A GPGPU implementation of the discrete element method applied to modeling the dynamic particulate environment inside a tumbling mill

by

Marius Hromnik

Supervised by Indresan Govender
Co-supervised by Spencer Wheaton

A thesis submitted in fulfillment of the requirements for the degree of

Master of Science

University of Cape Town

ABSTRACT

Tumbling mills have been an integral part of the comminution circuit for more than a century. With the advent of better computing, discrete element modeling (DEM) has taken on the challenge to model the dynamic particulate environment inside these devices in the search for understanding and hence improving the process of the size reduction of ore. This process represents a large percentage of the energy consumption of a mine. In this work, a discrete element modeling tool was built on a GPU-based platform to perform simulations on a single commodity hardware PC. With a view to elucidating the governing mechanisms inside such devices, the extreme capabilities of the GPU are utilised to provide performance and accurate simulation. The simulation environment offers control that can never be achieved in an experimental setup. Notwithstanding, when agreement with physical experiment is achieved, confidence can be gained in the computational results.

In this work the foundations and framework for a large scale GPU based discrete element modeling tool have been built with an emphasis on strict physics requirements, rather than on performance or appearance. In this regard we demonstrate the validity of the GPU implementation of a Hertz-Mindlin-based contact model.

ACKNOWLEDGMENTS

The person whom I owe this project to, is my supervisor Indresan Govender. He has inspired me through his vibrant approach to research and his willingness to experiment with new ideas allowed for the use of GPU technology in this project, in an industry where it is rarely encountered. He has guided my often over-enthusiastic and optimistic ideas and encouraged me through this process.

Much more importantly than this to me is that I have learned more and developed more as a person through my interaction with him than I could ever have hoped from a supervisor. He is a man whom I respect for his ethics and his professional abilities and he has shown me through many occurrences throughout my interaction with him that he is a good man.

Gary Tupper has been an unsung voice of reason throughout my research. With an unusual sense of humour and a knowledge of physics beyond that of anyone I've ever encountered, Gary has been able to solve, or suggest approaches to many problems I have come across along the way. His intuition is invaluable and an inspiration to any admirer of physics.

The National Institute of Theoretical Physics and the Centre for Minerals Research provided the funding for this work. My interaction with the CMR and the world of chemical engineering has also been very useful to me in this work.

Contents

1 Introduction

Devices such as tumbling mills have been used as an integral part of the comminution process in the mining industry for decades. The comminution process (the process of size reduction of mined ore to allow liberation of the desired metals) is responsible for ca. 50% of the operating cost of mining [DL11] and still the internal mechanisms are little understood. That only about 20% of the energy used by a tumbling mill goes into breaking down the ore inside it, is a good illustration of this [DL11]. A brief look at global energy consumption and mining energy consumption shows why it is so important that we improve the status of comminution technology. We believe that this problem should be tackled by improving the very shaky foundations of comminution theory. While black-box models have yielded highly tweaked recipes for mill optimisation, their validity to narrow windows of operating conditions (boundary conditions) prevent, *apriori*, the ability to extrapolate to new operating regimes. In the face of ever dwindling resources and depleting ore bodies, the issue of viability becomes critical, precipitating the need for adaptive models that yield new (and dynamic) optima for profitable operations. Only mechanistic models - based on fundamental principles of granular flow and fracture mechanics - offer the capability to meet these new challenges. To develop the theory of a tumbling mill from a solid foundation requires understanding of the internal processes within a tumbling mill. More generally, one requires an understanding of the internal mechanisms of a dynamic particulate system.

The problem of studying internal mechanisms of dynamic particulate systems has been attacked using the computationally intensive discrete element method (DEM) since the 1970s. The idea is to simulate the particulate system computationally and to retrieve from the simulation, information that can be used to make the operation more efficient. Once the computational results have been shown to reflect reality, through some kind of validation, they give us a window into the processes occurring inside the mill. After three decades of development, several commercial DEM packages (EDEM by DEM Solutions, PFC by Itasca and Rocky by Granular Dynamics) and a few in-house computer codes by big research institutions (CSIRO, Lawrence Livermore, Birmingham University and Michigan Institute of Technology (MIT)) have attacked the problem of simulating the environment inside a tumbling mill, but little work has been done using vector processing units to perform DEM for scientific purposes[1]. Developments (hardware and software), initially intended for the computer gaming industry, have opened the field of high performance scientific computing to GPU (Graphics Processing Unit) and other vector processing units.

DEM computer simulations do not develop tumbling mill theory directly. What they do offer is the possibility of testing such theory. Ideal situations can be simulated (for example with perfect particle shape) with very simplified operating conditions and detailed results can be obtained at every stage in the process. This allows for comparison

[1]DEM has been implemented and used for many purposes other than scientific simulations and vector processing implementations for such scenarios exist

of theory and reality in a way that is impossible to obtain from experimental setups because of the ill-defined irregularities (like shape and homogeneity) and the variety of uncertainty sources in physical experiments that make it very difficult to identify the causes of certain measured results.

In this thesis, a tool and its framework is developed to simulate specifically the environment inside a tumbling mill, but more generally to investigate a wide variety of particulate systems. The discrete element method is the technique employed to determine the dynamics of the system and the C++/CUDA framework is used to implement this tool. The extremely computationally intensive calculation is performed on a GPU (graphics processing unit) rather than a CPU (central processing unit). The physics employed to evolve the particulate system is implemented and tested in the controlled environment of numerical/functional programming and the physical material properties used are obtained from experimental data.

2 Motivation

In the search for understanding of the comminution process, we find ourselves in an interesting position. To fully characterise a complex system like a tumbling mill, three approaches have been identified as useful and complimentary. Continuum-based granular flow theories model the underlying mechanisms governing the flow field inside a tumbling mill, experimental capabilities like positron emission particle tracking (PEPT) give us accurate, time-averaged kinematic measurements, and computational technologies based on elementary contact mechanics theory attempt to model the kinematic and dynamical environment between the particle constituents. Each of these techniques have been shown to give good results to subsets of the real tumbling mill system, however, no single approach offers greater (quantifiable) advantage than the others. What is clear is that all three approaches require further research and development to improve their contribution to the solution. In this thesis, a computational study was built from the ground up to complement research being done in the other two fields. Gained by this work is full control of the mechanisms used, which forces rigour and adds considerable meaning to the ideas drawn from it. A necessary platform has been built on which a further, extended and more complete simulation environment will be built that will be used to give insight into fundamental comminution theories that are developing in this field.

Existing computational simulation technologies currently have grave short-comings that hinder their usefulness in research. The performance of the tool developed in this thesis has allowed it to be used in scenarios that previously took so long to compute that they were hardly useful or practically impossible to run. We achieved, through full control of the physics implemented, a greater understanding of the theory implemented by discrete element modeling. We achieved, through control of the extraction of information from a discrete element method simulation, exactly (and only) the information we required. This is a great improvement on available computational tools, especially those that do not allow full control of the contact mechanics and logging process.

The use of a GPU-based simulation has been demonstrated to be comparable, but more effective for us, in the modeling of the environment inside a tumbling mill than the CPU-based simulations or simulation packages at our disposal.

3 Background

Tumbling mills utilise perhaps the simplest theory imaginable for crushing ore. Large drums fitted with lifters or drum liners are filled with steel balls (referred to as grinding media) and mined ore typically to 40 - 65% of the mill volume and then rotated about the major axis at typically 50 - 80% of the critical speed of the mill. The critical speed of a mill (diameter D) is calculated as $\frac{60}{2\pi}\sqrt{\frac{2g}{D}}$ (rpm) and denotes the (theoretical) speed at which a (infinitely) small particle just centrifuges under no-slip boundary conditions. The resulting cataracting and cascading charge dissipates energy in the ore some of which breaks it. This process is thought to be only about 20% energy efficient [FR81]. To illustrate the significance of this, it is worthwhile noting that a study performed in South Africa in 2002 estimated that the mining industry was responsible for 18% of the total national electricity consumption [Kil02]. More than half of this energy is used in the comminution circuit. A graph taken from the US department of energy, 2001 [DOE01], shows a breakdown of usage of energy by mining in the US industry.

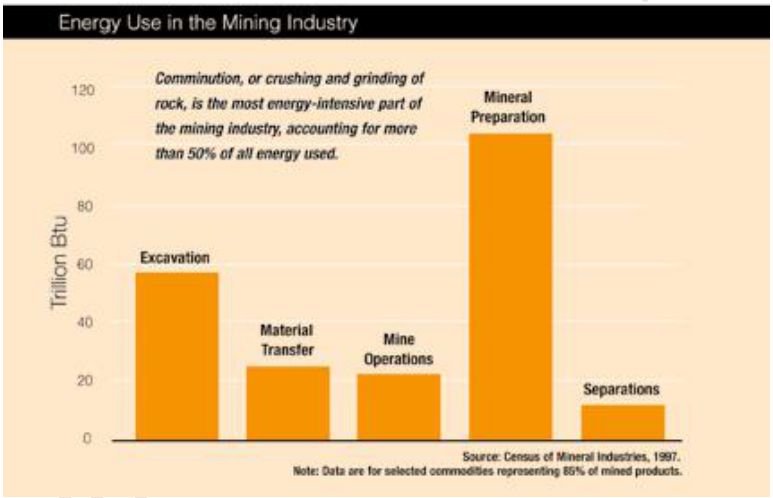


Fig 3.1: Annual energy usage in US mining industry [DOE01]

The environment inside a tumbling mill can definitely be classified as hostile. Slurry is used to transport finely crushed ore out of the mill through a mesh grating at the end of the mill that controls the size of the output. The temperature inside a mill is high and even determining the size distribution of the charge is a tedious, unpleasant and difficult task. Obtaining a picture of what is happening inside an industrial scale tumbling mill, at this point, is still impossible. However, some techniques such as using Xray 3D vision to track an opaque particle in a transparent pilot scale mill have successfully been used to obtain the velocity field of the size class of the tracer particle. Xrays have an energy range of 0.12 - 12 keV (soft Xrays with low penetration) and 12 - 120 keV (hard Xrays with higher penetration), but still the energy and wavelength of Xrays limit their usage to small and transparent mills. The theory is that, by tracking a single particle over a large number of revolutions of the mill, one obtains a statistically satisfactory image

of the dynamics of any particle of that size class inside the mill. Of course this does require a slight stretch of the imagination and one ends up with limited data about a single insignificant particle.

Another experimental technique, known as positron emission particle tracking, uses a tracer particle impregnated with a positron emitting radionuclide that emits coincident 511keV gamma rays that can be detected and used by triangulation to quite accurately determine the position of the tracer particle. This technique can be used in a far wider range of scenarios, as the gamma rays have good penetration through Xray opaque media as well. This technique is less accurate, but it will be discussed in detail later.

All experimental setups to determine the internal environment of a tumbling mill have grave shortcomings. One can obtain very limited information (velocity and position) of a particle (or size class) inside a mill and the accuracy of this data is still subject to experimental inaccuracies. Experiments also tend to be expensive and time consuming to investigate and require expensive technology. It is also impossible to experimentally run all the scenarios one would like to run. Essential information about inter-particle collisions is simply impossible to obtain experimentally. Computer simulation of this environment is a natural tool, despite the shortcomings of the discrete element method.

There are a number of techniques for modeling the environment. Among them are the continuum methods that treat the charge as if it were a viscous fluid. This works well for describing the slurry containing the finely ground ore (known as fines), but granular effects in the charge are lost. As far as discrete methods go, the technique of using cellular automata has, to our knowledge, not been used, and rigid body methods, which make use of rigid-body representations of particles in the charge, are very computationally expensive, although much unpublished work has gone into this field and tools do currently exist that seem to be able to handle rigid body methods rather well.

The environment of a tumbling mill contains various elements, including large oddly shaped ore pieces, which are difficult to justify modeling as spheres (or granules made of clusters of spheres - cluster (granule) logic will be discussed later), fine micro particles that one simply can't model using DEM and slurry that could be treated as a fluid. In light of this, it is worthwhile investigating each of the computational modeling approaches in greater detail because care must be taken in the implementation of any method if meaningful results are to be obtained.

3.1 Continuum methods

The fluid dynamics approach treats the environment as a set of physical quantities described by differential equations that can be solved at various points and evolved with time. The differential equations governing fluid dynamics are well supported by theory and fluid dynamics simulations have been implemented on both CPU and GPU [GWL+03] [Har05] [WC10]. The first application of the continuum approach to

modeling granular materials was made by Savage who used the continuum equations of motion and kinetic theories [Sav79]. Granular materials like sand have been modeled using the fluid dynamics approach for the engineering and computer graphics industries [HL98] but, in this case, the justification for using this approach was that sand responds to compression and shear stresses in a way that is typical of viscous fluids [L09]. This justification would not hold for the scenario of a tumbling mill.

A more complex and physically justifiable technique was suggested and implemented by Zhu and Bridson [ZB05] which treated the surface of the granular material as a mesh of particles while superimposing boundary conditions and incompressibility. The differential equations solver they used included both inter-granule and boundary friction.

For our purposes, the continuum approach could conceivably give us the shape of the charge inside a tumbling mill, if the differential equation solver were to include corrections for this scenario (particle size distribution, typical charge velocities...etc) but it would never give us inter-particle (or inter-granule) information which is useful to us if we want to investigate the effect on breakage or power draw of a mill of the operating parameters and design. The necessary corrections however, might not even be possible to define or implement. However, modeling the effect of slurry in a mill can be well modeled using this approach and hence it is important to take note of it. Coupling a continuum approach, able to model the slurry inside a mill, to another approach better suited to handle the granular nature of the charge would be a useful extension of the work undertaken in this thesis.

3.2 Cellular automata approach

I include this just for interest sake (and for completeness) because it is an interesting idea. The idea is that the space inside the mill is divided up into voxels (cells) each with a state that is evolved with time based on the state of its neighbouring voxels using a set of predefined rules applied to each voxel, or set of voxels, at each timestep. This design, though very simple, has been used in many fields including random number generation [W86], bacterial and tumour growth modeling, fluid turbulance modeling, smoke evolution and fire modeling [L09]. It has been used in granular modeling for modeling avalanches [BTW87] [NG01] and sand pile formation [GGH+98].

Cellular automata approaches give very unpredictable results because the rule set defining the evolution cannot be based directly on physical properties. Also, the fact that space is continuous, not discrete as it is modeled using this approach, limits the accuracy of the results.

3.3 Discrete element methods

First introduced and implemented by Cundall and Strack in 1979 [CS79], DEM was used to model soil mechanics using 2D disk shaped particles. Ideas from molecular dynamics were used. Molecular dynamics has some fundamental differences to the ideas of DEM today (as used in this thesis). In molecular dynamics, groups of atoms are modeled, so distances between elements are on the atomic scale. Time-steps must be in the nanosecond region and, to prevent the number of elements from being unmanageably enormous, periodic boundary conditions are imposed. Of course, a uniform environment is assumed.

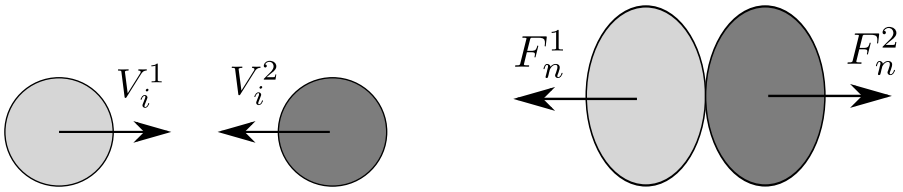
DEM uses macroscopic or mesoscopic particles (in this thesis we did not go below the 1mm diameter particle limit). As in molecular dynamics, spherical particles were used but the time-step is in the microsecond region. It is interesting to note that for the purpose of appearance only, time-steps in the millisecond region have been successfully used in the computer gaming industry which allows, with the use of clever techniques, for realtime DEM [L09]. Although, in this work, emphasis was placed on performance rather than physical accuracy as it was done for the computer gaming industry where realtime performance is a baseline requirement. Boundary conditions are explicitly modeled through interaction with geometric surfaces, modeled as triangle or polygon meshes. Mesh to particle conversion methods have been successfully used to remove this interaction from DEM simulations to improve performance. Particle numbers need to be large for realistic results.

DEM evolves a system of particles by calculating the net force on each particle and integrating over the time-step (using Newton's 2nd law of motion[2]). This net force is a combination of external forces such as gravity, inter-particle forces due to collisions, long range inter-particle forces such as electrostatic attractions and repulsions, and forces imposed by geometries within the simulation also due to collisions. It is important to look at the physical forces a particle experiences (as opposed to the forces modeled) because the differences between these forces and the forces modeled must be considered before using a contact model to model one's specific scenario. One should note that none of the contact models used practically to perform large scale DEM simulations are built entirely from physical properties. This will be discussed in more detail when the contact model we used is described.

The force a particle experiences in a collision can be split into three components. When a particle collides with another particle it experience a head-on repulsive force that acts in the normal direction. This direction is defined as the vector connecting the centres of the two particles (spheres). This force can be understood absolutely using Young's theory that deals with stresses and strains within solids. It is a perfectly physical force. Also in this direction acts a normal damping force. This is physically related to the energy lost in a collision and goes into heating and deforming the particles, as well as

[2]Sometimes very liberally interpreted!

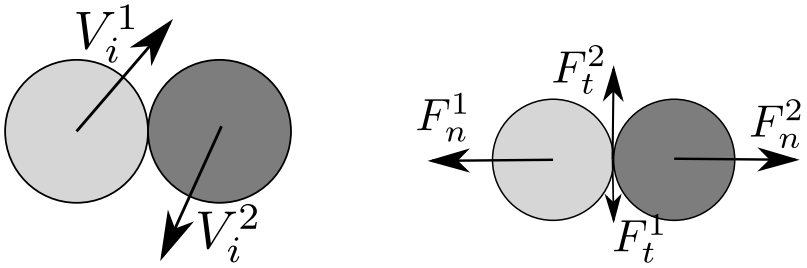
breaking them. This force is very important to us if we want to understand breakage mechanisms, but it is very difficult to model it on physical material properties. Fig 3.2 illustrate what physically occurs in a head-on collision where breakage does not occur. The incoming velocities in (a) result in deformation when in contact and the only force each particle experiences is in the normal direction (b). Note that this deformation is not uniform, as illustrated, and there is no current theory able to predict it.



