}

public void setV1(Vector3D v1) {
    this.v1 = v1;
}

public Vector3D getV2() {
    return v2;
}

public void setV2(Vector3D v2) {
    this.v2 = v2;
```

```
    }

    /**
     * Returns a string representation of this PSCollisionStorage.
     *
     * @return a string representation of this PSCollisionStorage.
     */
    public String toString(){
        return("\n minTime: " + this.minTime + "\n contactPoint: " + this.contactPoint
            + "\n plane normal: " + this.getPlaneNormal());
    }

}

public class transferStorage
{
    /**
     * The cube that ownerParticle will enter next.
     */
    private enteredCubeType enteredCube;

    /**
     * The time of the next trasner event of ownerParticle.
     */
    private double time;

    /**
     * The time of the ownerParticle before handling the next trasnfer.
     */
    private double prevSystemTime;

    /**
     * The position of the ownerParticle after handling its next transfer event.
     */
    public Vector3D testNextPosition;

    /**
     * Creates a new instance of transferStorage.
     */
    public transferStorage()
    {
        enteredCube = Event.enteredCubeType.undefinedCube;
        time = Double.POSITIVE_INFINITY;
        prevSystemTime = -1;
    }

    /**
      * Resets the information of this storage to default values.
      */
    public void resetInfo(){
        time = Double.POSITIVE_INFINITY;
        enteredCube = Event.enteredCubeType.undefinedCube;
    }

    /**
     * Assigns new values to the variables of this storage.
     *
     * @param t the time of the next transfer for ownerParticle.
     * @param dir the direction that ownerParticle takes to enter a new cell of the
          environment.
     */
    public void setInfo(double t, Event.enteredCubeType dir){
        time = t;
        enteredCube = dir;
```

```
    }

    public double getMinTime(){
        return time;
    }

    public Event.enteredCubeType getEnteredCubeType(){
        return enteredCube;
    }

    public void setPrevSystemTime()
    {
        prevSystemTime = SimulationManager.SystemTime;
    }

    public double getPreSystemTime()
    {
        return prevSystemTime;
    }

    /**
     * Returns a string representation of this transferStorage.
     *
     * @return a string representation of this transferStorage.
     */
    public String toString()
    {
        return ("ExitingCubeEvent: " + "\n1. time: "+time + "\n2. enteredCube: "+
            enteredCube +
        "\n3. prevSystTime: "+ prevSystemTime + "\n4. SystTime: "+ SimulationManager.
            SystemTime);
    }

}

/**
 * Sets the information of the EventNode of the ownerParticle to its earlies event.
 */
public void pickNextEventInfo()
{
    double minTime = Double.POSITIVE_INFINITY;
    EventType type = EventType.undefinedType;

    if(PPCollision == null && PSCollision == null && Transfer == null)
        Objects3D.VectorOperation.errMassageExit("there is not any defined event for
            this particle!",
                                                 "Error! inside Event/pickNextEventInfo");

    if(PPCollision!=null && PPCollision.PPCollTime < minTime)
    {
        minTime = PPCollision.PPCollTime;
        type = EventType.PPCollisionType;
    }

    if(PSCollision!=null && PSCollision.getMinTime()<minTime)
    {
        minTime = PSCollision.getMinTime();
        type = EventType.PSCollisionType;
    }

    if(Transfer!=null && Transfer.getMinTime()<minTime)
    {
        minTime = Transfer.getMinTime();
        type = EventType.transferType;
```

```
        }

        getOwnerParticle().getEventNode().setNextEventNodeInfo(minTime, type);
    }

    /**
     * Resets the information of this event to default values.
     */
    public void resetInfo(){
        getOwnerParticle().getEventNode().resetNextEventNodeInfo();
        this.PPCollision.resetInfo();
        this.Transfer.resetInfo();
        this.PSCollision.resetInfo();
    }

    /**
     *  Update the information of the ownerParticle's event after the trasnfer.
     */
    public void setUpEvent_After_Transfer()
    {
        resetInfo();
        // When the particle is in the boundary and is moving outward we have to remove it
        try
        {
            if(!EventTransfer.detectTransfer(getOwnerParticle()))
            {
                EventTransfer.removeParticle(getOwnerParticle());
                return;
            }



            if(getOwnerParticle().getCurrentSpace().index3 == SimulationManager.
                AJM_Environment.getNumOfCubes().getZ()-1)
            {
                if(!SimulationManager.AJM_Substrate.predictPSCollision(getOwnerParticle()))
                {

                    if(getOwnerParticle().isUnusualSituation() || !EventTransfer.
                        detectTransfer(getOwnerParticle()))
                    {
                        EventTransfer.removeParticle(getOwnerParticle());
                        return;
                    }
                }

            }
        }
        catch(StackOverflowError e){
              System.out.println("\n\n*************************\nP.info: " +
                  getOwnerParticle() +
                   "\nP.event: " + getOwnerParticle().getEvent() +
                   "\ncurrEvent: " + SimulationManager.currEventType);

            getOwnerParticle().setUnusualSituation(true,"StackOverflowError");
            EventTransfer.removeParticle(getOwnerParticle());
            return;
        }
            EventPPCollision.predictEvent(getOwnerParticle());

    }

    /**
     *  Sets the information of the ownerParticle's event after it is launched by the
```

```
     nozzle.
 */
public void setUpEvent_After_Lunching()
{
    resetInfo();
    boolean r = EventTransfer.detectTransfer(getOwnerParticle());
    this.pickNextEventInfo();
    SimulationManager.AJM_EventHeap.insert(getOwnerParticle().getEventNode());
    EventPPCollision.predictEvent(getOwnerParticle());
}

/**
 *  Update the information of the particle that its partner left their common
     environment cells.
 */
public void setUpEvent_After_PartnerLeaving()
{
    getOwnerParticle().getEventNode().resetNextEventNodeInfo();
    this.PPCollision.resetInfo();
    EventPPCollision.predictEvent(getOwnerParticle());
}

/**
 *  Update the information of the ownerParticle's event after the pp-collision.
 */
public void setUpEvent_After_PPCollision()
{
    Particle partner = PPCollision.getInvolvingParticle();
    resetInfo();

    try{
        if(!EventTransfer.detectTransfer(getOwnerParticle()))
        {
            EventTransfer.removeParticle(getOwnerParticle());
            setOwnerParticle(null);
        }
        else
        {
            getOwnerParticle().getEventNode().changeElement_Place_Info(getOwnerParticle
                ());
            if(getOwnerParticle().getCurrentSpace().index3 == SimulationManager.
                AJM_Environment.getNumOfCubes().getZ()-1)
            {
                if(getOwnerParticle().isUnusualSituation() || !SimulationManager.
                    AJM_Substrate.predictPSCollision(getOwnerParticle()))
                {
                    if(!EventTransfer.detectTransfer(getOwnerParticle()))
                    {
                        EventTransfer.removeParticle(getOwnerParticle());
                        setOwnerParticle(null);
                    }
                }
            }
        }
    }
    catch(StackOverflowError e){
        System.out.println("\n\n**************************\nP.info: " +
            getOwnerParticle() +
               "\nP.event: " + getOwnerParticle().getEvent() +
               "\ncurrEvent: " + SimulationManager.currEventType);
        getOwnerParticle().setUnusualSituation(true,"StackOverflowError");
        EventTransfer.removeParticle(getOwnerParticle());
        setOwnerParticle(null);
    }
```

```
    if(partner==null || partner.getCurrentSpace()==null)
    {
        if(partner.getEventNode()!=null)
            EventTransfer.removeParticle(partner);
        partner=null;
    }
    else
    {
        partner.getEvent().resetInfo();
        try{
            if(!EventTransfer.detectTransfer(partner))
            {
                EventTransfer.removeParticle(partner);
                partner=null;
            }
            else
            {
                partner.getEventNode().changeElement_Place_Info(partner);
                if(partner.getCurrentSpace().index3 == SimulationManager.
                    AJM_Environment.getNumOfCubes().getZ()-1)
                {
                    if(partner.isUnusualSituation() || !SimulationManager.AJM_Substrate
                        .predictPSCollision(partner))
                    {
                        if(!EventTransfer.detectTransfer(partner))
                        {
                            EventTransfer.removeParticle(partner);
                            partner=null;
                        }
                    }

                }
            }
        }
        catch(StackOverflowError e){
            partner.setUnusualSituation(true,"StackOverflowError");
            EventTransfer.removeParticle(partner);
            partner = null;
        }
    }
    EventPPCollision.predictEvent(getOwnerParticle(), partner);
}

/**
 * Update the information of the ownerParticle's event after the ps-collision.
 */
public void setUpEvent_After_PSCollision()
{
    resetInfo();

    try
    {
        if(!EventTransfer.detectTransfer(getOwnerParticle()))
        {
            EventTransfer.removeParticle(getOwnerParticle());
            return;
        }

        if(getOwnerParticle().getCurrentSpace().index3 == SimulationManager.
            AJM_Environment.getNumOfCubes().getZ()-1)
        {
            if(getOwnerParticle().isUnusualSituation() || !SimulationManager.
                AJM_Substrate.predictPSCollision(getOwnerParticle()))
```

```
                {
                    if(!EventTransfer.detectTransfer(getOwnerParticle()))
                    {
                        EventTransfer.removeParticle(getOwnerParticle());
                        return;
                    }
                }

            }
        }
        catch(StackOverflowError e){
            System.out.println("\n\n**************************\nP.info: " +
                getOwnerParticle() +
                    "\nP.event: " + getOwnerParticle().getEvent() +
                    "\ncurrEvent: " + SimulationManager.currEventType);
            getOwnerParticle().setUnusualSituation(true,"StackOverflowError");
            EventTransfer.removeParticle(getOwnerParticle());
            return;
        }
        EventPPCollision.predictEvent(getOwnerParticle());
    }

    public Particle getOwnerParticle(){
        return ownerParticle;
    }

    public void setOwnerParticle(Particle ownerParticle) {
        this.ownerParticle = ownerParticle;
    }

    public PPCollisionStorage getPPCollision(){
        return PPCollision;
    }

    public PSCollisionStorage getPSCollision(){
        return PSCollision;
    }

    public transferStorage getTransfer(){
        return Transfer;
    }

    /**
     * Returns a string representation of this event.
     *
     * @return a string representation of this event.
     */
    public String toString(){
        return("\n    PPColl: " + this.PPCollision.PPCollTime + "\n    PSColl: "+ this.
            PSCollision.minTime +
                "\n    ExitintCube: "+ this.Transfer.time +"    enteredCube: "+this.
                    Transfer.enteredCube);
    }
}
```

**EventHeap.java:** The EventHeap class implements the event queue which is used to store the information of the predicted events of particles. This class provides methods to perform the heap operations, such as inserting a new element, deleting an element, and updating the position of an element.

```
package SystemEvents;

import java.lang.Math;

import mySimulation.Particle;

import SystemEvents.EventNode.State;

import mySimulation.SimulationManager;

public class EventHeap {
    /**
     * The reet of the heap.
     */
    EventNode heapRoot;

    /**
     * Deapest level ( heapRoot is in level 0)
     */
    private int lastLevel;

    /**
     * The total number of heap nodes.
     */
    private static int memberCounter;

    /**
     * The maximum possible number of nodes in the last level.
     */
    private static int possibleNodes;

    /**
     * The variable used to store the last leaf of the heap.
     */
    private EventNode lastNode;

    /**
     * The variable used to store the bottommost, leftmost node of the heap
     */
    private EventNode bottomLeftNode;

    /**
     * An auxiliary variable used to keep the heap complete.
     */
    private EventNode bottomRightNode;

    /**
     * An auxiliary variable used to handle errors.
     */
    public  EventNode secondNodeL;

    /**
```

```
 * An auxiliary variable used to handle errors.
 */
public  EventNode secondNodeR;

/**
 * Creates a new instance of EventHeap
 */
public EventHeap()
{
    heapRoot = null;
    lastLevel = -1;
    memberCounter = 0;
    possibleNodes = 0;
    lastNode = null;
    bottomLeftNode = null;
    bottomRightNode = null;
}

/**
 * Inserts a new node to the heap.
 *
 * @param newNode a new node.
 */
public void insert(EventNode newNode)
{
    EventNode currNode;

    /*****************************
     * 1. When the Heap is empty *
     *****************************/
    if(heapRoot==null)
    {
        //System.out.println("if(heapRoot==null)---newNode Value: " + newNode.
            getTimeValue());
        if(memberCounter!=0)
            Objects3D.VectorOperation.errMassageExit("ERRORRRR! heapRoot==NULL but
                memberCounter!=0",
                    "EventHeap/insert");

        /* updating newNode information */
        newNode.setIndex(0);
        newNode.setLevel(0);
        newNode.setState(State.ROOT);

        heapRoot = newNode;

         /* updating Heap information */
        lastLevel = 0;
        memberCounter++;
        possibleNodes = 1;
        lastNode = newNode;
        bottomLeftNode = newNode;
        bottomRightNode = newNode;
        return;
    }

    /*************************************************************
     * 2. When last level is full and we have to jump to the new level *
     *************************************************************/
    else if(memberCounter == possibleNodes)
    {
        currNode = bottomLeftNode;

        /* updating newNode information */
```

```
        newNode.setIndex(memberCounter);
        newNode.setLevel(currNode.getLevel()+1);
        newNode.setState(State.LEFT);
        newNode.setParent(currNode);

        currNode.setLeftChild(newNode);

        /* updating Heap information */
        lastLevel++;
        memberCounter++;
        possibleNodes = possibleNodes + (int)Math.pow(2,lastLevel);
        lastNode = newNode;
        bottomLeftNode = newNode;
    }

    /**
     * 3. Lastlevel is not full & lastNode is LeftChild, So the RightChild place of its
          Parent is empty
     */
    else if(lastNode.getState()==State.LEFT)
    {
        if(lastNode.getParent().getRightChild()!=null)
            Objects3D.VectorOperation.errMassageExit("1.ERRORRRRRR!(inside INSERT)",
                    "EventHeap/insert");

        currNode = lastNode.getParent();

        /* updating newNode information */
        newNode.setIndex(memberCounter);
        newNode.setLevel(currNode.getLevel()+1);
        newNode.setState(State.RIGHT);
        newNode.setParent(currNode);

        currNode.setRightChild(newNode);

        /* updating Heap information */
        memberCounter++;

        if(memberCounter==possibleNodes)
            bottomRightNode = newNode;
        lastNode = newNode;
    }

    /*************************************************************
     * 4. When last level is not full and lastNode is the RightChild *
     *************************************************************/
    else if(lastNode.getState()==State.RIGHT)
    {
        if(lastNode.getParent().getLeftChild()==null)
            Objects3D.VectorOperation.errMassageExit("2.ERRORRRRRR! (inside INSERT)",
                    "EventHeap/insert");

        currNode = lastNode;
        while(currNode.getState()==State.RIGHT)
            currNode = currNode.getParent();

        currNode = currNode.getParent().getRightChild();

        while(currNode.getLeftChild()!=null)
            currNode = currNode.getLeftChild();

        /* updating newNode information */
        newNode.setIndex(memberCounter);
        newNode.setLevel(currNode.getLevel()+1);
```

```
            newNode.setState(State.LEFT);
            newNode.setParent(currNode);

            currNode.setLeftChild(newNode);

             /* updating Heap information */
            memberCounter++;
            lastNode = newNode;
        }
        else
            Objects3D.VectorOperation.errMassageExit("The Insertion is impossible!!!",
                        "EventHeap/insert");

        currNode = lastNode;
        while( currNode.getIndex()!=0 && currNode.getTime()<currNode.getParent().getTime())
        {
            currNode.swapNode(currNode.getParent());
            currNode = currNode.getParent();
        }
        return;
    }

    /**
     * Updates the position of the given node.
     *
     * @param currNode a node whose key has been updated.
     */
    public void UpdateNodePosition(EventNode currNode)
    {
        EventNode siblingNode;
        if(this.memberCounter==1)
            return;
        if(currNode.getIndex()!=0  &&  currNode.getTime()<currNode.getParent().getTime())
        {
            do
            {
                if(currNode.getParent().getRightChild()!=null)
                    currNode = (currNode.getParent().getLeftChild().getTime()<=currNode.
                        getParent().getRightChild().getTime())?
                                currNode.getParent().getLeftChild():currNode.getParent().
                                    getRightChild();

                 currNode.swapNode(currNode.getParent());
                 currNode = currNode.getParent();
            }
            while( currNode.getIndex()!=0 && currNode.getTime()<currNode.getParent().
                getTime());

            return;
        }
        else if( ( currNode.getLevel()<lastLevel ) &&
                 ( currNode.getLeftChild()!=null ) &&
                 ( currNode.getTime()>=currNode.getLeftChild().getTime() ||
                 ( currNode.getRightChild()!=null && currNode.getTime()>currNode.
                     getRightChild().getTime()) ) )
        {
            do
            {
                EventNode child = null;

                if( currNode.getLeftChild()!=null && currNode.getRightChild()!=null)
                    child = (currNode.getLeftChild().getTime()<=currNode.getRightChild().
                        getTime()) ? currNode.getLeftChild():currNode.getRightChild();
                else if(currNode.getLeftChild()!=null && currNode.getRightChild()==null)
```

```
                child = currNode.getLeftChild();
            else if(currNode.getLeftChild()==null && currNode.getRightChild()==null)
            {
                if(currNode.getParent()==null)
                    Objects3D.VectorOperation.errMassageExit("Error! inside EventHeap/
                        UpdateNodePosition()",
                            "EventHeap/UpdateNodePosition");
                else if(currNode.getTime()>=currNode.getParent().getTime())
                {
                    /* there is no more child, so currNode is in the lastLevel of the
                        EventHeap, SUCCESSFUL UpdateNodePosition*/
                    secondNodeL = heapRoot.getLeftChild();
                    secondNodeR = heapRoot.getRightChild();
                    return;
                }
            }
            else
            {
                Objects3D.VectorOperation.errMassageExit("Error! inside EventHeap/
                    UpdateNodePosition()",
                            "EventHeap/UpdateNodePosition");
            }
            if(child==null)
                Objects3D.VectorOperation.errMassageExit("
                    oooooooooooooooooooooooooooooo"+child,
                            "EventHeap/UpdateNodePosition");

            if(currNode.getTime()>child.getTime())
            {
                currNode.swapNode(child);
                currNode = child;
            }
            else
            {
                /* currNode is in the middle of the EventHeap, SUCCESSFUL
                    UpdateNodePosition*/
                secondNodeL = heapRoot.getLeftChild();
                secondNodeR = heapRoot.getRightChild();
                return;
            }

        }
        while( currNode != null && currNode.getLevel()<lastLevel );

        secondNodeL = heapRoot.getLeftChild();
        secondNodeR = heapRoot.getRightChild();
        return;
    }
    secondNodeL = heapRoot.getLeftChild();
    secondNodeR = heapRoot.getRightChild();
    return;
}

/**
 * Reconnect the disconnected root to the heap.
 */
public void handleNullRoot(){
    Event e = new Event(secondNodeL.getEvent());
    heapRoot = new EventNode(e);
    heapRoot.setRightChild(secondNodeR);
    heapRoot.setLeftChild(secondNodeL);
    heapRoot.setParent(null);
    Delete(secondNodeL.getEvent().getOwnerParticle());
}
```

```
/**
 * Deletes a node of the given particle from the heap.
 *
 * @param P a particle which has left the boundary of the system.
 */
public void Delete(Particle P)
{
    EventNode currNode=null, deletedNode=lastNode;
    Particle remP;
    EventNode node = P.getEventNode();

    // The following part is for UPADETING heap variable after deleting node
    /**
     * 1. When the lastNode is root
     */
    if(node!=null && lastNode.getIndex()==node.getIndex())
    {
        //heapRoot = null;
        lastLevel = -1;
        memberCounter = 0;
        possibleNodes = 0;

        if(lastNode.getEvent()!=null)
        {
            remP = lastNode.getEvent().getOwnerParticle();
            remP.setEventNode(null);
            remP.setEvent(null);
        }
        remP = null;
        lastNode = null;
        bottomLeftNode = null;
        bottomRightNode = null;

        return ;
    }

    node.swapNode(lastNode);

    // setting the EventNode of the Particle of the deletedNode to the Null
    if(lastNode.getEvent()!=null)
        remP = lastNode.getEvent().getOwnerParticle();
    remP = null;

    /*
     * 2. When the lastNode is bottomLeftNode
     */
    if(lastNode.getIndex()==bottomLeftNode.getIndex())
    {
        lastLevel--;
        memberCounter--;
        possibleNodes = possibleNodes - (int)Math.pow(2,lastLevel+1);;
        lastNode = bottomRightNode;
        bottomLeftNode = bottomLeftNode.getParent();
    }
    /*
     * 3. When the lastNode is bottomRightNode
     */
    else if(lastNode.getIndex()==bottomRightNode.getIndex())
    {
        memberCounter--;
        lastNode = bottomRightNode.getParent().getLeftChild();
        bottomRightNode = bottomRightNode.getParent();
    }
```

```
    /*
     * 4. When the lastNode is the right child of a middle node
     */
    else if(lastNode.getState()==State.RIGHT)
    {
        memberCounter--;
        lastNode = lastNode.getParent().getLeftChild();
    }
    /*
     * 5. When the lastNode is the left child of a middle node
     */
    else if(lastNode.getState()==State.LEFT)
    {
        memberCounter--;

        /* probing the lastNode */
        currNode = lastNode;
        while(currNode.getState()==State.LEFT)
            currNode = currNode.getParent();

        currNode = currNode.getParent().getLeftChild();

         while(currNode.getRightChild()!=null)
             currNode = currNode.getRightChild();

        lastNode = currNode;
    }
    else
        Objects3D.VectorOperation.errMassageExit("---->(none of the above)"+
                "ERROR, the node can not be deleted!!!(inside EventHeap/Delete)",
                "EventHeap/Delete");

    /** following part is for removing the connection of the parent of the deletedNode
        */
    if(deletedNode.getState()==State.LEFT)
        deletedNode.getParent().setLeftChild(null);
    else if(deletedNode.getState()==State.RIGHT)
        deletedNode.getParent().setRightChild(null);
    else
        Objects3D.VectorOperation.errMassageExit("ERROR, the node to be deleted does'nt
            have Dir!!!(inside EventHeap/Delete)",
                "EventHeap/Delete");
    this.UpdateNodePosition(node);
    return ;
}

/**
 * Returns a string representation of the event queue.
 *
 * @return a string representation of this event queue.
 */
public String toString()
{
    return("\n\n-------------------------------"+
            "\n   Event Heap Information  "+
            "\n-------------------------------"+
            "\n1. lastLevel: " + lastLevel+
            "\n2. memberCounter: " + memberCounter+
            "\n3. possibleNodes: " + possibleNodes+
            "\n4. heapRoot Value: " + heapRoot.getTime()+
            "\n5. Root leftChild Value: " + heapRoot.getLeftChild().getTime()+
            "\n   Root left left Value: " + heapRoot.getLeftChild().getLeftChild().
                getTime()+
            "\n   Root left right Value: " + heapRoot.getLeftChild().getRightChild().
```

```
                    getTime()+
                "\n6. Root right Child Value: " + heapRoot.getRightChild().getTime()+
                "\n    Root right left Value: " + heapRoot.getRightChild().getLeftChild().
                    getTime()+
                "\n    Root right right Value: " + heapRoot.getRightChild().getRightChild().
                    getTime());
    }

    public EventNode getHeapRoot(){
        return heapRoot;
    }

    public int getMemberCounter(){
        return memberCounter;
    }
}
```

**EventNode.java:** The EventNode implements the elements of event queue. The nodes of the event queue, the particle nodes and the launch node, are instantiated from this class.

```
package SystemEvents;

import mySimulation.Particle;

import mySimulation.SimulationManager;

public class EventNode {
    /**
     * The states of nodes.
     * LEFT: left child of their parent.
     * RIGHT: right child of their parent.
     * ROOT: root of the heap.
     * UNDEFINED: undefied status.
     */
    public enum State { LEFT, RIGHT, ROOT, UNDEFINED }

    /**
     * The time of the earlies event of the associated particle.
     */
    private double time;

    /**
     * The type of the earlies event of the associated particle.
     */
    private Event.EventType nextEventType;

    /**
     * The index of the node in the event heap.
     */
    private int index;

    /**
     * The level of the node in the event heap.
     */
    private int level;

    /**
     * The state of the node.
     */
    private State state;

    /**
     * The event of the particle associated with this node.
     */
    private Event pEvent;

    /**
     * The parent of the node.
     */
    private EventNode parent;

    /**
     * The left child of the node.
     */
    private EventNode leftChild;
```

```
/**
 * The right child of the node.
 */
private EventNode rightChild;

/**
 * Creates a new instance of EventNode.
 *
 * @param e the event of the particle associated with this node.
 */
public EventNode(Event e){
    pEvent = e;

    if(pEvent==null)
        this.setNextEventNodeInfo(SimulationManager.SystemTime+(SimulationManager.
            AJM_Nozzle.getLunchingTimeInterval()),
                                    Event.EventType.launchType);
    else
        resetNextEventNodeInfo();

    index = -1;
    level = -1;
    state = State.UNDEFINED;
    parent = null;
    leftChild = null;
    rightChild = null;
}

/**
 * Updates the time for launching the next particle. This is only used for the launch
    node.
 */
public void setNextEventNodeInfo_LunchingParticle(){
    if(nextEventType!=Event.EventType.launchType)
        Objects3D.VectorOperation.errMassageExit("The function '
            setNextEventNodeInfo_LunchingParticle()' must be called for
            LunchingParticleEvent ONLY",
                                                "EventHeapElement/
                                                    setNextEventNodeInfo_LunchingParticle
                                                    ()");

    this.time += SimulationManager.AJM_Nozzle.getLunchingTimeInterval();
    SimulationManager.AJM_EventHeap.UpdateNodePosition(this);

    if(((SimulationManager.AJM_EventHeap.getHeapRoot().getLeftChild())!=null &&
       (SimulationManager.AJM_EventHeap.getHeapRoot().getLeftChild().getTime()<
            SimulationManager.AJM_EventHeap.getHeapRoot().getTime())) ||
       ((SimulationManager.AJM_EventHeap.getHeapRoot().getRightChild())!=null &&
       (SimulationManager.AJM_EventHeap.getHeapRoot().getRightChild().getTime()<
            SimulationManager.AJM_EventHeap.getHeapRoot().getTime())))
          Objects3D.VectorOperation.errMassageExit("wrong order in heap","
              EventHeapElement");
}

/**
 * Resets the time and tyoe of the event of this node to default values.
 */
public void resetNextEventNodeInfo(){
        time = Double.POSITIVE_INFINITY;
        nextEventType = Event.EventType.undefinedType;
}

/**
```

```
     * Swaps the position of this node with the given node.
     *
     * @param secondElement a node.
     */
    public void swapNode(EventNode secondElement){
        double tempTime = secondElement.getTime();
        Event.EventType tempType = secondElement.getNextEventType();
        Event tempEvent = secondElement.getEvent();

        secondElement.setEvent(this.pEvent);
        secondElement.setNextEventNodeInfo(this.time,this.nextEventType);

        this.setEvent(tempEvent);
        this.setNextEventNodeInfo(tempTime, tempType);

        if(this.getEvent()!=null)
            this.getEvent().getOwnerParticle().setEventNode(this);

        if(secondElement.getEvent()!=null)
            secondElement.getEvent().getOwnerParticle().setEventNode(secondElement);
    }

    /**
     * Updates the position of the node inside the event heap.
     *
     * @param P the particle associated with this node.
     */
    public void changeElement_Place_Info(Particle P){
        if(P!=pEvent.getOwnerParticle())
            Objects3D.VectorOperation.errMassageExit("P!=pEvent.getOwnerParticle()","
                EventHeapElement/changeElement_Place_Info()");
        this.pEvent.pickNextEventInfo();
        SimulationManager.AJM_EventHeap.UpdateNodePosition(this);
    }

    /**
     * Sets the time and the nextEventType variables of this node to new values.
     */
    public void setNextEventNodeInfo(double time, Event.EventType type){
            this.time = time;
            nextEventType = type;
    }

    public double getTime(){
        return time;
    }

    public Event.EventType getNextEventType(){
        return nextEventType;
    }

    public EventNode getLeftChild(){
        return leftChild;
    }

    public void setLeftChild(EventNode node) {
        if(node!=null)
        {
            node.state = State.LEFT;
            node.parent = this;
            leftChild = node;
        }
        else
            this.leftChild = null;
```

```
    }

    public EventNode getRightChild(){
        return rightChild;
    }

    public void setRightChild(EventNode node){
        if(node!=null)
        {
            node.state = State.RIGHT;
            node.parent = this;
            rightChild = node;
        }
        else
            this.rightChild = null;
    }

    public EventNode getParent(){
        return parent;
    }

    public void setParent(EventNode node){
        parent = node;
    }

    public Event getEvent(){
        return pEvent;
    }

    public void setEvent(Event e){
        pEvent = e;
    }

    public int getIndex(){
         return index;
     }

    public void setIndex( int i ){
         index = i;
    }

    public int getLevel(){
         return level;
     }

    public void setLevel( int l ){
         level = l;
     }

    public State getState(){
         return state;
     }

    public void setState( State s ){
         state = s;
     }
}
```

**EventPPCollision.java:** The EventPPCollision class is used to predict and handle collisions between particles.