(a) Incoming velocities (b) Forces experienced

Fig 3.2: Forces in head-on particle-particle collision

There is also a force in the tangential direction. This direction is defined as the direction of the component of the relative velocity of the particle that is not in the normal direction. This force is a result of the fact that particles are not always (realistically never) involved in head-on collisions. This force is related to the magnitude of the component of the relative velocity of the particle in the tangential direction but it is also related to the relative angular velocity of the particle. One must note that in a 3 dimensional DEM simulation one must add these two relative velocity components as vectors. The energy lost to this force, modeled as a damping force in the tangential direction, is also useful for us because it contributes to breakage and it contributes to breakage through a different mechanism. Again the models for this energy loss are fuzzy and not based directly on physical properties. Many models have been proposed, some making corrections to the mishandling of other models of various scenarios, but determining the ideal model is not the subject of this thesis. Fig 3.3 illustrates the velocities of two particles at the point of collision (a) and the resulting forces on each particle (b). Note that in three dimensions the forces in the tangential directions lie in the plane defined by the vector connecting the particles' centres.



(a) Incoming velocities (b) Forces experienced

Fig 3.3: Tangential force due to non coincident incoming velocities

The force of friction is the third force that must be considered. This affects the tangential component of the force and consists of static and kinetic friction. Static friction is applied

when particles are in contact for a prolonged period and dynamic friction when their relative tangential velocity is not zero. Dynamic friction again consists of sliding friction and rolling friction. The model we used for taking into account friction treats it in a very simple way that has been shown to yield good results for reasons discussed later.

A fourth force a particle can experience in a collision is a binding force to another particle or to a mesh. In this thesis only mono-sized spherical particles were used with meshes not conducive to binding, so it wasn't a consideration. Considering a particle or granule shape is the best way of dealing with this because in this way particle clumping or binding becomes a natural byproduct of the simulation rather than another imposed fuzzy parameter in the model. One often comes across an attraction parameter in DEM codes that is difficult to justify for uncharged particles. Using the damping forces in both the normal and tangential directions deals with cohesion between spherical particles.

The damping forces in both directions are, of course, calculated as vectors, but have not been represented graphically in Fig 3.3 because they are not physical forces. They are simply mechanisms the model uses to cope with energy lost in the collision. A clearer explanation of this is given when the contact model is laid out in full in the next chapter.

In a collision, a particle experiences a force because its structure is deformed. This is extremely difficult to model and would crush any DEM simulation (by the computational resources required). As a result, particles in a DEM simulation are considered to have rigid structure and are considered to interact with each other directly when their positions partially overlap. This is completely aphysical. The only justification for this is that we must never allow that overlap to become too large. This can also be shown mathematically (this is shown in the next chapter where a contact model is derived). This is judged by treating the overlap as a percentage of the particle size. Particles can overlap each other because given an initial velocity and net force on the particle, its velocity and position are updated by integrating over the timestep. Hence, the particle can move in a timestep to a position where it overlaps another particle. To ensure that this overlap doesn't get to big to preserve the justification of this manner of modeling the force, the timestep has to be kept very small. For the purpose of obtaining accurate physics out of such a simulation, rather than visually acceptable results, the timestep should be kept to the order of a microsecond.

Another large component of a DEM simulation is collision detection. It is very simple to determine whether two spherical particles overlap, but determining all the overlapping pairs of particles in a large simulation is enormously computationally intensive and the task must be treated with care and grace[3]. Detecting a collision between a particle and the geometry, if the geometry is imposed as a mesh (triangle mesh[4]) is more complicated and performance can be totally lost if the problem is attacked incorrectly. Scenario spe-

[3]I use the word "grace" here intentionally. A book could be dedicated to this aspect of any computer code or theory. One ignores grace and beauty at one's peril.

[4]Other meshes exist, such as tetrahedral meshes, but have application to scenarios not considered here

cific symmetries and other such factors can be used to deal with this problem effectively.

For spherical particles, a collision is defined as occurring when two particles' centres are nearer than the sum of their radii. Since mono-sized particles were used in this thesis, individual particle radii did not have to be stored. Support for size distribution is given by including a particle class ID parameter in each particle thus only storing one set of parameters for each particle class rather than for each particle. This is particularly essential when dealing with shaped particles, but allows for interesting and efficient handling of granules - e.g. tetrahedral configurations of spheres. Determining all the pairs of particles that are in contact at a particular timestep is potentially an $O(N^2)$ calculation (as with an N-body simulation) with N being the number of particles in the simulation. Since practically N tends to be greater than 10^5, the brute force technique of collision detection involves more than 10^{10} checks[5], which is computationally unreasonable. In Fig. 3.4, lines represent collision detection checks. However, one needs only to check particles in the near vicinity of the particle for which the collisions are required. This is fair because we do not have long range inter-particle forces.

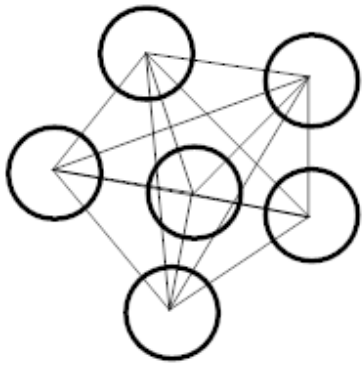


Fig 3.4: Connected graph illustrating brute force collision detection approach

Grid-based methods divide the world space (physical space in which particles can be) into cells and store in each cell the particles with centres in that cell. The search for the particles a particle is in contact with is then limited to only particles in the same cell as the particle in question, as well as particles in neighbouring cells. This reduces the number of collision pair checks to $N \times M$, where M is the number of particles in the neighbouring cells and the particle's cell. Hence, a grid with smaller cells limits the maximum possible value of M. Of course, cells must be larger than the radius of the largest particle to avoid having to check a neighbourhood in the grid of more than 1 cell away from the cell a particle is in. A uniform three dimensional grid is the simplest way to implement this. Fig. 3.5 illustrates a uniform grid with cells the same size as the particles. Lines in the graph indicate collision detection checks.

There are a number of problems with using a uniform rectangular grid. A grid consisting of tiny cells can contain an enormous number of cells. Since the world space a particle can

[5]As can be seen from Fig. 3.4, the number of checks is actually $\frac{N(N-1)}{2}$

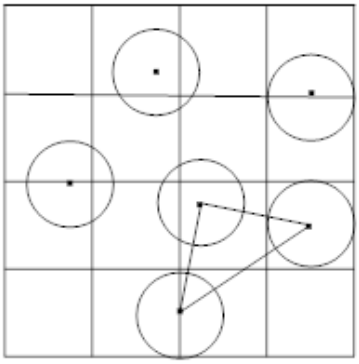


Fig 3.5: Spatial partition using uniform grid with cells of particle size

live in does not take into account boundaries imposed by geometries particles interact with, one can end up with a percentage of cells in one's grid that are never inhabited by any particles. A major problem in implementing a uniform rectangular grid with small cells is that each cell must be stored even if it's empty. Ideas from the theory of handling sparse arrays can be implemented to avoid this, but the added complexity outweighs the benefits achieved. In this work the largest grid size implemented was 128 x 128 x 128 cells (greater than 2 million cells). If one were to implement a grid with half the cell size, one would require storage of over 16 million cells. Memory limitation is a far greater concern on graphics cards than in ordinary RAM (random access memory). When particle size distribution is extreme, the number M can get quite large, even up to N in the extreme case (in which the grid contains just a single cell). Fig. 3.6 demonstrates how doubling the cell size in this example leads to the same number of checks as in the brute force case.

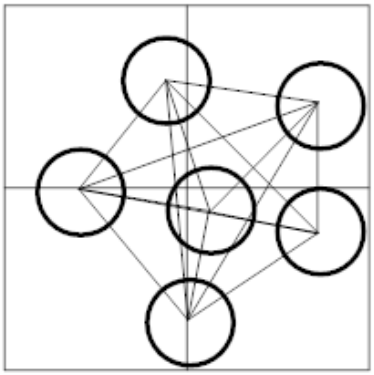


Fig 3.6: Spatial partition using uniform grid with cells of twice the particle size

For this reason, non-uniform rectangular grids may be used. The quadtree [FB74] [Sam84] is a two dimensional implementation and the octree is the three dimensional implementation. These techniques repeatedly subdivide occupied grid cells until finally each cell contains only one particle. This causes a high density of small cells in a grid in regions that are highly populated by particles, and a low density of large cells in regions

sparsely populated. This greatly reduces the number of cells required. The octree implementation of the non-uniform grid-based method described exists [KLS+05]. Fig. 3.7 illustrates the functioning of a 2D octree implementation.

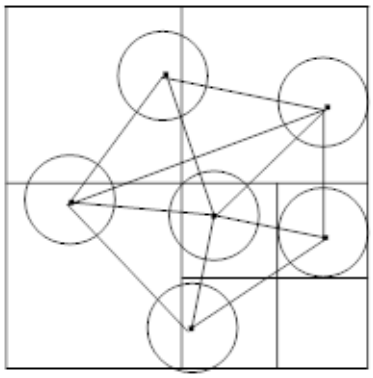


Fig 3.7: Quadtree spatial partition

In this thesis a uniform grid was used because of its simplicity and, for the scenarios investigated, this was proved to be sufficient. The use of a non-uniform grid in an N-body simulation was demonstrated to be very effective when making use of the new computing capabilities released with the Nvidia Kepler architecture supported by CUDA 5. It is convenient to use the same grid when logging the required data from the simulation[6]. A technique that is used to cope with a large size distribution is to superimpose grids of different resolutions on top of each other. This is necessary to keep M small enough while not having to search a larger neighbourhood than 1 cell. As monosized spheres were used in this work, this scheme was not implemented. My feeling is that an octree grid will be more effective.

Using discrete element methods gives one access to information about each particle at every timestep. This can be very useful but, with that, comes the responsibility to collect only useful information. Simply collecting all the possible information from a DEM simulation so that at a later time post processing analysis can be used to extract whatever physical information is required, is simply impossible because of the quantity of data produced and would hinder one so much that the research possible with such a tool would be limited. In this thesis the bare minimum amount of data necessary from each scenario specific simulation was recorded. The assumption was made that the requirements from the simulation are known before the simulation is run. In commercial applications this is not a fair assumption, because a client may easily change the requirements post simulation, and it is worth logging additional data to avoid having to run the simulation again. Data logging, however, if handled gracefully, is not a bottle-neck in the simulation.

[6]It should be noted that we are mixing the two totally different aspects of a computer simulation here but it will be discussed later in the section on data analysis.

3.4 DEM based on shaped particles

Rigid body methods employ undeformable particles with various shapes and structures to construct the situation required. This is a much more physical approach than what has been described before, as real world particles do have shape and facets. However, the assumption that particles are not deformable is suitable to the scenario of modeling a tumbling mill as particles in a tumbling mill are incompressible. The question of breakage is not included. The complications arising from rigid bodies are many. Each particle is allowed to translate in three dimensions as well as rotate in three planes giving six degrees of freedom. This adds significantly to the calculation that determines the effect of a force on such a particle. The representation of each particle is a problem as well as the detection of collisions and the determination of the force experienced during a collision. The two techniques for handling rigid body particle shape are the use of particles with shapes with functional representation, and the use of particles with vertex representation. The techniques each have their own advantages. Functional representation is much faster and allows for much larger simulations. There is a limit on the range of particle shapes that can be formed. The use of superquadrics has been very effectively shown by Cleary [CM11]. The big disadvantage of using particles with functional representation is that if one is modeling particle breakage, one has to use population balancing models (PBMs) to calculate the progeny. With a vertex representation, one can calculate stress lines within the particle, which gives more credibility in the results, as it is a more physical breakage model. Breakage is not considered in this thesis.

In small scale computational simulation environments, rigid body dynamics has been used and implemented in various ways. Mirtich [Mir96] constructed an impulse-based approach that treats rigid bodies physically and is able to model the interactions between these particles and the environment as collisions. However, as it is an impulse approach, it cannot handle static friction (when the relative velocity between two contacting points is zero) and this caused bodies in his simulation to unphysically creep slowly down inclined surfaces. This inability to handle friction makes this approach not useful to the problem of simulating dynamics in a tumbling mill.

However, a rigid body implementation of one thousand non-convex particles was demonstrated by Guendelman [GBF03] that incorporated friction, demonstrated shock propagation and solved the problem rigid body simulations experience of particles "jittering" when at rest. The particle numbers demonstrated are still 2 orders of magnitude below that needed for simulation of tumbling mills to obtain correct physical information, but there is possibility in the future for its use.

In this thesis, the idea of achieving the effect of shaped particles simply by using high friction spherical particles has been investigated.

3.5 GPU-based simulation

Graphics processing units have been used to assist with computation for many decades already, but they became popular in the late 1980s and early 1990s when visually based operating systems appeared. The large number of mathematical calculations required in real time by this, necessitated the development of vector processing computational architectures. Early GPUs acted as 2D display accelerators assisting in bitmap operations - mathematical functions applied to determine the colour value of each pixel displayed. Herein lies the heart of vector processing units. The same set of instructions is applied to many units of data at the same time and very quickly. This will be discussed and explained later in detail. In a discrete element method simulation, thousands of particles undergo the same operation each timestep. The operation consists of the determination of all particles colliding with a particle, it consists of applying external forces to the particle and it consists of updating the position, velocity and angular velocity of the particle based on the force calculated. This is all very different from computing the RGBA (red, green, blue, alpha) value of a pixel, but the idea of using a single instruction on multiple data (SIMD) is there and that is why DEM is so well suited to GPU-based simulations.

Early GPU-based simulations required the problems simulated to be phrased in a manner familiar to the GPU. It is quite a task to phrase a DEM simulation so that it looks to a GPU as if all the GPU is doing is computing the colour value of a pixel. However, when successfully implemented, the performance demonstrated was orders of magnitude higher than CPU-based equivalents. In 2001, with the release of the Nvidia Geforce 3 series of graphics cards and the Microsoft DirectX 8.0 standard that required that pixel shading and vertex pipelines be programmable, this task of phrasing a problem for a GPU became easier, opening the field to a wider variety of problems. However problems still had to be stored in graphics texture memory and manipulation was done through DirectX or OpenGL functions. Programming had to be done in pixel shading languages[7], and minute control of every aspect of the computation had to be controlled by the programmer. Again the performance demonstrated was phenomenal but programs were virtually unmaintainable, inextensible and totally unmalleable because low-level optimisations were so crucial and so scenario specific. Development of such programs was slow and difficult because breakpoint debugging was impossible and great care had to be taken.

The release of the Nvidia programming language CUDA in November 2006, which implemented the DirectX 10 standard, allowed for control of the GPU programmable pipeline through a language like C++. Low-level control is still necessary, as problems have to be transferred to graphics memory, although graphics memory no longer only consists of texture memory and registers as it used to, but now the tools are available to develop useful codes for scientific computing with greater flexibility and ease. Some perfor-

[7]Nvidia Cg language was probably the most popular. Microsoft's HLSL (High Level Shader Language) working in conjunction with the Microsoft Direct3D API (Application Programming Interface) is analogous to GLSL (OpenGL Shading Language) working with the OpenGL standard

mance is lost by this high-level approach to an intrinsically low-level field, but still, as demonstrated in this thesis, orders of magnitude improvements on equivalent CPU-based approaches are achieved. It is good to note that very successful pre-CUDA GPU-based discrete element method simulations have been created [L09]. The constraints on problems imposed by the GPU architecture have relaxed tremendously and in most scientific computing fields application of GPU-based simulations have been demonstrated successfully. The GTC (GPU Technology Conference held annually is the best showcase of this (http://www.gputechconf.com) [SK10].

One major problem is that existing techniques and codes do not port to this architecture. The mindset required for implementing an algorithm and phrasing a problem in terms of a vector processing unit architecture is very different from that required for implementing an algorithm on a CPU or collection of CPUs, be they in the form of a shared memory architecture[8] or be they in the form of a cluster[9].

Another major problem with GPU computing is that until recently debugging capabilities were extremely limited. Breakpoint debugging was impossible because of the nature of the architecture. This problem was for a long time combatted in the inadequate manner of emulating on the CPU what probably would have happened on the GPU if the code in question had been executed. This is very awkward, inaccurate and prone to the creation of difficult to track down bugs. Also that one is emulating the thousands or millions of threads spawned on a GPU serially on the CPU causes execution to be very slow, which can make debugging an enormous problem if much processing is required prior to the piece of code needing to be debugged. Debugging tools, essential for larger computational projects, are now available with the advent of Nvidia's Parallel Nsight[10].

A GPU-based discrete element method simulation was performed on a large number of particles by Kipfer et al. [KSW04] using pixel buffer objects to store particle positions and velocities and these were sent to the GPU to be processed for particle collisions. Harada et al. [HKK07] and Venetillo et al. [VC07] have managed to simulate over one million particles in real time on the GPU with all collision detection, processing and particle updating occurring on the GPU.

One must remember that although these are impressive demonstrations, they are not intended for the field of scientific computing but for the computer gaming industry. This means that physically accurate models can be neglected and the timestep can be much larger as a result. It does indicate, however, that the potential for the GPU-based approach to DEM simulation for scientific purposes does exist and it is this that has been explored in this thesis.

[8] IBM's Blue Gene is a good example of a super computer with shared memory architecture

[9] As of November 2011, the world's most powerful supercomputer was the K Computer at the RIKEN Advanced Institute for Computational Science in Kobe, Japan

[10] http://developer.nvidia.com/nvidia-parallel-nsight

3.6 Summary

Various techniques for simulating the environment of a tumbling mill have been introduced. Each technique has various advantages and disadvantages. The continuum methods lack inter-particle interactions and physical justification. Discrete element methods lack real-world particle representations. Rigid body methods cannot be implemented with large enough particle numbers to be useful. In this thesis the GPU-based approach was used. The uniform rectangular grid-based spatial partitioning technique was implemented. Mono-sized spherical particles were used, and scenario specific symmetries were exploited. The Hertz-Mindlin contact model was implemented to handle the physics of inter-particle collisions and this will now be discussed in detail.

4 A discrete element method model for tumbling mills

The heart of using DEM for modeling real world scenarios is the contact model employed that governs how elements in the simulation interact with each other. As mentioned in the previous section, many contact models have been proposed. They have various degrees of complexity and they are based on physical properties to various extents. The force experienced by each particle is calculated every timestep and this is used to update the velocity of the particle and, during the timestep, it moves accordingly. As a result, particles can come to have their centres nearer than the sum of their radii apart and one finds the situation at $t + \Delta t$ illustrated in Fig. 4.1. Note that this is not the same as what is happening physically (Fig. 3.2(b)).

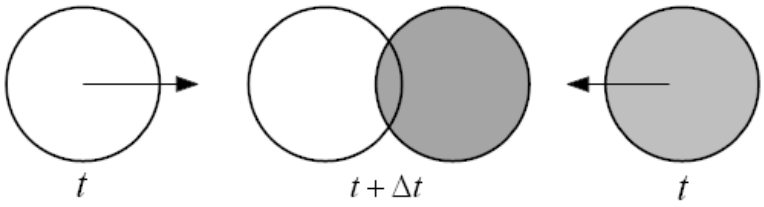


Fig 4.1: Particle overlap as a result of change of position of particles during a timestep

4.1 An introduction to force contact models - example model built from first principles

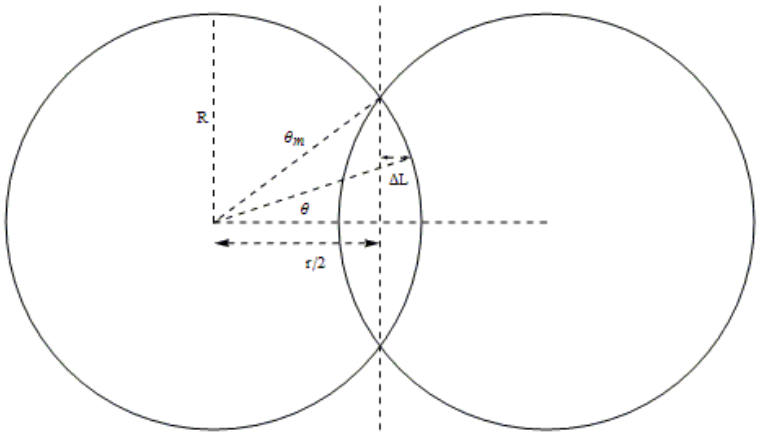


Fig 4.2: Determining force from overlapping particles

No large scale DEM simulation can allow for deformable particles. As a result, a force must be determined from the particle overlap rather than from the particle deformation. At first we attempted to build a model for the normal and tangential forces from first principles using Young's theory. It ended up not being the model used in the final version of the project in this thesis, but it is very illustrative of where the force experienced by the particle comes from. Also one gets a feeling for what in other contact models is based on physical properties, what are calculated coefficients, what are observed or fitted coefficients and what is simply fudge.

In Figure 4.2:

$$\begin{aligned} R &= \text{Particle Radius} \\ \theta_m &= \text{Maximum angle of overlapping region} \\ \Delta L &= \text{Amount element is compressed} \\ r/2 &= \text{Half distance between centres of particles} \\ s &= \text{Distance of compressed element from centre line} \end{aligned}$$

Let us suppose the collision is head on. With E, Young's modulus for compression, the force in the normal direction is given by

$$dF_n = 2E\frac{\Delta L}{L}dA,$$

where dA is the area of the infinitesimal element that is being compressed and contributes dF_n to the force in the normal direction. The factor of 2 comes from the fact that both spheres are being compressed.

$$\Delta L = R\cos(\theta) - r/2,$$

L is the original length of the compressed element.

$$dA = 2\pi s ds,$$

where

$$s = R\sin(\theta).$$

Therefore

$$dF_n = 2E\left(\cos(\theta) - \frac{r}{2R}\right)2\pi R^2\sin(\theta)d\left(\sin(\theta)\right).$$

Integrating:

$$\begin{aligned} F_n &= 4\pi R^2 E\left(\int_0^{\theta_m}\left(\cos^2(\theta)\sin(\theta)d\theta - \frac{r}{2R}\sin(\theta)\cos(\theta)d\theta\right)\right), \\ &= 4\pi R^2 E\left(\left(-\frac{\cos^3(\theta)}{3}\right)\Big|_0^{\theta_m} - \left(\frac{r}{2R}\right)\left(\frac{\sin^2(\theta)}{2}\right)\Big|_0^{\theta_m}\right), \\ &= 4\pi R^2 E\left(\frac{1-\cos^3(\theta_m)}{3} - \left(\frac{r}{2R}\right)\left(\frac{\sin^2(\theta_m)}{2}\right)\right), \end{aligned}$$

and

$$\begin{aligned}\cos(\theta_m) &= \frac{r}{2R},\\ \sin^2(\theta_m) &= 1-\left(\frac{r}{2R}\right)^2.\end{aligned}$$

Thus,

$$\begin{aligned}F_n &= 4\pi R^2 E\left(\frac{1}{3}\left(1-\left(\frac{r}{2R}\right)^3\right)-\frac{1}{2}\left(\frac{r}{2R}\right)\left(1-\left(\frac{r}{2R}\right)^2\right)\right),\\ &= 2\pi\left(R-\frac{r}{2}\right)^2 E-\frac{2}{3}\pi R^2 E\left(1-\frac{r}{2R}\right)^3.\end{aligned}$$

$\left(1-\frac{r}{2R}\right)$ must be small for it to make sense to drop the cubic term, i.e. $r \approx 2R$. In our scenario the maximum particle overlap should never be more than 5% of the particle radius. This is ensured by keeping the timestep low. Therefore:

$$F_n = 2\pi\left(R-\frac{r}{2}\right)^2 E$$

This says that the force in the normal direction is twice the area of the disk of radius $\left(R-\frac{r}{2}\right)$ multiplied by Young's modulus.

The normal direction is defined by the positions of the centres of the particles involved in the collision. The tangential direction, as discussed in the previous section, is dependent on the velocities of the particles involved in the collision as well as the angular velocities of the particles involved in the collision. For calculating the force on a particle, we are only concerned with the relative velocity of the contact point. The velocity of the contact point due to translation of the point is clearly the same as the velocity of the particle but the velocity of the contact point due to the angular velocity is dependent on the angular velocity of the particle as well as the vector connecting the centre of the particle to the contact point. This will be discussed later. It is convenient to define:

$$\begin{aligned}\vec{V} &\equiv \vec{V_1}-\vec{V_2},\\ \vec{\omega} &\equiv \vec{\omega_1}-\vec{\omega_2},\end{aligned}$$

where the subscripts label to particles involved in the collision.

If the collision is not head on, the relative velocity of the two particles is given by

$$\vec{V} = \vec{V_1}-\vec{V_2} = \vec{V_n}+\vec{V_t},$$

where $\vec{V_n}$ is the component of the relative velocity in the normal direction and $\vec{V_t}$ is the component of the relative velocity in the tangential direction. The normal direction is the direction of the vector joining the centres of the two particles, and the tangential direction is the direction of $\vec{V} - \vec{V_n}$ neglecting particle rotation. Note that the definition of the tangential direction in this manner makes it possible to decompose the relative velocity into these two components. Also note that these components have nothing to do with a coordinate frame. The tangential velocity will give a shearing force (associated with Young's shear modulus G) in the tangential direction. Thus, the total force from the collision is:

$$F = 2\pi \left(R - \frac{r}{2}\right)^2 \left(E\frac{\vec{V_n}}{|\vec{V}|} + G\frac{\vec{V_t}}{|\vec{V}|}\right)$$

With this experience we can now look at the most well known and one of the oldest contact models that is still used extensively and very successfully by leaders in the field of discrete element modeling simulations.

4.2 Linear spring-dashpot contact model - Cundall and Strack

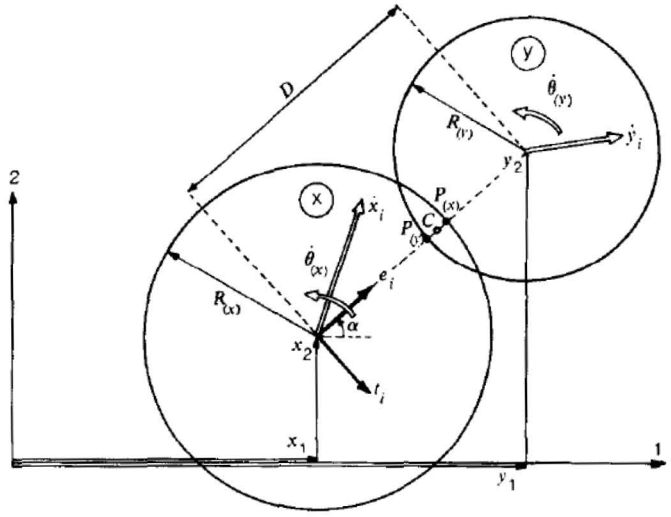


Fig 4.3: Diagram from Cundall and Strack 1979 paper [CS79]

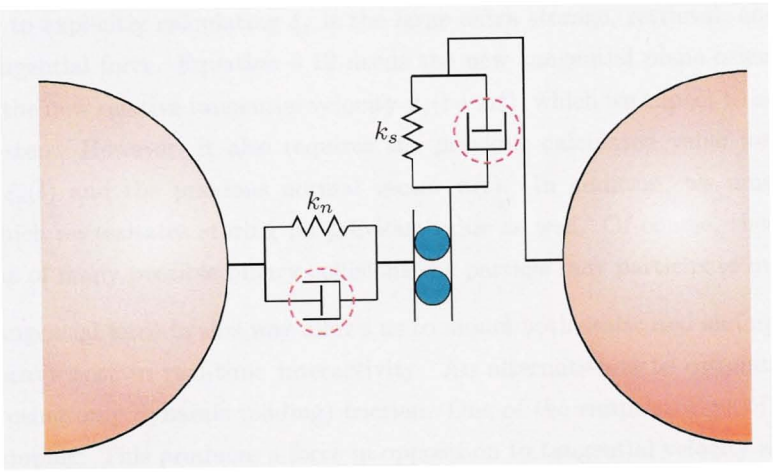


Fig 4.4: Diagram from [L09] illustrating the concept of two spring and dashpot combinations in perpendicular directions combining to model the force experienced by overlap with another particle

Fig. 4.3 is the diagram from the original 1979 paper of Cundall & Strack [CS79]. It is included to give a little bit of an idea how the theory was developed. I will not give their derivation here. Instead Fig. 4.4 is the classic picture to illustrate the concept of the linear spring dashpot model. The spring models the repulsive force caused by the particle overlap (in reality due to its deformation) and the dashpot models the energy lost in the collision. The sum of the forces in the two directions is thought to result in the force experienced by the particle due to the contact. The rings around the dashpots in Fig. 4.4 depict the limit imposed on the compression of the spring. In the normal

direction this limit is to avoid unrealistic forces that would be caused by a large particle overlap (possibly due to a too large timestep that allows particles to move too far in one timestep). In the tangential direction, where μ_k is the coefficient of kinetic friction, this limit is that of sliding friction. The tangential force can never be greater than $\mu_k F_n$. The diagram demonstrates that this "sliding" can occur by the blue rollers.

$$\begin{aligned} F_n &= -k_n\delta_n - c_n\dot{\delta}_n, \\ F_t &= -k_t\delta_t - c_t\dot{\delta}_t, \end{aligned}$$

with k_n (k_t) the spring stiffness in the normal (tangential) direction and δ_n (δ_t) the overlap of the two spheres in the normal (tangential) direction. The damping coefficient is given by

$$c_n = -2\ln(e_n)\sqrt{\frac{1}{\ln^2(e_n)+\pi^2}}\sqrt{\frac{1}{m_1^{-1}+m_2^{-1}}k_n}.$$

k_n is obtained by using a common ration of estimated maximum impact force to displacement. A suggestion by Di Maio and Di Renzo [MR04] is

$$k_n = \left(\frac{320}{81}m^*\dot{\delta}_{n0}^2 E^{*4}R^{*2}\right)^{\frac{1}{5}},$$

where $m^* = \frac{1}{m_1^{-1}+m_2^{-1}}$ is the reduced mass. E^* is the effective Young's Modulus and is a quantity that comes up often either defined as Young's modulus or as

$$E^* = \frac{E}{2(1-\nu^2)},$$

where E is Young's Modulus.

$$\vec{\delta_t} = \int \vec{V_t}dt.$$

Since $\vec{V_t}$ lies anywhere on a plane defined by the normal direction, this integral is not straight forward and it is computationally expensive to solve for $\vec{\delta_t}$ and $\dot{\vec{\delta}}_t$.

$$\delta_t(t+\Delta t) = \delta_t(t) - (\delta_t(t).n\widehat{(t+\Delta t)})\widehat{n(t)} + V_t(t+\Delta t)\Delta t,$$

is the recurrence relation derived by Fazekas [Faz07]. The implementation of this is costly, computationally, because it requires one to store and retrieve the normal vector and tangential overlap vector because these are needed in the calculation of the next step. The extra calculations required to compute the tangential force in this manner also come at a price when dealing with large scale simulations.

An alternative solution to the problem of friction is to handle friction, both static and kinetic, by means of viscous damping:

$$\vec{F_t} = \gamma_t \vec{V_t},$$

where γ_t is the viscous damping coefficient. In order to ensure that the tangential force never exceeds the kinetic frictional force one must force this by:

$$\vec{F_t} = \begin{cases} -\mu F_n \widehat{V_t} & \text{if } \gamma_t V_t < \mu F_n \\ -\gamma_t \vec{V_t} & \text{otherwise} \end{cases}$$

This does not take into account static friction which can be seen because when $V_t = 0$, $F_t = 0$ which is incorrect. A proposed solution [L09] to solving the problem of incorporating static friction into the simulation is to use shaped particles or granules made of clusters of spherical particles. As such shaped particles are likely to naturally mesh together, this solution was demonstrated to be effective. Cluster logic and the complications it entails, is discussed later in this chapter.

Experience working with the linear spring dashpot contact model has indicated that it is sensitive to the coefficients as well as to the incoming particle velocities. Use of this model requires one to zero the velocity of particles that exhibit behaviour outside the range of possible values. In this thesis, the Hertz-Mindlin contact model was used because it has been suggested to calculate the force on a particle better and the problem of instabilities was not encountered.

4.3 Non-linear spring dashpot model - Hertz-Mindlin

The Hertz-Mindlin contact model, coming out of elastic theory [LL86], gives the normal component of the force as:

$$F_n = c_n \delta_n^{\frac{3}{2}},$$

where

$$c_n = \frac{4}{3} E^* \sqrt{R^*},$$

where R^* is the effective radius and E^* is the effective Young's modulus.

$$\begin{aligned} R^* &= \left(\frac{1}{R} + \frac{1}{R} \right)^{-1} = \frac{R}{2}, \\ E^* &= \left(\frac{1-\nu^2}{E} + \frac{1-\nu^2}{E} \right)^{-1} = \frac{E}{2(1-\nu^2)}, \\ G^* &= \left(\frac{1-\nu^2}{G} + \frac{1-\nu^2}{G} \right)^{-1} = \frac{G}{2(1-\nu^2)}. \end{aligned}$$

Young's modulus is related to the material property, Young's shear modulus (G), by Poisson's ratio (ν) in the following way

$$E = 2(1+\nu)G.$$

The damping force in the normal direction, which accounts for some of the energy absorbed by the particle in the collision as well as some of the energy lost to the environment during the collision, is related to the restitution coefficient (ϵ) of the material. It is given by Tsuji et al. [TTI92] with slightly different coefficients as:

$$F_n^d = d_n \delta_n^{\frac{1}{4}} * V_n,$$

where [PWC+11]

$$d_n = 2\sqrt{\frac{5}{6}} \frac{-\ln(\epsilon)}{\sqrt{(\ln(\epsilon))^2 + \pi^2}} * \sqrt{2E^* \sqrt{R^*} M^*}.$$

where M^* is the effective mass of the two particles involved in the collision.