```
package SystemEvents;

import Objects3D.Vector3D;

import Objects3D.VectorOperation;

import java.lang.Math; import java.util.*;

import SystemEvents.Event.EventType;

import mySimulation.*;

import SystemEnvironment.*;

import SystemEvents.Event.enteredCubeType;

import Objects3D.Vector3D;

public class EventPPCollision {
    /**
     * Auxiliary variables used by the P_P_CollisionTime method. They defines as static
         variables to reuse their memory spaces.
     */
    static Vector3D X1=new Vector3D(), X2=new Vector3D();

    /**
     * Auxiliary variables used to store a pair of the last overlapped particles.
     */
    static int overLappedP1=0, overLappedP2=0;

    /**
     * Detects a particle-particle collision between the two given particles.
     *
     * @param P1 the first particle.
     * @param P2 the second particle.
     *
     * @return the time of the collision between two particles. If the particles are not
         colliding, infinity will be returned.
     */
    public static double P_P_CollisionTime(Particle P1, Particle P2)
    {
        if(theyJustCollided(P1,P2) || P1.getID()==P2.getID())
            return(Double.POSITIVE_INFINITY);

        int particleId = SimulationManager.collParticleId, partnerId=SimulationManager.
            collPartnerId;
        double R1=P1.getRadius(), R2=P2.getRadius();
        double distance, R;
        double diffP1Time = SimulationManager.SystemTime-P1.getParticleTime();
        double diffP2Time = SimulationManager.SystemTime-P2.getParticleTime();
        boolean backwardTime = false;
        double minTime = -1;

        X1.setPoint(P1.getPosition());
        X2.setPoint(P2.getPosition());
        Vector3D V1 = P1.getLinearVelocity();
        Vector3D V2 = P2.getLinearVelocity();
```

```
distance = VectorOperation.subtract(X1,X2).getNorm();
X1.addWith(VectorOperation.multiply(V1,diffP1Time));
X2.addWith(VectorOperation.multiply(V2,diffP2Time));
Vector3D deltaX = VectorOperation.subtract(X1,X2);

distance = deltaX.getNorm();
R = R1+R2;

if(distance==R)
{
    if(P1.getParticleTime()!=P2.getParticleTime())
        VectorOperation.errMassageExit("These_two_particles_have_already_collided_at
            _the_time_0_and_P1.getPar" +
                "ticleTime()!=P2.getParticleTime()!","EventHandler_PPCollision/
                    P_P_CollisionTime()");
    return Double.POSITIVE_INFINITY;
}
else if(distance<R)
{
    Vector3D V1test = VectorOperation.multiply(V1,-1.0);
    Vector3D V2test = VectorOperation.multiply(V2,-1.0);
    double timeRemOverlap = getRoot(deltaX, VectorOperation.subtract(V1test,V2test),
        R);
    X1 = VectorOperation.add(VectorOperation.multiply(V1test,timeRemOverlap),X1);
    X2 = VectorOperation.add(VectorOperation.multiply(V2test,timeRemOverlap),X2);
    Vector3D deltaXtest = VectorOperation.subtract(X1,X2);
    distance = deltaXtest.getNorm();
    if(timeRemOverlap!=Double.POSITIVE_INFINITY &&
            timeRemOverlap>0 && isInBoundary(X1,P1.getRadius()) && isInBoundary(X2,
                P2.getRadius()))
    {
        double t = SimulationManager.SystemTime-timeRemOverlap;
        if(t<P1.getParticleTime())
        {
            if(P1.getLastEvent()==Event.EventType.launchType)
            {
                if(!hadOverlap(P1.getID(),P2.getID()))
                {
                    if(P1.getLastEvent()==Event.EventType.launchType ||
                            P2.getLastEvent()==Event.EventType.launchType)
                        mySimulation.Counters.overAtLaunch++;
                    mySimulation.Counters.overlapCounting++;
                    overLappedP1 = P1.getID();
                    overLappedP2 = P2.getID();
                }
                return Double.POSITIVE_INFINITY;
            }
        }
        if(t<P2.getParticleTime())
        {
            if(P1.getLastEvent()==Event.EventType.launchType)
            {
                if(!hadOverlap(P1.getID(),P2.getID()))
                {
                    if(P1.getLastEvent()==Event.EventType.launchType ||
                            P2.getLastEvent()==Event.EventType.launchType)
                        mySimulation.Counters.overAtLaunch++;
                    mySimulation.Counters.overlapCounting++;
                    overLappedP1 = P1.getID();
                    overLappedP2 = P2.getID();
                }
                return Double.POSITIVE_INFINITY;
            }
        }
```

```
            return t;
        }
        else
        {
            if(!hadOverlap(P1.getID(),P2.getID()))
            {
                if(P1.getLastEvent()==Event.EventType.launchType ||
                        P1.getLastEvent()==Event.EventType.launchType)
                    mySimulation.Counters.overAtLaunch++;
                if(timeRemOverlap==Double.POSITIVE_INFINITY)
                    mySimulation.Counters.infinit++;
                if(timeRemOverlap<0)
                    mySimulation.Counters.remOverLapTime++;
                if(distance<R)
                    mySimulation.Counters.disLessR++;
                if(!isInBoundary(X1,P1.getRadius()) || !isInBoundary(X2,P2.getRadius()))
                    mySimulation.Counters.isNotInBound++;

                mySimulation.Counters.overlapCounting++;
                overLappedP1 = P1.getID();
                overLappedP2 = P2.getID();
            }
            return Double.POSITIVE_INFINITY;
        }
    }

    Vector3D deltaV = VectorOperation.subtract(V1,V2);
    /*
     * If particles are not overlapping at time 0, and the equation has two solution,
         then
     * the smaller solution is the time of their next collision.
     * Otherwise, if the equation has no solution, the particles will not collide if
         they
     * move along the same straight line indefinitely.
     */
    double result = getRoot(deltaX, deltaV, R);

    if(result==Double.POSITIVE_INFINITY)
        return result;

    if(result == Double.NaN || result == Float.NaN || result==0 || result<0 )
        VectorOperation.errMassageExit("result time = " + result, "
            EventHandler_PPCollision/P_P_CollisionTime()");

    if(result!=Double.POSITIVE_INFINITY)
    {
        if(backwardTime)
        {
            SimulationManager.SystemTime = minTime + result;
            return SimulationManager.SystemTime;
        }
        result = SimulationManager.SystemTime + result;
    }

    return result;
}

/**
 * Checks if the two particles had overlapped.
 *
 * @param i1 the id of the first particle.
 * @param i2 the id of the second particle.
 *
 * @return true if the particle with id, i1, had overlapped with the particle with id,
```

```
     i2; false otherwise.
 */
private static boolean hadOverlap(int i1, int i2){
    return((i1==overLappedP1 && i2==overLappedP2) || (i1==overLappedP2 && i2==
        overLappedP1));
}

/**
 * Checkes if the particle is about to leave the system.
 *
 * @param p the center of the particle.
 * @param radius the radius of the particle.
 *
 * @return true if particle is about to leave the system; flase otherwise.
 */
private static boolean isInBoundary(Vector3D p, double radius){
    return(p.getX()-radius>0 && p.getX()+radius<SimulationManager.AJM_Environment.
        getWidth()
    && p.getY()-radius>0 && p.getY()+radius<SimulationManager.AJM_Environment.getWidth
        ()
    && p.getZ()-radius>0 && p.getZ()+radius<
            (SimulationManager.AJM_Nozzle.getStand_off_dis()+SimulationManager.
                AJM_Substrate.getSubstrateDepth()));
}

/**
 * Solves a quadratic equation to find the collision time between two particles.
 * <p>
 * Solving the equation requires that particles are not overlapping at time 0.
 * If the solution of the equation, has only one positive solution, the solution is the
      time of the next collision,
 * and if it has two positive solution the smaller one is the time of the next
     collision.
 * If it does not have any positive solution, these two particles will not collide.
 *
 * @param deltaX subtraction of the positions of the two particles.
 * @param deltaV subtraction of the veloceties vectors of the two particles.
 * @param R sum of the radii of the two particles.
 * @return the time that two particles collide with each other if the particles are
     colliding; infinity otherwise.
 */
public static double getRoot(Vector3D deltaX, Vector3D deltaV, double R){
    double t1, t2, result;
    double a, b, c;

    a = deltaV.getSquareNorm();
    b = 2.0*VectorOperation.dotProduct(deltaV,deltaX);
    c = deltaX.getSquareNorm()-(R*R);

    /*
     * Now in order to gain the time "t" we have to calculate the roots of this equation
         (t1,t2):
     * a*(t^2)+b(t)+c=0;
     */
    if(((b*b)-(4.0*a*c))<0)
        return Double.POSITIVE_INFINITY;

    t1 = (-b + Math.pow( (b*b)-(4.0*a*c) , 0.5 )) / (2.0*a);
    t2 = (-b - Math.pow( (b*b)-(4.0*a*c) , 0.5 )) / (2.0*a);

    result = -1;
    if(t1>=0 && t2>=0)
        return Math.min(t1,t2);
```

```
    else if(t1<0 && t2>=0)
        return t2;

    else if(t1>=0 && t2<0)
        return t1;

    else if(t1<0 && t2<0)
        return Double.POSITIVE_INFINITY;

    else if(t1==Double.NaN  || t2==Double.NaN  || t1==Float.NaN  || t2==Float.NaN)
        return Double.POSITIVE_INFINITY;
    else
    {
        SimulationManager.printStatistics();
        System.exit(0);
    }
    return Double.POSITIVE_INFINITY;
}

/**
 * Checks if the two given particles were the last two particles that their collision
     was handled.
 *
 * @param P1 the first particle.
 * @param P2 the second particle.
 * @return true if the two particles were the last two particles that their collision
     was handled; false otherwise.
 */
public static boolean theyJustCollided(Particle P1,Particle P2){
    int particleId = SimulationManager.collParticleId, partnerId=SimulationManager.
        collPartnerId;

    if(P1.getID()==particleId)
        return(P2.getID()==partnerId);
    else
        return((P1.getID()==partnerId)&&(P2.getID()==particleId));
}

/**
 * Renew the memberships of the particle in environment cells and assigns the particle
     to the cells that contain it or the one
 * in which the particle is about to enter.
 *
 * @param P a particle that its particle-particle collision was just handled.
 */
public static void updatingCubesAfterPPColl(Particle P)
{
    if(P.getOldSpace()==null)
        return;
    if(P.getCurrentSpace()==null)
    {
        Objects3D.VectorOperation.errMassageExit("P.currCube==null!\n" + P + "\nold␣
            cube:␣" + P.getOldSpace() +
                "\ncurrCube:␣" + P.getCurrentSpace()
                ,"EventHandler_PPCollisoin/updatingCubesAfterPPColl");
        P.setCurrentSpace(P.getOldSpace());
        return;
    }

    SystemEnvironment.Cell oldCube = P.getOldSpace();
    double distance=0;

    //Back
    if(P.getOldSpace().index1==P.getCurrentSpace().index1+1)
```

```
            distance = oldCube.getCubePosition().getX()-P.getX();

        //Front
        if(P.getOldSpace().index1==P.getCurrentSpace().index1-1)
            distance = P.getX() - (oldCube.getCubePosition().getX()+oldCube.getSize().getX
                ());

        //Left
        if(P.getOldSpace().index2==P.getCurrentSpace().index2+1)
            distance = Math.abs(oldCube.getCubePosition().getY()-P.getY());


        //Right
        if(P.getOldSpace().index2==P.getCurrentSpace().index2-1)
            distance = P.getY() - (oldCube.getCubePosition().getY()+oldCube.getSize().getY
                ());

        //Top
        if(P.getOldSpace().index3==P.getCurrentSpace().index3+1)
            distance = oldCube.getCubePosition().getZ()-P.getZ();

        //Bottom
        if(P.getOldSpace().index3==P.getCurrentSpace().index3-1)
            distance = P.getZ() - (oldCube.getCubePosition().getZ()+oldCube.getSize().getZ
                ());

        distance = Math.abs(distance);
        if(distance>P.getRadius())
        {
            P.getOldSpace().deleteMemberFrom(P);
            P.setOldSpace(null);
        }
        else
        {
            SystemEnvironment.Cell tempCube;
            tempCube = P.getOldSpace();
            P.setOldSpace(P.getCurrentSpace());
            P.setCurrentSpace(tempCube);
        }
    }

    /**
     * Handles the collision between two given particles by updting their positions and
       their velocities.
     * To update their position moves the particle to time at which collision occurrs.
     * To update their velocities, a coefficient of restitution approach is used.
     *
     *
     * @param P1 first particle.
     * @param P2 second particle.
     * @param time the time of at which the two particles collide with each other.
     */
    public static void handleEvent(Particle P1, Particle P2, double time)
    {
        double t1 = time - P1.getParticleTime();
        double t2 = time - P2.getParticleTime();

        P1.increaseNumPPColl();
        P2.increaseNumPPColl();

        P1.updatingParticlePosition(t1);
        P2.updatingParticlePosition(t2);

        P1.setParticleTime();
```

```
        P2.setParticleTime();

        updatingVelocities(P1, P2);
        updatingCubesAfterPPColl(P1);
        updatingCubesAfterPPColl(P2);
    }

    /**
     * Update the  velocities of the two given particles using a coefficient of restitution
         approach.
     *
     * @param P1 first particle.
     * @param P2 second particle.
     */
    public static void updatingVelocities(Particle P1, Particle P2)//(Vector3D V1, Vector3D
        V2, Vector3D X1, Vector3D X2, double M1, double M2)
    {
        VectorOperation opr = new VectorOperation();
        Vector3D V1 = P1.getLinearVelocity();
        Vector3D V2 = P2.getLinearVelocity();
        Vector3D X1 = P1.getPosition();
        Vector3D X2 = P2.getPosition();

        Vector3D n = opr.subtract(X2,X1).normalize();
        Vector3D t1 = opr.crossProduct(n,V1).normalize();
        Vector3D t2 = opr.crossProduct(n,t1).normalize();

        double V1n = opr.dotProduct(V1,n);
        double V1t1 = opr.dotProduct(V1,t1);
        double V1t2 = opr.dotProduct(V1,t2);

        double V2n = opr.dotProduct(V2,n);
        double V2t1 = opr.dotProduct(V2,t1);
        double V2t2 = opr.dotProduct(V2,t2);

        double epp = Particle.get_epp();
        double M1 = P1.getMass();
        double M2 = P2.getMass();

        double finV1n = (M2*( V2n + epp*(V2n-V1n)) + (M1*V1n))/(M1+M2);
        double finV2n = (M1*( V1n - epp*(V2n-V1n)) + (M2*V2n))/(M1+M2);

        Vector3D vn,vt1,vt2;
        vn = opr.multiply(n,finV1n);
        vt1 = opr.multiply(t1,V1t1);
        vt2 = opr.multiply(t2,V1t2);
        P1.setLinearVelocity(opr.add(vn,vt1,vt2));

        vn = opr.multiply(n,finV2n);
        vt1 = opr.multiply(t1,V2t1);
        vt2 = opr.multiply(t2,V2t2);
        P2.setLinearVelocity(opr.add(vn,vt1,vt2));
    }

    /**
     * Finds the smallest particle-particle collision time between the given particle and
         members of the given list.
     *
     * previous partner. thurfore, we have to find the new partner for it.
     * @param P a particle
     * @param Members a list which includes neighbors of the particle.
     *
     * @return true if the particle is colliding with the partner of some other particle
         earlier than they collide with each other;
```

```
 * false if the list is empty, the particle is found to collide with a particle that
     does not have any partner,
 * or the particle is not colliding with any of its neighbor.
 */
public static boolean predictPListCollision(Particle P, List Members)
{
    double tempTime = 0, minTime=Double.POSITIVE_INFINITY;
    Particle collidingParticle=null, tempParticle;
    int count, i;
    SystemEnvironment.Cell pCube = P.getCurrentSpace();
    count =  Members.size();

    P.getEventNode().resetNextEventNodeInfo();
    P.getEvent().getPPCollision().resetInfo();

    //1. The list is empty.
    if(count==0)
    {
        P.getEventNode().changeElement_Place_Info(P);
        if(P.getEventNode().getTime()!=P.getEvent().getTransfer().getMinTime() &&
                P.getEventNode().getTime()!=P.getEvent().getPSCollision().getMinTime())
            Objects3D.VectorOperation.errMassageExit("if(count==0)","
                EventHandler_PPCollision/P_List_CollisionTime()");
        return false;
    }


    for(i=0; i<count; i++)
    {
        tempParticle = (Particle)Members.get(i);
        tempTime = P_P_CollisionTime(P,tempParticle);

        /**
         * The third expression inside the "if" means that the currentPartner(
             tempParticle) should not
         * collide with any other particles before colliding with particle P.
         *
         * IMPORTANT:
         *      Before, the logical operation in the third expression was "<=" but it
             was changed to "<" to
         *      ignore the case that the three particles collide with each other at the
             same time which is
         *      not cosidered in the model.
         */
        if(tempTime!= Double.POSITIVE_INFINITY &&
           tempTime<=minTime &&
           tempTime<tempParticle.getEvent().getPPCollision().getMinTime())
        {
            minTime = tempTime;
            collidingParticle = tempParticle;
        }
    }

    /**
     * When the condition inside the "if" stisfies, the particle P will not collide
         with any
     * particle from the List.
     *
     * 4. There was not any PPCollision between the particle and the list.
     */
    if(collidingParticle==null)
    {
        P.getEventNode().changeElement_Place_Info(P);
        return false;
```

```
    }

    if(minTime==Double.POSITIVE_INFINITY)
        Objects3D.VectorOperation.errMassageExit("minTime==Double.POSITIVE_INFINITY!!!"
            ,
                                                 "EventHandler_PPCollision/
                                                     P_List_CollisionTime()");

    if(collidingParticle.getEvent().getPPCollision().getMinTime()<=minTime)
        Objects3D.VectorOperation.errMassageExit("PPCollTime_of_collidingParticle_is_
            less_than_minTime!!!" + "We_are_not_supposed_to_pick_such_a_
            collidingParticle!",
                                                 "EventHandler_PPCollision/
                                                     P_List_CollisionTime()");



    /*
     * This "if" is for the case that the next PPCollision of the collidingParticle
         should be changed to
     * the collision with particle P. Because if the condition applies, it means that
         Collision between
     * collidingParticle and P will happen earlier than the curent next PPCollision of
         the collidingParticle.
     */

    else
    {
        P.getEvent().getPPCollision().setInfo(collidingParticle, minTime);
        P.getEventNode().changeElement_Place_Info(P);

        /*
         * 2. The smallest PPCollision time is found and the colliding particle has
             another partner.
         */
        Particle prePartner = collidingParticle.getEvent().getPPCollision().
            getInvolvingParticle();
        if(prePartner!=null && P!=prePartner)
            return true;
        else
        {
            /*
             * 3. The smallest PPCollision time is found and the colliding particle
                 does not have any other partner.
             */
            collidingParticle.getEvent().getPPCollision().setInfo(P, minTime);
            collidingParticle.getEventNode().changeElement_Place_Info(collidingParticle
                );
            return false;
        }
    }
    Objects3D.VectorOperation.errMassageExit("unusual_situation!!!" ,"
        EventHandler_PPCollision/P_List_CollisionTime()");
    return false;
}

/**
 * Adds a new list to the neighbor of the particle.
 *
 * @param neighbors the list of neighbor.
 * @param l a new list
 * @param c a cell associated with the list l
 * @param P a particle
 */
```

```
private static void addToNeighbors(List neighbors, List l, Cell c, Particle P){
    if(l==null)
        return;
    neighbors.addAll(l);
    c.addMemberTo(P);
    P.getContainingCube().add(c);
}

/**
 * Removed the membership of a particle from its previous cells.
 *
 * @param P a particle.
 */
public static void resetContainngCubes(Particle P){
    List cells = P.getContainingCube();
    P.setContainingCube(new LinkedList());
    Cell curr = P.getCurrentSpace(), prev = P.getOldSpace();
    for(int i=0;i<cells.size();i++)
    {
        Cell c = (Cell)cells.get(i);
        c.getMembers().remove(P);
    }
}

/**
 * Creates a list of particle current neighbors.
 *
 * @param P a particle.
 * @return a list of particle current neighbor.
 */
public static List createList(Particle P) {
    Cell origin=P.getOldSpace(), dest=P.getCurrentSpace(),
            temp1,temp2,temp3,temp4,temp5,temp6,temp7,temp8;
    List neighbors = new LinkedList(dest.getMembers());
    resetContainngCubes(P);
    if(origin!=null && origin!=dest)
        addToNeighbors(neighbors,origin.getMembers(),origin,P);
    double x=P.getX(), y=P.getY(), z=P.getZ(), r=P.getRadius();
    Vector3D v1=new Vector3D(), v2=new Vector3D(), v3=new Vector3D(), v4=new Vector3D()
        ,
            v5=new Vector3D(), v6=new Vector3D(), v7=new Vector3D(), v8=new Vector3D();

    v1.setPoint(x-r,y-r,z-r);
    v2.setPoint(x-r,y-r,z+r);
    v3.setPoint(x-r,y+r,z-r);
    v4.setPoint(x-r,y+r,z+r);
    v5.setPoint(x+r,y-r,z-r);
    v6.setPoint(x+r,y-r,z+r);
    v7.setPoint(x+r,y+r,z-r);
    v8.setPoint(x+r,y+r,z+r);

    temp1 = mySimulation.SimulationManager.AJM_Environment.findCube(v1);
    if(temp1!=null && temp1!=origin && temp1!=dest)
        addToNeighbors(neighbors,temp1.getMembers(),temp1,P);

    temp2 = mySimulation.SimulationManager.AJM_Environment.findCube(v2);
    if(temp2!=null && temp2!=origin && temp2!=dest &&
            temp2!=temp1)
        addToNeighbors(neighbors,temp2.getMembers(),temp2,P);

    temp3 = mySimulation.SimulationManager.AJM_Environment.findCube(v3);
    if(temp3!=null && temp3!=origin && temp3!=dest &&
            temp3!=temp2 && temp3!=temp1)
        addToNeighbors(neighbors,temp3.getMembers(),temp3,P);
```

```
        temp4 = mySimulation.SimulationManager.AJM_Environment.findCube(v4);
        if(temp4!=null && temp4!=origin && temp4!=dest &&
                temp4!=temp3 && temp4!=temp2 && temp4!=temp1)
            addToNeighbors(neighbors,temp4.getMembers(),temp4,P);

        temp5 = mySimulation.SimulationManager.AJM_Environment.findCube(v5);
        if(temp5!=null && temp5!=origin && temp5!=dest &&
                temp5!=temp4 && temp5!=temp3 && temp5!=temp2 && temp5!=temp1)
            addToNeighbors(neighbors,temp5.getMembers(),temp5,P);

        temp6 = mySimulation.SimulationManager.AJM_Environment.findCube(v6);
        if(temp6!=null && temp6!=origin && temp6!=dest &&
                temp6!=temp5 && temp6!=temp4 && temp6!=temp3 && temp6!=temp2 &&
                temp6!=temp1)
            addToNeighbors(neighbors,temp6.getMembers(),temp6,P);

        temp7 = mySimulation.SimulationManager.AJM_Environment.findCube(v7);
        if(temp7!=null && temp7!=origin && temp7!=dest &&
                temp7!=temp6 && temp7!=temp5 && temp7!=temp4 && temp7!=temp3 &&
                temp7!=temp2 && temp7!=temp1)
            addToNeighbors(neighbors,temp7.getMembers(),temp7,P);

        temp8 = mySimulation.SimulationManager.AJM_Environment.findCube(v8);
        if(temp8!=null && temp8!=origin && temp8!=dest &&
                temp8!=temp7 && temp8!=temp6 && temp8!=temp5 && temp8!=temp4 &&
                temp8!=temp3 && temp8!=temp2 && temp8!=temp1)
            addToNeighbors(neighbors,temp8.getMembers(),temp8,P);

        return neighbors;
    }

    /**
     * Checks if a list of neighbors must be created.
     *
     * @param P a particle.
     * @param c a cell containing the particle.
     *
     * @return true if list must be created; false otherwise.
     */
    public static boolean listMustBeCreated(Particle P, Cell c)
    {
        double pr = SimulationManager.precision, r=P.getRadius();
        Vector3D p = c.getCubePosition();
        Vector3D s = c.getSize();

        return((P.getX()+r<=(p.getX()+s.getX()+pr) && P.getX()-r>=(p.getX()-pr) &&
                P.getY()+r<=(p.getY()+s.getY()+pr) && P.getY()-r>=(p.getY()-pr) &&
                P.getZ()+r<=(p.getZ()+s.getZ()+pr) && P.getZ()-r>=(p.getZ()-pr)));
    }

    /**
     * Predicts the next particle-particle collision for the given particle.
     *
     * @param P a particle.
     */
    public static void predictEvent(Particle P)
    {
        mySimulation.Counters.predictPPColl++;
        List members = new LinkedList(P.getCurrentSpace().getMembers());
        if(P.getOldSpace()!=null)
            members.addAll(P.getOldSpace().getMembers());
        List removedMembers = new LinkedList();
        int i=0;
```

```
Particle currP = P;
Particle prePartner;
Particle prevP=null;
boolean result=false;

if(P!=null && listMustBeCreated(P,P.getCurrentSpace()))
    members = createList(P);

while(members.size()>=1)
{
    i++;
    members.remove(currP);

    // find the nearest collision between currP and members list
    result = predictPListCollision(currP, members);

    // continute - find the collision between the old partner of the new partner of
        the currP and rest of the list
    if(result)
    {
        removedMembers.add(currP);
        prePartner = currP.getEvent().getPPCollision().getInvolvingParticle();
        if(!members.contains(prePartner))
            Objects3D.VectorOperation.errMassageExit("!members.contains(prePartner)
                ","EventHandler_PPCollision/predictEvent");

        members.remove(prePartner);
        removedMembers.add(prePartner);

        prevP = currP;

        /*
         * Finds a new partner for the prePartner.
         */
        currP = prePartner.getEvent().getPPCollision().getInvolvingParticle();

        if(currP==null)
            mySimulation.Counters.predictPPColl--;

        if(currP==null)
            return;

        if(currP==prevP)
             Objects3D.VectorOperation.errMassageExit("currP == prevP" + "\
                 nEventHandler_PPCollision/setCubePPColl_After_Transfer_Lunching()"
                 ,
                     "EventHandler_PPCollision/predictEvent");

        prePartner.getEvent().getPPCollision().setInfo(prevP, prevP.getEvent().
            getPPCollision().getMinTime());
        prePartner.getEventNode().changeElement_Place_Info(prePartner);

        try{
            currP.getEventNode().resetNextEventNodeInfo();
            currP.getEvent().getPPCollision().resetInfo();
        }catch(NullPointerException e)
        {
            EventTransfer.removeParticle(currP);
            Objects3D.VectorOperation.errMassageExit("currP: " + currP,"
                EventPPCollision/predictEvent");
            if(currP!=null)
            {
            }
            //mySimulation.Counters.removeParticleInPPPredictEvent++;
```

```
                mySimulation.Counters.predictPPColl--;
                return;
            }


            if(currP.getCurrentSpace()!=prevP.getCurrentSpace())
            {
                members = new LinkedList(currP.getCurrentSpace().getMembers());
                if(currP.getOldSpace()!=null)
                    members.addAll(currP.getOldSpace().getMembers());
                members.removeAll(removedMembers);
            }
        }
        else
            break;
    }
    mySimulation.Counters.predictPPColl--;
}

/**
 * Predicts the next particle-particle collisions for the two given particle after
     their collision.
 *
 * @param P1 first particle.
 * @param P2 second particle.
 */
public static void predictEvent(Particle P1, Particle P2)
{
    mySimulation.Counters.predictPPColl++;
    List members;
    List removedMembers = new LinkedList();
    int i=0;
    Particle temtPartner;
    Particle currP = P1;
    Particle prePartner;
    Particle prevP;
    boolean result=false;

    if(P1!=null)
    {
        members = new LinkedList(P1.getCurrentSpace().getMembers());
        if(P1.getOldSpace()!=null)
            members.addAll(P1.getOldSpace().getMembers());
        members.remove(P2);
        removedMembers.add(P2);
    }
    else
        members = new LinkedList();

    if(P1!=null && !listMustBeCreated(P1,P1.getCurrentSpace()))
        members = createList(P1);
    System.out.println(members.size() + "____" + P1.getCurrentSpace().getMembers().size
        ());

    while(members.size()>=1)
    {
        i++;
        members.remove(currP);
        result = predictPListCollision(currP, members);

        if( result )
        {
            removedMembers.add(currP);
            prePartner = currP.getEvent().getPPCollision().getInvolvingParticle();
```

```
                members.remove(prePartner);
                removedMembers.add(prePartner);
                prevP = currP;

                currP = prePartner.getEvent().getPPCollision().getInvolvingParticle();

                if(currP==null)
                    mySimulation.Counters.predictPPColl--;

                if(currP==null)
                    return;

                prePartner.getEvent().getPPCollision().setInfo(prevP, prevP.getEvent().
                    getPPCollision().getMinTime());
                prePartner.getEventNode().changeElement_Place_Info(prePartner);


                try{
                    currP.getEventNode().resetNextEventNodeInfo();
                    currP.getEvent().getPPCollision().resetInfo();
                }catch(NullPointerException e)
                {
                    EventTransfer.removeParticle(currP);
                    Objects3D.VectorOperation.errMassageExit("currP: " + currP,"
                        EventPPCollision/predictEvent");
                    mySimulation.Counters.predictPPColl--;
                    return;
                }
                if(currP.getCurrentSpace()!=prevP.getCurrentSpace())
                {
                    members = new LinkedList(currP.getCurrentSpace().getMembers());
                    if(currP.getOldSpace()!=null)
                        members.addAll(currP.getOldSpace().getMembers());
                    members.removeAll(removedMembers);
                }
            }
            else break;
        }
        mySimulation.Counters.predictPPColl--;
        if(P2!=null)
            predictEvent(P2);
    }
}
```

**EventPSCollision.java:** The EventPSCollision class is used to predict and handle particle-surface collisions.

```
package SystemEvents;

import Objects3D.Vector3D;

import OcTreeADT.*;

import com.sun.org.apache.bcel.internal.generic.Type;

import com.sun.org.apache.bcel.internal.verifier.statics.DOUBLE_Upper;

import com.threed.jpct.OcTree;

import javax.xml.transform.Result;

import mySimulation.*;

import mySimulation.SimulationManager;

import SystemEnvironment.*;

import java.util.*;

import Objects3D.VectorOperation;

import Visualization.Plot;

import SystemEvents.EventSphereCubeCollision.CollisionPacket;

import SystemEvents.EventSphereCubeCollision.Element;

import SystemEvents.EventSphereCubeCollision;

import OcTreeADT.CellularOcTree.Dir;

import Jama.*;

import Jama.util.*;

public class EventPSCollision {
    /**
     * The depth of the erosion profile.
     */
    public static double maxDepth = Double.NEGATIVE_INFINITY;

    /**
     * The deepest cell.
     */
    private static OcTreeADT.OctNode deepestCell = null;

    /**
     * The number of points at the profile cross-section to find the best curve fit.
     */
    public static int num_points = (int)(SimulationManager.AJM_Environment.getWidth()/
        OcTreeADT.CellularOcTree.getCellSize()) + 1;

    /**
     * The array used to save the points at the profile cross-section to find the best
```

```
     curve fit.
 */
public static double [][] point = new double [num_points][2];

/**
 * The variable used to save the wased energy of particles that is not trasnfer to the
     surface.
 */
public static double wastedEnergy = 0;

/**
 * Initializes the array, point, used to save the points at the profile cross-section.
 */
public static void initPoint(){
    for(int i=0; i<num_points; i++)
        point[i][0] = i*OcTreeADT.CellularOcTree.getCellSize();
}

/**
 * An auxiliary variable used in the getNormalVar method.
 */
public static double maxVar=0;

/**
 * The last surface cell impacted by a particle.
 */
private static OctNode particleTargetBox;

/**
 * Sets the particleTargetBox variable to the target cell impacted by the last particle
     hitting the surface.
 */
public static void setParticleTargetBox(OctNode cell){
    particleTargetBox = cell;
 }

/**
 * Handle particle-Surface collision for the given particle if target cell had not been
     removed before by some other particle.
 *
 * @param P the particle that is hitting the target surface.
 * @return true if a successful particle-surface collision occurrs; false if the
     collision was unsuccessful which happen when
 * the target cell had been removed earlier by the impact of some other partilce and
     was turned to the white cell..
 */
public static boolean handleEvent(Particle P)
{
    particleTargetBox = null;
    Vector3D normal = new Vector3D(P.getEvent().getPSCollision().getPlaneNormal());
    Vector3D contact = P.getEvent().getPSCollision().getContactPoint();

    if(mySimulation.SimulationManager.AJM_Substrate.removeCellFromOctree(P))
    {
        if(!normal.isEqualTo(P.getEvent().getPSCollision().getPlaneNormal()))
            Objects3D.VectorOperation.errMassageExit("normal has been changed","
                handlePScollision");

        double t = (0.2)*P.getRadius()/(P.getLinearVelocity().getNorm());
        settingStatistics(P);
        updateParticleVelocity(P, contact);
        if(P.getOldSpace()!=null)
        {
            double distance = Math.abs(SimulationManager.AJM_Environment.getDepth() - P
```

```
                .getPosition().getZ());
            if(distance>P.getRadius())
            {
                P.getOldSpace().deleteMemberFrom(P);
                P.setOldSpace(null);
            }
        }
        return true;
    }
    if(P.isUnusualSituation())
        EventTransfer.removeParticle(P);

    /**
     * When the target cell had been removed before, the unsuccessful partcle-surface
         collisoin occurrs and
     * method returns false.
     */
    return false;
}

/**
 * An auxiliary variables used by the setCellLoss method; it is defined as a global
     variable to allow reusing its space instead of
 * allocating a new space.
 */
private static Vector3D pV = new Vector3D();

/**
 * The variable used to store the value of (D/(h*Density*A)) which is used to calculate
     the loss for cells after each impact.
 */
private static double erosionConstant;

/**
 * Calculates the value of the erosionConstant (D/(h*Density*A)).
 */
public static void setErosionConstant(){
    erosionConstant = (SimulationManager.AJM_Substrate.getD()/
            (Math.pow(OcTreeADT.CellularOcTree.getCellSize()/1000.0,3.0)*
                SimulationManager.AJM_Substrate.getSurfaceDensity()));
}

/**
 * Calculates the loss variable of the cell which determines the removed volume of the
     cell.
 * <p>
 * Loss = (D/(h*Density*A))*Mp()*(V*cosAngle)^K
 * @param cell
 * @param P
 * @param normal
 */
public static void setCellLoss(OctNode cell, Particle P, Vector3D normal){
    // cos of the impact angle
    double cosTheta;
    double loss,preLoss=cell.getLoss();
    double h =  OcTreeADT.CellularOcTree.getCellSize()/1000.0;
    particleTargetBox = cell;
    cell.numOfColls++;
    double N = 1.0;
    pV.setPoint(P.getLinearVelocity());
    pV = VectorOperation.divide(pV,1000.0);
    double V = pV.getNorm();
    double dotProductNV = (VectorOperation.dotProduct(pV,normal)*-1.0);
    double A = h*h;
```

```
        cosTheta = Math.abs(dotProductNV/(N*V));

        if(SimulationManager.AJM_Substrate.isBrittle())
            loss = erosionConstant*P.getMass()*Math.pow(V*cosTheta,SimulationManager.
                AJM_Substrate.getK());
        else
            loss = erosionConstant*P.getMass()*Math.pow(V,SimulationManager.AJM_Substrate.
                getK())*
                    (Math.pow(cosTheta,SimulationManager.AJM_Substrate.getN1())*
                    Math.pow(1.0+SimulationManager.AJM_Substrate.getHv()*(1.0-cosTheta),
                        SimulationManager.AJM_Substrate.getN2()));

        if(loss!=Double.NaN)
            cell.setLoss(cell.getLoss()+loss);
    }

    /**
     * Remove the cell from the surface whose loss value is equal or greater than 1.
     */
    public static void removeCell(){
        particleTargetBox.setOcType(OctNode.Type.WHITE);
        OctNode Parent = particleTargetBox.parent;
        while(Parent!=null && Parent.getOcType()!=OctNode.Type.GRAY)
        {
            Parent.setOcType(OctNode.Type.GRAY);
            Parent = Parent.parent;
        }
        mySimulation.Counters.numOfRemovedCells++;
    }

    /**
     * Check if the given cell must be removed and if yes invokes the removeCell method to
       removed the cell from substrate.
     *
     * @param cell the last cell impacted by a particle.
     * @param P the last particle impacting the surface.
     *
     * @return true if the cell has been removed; false otherwise.
     */
    public static boolean checkCellToBeRemoved(OctNode cell, Particle P){
        double loss = cell.getLoss();
        Vector3D cellPosition = cell.getPosition();

        if(loss>=1.0)
        {
            wastedEnergy += (loss-1.0);
            try
            {
                SimulationManager.writeBoundaryCellInExcel2DY(cellPosition, 0.0);
            }catch(ArrayIndexOutOfBoundsException e){}

            Visualization.Plot.setRemovedCell(cell.getPosition());
            if(cell.getPosition().getZ()>maxDepth)
            {
                maxDepth = cell.getPosition().getZ();
                deepestCell = cell;
            }
            removeCell();
            return true;
        }
        else
        {
            cell.setOcType(OctNode.Type.CELL);
            if(loss<0)
```

```
            {
                System.out.println("Loss is negetive!!!!");
                System.exit(0);
            }
            double z = loss*OcTreeADT.CellularOcTree.getCellSize();
            try
            {
                SimulationManager.writeBoundaryCellInExcel2DY(cellPosition, z);
            }catch(ArrayIndexOutOfBoundsException e){}
        }
        return false;
    }


    /**
     * Returns the direction which is outward and perpendicular to the side face containing
        the impacted edge.
     *
     * @param cube the cube impacted by a particle.
     * @param v1 an end of the cube edge impacted by a particle.
     * @param v2 an end of the cube edge impacted by a particle.
     * @param normal the normal to the side face containing the impacted edge.
     *
     * @return the direction which is outward and perpendicular to the side face containing
        the impacted edge.
     */
    public static Dir getMarchingDirection(OctNode cube, Vector3D v1, Vector3D v2, Vector3D
         normal)
    {
        // B: back   F: front   L: left   R: right
        Vector3D center = cube.getPosition();
        if(normal.isEqualTo(EventSphereCubeCollision.topNorm))
        {
            // p1->p2 or p2->p3
            if(v1.getX()<center.getX())
            {
                if(v2.getX()<center.getX())
                    return Dir.B;
                else
                    return Dir.R;
            }
            // p3->p4 or p4->p1
            else
            {
                if(v2.getX()>center.getX())
                    return Dir.F;
                else
                    return Dir.L;
            }
        }
        else if(normal.isEqualTo(EventSphereCubeCollision.bottomNorm))
        {
            // p1->p2 or p2->p3
            if(v1.getX()<center.getX())
            {
                if(v2.getX()<center.getX())
                    return Dir.B;
                else
                    return Dir.R;
            }
            // p3->p4 or p4->p1
            else
            {
                if(v2.getX()>center.getX())
```

```
                return Dir.F;
            else
                return Dir.L;
        }
    }
    else if(normal.isEqualTo(EventSphereCubeCollision.backNorm))
    {
        return Dir.B;
    }
    else if(normal.isEqualTo(EventSphereCubeCollision.frontNorm))
    {
        return Dir.F;
    }
    else if(normal.isEqualTo(EventSphereCubeCollision.leftNorm))
    {
        return Dir.L;
    }
    else if(normal.isEqualTo(EventSphereCubeCollision.rightNorm))
    {
        return Dir.R;
    }
    return Dir.none;
}

/**
 * Returns a vector parallel to both the xy-plane and the face of the cube on the given
     side.
 *
 * @param cube a cube whose egde is impacted by a particle.
 * @param determines the direction at which the face of the cube is located.
 *
 * @return a vector parallel to both the xy-plane and the face of the cube on the given
     side.
 */
private static Vector3D getVectorParallelEdge(OctNode cube, Dir dir){
    Vector3D p1 = cube.getPosition();
    Vector3D p2=null;
    if(dir==Dir.R || dir==Dir.L)
        p2 = new Vector3D(p1.getX()+0.1,p1.getY(),p1.getZ());
    else if(dir==dir.B || dir==Dir.F)
        p2 = new Vector3D(p1.getX(),p1.getY()+0.1,p1.getZ());
    else
        VectorOperation.errMassageExit("wrong direction","GEometryHandler/getEdge()");

    return VectorOperation.subtract(p1,p2);
}

/**
 * Changes the given indices to reflect the cell in the given direction.
 *
 * @param index a two-dimensional array which stores the indices of an element in the
     removedCell array.
 * @param d a direction
 */
public static void setNeighborIndex(int [][]index, Dir d){
  if(d == Dir.L)
      index[0][1]--;
  else if(d == Dir.R)
      index[0][1]++;
  else if(d == Dir.B)
      index[0][0]--;
  else if(d == Dir.F)
      index[0][0]++;
}
```

```
/**
 * Sets the local coordinate system for the particle-surface impact.
 * For the cases particles hit\ edges or vertices of the cells, it estimates the
     bisecting plane and
 * calculates the normal and tangential vectors to the plane.
 *
 * @param P the particle impacting the surface.
 * @param n the normal vector to the impact surface.
 * @param t1 a tangential vector to the impact surface.
 * @param t2 a tangential vector to the impact surface.
 */
private static void setCoordinateSystem_HitEdge(Particle P, Vector3D n, Vector3D t1,
    Vector3D t2)
{
   double cellsize = OcTreeADT.CellularOcTree.getCellSize();
   int index[][] = new int[1][2];
   index[0][0] = (int)(particleTargetBox.getPosition().getX()/cellsize);
   index[0][1] = (int)(particleTargetBox.getPosition().getY()/cellsize);

   int X0=index[0][0], Y0=index[0][1];
   if(!(X0<Visualization.Plot.count && Y0<Visualization.Plot.count))
   {
       setCoordinateSystem_HitSquare(P, n, t1, t2);
       return;
   }

   double Z01 = Math.abs(Plot.removedCell[X0] [Y0]);
   double Z02=0.0;
   if(particleTargetBox.getPosition().getZ()>(SimulationManager.AJM_Environment.
       getDepth()+cellsize))
       Z02 = particleTargetBox.getPosition().getZ() + (cellsize/2.0) -
           SimulationManager.AJM_Environment.getDepth();

   double Z0 = Math.min(Z02,Z01);

   Vector3D v1 = P.getEvent().getPSCollision().getV1();
   Vector3D v2 = P.getEvent().getPSCollision().getV2();
   Vector3D planeNormal = P.getEvent().getPSCollision().getPlaneNormal();
   Dir dir = getMarchingDirection(particleTargetBox, v1, v2, planeNormal);
   Vector3D edge1=null;
   double x, y, z;
   double currZ;
   setNeighborIndex(index,dir);
   z=cellsize;
   double testCurrx = index[0][0];
   double testCurry = index[0][1];
   double testCurrz;

   try
   {
   testCurrz = Plot.removedCell[index[0][0]][index[0][1]];
   }catch(ArrayIndexOutOfBoundsException e){
       setCoordinateSystem_HitSquare(P,n,t1,t2);
       return;
   }

   currZ = Math.abs(Plot.removedCell[index[0][0]][index[0][1]]);

   // if this cell and neighbor are in the same level so FLAT surface is used.
   if(currZ==Z0)
   {
       setCoordinateSystem_HitSquare(P,n,t1,t2);
       return;
```

```
        }
        // if neighbor cell is deeper
        else if(currZ+(cellsize/2.0)>Z0)
        {
            double i=index[0][0], j=index[0][1];
            // if different between tergetCell and neighbor is 1 level
            if((currZ-Z0)<=cellsize)
            {
                double tempZ = currZ;
                while(tempZ==currZ)
                {
                    i = index[0][0];
                    j = index[0][1];
                    setNeighborIndex(index,dir);
                    try{
                    currZ = Math.abs(Plot.removedCell[index[0][0]][index[0][1]]);
                    }catch(ArrayIndexOutOfBoundsException e)
                    {
                        break;
                    }
                }
                // this is like a removed cell that between two unremoved cells
                if(Math.max(i-X0,j-Y0)==1 && currZ==Z0)
                {
                    setCoordinateSystem_HitSquare(P,n,t1,t2);
                    return;
                }
                else
                    edge1 = new Vector3D(i-X0,j-Y0,1.0);
            }
            // if different is more than 1 level
            else
            {
                int stepZ = (int)((currZ-Z0)/cellsize + 0.5);
                edge1 = new Vector3D(i-X0,j-Y0,stepZ);
            }

            Vector3D edge2 = getVectorParallelEdge(particleTargetBox,dir);
            n.setPoint(VectorOperation.crossProduct(edge1,edge2).normalize());
            t1.setPoint(VectorOperation.crossProduct(n,P.getLinearVelocity()).normalize());
            t2.setPoint(VectorOperation.crossProduct(n,t1).normalize());
        }
        else
            setCoordinateSystem_HitSquare(P,n,t1,t2);
    }

    /**
     * Sets the local coordinate system for particle-surface impacts on the square planes
         of cells.
     *
     * @param P the particle impacting the surface.
     * @param n the normal vector to the impact surface.
     * @param t1 a tangential vector to the impact surface.
     * @param t2 a tangential vector to the impact surface.
     */
    private static void setCoordinateSystem_HitSquare(Particle P, Vector3D n, Vector3D t1,
        Vector3D t2)
    {
        mySimulation.Counters.numPlaneNormal++;
        Vector3D planeNormal = P.getEvent().getPSCollision().getPlaneNormal();

        if(planeNormal.getX()!=0)
        {
            n.setPoint(EventSphereCubeCollision.frontNorm);
```

```
            t1.setPoint(EventSphereCubeCollision.rightNorm);
            t2.setPoint(EventSphereCubeCollision.bottomNorm);
        }
        else if(planeNormal.getY()!=0)
        {
            n.setPoint(EventSphereCubeCollision.rightNorm);
            t1.setPoint(EventSphereCubeCollision.bottomNorm);
            t2.setPoint(EventSphereCubeCollision.frontNorm);
        }
        else if(planeNormal.getZ()!=0)
        {
            n.setPoint(EventSphereCubeCollision.bottomNorm);
            t1.setPoint(EventSphereCubeCollision.frontNorm);
            t2.setPoint(EventSphereCubeCollision.rightNorm);
        }
        else
            VectorOperation.errMassageExit("undefined coordinate system" + "\nnormal: " +
                planeNormal,"EventHandler_PSCollision/setCoordinateSystem");
    }

    /**
     * The degree of the polynomial of the curve fitted to the cross-section of the profile
     .
     */
    public static int degreeOfPoly = 3;

    /**
     * The array representing the best fit polynomial.
     */
    public static double parameters[] = new double[(degreeOfPoly+1)];

    /**
     * Calculates the best curve fit to the cross-section of the erosion profile.
     */
    public static void getBestCurveFit()
    {
        int ignoredPoints = 6;
        int skip = 0;
        int numParam = degreeOfPoly+1;
        double [][] alpha  = new double[numParam][numParam];
        double [] beta = new double[numParam];
        double term = 0;
        /**
         * to avoid extra calculation the points[i][0]^k are saved in this array.
         */
        double [] pointPowK = new double[num_points];

        /**
         * Row of equarions matrix
         */
        for (int k=0; k < numParam; k++)
        {
            for (int i=skip; i < num_points-skip; i=i+ignoredPoints)
                /**
                 * Calculate x^k
                 */
                pointPowK[i]=Math.pow(point[i][0],(double)k);

            /**
             * Only need to calculate diagonal and upper half of symmetric matrix.
             */
            for (int j=k; j < numParam; j++)
            {
                /**
```

```
                * Calculates terms over the data points
                */
               term = 0.0;
               alpha[k][j] = 0.0;
               /**
                * Calculate sigma over points  Sigma{x^j * x^k} which is the value of
                    elements alpha[k][j] and alpha[j][k]
                */
               for (int i=skip; i < num_points-skip; i=i+ignoredPoints)
               {
                   /**
                    * Calculate x^k * x^j
                    */
                   term = pointPowK[i]* Math.pow(point[i][0],(double)j);
                   alpha[k][j] += term;
               }
               /**
                * matrix is symmetric
                */
               alpha[j][k] = alpha[k][j];
           }

           for (int i=skip; i < num_points-skip; i=i+ignoredPoints)
           {
               term =  (point[i][1] * pointPowK[i]);
               beta[k] +=term;
           }
       }

       /**
        * Use the Jama QR Decomposition classes to solve for the parameters.
        */
       Matrix alpha_matrix = new Matrix (alpha);
       QRDecomposition alpha_QRD = new QRDecomposition (alpha_matrix);
       Matrix beta_matrix = new Matrix (beta,numParam);
       Matrix param_matrix;
       try {
          param_matrix = alpha_QRD.solve (beta_matrix);
       }
       /**
        * QRD solve failed
        */
       catch (Exception e) {
         return;
       }

       if(alpha_matrix.det()==0)
       {
           System.out.println("dte == 0");
           return;
       }

       for (int k=0; k<numParam ; k++)
           parameters[k] = param_matrix.get (k,0);

       mySimulation.Counters.preNumOfRemovedCells = SimulationManager.countPoints;
       return;
   }

   /**
    * Caluclates the normal of the particle-surface collision using the bet curve fit.
    *
    * @param point the point at which the particle hits the surface.
    * @param n the normal vector to the impact surface.
```

```
     * @param t1 a tangential vector to the impact surface.
     * @param t2 a tangential vector to the impact surface.
     * @param P the particle impacting the surface.
     */
    public static boolean getNormalToPolynomial(Vector3D point, Vector3D n, Vector3D t1,
        Vector3D t2, Particle P){
        double x = point.getX();
        double y = point.getY();
        double z = point.getZ();
        int count = 0;

        // if the number of removedCells increased calculate the Parameter of Polynomial.
        if(SimulationManager.countPoints<10 ||  SimulationManager.countPoints ==
            mySimulation.Counters.preNumOfRemovedCells+10)
        {
            mySimulation.Counters.numOfPloyCalc++;
            getBestCurveFit();
        }

        Vector3D r = VectorOperation.subtract(point,SimulationManager.AJM_Nozzle.
            getPosition());
        r.setZ(0);
        double radius = r.getNorm();
        double pY = SimulationManager.AJM_Nozzle.getPosition().getY() + radius;
        double newY2 = SimulationManager.AJM_Nozzle.getPosition().getY() - radius;

        /**
         * polynomial:
         *   parameters[0] + parameters[1]*x + parameters[2]*x^2 + ... + parameters[7]*x^7

         * Derivation of the polynomial:
         *   parameters[1] + 2*parameters[2]*x + 3*parameters[3]*x^2 + ... + 7*parameters
             [7]*x^6
         */

        // derivative at point (pX,pY)
        double drv=0;

        for(int i=1;i<(degreeOfPoly+1);i++)
            drv += i*parameters[i]*Math.pow(pY,(i-1));

        if(drv==0.0)
            return false;

        // slop of the normal to the polynomial at point (pY,pZ).  pZ = slope*pY
        double slope = -1.0/drv;
        Vector3D normalYZ = new Vector3D(0.0,1.0,slope);
        normalYZ.normalize();

        // two vector on the plane(X,Y). The normalYZ must rotate around deepest cell
        Vector3D v1 = new Vector3D(0.0,-0.1,0.0);
        Vector3D v2 = new Vector3D(x-SimulationManager.AJM_Nozzle.getPosition().getX(),
                y-SimulationManager.AJM_Nozzle.getPosition().getY(),0.0);

        if(v2.getNorm()==0)
        {
            n.setPoint(EventSphereCubeCollision.bottomNorm);
            t1.setPoint(EventSphereCubeCollision.frontNorm);
            t2.setPoint(EventSphereCubeCollision.rightNorm);
            return true;
        }

        // det(A,B) = AxBy-BxAy
        double det = (v1.X*v2.Y) - (v2.X*v1.Y);
```

```
        // the angle theta is not counter-clockwise
        if(det<0)
        {
            v1.multiplyBy(-1.0);
            normalYZ.multiplyBy(-1.0);
        }

        // theta is counter-clockwise rotation angle
        double cosTheta = VectorOperation.dotProduct(v1,v2)/(v1.getNorm()*v2.getNorm());
        double theta = Math.acos(cosTheta);
        double sinTheta = Math.sin(theta);

        /**
         * rotations of z-axe in a counterclockwise direction when looking towards the
             origin
         * rotation matix:
         * Rz = (cos,-sin,0)(sin,cos,0)(0,0,1);
         * rotating using Rz: Rz*normalYZ
         */
        double Nx = cosTheta*normalYZ.X - sinTheta*normalYZ.Y;
        double Ny = sinTheta*normalYZ.X + cosTheta*normalYZ.Y;
        double Nz = normalYZ.getZ();

        n.setPoint(Nx,Ny,Nz);
        n.normalize();
        Vector3D vector = VectorOperation.subtract(point,P.getPosition());
        t1.setPoint(VectorOperation.crossProduct(n,vector).normalize());
        t2.setPoint(VectorOperation.crossProduct(n,t1).normalize());
        return true;
    }

    /**
     * Update the velocities of the given particle using a coefficient of restitution
         approach.
     *
     * @param P the particle hitting the surface.
     * @param contact point the point at which particle hits the surface.
     */
    public static Vector3D updateParticleVelocity(Particle P, Vector3D contact)
    {
        VectorOperation opr = new VectorOperation();
        Vector3D V = P.getLinearVelocity();
        Vector3D W = P.getAngularVelocity();
        Vector3D X = P.getPosition();

        /**
         * setting the coordinate system
         */
        Vector3D n=new Vector3D(), t1=new Vector3D(), t2=new Vector3D();
        Element e = P.getEvent().getPSCollision().getElement();

        /**
         * This if is deactivated by using false, since we decided to not using the best
             curve fit approach.
         */
        if(false && SimulationManager.AJM_Nozzle.getVelocity().getX()==0.0)
        {
            if( SimulationManager.SystemTime>0.01)
            {
                boolean result = getNormalToPolynomial(contact,n,t1,t2,P);

                if(!result)
                {
```

```
            if(e == Element.square)
                setCoordinateSystem_HitSquare(P, n, t1, t2);
            else
                setCoordinateSystem_HitEdge(P, n, t1, t2);
        }
    }
    else
    {
        if(e == Element.square)
            setCoordinateSystem_HitSquare(P, n, t1, t2);
        else
            setCoordinateSystem_HitEdge(P, n, t1, t2);
    }
}
else
{
     if(e == Element.square)
        setCoordinateSystem_HitSquare(P, n, t1, t2);
    else
        setCoordinateSystem_HitEdge(P, n, t1, t2);
}


double vn = opr.dotProduct(V,n);

if(vn==0.0)
{
    P.getEvent().getPSCollision().setElement(Element.square);
    setCoordinateSystem_HitSquare(P, n, t1, t2);
    vn = opr.dotProduct(V,n);
}

setCellLoss(particleTargetBox, P, n);
EventPSCollision.checkCellToBeRemoved(particleTargetBox,P);

double vt1 = opr.dotProduct(V,t1);
double vt2 = opr.dotProduct(V,t2);

// components of incident angular velocity
double wn = opr.dotProduct(W,n);
double wt1 = opr.dotProduct(W,t1);
double wt2 = opr.dotProduct(W,t2);

double eps = mySimulation.Particle.get_eps();
double rp = P.getRadius();
double f = SimulationManager.AJM_Substrate.getFriction();

// kinetic coefficients or impulse ratios
double ut1=0,ut2=0;
// the critical impulse ratios
double uc_t1, uc_t2;
// the resultant critical impulse ratio
double uc;

uc_t1 = (2*(vt1-rp*wt2))/(7*vn*(1+eps));
uc_t2 = (2*(vt2+rp*wt1))/(7*vn*(1+eps));
uc = Math.sqrt( uc_t1*uc_t1 + uc_t2*uc_t2 );

// f>uc => sphere begins rolling at some point during contact
if (Math.abs(f) > Math.abs(uc))
{
    ut1 = uc_t1;
    ut2 = uc_t2;
}
```

```
        // f<=uc => sphere slides throughout the whole impact
        else
        {
            // from figure 2.5 in david's thesis
            double a = Math.atan((vt2+rp*wt1)/(vt1-rp*wt2));
            ut1 = Math.abs(f*Math.cos(a));
            ut2 = Math.abs(f*Math.sin(a));

            // the signs of ux & uy are the same as the signs of uxc & uyc
            ut1 *= Math.signum(uc_t1);
            ut2 *= Math.signum(uc_t2);
        }

        // updated linear velocity components
        double Vn = -1*vn*eps;
        double Vt1 = vt1 - ut1*vn*(1+eps);
        double Vt2 = vt2 - ut2*vn*(1+eps);

        // updated angular velocity components
        double Wn = wn;
        double Wt1 = wt1 - (5/(2*rp))*ut2*vn*(1+eps);
        double Wt2 = wt2 + (5/(2*rp))*ut1*vn*(1+eps);

        Vector3D finVn = opr.multiply(n,Vn);
        Vector3D finVt1 = opr.multiply(t1,Vt1);
        Vector3D finVt2 = opr.multiply(t2,Vt2);
        P.setLinearVelocity(opr.add(finVn,finVt1,finVt2));

        Vector3D finWn = opr.multiply(n,Wn);
        Vector3D finWt1 = opr.multiply(t1,Wt1);
        Vector3D finWt2 = opr.multiply(t2,Wt2);
        P.setAngularVelocity(opr.add(finWn,finWt1,finWt2));
        return n;
    }


    /**
     * Sets the counters monitoring the number of surface impacts of this particle and
       surface impacts of all particles.
     *
     * @param P the particle that just impacted the surface.
     */
    public static void settingStatistics(Particle P)
    {
        mySimulation.Counters.numPSColl++;

        P.increaseHitSurface();

        if(P.getNumHitSurface()==1)
        {
            mySimulation.Counters.numHitSurface1++;
            if(P.getNumPPColl()==0)
                mySimulation.Counters.numHitSurfBeforePPColl++;
            else
                mySimulation.Counters.numHitSurfAfterPPColl++;
        }
        else if(P.getNumHitSurface()==2)
            mySimulation.Counters.numHitSurface2++;
        else if(P.getNumHitSurface()>2)
            mySimulation.Counters.numHitSurfaceMore++;
    }
}
```

**EventSphereCubeCollision.java:** The EventSphereCubeCollision class is used to detect the collision between a spherical particle and a cube.