This non-linear damping term was observed by Zhang and Whiten [ZW96] to make the model more realistic. The coefficient d_n used here, coming from [PWC+11], differs from

the original by a factor of 2. In the tangential direction, the repulsive force is given by [Cle98] [MR04(2)]:

$$F_t = c_t\sqrt{\delta_n} * \delta_t,$$

where [PWC+11]

$$c_t = 8G\sqrt{R^*},$$

and the very unphysical quantity δ_t is defined as

$$\delta_t = V_t\Delta t.$$

The damping force in the tangential direction is given by:

$$F_t^d = d_t(\delta_n)^{\frac{1}{4}} * V_t,$$

where [PWC+11]

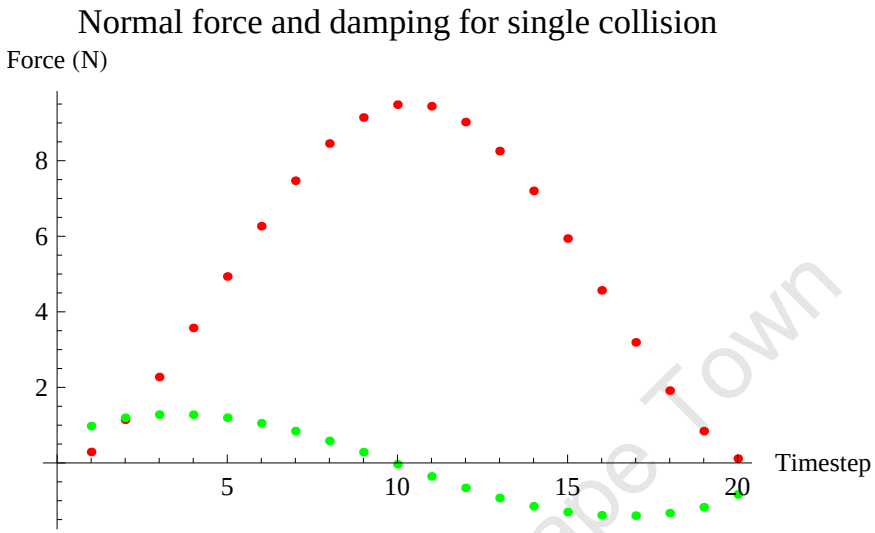
$$d_t = 2\sqrt{\frac{5}{6}}\frac{-\ln(\epsilon)}{\sqrt{(\ln(\epsilon))^2+\pi^2}}\sqrt{8G\sqrt{R^*}M^*}$$

In this model the tangential force must be limited by friction, as in section 4.2, but in this model we do not have a viscous damping term. Instead:

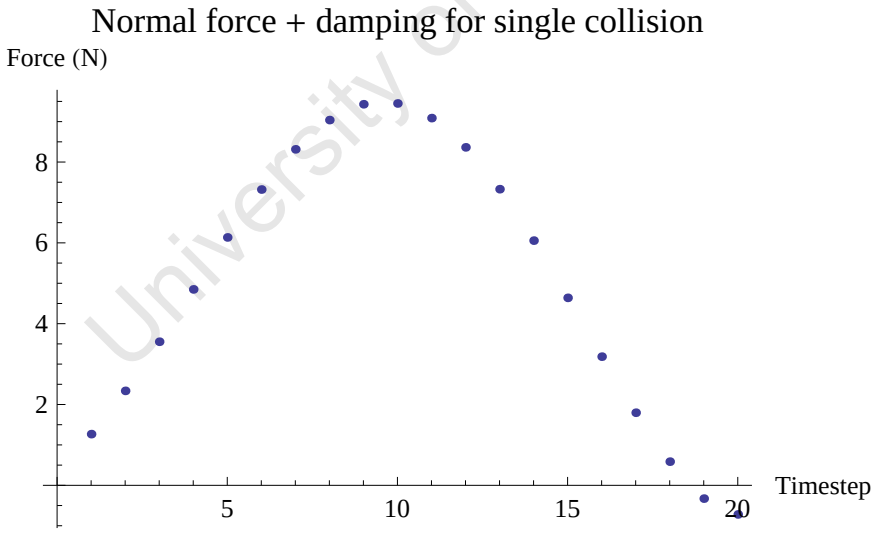
$$\vec{F_t} = min\left\{\mu F_n, c_t\sqrt{\delta_n} * \delta_t - d_t(\delta_n)^{\frac{1}{4}} * V_t\right\}\widehat{V_t}$$

One problem with this model, as well as with the linear spring-dashpot model, noted by Zhang and Whiten [ZW96], was that before particles separated, the force changed direction. This is totally unrealistic. In a controlled numerical programming implementation of this theory, we made the same observation. Fig. 4.5(a) shows the modeled spring force in red and the modeled dashpot force in green. Since the scenario modeled here was 1 particle falling vertically from initially at rest three particle radii directly above a second, but fixed, particle, the sign of the force indicated its direction. As one would expect, the spring force always acts in the positive direction (up), and the dashpot acts in the opposite direction to the direction of motion. Since DEM computes the force at each timestep, the graph cannot be continuous. As can be seen from the graph, the particles were in contact for 20 timesteps ($20 * 10^{-6}$ s). Fig. 4.5 (b) shows

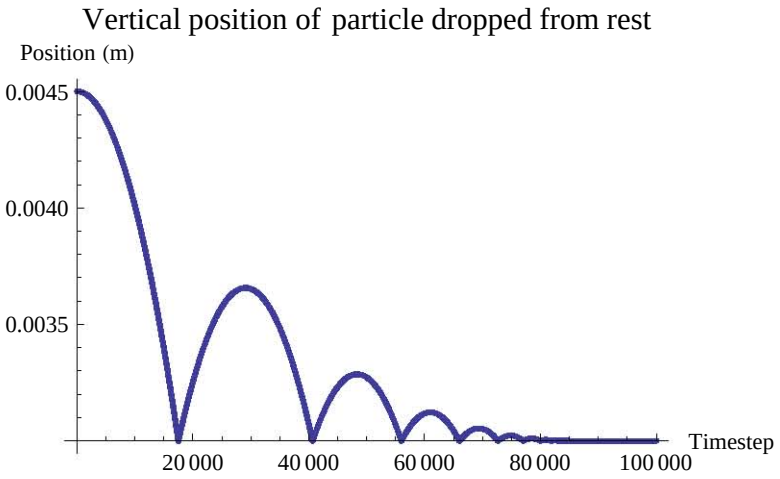
these forces added together. The point to note is that at the end of the collision, more specifically the last two timesteps before the collision ended, the total force experienced by the particle is negative (hence pointing down)! This means that the model is causing the particles to be attracted to each other. Fig. 4.5(c) and (d) give the vertical position and velocity of the particle over a period of 10^5 timesteps or 0.1s.



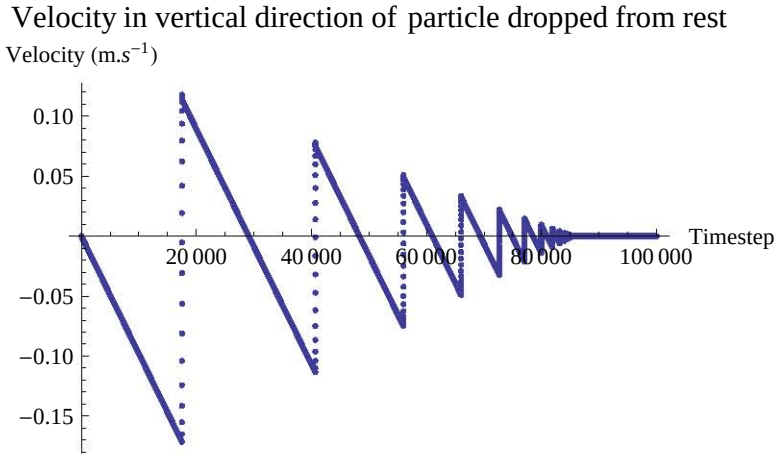
(a) Normal (red) and dashpot (green) forces



(b) Total force experienced by particle



(c) Vertical position of particle



(d) Vertical component of velocity of particle

Fig 4.5: Hertz-Mindlin contact model modeling a single collision

A number of other modifications to this model have been proposed. Hunt and Crossley [HC75] proposed $F_n^d = \lambda \delta_n^p \dot{\delta}_n$ for the normal damping term. I will not explain this model, but have included it to give an idea of a possible modification to the contact model used.

4.4 Rotation

Rotation introduces the third property that one must store for each particle in the simulation. It can be dealt with in a simple manner for the case of spherical particles. When clusters of particles are used (see the next subsection on cluster logic) care must be taken regarding the configuration of the cluster and it is convenient to use a configuration where the moment of inertia is independent of the axis of rotation of the grain. A

tetrahedral configuration is an example of this. However, for the spherical particle case:

$$\begin{aligned}
\vec{\tau} &= \vec{r_1} \times \vec{F} = F_t \vec{r_1} \times \widehat{V_t} \\
\vec{\tau} &= I\vec{\alpha} \\
\vec{V_t} &= \vec{V} - \vec{V_n} + \vec{\omega_1} \times \vec{r_1} - \vec{\omega_2} \times \vec{r_2} \\
I &= \frac{2}{5} M R^2
\end{aligned}$$

where $\vec{\tau}$ is the torque on particle 1 by particle 2, F_t is the force on particle 1 in the tangential direction (hence responsible for the torque experienced), I is the moment of inertia of a sphere and $\vec{r_1}$ and $\vec{r_2}$ are vectors joining the centre of particle 1 and 2 respectively to the central point of contact. This point is midway between the centres of the particles.

4.5 Rolling resistance

Rolling resistance is an uncomfortable property to model from physical properties. It is based on particle shape as well as material properties. The omission of rolling resistance from the contact model results in aphysically high angular velocities in particles. Particles that are rolling down a plane have a zero relative velocity of the contact point and hence no force is observed in the contact model above. This is best handled by using granule shaped particles. In this thesis, however, the following model for rolling resistance was used [ZY03]:

$$\tau_{rolling} = \mu_R * min(|F_n|, \omega_{rel}) \widehat{\omega_{rel}},$$

where μ_R is the coefficient of rolling friction.

4.6 Integration

The simplest possible integration scheme, the forward Euler integration scheme, was found to be sufficient because the timesteps were kept very small. It is called explicit because the properties $\vec{v}(t+\Delta t)$ and $\vec{r}(t+\Delta t)$ can be determined from properties already known. Acceleration is simply determined from Newton's second law:

$$\vec{a} = \frac{\vec{F_{net}}}{m} + \vec{g}.$$

where F_{net} is the force on a particle due to all the collisions it is involved in at a specific timestep. Integrating gives velocity and position:

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \vec{a}(t)\Delta t,$$

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t)\Delta t.$$

Similarly integrating the angular acceleration gives:

$$\vec{\omega}(t + \Delta t) = \vec{\omega}(t) + \vec{\alpha}\Delta t,$$

where $\vec{\alpha}$ is determined as shown above.

The backward Euler integration scheme is unconditionally stable.

$$\vec{a}(t + \Delta t) = \frac{\vec{F_{net}}(t + \Delta t)}{m} + \vec{g}$$

Integrating gives velocity and position:

$$\vec{v}(t + \Delta t) = \vec{v}(t) + \vec{a}(t + \Delta t)\Delta t$$

$$\vec{r}(t + \Delta t) = \vec{r}(t) + \vec{v}(t + \Delta t)\Delta t$$

Similarly integrating the angular acceleration gives:

$$\vec{\omega}(t + \Delta t) = \vec{\omega}(t) + \vec{\alpha}(t + \Delta t)\Delta t$$

However, this is an implicit method, as the equation needs to be solved using it's solution. A technique for doing this is to use a Taylor expansion to estimate this implicit function [PTV+92]. This adds complexity to the computation and is not justified unless the timestep is allowed to be larger. A larger timestep would result in a much faster simulation because few timesteps would be required to simulate the amount of time required to be simulated, but a larger timestep would also cause the maximum possible particle overlap to be larger, which would invalidate the contact model used. For this reason, it is safe and practical to use the explicit forward Euler integration scheme.

4.7 Cluster logic

Binding spherical particles together is the easiest way of obtaining shaped particles. This does not reproduce the properties that a rigid body DEM would possess, but properties such a grains binding together, are reproduced nicely. This is implemented by treating each constituent sphere in each grain individually and, as in the case of simple spherical particles, computing the force on each particle but then summing the forces on each particle in a grain to obtain the force on the grain to determine the linear acceleration. The resultant torque on the grain is determined by summing the torques (as vectors) on each constituent particle. The moment of inertia tensor, for example, of a tetrahedral configuration of particles is:

$$I(0) = \begin{pmatrix} \frac{1}{20}MR^2 & 0 & 0 \\ 0 & \frac{1}{20}MR^2 & 0 \\ 0 & 0 & \frac{1}{20}MR^2 \end{pmatrix}.$$

The moment of inertia tensor for a grain at time t can be determined as follows:

$$I(t)^{-1} = R(t)I(0)^{-1}R(t)^T,$$

where $R(t)$ is the rotation matrix that would transform the orientation of a grain in its original state to that at time t. The angular acceleration of the grain is then determined as follows:

$$\vec{\alpha} = I(t)^{-1}\vec{\tau},$$

and then the grain's angular velocity can be updated:

$$\vec{\omega}(t + \Delta t) = \vec{\omega}(t) + \vec{\alpha}(t + \Delta t).$$

The rotation matrix, R, must be stored for each grain and be updated each timestep.

4.8 DEM technique

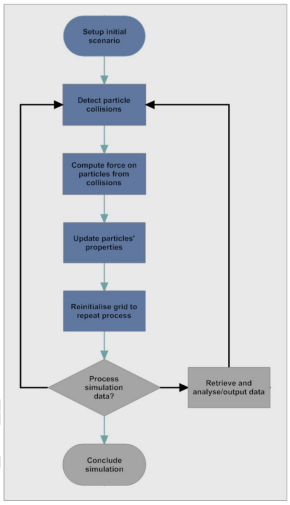


Fig 4.6: Simulation procedure. The decision whether to process simulation data at a particular timestep affects the speed of the simulation and the statistics of the data obtained.

Fig. 4.6 gives the steps in a DEM simulation. The collision detection scheme used has been outlined in the previous section, and the implementation is detailed in the section on algorithms. The contact model is used to compute the forces on the particles. One generally chooses to process the simulation data every 1000 timesteps for a number of reasons including the fact that it is generally unnecessary to do so more often. A GPU implementation of this scheme limits how often data should be retrieved from the simulation by the length of time it takes to retrieve the grid from the graphics card. This limit has no practical implication. Retrieving and processing too much data can give one a false sense of accuracy because of the enormous number of events processed, but one must remember always the real time period simulated and it is important to get statistics over a large enough, and correct, period. Hence one can see that the first two processes in Fig. 4.6 are where the vast majority of the computation takes place.

4.9 Summary

In the contact models considered, the forces experienced in a collision are modeled by overlapping particles rather than stresses due to particle deformation. In the linear spring dashpot model, using viscous damping instead of computing the tangential overlap and the rate of change of this, is a more practical implementation, especially for GPU programming, and thus the forces are modeled by:

$$\begin{aligned} F_n &= -k_n\delta_n - c_n\dot{\delta}_n \\ \vec{F_t} &= \begin{cases} -\mu F_n \widehat{V_t} & \text{if } \gamma_t V_t < \mu F_n \\ -\gamma_t \vec{V_t} & \text{otherwise} \end{cases} \end{aligned}$$

In the Hertz-Mindlin contact model the forces are modeled by:

$$\begin{aligned} F_n &= c_n\delta_n^{\frac{3}{2}} - d_n\delta_n^{\frac{1}{4}} * V_n \\ \vec{F_t} &= min\left\{\mu F_n, c_t\sqrt{\delta_n} * \delta_t - d_t(\delta_n)^{\frac{1}{4}} * V_t\right\}\widehat{V_t} \end{aligned}$$

In both contact models the normal direction is defined as the vector connecting the particles' centres. The tangential direction is defined by the relative velocity of the point midway between the centres of the colliding particles. This is computed by:

$$\vec{V_t} = \vec{V} - \vec{V_n} + \vec{\omega_1} \times \vec{r_1} - \vec{\omega_2} \times \vec{r_2}$$

5 GPU Implementation and Algorithms

In this chapter, a strategy and its realisation are laid out for performing a large scale DEM simulation with the bulk of the computation done on a graphics card. This implementation is limited by the available memory on the graphics card, as it is an implementation for only one graphics card. It can easily handle particle numbers up to 2 million particles, although the largest simulation run in this thesis is ca. 303 000 particles. The reason for this is that simulations were required of a pilot scale 300mm diameter tumbling mill containing spherical 3mm diameter glass beads. This is the scenario that was run experimentally in the PEPT experiment described in section 6.2. To introduce the idea of using a graphics card to solve physics problems, more specifically to perform large scale simulations, it is worth motivating why such an idea is attractive.

5.1 A motivation for developing a GPU based DEM code

5.1.1 Introduction

Graphics processing units are designed for a specific purpose, namely to make the rendering of a scene possible and fast enough to have it look smooth. To do this, at least 30 frames must be rendered every second. When one considers the complexity of scenes that are rendered in realtime, this is not an easy task, and it is not a task that a CPU could perform. A CPU is simply not built for the purpose. A CPU must be able to run as flexible a computer code as an operating system, requiring it to be able to execute a variety of instructions from brute force heavy-duty calculation (e.g. testing a memory bank or indexing a database) to very flexible codes (e.g. recursive codes). GPUs get their power from massive parallelisation and very high memory bandwidths. The idea is that since a limited set of instructions is going to be executed, relatively simple stream multiprocessors, containing multiple scalar processors each, are able to perform them and many can be grouped together onto a GPU chip. Fig. 5.1 illustrates the different concept in the design of a GPU as opposed to a CPU. The green squares represent arithmetic logic units [NV08].

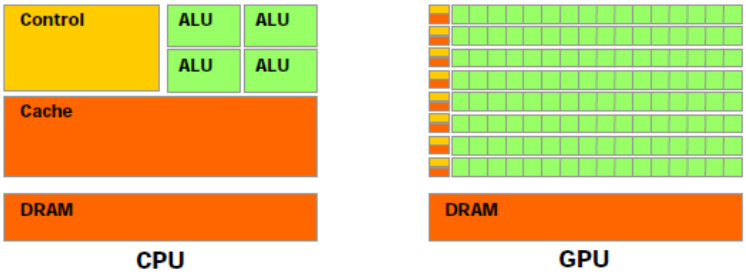


Fig 5.1: CPU and GPU use of arithmetic logic units (ALUs)

5.1.2 Technical overview - processing and memory management

The ability GPUs have of being able to handle thousands of threads concurrently is made possible not only by the large number of scalar processors, but by an architecture called SIMT (Single Instruction Multiple Threads). Up to now I have referred to GPUs as part of the family of vector processing architectures. This is not strictly correct. The multiprocessor SIMT unit groups threads into warps of 32 threads. There is a deterministic scheme for splitting blocks of threads into warps and each thread in a warp starts execution at the same instruction. Each thread is mapped to a scalar processor core that is equipped with its own instruction register and registers. Each scalar processor core handles execution and branching behaviour of its thread independently. This is where it differs from SIMD (Single Instruction Multiple Data) vector processing architectures. With SIMD, software has to control branching operations as each thread in a warp must perform the same operation on the data in its registers. With SIMT it is possible to write thread-level parallelisation [NV08]. This difference is very subtle and can be totally ignored, but it can also be utilised through low-level optimisations to greatly improve performance. Each stream multiprocessor handles thread spawning, thread scheduling and execution and thread destruction in the hardware at zero overhead cost. This is why it is possible to spawn thousands of threads on a GPU, while the thread management on a CPU-based program would crush such an attempt. Fig. 5.2 is an illustration of a stream multiprocessor on an Nvidia GeForce 8800GTX. It consists of 8 scalar processors (SP), 2 special function units (SFU), a 16K bank of software controlled shared memory, a multithreading issuing unit, an instruction cache, and a constant cache [Liu09].