```
package SystemEvents;

import Objects3D.Vector3D;

import Objects3D.VectorOperation;

import Objects3D.Plane;

import mySimulation.Particle;

import OcTreeADT.OctNode;

import mySimulation.SimulationManager;

import OcTreeADT.CellularOcTree.Dir;

public class EventSphereCubeCollision {
    /**
     * Represents the elements at which a particle can hit a cube.
     */
    public static enum Element { square, vertex, edge, none}

    /**
     * Creates a new instance of EventSphereCubeCollision.
     */
    public EventSphereCubeCollision() {
    }

    /**
     * Normals at 6 different faces of each cube associated with a node in the cellular
         octree..
     */
    public  static final Vector3D
            backNorm   = new Vector3D(-1,0,0),
            frontNorm  = new Vector3D(1,0,0),
            leftNorm   = new Vector3D(0,-1,0),
            rightNorm  = new Vector3D(0,1,0),
            topNorm    = new Vector3D(0,0,-1),
            bottomNorm = new Vector3D(0,0,1);

    /**
     * The CollisionPacket class used to store the information of an impact on the cube.
     */
    public static class CollisionPacket
    {
        /**
         * The spherical particle.
         */
        public Particle particle;

        /**
         * The cube.
         */
        public OctNode targetBox;

        /**
         * The normalized velocity of the impacting particle.
```

```
 */
public Vector3D normalizedVelocity;

/**
 * The variable that is set to true if the collision between the particle with the
     given velocity and the cube is detected.
 */
public boolean foundCollision;

/**
 * The nearest dist
 */
public double nearestTime;

/**
 * The point at which the particle his the cube.
 */
public Vector3D intersectionPoint;

/**
 * The element at which the particle hits the cube.
 */
public Element element;

/**
 * The normal to the face at which particle hits the cube.
 */
public Vector3D planeNormal;


/**
 * The end point of a edge impacted by particle (for the cases that particle hits
     an edge or a vertex). Otherwise is set to null.
 */
public Vector3D v1;

/**
 * The end point of a edge impacted by particle (for the cases that particle hits
     edge). Otherwise is set to null.
 */
public Vector3D v2;

/**
 * Creates a new instance of CollisionPacket.
 */
public CollisionPacket(){
}

/**
 * Creates a new instance of CollisionPacket using the information of the given
     particle.
 * @param P a particle.
 */
public CollisionPacket(Particle P){
    this.particle = P;
    this.normalizedVelocity = VectorOperation.normalize(this.particle.
        getLinearVelocity());
    this.foundCollision = false;
    this.nearestTime = Double.POSITIVE_INFINITY;
    this.intersectionPoint = null;
    this.targetBox = null;
    this.planeNormal = null;
    this.v1 = null;
    this.v2 = null;
```

```
    }

    /**
     * Sets the fields of this instance of CollisionPacket to the default values.
     */
    public void reset(){
        this.foundCollision = false;
        this.nearestTime = Double.POSITIVE_INFINITY;
        this.intersectionPoint=null;
        this.targetBox = null;
        this.planeNormal = null;
        this.v1 = null;
        this.v2 = null;
    }

    /**
     * Sets the fields of this instance of CollisionPacket to the fields values of the
       temp CollisionPacket.
     *
     * @param temp a storage including the information of an impact between the
       particle and the cube.
     */
    public void set(CollisionPacket temp){
        this.element = temp.element;
        this.foundCollision = temp.foundCollision;
        this.nearestTime = temp.nearestTime;
        this.intersectionPoint = temp.intersectionPoint;
        this.targetBox = temp.targetBox;
        this.planeNormal = temp.planeNormal;
        this.v1 = temp.v1;
        this.v2 = temp.v2;
    }

    /**
     * Returns a string representation of this CollisionPacket.
     *
     * @return a string representation of this CollisionPacket.
     */
    public String toString()
    {
        return( "\n element: "+element + "\n foundCollision: "+foundCollision +
                "\n intersection point: "+intersectionPoint + "\nplane normal: " +
                planeNormal + "\ntargetBox: " + this.targetBox + "\nvelocity: " +
                this.particle.getLinearVelocity() + "\n\n");
    }
}

/**
 * Checks for the collision between the given particle and the bounding sphere of the
   given cube.
 *
 * @param cube a cube.
 * @param P a spherical particle.
 *
 * @return true if the particle is going to collide with the bounding sphere of the
   cube; false otherwise.
 */
public static boolean IsPossibilityForCollision(OctNode cube, Particle P){
    Vector3D deltaV = P.getLinearVelocity();
    Vector3D deltaX = VectorOperation.subtract(P.getPosition(),cube.getPosition());
    // distance between the Box center and Particle center
    double distance = deltaX.getNorm();
    double R = P.getRadius() + cube.diameter;
    boolean checking = true;
```

```
        if(!(distance<R+SimulationManager.precision))
        {
            double a = deltaV.getSquareNorm();
            double b = 2.0*VectorOperation.dotProduct(deltaV,deltaX);
            double c = deltaX.getSquareNorm()-(R*R);
            double D = (b*b)-(4.0*a*c);
            if(D<0)
                return false;

            double sqrtD = Math.sqrt(D);
            double t1 = (-b + sqrtD)/(2.0*a);
            double t2 = (-b - sqrtD)/(2.0*a);
            if(!(t1>=0 || t2>=0))
                return false;
        }
        return true;
    }

    /**
     * Detects the collision between the given spherical particle and the given cube.
     *
     * @param infoPack a storage for holding the impact information.
     * @param cube a cube.
     * @param P a spherical particle.
     *
     * @return true if the particle is going to collide with the cube; false otherwise.
     */
    public static boolean detectSphereCubeCollision(CollisionPacket infoPack, OctNode cube,
         Particle P){
        Vector3D deltaV = P.getLinearVelocity();
        Vector3D deltaX = VectorOperation.subtract(P.getPosition(),cube.getPosition());
        // distance between the cube center and Particle center
        double distance = deltaX.getNorm();
        double R = P.getRadius() + cube.diameter;
        boolean checking = true;
        if(!(distance<R+SimulationManager.precision))
        {
            double a = deltaV.getSquareNorm();
            double b = 2.0*VectorOperation.dotProduct(deltaV,deltaX);
            double c = deltaX.getSquareNorm()-(R*R);
            double D = (b*b)-(4.0*a*c);
            if(D<0)
                return(infoPack.foundCollision);
            double sqrtD = Math.sqrt(D);
            double t1 = (-b + sqrtD)/(2.0*a);
            double t2 = (-b - sqrtD)/(2.0*a);
            if(!(t1>=0 || t2>=0))
                return(infoPack.foundCollision);
        }
        double halfSize = cube.getHalfSize();
        boolean found = false;
         if( (P.getZ()+P.getRadius()) <= (cube.getPosition().getZ()-halfSize) && P.
             getLinearVelocity().getZ()>0)
        {
            found = EventSphereCubeCollision.detectSphereSquareCollision(infoPack,
                        new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                            .getY()-halfSize, cube.getPosition().getZ()-halfSize),
                        new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                            .getY()+halfSize, cube.getPosition().getZ()-halfSize),
                        new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                            .getY()+halfSize, cube.getPosition().getZ()-halfSize),
                        new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                            .getY()-halfSize, cube.getPosition().getZ()-halfSize),
                        P,cube,EventSphereCubeCollision.topNorm);
```

```
        if(found)
            return infoPack.foundCollision;
    }
    else if( (P.getZ()-P.getRadius()) >= (cube.getPosition().getZ()+halfSize) && P.
        getLinearVelocity().getZ()<0)
    {
        found = EventSphereCubeCollision.detectSphereSquareCollision(infoPack,
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()+halfSize),
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()+halfSize),
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()+halfSize),
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()+halfSize),
                    P,cube,EventSphereCubeCollision.bottomNorm);
        if(found)
            return infoPack.foundCollision;
    }
    if( (P.getX()+P.getRadius()) <= (cube.getPosition().getX()-halfSize) && P.
        getLinearVelocity().getX()>0)
    {
        found = EventSphereCubeCollision.detectSphereSquareCollision(infoPack,
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()-halfSize),
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()-halfSize),
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()+halfSize),
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()+halfSize),
                    P,cube,EventSphereCubeCollision.backNorm);
        if(found)
            return infoPack.foundCollision;
    }
    else if( (P.getX()-P.getRadius()) >= (cube.getPosition().getX()+halfSize) && P.
        getLinearVelocity().getX()<0)
    {
        found = EventSphereCubeCollision.detectSphereSquareCollision(infoPack,
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()-halfSize),
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()-halfSize),
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()+halfSize),
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()+halfSize),
                    P,cube,EventSphereCubeCollision.frontNorm);
        if(found)
            return infoPack.foundCollision;
    }

    if( (P.getY()+P.getRadius()) <= (cube.getPosition().getY()-halfSize) && P.
        getLinearVelocity().getY()>0)
    {
        found = EventSphereCubeCollision.detectSphereSquareCollision(infoPack,
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()-halfSize),
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()-halfSize),
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()+halfSize),
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()-halfSize, cube.getPosition().getZ()+halfSize),
```

```
                    P,cube,EventSphereCubeCollision.leftNorm);
            if(found)
                return infoPack.foundCollision;
        }
        else if( (P.getY()-P.getRadius()) >= (cube.getPosition().getY()+halfSize) && P.
            getLinearVelocity().getY()<0)
        {
            found = EventSphereCubeCollision.detectSphereSquareCollision(infoPack,
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()-halfSize),
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()-halfSize),
                    new Vector3D(cube.getPosition().getX()+halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()+halfSize),
                    new Vector3D(cube.getPosition().getX()-halfSize, cube.getPosition()
                        .getY()+halfSize, cube.getPosition().getZ()+halfSize),
                    P,cube,EventSphereCubeCollision.rightNorm);
        }
        return(infoPack.foundCollision);
    }

    /**
     * Detects the collision between the given spherical particle and the square plane of a
         cube.
     *
     * @param infoPack a storage for holding the impact information.
     * @param p1 a corner of the square plane.
     * @param p2 a corner of the square plane.
     * @param p3 a corner of the square plane.
     * @param p4 a corner of the square plane.
     * @param P a spherical particle.
     * @param targetCube a cube including the square plane.
     * @param planeNormal a normal to the square plane.
     * @return true if the particle is going to collide with the square plane; false
         otherwise.
     */
    private static boolean detectSphereSquareCollision(CollisionPacket infoPack, Vector3D
        p1, Vector3D p2,
            Vector3D p3, Vector3D p4, Particle P, OctNode targetCube, Vector3D planeNormal)
    {
        /** NOTE: The parameters for corner of the square plane must be entered clockwise.*
            */

        // Creates the plane containing this square
        Plane squarePlane = new Plane(p1,planeNormal);
        boolean foundCollision = false;
        Element element = Element.none;
        Vector3D v1=null, v2=null;

        if(squarePlane.isFrontFacingTo(infoPack.normalizedVelocity))
        {
            /** Get interval of plane intersection */
            double t0, t1;
            boolean embeddedInPlane = false;

            /** Calculate the signed distance from sphere position to square plane */
            double signedDisToSquarePlane = squarePlane.signedDistanceTo(infoPack.particle.
                getPosition()); //infoPack.basePoint);

            double normalDotVelocity = VectorOperation.dotProduct(squarePlane.getNormal(),
                infoPack.particle.getLinearVelocity());

            /** if sphere is travelling parrallel to the plane */
            if(normalDotVelocity==0)
```

```
{
    if(Math.abs(signedDisToSquarePlane)>=P.getRadius())
    {
        /** Sphere is not embedded in plane. So, the collision is impossible */
        return false;
    }
    else
    {
        /** Sphere is embedded in plane. It intersects in the whole range[0..1]
            */
        embeddedInPlane = true;
        t0 = 0.0;
        t1 = Double.POSITIVE_INFINITY;
    }
}
else
{
    t0 = (-P.getRadius()-signedDisToSquarePlane)/normalDotVelocity;
    t1 = (P.getRadius()-signedDisToSquarePlane)/normalDotVelocity;

    if(t0 > t1)
    {
        double temp = t1;
        t1 = t0;
        t0 = temp;
    }
}

Vector3D collisionPoint = null;
double t = Double.POSITIVE_INFINITY;

if(embeddedInPlane)
{
    System.out.println("embeddedInPlane: "+embeddedInPlane);
    Objects3D.VectorOperation.errMassageExit("the particle is embedded inside
        plane!!",
            "EventHandler_PSCollision/checkSquare()");
}

/**
 * Check for collision inside square. If this happens it must be at time t0 as
     this is when the
 * sphere restson the front side. Note, this can only happen if the sphere in
     not embedded in the
 * square.
 */
if(!embeddedInPlane)
{
    Vector3D planeIntersectionPoint = VectorOperation.add( infoPack.particle.
        getPosition(),
            VectorOperation.multiply(infoPack.particle.getLinearVelocity(), t0)
                );
    boolean test = checkPointInSquare(planeIntersectionPoint, p1, p2, p3,p4);
    if(test)
    {
        element = Element.square;
        foundCollision = true;
        t = t0;
        collisionPoint = planeIntersectionPoint;
    }
}

/**
 * If we have not found the collision already we will have to sweep aphere
```

```
     against points and edges
 * of the square. Note: A collision inside the square always happen before a
     vertex or edge collision.
 * So, if we found the collision we can skip the sweep test.
 */
if(foundCollision==false)
{
    Vector3D velocity = infoPack.particle.getLinearVelocity();
    Vector3D base = infoPack.particle.getPosition();
    double velocitySquaredLength = velocity.getSquareNorm();
    double a, b, c;
    double newT;
    double sqrR = Math.pow(P.getRadius(),2.0);

    /**
     * For each vertex or edge a quadratic equation have to be solved. We
         parameterize this
     * equation as a*t^2 + b*t + c = 0. Below we calculate the parameters a, b
         and c for each
     * test.
     */

    /** Checking collision against vertices  */
    a = velocitySquaredLength;

    // vertex P1
    b = 2.0*(VectorOperation.dotProduct(velocity,VectorOperation.subtract(base,
        p1)));
    c = VectorOperation.subtract(p1,base).getSquareNorm()-sqrR;
    newT = getLowestRoot(a,b,c,t);
    if(newT!=Double.POSITIVE_INFINITY)
    {
        element = Element.vertex;
        t = newT;
        foundCollision = true;
        collisionPoint = p1;
        v1 = new Vector3D(p1);
        // meaningless - it is used randomly setting the normal for scattering
            equation after PSCollision
        v2 = new Vector3D(p2);
    }

    // vertex P2
    if(!foundCollision)
    {
        b = 2.0*(VectorOperation.dotProduct(velocity,VectorOperation.subtract(
            base,p2)));
        c = VectorOperation.subtract(p2,base).getSquareNorm()-sqrR;
        newT = getLowestRoot(a,b,c,t);
        if(newT!=Double.POSITIVE_INFINITY)
        {
            element = Element.vertex;
            t = newT;
            foundCollision = true;
            collisionPoint = p2;
            v1 = new Vector3D(p2);
            // meaningless - it is used randomly setting the normal for
                scattering equation after PSCollision
            v2 = new Vector3D(p3);
        }
    }

    // vertex P3
    if(!foundCollision)
```

```
{
    b = 2.0*(VectorOperation.dotProduct(velocity,VectorOperation.subtract(
        base,p3)));
    c = VectorOperation.subtract(p3,base).getSquareNorm()-sqrR;
    newT = getLowestRoot(a,b,c,t);
    if(newT!=Double.POSITIVE_INFINITY)
    {
        element = Element.vertex;
        t = newT;
        foundCollision = true;
        collisionPoint = p3;
        v1 = new Vector3D(p3);
        // meaningless - it is used randomly setting the normal for
            scattering equation after PSCollision
        v2 = new Vector3D(p4);
    }
}

// vertex P4
if(!foundCollision)
{
    b = 2.0*(VectorOperation.dotProduct(velocity,VectorOperation.subtract(
        base,p4)));
    c = VectorOperation.subtract(p4,base).getSquareNorm()-sqrR;
    newT = getLowestRoot(a,b,c,t);
    if(newT!=Double.POSITIVE_INFINITY)
    {
        element = Element.vertex;
        t = newT;
        foundCollision = true;
        collisionPoint = p4;
        v1 = new Vector3D(p4);
        // meaningless - it is used randomly setting the normal for
            scattering equation after PSCollision
        v2 = new Vector3D(p1);
    }
}

/** Checking collision against edges */
boolean edgeCollFound = false;

// P1--->P2:
Vector3D edge = VectorOperation.subtract(p2,p1);
Vector3D baseToVertex = VectorOperation.subtract(p1,base);
double edgeSquaredLength = edge.getSquareNorm();
double edgeDotVelocity = VectorOperation.dotProduct(edge,velocity);
double edgeDotBaseToVertex = VectorOperation.dotProduct(edge,baseToVertex);

/** Calculate parameters for equation */
a = edgeSquaredLength * (-velocitySquaredLength) + (edgeDotVelocity*
    edgeDotVelocity);
b = edgeSquaredLength*(2.0*VectorOperation.dotProduct(velocity,baseToVertex
    ))-
        2.0*edgeDotVelocity*edgeDotBaseToVertex;
c = edgeSquaredLength*(sqrR-baseToVertex.getSquareNorm())+
        edgeDotBaseToVertex*edgeDotBaseToVertex;
/** Check if the swept sphere collides against infinite edge */
newT = getLowestRoot(a,b,c,t);
if(newT!=Double.POSITIVE_INFINITY)
{
    /** Check if intersection is within line segment */
    double f = (edgeDotVelocity*newT-edgeDotBaseToVertex)/edgeSquaredLength
        ;
    if(f>=0&&f<=1.0)
```

```
        {
            /** Intersection took place within the segment */
            element = Element.edge;
            t = newT;
            foundCollision = true;
            collisionPoint = VectorOperation.add(p1, VectorOperation.multiply(
                edge,f));
            v1 = new Vector3D(p1);
            v2 = new Vector3D(p2);
            edgeCollFound = true;
        }
    }

    // P2--->P3:
    if(!edgeCollFound)
    {
        edge = VectorOperation.subtract(p3,p2);
        baseToVertex = VectorOperation.subtract(p2,base);
        edgeSquaredLength = edge.getSquareNorm();
        edgeDotVelocity = VectorOperation.dotProduct(edge,velocity);
        edgeDotBaseToVertex = VectorOperation.dotProduct(edge,baseToVertex);

        /** Calculate parameters for equation */
        a = edgeSquaredLength * (-velocitySquaredLength) + (edgeDotVelocity*
            edgeDotVelocity);
        b = edgeSquaredLength*(2.0*VectorOperation.dotProduct(velocity,
            baseToVertex))-
                2.0*edgeDotVelocity*edgeDotBaseToVertex;
        c = edgeSquaredLength*(sqrR-baseToVertex.getSquareNorm())+
                edgeDotBaseToVertex*edgeDotBaseToVertex;
        /** Check if the swept sphere collides against infinite edge */
        newT = getLowestRoot(a,b,c,t);
        if(newT!=Double.POSITIVE_INFINITY)
        {
            /** Check if intersection is within line segment */
            double f = (edgeDotVelocity*newT-edgeDotBaseToVertex)/
                edgeSquaredLength;
            if(f>=0&&f<=1.0)
            {
                /** Intersection took place within the segment */
                element = Element.edge;
                t = newT;
                foundCollision = true;
                collisionPoint = VectorOperation.add(p2, VectorOperation.
                    multiply(edge,f));
                v1 = new Vector3D(p2);
                v2 = new Vector3D(p3);
                edgeCollFound = true;
            }
        }
    }

    // P3--->P4:
    if(!edgeCollFound)
    {
        edge = VectorOperation.subtract(p4,p3);
        baseToVertex = VectorOperation.subtract(p3,base);
        edgeSquaredLength = edge.getSquareNorm();
        edgeDotVelocity = VectorOperation.dotProduct(edge,velocity);
        edgeDotBaseToVertex = VectorOperation.dotProduct(edge,baseToVertex);

        /** Calculate parameters for equation */
        a = edgeSquaredLength * (-velocitySquaredLength) + (edgeDotVelocity*
            edgeDotVelocity);
```

```
        b = edgeSquaredLength*(2.0*VectorOperation.dotProduct(velocity,
            baseToVertex))-
                2.0*edgeDotVelocity*edgeDotBaseToVertex;
        c = edgeSquaredLength*(sqrR-baseToVertex.getSquareNorm())+
                edgeDotBaseToVertex*edgeDotBaseToVertex;
        /** Check if the swept sphere collides against infinite edge */
        newT = getLowestRoot(a,b,c,t);
        if(newT!=Double.POSITIVE_INFINITY)
        {
            /** Check if intersection is within line segment */
            double f = (edgeDotVelocity*newT-edgeDotBaseToVertex)/
                edgeSquaredLength;
            if(f>=0&&f<=1.0)
            {
                /** Intersection took place within the segment */
                element = Element.edge;
                t = newT;
                foundCollision = true;
                collisionPoint = VectorOperation.add(p3, VectorOperation.
                    multiply(edge,f));
                v1 = new Vector3D(p3);
                v2 = new Vector3D(p4);
                edgeCollFound = true;
            }
        }
    }

    // P4--->P1:
    if(!edgeCollFound)
    {
        edge = VectorOperation.subtract(p1,p4);
        baseToVertex = VectorOperation.subtract(p4,base);
        edgeSquaredLength = edge.getSquareNorm();
        edgeDotVelocity = VectorOperation.dotProduct(edge,velocity);
        edgeDotBaseToVertex = VectorOperation.dotProduct(edge,baseToVertex);

        /** Calculate parameters for equation */
        a = edgeSquaredLength * (-velocitySquaredLength) + (edgeDotVelocity*
            edgeDotVelocity);
        b = edgeSquaredLength*(2.0*VectorOperation.dotProduct(velocity,
            baseToVertex))-
                2.0*edgeDotVelocity*edgeDotBaseToVertex;
        c = edgeSquaredLength*(sqrR-baseToVertex.getSquareNorm())+
                edgeDotBaseToVertex*edgeDotBaseToVertex;
        /** Check if the swept sphere collides against infinite edge */
        newT = getLowestRoot(a,b,c,t);
        if(newT!=Double.POSITIVE_INFINITY)
        {
            /** Check if intersection is within line segment */
            double f = (edgeDotVelocity*newT-edgeDotBaseToVertex)/
                edgeSquaredLength;
            if(f>=0&&f<=1.0)
            {
                /** Intersection took place within the segment */
                element = Element.edge;
                t = newT;
                foundCollision = true;
                collisionPoint = VectorOperation.add(p4, VectorOperation.
                    multiply(edge,f));
                v1 = new Vector3D(p4);
                v2 = new Vector3D(p1);
            }
        }
    }
```

```
            } // if(foundCollision==false)

            /** Set result */
            if(foundCollision == true)
            {
                /** 't' is the time of the collision */
                double disToCollision = t*infoPack.particle.getLinearVelocity().getNorm();
                //System.out.println("inside check square: " + disToCollision);
                /** Check if this square qualifies fro the closest hit (first hit or the
                    closest???) */
                if(infoPack.foundCollision==false || t<infoPack.nearestTime)
                {
                    if(t>0)
                    {
                        infoPack.nearestTime = t;
                        infoPack.intersectionPoint = collisionPoint;
                        infoPack.foundCollision = true;
                        infoPack.element = element;
                        infoPack.targetBox = targetCube;
                        infoPack.planeNormal = planeNormal;
                        infoPack.v1 = v1;
                        infoPack.v2 = v2;
                    }
                }
            }
        }
        return foundCollision;
    }

    /**
     * Solves the quadratic equation.
     *
     * @param a the coefficient of x^2.
     * @param b the coefficient of x.
     * @param c the constant term.
     * @param maxR the minimum calculated time of collisions between the particle and
          elements of the cube.
     *
     * @return the smallest positive soluation if there is any and the solution also is
          smaller that the previously calculated time, maxR; infinity otherwise.
     */
    private static double getLowestRoot(double a, double b, double c, double maxR)
    {
        /** Check if the solution exists */
        double determinant = b*b - 4.0*a*c;
        double invalidAns=Double.POSITIVE_INFINITY;

        /** If the determinant is negative it means no solutions*/
        if(determinant<0.0)
            return invalidAns;

        double sqrtD = Math.sqrt(determinant);
        double r1 = (-b - sqrtD)/(2.0*a);
        double r2 = (-b + sqrtD)/(2.0*a);

        if(r1>r2)
        {
            double temp = r2;
            r2 = r1;
            r1 = temp;
        }

        if(r1>0 && r1<maxR)
            return r1;
```

```
        if(r2>0 && r2<maxR)
            return r2;

        /** No (valid) solutions */
        return invalidAns;
    }

    /**
     * Checks if the given point is contained inside the square.
     *
     * @param point a point on the plane containing the square plane.
     * @param p1 a corner of the square plane.
     * @param p2 a corner of the square plane.
     * @param p3 a corner of the square plane.
     * @param p4 a corner of the square plane.
     *
     * @return true if the given point is contained in the square plane; false otherwise.
     */
    private static boolean checkPointInSquare(Vector3D point,Vector3D p1,Vector3D p2,
        Vector3D p3,Vector3D p4){
        double recX=Double.POSITIVE_INFINITY, recY=Double.POSITIVE_INFINITY,
                x=Double.POSITIVE_INFINITY, y=Double.POSITIVE_INFINITY, width=0, height=0;

        if(p1.getX()==p2.getX() && p2.getX()==p3.getX())
        {
            recX = Math.min(Math.min(p1.getY(),p2.getY()),p3.getY());
            recY = Math.max(Math.max(p1.getZ(),p2.getZ()),p3.getZ());
            x = point.getY();
            y = point.getZ();

           point.setX(p1.getX());
            if((p1.getY()-p2.getY())!=0)
                width = Math.abs(p1.getY()-p2.getY());
            else
                width = Math.abs(p1.getY()-p3.getY());

            if((p1.getZ()-p2.getZ())!=0)
                height = Math.abs(p1.getZ()-p2.getZ());
            else
                height = Math.abs(p1.getZ()-p3.getZ());
        }
        else if(p1.getY()==p2.getY() && p2.getY()==p3.getY())
        {
            recX = Math.min(Math.min(p1.getX(),p2.getX()),p3.getX());
            recY = Math.max(Math.max(p1.getZ(),p2.getZ()),p3.getZ());
            x = point.getX();
            y = point.getZ();

            point.setY(p1.getY());
            if((p1.getX()-p2.getX())!=0)
                width = Math.abs(p1.getX()-p2.getX());
            else
                width = Math.abs(p1.getX()-p3.getX());

            if((p1.getZ()-p2.getZ())!=0)
                height = Math.abs(p1.getZ()-p2.getZ());
            else
                height = Math.abs(p1.getZ()-p3.getZ());
        }
        else if(p1.getZ()==p2.getZ() && p2.getZ()==p3.getZ())
        {
            recX = Math.min(Math.min(p1.getX(),p2.getX()),p3.getX());
            recY = Math.max(Math.max(p1.getY(),p2.getY()),p3.getY());
```

```
            x = point.getX();
            y = point.getY();

            point.setZ(p1.getZ());
            if((p1.getX()-p2.getX())!=0)
                width = Math.abs(p1.getX()-p2.getX());
            else
                width = Math.abs(p1.getX()-p3.getX());

            if((p1.getY()-p2.getY())!=0)
                height = Math.abs(p1.getY()-p2.getY());
            else
                height = Math.abs(p1.getY()-p3.getY());
        }
        else
            Objects3D.VectorOperation.errMassageExit("its not straight rectangleee!!","
                checkPointInSquare");
        return (x>recX&&x<(recX+width)&&y<recY&&y>(recY-height));
    }
}
```

**EventTransfer.java:** The EventTransfer class is used to predict and handle the transfer events.

```
package SystemEvents;

import Objects3D.VectorOperation;

import Objects3D.Vector3D;

import java.util.List;

import mySimulation.*;

import SystemEnvironment.*;

public class EventTransfer {
    /**
     * An auxiliary variable which allows for reusing memory.
     */
    private static eventStorage BackFront = new eventStorage();

    /**
     * An auxiliary variable which allows for reusing memory.
     */
    private static eventStorage LeftRight = new eventStorage();

    /**
     * An auxiliary variable which allows for reusing memory.
     */
    private static eventStorage TopBottom = new eventStorage();

    /**
     * Creates a new instance of EventTransfer
     */
    public EventTransfer() {
    }

    /**
     * The eventStorage class is used to store the information of the transfer events.
     */
    public static class eventStorage
    {
        /**
         * The time of the transfer.
         */
        double time;

        /**
         * The velocity at which the particle trasfers.
         */
        double velocity;

        /**
         * The position of the particle after transfer.
         */
        double position;

        /**
         * The type of the cell neighbor to which the particle is transfering.
         */
```

```
    Event.enteredCubeType type;

    /**
     * Set the values of the fields defined in the eventStorage to the new values.
     */
    public void setInfo(double T, double position, double velocity, Event.
        enteredCubeType Type)
    {
        time = T;
        type = Type;
        this.velocity = velocity;
        this.position = position;
    }
}

 /**
 * Handles the trasfer event of a given particle. It makes the particle to enter a new
     cube.
 *
 * @param  P  a given particle.
 */
public static void handelTransfer(Particle P)
{
    SystemEnvironment.Cell currentCube = P.getCurrentSpace();
    Event.enteredCubeType type = P.getEvent().getTransfer().getEnteredCubeType();
    Vector3D Destination = new Vector3D();
    Cell originCube=null, destCube=null;
    int newCubeX=-1, newCubeY=-1, newCubeZ=-1;

    if(currentCube==null){
        P.setUnusualSituation(true,"");
        EventTransfer.removeParticle(P);
        return;
    }

    newCubeX = currentCube.index1;
    newCubeY = currentCube.index2;
    newCubeZ = currentCube.index3;

    if(type==Event.enteredCubeType.back)
       newCubeX--;
    else if(type==Event.enteredCubeType.front)
       newCubeX++;
    else if(type==Event.enteredCubeType.left)
       newCubeY--;
    else if(type==Event.enteredCubeType.right)
       newCubeY++;
    else if(type==Event.enteredCubeType.bottom)
       newCubeZ++;
    else if(type==Event.enteredCubeType.top)
       newCubeZ--;

    mySimulation.SimulationManager.SystemTime = P.getEvent().getTransfer().getMinTime()
        ;


    /**
     * For the cases that the particle is leaving the boundary of the system.
     */
    if( newCubeX > mySimulation.SimulationManager.AJM_Environment.numOfCubes.getX()-1 ||
        newCubeX < 0 ||
        newCubeY > mySimulation.SimulationManager.AJM_Environment.numOfCubes.getY()-1 ||
            newCubeY < 0 ||
        newCubeZ < 0)
```

```
    {
        EventTransfer.removeParticle(P);
        return;
    }

     Destination = P.getNextDestination();
     Vector3D prevDist = new Vector3D(P.getPosition());

     if(P.getOldSpace()!=null)
     {
         P.getOldSpace().deleteMemberFrom(P);
         P.setOldSpace(null);
     }

     /*
      * This "if" is for the case that particle hit the surface.
      */
     if(newCubeZ > mySimulation.SimulationManager.AJM_Environment.numOfCubes.getZ()-1 )
     {
         System.out.println("\n P Position : "+ P.getPosition() +"\n P. Velocity : " + P.
             getLinearVelocity()+ "\n P.currSpace : " + P.getCurrentSpace() +
                 "\n P.event : "+ P.getEvent() +
                 "\nP--->" + P +
                 "\ncurrEvent--> " + mySimulation.SimulationManager.currEventType +
                 "\nSystime : " +  mySimulation.SimulationManager.SystemTime +
                 "\nP.time : " + P.getParticleTime() +
                 "\nprevDist : " + prevDist);
         VectorOperation.errMassage("!!!!!!!!!!!!!!!!!!!!!!!!!","Environment/exitCube()")
             ;
         EventTransfer.removeParticle(P);
         return;
     }

    originCube = P.getCurrentSpace();
    P.setCurrentSpace(null);
    P.setOldSpace(originCube);
    P.setPosition(Destination);
    P.setParticleTime();

    destCube = mySimulation.SimulationManager.AJM_Environment.getSpaceElement(newCubeX,
        newCubeY, newCubeZ);
    destCube.addMemberTo(P);
    P.setCurrentSpace(destCube);
    return;
}

/**
 * Detects the transfer event for the given particle.
 *
 * @param P a particle.
 *
 * @return false if the particle has left the system.
 */
public static boolean detectTransfer(Particle P)
{
    mySimulation.Counters.predictTransfer++;
    double time, dis;
    Event.enteredCubeType dir;
    double Rp = P.getRadius() + (0.3*P.getRadius());
    SystemEnvironment.Cell tempCube=null;
    eventStorage result;
    SystemEnvironment.Cell currentCube = P.getCurrentSpace();
    Event.enteredCubeType cubeType = Event.enteredCubeType.undefinedCube;
    Vector3D cubeSize;
```

```
cubeSize = currentCube.getSize();
Vector3D Xp = P.getPosition();
Vector3D Vp = P.getLinearVelocity();
Vector3D envNumCubes = SimulationManager.AJM_Environment.getNumOfCubes();

if(Vp.getX()<=0)
{
    dis = currentCube.getCubePosition().getX()- (Xp.getX()-Rp);
    dir = Event.enteredCubeType.back;
}
else
{
    dis =  currentCube.getCubePosition().getX() + cubeSize.getX() - (Xp.getX()+Rp);
    dir = Event.enteredCubeType.front;
}

time = dis/Vp.getX();
BackFront.setInfo(time,dis,Vp.getX(),dir);

if(BackFront.time<0)
{
    if(P.getLinearVelocity().getX()<0)
    {
        if((P.getCurrentSpace().index1-1)<0)
        {
            mySimulation.Counters.predictTransfer--;
            return false;
        }
        if(P.getOldSpace()!=null)
            P.getOldSpace().deleteMemberFrom(P);

        P.setOldSpace(P.getCurrentSpace());

        tempCube = SimulationManager.AJM_Environment.getSpaceElement(P.
            getCurrentSpace().index1-1,P.getCurrentSpace().index2,P.getCurrentSpace
            ().index3);
        tempCube.addMemberTo(P);
        P.setCurrentSpace(tempCube);
    }
    else if(P.getLinearVelocity().getX()>0)
    {
        if((P.getCurrentSpace().index1+1)>envNumCubes.getX()-1)
        {
            mySimulation.Counters.predictTransfer--;
            return false;
        }
        if(P.getOldSpace()!=null)
            P.getOldSpace().deleteMemberFrom(P);
        P.setOldSpace(P.getCurrentSpace());
        tempCube = SimulationManager.AJM_Environment.getSpaceElement(P.
            getCurrentSpace().index1+1,P.getCurrentSpace().index2,P.getCurrentSpace
            ().index3);
        tempCube.addMemberTo(P);
        P.setCurrentSpace(tempCube);
    }
    mySimulation.Counters.predictTransfer--;
    return (detectTransfer(P));
}
if(Vp.getY()<=0)
{
    dis = currentCube.getCubePosition().getY()+Rp-Xp.getY();
    dir = Event.enteredCubeType.left;
}
else
```

```
{
    dis = currentCube.getCubePosition().getY()+cubeSize.getY()-Rp-Xp.getY();
    dir = Event.enteredCubeType.right;
}

time = dis/Vp.getY();
LeftRight.setInfo(time,dis,Vp.getY(),dir);

if(LeftRight.time<0)
{
    if(P.getLinearVelocity().getY()<0)
    {
        if((P.getCurrentSpace().index2-1)<0)
           {
            mySimulation.Counters.predictTransfer--;
            return false;
        }
        if(P.getOldSpace()!=null)
            P.getOldSpace().deleteMemberFrom(P);
        P.setOldSpace(P.getCurrentSpace());
        tempCube = SimulationManager.AJM_Environment.getSpaceElement(P.
            getCurrentSpace().index1,P.getCurrentSpace().index2-1,P.getCurrentSpace
            ().index3);
        tempCube.addMemberTo(P);
        P.setCurrentSpace(tempCube);
    }
    else if(P.getLinearVelocity().getY()>0)
    {
        if((P.getCurrentSpace().index2+1)>envNumCubes.getY()-1)
        {
            mySimulation.Counters.predictTransfer--;
            return false;
        }
        if(P.getOldSpace()!=null)
            P.getOldSpace().deleteMemberFrom(P);
        P.setOldSpace(P.getCurrentSpace());
        tempCube = SimulationManager.AJM_Environment.getSpaceElement(P.
            getCurrentSpace().index1,P.getCurrentSpace().index2+1,P.getCurrentSpace
            ().index3);
        tempCube.addMemberTo(P);
        P.setCurrentSpace(tempCube);
    }
    mySimulation.Counters.predictTransfer--;
    return (detectTransfer(P));
}
if(Vp.getZ()<=0)
{
    dis = currentCube.getCubePosition().getZ()+Rp-Xp.getZ();
    dir = Event.enteredCubeType.top;
}
else
{
    dis = currentCube.getCubePosition().getZ()+cubeSize.getZ()-Rp-Xp.getZ();
    dir = Event.enteredCubeType.bottom;
}

time = dis/Vp.getZ();
TopBottom.setInfo(time,dis,Vp.getZ(),dir);

if(TopBottom.time<0)
{
    if(P.getLinearVelocity().getZ()<0)
    {
        if((P.getCurrentSpace().index3-1)<0)
```

```
                {
                    mySimulation.Counters.predictTransfer--;
                    return false;
                }
                if(P.getOldSpace()!=null)
                    P.getOldSpace().deleteMemberFrom(P);
                P.setOldSpace(P.getCurrentSpace());
                tempCube = SimulationManager.AJM_Environment.getSpaceElement(P.
                    getCurrentSpace().index1,P.getCurrentSpace().index2,P.getCurrentSpace()
                    .index3-1);
                tempCube.addMemberTo(P);
                P.setCurrentSpace(tempCube);
            }
            else if(P.getLinearVelocity().getZ()>0)
            {
                if((P.getCurrentSpace().index3+1)>envNumCubes.getZ()-1)
                {
                    mySimulation.Counters.predictTransfer--;
                    return false;
                }
                if(P.getOldSpace()!=null)
                    P.getOldSpace().deleteMemberFrom(P);
                P.setOldSpace(P.getCurrentSpace());

                tempCube = SimulationManager.AJM_Environment.getSpaceElement(P.
                    getCurrentSpace().index1,P.getCurrentSpace().index2,P.getCurrentSpace()
                    .index3+1);
                tempCube.addMemberTo(P);
                P.setCurrentSpace(tempCube);
            }
            mySimulation.Counters.predictTransfer--;
            return (detectTransfer(P));
        }
        result = BackFront;
        result = (LeftRight.time<result.time) ? LeftRight : result;
        result = (TopBottom.time<result.time) ? TopBottom : result;
        double t = (Rp/Vp.getNorm())*-1.0;
        P.setNextDestination(VectorOperation.add(VectorOperation.multiply( (VectorOperation
            .divide(Vp,result.velocity)) , result.position) , Xp));
        P.getEvent().getTransfer().setPrevSystemTime();
        P.getEvent().getTransfer().setInfo( result.time + P.getParticleTime() , result.type
            );
        mySimulation.Counters.predictTransfer--;
        return true;
    }

    /**
     * Removed the particle and all of its associated components from the system.
     *
     * @param P a particle.
     */
    public static void removeParticle(Particle P)
    {
        if(P.getNumHitSurface()==0 &&  P.getNumPPColl()==0)
            mySimulation.Counters.numParExitBeforeColl_Hit++;
        mySimulation.Counters.numExitEnv1++;
        mySimulation.Counters.numOfCurrParticles--;

        List cells = P.getContainingCube();
        P.setContainingCube(null);
        Cell curr = P.getCurrentSpace(), prev = P.getOldSpace();

        if(cells!=null)
        {
```

```
            for(int i=0;i<cells.size();i++)
            {
                Cell c = (Cell)cells.get(i);
                c.getMembers().remove(P);
            }
        }
        if(P.isUnusualSituation()==true)
            mySimulation.Counters.unusual++;

        P.setDeleted(true);

        if(P.getEvent()!=null)
            SimulationManager.AJM_EventHeap.Delete(P);
        else
        {
            System.out.println("HeapRoot: " + SimulationManager.AJM_EventHeap.getHeapRoot()
                );
            SimulationManager.AJM_EventHeap.handleNullRoot();
        }

        if(P.getCurrentSpace()!=null)
        {
            P.getCurrentSpace().deleteMemberFrom(P);
            P.setCurrentSpace(null);
        }
        if(P.getOldSpace()!=null)
        {
            P.getOldSpace().deleteMemberFrom(P);
            P.setOldSpace(null);
        }
    }
}
```

**Plot.java:** The Plot class is used to presents the three-dimensional view of the erosion profiled. It is implemented using the the VisAD package. The implementation is based on the VisAD Tutorial example 4-06.

```
package Visualization;

import mySimulation.SimulationManager;

import visad.*;

import visad.util.*;

import visad.java3d.DisplayImplJ3D;

import java.rmi.RemoteException;

import java.awt.*;

import javax.swing.*;

import java.awt.event.*; import OcTreeADT.*;

import OcTreeADT.CellularOcTree.Dir;

public class Plot {
    /**
     * The number of cells in one row of the surface (gives the dimensions of the
         removedCell array).
     */
    public static int count = (int)(SimulationManager.AJM_Environment.getWidth()/OcTreeADT.
        CellularOcTree.getCellSize());

    /**
     * An array used to stored the depth of the surface cells.
     */
    public static double[][] removedCell = new double[count][count];

    /**
     * The domain quantity longitude.
     */
    private RealType longitude;

    /**
     * The domain quantity latitude.
     */
    private RealType latitude;

    /**
     * The dependent quantity altitude.
     */
    private RealType altitude;

    /**
     * Tuple to pack longitude and latitude together.
     */
    private RealTupleType domain_tuple;

    /**
     * The function (domain_tuple -> altitude)
```

```
 */
private FunctionType func_domain_alt;

/**
 * Our Data values for the domain are represented by the Set.
 */
private Set domain_set;

/**
 * The Data class FlatField.
 */
private FlatField vals_ff;

/**
 * The DataReference from data to display.
 */
private DataReferenceImpl data_ref;

/**
 * The 2D display.
 */
private DisplayImpl display;

/**
 * The maps of 2D display.
 */
private ScalarMap latMap, lonMap;

/**
 * The maps of 2D display.
 */
private ScalarMap altMap, altRGBMap;

/**
 * The maps of 2D display.
 */
private ScalarMap altAlphaMap;

/**
 * The control for 3-component Color DisplayRealType
 */
private ColorControl colCont;

/**
 * the depths of the deepest and shallowest cells.
 */
private double maxDepth, minDepth;

/**
 * The color table.
 */
private float[][] myColorTable;

/**
 * Creates a new instance of Plot and displays the three-dimensional view of the
    erosion profile.
 *
 * @param maxDepth the depth of the deepest cell.
 */
public Plot(double maxDepth) throws RemoteException, VisADException
{
  this.maxDepth = maxDepth;
  this.minDepth = 0.5*(-maxDepth);
  System.out.println(this.maxDepth);
```

```
/**
 * Creates the quantities
 * Uses RealType(String name, Unit unit, Set set);
 */
latitude = RealType.getRealType("latitude", SI.meter, null);
longitude = RealType.getRealType("longitude", SI.meter, null);
domain_tuple = new RealTupleType(latitude, longitude);
altitude = RealType.getRealType("altitude", null, null);

/**
 * Create a FunctionType (domain_tuple -> range_tuple )
 * Uses FunctionType(MathType domain, MathType range)
 */
func_domain_alt = new FunctionType( domain_tuple, altitude);

/** Creates the domain Set
 * LinearDSet(MathType type, double first1, double last1, int lengthX, double first2,
      double last2, int lengthY)
 */
int NCOLS = count;
int NROWS = count;
domain_set = new Linear2DSet(domain_tuple, 15.0, 0.0, NROWS,
        15.0, 0.0, NCOLS);


/**
 * The flat array
 */
double[][] flat_samples = new double[1][NCOLS * NROWS];

/**
 * Fills up the flat_samples array with the altitude values.
 */
int index = 0;
for(int c = 0; c < NCOLS; c++)
{
    for(int r = 0; r < NROWS; r++)
    {
    flat_samples[0][ index ] = this.getRemovedCell()[r][c];
    index++;
    }
}

/**
 * Creates a FlatField
 * Uses FlatField(FunctionType type, Set domain_set)
 */
vals_ff = new FlatField( func_domain_alt, domain_set);
vals_ff.setSamples( flat_samples , false );

/**
 * Creates Display and its maps
 */
display = new DisplayImplJ3D("surface_plot");

/**
 * Gets display's graphics mode control and draw scales
 */
GraphicsModeControl dispGMC = (GraphicsModeControl)  display.getGraphicsModeControl()
    ;
dispGMC.setScaleEnable(true);

/**
```

```
 * Enables Texture
 */
dispGMC.setTextureEnable(false);

/**
 * Creates the ScalarMaps: latitude to XAxis, longitude to YAxis and
 * altitude to ZAxis and to RGB
 * Uses ScalarMap(ScalarType scalar, DisplayRealType display_scalar)
 */
latMap = new ScalarMap( latitude,    Display.YAxis );
lonMap = new ScalarMap( longitude, Display.XAxis );

altRGBMap = new ScalarMap( altitude,  Display.RGB );
altMap = new ScalarMap( altitude,  Display.ZAxis );
altAlphaMap = new ScalarMap( altitude,  Display.Alpha );

/**
 * Adds maps to display
 */
display.addMap( latMap );
display.addMap( lonMap );
display.addMap( altMap );
display.addMap( altRGBMap );

/**
 * Creates a different color table
 */
int tableLength = 10;
myColorTable = new float[3][tableLength];

for(int i=0;i<tableLength;i++)
{
    // red component
    myColorTable[0][i]= (float) 1.0f - (float)i / ((float)tableLength-1.0f);
    // green component
    myColorTable[1][i]= (float) (float)i / ((float)tableLength-1.0f);
    // blue component
    myColorTable[2][i]= (float) 0.50f;

}

myColorTable[0][9]=0.9f;
myColorTable[1][9]=0.9f;
myColorTable[2][9]=0.9f;

/**
 * Gets the ColorControl from the altitude RGB map
 */
colCont = (ColorControl) altRGBMap.getControl();

/**
 * Sets the table
 */
colCont.setTable(myColorTable );

/**
 * Creates a data reference and sets the FlatField as our data
 */
data_ref = new DataReferenceImpl("data_ref");
data_ref.setData( vals_ff );
display.addReference( data_ref);

/**
 * Sets maps ranges
```

```
        */
        latMap.setRange(-1.0f, 16.0f);
        lonMap.setRange(-1.0f, 16.0f);
        altMap.setRange(this.maxDepth, this.minDepth);

        /**
         * Create application window and add display to window
         */
        JFrame jframe = new JFrame("Erosion Profile");
        jframe.getContentPane().add(display.getComponent());
        jframe.setSize(600, 600);
        jframe.setVisible(true);
    }

    /**
     * Updates the value of an element in the removedCell array which is associated with
         the cell centered at the given poisiton.
     *
     * @param cellPosition the position of the removed cell.
     */
    public static void setRemovedCell(Objects3D.Vector3D cellPosition)
    {
        double cellSize = OcTreeADT.CellularOcTree.getCellSize();
        int X = (int)(cellPosition.getX()/cellSize);
        int Y = (int)(cellPosition.getY()/cellSize);
        if(X>=0 && X<count && Y>=0 && Y<count)
        {
            try{
                removedCell[X][Y]=-(cellPosition.getZ()-SimulationManager.AJM_Environment.
                    getDepth()+CellularOcTree.getMinHalfSize());
            } catch(ArrayIndexOutOfBoundsException e){}
        }
    }

    /**
     * Returns the removedCell array.
     *
     * @return  the removedCell array.
     */
    public double[][] getRemovedCell() {
        return removedCell;
    }
}
```

**Counters.java:** The Counters class defines all the counters monitoring the behavior of the simulation.

```
package mySimulation;

public class Counters {
    /**
     * The number of predicted particle-particle collisions.
     */
    public static int predictPPColl=0;

    /**
     * The number of predicted particle-surface collisions.
     */
    public static int predictPSColl=0;

    /**
     * The number of predicted transfers.
     */
    public static int predictTransfer=0;

    /**
     * The number of reused memory spaces for creating new particles.
     */
    public static int reusedSpaces=0;

    /**
     * The number of unsuccessful attemps to reuse memory spaces for creating new particles
     */
    public static int notReusedSpaces=0;

    /**
     * The number of unusual situations.
     */
    public static int unusual = 0;

    /**
     * The number of obsereved situations a particle has overlapped with some other
       particles at the time of its launch.
     */
    public static int overAtLaunch = 0;

    /**
     * Different reasons for overlapping.
     */
    public static int infinit = 0, remOverLapTime = 0, disLessR = 0, isNotInBound = 0;

    /**
     * The number of removed cells from the substrate.
     */
    public static int numOfRemovedCells = 0;

    /**
     * The number of removed cells at the time that best-curve-fit was calculated.
     */
    public static int preNumOfRemovedCells = 0;

    /**
     * The number of caculation of best-curve-fit
```

```
 */
public static int numOfPloyCalc = 0;

public int numOfEvents = 0;

/**
 * The number of particle-particle collisions.
 */
public static int numPPColl = 0;

/**
 * The number of successful particle-surface collisions.
 */
public static int numPSColl = 0;

/**
 * The number of unsuccessful particle-surface collisions.
 */
public static int numUnsuccessPSColl = 0;

/**
 * The number of lauched particles.
 */
public static int numLunch = 0;

/**
 * The number of trasfers.
 */
public static int numTransfer = 0;

/**
 * The number of particles that hit the target surface once.
 */
public static int numHitSurface1 = 0;

/**
 * The number of particles that hit the target surface twice.
 */
public static int numHitSurface2 = 0;

/**
 * The number of particles that hit the target surface more than two times.
 */
public static int numHitSurfaceMore = 0;

/**
 * The number of particles that hit the target surface before going through any
     particle-particle collision.
 */
public static int numHitSurfBeforePPColl = 0;

/**
 * The number of particles that hit the target surface and have gone though at least
     one particle-particle collision.
 */
public static int numHitSurfAfterPPColl = 0;

/**
 * The number of particles that have left the system without undergoing any particle-
     particle collision and
 * particle-surface collision.
 */
public static int numParExitBeforeColl_Hit = 0;
```

```
    /**
     * The number of particles that have left the boundary of the system.
     */
    public static int numExitEnv1 = 0;

    /**
     * The number of overlaps observed between particles.
     */
    public static int overlapCounting = 0;

    /**
     * The number of particles inside the system.
     */
    public static int numOfCurrParticles = 0;

    /**
     * The number of particles partially embedded iside their tsarget cell.
     */
    public static int foundPSCollEmbed = 0;

    /**
     * The number of particle-surface collisoins on the square planes of the surface cells.
     */
    public static int numPlaneNormal = 0;

    /**
     * Creates a new instance of Counters.
     */
    public Counters() {
    }
}
```

**Nozzle.java:** The Nozzle class is implemented to handle and predict launch events of particle and store the nozzle information.

```
import SystemEvents.EventPPCollision;

import Objects3D.Vector3D;

import Objects3D.VectorOperation;

import java.lang.Math;

import SystemEvents.Event;

public class Nozzle {
    /**
     * The radius of the nozzle.
     */
    private double nozzleRadius;

    /**
     * The number of particles launched per second.
     */
    private double lunchFrequency;

    /**
     * The interval between two subsequent launch events.
     */
    private double lunchingTimeInterval;

    /**
     * The distance that nozzle scans to create the channel.
     */
    private double passDistance;

    /**
     * The distance between the nozzle and the target surface along the nozzle centerline.
     */
    private double stand_off_dis;

    /**
     * The focus coefficient which defines divergence of the incident stream.
     */
    private double beta;

    /**
     * The variable that is set to true for the case of non-stationary nozzle.
     */
    private boolean channel;

    /**
     * The current position of the nozzle.
     */
    private Vector3D position;

    /**
     * The velocity of the nozzle.
     */
    private Vector3D velocity;

    /**
```

```
 * The start point of the pass distance.
 */
private double startPoint;

/**
 * The end point of the pass distance.
 */
private double endPoint;

/**
 * The angle by which the nozzle is oriented.
 */
private double orientationAngle;

/**
 * The sine of the orientationAngle for the ablique nozzle.
 */
private double sinA;

/**
 * The cosine of the orientationAngle for the ablique nozzle.
 */
private double cosA;

/**
 * The cutoff radius at which it is highly unlikely any particle arrive.
 */
private double Rc;

/**
 * The radius of particles.
 */
private double particleRadius;

/**
 * The density of abrasive particles.
 */
private double particleDensity;

/**
 * The mass of particles.
 */
private double particleMass;

/**
 * The constants used in particle velocity distribution (V(r)=V(0){vConstant1-
     vConstant1r/h}).
 */
private double vConstant1, vConstant2;

/**
 * The maximum velocity of particles observed at the nozzle centerline.
 */
private double particleVmax;

/**
 * Creates a new instance of Nozzle
 *
 * @param nozzRadius the radius of the nozzle.
 * @param lunchFrequency the number of particles launched per second.
 * @param stand_off_dis the distance between the nozzle and the target surface along
     the nozzle centerline.
 * @param beta the focus coefficient.
 * @param position the initial position of the nozzle.
```

```
     * @param velocity the velocity of the nozzle.
     * @param particleRadius the radius of particles.
     * @param particleMass the mass of particles.
     * @param passDistance the distance that nozzle scans to create the channel.
     * @param channel the variable that is set to true for the case of non-stationary
       nozzle.
     * @param vConstant1 the first constant used is particle velocity distribution.
     * @param vConstant2 the second constant used is particle velocity distribution.
     * @param orientationAngle the angle by which the nozzle is oriented.
     */
    public Nozzle( double nozzRadius, double lunchFrequency, double stand_off_dis, double
        beta, Vector3D position, Vector3D velocity,
                    double particleRadius, double particleMass, double particleDensity,
                        double passDistance, boolean channel,
                    double vConstant1, double vConstant2, double particleVmax, double
                        orientationAngle)
    {
        /**** Nozzle features ****/
        this.nozzleRadius = nozzRadius;
        this.lunchFrequency = lunchFrequency;
        this.lunchingTimeInterval = 1/lunchFrequency;
        this.stand_off_dis = stand_off_dis;
        this.beta = beta;
        this.position = new Vector3D(position);
        this.velocity = new Vector3D(velocity);
        this.Rc = stand_off_dis*(Math.sqrt(-4 * Math.log(1.0-0.9999)))/(2.0*beta);
        this.passDistance = passDistance;
        this.vConstant1 = vConstant1;
        this.vConstant2 = vConstant2;
        this.particleVmax = particleVmax;
        this.orientationAngle = orientationAngle;
        if(orientationAngle!=0)
        {
            orientationAngle = Math.toRadians(orientationAngle);
            sinA = Math.sin(orientationAngle);
            cosA = Math.cos(orientationAngle);
        }
        this.channel = channel;
        if(channel)
        {
            this.startPoint = position.getX();
            this.endPoint = position.getX() + passDistance;
        }
        /**** Particle features ****/
        this.particleRadius = particleRadius;
        this.particleMass = particleMass;
        this.particleDensity = particleDensity;
        return;
    }

    /**
     * Updates the position of nozzle after time t (X=Vt+X0)
     *
     * @param time the time interval used to move the nozzle from its current position.
     */
    private void updatingPositon(double time)
    {
        position.addWith(VectorOperation.multiply(velocity,time));
    }

    /**
     * Calculates the initial position and velocity of a new particle.
     *
     * @param Vmax the maximum velocity of particles which is observed at the nozzle
```

```
     centerline.
 * @param V0 the initial velocity of the new particle which is filled up by this method
     .
 * @param X0 the initial position of the new particle which is filled up by this method
     .
 */
private void setParticleVectors(Vector3D V0, Vector3D X0)
{
    double x, y, z, a, size;
    double r = getWeibullRV((stand_off_dis/beta),2.0);
    double theta  = Math.PI*2*Math.random();
    double Rn = this.nozzleRadius;
    double r1 = (Rn/Rc)*r;
    r = r - r1;

    /********* Sets particle velocity *********/
    double velocity = getParticleVmax() * (getVConstant1() - (vConstant2*r/
        stand_off_dis));
    x = r*Math.sin(theta);
    y = r*Math.cos(theta);
    z = stand_off_dis;
    size = Math.sqrt(x*x+y*y+z*z);
    // Makes V0 equal to velocity
    a = velocity/size;
    x = a*x;
    y = a*y;
    z = a*z;
    // rotate velocity around x-axis
    V0.setPoint(x,y,z);
    if(orientationAngle!=0)
    {
      double tempY = y;
      y = y*cosA - z*sinA;
      z = tempY*sinA + z*cosA;
    }
    V0.setPoint(x,y,z);

    /********* Sets particle position *********/
    x = position.X + r1*Math.sin(theta);
    y = position.Y + r1*Math.cos(theta);
    z = position.Z;
    X0.setPoint(x,y,z);
    return;
}

/**
 * The time at which the last particle was launched.
 */
static double preTime = 0;

/**
 * Recalculate the position and velocity of a nozzle for the case of channel.
 */
private void updateNozzleVectors(){
    double time = SimulationManager.SystemTime - preTime;
    preTime = SimulationManager.SystemTime;

    this.position.movePoint(this.velocity,time);
    if(position.getX()>=this.endPoint || position.getX()<=this.startPoint)
    {
        this.velocity.multiplyBy(-1.0);
        System.out.println("Nozzle velocity: " + this.velocity + "     time: " + time +
            "     position: " + position);
    }
```

```
}

/**
 * Launches a new particle, sets it vectors, and updates nozzle vectors (for the case
     of the channel).
 *
 * @return a new partilce which was just launched and entered the system.
 */
public Particle lunchParticle()
{
    Particle P;
    Vector3D V0=null;
    Vector3D X0=null;
    mySimulation.Counters.numOfCurrParticles++;
    V0=new Vector3D();
    X0=new Vector3D();
    if(this.isChannel())
        updateNozzleVectors();
    setParticleVectors(V0,X0);
    P = new Particle(getParticleRadius(), getParticleMass(), X0, V0);
    if(orientationAngle!=0)
    {
        double dis = 2.0*(this.position.getY()-P.getPosition().getY());
        P.getPosition().setY(P.getPosition().getY()+dis);
        P.getLinearVelocity().setY(P.getLinearVelocity().getY()*-1.0);
    }
    mySimulation.Counters.notReusedSpaces++;
    return P;
}

/**
 * It creates a weibull random variable by using a uniform random variable using the
     following formula.
 *                W=landa*(-Ln(U))^(1/k)
 *
 * pdf. function of W is:
 *                f(W) = (k/landa) * (W/landa)^(k-1) * e^(-(W/landa)^k)
 *
 * @param landa one of the parameter of the weibull distribution.
 * @param k one of the parameter of the weibull distribution.
 * @return a weibull random variable.
 */
private double getWeibullRV(double landa, double k)
{
    // random() creates a Uniform RV.
    double W, U = Math.random();
    W = landa*(Math.sqrt(-Math.log(U)));
 return W;
}

public double getRadius(){
    return nozzleRadius;
}

public double getLunchFrequency(){
    return lunchFrequency;
}

public double getLunchingTimeInterval(){
    return lunchingTimeInterval;
}

public double getStand_off_dis(){
    return stand_off_dis;
```

```
    }

    public double getBeta(){
        return beta;
    }

    public Vector3D getPosition(){
        return position;
    }

    public Vector3D getVelocity(){
        return velocity;
    }

    public double getOrientationAngle() {
        return orientationAngle;
    }

    public double getPassDistance() {
        return passDistance;
    }

    public double getVConstant1() {
        return vConstant1;
    }

    public double getVConstant2() {
        return vConstant2;
    }

    public boolean isChannel() {
        return channel;
    }

    public double getParticleRadius() {
        return particleRadius;
    }

    public double getParticleMass() {
        return particleMass;
    }

    public double getParticleVmax() {
        return particleVmax;
    }

    public double getParticleDensity() {
        return particleDensity;
    }
}
```

**Particle.java:** The instances of the Particle class used to store the information of particles.