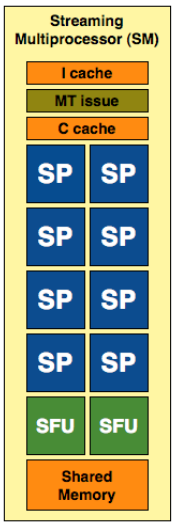


Fig 5.2: Logical illustration of a stream multiprocessor

The way memory is accessed on a GPU is another crucial component that allows GPUs to be so effective. Fig 5.3 is a logical illustration of the memory layout on a GPU [Liu09(2)]. A GPU has no direct access to RAM and only communicates with GPU memory. This

forces one to copy all data required from RAM (random access memory) to GPU memory before any processing can be done. One also has to copy data back from the GPU after processing has been done. This is the largest overhead cost of using GPU computing. However, memory bandwidths between the GPU and GPU memory are an order of magnitude faster than between CPU and RAM, so if copying between RAM and GPU memory can be limited or can be performed in the background, then the high GPU memory bandwidth can boost performance tremendously. This high memory bandwidth is due to a number of reasons. GPU memory is split into a number of categories with different properties. Device memory is accessible by all cores and contains the operands as well as results of computations performed. It is to device memory that one must transfer data from RAM. This is where the limit on the simulation size is imposed. If the number of particles in the simulation requires more memory than device memory can offer, then part of the particle grid would have to be stored in RAM and be transferred to device memory each timestep. The overhead of this transfer would nullify any performance gained by using the GPU.

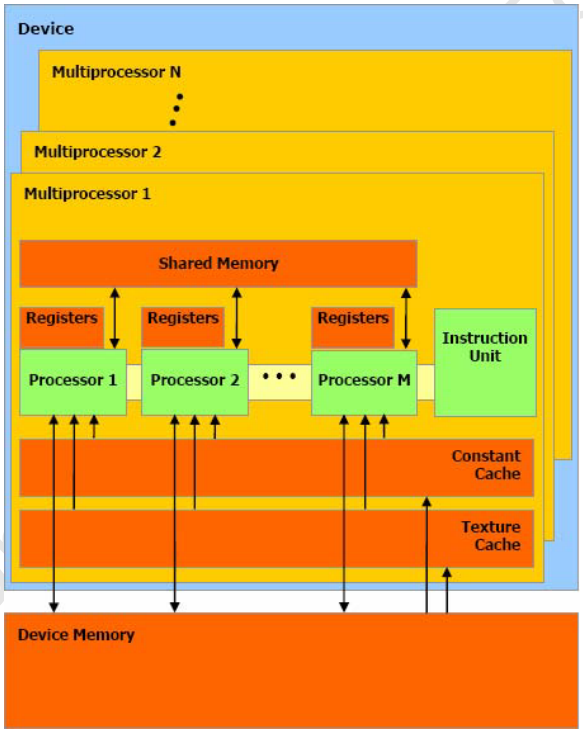


Fig 5.3: Logical illustration of a memory management in a GPU

The main memory spaces in device memory that one should be aware of are global memory, texture memory and constant memory. Device memory has the highest latency for access by a thread on a scalar processor on a stream multiprocessor. Texture memory offers threads readonly access, but has the advantage that it is cached in dedicated[11] texture cache that is loaded using a 2D block (see Fig. 5.3) so if threads on a single SM access data in texture memory that is located 2D spatially together, the reads are as

[11]The advantage of having a dedicated cache is that it is less likely to suffer from cache ousting

fast as a read from cache (i.e. 1 clock cycle latency). Global memory has a latency of ca. 500 clock cycles. Constant memory is limited to 64K and offers readonly access by threads. It is used to store parameters that are constant and are required by each thread (e.g. particle normal stiffness). Reads from constant memory are cached in the constant cache on the SM. Memory accesses by threads to shared memory can be performed simultaneously if either accesses are to different physical memory banks that the shared memory is divided into or if accesses are all to the same address (broadcasting). The largest consideration that must be made by the programmer is that memory accesses should be coalesced wherever possible. This makes use of the functionality that GPU memory architecture provides that allows memory transactions of 32, 64 or 128 bytes to be performed as a single memory transaction if memory transactions are naturally aligned, meaning that its address is a multiple of its size [NV10]. To take full advantage of this, thread memory accesses should be designed to require that much data, otherwise unused data is transferred anyway and throughput is reduced. This means that all threads can access in one memory transaction up to 128 bytes each because the data will be mapped across available memory banks to make use of the ability to access different memory banks simultaneously. This is the heart of where the enormous memory bandwidth comes from. Another feature of memory access on a GPU is that global memory can access data types of 1, 2, 4, 8, or 16 bytes in a single memory instruction and so it is useful to restrict one's data types to such sizes. This will be commented on when the data storage system for the simulation is discussed.

5.1.3 Performance

The GPU used in this thesis was an Nvidia GeForce GTX560 Ti, which is the desktop version of the Nvidia Fermi architecture. Note this is not a compute card[12]. Specifications for this GPU are given in Figure 5.4. The Fermi architecture is different to previous architectures in that it has inherent double floating point precision support (along with support for CUDA version 4). The code in this thesis was written for CUDA 4.1 and uses double floating point precision.

A comparison between a high-end CPU on a high-end system and a high-end GPU is shown in Fig. 5.5. The peak performance is given for single precision floating point operations. This figure can roughly be halved for double precision floating point operations since the introduction of intrinsic 64 bit support on both CPU and GPU architectures. The performance quoted in GFLOPS means the number of billion floating point operations per second that can be performed. Each operation consists of a multiply, then an add and a special function (e.g. a*b+c). The special function could represent transcendental operations. The type of memory used in a Xeon system (server) is specified

[12]A compute card is a graphics card designed for computing rather than for aiding graphics rendering. It is able to run in TCC (Tesla Compute Cluster) configuration using the TCC driver. Compute cards have more double precision floating point processing capability and more emphasis on strictly preserving accuracy in calculations.

# streaming multiprocessors	8
# CUDA cores	384
# registers per SM	32768
# threads per block	Up to 1024
# threads per grid	Up to 65535 in each dimension
Global memory	1GB
Constant memory	64KB
Shared memory per SM	48KB
Peak	1262.592 GFLOPS
Memory bandwidth	128.256 GB/s

Fig 5.4: Specification of an Nvidia GTX 560 Ti

as ECC (error correcting codes). This is because it has built in hardware controlled error detection and correction algorithms. This is standard on server platforms. Another point to note is that DDR3 RAM peak memory bandwidth drops when more than 48GB of RAM is installed.

	CPU Intel Xeon Processor E7-2803	**GPU** Nvidia Tesla M2090
Launch date	2nd quarter 2011	2nd quarter 2011
Number of cores:	6	512
Clock speed	1.73GHz	1.3GHz
Peak Performance:	41.592 GFLOPS	1331 GFLOPS
Cache:	18MB	limited
Memory:	32*4 GB	6GB
Type:	DDR3 1333MHz ECC	GDDR5 136-pin BGA
Peak memory bandwidth:	32GB/s	177GB/s

Fig 5.5: Comparison of specifications between that of a CPU and GPU

5.1.4 Summary

The way CPUs and GPUs perform computation is quite different and clearly there is strong motivation for attempting to use GPUs in the field of high performance computer simulations. The raw processing power offers possibilities of orders of magnitude improvement in performance. However, the mindset when approaching the task of designing and implementing a scheme to solve a problem such as implementing a large scale DEM simulation, is very different. Low-level technical considerations are important. In the earlier section on processing and memory management, it was mentioned that certain technical features of the chip architecture could be safely ignored. However, low-level optimisations in GPU programming can result in performance improvements easily up to 30x. In this thesis, CUDA was used as it controls the GPU and allows one to treat it as an independent processing unit without having to worry too much about

how it performs the computation. The advantages of assuming full control without any abstraction are definite and an example is the fact that the rendering can be included in the same pipeline, hence adding little cost to performance [L09]. For our purposes this is definitely a future consideration because getting a good and detailed visual picture of different aspects of the simulation as it is in progress can allow one to adjust parameters on the fly and get a feeling for the processes at play. However, abstraction is also important as it allows one to produce flexible and extendible code. As a last note in this section, intended to give a small window into what GPU computing entails, I will propose that CUDA offers the right balance between hardware optimisations (it is specific to Nvidia GPU architectures, unlike the cross platform equivalent OpenCL[13]), and high-level abstraction.

[13]Open Computing Language (OpenCL) must not be confused with OpenGL, the rendering language.

5.2 Algorithms

Fig. 5.6 gives a detailed view of the steps the DEM simulation implemented was broken into. The light coloured boxes occur on the CPU and the dark coloured boxes occur on the GPU. The entire simulation loop occurs on the GPU. Even the rendering is merged into this, which is natural, since the particle system has to be stored on the GPU and it is stored in a manner that makes rendering natural. This brings us to the most fundamental difference in approach between a CPU based DEM simulation and a GPU based DEM simulation, which is the manner in which data is stored.

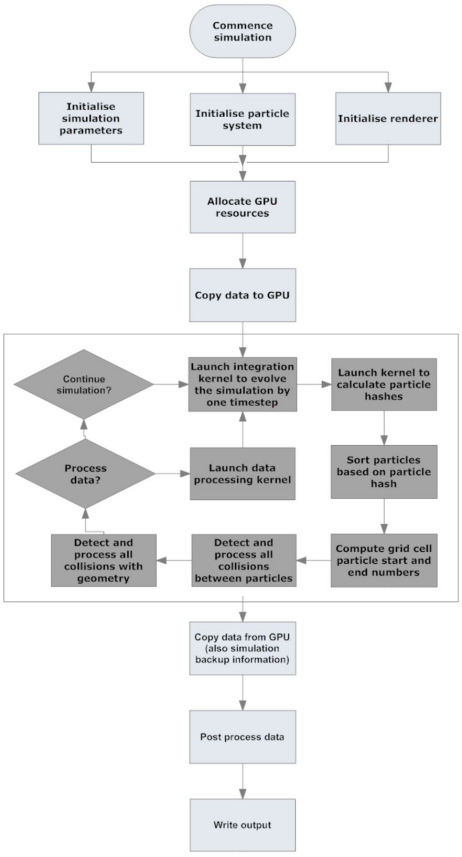


Fig 5.6: Flow chart identifying processes in the simulation to be examined

5.2.1 Particle system initialisation

The traditional way to store particles in a DEM simulation is to use a structure or object that contains the properties of the individual particle and create as many particles as are required by the simulation. Such an approach, though well suited to a CPU based simulation, is not effective for a GPU based simulation. It implements the idea of object orientation, which tries to make code much clearer by splitting it up into units

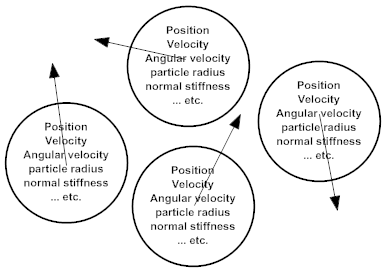


Fig 5.7: Object oriented approach to DEM data storage

Position	Velocity	Angular velocity
x^1	v_x^1	ω_1^1
y^1	v_y^1	ω_2^1
z^1	v_z^1	ω_3^1
α^1	α^1	α^1
x^2	v_x^2	ω_1^2
y^2	v_y^2	ω_2^2
z^2	v_z^2	ω_3^2
α^2	α^2	α^2
...	...	...
x^N	v_x^N	ω_1^N
y^N	v_y^N	ω_2^N
z^N	v_z^N	ω_3^N
α^N	α^N	α^N

Fig 5.8: DEM data storage technique for GPU based simulation

that represent real-world objects and having them interact with each other in a natural way. Fig. 5.7 illustrates this approach.

As explained briefly in the previous section, GPU algorithm performance profits greatly from keeping the data types to the sizes of 1, 2, 4, 8, or 16 bytes and storing it in a naturally aligned manner. It is difficult to make the object oriented data storage model comply to this, allowing the enormous GPU memory bandwidth to be utilised, so the idea must be discarded completely. Better is to store all the particles' positions (3-vectors), for example, in a 1D array of length $4N$, where N is the number of particles. Every 4th element of this array seems to be unnecessary but must be there to assure memory alignment. One tends to find this value extremely useful for storing identifying data about the individual particle and since these lists of particle properties are sorted every timestep, this is the simplest way of keeping track of a single particle. Fig. 5.8 shows the arrays used to represent all the particles.

This technique has the major disadvantage that particles cannot be added to, or removed from, the simulation without rebuilding all the arrays with different dimensions since reallocation of graphics memory is not supported. Since GPU resources are allocated

at the initialisation phase, this would require reinitialisation of the particle system. A technique commonly used to initially place the particles inside the simulation world is to add them gradually during the initial phase of the simulation. The idea behind this is that adding them all at once could cause particles to have large overlaps with each other which would cause the contact model to breakdown. The way to implement this on a GPU would be to label the particles that haven't yet been added to the simulation as ghost particles (easily done using the fourth spare parameter each particle carries) and bring them into existence at the correct time (this would mean data structures would not have to resize) but this is clumsy. Better is to initialise the particles to unique, non-overlapping positions. For monosized spherical particles this is very easy to implement. A standard grid packing would ensure this, with the additional constraint being that the particles must all be inside the mill which lives inside the simulation world. However, the packing density for grid packing is

$$\frac{4/3\pi r^3}{8r^3} = 0.52,$$

and since for grid packing it is natural to build the particles initially as a block, not a cylinder which is the shape of the mill, it can be difficult to position all particles required inside the mill. Hexagonal close packing is known to be the densest packing of monosized spheres [WeiHCP] and was used to find the initial particle positions. It is possible to implement it in the initialisation of the particle system, but this has drawbacks. In this thesis, tumbling mills with lifters were simulated and mills with drum liners (i.e. no lifters) were also simulated. The initial particle positioning routine must be careful not to overlap a particle's position with the geometry of the mill. To ensure this and to keep the method general, it was decided that particle positions would be read in from a binary file and be determined elsewhere along with the geometry.

In hexagonal close-packing (HCP) and face-centered cubic (FCC) packing each sphere is in contact with 12 other spheres. Both configurations achieve a packing density of

$$\frac{\pi}{3\sqrt{2}} = 0.74.$$

It was suggested to be the densest packing of monosized spheres by Kepler in 1611 [WeiKC] but proved only in 2005 by Hales [Hal05]. The HCP configuration was selected for our purpose because in it alternate layers are identical instead of every third layer, as in FCC. The following code listing (Fig. 5.9) gives a one line implementation of HCP in the programming language Mathematica version 7.0. The variable, *centres*, is an array containing the position of the centre of each sphere in the HCP that is within the boundaries specified by, {xWidth, yWidth, zWidth}. These boundaries determine the number of particles that can fit in each direction in a rectangular prism inside which the mill would fit. Of course this generates far too many spheres, some of which are outside the mill, but as this computation is run only occasionally, it was not worth improving it. From the 2D array, *centres*, the required number of spheres fulfilling the criterium that they must all be inside the mill geometry without overlapping it anywhere are selected. To ensure that particles do not overlap with each other at all, even due only

```
centres = Flatten[Table[
   If[Mod[k, 2] == 1,
   {j*2 r - r*Mod[i + 1, 2], r Sqrt[3]/3 + r*(i - 1)*Sqrt[3], r},
   {j*2 r - r*Mod[i, 2], r + r*(i - 1)*Sqrt[3], r}] +
   {0, 0, 2 r (k - 1) Sqrt[6]/3},
   {i, xWidth}, {j, yWidth}, {k, zWidth}],
   2]
```

Fig 5.9: Mathematica implementation of HCP sphere configuration.

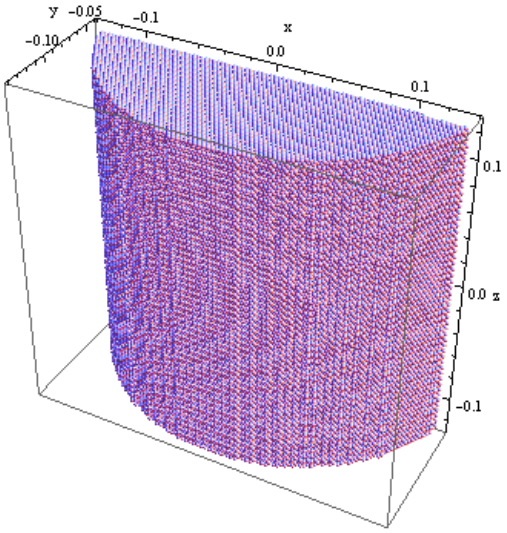


Fig 5.10: Initial positions of particles - 253125 particles

to numerical precision, the value, r, is set to be 2% larger than the size of the particle simulated. This provides a cushion of air around each particle.

Algorithm 5.1 gives the pseudo-code for the algorithm determining the initial positions of the particles inside a mill without lifters. When outputting computed particle centres to file, a fourth parameter was added to each position and it was written to a binary file in the format given in Fig 5.8 so that a single DMA memory read could be used to initialise the positions of the particles without delaying the program. Particles are initialised with zero velocity and zero angular velocity. Fig. 5.10 shows the output from this procedure. Layers of close packed spheres are added from the bottom of the mill until the required number of particles is obtained - in this case 253125 particles. The reason for this number will be discussed later.