```
package mySimulation;

import Objects3D.Vector3D;

import Objects3D.VectorOperation;

import java.util.LinkedList;

import java.util.List;

public class Particle {
    /**
     * The radius of the particle.
     */
    private double Radius;

    /**
     * The mass of the particle.
     */
    private double Mass;

    /**
     * The unique id provided with particle.
     */
    private int ID;

    /**
     * For the particle which has left the system, this variable is changed to true.
     */
    private boolean deleted = false;

    /**
     * The time associated with the position of the particle.
     */
    private double particleTime;

    /**
     * The linear velocity of the particle.
     */
    private Vector3D linearVelocity;

    /**
     * The rotational velocity of the particle.
     */
    private Vector3D angularVelocity;

    /**
     * The position of the particle at time of its last handlled event.
     */
    private Vector3D position;

    /**
     * The destination of particle after after handling its next precited trasfer.
     */
    private Vector3D nextDestination;

    /**
```

```
 * The variable used to determine the particle with unusual behaviors.
 */
private boolean unusualSituation = false;

/**
 * The number of particle-surface collisions of this particle.
 */
private int numHitSurface;

/**
 * The number of particle-particle collisions of this particle.
 */
private int numPPColl;

/**
 * The last environment cell in which the particle enters.
 */
public SystemEnvironment.Cell currentSpace;

/**
 * The second last environment cell in which the particle enters.
 */
private SystemEnvironment.Cell oldSpace;

/**
 * The variavle that holdsthe  information of all detected events for this particle.
 */
private SystemEvents.Event event;

/**
 * The associated node with this particle which is stored in event queue.
 */
private SystemEvents.EventNode eventNode;

/**
 * The type of the last handled event of this particle.
 */
private SystemEvents.Event.EventType lastEvent;

/**
 * The coefficient restitution for particle-particle collisions.
 */
private static double epp;

/**
 * The coefficient restitution for particle-surface collisions.
 */
private static double eps;

private List containingCube;

/**
 * Creates a new instance of Particle
 *
 * @param Rp the radius of the particle.
 * @param Mp the mass of the particle.
 * @param X0 the initial position of the particle at the nozzle exit plane.
 * @param V0 the initial velocity of the particle at the nozzle exit plane.
 */
public Particle( double Rp, double Mp, Vector3D X0, Vector3D V0)
{
 setRadius(Rp);
 setMass(Mp);
 numHitSurface = 0;
```

```
numPPColl = 0;
position = new Vector3D(X0);
this.nextDestination = new Vector3D(-1,-1,-1);
linearVelocity = new Vector3D(V0);
// It is assumed that particles do not have angular velocity when they are lunched
    from the nozzle.
angularVelocity = new Vector3D(0,0,0);
event = new SystemEvents.Event(this);
particleTime = SimulationManager.SystemTime;
eventNode = new SystemEvents.EventNode(event);
oldSpace = null;
setContainingCube(new LinkedList());
return;
}

/**
 * Resets the information of the particle to default values.
 */
public void resetParticle(double Rp, double Mp)
{
 setRadius(Rp);
 setMass(Mp);
 numHitSurface = 0;
 numPPColl = 0;
 this.nextDestination = new Vector3D(-1,-1,-1);
 // It is assumed that particles do not have angular velocity when the lunched from the
     nozzle
 angularVelocity = new Vector3D(0,0,0);
 if(this.event.getOwnerParticle()==null)
    this.event.setOwnerParticle(this);
 event.resetInfo();
 particleTime = SimulationManager.SystemTime;
 oldSpace = null;
 return;
}

/**
 * Moves the particle for the given time interval.
 *
 * @param time a time interval.
 * @return the new position of the particle after the given time interval.
 */
public Vector3D updatingParticlePosition(double time){
  this.position.setPoint(VectorOperation.add(VectorOperation.multiply(linearVelocity,
      time),this.position));
  this.particleTime += time;
  return position;
}

/**
 * Calculates the moment of inertia of the particle.
 */
public double GetMomentOfInertia(){
return (2.0*this.Mass*this.Radius*this.Radius)/5.0;
}

/**
 *  Calculates the kinetic energy of the particle.
 */
public double GetKineticEnergy(){
    return ((Mass*this.linearVelocity.getSquareNorm() + this.GetMomentOfInertia()*
        angularVelocity.getSquareNorm())/2.0);
}
```

```
/**
 * Increases the number of particle-surface collisions by one. This method is invoked
     after each time that the particle hits the surface.
 */
public void increaseHitSurface(){
    this.numHitSurface++;
}

public double getNumHitSurface() {
    return numHitSurface;
}

public int getID() {
    return ID;
}

/**
 * Increases the number of particle-particle collisions by one. This method is invoked
     after each time that the particle undergoes a new particle-particle collision.
 */
public void increaseNumPPColl(){
    this.numPPColl++;
}

public double getNumPPColl() {
    return numPPColl;
}

public double getParticleTime(){
    return particleTime;
}

public void setParticleTime(){
    particleTime = SimulationManager.SystemTime;
}

public void setParticleTime(double t){
    particleTime = t;
}

public void setID(int i) {
    ID = i;
}

public void setEvent(SystemEvents.Event e){
    event = e;
}

public SystemEvents.Event getEvent(){
    return event;
}

 public void setEventNode(SystemEvents.EventNode node){
    eventNode = node;
 }

public SystemEvents.EventNode getEventNode(){
    return eventNode;
}

public double getRadius(){
    return Radius;
}
```

```
public double getMass(){
    return Mass;
}

public Vector3D getLinearVelocity(){
    return linearVelocity;
}

public Vector3D getPosition(){
    return position;
}

public double getX(){
    return position.getX();
}

public double getY(){
    return position.getY();
}

public double getZ(){
    return position.getZ();
}

public void setPosition(double x, double y, double z){
    position.setPoint(x, y, z);
}

public void setPosition(Vector3D P){
    position = P;
}

public void setLinearVelocity(Vector3D v){
    linearVelocity = v;
}

public void setLinearVelocity(double x, double y, double z){
    linearVelocity.setPoint(x, y, z);
}

public void setCurrentSpace(SystemEnvironment.Cell c){
    currentSpace = c;
}

public SystemEnvironment.Cell getCurrentSpace(){
    return currentSpace;
}

public SystemEnvironment.Cell getOldSpace() {
    return oldSpace;
}

public void setOldSpace(SystemEnvironment.Cell oldSpace) {
    this.oldSpace = oldSpace;
}

public Vector3D getNextDestination() {
    return nextDestination;
}

public void setNextDestination(Vector3D nextDestination) {
    this.nextDestination = nextDestination;
}
```

```
public Vector3D getAngularVelocity() {
    return angularVelocity;
}

public void setAngularVelocity(double x, double y, double z) {
    this.angularVelocity.setPoint(x,y,z);
}

public void setAngularVelocity(Vector3D angularVelocity){
    this.angularVelocity = angularVelocity;
}

public boolean isUnusualSituation() {
    return unusualSituation;
}

public void setUnusualSituation(boolean unusualSituation, String reason) {
    this.unusualSituation = unusualSituation;
}

public SystemEvents.Event.EventType getLastEvent() {
    return lastEvent;
}

public void setLastEvent(SystemEvents.Event.EventType lastEvent) {
    this.lastEvent = lastEvent;
}

public void setRadius(double Radius) {
    this.Radius = Radius;
}

public void setMass(double Mass) {
    this.Mass = Mass;
}

/**
 * Returns a string representation of the ranges occupied by the particle in three axes
     .
 *
 * @return the string representing the range of the particle.
 */
public String getRange(){
   return ("\n   x: " + (position.getX()-this.Radius)+"-"+(position.getX()+this.Radius)+
           "\n   y: " + (position.getY()-this.Radius)+"-"+(position.getY()+this.Radius)+
           "\n   z: " + (position.getZ()-this.Radius)+"-"+(position.getZ()+this.Radius)+
                "\n");
}

/**
 * Returns a string representation of this particle.
 *
 * @return a string representation of this particle.
 */
public String toString(){
    return("Particle Info: \n   Position: " + this.position + "\n   Velocity: " +
           this.linearVelocity + "\n ID: " + this.getID()+"\n range: " + this.getRange
               ()+
           "\n   currentSpace: " + this.currentSpace + "\n   oldSpace: " + this.oldSpace
               );
}

public static double get_epp() {
    return epp;
```

```
    }

    public static void set_epp(double e) {
        epp = e;
    }

    public static double get_eps() {
        return eps;
    }

    public static void set_eps(double e) {
        eps = e;
    }

    public boolean isDeleted() {
        return deleted;
    }

    public void setDeleted(boolean deleted) {
        this.deleted = deleted;
    }

    public List getContainingCube() {
        return containingCube;
    }

    public void setContainingCube(List containingCube) {
        this.containingCube = containingCube;
    }
}
```

**SimulationManager.java:** The SimulationManager class implements the general event-based algorithm of the simulation. It chooses the easrliest event, the one stored in the root of the event heap. It also invokes appropriate haldler at each event to update the system and invokes appropriate detectors to update the event queue after each event handled.

```
package mySimulation;
import OcTreeADT.CellularOcTree;
import ProjectInterface.Interface;
import SystemEnvironment.Cell;
import Visualization.Plot;
import Objects3D.Vector3D;
import SystemEnvironment.Environment;
import java.awt.EventQueue;
import java.awt.event.ActionEvent;
import java.lang.Math;
import SystemEvents.*;
import SystemEvents.Event.*;
import java.rmi.RemoteException;
import java.util.*;
import javax.swing.JProgressBar;
import org.apache.poi.poifs.filesystem.*;
import org.apache.poi.hssf.usermodel.*;
import java.io.*;
import ProjectInterface.Interface;
import javax.swing.JOptionPane;
import javax.swing.SwingUtilities;
import javax.swing.JFrame;
import SystemEvents.Event.EventType;
import visad.VisADException;
public class SimulationManager extends ProjectInterface.Interface {
    /**
```

```
 * The nozzle of the AJM process.
 */
public static Nozzle AJM_Nozzle;

/**
 * The environment of AJM process holding all particles in the system.
 */
public static Environment AJM_Environment;

/**
 * The event queue of the simulation.
 */
public static EventHeap AJM_EventHeap;

/**
 * The substrate which is used in the AJM process and is exposed to the particles
     stream.
 */
public static CellularOcTree AJM_Substrate;

/**
 * An auxiliary variable used to determine the particle close to the surface cells.
 */
public static double precision = (double)0.0000001;

/**
 * The actual time of the simulation.
 */
public static double SystemTime;

/**
 * The time interval at which the results of the simulation are created.
 */
private static double printingTime;

/**
 * The time at which the results of the simulation are created. The time interval,
     printingTime, is used to update this variable.
 */
static public double timeToWrite;

/**
 * The progress of the simulation in terms of time which is used to update the progress
      bar of GUI.
 */
private int pBarPercent;

/**
 * The workbook used to create excel files including partially eroded depths of the
     surface cells at cross-sections.
 */
public static HSSFWorkbook cell2DWbY=null;

/**
 * The worksheet used to create excel files including partially eroded depths of the
     surface cells at cross-sections.
 */
public static HSSFSheet cell2DSheetY;

/**
 * A row of the spreadsheet used to create excel files including partially eroded
     depths of the surface cells at cross-sections.
 */
public static HSSFRow cell2DRowY;
```

```
/**
 * The workbook used to create excel files including the partially eroded depths of all
     the surface cells.
 */
public static HSSFWorkbook cell3DWb=null;

/**
 * The worksheet used to create excel files including the partially eroded depths of
     all the surface cells.
 */
public static HSSFSheet cell3DSheet;

/**
 * A row of the spreadsheet used to create excel files including the partially eroded
     depths of all the surface cells.
 */
public static HSSFRow cell3DRow;

/**
 * The workbook used to create an excel files including information from the simulation
     such as number of particel-particle
 * collisions, particle-surface collision, and etc.
 */
public static HSSFWorkbook wb;

/**
 * The worksheet used to create an excel files including information from the
     simulation such as number of particel-particle
 * collisions, particle-surface collision, and etc.
 */
private static HSSFSheet sheet;

/**
 * A row of the spreadsheet used to create an excel files including information from
     the simulation such as number of particel-particle
 * collisions, particle-surface collision, and etc.
 */
private static HSSFRow row;

/**
 * The counter to keep track of the number of created row in the worksheet including
     information from the simulation.
 */
private static int excelRow=1;

/**
 * Initializes the fields of this SimulationManager.
 */
public void initElements()
{
    /** Nozzle Information **/
    double nozzleRadius;
    double nozzlelunchFreq;
    double nozzleHeight;
    double weibullBeta;
    double nozzlePassDistance;
    Vector3D nozzlePosition, nozzleVelocity;
    boolean channel;
    double vContant1;
    double vContant2;
    double orientationAngle;

    /** Particles Information **/
```

```
double Vmax;
double particleRadius;
double particleMass;
double particleDensity;
double epp, eps;

/** Surface Information **/
double surfaceDensity;
double substrateDepth;
double D;
double K;
boolean brittle;
double n1, n2, Hv;
double frictionCoefficient;

/** Environment Information **/
double envEdgeSize;
double envDepth;
timebeg = System.nanoTime();

nozzlePosition = new Vector3D(Double.parseDouble(nXText.getText()),
                              Double.parseDouble(nYText.getText()),
                              Double.parseDouble(nZText.getText()));

nozzleVelocity  = new Vector3D(Double.parseDouble(nVXText.getText()),
                               0.0,
                               0.0);

nozzleRadius = Double.parseDouble(nRadiusText.getText());
nozzlelunchFreq = Double.parseDouble(nlunchFreqText.getText());
nozzleHeight = Double.parseDouble(nHeightText.getText());
weibullBeta = Double.parseDouble(betaText.getText());
nozzlePassDistance = Double.parseDouble(nPassDistanceText.getText());

if(nozzleVelocity.getX()==0 && nozzleVelocity.getY()==0 && nozzleVelocity.getZ()
    ==0)
    channel = false;
else
    channel = true;

Vmax = Double.parseDouble(pVSizeText.getText());
vContant1 = Double.parseDouble(this.vConst1.getText());
vContant2 = Double.parseDouble(this.vConst2.getText());
orientationAngle = Double.parseDouble(this.nOrientAngleText.getText());

/**
 * changing Radius from micron unit to mm
 */
particleRadius = Double.parseDouble(pRadiusText.getText())/1000.0;

/**
 * Kg/m^3
 */
particleDensity = Double.parseDouble(this.pDensityText.getText());

/** M=Desity*V  (V=4/3*PI*R^3)
/**
 * changing Radius unit to m
 */
double R = particleRadius/1000.0;
double pV = (4.0/3.0)*Math.PI*Math.pow(R,3.0);
particleMass = particleDensity*pV;

epp = Double.parseDouble(PPCo_ResText.getText());
```

```
System.out.println(epp);
Particle.set_epp(epp);

eps = Double.parseDouble(PSCo_ResText.getText());
System.out.println(eps);
Particle.set_eps(eps);

frictionCoefficient = Double.parseDouble(this.fricCoText.getText());
surfaceDensity = Double.parseDouble(surfDensityText.getText());
substrateDepth = Double.parseDouble(surfDepthText.getText());
D = Double.parseDouble(surfDText.getText());
K = Double.parseDouble(surfKText.getText());
brittle = this.brittleBool.isSelected();
n1 = Double.parseDouble(this.surfN1Const.getText());
n2 = Double.parseDouble(this.surfN2Const.getText());
Hv = Double.parseDouble(this.surfHv.getText());
envEdgeSize = Double.parseDouble(envWidthText.getText());
envDepth = nozzleHeight;

/**
 * micron to mm
 */
double surfCellSize = (Double.parseDouble(cellSizeText.getText())/1000);
OcTreeADT.CellularOcTree.setCellSize(surfCellSize);

AJM_Substrate = new CellularOcTree(calculateOctreeSize(envEdgeSize), envDepth,
    surfaceDensity, frictionCoefficient,
        D, K, brittle, n1, n2, Hv, substrateDepth);

System.out.println("brittle: " + brittle);
System.out.println("ductile: " + this.ductileBool.isSelected());

stopTime = Double.parseDouble(this.timeTxt.getText());

AJM_Nozzle = new Nozzle( nozzleRadius, nozzlelunchFreq, nozzleHeight, weibullBeta,
        nozzlePosition, nozzleVelocity, particleRadius, particleMass,
            particleDensity,
        nozzlePassDistance, channel, vContant1, vContant2, Vmax, orientationAngle);

AJM_Environment = new Environment(envEdgeSize, envEdgeSize, envDepth,0.8);
setPrintingBounds();
AJM_EventHeap = new EventHeap();

/**** For writing in Excel ****/
cell2DWbY = new HSSFWorkbook();
cell2DSheetY = cell2DWbY.createSheet();
cell3DWb = new HSSFWorkbook();
cell3DSheet = cell3DWb.createSheet();
wb = new HSSFWorkbook();
sheet = wb.createSheet();

row = sheet.createRow((short)0);
row.createCell((short)0).setCellValue("Time");
row.createCell((short)1).setCellValue("Launched Particles");
row.createCell((short)2).setCellValue("PP-Collisions");
row.createCell((short)3).setCellValue("PS-Collisions");
row.createCell((short)4).setCellValue("Particles Hit the Surface Before Colliding")
    ;
row.createCell((short)5).setCellValue("Depth of Erosion Profile");

File f1 = new File("mySimResult");
if(!f1.exists())f1.mkdir();

SystemTime=0;
```

```
        printingTime = Double.parseDouble(this.sysPrintTime.getText());
        timeToWrite = printingTime;
        pBarPercent = 0;
        EventPSCollision.initPoint();
    }

    /**
     * Calculates the appropriate edge for the substrate.
     * <p>
     * Note: Simulation assumes that the substrate is a cube. Since the present simulation
        used the
     * cellular octree, the logarithm of the number of cells in a row to the base 2 must be
         a positive integer.
     * If the user input for the size of the surface does not preserve this condition, the
        calculateOctreeSize
     * method calculates the smallest number to the user input which is larger than the
        input and preserves the
     * condition.
     *
     * @param envWidth the edge of the cubic target substrate.
     *
     * @return the appropriate size for the substrate edge.
     */
    public double calculateOctreeSize(double envWidth){
        double envEdgeSize = envWidth;
        double preSize = envEdgeSize;
        double numCells = envEdgeSize/OcTreeADT.CellularOcTree.getCellSize();
        double log2 = Math.log(numCells)/Math.log(2.0);

        if(log2!=Math.floor(log2))
        {
            double correctedLog2 = Math.floor(log2 + 0.4);
            envEdgeSize = Math.pow(2.0,correctedLog2)*OcTreeADT.CellularOcTree.getCellSize
                ();
            if(envEdgeSize<preSize)
            {
                correctedLog2 = Math.floor(log2 + 1.4);
                envEdgeSize = Math.pow(2.0,correctedLog2)*OcTreeADT.CellularOcTree.
                    getCellSize();
            }
        }
        return envEdgeSize;
    }

    /**
     * The current event of the system.
     */
    static public Event.EventType currEventType=null;

    /**
     * The element of the event queue including the earliest event.
     */
    static public EventNode currElement;

    /**
     * The id of a particle attended in the last particle-particle collision of the system.
     */
    static public int collParticleId;

    /**
     * The id of a particle attended in the last particle-particle collision of the system.
     */
    static public int collPartnerId;
```

```
/**
 * Starts the simulation of creating the system components and invoking appropriate
     handlers and detectors.
 */
public void runSimulation() throws InterruptedException
{
    /** Creates components and initializes them. **/
    initElements();

    /**Sets the value for calculating the volume removed from the surface. **/
    SystemEvents.EventPSCollision.setErosionConstant();

    long time;
    long startTime = System.nanoTime();

    /** Creates a text file including the inputs to the simulation. **/
    createInputsFile();

    /** the type of the previous event that has been occurred. **/
    Event.EventType prevEventType;

    /** Sets the initial time of the simulation. **/
    this.SystemTime = 0;

    Cell origin, destination;
    int pBarOldPercent = 0;
    int testNumHitSurf = 0;

    /** Creates the launch node of the event queue. **/
    currElement = new EventNode(null);

    /** The particle involving the the current event of the system. **/
    Particle P = null;

    /** The partner of the particle, P, before its current event was handled. **/
    Particle prePartner;

    /** Counters to monitor the time for printing information on the sceen. **/
    int limitPrintScreen = 500;
    int limitPrintTime = 500;

    /**Adds the launch node to the heap. Note: At this point, event queue has only one
        element. **/
    this.AJM_EventHeap.insert(currElement);

    /** A counter monitoring the number of handled events in the system. **/
    int i=0;

    while(true)
    {
        /** Updates the integer determining the progress of the simulation. **/
        pBarPercent = ((int)(this.SystemTime*100/this.stopTime));

        /** Checks if the time is up. If so the execution is terminated. **/
        if(SystemTime>=this.stopTime)
        {
            System.out.println("time to stop: " + this.SystemTime);
            printStatistics();
            /** Fully fills up the progress bar **/
            SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    updateBar(pBar.getMaximum());
                }
            });
```

```
    break;
}

/** Checks if the progress bar must be updated. **/
if( pBarPercent > pBarOldPercent)
{
    SwingUtilities.invokeLater(new Runnable() {
        public void run() {
            updateBar(pBarPercent);
        }
    });
}

pBarOldPercent = pBarPercent;
prevEventType = currEventType;

/** Sets the currElement, which is current event, to root of the minimized
    event heap **/
currElement = AJM_EventHeap.getHeapRoot();

if(currElement==null)
    System.out.println("currEvent:" + currEventType );

/** Sets the type of the current cube **/
currEventType = currElement.getNextEventType();

/** Sets the last handled event for the particle involving in the current event
    **/
if(currElement.getEvent()!=null)
    currElement.getEvent().getOwnerParticle().setLastEvent(currEventType);

/** Reports error **/
if(currElement.getTime()==0)
    Objects3D.VectorOperation.errMassageExit("currElement.getTimeValue()==0","
        SysImpelementsation");

/**
 * The switch statement divided the algorithm into branches based on a number
    of events type.
 * Each branch is resposible to handle the certain type of events. It also
    updates the event queue
 * after the event was handled.
 */
switch(currEventType)
{
    /** The branch resposible for particle-particle collisions **/
    case PPCollisionType:
    {
        /** Increments the number of launched particles. **/
        mySimulation.Counters.numPPColl++;

        if(mySimulation.Counters.numPPColl==1)
            System.out.println("first PPColl at time: "+ SystemTime);
        P = currElement.getEvent().getOwnerParticle();
        Particle partner = P.getEvent().getPPCollision().getInvolvingParticle()
            ;
        collParticleId = P.getID();
        collPartnerId = partner.getID();

        /** Updates the time of the system to the time of the current Event.*
            */
        SystemTime = currElement.getTime();

        /** Handles the particle-particle collision event. **/
```

```
        EventPPCollision.handleEvent(P, partner, currElement.getTime());

        /** Updates the event heap **/
        P.getEvent().setUpEvent_After_PPCollision();

        break;
    }

    /** The branch resposible for particle-surface collisions. **/
    case PSCollisionType:
    {
        P = currElement.getEvent().getOwnerParticle();
        double preTime = currElement.getTime();
        prePartner = P.getEvent().getPPCollision().getInvolvingParticle();

         /** Updates the time of the system to the time of the current Event.*
             */
        SystemTime = preTime;

        /** Handles the particle-surface event. **/
        boolean result = EventPSCollision.handleEvent(P);
        if(prePartner!=null && result)
        {
            if(prePartner.getCurrentSpace()!=null)
                prePartner.getEvent().getPPCollision().setInfo(null,Double.
                    POSITIVE_INFINITY);
            else
                Objects3D.VectorOperation.errMassage("prePartner.getEvent()==
                    null","in_PSCollisionType_case");
        }
        /** If the partcile-surface collision was successful. **/
        if(result)
        {
            /** Updates the event heap **/
            P.getEvent().setUpEvent_After_PSCollision();
            if(prePartner != null)
            {
                if(prePartner.getCurrentSpace()!=null)
                    prePartner.getEvent().setUpEvent_After_PartnerLeaving();
                else
                    SystemEvents.EventTransfer.removeParticle(prePartner);
            }
        }
        /** If the particle-surface cllision was unsuccessful. **/
        else
        {
            if(prePartner != null)
                prePartner.getEvent().setUpEvent_After_PartnerLeaving();
            else if(!P.isUnusualSituation())
            {
                /** Increments the number of unsuccessful particle-surface
                    collision **/
                mySimulation.Counters.numUnsuccessPSColl++;
                if(!P.isUnusualSituation())
                {
                    /** Updates the event heap **/
                    P.getEvent().setUpEvent_After_PSCollision();
                    if(P.getEvent().getPSCollision().getMinTime()==preTime)
                    {
                        P.setUnusualSituation(true,"handleUnsuccPScollision-
                            NullPointerException");
                        EventTransfer.removeParticle(P);
                        SystemTime = currElement.getTime();
                    }
```

```
                }
            }
        }
        break;
    }

    /** The branch resposible for transfer events. **/
    case transferType:
    {
        /** Increments the number of trasfers **/
        mySimulation.Counters.numTransfer++;
        testNumHitSurf = Counters.numPSColl;
        P = currElement.getEvent().getOwnerParticle();

        prePartner = P.getEvent().getPPCollision().getInvolvingParticle();
        origin = P.getCurrentSpace();
        /** Handles the transfer event. **/
        EventTransfer.handelTransfer(P);

        /**
         * The last environment cell that the particle has entered. For the
             cases that particle has  left the system
         * this variable is set to null.
         */
        destination = P.getCurrentSpace();

        /**
         * After updating the events of P, P it could be different from "
             currElement.getEvent().getOwnerParticle()",
         * the reason is after set up its event we have to update its node
             position in the heap. In updating heap,
         * we do not really change the node posion after swapping the node, we
             just change the CONTENT of the nodes.
         * so the "currElement.getEvent().getOwnerParticle()" is the old place
             for "P" node, but now after updating
         * its node is changed for the new place and the reference to this new
             place is put inside P->eventNode.
         **/

        if(prePartner != null)
        {
            if(prePartner.getCurrentSpace()!=null)
                prePartner.getEvent().setUpEvent_After_PartnerLeaving();
        }
        /** If particle is still flowing inside the system. **/
        if(destination!=null)
        {
            P.getEvent().setUpEvent_After_Transfer();
            if(P.getEventNode()!=null && P.getEventNode().getTime()==Double.
                POSITIVE_INFINITY)
                Objects3D.VectorOperation.errMassageExit("P.getEventNode().
                    getTimeValue()==Double.POSITIVE_INFINITY",
                        "SystemImp/runSimulation");
        }
        break;
    }

    /** The branch resposible for lauchnig new particles. **/
    case launchType:
    {
        mySimulation.Counters.numLunch++;

        /** Updates the time of the system to the time of the current Event.**/
        SystemTime = currElement.getTime();
```

```
            /** Updates the time of the next launching event in the system. **/
            currElement.setNextEventNodeInfo_LunchingParticle();

            /** Handles the launch evet == Launches a new particle **/
            P = AJM_Nozzle.lunchParticle();
            P.setID(mySimulation.Counters.numLunch);
            P.setLastEvent(EventType.launchType);

            while(AJM_Environment.addNewParticle(P)==null)
            {
                P = AJM_Nozzle.lunchParticle();
                P.setID(mySimulation.Counters.numLunch);
            }

            /** Updates the event queue **/
            P.getEvent().setUpEvent_After_Lunching();
            break;
        }

        /** Catches and Handles errors which are caused by the invalid value of the
             root **/
        case undefinedType:
        {
            Objects3D.VectorOperation.errMassageExit("case undefinedType of event"
                + "\nP: " + P +
                    "\nEvent: " + P.getEvent(),"SystemImplementation" +
                    "\nPreEvent: " + prevEventType);
            P.getEventNode().resetNextEventNodeInfo();
            break;
        }
    }
    i++;

    if(i==limitPrintTime)
    {
        //System.out.println("-----> "+ this.SystemTime);
        /** Updates the system clock **/
        this.sysTimeText.setText(String.valueOf(SystemTime));
        limitPrintTime +=900000;
    }

    if(i==limitPrintScreen)
    {
        System.out.println("------> "+ this.SystemTime);
        printStatistics();
        limitPrintScreen +=25000000;
    }
} // end of while

printStatistics();
System.out.println("number of removed cells: "+mySimulation.Counters.
    numOfRemovedCells);
time = System.nanoTime() - startTime;
System.out.println("time: " + time);
System.exit(0);


/** This part is implemented to visualize the erosion profile in three-dimensions
    using the VisAD package. **/
if(false)
{
    depthScale = Double.parseDouble(this.surfDepthScaleText.getText())*-1;
    try {
```

```
            new Plot(depthScale);
        } catch (RemoteException ex) {
            ex.printStackTrace();
        } catch (VisADException ex) {
            ex.printStackTrace();
        }
        JOptionPane.showMessageDialog(this, "Plotting the Erosion Profile");
    }
    return;
}

/**
 * Creates a text file including the inputs to the simulation.
 */
private static void createInputsFile()
{
    try
    {
        String outputFileName = "mySimResult/Inputs.txt";
        FileWriter outputFileReader  = new FileWriter(outputFileName);
        PrintWriter     outputStream  = new PrintWriter(outputFileReader);
        outputStream.println("+----------Inputs to the Simulation----------+" +
                "\n\nNozzle:" +
                "\n      radius (mm): " + AJM_Nozzle.getRadius() +
                "\n      stand-off-distance (mm): " + AJM_Nozzle.getStand_off_dis() +
                "\n      velocity (mm/s): " + AJM_Nozzle.getVelocity() +
                "\n      initial position (mm): " + AJM_Nozzle.getPosition() +
                "\n      launch frequency: " + AJM_Nozzle.getLunchFrequency() +
                "\n      weibull beta: " + AJM_Nozzle.getBeta() +
                "\n      orientation angle: " + AJM_Nozzle.getOrientationAngle());
         if(AJM_Nozzle.getVelocity().getX()!=0)
             outputStream.println("      channel pass (mm): " + AJM_Nozzle.
                 getPassDistance());
              outputStream.println("\n\nParticle:" +
                      "\n      radius (mu): " + (AJM_Nozzle.getParticleRadius()*1000) +
                      "\n      density (kg/m^3): " + AJM_Nozzle.getParticleDensity() +
                      "\n      maximum velocity (m/s): " +  (AJM_Nozzle.getParticleVmax()
                          /1000) +
                      "\n      constants for velocity distribution: " +
                      "\n            v1: " + AJM_Nozzle.getVConstant1() + "        v2: "
                          + AJM_Nozzle.getVConstant2() +
                      "\n\n\nSurface: " );
         if(AJM_Substrate.isBrittle())
             outputStream.println("      erosive system: brittle");
         else
             outputStream.println("      erosive system: ductile" +
                     "\n      erosion rate constants: " +
                     "\n            n1: " + AJM_Substrate.getN1() +
                     "       n2: " + AJM_Substrate.getN2() +
                     "       Hv: " + AJM_Substrate.getHv());
         outputStream.println("      density (kg/m^3): " + AJM_Substrate.getSurfaceDensity
             () +
                      "\n      constant D: " + AJM_Substrate.getD() +
                      "\n      constant K: " + AJM_Substrate.getK() +
                      "\n      edge of surface cells (mu): " + (OcTreeADT.CellularOcTree.
                          getCellSize()*1000) +
                 "\n\n\nCeofficients: " +
                 "\n      epp: " + Particle.get_epp() +
                 "\n      eps: " + Particle.get_eps() +
                 "\n      friction: " + AJM_Substrate.getFriction() +
                 "\n\n+-----------------------------------------------------+");
         outputStream.close();
     } catch (IOException e) {
         System.out.println("IOException:");
```

```
    }
}

/**
 * Prints the values of counters defined in the simulation.
 */
public static void printStatistics(){
    System.out.println( "\nPP-Collisions: " + Counters.numPPColl +
                        "\nLunched Particles: " + Counters.numLunch +
                        "\nTransfers: " + Counters.numTransfer +
                        "\nLeft Environment: " + Counters.numExitEnv1 +
                        "\noverlapCounting: "+ Counters.overlapCounting +
                        "\nunsuccessful PScollisions: "+ Counters.numUnsuccessPSColl +
                        "\nfound PSColls for embbeded particle case: " + Counters.
                            foundPSCollEmbed +
                        "\nnumber of removed cells: " + Counters.numOfRemovedCells +
                        "\nunusual situation: " + Counters.unusual +
                        "\nmaxDepthEroded: " + EventPSCollision.maxDepth +
                        "\nnozzle position: " + AJM_Nozzle.getPosition() +
                        "\n----------\nTotal MembersOfCubes: " + AJM_Environment.
                            getTotalMembersOfCubes() +
                        "\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^" +
                        "\nHitSurfBeforePPColl:" + Counters.numHitSurfBeforePPColl +
                        "\nPS-Collisions: "+ Counters.numPSColl +
                        "\nnumPlaneNormal: " + Counters.numPlaneNormal +
                        "\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^" +
                        "\nnum of curve fit calculation: " + Counters.numOfPloyCalc +
                        "\nnum of countPoint: " + countPoints +
                        "\n^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^\n" +
                        "\nwastedEnergy: " + EventPSCollision.wastedEnergy +
                        "\nAverage of wastedEnergy: " + (EventPSCollision.wastedEnergy/
                            mySimulation.Counters.numOfRemovedCells) +
                        "\nNum of overlap at launch: " + mySimulation.Counters.
                            overAtLaunch);
    System.out.println("\n\n\n----------");

    /**
     * Prints the cefficient of best curve fit euqaiotn.
     */
    for(int p=0;p<(SystemEvents.EventPSCollision.degreeOfPoly+1);p++)
        System.out.println(SystemEvents.EventPSCollision.parameters[p] );

    System.out.println("\n\nreused spaces: "+Counters.reusedSpaces +
            "\nnotReusedSpaces: "+Counters.notReusedSpaces );
}

/**
 * The variables that define the printing bounds.
 */
public static double bound1Y, bound2Y, bound1X, bound2X;

/**
 * Sets the variables that define the printing bounds.
 */
public static void setPrintingBounds()
{
    double  X = AJM_Nozzle.getPosition().getX();
    double  Y = AJM_Nozzle.getPosition().getY();

    if(AJM_Nozzle.isChannel())
    {
        bound1X = X + AJM_Nozzle.getPassDistance()/2.0 - 3.0;
        bound2X = X + AJM_Nozzle.getPassDistance()/2.0 + 3.0;
    }
```

```
        else
        {
            bound1X = X - (OcTreeADT.CellularOcTree.getCellSize()*10);
            bound2X = X + (OcTreeADT.CellularOcTree.getCellSize()*10);
        }

        if(AJM_Nozzle.getOrientationAngle()!=0)
        {
            bound1Y = 0;
            bound2Y = AJM_Environment.getWidth();
        }
        else
        {
            bound1Y = Y - 3.5;
            bound2Y = Y + 3.5;
        }
    }

    /**
     * Number of changes made to the array storing the the depth of the surface cells at
         the cross-section.
     */
    public static int countPoints = 0;

    /**
     * The partially eroded depth of the deepest cell.
     */
    private static double maxDepth=Double.POSITIVE_INFINITY;

    /**
     * Updates the worksheet storing the depth of the surface cells.
     *
     * @param P the position of a cell.
     * @param z the partially eroded depth of a cell.
     */
    public static void WriteCellInExcel3D(Vector3D P, double z)
    {
        int x = (int)((P.getX())/OcTreeADT.CellularOcTree.getCellSize());
        int y = (int)((P.getY())/OcTreeADT.CellularOcTree.getCellSize());

        int skip = 8;
        if(x%skip==0 && y%skip==0)
        {
            x = x/skip;
            y = y/skip;
            cell3DRow = cell3DSheet.getRow(x);
            if(cell3DRow!=null)
            {
                if(cell3DRow.getCell((short)y)==null)
                    cell3DRow.createCell((short)y).setCellValue(z);
                else
                    cell3DRow.getCell((short)y).setCellValue(z);
            }
            else
            {
                cell3DRow = cell3DSheet.createRow((short)x);
                cell3DRow.createCell((short)y).setCellValue(z);
            }
        }
    }

    /**
     * Updates the worksheet storing the depth of the surface cells at the cross-section.
     *
```

```
 * @param P the position of a cell.
 * @param L the partially eroded depth of a cell. For the cases that cell has been
     removed L equals 0.
 */
static long time1=0;
static long timebeg=0;
public static void writeBoundaryCellInExcel2DY(Vector3D P, double L){
    double Z;
    double h = AJM_Nozzle.getStand_off_dis();

    /** if cell is partially eroded **/
    if(L!=0)
    {
        Z = (-P.getZ()+OcTreeADT.CellularOcTree.getMinHalfSize()+h)-L;

    }
    else
    {
       Z = (P.getZ()+OcTreeADT.CellularOcTree.getMinHalfSize()-h)*-1;

    }


    int i;
    if(AJM_Nozzle.getVelocity().getX()==0 && L==0)
    {
        countPoints++;
        i = (int)(P.getY()/OcTreeADT.CellularOcTree.getCellSize());
        if(Math.abs(Z)>EventPSCollision.point[i][1])
        {
            EventPSCollision.point[i][0] = P.getY();
            EventPSCollision.point[i][1] = Math.abs(Z);
        }
    }

    /** Update the worksheep for 3D plotting. **/
    WriteCellInExcel3D(P,Z);

    if(!(P.getY()>=bound1Y && P.getY()<=bound2Y && P.getX()>=bound1X && P.getX()<=
        bound2X))
        return;

    double Y = P.getY()-bound1Y;
    int row = (int)(Y/OcTreeADT.CellularOcTree.getCellSize());

    if(Z<maxDepth)
        maxDepth=Z;

    if(cell2DSheetY.getRow(row)!=null)
    {
        double val = cell2DSheetY.getRow(row).getCell((short)1).getNumericCellValue();
        double currLength = Math.floor(Math.abs(val)/OcTreeADT.CellularOcTree.
            getCellSize())*OcTreeADT.CellularOcTree.getCellSize();

        if(Z<val)
        {
            if(L==0)
            {
                cell2DSheetY.getRow(row).getCell((short)1).setCellValue(Z);
            }
             else
             {
                cell2DSheetY.getRow(row).getCell((short)1).setCellValue(Z);
             }
```

```
            }
        }
        else
        {
            cell2DRowY = cell2DSheetY.createRow((short)row);
            if(AJM_Nozzle.getOrientationAngle()!=0)
                cell2DRowY.createCell((short)0).setCellValue(Y);
            else
                cell2DRowY.createCell((short)0).setCellValue(Y-(bound2Y-bound1Y)/2.0);
            if(L==0)
            {
                cell2DRowY.createCell((short)1).setCellValue(Z);
            }
            else
            {
                cell2DRowY.createCell((short)1).setCellValue(Z);
            }
        }
        /** If it is time to print results, creates the excel files. **/
        if(SystemTime>=timeToWrite)
        {
            try
             {
                System.out.println("\n\n\nAt time: " + SystemTime +
                        "\nDuration: " + (System.nanoTime()-timebeg) +
                        "\nInterval: " + (System.nanoTime()-time1));
                time1 = System.nanoTime();
                writeInExcel();
                System.out.println("\n\n\nprinting cross-section at TIME: " + SystemTime)
                    ;
                String sheetName = "mySimResult/CrossSection-Time" + timeToWrite + ".xls";
                FileOutputStream fileOut = new FileOutputStream(sheetName);
                cell2DWbY.write(fileOut);
                fileOut.close();
                fileOut=null;

                sheetName = "mySimResult/3DProfile" + timeToWrite + ".xls";
                fileOut = new FileOutputStream(sheetName);
                cell3DWb.write(fileOut);
                fileOut.close();
                fileOut=null;
             }catch(IOException e)
             {
                 System.err.println("error in entering data into boundary2D excel sheet");
             }
            timeToWrite += printingTime;
            printStatistics();
        }
    }

    /**
     * Add a row in the worksheet used to create the report from the simulation.
     */
    public static void writeInExcel(){
        row = sheet.createRow((short)excelRow);
        row.createCell((short)0).setCellValue(SystemTime);
        row.createCell((short)1).setCellValue(Counters.numLunch);
        row.createCell((short)2).setCellValue(Counters.numPPColl);
        row.createCell((short)3).setCellValue(Counters.numPSColl);
        row.createCell((short)4).setCellValue(Counters.numHitSurfBeforePPColl);
        row.createCell((short)5).setCellValue(maxDepth);
        excelRow++;
    }
```

```
    /**
     * The standard entry point into the program.
     */
    public static void main(String args[]){
       SimulationManager sys= new SimulationManager();
       sys.setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {}
}
```

**Interface.java:** The Interface class implements the graphical user interface (GUI) for the simulation. GUI enables user to enter the input parameters and monitor the progress and the actual time of the system.

```
package ProjectInterface;

import OcTreeADT.CellularOcTree;

import Objects3D.Vector3D;

import java.io.FileOutputStream;

import java.io.IOException;

import java.rmi.RemoteException;

import javax.swing.*;

import java.awt.*;

import java.awt.event.*;

import SystemEnvironment.*;

import SystemEvents.*;

import mySimulation.*;

import java.util.*;

import java.util.List;

import OcTreeADT.OctNode;

import Visualization.Plot;

import visad.VisADException;

abstract public class Interface extends javax.swing.JFrame implements
Runnable, ActionListener {
    /**
     * The time duration of the simulation which is considered as a criteria to stop the
         simulation.
     */
    public double stopTime=0;

    /**
     * The variable used to adjust the size of the graph plotted by the Plot class.
     */
    public static double depthScale=1;

    /**
     * The thread which starts the simulation run.
     */
    Thread thread;

    /**
     * Creates new form Interface
     */
```

```
public Interface()
{
    initComponents();
    this.nVYText.setEnabled(false);
    this.nVZText.setEnabled(false);
    pBar.setMinimum(0);
    pBar.setMaximum(100);
    pBar.setStringPainted(true);
    this.surfKText.setSize(10,10);
}

/**
 * Updates the progress of the simulation in the progress par.
 *
 * @param newValue the percentage of the simulation progress in terms of time.
 */
public void updateBar(int newValue) {
    pBar.setValue(newValue);
}

/**
 * This method is called from within the constructor to
 * initialize the form.
 */
// <editor-fold defaultstate="collapsed" desc=" Generated Code ">
private void initComponents() {
    jSeparator1 = new javax.swing.JSeparator();
    buttonGroup1 = new javax.swing.ButtonGroup();
    nozzleInfo = new javax.swing.JPanel();
    nRadius = new javax.swing.JLabel();
    nHeight = new javax.swing.JLabel();
    nlunchFreq = new javax.swing.JLabel();
    nPosition = new javax.swing.JLabel();
    nVelocity = new javax.swing.JLabel();
    nRadiusText = new javax.swing.JTextField();
    nHeightText = new javax.swing.JTextField();
    nVXText = new javax.swing.JTextField();
    nXText = new javax.swing.JTextField();
    nlunchFreqText = new javax.swing.JTextField();
    nYText = new javax.swing.JTextField();
    nZText = new javax.swing.JTextField();
    nVYText = new javax.swing.JTextField();
    nVZText = new javax.swing.JTextField();
    beta = new javax.swing.JLabel();
    betaText = new javax.swing.JTextField();
    pVSize = new javax.swing.JLabel();
    pVSizeText = new javax.swing.JTextField();
    nPassDistance = new javax.swing.JLabel();
    nPassDistanceText = new javax.swing.JTextField();
    jLabel2 = new javax.swing.JLabel();
    jLabel3 = new javax.swing.JLabel();
    vConst1 = new javax.swing.JTextField();
    jLabel4 = new javax.swing.JLabel();
    vConst2 = new javax.swing.JTextField();
    jLabel5 = new javax.swing.JLabel();
    jLabel6 = new javax.swing.JLabel();
    nOrientAngleText = new javax.swing.JTextField();
    jLabel9 = new javax.swing.JLabel();
    bottomPanel = new javax.swing.JPanel();
    runButton = new javax.swing.JButton();
    jButton1 = new javax.swing.JButton();
    plot = new javax.swing.JButton();
    sysTimeText = new javax.swing.JTextField();
    jLabel1 = new javax.swing.JLabel();
```

```
timeTxt = new javax.swing.JTextField();
jLabel7 = new javax.swing.JLabel();
jLabel13 = new javax.swing.JLabel();
sysPrintTime = new javax.swing.JTextField();
ParticlesInfo = new javax.swing.JPanel();
pRadius = new javax.swing.JLabel();
PPCo_Res = new javax.swing.JLabel();
pMass = new javax.swing.JLabel();
PSCo_Res = new javax.swing.JLabel();
pRadiusText = new javax.swing.JTextField();
pDensityText = new javax.swing.JTextField();
PPCo_ResText = new javax.swing.JTextField();
PSCo_ResText = new javax.swing.JTextField();
EnvironmentInfo = new javax.swing.JPanel();
envWidth = new javax.swing.JLabel();
envWidthText = new javax.swing.JTextField();
SurfaceInfo = new javax.swing.JPanel();
surfDensity = new javax.swing.JLabel();
surfK = new javax.swing.JLabel();
surfD = new javax.swing.JLabel();
surfDepth = new javax.swing.JLabel();
surfDensityText = new javax.swing.JTextField();
surfDepthText = new javax.swing.JTextField();
surfDText = new javax.swing.JTextField();
surfKText = new javax.swing.JTextField();
fricCoText = new javax.swing.JTextField();
fricCo = new javax.swing.JLabel();
surfDepthScale = new javax.swing.JLabel();
cellSize = new javax.swing.JLabel();
cellSizeText = new javax.swing.JTextField();
brittleBool = new javax.swing.JRadioButton();
ductileBool = new javax.swing.JRadioButton();
jLabel8 = new javax.swing.JLabel();
jLabel10 = new javax.swing.JLabel();
jLabel11 = new javax.swing.JLabel();
jLabel12 = new javax.swing.JLabel();
surfN1Const = new javax.swing.JTextField();
surfN2Const = new javax.swing.JTextField();
surfHv = new javax.swing.JTextField();
surfDepthScaleText = new javax.swing.JTextField();

setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
setTitle("Computer Simulation of Unmasked Profile");
setBackground(getBackground());
setCursor(new java.awt.Cursor(java.awt.Cursor.HAND_CURSOR));
setFont(new java.awt.Font("Tahoma", 0, 11));
setForeground(new java.awt.Color(51, 51, 51));
setResizable(false);
nozzleInfo.setBackground(EnvironmentInfo.getBackground());
nozzleInfo.setBorder(javax.swing.BorderFactory.createTitledBorder(null, "Nozzle
    Specification", javax.swing.border.TitledBorder.DEFAULT_JUSTIFICATION, javax.
    swing.border.TitledBorder.DEFAULT_POSITION, new java.awt.Font("Tahoma", 1, 12),
     new java.awt.Color(51, 51, 51)));
nozzleInfo.setFocusable(false);
nozzleInfo.setFont(getFont());
nozzleInfo.setPreferredSize(new java.awt.Dimension(0, 0));
nozzleInfo.setRequestFocusEnabled(false);
nozzleInfo.setVerifyInputWhenFocusTarget(false);
nRadius.setBackground(new java.awt.Color(0, 0, 0));
nRadius.setFont(getFont());
nRadius.setLabelFor(nRadius);
nRadius.setText("Radius(mm): ");
nRadius.setPreferredSize(new java.awt.Dimension(146, 16));
```

```
nHeight.setBackground(new java.awt.Color(0, 0, 0));
nHeight.setFont(getFont());
nHeight.setText("stand off distance(mm): ");
nHeight.setPreferredSize(new java.awt.Dimension(146, 16));

nlunchFreq.setBackground(new java.awt.Color(0, 0, 0));
nlunchFreq.setFont(getFont());
nlunchFreq.setText("Launch Frequency: ");
nlunchFreq.setPreferredSize(new java.awt.Dimension(146, 16));

nPosition.setBackground(new java.awt.Color(0, 0, 0));
nPosition.setFont(getFont());
nPosition.setText("Starting Position(mm): ");
nPosition.setPreferredSize(new java.awt.Dimension(146, 16));

nVelocity.setBackground(new java.awt.Color(0, 0, 0));
nVelocity.setFont(getFont());
nVelocity.setText("Velocity(mm/s): ");
nVelocity.setPreferredSize(new java.awt.Dimension(146, 16));

nRadiusText.setText("0.38");
nRadiusText.setPreferredSize(new java.awt.Dimension(100, 16));

nHeightText.setText("20");
nHeightText.setPreferredSize(new java.awt.Dimension(100, 16));

nVXText.setText("0");
nVXText.setPreferredSize(new java.awt.Dimension(10, 14));

nXText.setText("4");
nXText.setPreferredSize(new java.awt.Dimension(10, 14));

nlunchFreqText.setText("1.2e+6 ");
nlunchFreqText.setPreferredSize(new java.awt.Dimension(100, 16));

nYText.setText("4");
nYText.setPreferredSize(new java.awt.Dimension(10, 14));

nZText.setText("0");
nZText.setPreferredSize(new java.awt.Dimension(10, 14));

nVYText.setText("0");
nVYText.setPreferredSize(new java.awt.Dimension(10, 14));

nVZText.setText("0");
nVZText.setPreferredSize(new java.awt.Dimension(10, 14));

beta.setBackground(new java.awt.Color(0, 0, 0));
beta.setFont(getFont());
beta.setText("Weibull Beta:  ");
beta.setPreferredSize(new java.awt.Dimension(146, 16));

betaText.setText("15");
betaText.setPreferredSize(new java.awt.Dimension(100, 16));

pVSize.setBackground(new java.awt.Color(0, 0, 0));
pVSize.setFont(getFont());
pVSize.setText("Particle Max Velocity (mm/s):");
pVSize.setPreferredSize(new java.awt.Dimension(146, 16));

pVSizeText.setText("162000");
pVSizeText.setPreferredSize(new java.awt.Dimension(100, 16));

nPassDistance.setFont(getFont());
```

```
nPassDistance.setText("Pass Distance (mm): ");
nPassDistance.setPreferredSize(new java.awt.Dimension(146, 16));

nPassDistanceText.setText("7");
nPassDistanceText.setPreferredSize(new java.awt.Dimension(100, 16));

jLabel2.setFont(getFont());
jLabel2.setText("Velocity =");

jLabel3.setFont(getFont());
jLabel3.setText("Vmax  (");

vConst1.setText("1.0");

jLabel4.setFont(getFont());
jLabel4.setText(" -");

vConst2.setText("4.92");
vConst2.setPreferredSize(new java.awt.Dimension(40, 16));

jLabel5.setFont(getFont());
jLabel5.setText(" * (r/h))");

jLabel6.setFont(getFont());
jLabel6.setText("Orientation Angle: ");
jLabel6.setAlignmentX(getAlignmentX());
jLabel6.setPreferredSize(new java.awt.Dimension(146, 16));

nOrientAngleText.setText("0");
nOrientAngleText.setPreferredSize(new java.awt.Dimension(100, 16));

jLabel9.setFont(new java.awt.Font("Tahoma", 1, 11));
jLabel9.setText("Distribution of Initial Velocities of Particle: ");

org.jdesktop.layout.GroupLayout nozzleInfoLayout = new org.jdesktop.layout.
    GroupLayout(nozzleInfo);
nozzleInfo.setLayout(nozzleInfoLayout);
nozzleInfoLayout.setHorizontalGroup(
    nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
    .add(nozzleInfoLayout.createSequentialGroup()
        .addContainerGap()
        .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
            LEADING)
            .add(nozzleInfoLayout.createSequentialGroup()
                .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.
                    GroupLayout.LEADING)
                    .add(nozzleInfoLayout.createSequentialGroup()
                        .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.
                            layout.GroupLayout.LEADING)
                            .add(nRadius, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 146, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .add(nVelocity, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 146, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .add(nHeight, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 146, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .add(nPassDistance, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 146, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE))
                        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED))
                    .add(org.jdesktop.layout.GroupLayout.TRAILING, nozzleInfoLayout
                        .createSequentialGroup()
```

```
                .add(jLabel2, org.jdesktop.layout.GroupLayout.
                    PREFERRED_SIZE, 50, org.jdesktop.layout.GroupLayout.
                    PREFERRED_SIZE)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                .add(jLabel3)
                .add(4, 4, 4)))
            .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.
                GroupLayout.LEADING, false)
                .add(nozzleInfoLayout.createSequentialGroup()
                    .add(2, 2, 2)
                    .add(nRadiusText, org.jdesktop.layout.GroupLayout.
                        PREFERRED_SIZE, 100, org.jdesktop.layout.GroupLayout.
                        PREFERRED_SIZE))
                .add(nHeightText, org.jdesktop.layout.GroupLayout.
                    PREFERRED_SIZE, 100, org.jdesktop.layout.GroupLayout.
                    PREFERRED_SIZE)
                .add(nozzleInfoLayout.createSequentialGroup()
                    .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.
                        layout.GroupLayout.TRAILING)
                        .add(org.jdesktop.layout.GroupLayout.LEADING,
                            nozzleInfoLayout.createSequentialGroup()
                            .add(nVXText, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 34, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .addPreferredGap(org.jdesktop.layout.LayoutStyle.
                                RELATED)
                            .add(nVYText, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 28, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .addPreferredGap(org.jdesktop.layout.LayoutStyle.
                                RELATED)
                            .add(nVZText, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 26, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE))
                        .add(org.jdesktop.layout.GroupLayout.LEADING,
                            nPassDistanceText, org.jdesktop.layout.GroupLayout.
                            PREFERRED_SIZE, 100, org.jdesktop.layout.
                            GroupLayout.PREFERRED_SIZE)
                        .add(nozzleInfoLayout.createSequentialGroup()
                            .addPreferredGap(org.jdesktop.layout.LayoutStyle.
                                RELATED)
                            .add(vConst1, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 40, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .addPreferredGap(org.jdesktop.layout.LayoutStyle.
                                RELATED)
                            .add(jLabel4, org.jdesktop.layout.GroupLayout.
                                DEFAULT_SIZE, 12, Short.MAX_VALUE)
                            .addPreferredGap(org.jdesktop.layout.LayoutStyle.
                                RELATED)
                            .add(vConst2, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 40, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .addPreferredGap(org.jdesktop.layout.LayoutStyle.
                                RELATED)))
                    .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                    .add(jLabel5))))
        .add(jLabel9))
    .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED, 101, Short.
        MAX_VALUE)
    .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
        TRAILING)
        .add(beta, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146, org.
            jdesktop.layout.GroupLayout.PREFERRED_SIZE)
```

```
                .add(jLabel6, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(pVSize, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nlunchFreq, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nPosition, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
            .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
                LEADING)
                .add(betaText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 100, org
                    .jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(pVSizeText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 100,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nlunchFreqText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                    100, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nozzleInfoLayout.createSequentialGroup()
                    .add(nXText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 35,
                        org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                    .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                    .add(nYText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 29,
                        org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                    .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                    .add(nZText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 24,
                        org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
                .add(nOrientAngleText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                    100, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .addContainerGap())
    );
    nozzleInfoLayout.setVerticalGroup(
        nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
        .add(nozzleInfoLayout.createSequentialGroup()
            .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
                BASELINE)
                .add(nRadius, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nRadiusText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nXText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nYText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nZText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nPosition, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org
                    .jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .add(6, 6, 6)
            .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
                BASELINE)
                .add(nHeight, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nHeightText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nlunchFreqText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                    16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nlunchFreq, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
            .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
                BASELINE)
                .add(nVelocity, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org
                    .jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(nVXText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
```

```
                jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(nVYText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(pVSizeText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(nVZText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(pVSize, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE))
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
        .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
            BASELINE)
            .add(nPassDistance, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(nPassDistanceText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(betaText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(beta, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE))
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
        .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
            TRAILING)
            .add(nozzleInfoLayout.createSequentialGroup()
                .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.
                    GroupLayout.BASELINE)
                    .add(nOrientAngleText, org.jdesktop.layout.GroupLayout.
                        PREFERRED_SIZE, 16, org.jdesktop.layout.GroupLayout.
                        PREFERRED_SIZE)
                    .add(jLabel6, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                        16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
                .add(28, 28, 28))
            .add(nozzleInfoLayout.createSequentialGroup()
                .add(jLabel9)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                .add(nozzleInfoLayout.createParallelGroup(org.jdesktop.layout.
                    GroupLayout.BASELINE)
                    .add(jLabel5, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                        16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                    .add(jLabel3, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                        16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                    .add(jLabel2, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                        16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                    .add(vConst1, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                        16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                    .add(jLabel4, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                        16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                    .add(vConst2, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                        16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)))
        .addContainerGap(19, Short.MAX_VALUE))
);

nozzleInfoLayout.linkSize(new java.awt.Component[] {nHeight, nRadius}, org.jdesktop
    .layout.GroupLayout.VERTICAL);

runButton.setText("run");
runButton.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mouseClicked(java.awt.event.MouseEvent evt) {
        runHandler(evt);
    }
});

jButton1.setText("exit");
```

```
jButton1.addMouseListener(new java.awt.event.MouseAdapter() {
    public void mouseClicked(java.awt.event.MouseEvent evt) {
        exitHandler(evt);
    }
});

plot.setText("Plot");
plot.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(java.awt.event.ActionEvent evt) {
        plotHandler(evt);
    }
});

sysTimeText.setPreferredSize(new java.awt.Dimension(100, 16));

jLabel1.setFont(new java.awt.Font("Tahoma", 1, 11));
jLabel1.setText("System clock (s): ");

timeTxt.setText("1");
timeTxt.setDoubleBuffered(true);
timeTxt.setPreferredSize(new java.awt.Dimension(100, 16));

jLabel7.setFont(new java.awt.Font("Tahoma", 1, 11));
jLabel7.setText("Time Duration (s): ");

jLabel13.setFont(new java.awt.Font("Tahoma", 1, 11));
jLabel13.setText("Printing Time (s): ");

sysPrintTime.setText("1");
sysPrintTime.setDoubleBuffered(true);
sysPrintTime.setPreferredSize(new java.awt.Dimension(100, 16));

org.jdesktop.layout.GroupLayout bottomPanelLayout = new org.jdesktop.layout.
    GroupLayout(bottomPanel);
bottomPanel.setLayout(bottomPanelLayout);
bottomPanelLayout.setHorizontalGroup(
    bottomPanelLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
    .add(bottomPanelLayout.createSequentialGroup()
        .add(20, 20, 20)
        .add(bottomPanelLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
            LEADING)
            .add(bottomPanelLayout.createSequentialGroup()
                .add(runButton, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 75,
                     org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(24, 24, 24)
                .add(jButton1, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 76,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(24, 24, 24)
                .add(plot, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 73, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .add(bottomPanelLayout.createSequentialGroup()
                .add(bottomPanelLayout.createParallelGroup(org.jdesktop.layout.
                    GroupLayout.LEADING)
                    .add(jLabel13)
                    .add(jLabel7))
                .add(7, 7, 7)
                .add(bottomPanelLayout.createParallelGroup(org.jdesktop.layout.
                    GroupLayout.TRAILING)
                    .add(timeTxt, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                        100, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                    .add(sysPrintTime, org.jdesktop.layout.GroupLayout.
                        PREFERRED_SIZE, 100, org.jdesktop.layout.GroupLayout.
                        PREFERRED_SIZE))
                .add(225, 225, 225)
```

```
                .add(jLabel1)
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                .add(sysTimeText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                    100, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .add(pBar, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 636, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE))
        .add(24, 24, 24))
);
bottomPanelLayout.setVerticalGroup(
    bottomPanelLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
    .add(org.jdesktop.layout.GroupLayout.TRAILING, bottomPanelLayout.
        createSequentialGroup()
        .addContainerGap()
        .add(bottomPanelLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
            BASELINE)
            .add(sysTimeText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(timeTxt, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(jLabel7, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(jLabel1, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE))
        .add(14, 14, 14)
        .add(bottomPanelLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
            BASELINE)
            .add(sysPrintTime, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(jLabel13, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE))
        .add(22, 22, 22)
        .add(pBar, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 28, org.jdesktop
            .layout.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED, 14, Short.
            MAX_VALUE)
        .add(bottomPanelLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
            BASELINE)
            .add(runButton)
            .add(jButton1, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 23, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(plot))
        .addContainerGap())
);

ParticlesInfo.setBorder(javax.swing.BorderFactory.createTitledBorder(null, "
    Particles Specification", javax.swing.border.TitledBorder.DEFAULT_JUSTIFICATION
    , javax.swing.border.TitledBorder.DEFAULT_POSITION, new java.awt.Font("Tahoma",
     1, 12), new java.awt.Color(51, 51, 51)));
ParticlesInfo.setDoubleBuffered(false);
ParticlesInfo.setFocusable(false);
ParticlesInfo.setFont(getFont());
ParticlesInfo.setPreferredSize(new java.awt.Dimension(675, 0));
ParticlesInfo.setRequestFocusEnabled(false);
ParticlesInfo.setVerifyInputWhenFocusTarget(false);
pRadius.setBackground(new java.awt.Color(0, 0, 0));
pRadius.setFont(getFont());
pRadius.setText("Radius(mn):");
pRadius.setPreferredSize(new java.awt.Dimension(146, 16));

PPCo_Res.setBackground(new java.awt.Color(0, 0, 0));
PPCo_Res.setFont(getFont());
PPCo_Res.setText("P-P Coefficient of Restitution: ");
PPCo_Res.setPreferredSize(new java.awt.Dimension(146, 16));
```

```
pMass.setBackground(new java.awt.Color(0, 0, 0));
pMass.setFont(getFont());
pMass.setText("Density(Kg/m^3):");
pMass.setPreferredSize(new java.awt.Dimension(146, 16));

PSCo_Res.setBackground(new java.awt.Color(0, 0, 0));
PSCo_Res.setFont(getFont());
PSCo_Res.setText("P-S Coefficient of Restitution:");
PSCo_Res.setPreferredSize(new java.awt.Dimension(146, 16));

pRadiusText.setText("12.5");
pRadiusText.setPreferredSize(new java.awt.Dimension(100, 16));

pDensityText.setText("4000");
pDensityText.setPreferredSize(new java.awt.Dimension(100, 16));

PPCo_ResText.setText("1");
PPCo_ResText.setPreferredSize(new java.awt.Dimension(100, 16));

PSCo_ResText.setText("0.5");
PSCo_ResText.setPreferredSize(new java.awt.Dimension(100, 16));

org.jdesktop.layout.GroupLayout ParticlesInfoLayout = new org.jdesktop.layout.
    GroupLayout(ParticlesInfo);
ParticlesInfo.setLayout(ParticlesInfoLayout);
ParticlesInfoLayout.setHorizontalGroup(
    ParticlesInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING
        )
    .add(ParticlesInfoLayout.createSequentialGroup()
        .addContainerGap()
        .add(ParticlesInfoLayout.createParallelGroup(org.jdesktop.layout.
            GroupLayout.LEADING)
            .add(pRadius, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146, org:
                jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(pMass, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE))
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
        .add(ParticlesInfoLayout.createParallelGroup(org.jdesktop.layout.
            GroupLayout.LEADING)
            .add(pDensityText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 100,
                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(pRadiusText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 100,
                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED, 143, Short.
            MAX_VALUE)
        .add(ParticlesInfoLayout.createParallelGroup(org.jdesktop.layout.
            GroupLayout.TRAILING)
            .add(PPCo_Res, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146, org
                .jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(PSCo_Res, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146, org
                .jdesktop.layout.GroupLayout.PREFERRED_SIZE))
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
        .add(ParticlesInfoLayout.createParallelGroup(org.jdesktop.layout.
            GroupLayout.LEADING)
            .add(PSCo_ResText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 100,
                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(ParticlesInfoLayout.createSequentialGroup()
                .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                .add(PPCo_ResText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                    100, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)))
        .addContainerGap())
);
ParticlesInfoLayout.setVerticalGroup(
    ParticlesInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING
```