Part of the particle system initialisation is the initialisation of the geometry of the mill. There are a number of techniques that can be used to do this. A triangle mesh can be used to describe almost any geometry nicely. The resolution of the mesh (density of triangles in the mesh) determines how realistically it represents what it is modeling.

Algorithm 5.1 Pseudo-code for algorithm using HCP to determine initial position of particles

procedure HEXAGONAL CLOSE PACKING
  $r \leftarrow 1.02 * R$ ▷ Particles get 2% padding
  $xWidth \leftarrow \left\lfloor \frac{drumRadius-\epsilon}{r} \right\rfloor * 2$
  $yWidth \leftarrow \left\lfloor \frac{drumRadius-\epsilon}{r} \right\rfloor * 2$
  $zWidth \leftarrow \left\lfloor \frac{drumLength-\epsilon_2}{r} \right\rfloor$
  for $k = 1$ **to** $zWidth$ **do** ▷ Dimension down length of drum
    for $j = 1$ **to** $yWidth$ **do** ▷ Vertical direction
      for $i = 1$ **to** $xWidth$ **do** ▷ Horizontal direction
        if $k \bmod 2 = 1$ **then** ▷ if k is odd
          if $i \bmod 2 = 0$ **then** ▷ if i is even
            $center[i, j, k] \leftarrow \{j * 2r - r, r\sqrt{3}(i - \frac{2}{3}), r\}$
          else ▷ if i is odd
            $center[i, j, k] \leftarrow \{j * 2r, r\sqrt{3}(i - \frac{2}{3}), r\}$
          end if
        else ▷ if k is even
          if $i \bmod 2 = 0$ **then** ▷ if i is even
            $center[i, j, k] \leftarrow \{j * 2r, r(1 + \sqrt{3}(i - 1)), r\}$
          else ▷ if i is odd
            $center[i, j, k] \leftarrow \{j * 2r - r, r(1 + \sqrt{3}(i - 1)), r\}$
          end if
        end if
        $center[i, j, k] \leftarrow center[i, j, k] + \{0, 0, 2r(k - 1)\frac{\sqrt{6}}{3}\}$
      end for
    end for
  end for
  $centre \leftarrow centre - middle$ ▷ centres particles at {0,0,0}
  $particleCentres \leftarrow$ Select N elements in $centres$ that are within the drum
end procedure

GPUs are designed to process triangles, as most rendering of complex scenes requires the rendering of large meshes at high speed. For example, models of >40 000 triangles are commonly used in computer games. Although not necessary to the specific objectives explored in this thesis, a simple technique for handling triangle meshes was incorporated into the simulation as its development will clearly be necessary in the future, for example for lifter design research, simulating other types of mills, and including end features into the mill geometry.

The common SVG (Scalable Vector Graphics) format for representing meshes is completely unsuitable to our purpose, as is the CASE file format employed with DEM packages like EDEM [EDEM] and Ensight [CEI]. These formats are used because gradient meshes and diffusion curves can easily be stored in them, but they are clumsy, slow and very large. The solution we employed is the cleanest format with the least redundancy and is used by the AGEIA physics-engine programming API, PhysX, which was bought out by Nvidia and been incorporated into the CUDA framework. Fig. 5.11 gives a description of this format. The first two numbers are the numbers of vertices in the model (N_v) and the number of triangles (N_t) in the model. Then follow N_v lines each containing the {x,y,z} coordinates of each vertex in the model (floating point numbers). Then follow N_t lines each containing the indices of the points comprising the triangle (integers). Fig 5.12 is a simple example of how a box would be represented in this format. Note that no information is redundant or repeated in this format.

Figure 5.13 shows the full mesh model of a 12cm diameter pilot mill with 12 lifters and a length of 12cm. Notice that the mesh is totally enclosed, which is a requirement for example for EDEM. This means that every face has two sides and a thickness. The number of triangles in this model is 384 and the number of vertices in this model is 146.

However, it is unnecessary for us to strictly observe the rules of a triangle mesh. For our purposes, we know that the end faces of the mill are featureless and flat. We also know that there are no interactions with anything but the inner surface of the mill. Hence the mesh can be reduced to Fig. 5.14. In this model, the number of triangles is 96 and the number of vertices is 96.

The model in Fig. 5.15 was used to simulate a pilot scale mill with radius 15cm and length 27cm. This is a mill for which we have experimental data from PEPT experiments. One notices that Fig. 5.15 does not show a triangle mesh! This is because in this case, there is no information gained by keeping the drum as a 3 dimensional model. There is nothing in this model that changes with the z parameter that runs along the axis of the drum, except the end walls, but those are dealt with explicitly and separately, as is explained later. As a result, all that is necessary is the front face, which consists of 259 points and 259 lines. Fig. 5.15 is just to visualise the mill represented by its front face.

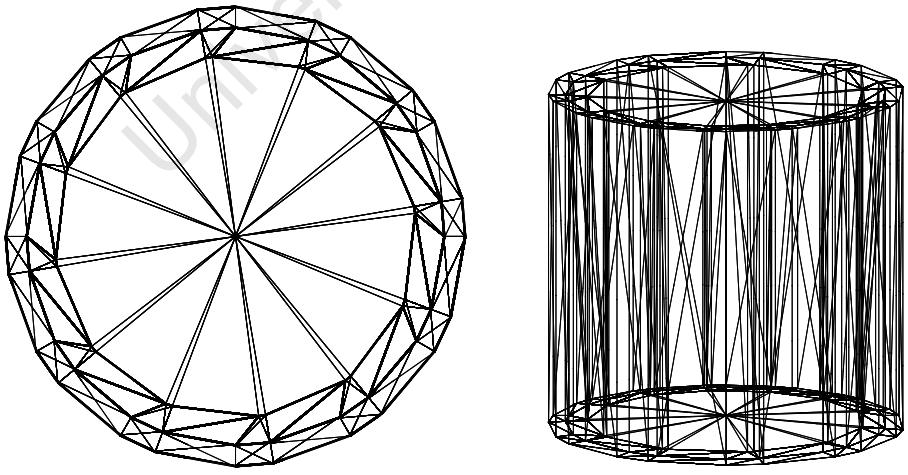
Another technique of handling geometries is to use a mesh to particle conversion. The idea is that enough particles close enough together form a surface that can model the

$$
\begin{array}{ccc}
N_v & N_t & \\
v_x^1 & v_y^1 & v_z^1 \\
v_x^2 & v_y^2 & v_z^2 \\
\ldots & \ldots & \ldots \\
v_x^{N_v} & v_y^{N_v} & v_z^{N_v} \\
t_{v1}^1 & t_{v2}^1 & t_{v3}^1 \\
t_{v1}^2 & t_{v2}^2 & t_{v3}^2 \\
\ldots & \ldots & \ldots \\
t_{v1}^{N_t} & t_{v2}^{N_t} & t_{v3}^{N_t}
\end{array}
$$

Fig 5.11: Description of format for triangle mesh representations of geometries

```
8   12
-1  -1  -1
-1  -1   1
-1   1  -1
-1   1   1
 1  -1  -1
 1  -1   1
 1   1  -1
 1   1   1
 1   2   3
 2   3   4
 1   2   5
 2   5   6
 1   3   5
 3   5   7
 5   6   7
 6   7   8
 3   4   7
 4   7   8
 2   4   6
 4   6   8
```

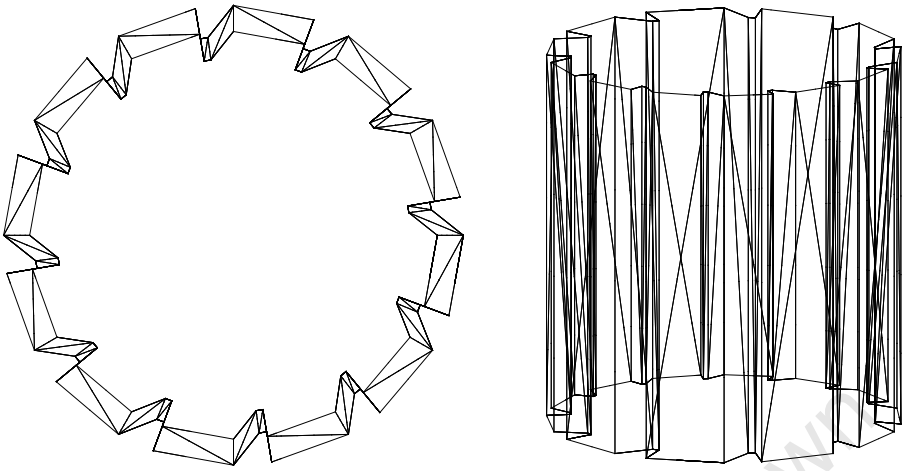
Fig 5.12: Example of triangle mesh file format, storing the data of the mesh of a box



(a) Right View

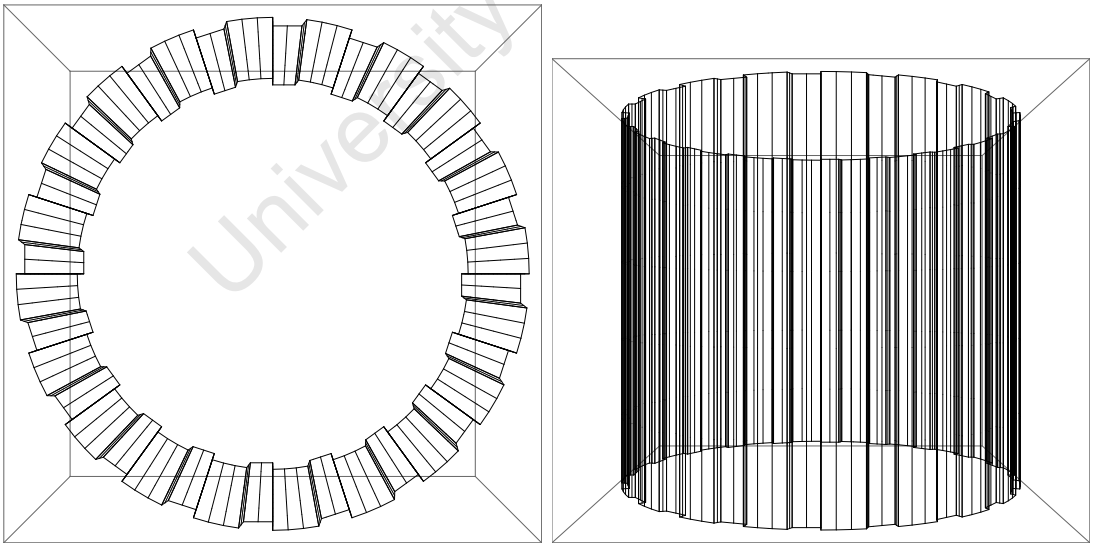
(b) Top View

Fig 5.13: View from top and right of triangle mesh model of a 12cm x 12cm x 12cm pilot scale tumbling mill with 12 lifters



(a) Right View (b) Top View

Fig 5.14: Reduced triangle mesh model of the mill in Fig. 5.13, preserving only necessary information.



(a) Right View (b) Top View

Fig 5.15: Model of 30cm diameter pilot scale mill, length 27cm, used in this thesis for the largest simulation runs.

required geometry. These particles are then identified differently to the free particles in the simulation. They experience forces in the same way as free particles, but their motion is updated based on the motion of the object they are part of. That object in turn, experiences a force based on the forces experienced by the particles it is made of. A simple example of this is cluster logic and the idea of granules as opposed to particles, discussed in section 4.6. Usually this technique is used because including the additional particles forming the geometry is computationally much cheaper than including meshes in the simulation. In this work, mesh to particle conversion was implemented for a different reason. The contact model for determining the force on a particle in a collision has parameters based on the materials of the colliding objects. Thus one must take into account two sets of parameters, as can be seen in Fig 5.18. This effect complicates matters when it comes to theoretically modeling the processes inside the mill using granular flow theory as it is not included in the models. The conversion of the geometry to a cluster of particles removes this aspect from the DEM to allow for a more direct comparison. In our case, one does not need to be concerned about the force experienced by the mill (geometry) causing it to accelerate. Thus we can update the position of the particle comprising the geometry each step irrespective of the forces on these particles. The drum is rotating around its axis ($\{0,0,1\}$) centred at the origin, so updating these particles simply involves rotating them around the origin in the X-Y plane. The timestep is very small, however, and this results in infinitesimal rotations every timestep. There are major numerical problems associated with infinitesimal rotations, especially when using single floating point precision, because $\sin(\theta) \approx 0$. Thus it is better to keep the position of the particles fixed and each timestep temporarily rotate them into the correct position based on the total angle through which the drum has rotated. The technique used here was to label the fourth component of position with a 1 if the particle is part of the geometry and with a 0 if it is free.

The algorithm for calculating the positions of the particles that make up the curved surface of the mill (without lifters) is given in pseudo-code in Algorithm 5.2. Again hexagonal close packing was used to produce the effect of lining the inner surface of the drum with a dense mat of particles. Looking at the drum from the front view one sees the picture like in Fig. 5.16(a). The diagram on the right is exaggerated to illustrate the algorithm for finding the centres of the spheres.

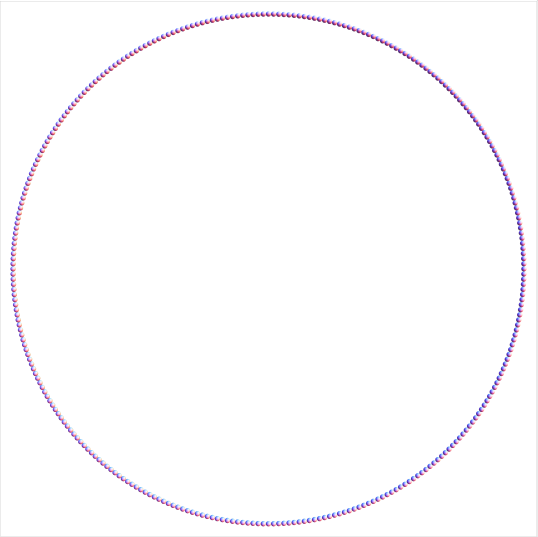
From Fig. 5.16(b) it is clear that

$$\sin(\theta) = \frac{r}{R-r}.$$

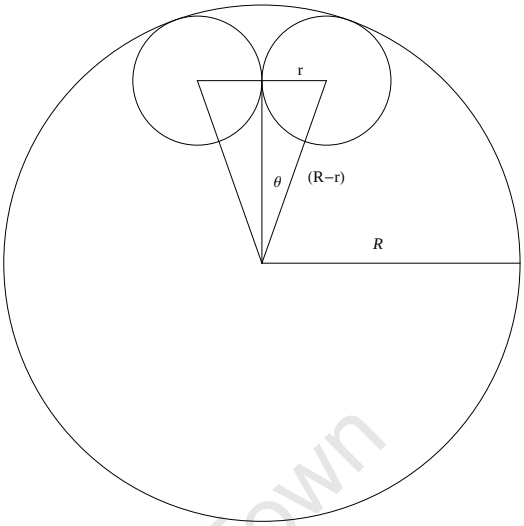
Hence the number of spheres in the circle is:

$$numCirc = \frac{\pi}{\arcsin\left(\frac{r}{R-r}\right)},$$

and they are evenly spaced at a radius $(R-r)$ from the origin. The result is Fig. 5.16(a). Subsequent layers in the direction down the axis of the mill are shifted $r\sqrt{3}$ units per layer and alternate layers are rotated by an angle of $\arcsin\left(\frac{r}{R-r}\right)$.

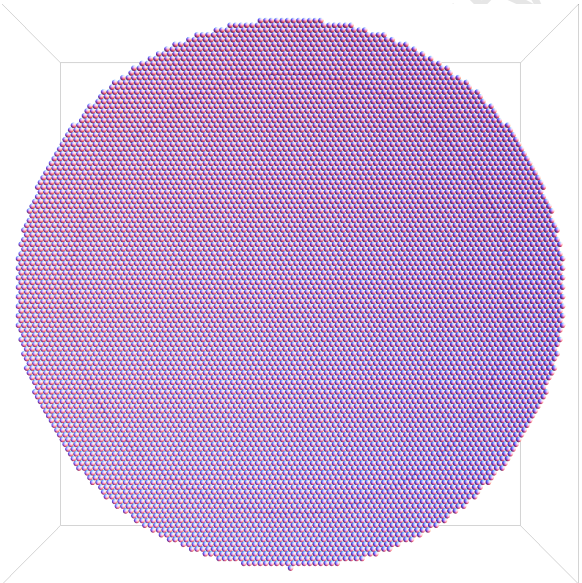


(a) One layer of packed spheres on curved face of drum

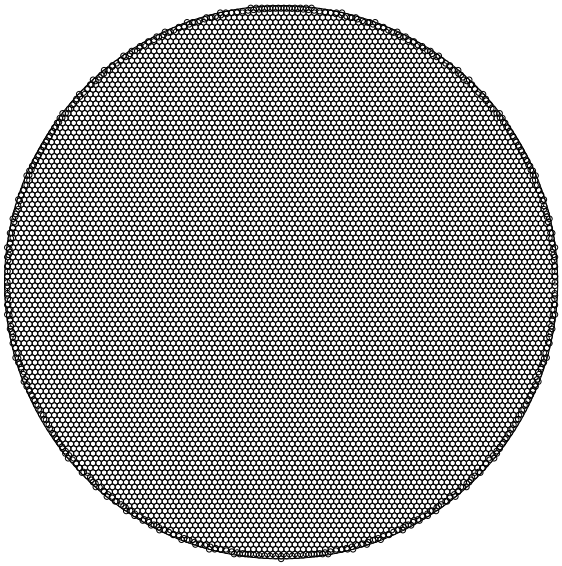


(b) Diagram illustrating algorithm

Fig 5.16: Front view of curved face of mill converted to HCP spheres. Diagram to illustrate algorithm.