```
            )
        .add(ParticlesInfoLayout.createSequentialGroup()
            .addContainerGap()
            .add(ParticlesInfoLayout.createParallelGroup(org.jdesktop.layout.
                GroupLayout.BASELINE)
                .add(pRadius, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(pRadiusText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(PPCo_ResText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(PPCo_Res, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .add(7, 7, 7)
            .add(ParticlesInfoLayout.createParallelGroup(org.jdesktop.layout.
                GroupLayout.BASELINE)
                .add(pMass, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(PSCo_ResText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(pDensityText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                .add(PSCo_Res, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                    jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .addContainerGap(14, Short.MAX_VALUE))
);

EnvironmentInfo.setBorder(javax.swing.BorderFactory.createTitledBorder(null, "
    Environment␣Specification", javax.swing.border.TitledBorder.
    DEFAULT_JUSTIFICATION, javax.swing.border.TitledBorder.DEFAULT_POSITION, new
    java.awt.Font("Tahoma", 1, 12), new java.awt.Color(51, 51, 51)));
EnvironmentInfo.setFont(getFont());
envWidth.setFont(getFont());
envWidth.setText("Width/Height(mm):␣");
envWidth.setPreferredSize(new java.awt.Dimension(146, 16));

envWidthText.setText("8");
envWidthText.setPreferredSize(new java.awt.Dimension(100, 16));

org.jdesktop.layout.GroupLayout EnvironmentInfoLayout = new org.jdesktop.layout.
    GroupLayout(EnvironmentInfo);
EnvironmentInfo.setLayout(EnvironmentInfoLayout);
EnvironmentInfoLayout.setHorizontalGroup(
    EnvironmentInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
        LEADING)
    .add(org.jdesktop.layout.GroupLayout.TRAILING, EnvironmentInfoLayout.
        createSequentialGroup()
        .addContainerGap()
        .add(envWidth, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 146, org.
            jdesktop.layout.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
        .add(envWidthText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 100, org
            .jdesktop.layout.GroupLayout.PREFERRED_SIZE)
        .add(549, 549, 549))
);
EnvironmentInfoLayout.setVerticalGroup(
    EnvironmentInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
        LEADING)
    .add(EnvironmentInfoLayout.createSequentialGroup()
        .addContainerGap()
        .add(EnvironmentInfoLayout.createParallelGroup(org.jdesktop.layout.
            GroupLayout.BASELINE)
            .add(envWidth, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16, org.
                jdesktop.layout.GroupLayout.PREFERRED_SIZE)
```

```
                .add(envWidthText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                    org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
            .addContainerGap(18, Short.MAX_VALUE))
    );

    SurfaceInfo.setBorder(javax.swing.BorderFactory.createTitledBorder(null, "Surface
        Specification", javax.swing.border.TitledBorder.DEFAULT_JUSTIFICATION, javax.
        swing.border.TitledBorder.DEFAULT_POSITION, new java.awt.Font("Tahoma", 1, 12),
         new java.awt.Color(51, 51, 51)));
    SurfaceInfo.setDoubleBuffered(false);
    SurfaceInfo.setFont(getFont());
    SurfaceInfo.setOpaque(false);
    SurfaceInfo.setPreferredSize(new java.awt.Dimension(670, 0));
    SurfaceInfo.setRequestFocusEnabled(false);
    SurfaceInfo.setVerifyInputWhenFocusTarget(false);
    surfDensity.setFont(getFont());
    surfDensity.setText("Density(Kg/m^3):");
    surfDensity.setPreferredSize(new java.awt.Dimension(146, 16));

    surfK.setFont(getFont());
    surfK.setText("Constant K:");
    surfK.setPreferredSize(new java.awt.Dimension(146, 16));

    surfD.setFont(getFont());
    surfD.setText("Constant D:");
    surfD.setPreferredSize(new java.awt.Dimension(146, 16));

    surfDepth.setFont(getFont());
    surfDepth.setText("Substrate Depth(mm):");
    surfDepth.setPreferredSize(new java.awt.Dimension(146, 16));

    surfDensityText.setText("2200");
    surfDensityText.setPreferredSize(new java.awt.Dimension(100, 16));

    surfDepthText.setText("7");
    surfDepthText.setPreferredSize(new java.awt.Dimension(100, 16));

    surfDText.setText("6.3e-6");
    surfDText.setPreferredSize(new java.awt.Dimension(100, 16));

    surfKText.setText("1.43");
    surfKText.setPreferredSize(new java.awt.Dimension(100, 16));

    fricCoText.setText("0");
    fricCoText.setPreferredSize(new java.awt.Dimension(100, 16));

    fricCo.setFont(getFont());
    fricCo.setText("Friction Coefficient: ");
    fricCo.setPreferredSize(new java.awt.Dimension(146, 16));

    surfDepthScale.setFont(getFont());
    surfDepthScale.setText("Depth Scale for plot:");
    surfDepthScale.setPreferredSize(new java.awt.Dimension(146, 16));

    cellSize.setFont(getFont());
    cellSize.setText("Cell Size(mm): ");
    cellSize.setPreferredSize(new java.awt.Dimension(146, 16));

    cellSizeText.setText("14");
    cellSizeText.setPreferredSize(new java.awt.Dimension(100, 16));

    buttonGroup1.add(brittleBool);
    brittleBool.setFont(getFont());
    brittleBool.setSelected(true);
```

```
brittleBool.setText("Brittle");
brittleBool.setBorder(javax.swing.BorderFactory.createEmptyBorder(0, 0, 0, 0));
brittleBool.setMargin(new java.awt.Insets(0, 0, 0, 0));

buttonGroup1.add(ductileBool);
ductileBool.setFont(getFont());
ductileBool.setText("Ductile");
ductileBool.setBorder(javax.swing.BorderFactory.createEmptyBorder(0, 0, 0, 0));
ductileBool.setMargin(new java.awt.Insets(0, 0, 0, 0));
ductileBool.addItemListener(new java.awt.event.ItemListener() {
    public void itemStateChanged(java.awt.event.ItemEvent evt) {
        activeDuctile(evt);
    }
});

jLabel8.setFont(new java.awt.Font("Tahoma", 1, 11));
jLabel8.setText("Erosive System: ");

jLabel10.setFont(getFont());
jLabel10.setText("     n1:");

jLabel11.setFont(getFont());
jLabel11.setText("        n2:");

jLabel12.setFont(getFont());
jLabel12.setText("Hv(GPa):");

surfN1Const.setText("1.27");
surfN1Const.setEnabled(false);

surfN2Const.setText("15.5");
surfN2Const.setEnabled(false);

surfHv.setText("0.25");
surfHv.setEnabled(false);

surfDepthScaleText.setText("1");
surfDepthScaleText.setPreferredSize(new java.awt.Dimension(100, 16));

org.jdesktop.layout.GroupLayout SurfaceInfoLayout = new org.jdesktop.layout.
    GroupLayout(SurfaceInfo);
SurfaceInfo.setLayout(SurfaceInfoLayout);
SurfaceInfoLayout.setHorizontalGroup(
    SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
    .add(SurfaceInfoLayout.createSequentialGroup()
        .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
            LEADING)
            .add(SurfaceInfoLayout.createSequentialGroup()
                .addContainerGap()
                .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.
                    GroupLayout.LEADING)
                    .add(SurfaceInfoLayout.createSequentialGroup()
                        .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.
                            layout.GroupLayout.LEADING)
                            .add(surfDensity, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 146, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .add(surfDepth, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 146, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .add(cellSize, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 146, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)
                            .add(fricCo, org.jdesktop.layout.GroupLayout.
```

```
                PREFERRED_SIZE, 146, org.jdesktop.layout.
                GroupLayout.PREFERRED_SIZE))
            .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
            .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.
                layout.GroupLayout.LEADING)
                .add(SurfaceInfoLayout.createSequentialGroup()
                    .add(SurfaceInfoLayout.createParallelGroup(org.
                        jdesktop.layout.GroupLayout.TRAILING)
                        .add(SurfaceInfoLayout.createSequentialGroup()
                            .add(SurfaceInfoLayout.createParallelGroup(
                                org.jdesktop.layout.GroupLayout.LEADING
                                )
                                .add(surfDensityText, org.jdesktop.
                                    layout.GroupLayout.PREFERRED_SIZE,
                                    100, org.jdesktop.layout.
                                    GroupLayout.PREFERRED_SIZE)
                                .add(surfDepthText, org.jdesktop.layout
                                    .GroupLayout.PREFERRED_SIZE, 100,
                                    org.jdesktop.layout.GroupLayout.
                                    PREFERRED_SIZE)
                                .add(cellSizeText, org.jdesktop.layout.
                                    GroupLayout.PREFERRED_SIZE, 100,
                                    org.jdesktop.layout.GroupLayout.
                                    PREFERRED_SIZE))
                            .addPreferredGap(org.jdesktop.layout.
                                LayoutStyle.RELATED, 39, Short.
                                MAX_VALUE))
                        .add(SurfaceInfoLayout.createSequentialGroup()
                            .add(jLabel10)
                            .addPreferredGap(org.jdesktop.layout.
                                LayoutStyle.RELATED)))
                    .add(surfN1Const, org.jdesktop.layout.GroupLayout.
                        PREFERRED_SIZE, 70, org.jdesktop.layout.
                        GroupLayout.PREFERRED_SIZE)
                    .add(23, 23, 23)
                    .add(SurfaceInfoLayout.createParallelGroup(org.
                        jdesktop.layout.GroupLayout.LEADING)
                        .add(org.jdesktop.layout.GroupLayout.TRAILING,
                            SurfaceInfoLayout.createSequentialGroup()
                            .add(surfD, org.jdesktop.layout.GroupLayout
                                .PREFERRED_SIZE, 146, org.jdesktop.
                                layout.GroupLayout.PREFERRED_SIZE)
                            .addPreferredGap(org.jdesktop.layout.
                                LayoutStyle.RELATED)
                            .add(surfDText, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE, 100, org.
                                jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE))
                        .add(org.jdesktop.layout.GroupLayout.TRAILING,
                            SurfaceInfoLayout.createParallelGroup(org.
                            jdesktop.layout.GroupLayout.LEADING)
                            .add(SurfaceInfoLayout.
                                createSequentialGroup()
                                .add(SurfaceInfoLayout.
                                    createParallelGroup(org.jdesktop.
                                    layout.GroupLayout.TRAILING)
                                    .add(SurfaceInfoLayout.
                                        createSequentialGroup()
                                        .add(jLabel11)
                                        .addPreferredGap(org.jdesktop.
                                            layout.LayoutStyle.RELATED)
                                        .add(surfN2Const, org.jdesktop.
                                            layout.GroupLayout.
                                            PREFERRED_SIZE, 70, org.
```

```
                                                    jdesktop.layout.GroupLayout
                                                    .PREFERRED_SIZE)
                                                .add(29, 29, 29))
                                            .add(SurfaceInfoLayout.
                                                createSequentialGroup()
                                                .add(surfDepthScale, org.
                                                    jdesktop.layout.GroupLayout
                                                    .PREFERRED_SIZE, 132, org.
                                                    jdesktop.layout.GroupLayout
                                                    .PREFERRED_SIZE)
                                                .addPreferredGap(org.jdesktop.
                                                    layout.LayoutStyle.RELATED)
                                                    ))
                                        .add(SurfaceInfoLayout.
                                            createParallelGroup(org.jdesktop.
                                            layout.GroupLayout.LEADING)
                                            .add(SurfaceInfoLayout.
                                                createSequentialGroup()
                                                .addPreferredGap(org.jdesktop.
                                                    layout.LayoutStyle.RELATED)
                                                .add(jLabel12)
                                                .addPreferredGap(org.jdesktop.
                                                    layout.LayoutStyle.RELATED)
                                                .add(surfHv, org.jdesktop.
                                                    layout.GroupLayout.
                                                    PREFERRED_SIZE, 70, org.
                                                    jdesktop.layout.GroupLayout
                                                    .PREFERRED_SIZE))
                                            .add(org.jdesktop.layout.
                                                GroupLayout.TRAILING,
                                                surfDepthScaleText, org.
                                                jdesktop.layout.GroupLayout.
                                                PREFERRED_SIZE, 100, org.
                                                jdesktop.layout.GroupLayout.
                                                PREFERRED_SIZE)))
                                    .add(org.jdesktop.layout.GroupLayout.
                                        TRAILING, SurfaceInfoLayout.
                                        createSequentialGroup()
                                        .add(surfK, org.jdesktop.layout.
                                            GroupLayout.PREFERRED_SIZE, 146,
                                            org.jdesktop.layout.GroupLayout.
                                            PREFERRED_SIZE)
                                        .addPreferredGap(org.jdesktop.layout.
                                            LayoutStyle.RELATED)
                                        .add(surfKText, org.jdesktop.layout.
                                            GroupLayout.PREFERRED_SIZE, 100,
                                            org.jdesktop.layout.GroupLayout.
                                            PREFERRED_SIZE)))))
                            .add(fricCoText, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 100, org.jdesktop.layout.
                                GroupLayout.PREFERRED_SIZE)))
                    .add(jLabel8)))
            .add(SurfaceInfoLayout.createSequentialGroup()
                .add(35, 35, 35)
                .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.
                    GroupLayout.LEADING)
                    .add(ductileBool)
                    .add(brittleBool))))
        .addContainerGap())
);
SurfaceInfoLayout.setVerticalGroup(
    SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
    .add(SurfaceInfoLayout.createSequentialGroup()
        .addContainerGap()
```

```
                .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
                    TRAILING)
                    .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.
                        GroupLayout.BASELINE)
                        .add(cellSize, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                            org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                        .add(cellSizeText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                            16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
                    .add(SurfaceInfoLayout.createSequentialGroup()
                        .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.
                            GroupLayout.BASELINE)
                            .add(surfDensity, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 16, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE)
                            .add(surfDensityText, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 16, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE)
                            .add(surfDText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                                16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                            .add(surfD, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
                        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                        .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.
                            GroupLayout.BASELINE)
                            .add(surfDepth, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                                16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                            .add(surfDepthText, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 16, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE)
                            .add(surfKText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                                16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                            .add(surfK, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))
                        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                        .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.
                            GroupLayout.BASELINE)
                            .add(surfDepthScaleText, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 16, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE)
                            .add(surfDepthScale, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 16, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE))))
                .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.GroupLayout.
                    LEADING)
                    .add(SurfaceInfoLayout.createSequentialGroup()
                        .add(33, 33, 33)
                        .add(jLabel8, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 16,
                            org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                        .add(14, 14, 14)
                        .add(brittleBool)
                        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                        .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.
                            GroupLayout.BASELINE)
                            .add(ductileBool)
                            .add(surfHv, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                                16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                            .add(surfN2Const, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 16, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE)
                            .add(jLabel11)
                            .add(surfN1Const, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE, 16, org.jdesktop.layout.GroupLayout.
                                PREFERRED_SIZE)
                            .add(jLabel10)
```

```
                        .add(jLabel12)))
                .add(SurfaceInfoLayout.createSequentialGroup()
                    .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
                    .add(SurfaceInfoLayout.createParallelGroup(org.jdesktop.layout.
                        GroupLayout.BASELINE)
                        .add(fricCoText, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE
                            , 16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
                        .add(fricCo, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                            16, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE))))
            .addContainerGap(26, Short.MAX_VALUE))
);

org.jdesktop.layout.GroupLayout layout = new org.jdesktop.layout.GroupLayout(
    getContentPane());
getContentPane().setLayout(layout);
layout.setHorizontalGroup(
    layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
    .add(layout.createSequentialGroup()
        .addContainerGap()
        .add(bottomPanel, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 675, org.
            jdesktop.layout.GroupLayout.PREFERRED_SIZE)
        .addContainerGap(org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, Short.
            MAX_VALUE))
    .add(org.jdesktop.layout.GroupLayout.TRAILING, layout.createSequentialGroup()
        .addContainerGap(org.jdesktop.layout.GroupLayout.DEFAULT_SIZE, Short.
            MAX_VALUE)
        .add(layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
            .add(SurfaceInfo, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 675,
                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(nozzleInfo, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 675,
                org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(EnvironmentInfo, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE,
                675, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE)
            .add(ParticlesInfo, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org
                .jdesktop.layout.GroupLayout.DEFAULT_SIZE, org.jdesktop.layout.
                GroupLayout.PREFERRED_SIZE))
        .add(15, 15, 15))
);
layout.setVerticalGroup(
    layout.createParallelGroup(org.jdesktop.layout.GroupLayout.LEADING)
    .add(layout.createSequentialGroup()
        .addContainerGap()
        .add(nozzleInfo, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 179, org.
            jdesktop.layout.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
        .add(ParticlesInfo, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 92, org
            .jdesktop.layout.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
        .add(SurfaceInfo, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, 225, org.
            jdesktop.layout.GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED)
        .add(EnvironmentInfo, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.
            jdesktop.layout.GroupLayout.DEFAULT_SIZE, org.jdesktop.layout.
            GroupLayout.PREFERRED_SIZE)
        .addPreferredGap(org.jdesktop.layout.LayoutStyle.RELATED, 15, Short.
            MAX_VALUE)
        .add(bottomPanel, org.jdesktop.layout.GroupLayout.PREFERRED_SIZE, org.
            jdesktop.layout.GroupLayout.DEFAULT_SIZE, org.jdesktop.layout.
            GroupLayout.PREFERRED_SIZE)
        .addContainerGap())
);
java.awt.Dimension screenSize = java.awt.Toolkit.getDefaultToolkit().getScreenSize
    ();
setBounds((screenSize.width-708)/2, (screenSize.height-815)/2, 708, 815);
```

```
}// </editor-fold>

/**
 * Changes the current state of the input fields of constants for ductile erosive
     systems.
 *
 * @param evt the itemStateChanges event of Dutile check box.
 */
private void activeDuctile(java.awt.event.ItemEvent evt) {
    this.surfN1Const.setEnabled(!this.surfN1Const.isEnabled());
    this.surfN2Const.setEnabled(this.surfN1Const.isEnabled());
    this.surfHv.setEnabled(this.surfN1Const.isEnabled());
}

/**
 * Invokes a method from the Plot class to show the three-dimensional view of the
     surface
 *
 * @param evt the actionPerformed event of the Plot button.
 */
private void plotHandler(java.awt.event.ActionEvent evt) {
    depthScale = Double.parseDouble(surfDepthScaleText.getText())*-1;
    try {
        new Plot(depthScale);
    } catch (RemoteException ex) {
        ex.printStackTrace();
        JOptionPane.showMessageDialog(this, "first");
    } catch (VisADException ex) {
        ex.printStackTrace();
        JOptionPane.showMessageDialog(this, "second");
    }
}

/**
 * Creates some results and terminates the execution.
 *
 * @param evt the mouseClicked event of the exit button.
 */
private void exitHandler(java.awt.event.MouseEvent evt) {

    if(SimulationManager.SystemTime==0)
        System.exit(0);

    if(SimulationManager.SystemTime>0)
        mySimulation.SimulationManager.printStatistics();
        try
        {
            FileOutputStream fileOut;
            fileOut = new FileOutputStream("mySimResult/ExcelReport.xls");
            mySimulation.SimulationManager.wb.write(fileOut);
            fileOut.close();
            fileOut=null;

            if(SimulationManager.cell2DWbY!=null)
            {
                System.out.println("writing last excel file at time " +
                    SimulationManager.SystemTime);
                String sheetName = "mySimResult/CrossSection-Time" + SimulationManager.
                    SystemTime + ".xls";
                fileOut = new FileOutputStream(sheetName);
                SimulationManager.cell2DWbY.write(fileOut);
                fileOut.close();
                fileOut=null;
            }
```

```
        }catch(IOException e)
        {
            System.err.println("error in entering data into excel sheet");
        }
    System.exit(0);
}

/**
 * Creates new thread for running the simulation and invokes its run method to start
     the simulation.
 * The old thread is used to update GUI.
 *
 * @param evt the mouseClicked event of the run button.
 */
private void runHandler(java.awt.event.MouseEvent evt) {
   thread = new Thread( this );
   thread.start();
}

/**
 * Represents the code executed by the thread created by runHandler.
 */
public void run(){
    int i;
    this.runButton.setEnabled( false );
    try{
        runSimulation();
    }catch(InterruptedException e){;}

    this.runButton.setEnabled(true);
    this.plot.setEnabled(true);
}

/**
 * An abstract method whose implementation is presented in the SimulationManager class
     extending Interface.
 *
 * @throws java.lang.InterruptedException
 */
abstract public void runSimulation() throws InterruptedException;

// Variables declaration - do not modify
private javax.swing.JPanel EnvironmentInfo;
private javax.swing.JLabel PPCo_Res;
public javax.swing.JTextField PPCo_ResText;
private javax.swing.JLabel PSCo_Res;
public javax.swing.JTextField PSCo_ResText;
private javax.swing.JPanel ParticlesInfo;
private javax.swing.JPanel SurfaceInfo;
private javax.swing.JLabel beta;
public javax.swing.JTextField betaText;
private javax.swing.JPanel bottomPanel;
public javax.swing.JRadioButton brittleBool;
private javax.swing.ButtonGroup buttonGroup1;
private javax.swing.JLabel cellSize;
public javax.swing.JTextField cellSizeText;
public javax.swing.JRadioButton ductileBool;
private javax.swing.JLabel envWidth;
public javax.swing.JTextField envWidthText;
private javax.swing.JLabel fricCo;
public javax.swing.JTextField fricCoText;
private javax.swing.JButton jButton1;
private javax.swing.JLabel jLabel1;
private javax.swing.JLabel jLabel10;
```

```
    private javax.swing.JLabel jLabel11;
    private javax.swing.JLabel jLabel12;
    private javax.swing.JLabel jLabel13;
    private javax.swing.JLabel jLabel2;
    private javax.swing.JLabel jLabel3;
    private javax.swing.JLabel jLabel4;
    private javax.swing.JLabel jLabel5;
    private javax.swing.JLabel jLabel6;
    private javax.swing.JLabel jLabel7;
    private javax.swing.JLabel jLabel8;
    private javax.swing.JLabel jLabel9;
    private javax.swing.JSeparator jSeparator1;
    private javax.swing.JLabel nHeight;
    public javax.swing.JTextField nHeightText;
    public javax.swing.JTextField nOrientAngleText;
    private javax.swing.JLabel nPassDistance;
    public javax.swing.JTextField nPassDistanceText;
    private javax.swing.JLabel nPosition;
    private javax.swing.JLabel nRadius;
    public javax.swing.JTextField nRadiusText;
    public javax.swing.JTextField nVXText;
    public javax.swing.JTextField nVYText;
    public javax.swing.JTextField nVZText;
    private javax.swing.JLabel nVelocity;
    public javax.swing.JTextField nXText;
    public javax.swing.JTextField nYText;
    public javax.swing.JTextField nZText;
    private javax.swing.JLabel nlunchFreq;
    public javax.swing.JTextField nlunchFreqText;
    private javax.swing.JPanel nozzleInfo;
    public final javax.swing.JProgressBar pBar = new javax.swing.JProgressBar();
    public javax.swing.JTextField pDensityText;
    private javax.swing.JLabel pMass;
    private javax.swing.JLabel pRadius;
    public javax.swing.JTextField pRadiusText;
    private javax.swing.JLabel pVSize;
    public javax.swing.JTextField pVSizeText;
    public javax.swing.JButton plot;
    private javax.swing.JButton runButton;
    private javax.swing.JLabel surfD;
    public javax.swing.JTextField surfDText;
    private javax.swing.JLabel surfDensity;
    public javax.swing.JTextField surfDensityText;
    private javax.swing.JLabel surfDepth;
    private javax.swing.JLabel surfDepthScale;
    public javax.swing.JTextField surfDepthScaleText;
    public javax.swing.JTextField surfDepthText;
    public javax.swing.JTextField surfHv;
    private javax.swing.JLabel surfK;
    public javax.swing.JTextField surfKText;
    public javax.swing.JTextField surfN1Const;
    public javax.swing.JTextField surfN2Const;
    public javax.swing.JTextField sysPrintTime;
    public javax.swing.JTextField sysTimeText;
    public javax.swing.JTextField timeTxt;
    public javax.swing.JTextField vConst1;
    public javax.swing.JTextField vConst2;
    // End of variables declaration
}
```

# Bibliography


[1] A.P. Verma and G.K. Lal. An experimental study of abrasive jet machining. *International Journal of Machine Tool Design and Research*, 24(1):19–29, 1984.

[2] V.C. Venkatesh, T.N. Goh, K.H. Wong, and M.J. Lim. An empirical study of parameters in abrasive jet machining. *International Journal of Machine Tools and Manufacture*, 29(4):471–479, 1989.

[3] R. Balasubramaniam, J. Krishnan, and N. Ramakrishnan. An empirical study on the generation of an edge radius in abrasive jet external deburring (AJED). *Journal of Materials Processing Technology*, 99:49–53, 2000.

[4] P.J. Slikkerveer and F.H. in't Veld. Model for patterned erosion. *Wear*, 233-235:377–386, 1999.

[5] R. Balasubramaniam, J. Krishnan, and N. Ramakrishnan. A study on the shape of the surface generated by abrasive jet machining. *Journal of Materials Processing Technology*, 121(1):102–106, 2002.

[6] J.H.M. Ten Thije Boonkkamp and J.K.M Jansen. An analytical solution for mechanical etching of glass by powder blasting. *Journal of Engineering Mathematics*, 43(2-4):385–399, 2002.

[7] M. Achtsnick, P.F. Geelhoed, A.M. Hoogstrate, and B. Karpuschewski. Modelling and evaluation of the micro abrasive blasting process. *Wear*, 259(1-6):84–94, 2005.

[8] I. Finnie. Erosion of surfaces by solid particles. *Wear*, 3:87–103, 1960.

[9] J.G.A. Bitter. A study of erosion phenomena, part I. *Wear*, 6:5–21, 1963.

[10] I.M. Hutchings and R.E. Winter. Particle erosion of ductile metals: a mechanism of material removal. *Wear*, 27:121–128, 1974.

[11] G.L. Sheldon and I. Finnie. On the ductile behaviour of nominally brittle materials during erosive cutting. *Journal of Engineering for Industry*, 88:387–392, 1966.

[12] Y. Ballout, J.A. Mathis, and J.E. Talia. Solid particle erosion mechanism in glass. *Wear*, 196(1):263–269, 1996.

[13] R.E. Jewett, P. I. Hagouel, A.R. Neureuther, and T. Van Duzer. Line-profile resist development simulation techniques. *Polymer Engineering and Science*, 17(6):381–384, 1977.

[14] P. I. Hagouel. *X-ray lithography fabrication of blazed diffraction gratings.* PhD dissertation, University of California at Berkeley, 1976.

[15] S. Osher and James A. Sethian. Fronts propagating with curvature-dependent speed: algorithms based on Hamilton-Jacobi formulations. *Journal of Computational Physics*, 79(1):12–49, 1988.

[16] F.H. Dill, A.R. Neureuther, J.A. Tuttle, , and E.J. Walker. Modeling projection printing of positive photoresists. *IEEE Transactions on Electron Devices*, 22(7):456–464, 1975.

[17] K.K.H. Toh, A.R. Neureuther, and E.W. Scheckler. Algorithms for simulation of three-dimensional etching. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(5):616–624, 1994.

[18] D. Adalsteinsson and J. A. Sethian. A level set approach to a unified model for etching, deposition, and lithography III: redeposition, reemission, surface diffusion, and complex simulations. *Journal of Computational Physics*, 138(1):193–223, 1997.

[19] E.W. Scheckler, N.N. Tam, A.K. Pfau, and A.R. Neureuther. An efficient volume-removal algorithm for practical three-dimensional lithography simulation with experimental verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(9):1345–1356, 1993.

[20] E. Strasser and S. Selberherr. Algorithms and models for cellular based topography simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(9):1104–1114, 1995.

[21] R.M. Brach. Formulation of rigid body impact problems using generalized coefficients. *International Journal of Engineering Science*, 36(1):61–71, 1998.

[22] David Ciampini. Computer simulation of interference effects in particle streams. Master's thesis, University of Toronto, Department of Mechanical and Industrial Engineering, 2002.

[23] A. Ghobeity, T. Krajac, T. Burzynski, M. Papini, and J.K. Spelt. Surface evolution models in abrasive jet micromachining. *Wear*, 264(3-4):185–198, 2008.

[24] R.M. Brach. Impact dynamics with application to solid particle erosion. *International Journal of Impact Engineering*, 7(1):37–53, 1998.

[25] H. Getu, A. Ghobeity, J.K. Spelt, and M. Papini. Abrasive jet micromachining of polymethylmethacrylate. *Wear*, 263(7-12):1008–1015, 2007.

[26] B.J. Alder and T.E. Wainwright. Studies in molecular dynamics. I. general method. *Journal of Chemical Physics*, 31(2):459–466, 1959.

[27] H. Sigurgeirsson, A. Stuart, and W-L. Wan. Algorithms for particle-field simulations with collisions. *Journal of Computational Physics*, 172(2):766–807, 2001.

[28] Hanan Samet. *The design and analysis of spatial data structures.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[29] D. Ciampini, J.K. Spelt, and M. Papini. Simulation of interference effects in particle streams following impact with a flat surface, Part I: Theory and analysis. *Wear*, 254(3-4):237–249, 2003.

[30] A. Ghobeity, H. Getu, T. Krajac, J. K. Spelt, and M. Papini. Process repeatability in abrasive jet micro-machining. *Journal of Materials Processing Technology*, 190(1-3):51–60, 2007.

[31] A. Ghobeity, H. Getu, M. Papini, and J.K. Spelt. Surface evolution models for abrasive jet micromachining of holes in glass and polymethylmethacrylate (PMMA). *Journal of Micromechanics and Microengineering*, 17(11):2175–2185, 2007.

[32] Periasamy Chinnapalaniandi. *An experimental study of particle-laden jet interactions with cocurrent flows.* PhD dissertation, Case Western Reserve University, Department of Mechanical and Areospace Engineering, Aug 1992.

[33] I.M. Hutchings. Deformation of metal surfaces by the oblique impact of square plates. *International Journal of Mechanical Sciences*, 19(1):45–52, 1977.

[34] S. Dhar, T. Krajac, D. Ciampini, and M. Papini. Erosion mechanisms due to impact of single angular particles. *Second International Conference on Erosive and Abrasive Wear*, 258(1-4):567–579, 2005.