(a) Drum face composed of HCP spheres



(b) Diagram illustrating algorithm

Fig 5.17: Mesh to particle conversion of end faces of the mill

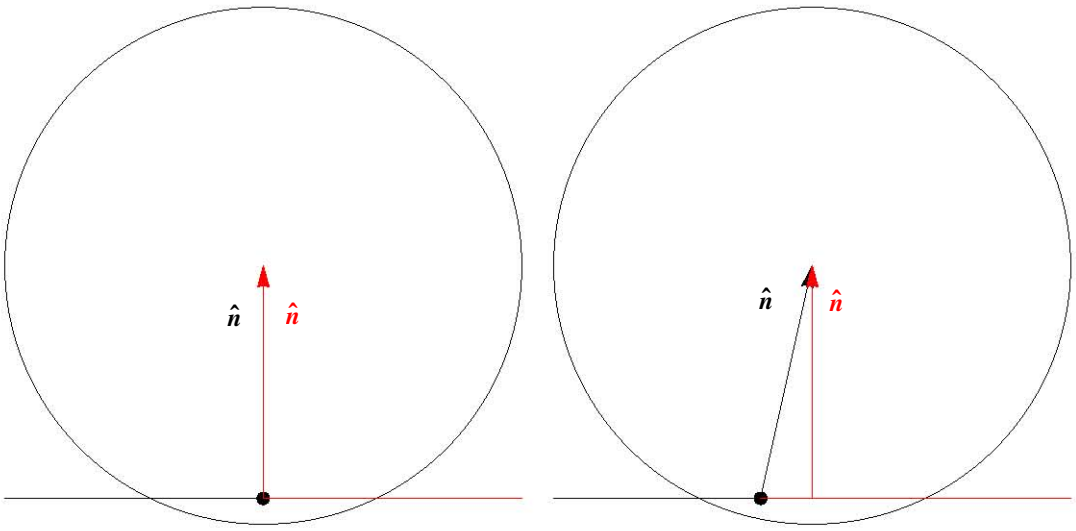
Algorithm 5.2 Pseudo-code for algorithm using HCP to determine position of particles composing the curved face of the mill

procedure HEXAGONAL CLOSE PACKING
  $numCirc \leftarrow \frac{\pi}{\arcsin\left(\frac{r}{R-r}\right)}$
  for $z = 0$ **to** $\left\lceil \frac{drumLength-2r}{r\sqrt{3}} \right\rceil$ **do**
    for $\theta = 0$ **to** $2\pi - \frac{2\pi}{numCirc}$ **do**
      if $z \bmod 2 = 0$ **then** ▷ if z is even
        $\theta_1 \leftarrow \theta$ ▷ do not rotate layer
      else
        $\theta_1 \leftarrow \theta + \arcsin\left(\frac{r}{R-r}\right)$ ▷ rotate layer
      end if
      $centre \leftarrow \{(R-r)\cos(\theta_1), (R-r)\sin(\theta_1), -\frac{drumLength}{2} + r(1+z\sqrt{3})\}$
      $\theta \leftarrow \theta + \frac{2\pi}{numCirc}$ ▷ Increment θ by $\Delta\theta$
    end for
    $z \leftarrow z + r\sqrt{3}$ ▷ Increment z by Δz
  end for
end procedure

To convert the end faces from a flat surface to a tightly packed layer of spheres is not straight forward. The best source of data on dense packing of monosized circles in a circle seems to come from Specht [Spe11]. In our scenario, we have $N \sim 18000$ and the problem of finding the best packing is not trivial. The solution proposed here is to take a square sheet of HCP spheres and to select the particles that have at least some part within the circular end face of the mill. The pseudo-code for this algorithm is given in Algorithm 5.3. The result is Fig. 5.17. Fig. 5.17(a) shows the packed particles, but (b) shows the quality of the packing. One notices that particles overlap each other and overlap the edge of the mill, but this is not a problem because mesh particles do not experience any forces. Any inconsistency outside the mill is also of no relevance as no free particles ever leave the mill.[14]

In this work, there was no dependency of the geometry on the z parameter excluding the end faces of the mill. The end faces are treated separately and can be controlled by the desired boundary conditions, be they continuous or simply a flat wall, or some kind of screen. This allows one to convert the triangle mesh representation of the geometry to a 2D profile of the mill. Algorithm 5.6. is simplified to simply calculate the closest point on a line segment to a point. As with the 3D case with triangles, there are complications when a line segment partially overlaps with the 2D projection of a

[14]Hopefully! No, DEM simulations with large timesteps have difficulty with particles passing through bounding geometries or other particles. The same can occur for particles with extremely high velocities, but the stability of the implementation proposed in this thesis has been demonstrated (see maximum overlap of particles in the Results section). For DEM simulations that require large timesteps, different approaches such as CCD (Continuous Collision Detection) must be implemented. The AGEIA PhysX physics engine implemented this specifically for dealing with high velocity particles in computer games (bullets!).



(a) Equally overlapping 2 line segments (b) Unequally overlapping 2 line segments

Fig 5.18: Issues with wall logic.

particle. Current DEM codes do not take this factor into consideration. The result is that a particle subjected to a simple drop test directly on the join of two parallel line segments experiences twice the force and twice the damping than a particle dropped on a single line segment. It is simple to compensate for this in the case where the particle's centre is directly over the join as in Fig 5.18(a). However, when overlap with one line segment is not the same as the overlap with the adjoining segment (Fig 5.18(b)), it is difficult to calculate what proportion of each force should be used. One must bear in mind that in this case, the force is not in the same direction from each line segment. In the case where the nearest point on a line segment is the endpoint of the line, it is interesting to note that the normal direction is not the normal to the line segment, but the direction from the endpoint to the centre of the particle [PFC3D].

A solution we propose to this problem is to make the force proportional to the area of the segment of the projection of the particle marked off by the line segment. Thus the force is computed as if the projection intersected the line defined by the line segment but force applied is a fraction of this force. This fraction is determined by the area marked off by the line segment divided by the area marked off by the line. This solution is a sensible way to deal with the force experienced at the corners of the mesh, irrespective of how sharp they are. The calculation is given by the Mathematica code in Fig 5.19. with the answer given. The parameter a is defined as the percentage of the line segment overlap to the overlap of the full line. R in this code is the radius of the particle, and as the fraction being computed is independent of this value, the limit can be taken with $R \to \infty$ resulting in the very nice closed form solution:

$$p = a^2(3 - 2a),$$

which is plotted in Fig 5.20.

```
centres = t1 = Integrate[Sqrt[R^2 - (x - 1/2)^2] - Sqrt[R^2 - 1/4], x];
t2 = t1 /. x -> 0;
t3 = t1 /. x -> a;
t4 = t1 /. x -> 1;
t5 = t3 - t2;
t6 = t4 - t2;
percent = FullSimplify[t5/t6];
The result is:
(-Sqrt[-1 + 4 R^2] + 4 a Sqrt[-1 + 4 R^2] +
   Sqrt[-1 - 4 (-1 + a) a + 4 R^2] -
   2 a Sqrt[-1 - 4 (-1 + a) a + 4 R^2] +
   4 R^2 (-ArcCot[Sqrt[-1 + 4 R^2]] +
      ArcTan[(1 - 2 a)/
       Sqrt[-1 - 4 (-1 + a) a + 4 R^2]]))/(2 (Sqrt[-1 + 4 R^2] -
     4 R^2 ArcCot[Sqrt[-1 + 4 R^2]]))
```

Fig 5.19: Solving fraction of force by partial overlap of particle with a line segment.

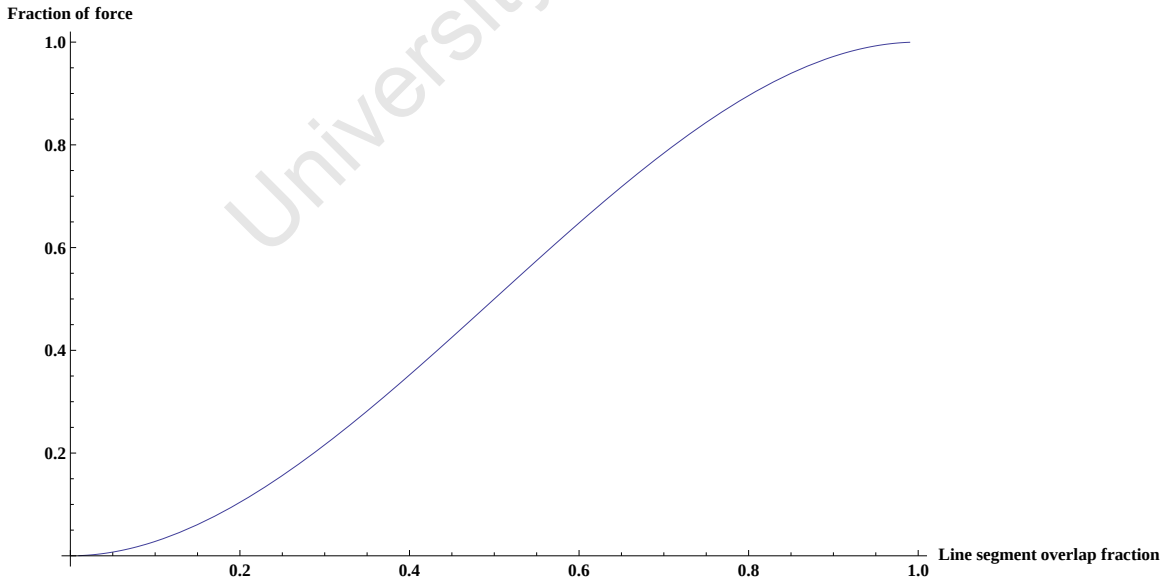
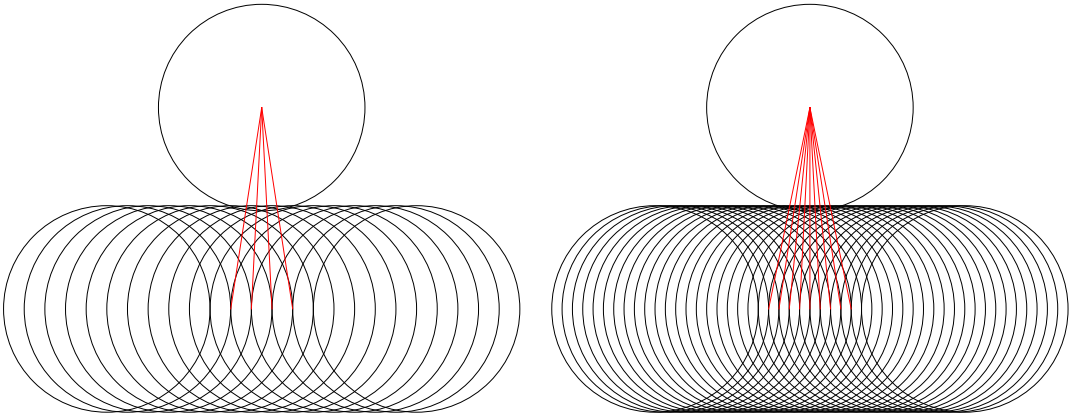


Fig 5.20: Percentage of force to be applied as a function of line segment overlap

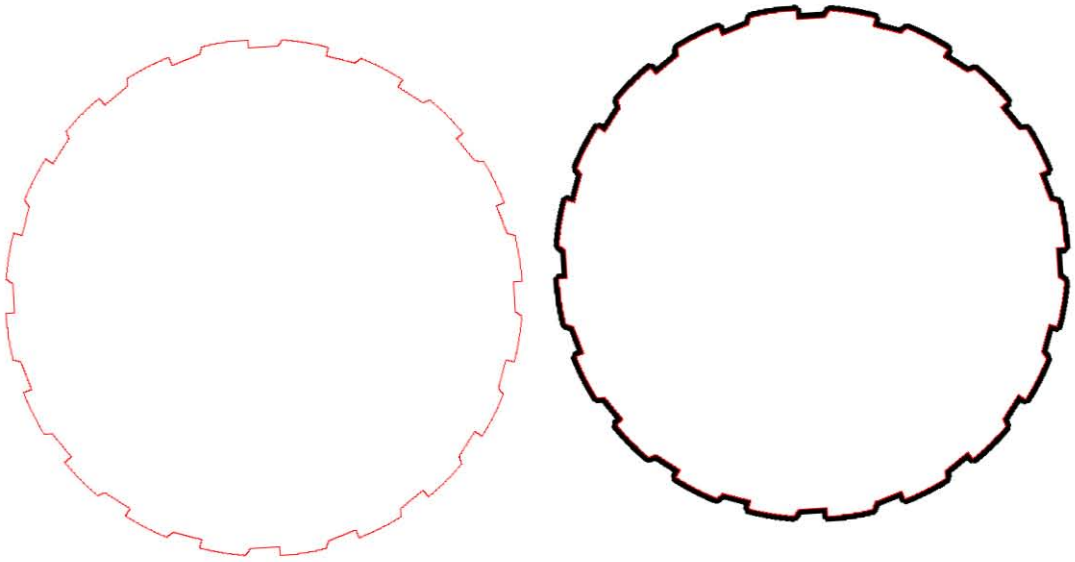


(a) Mesh particles 0.2r apart (b) Mesh particles 0.1r apart

Fig 5.21: Mesh built of particles.

The particle representation of the 2D projection of the mesh was implemented in this work as well. This scheme has the advantage again that it's much faster to run, eliminates sharp edges in the mesh, which is more accurate, as sharp edges in a real mill would be worn slightly smooth very quickly, and as a result, removes the problems associated with sharp edges in the contact model. Two ideas were implemented on how to create the particle representation. The first was to use small particles that just touch each other. This approach gives a relatively good representation of the geometry if the mesh particles are small enough. However, what was noticed was that changing the particle size had a big influence on the results of a simple drop test. This is because the contact model is dependent on the effective radii, R^*, and the effective mass, M^*, of the two particles involved in the collision. This is a point that is not addressed in the literature. In a collision with a wall, what is the mass of the wall! The second implementation was to build the wall out of particles of the same size as used in the simulation but by overlapping them with each other. Fig 5.21(a) and Fig 5.21(b) show the interactions that a particle colliding with a flat wall made of overlapping particles undergoes. The red lines indicate the direction of the normal force exerted by each wall particle in contact.

As is obvious, the closer the mesh particles are, the flatter the surface they present to the incoming particle, but the greater the number of interactions processed, which brings us to a similar problem as encountered when using line segments. Scaling the force in inverse proportionality to the density of the mesh particles proved to give reasonable stability to a simple drop test indicating that this approach is physically acceptable. In this work, a mesh particle density of 10% of the particle radius was used. Fig 5.22(a) shows the profile of the mill used, made up of 739 points, and Fig 5.22(b) shows its conversion to a mesh. In Fig 5.22(b) the set of red particles is simply the set of mesh particles rotated.



(a) Mill profile (b) Mill profile made of particles

Fig 5.22: Mesh to particle conversion.

5.2.2 Simulation parameters initialisation

When the simulation begins, the simulation parameters that govern the simulation are initialised. These parameters are modifiable during the simulation and that ties in with the third initialisation process shown in Fig. 5.6, which is the initialisation of the renderer. Very little attention was given to the rendering on the simulation in this project because ensuring the correct physics was the focus of the work, but it is clear that a high quality renderer would give the user the power to fine tune the simulation parameters on the fly, which would be very useful. This will certainly be a future development of the work. Fig. 5.23 is a table of the simulation parameters. The coefficients are calculated from the physical properties of the setup simulated and Fig. 5.24 gives a table of these properties. The use of pp indicates particle-particle interaction, while ps indicates particle-geometry interaction in Fig. 5.24. The coefficients in Fig. 5.23 include all the constant terms in the contact model that do not need to be computed individually for each collision.

5.2.3 Allocating GPU resources and copying data across

As explained earlier, the GPU processing devices only have access to GPU memory. GPU memory has to be allocated in a C-like style with memory allocation commands. Allocating and filling the arrays that store the particle properties is straight forward. The same structures have been initialised in RAM and their mirror versions on the GPU are created. A difference is, however, that the particle positions array is copied to a VBO (Vertex Buffer Object) that resides in GPU texture memory. This is because the positions are used to render the particles and rendering naturally takes place on a

Parameter	Value
Fixed parameters	
grid size	{128,128,128}
Number of particles	303156
World origin	{0,0,0}
Number of lines (mesh face)	259
drumLength	0.27m
drumRadius	0.15m
Modifiable parameters	
Particle radius	0.0015 m
Cell size	2*particleRadius
ppNormalForceCoeff	2335490000
ppNormalDampingCoeff	59.5637
ppTanForceCoeff	5696.31
ppTanDampingCoeff	75.9528
psNormalForceCoeff	32147700.0
psNormalDampingCoeff	9.96332
psTanForceCoeff	67.9176
psTanDampingCoeff	11.8243
angularIntegrationCoeff	31438
drumRotation	6.46991rad/s
ppFriction	0.154
psFriction	0.3
particleMass	0.0000353429kg
gravity	{0,-9.81,0}m.s^{-1}

Fig 5.23: Modifiable and fixed simulation parameters with example values from simulation run

Parameter	Value
Particle Radius	0.0015m
Mesh particle radius	0.0015m
Particle density	2500kg.m^3
Mesh particle density	950km.m^3
Poisson ratio pp	0.23
Poisson ratio ps	0.42
Restitution coeff pp	0.66
Restitution coeff ps	0.55
Young's Shear Mod pp	26GPa
Young's Shear Mod ps	0.31GPa
Mill rotation speed	40%-110% Critical
Friction pp	0.154
Friction ps	0.32

Fig 5.24: Physical properties of a setup simulated

texture. The velocity and angular velocity arrays can be bound to textures at a later stage to speedup data access, but as these arrays require read and write access, they are not bound to a VBO in the same way as the positions are. Of course the positions array must be updated but after this is done, it is bound to a VBO. The data representing the geometry, whether it be in the form of a triangle mesh or be it in the form of a 2D drum face (points and lines), is stored in constant memory, as the number of vertices and triangles is small and each thread requires fast access to this data. However, this requires one to know the size of the vertex and triangle arrays at compile time, which prevents one from changing the geometry once the simulation has started. This is, at present, acceptable, but can be altered by another scheme, if necessary. The simulation parameters are also copied into constant memory. GPU resources must also be allocated at this stage for all the GPU processes that occur (Fig 5.6) because if device code were to allocate device memory, it would be difficult to retrieve a handle to this memory. Anyway, the malloc and copy routines (cudaMalloc() and cudaMemcpy()) are run on the host (CPU).

The particle grid must be explained at this point. The world space inside which the simulation lives is divided into a uniform rectangular grid, as described in section 3. The sole purpose of this grid is to make the task of collision detection more manageable. The reasons for choosing a uniform grid are laid out in section 3. A grid was chosen to have cell size the same size as the particle diameter. This limits the maximum number of particles that can fit into a grid cell and it also forces a particle to inhabit at most two cells in any direction. This second quality is crucial to the collision detection algorithm implemented. The grid makes up the world space and must provide sufficient space for the simulation. It was chosen to be {128,128,128} cells large. This is because the particle diameter is 3mm, so the world space becomes 384mm x 384mm x 384mm, which is sufficient to house the mill with the properties listed. One can implement a larger world size than grid size by wrapping it around so that it fits into the grid. However, this increases M as described before on the order of the search algorithm. Lines 5 - 7 in Algorithm 5.3 implement this using the bitwise operation (& $(2^q - 1)$)[15] but there is an incentive to not using this feature. If one keeps the world size smaller than the grid, each grid cell can be used as a voxel in real space and processing of the data required from the simulation can take place on the grid instead of another extra data structure. Grid size should be restricted to powers of 2, as ideally all resource allocation on the GPU should be (to maximise performance), and this neat bitwise & wrapping technique must be changed if this is changed. Clearly a grid size of {64,64,64} would provide too small a world space and a grid size of {256,256,256} would require storage for 2^{24} (>16 million) cells which would put a large, but not unmanageable requirement on memory. One must be careful never to launch a kernel that spawns a thread for each cell in the grid because there can easily be too many cells and the program should not be dependent on this factor. Spawning threads per particle is acceptable and all operations should be thought of in terms of this.

[15] assuming the grid dimension of 2^q

5.2.4 Calculating particle hash kernel

The position vector of each particle is used to calculate a hash value that can be used to sort the particles so that particles in the same grid cell lie sequentially after each other in a sorted particle array. This is so that threads in a warp, that are spawned on a per particle basis and naturally map sequentially to particles in the array, access data that is located spatially together. This allows for good use of the GPU memory architecture, as explained in section 5.1.2. The technique for hashing the particles is given in pseudo-code in Algorithm 5.3. Notice the branching operation on line 2. As explained before, if one of the threads in an executing half warp[16] diverges, the way the SM (stream multiprocessor) handles this is to run both operation paths, but with only the appropriate threads active. At this point in the code, the particles are still sequentially grouped as free particles or geometry particles, so there will be only 1 warp that contains particles of both types. Thus the thread divergence problem is not an issue.

Algorithm 5.3 Calculating a hash based on particle position

1: **procedure** HASHING($x, y, z, theta$) ▷ theta is the mill orientation
2: **if** $pos.w = 1$ **then** ▷ if particle is part of geometry
3: $\{x, y\} \leftarrow \{x\cos(\theta) - y\sin(\theta), x\sin(\theta) + y\cos(\theta)\}$ ▷ Rotate particle through θ
4: **end if**
5: $gridPos.x \leftarrow \left\lfloor \frac{x}{cellsize.x} \right\rfloor \&(gridDim.x - 1)$ ▷ snaps position to grid
6: $gridPos.y \leftarrow \left\lfloor \frac{y}{cellsize.y} \right\rfloor \&(gridDim.y - 1)$ ▷ bitwise & with $gridDim$ of form 2^n
7: $gridPos.z \leftarrow \left\lfloor \frac{z}{cellsize.z} \right\rfloor \&(gridDim.z - 1)$ ▷ $gridDim - 1$ has form $(11\ldots1_2)$
8: $hash \leftarrow gridPos.z * gridPos.y * gridPos.x + gridPos.y * gridPos.x + gridPos.x$
9: **return**$\{particle.index, hash\}$ ▷ Return index with hash, so after sorting by hash, particle is locatable
10: **end procedure**

5.2.5 Sorting particles

What we now have is a list of particles with their positions in the list being their particle IDs; a list of particle velocities and angular velocities in the same order; a list of particle IDs, which are simply the particle's position in the list; and a list of hashes, which are the IDs of the cell each particle is in [Gre10]. Figure 5.25 gives an example for the situation in Fig. 5.26. The coupled list of $\{ID, hash\}$ is sorted using the fast radix sort implemented in the CUDPP library. The algorithm for this sort was created by Satish, Harris and Garland [SHG09]. The CUDA Data Parallel Primitives library contains many efficient and useful tools for common operations such as matrix reduction, parallel sorting, stream compaction, building data structures and summed area trees.

[16]Fermi architecture supports simultaneous execution of whole warps (32 threads) but earlier architectures supported only simultaneous execution of half warps (16 threads)

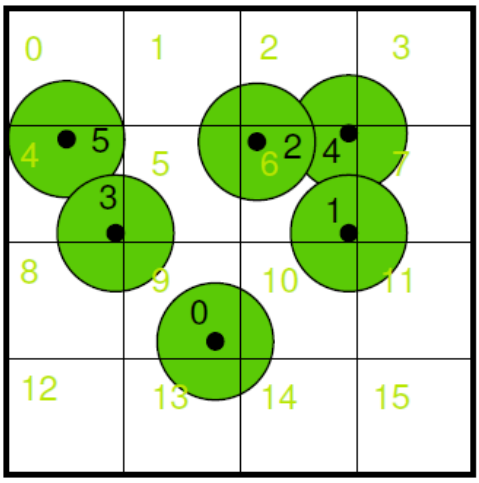


Fig 5.25: 2D scenario to illustrate use of the sorting technique

The sorting algorithm does not reorder the particle data. Rather, a sorted copy is produced using the sorted $\{ID, hash\}$ couple. This is because the unsorted particle data must be preserved for the ID property to mean anything. Without the particle ID being preserved in this manner, it would be impossible to track a single particle. The grid is sorted each time step, and the sorted data properties are used to compute the forces on each particle, but the updated position and velocity values are written to the correct unsorted positions. The copy of the data properties used to determine the forces cannot be altered during a timestep because one has no control over the order in which particles are processed. Actually this couldn't be allowed even if we could control the order of operation because the result of processing a timestep must not depend on the order in which the particles are handled. This scheme has the advantage that the sorted data properties can be bound to textures, since they are not altered and since they are sorted, maximum use of spatial locality caching is made. The disadvantage is that the write-back to the original particle location is not ordered so memory coalescing cannot be utilised.

5.2.6 Calculating cell start particle number and end particle number kernel

The collision detection algorithm, as described in section 3.3, searches the grid cell a particle is in, as well as the neighbouring cells, for particles which are then tested to see if a collision has occurred. To do this, one must be able to determine which particles are in a specific cell. The particle hashing algorithm gives us the cell ID a particle is in, but here we are concerned with the reverse. Since the particle properties lists have been sorted (rather a sorted copy made) and bound to textures, all that is required is that each cell stores the position in the sorted list of the first particle in it, and the position in this list of the last particle in the cell. The array of grid cells is the largest data structure in the simulation. It has $gridDim.x * gridDim.y * gridDim.z = 128^3 = 2097152$ cells. Care must be taken to minimize what is stored in each cell. Algorithm 5.4 gives the

Position	Velocity	Angular Velocity	ID	Hash (cell ID)
$\vec{r^0}$	$\vec{v^0}$	$\vec{\omega^0}$	0	9
$\vec{r^1}$	$\vec{v^1}$	$\vec{\omega^1}$	1	6
$\vec{r^2}$	$\vec{v^2}$	$\vec{\omega^2}$	2	6
$\vec{r^3}$	$\vec{v^3}$	$\vec{\omega^3}$	3	4
$\vec{r^4}$	$\vec{v^4}$	$\vec{\omega^4}$	4	6
$\vec{r^5}$	$\vec{v^5}$	$\vec{\omega^5}$	5	4

(a) Data before sorting

Sorted Position	Sorted Velocity	Sorted Angular Velocity	ID	Hash (cell ID)
$\vec{r^3}$	$\vec{v^3}$	$\vec{\omega^3}$	3	4
$\vec{r^5}$	$\vec{v^5}$	$\vec{\omega^5}$	5	4
$\vec{r^1}$	$\vec{v^1}$	$\vec{\omega^1}$	1	6
$\vec{r^2}$	$\vec{v^2}$	$\vec{\omega^2}$	2	6
$\vec{r^4}$	$\vec{v^4}$	$\vec{\omega^4}$	4	6
$\vec{r^0}$	$\vec{v^0}$	$\vec{\omega^0}$	0	9

(b) Data sorted by hash

Fig 5.26: Tables illustrating data sorting technique to allow for memory coalescing

pseudo-code of the algorithm this kernel performs. The particles in a cell are in the interval $[cellStart[i], cellEnd[i])$. The array *cellEnd* has all the values not set by the algorithm, set to 0. Similarly, the array *cellStart* has all the values not set by the algorithm, set to $0 \times FFFFFFFF$. This indicates an empty cell.

5.2.7 Detecting and processing collisions

The collision detection algorithm launches a CUDA kernel with 1 thread per particle. The properties of that the particle are retrieved from the sorted arrays using texture lookups. Due to the sorting, each thread accesses the next particle's properties and since each property is a 4-double, or 32 bytes, they are naturally aligned in memory making good use of the memory transaction capability described in section 5.1.2. A 3-double force variable is initialised and to it is added the force that particle experiences by the collision with each particle it is in contact with and this force is then used to update the particle's velocity. The alignment requirement is not enforced on this variable because it resides in per thread registers[17] (unless spilled to local memory in which case coalescing is guaranteed by the compiler). The same cannot be done for the particle's angular velocity because it is not just the force that gives the angular velocity. It is also the central collision point of each contact that determines this. As a result, the angular velocity must be updated every collision. The newly computed velocities are stored in the original unsorted arrays. The position of the particle is not updated at this stage.

[17] registers local to a specific streaming processor and hence thread

Algorithm 5.4 Calculating the position in the sorted particle list of the first and last particle in each cell

```
1:  procedure CALCULATING CELL START AND END
2:      hash ← hashArray[index]                          ▷ See below
3:      sharedHash[threadID + 1] ← hash                  ▷ See below
4:      if threadID = 0 AND index > 0 then
5:          sharedHash[0] = hashArray[index − 1]         ▷ See below
6:      end if
7:      __syncthreads()                  ▷ CUDA thread management function
8:      if index = 0 then                ▷ first particle must be first particle in cell
9:          cellStart[hash] ← 0                          ▷ hash is cell ID
10:     else if hash ≠ sharedHash[threadID] then         ▷ See below
11:         cellStart[hash] ← index
12:         cellEnd[sharedHash[threadID]] ← index        ▷ Cell end for previous cell
13:     end if
14:     if index = N − 1 then                            ▷ if particle is last particle
15:         cellEnd[hash] = N
16:     end if
17: end procedure
```

Line	Extended comment
2:	1 Thread per particle so block and thread ID gives particle the index. Each CUDA thread knows its thread and block ID.
3:	Shared hash is stored in shared memory and is accessible by all threads in a warp. This avoids each thread loading 2 hashes.
4:	if thread is first thread in a block, but not first thread in the grid.
5:	First thread must check previous particle's hash. This thread loads 2 hashes.
7:	At this point threads must wait until all threads catch up otherwise *sharedHash* could be accessed before being initialised.
10:	Particle's cell ≠ previous particle's cell

Algorithm 5.5 is the pseudo-code for the collision detection and processing algorithm implemented. The coefficients used on lines 48 - 51, 56 are calculated from the contact model described in section 4.3.

Algorithm 5.5 Pseudo-code for the collision detection and collision processing algorithm implemented

1: **procedure** COLLISION DETECTION & PROCESSING
2: $F \leftarrow \{0,0,0\}$ ▷ Initialise force to $\vec{0}$
3: $\vec{p} \leftarrow$ **FETCH**$(posArrayTex, index)$ ▷ Fetch particle position from texture
4: $\vec{v} \leftarrow$ **FETCH**$(velArrayTex, index)$ ▷ Fetch particle velocity from texture
5: $\vec{\omega} \leftarrow$ **FETCH**$(\omega ArrayTex, index)$ ▷ Fetch particle angular velocity from texture
6: **if** p.w=0 **then** ▷ If particle is free
7: $gridPos \leftarrow \{\lfloor \frac{x}{cellsize.x} \rfloor, \left\lfloor \frac{y}{cellsize.y} \right\rfloor, \lfloor \frac{z}{cellsize.z} \rfloor\}$
8: **for** $k = -1$ **to** 1 **do**
9: **for** $j = -1$ **to** 1 **do**
10: **for** $i = -1$ **to** 1 **do**
11: $\vec{F} \leftarrow \vec{F}$ + collideCell(originalParticle,particles in cell $gridPos + \{i,j,k\}$)
12: $i \leftarrow i + 1$
13: **end for**
14: $j \leftarrow j + 1$
15: **end for**
16: $k \leftarrow k + 1$
17: **end for**
18: **end if**
19: $originalIndex \leftarrow particleID[index]$ ▷ See below
20: $unsortedVelArray[originalIndex] \leftarrow \vec{v} + \frac{F}{m}\Delta t$
21: $unsorted\omega Array[originalIndex] \leftarrow \vec{\omega}$
22: **end procedure**

CONTINUED ON NEXT PAGE

Line	Extended comment
2:	Data retrieved from sorted arrays
7:	Grid size is larger than the mill size so grid wrappings is not required
11:	The memory address of ω is also passed to collideCell
19:	Retrieves index for unsorted arrays from sorted $\{ID, hash\}$ tuples
21:	$\vec{\omega}$ has been updated already in the CollideSpheres procedure.

The technique for detecting and processing collisions between particles and the geometry when the geometry is represented by a triangle mesh, stored in device constant memory, requires one to be able to compute the nearest point on a triangle and test if the distance from that point to the centre of the particle is less than the radius of the particle. The algorithm to determine that point is explained in pseudo-code in Algorithm 5.6 and comes from [Eri05]. This algorithm has been optimised for graphics cards, as all cross

23: **procedure** COLLIDECELL(originalParticle, cell $gridPos$)
24: $hash \leftarrow gridPos.x * gridPos.y * gridPos.z + gridPos.y * gridPos.z + gridPos.x$
25: $startIndex \leftarrow$**FETCH**$(cellStartTex, hash)$
26: $\vec{F} = \{0, 0, 0\}$ ▷ total force exerted by particles in this cell
27: **if** $startIndex \neq$ 0xFFFFFFFF **then** ▷ if cell isn't empty
28: $endIndex \leftarrow$**FETCH**$(cellEndTex, hash)$
29: **for** $j = startIndex$ **to** $endIndex - 1$ **do** ▷ $[startIndex, endIndex)$
30: **if** $j \neq index$ **then** ▷ particle cannot collide with itself
31: $pos2 \leftarrow$**FETCH**$(posArrayTex, j)$
32: $vel2 \leftarrow$**FETCH**$(velArrayTex, j)$
33: $\omega 2 \leftarrow$**FETCH**$(\omega ArrayTex, j)$
34: $\vec{F} \leftarrow \vec{F} +$ collideSpheres(originalParticle,particle j))
35: **end if**
36: **end for**
37: **end if**
38: **end procedure**

39: **procedure** COLLIDESPHERES(originalParticle,particle 2)
40: $dist \leftarrow \text{length}(\vec{posA} - \vec{posB})$ ▷ Original particle A
41: $\vec{F} \leftarrow \{0, 0, 0\}$
42: **if** $dist < 2r$ **then** ▷ if particles overlap
43: $\widehat{n} \leftarrow \frac{\vec{posA} - \vec{posB}}{dist}$ ▷ normal direction
44: $\vec{V} \leftarrow \vec{velA} - \vec{velB}$
45: $\vec{colPos} \leftarrow \frac{1}{2}\left(\vec{posA} + \vec{posB}\right)$
46: $\vec{V_t} \leftarrow \vec{V} - \vec{V}.\widehat{n} + \vec{\omega A} \times \left(\vec{colPos} - \vec{posA}\right) - \vec{\omega B} \times \left(\vec{colPos} - \vec{posB}\right)$
47: $\delta_n \leftarrow r - \frac{1}{2}dist$
48: $\vec{F_n} \leftarrow \text{ppNormalForceCoeff} * \delta_n^{\frac{3}{2}}$ ▷ See Fig. 5.18 for example of coeffs.
49: $\vec{F_n} \leftarrow \vec{F_n} - \text{ppNormalDampingCoeff} * \delta_n^{\frac{1}{4}} * \left(\vec{V}.\widehat{n}\right) * \widehat{n}$
50: $\vec{F_t} \leftarrow -\text{ppTanForceCoeff} * \sqrt{\delta_n} * \vec{V_t}$ ▷ tangential force
51: $\vec{F_t} \leftarrow \vec{F_t} - \text{ppTanDampingCoeff} * \delta_n^{\frac{1}{4}} * \vec{V_t}$
52: **if** $\mu * F_n < F_t$ **then** ▷ friction
53: $\vec{F_t} \leftarrow -\mu F_n * \widehat{V_t}$
54: **end if**
55: $\vec{F} \leftarrow \vec{F_n} + \vec{F_t}$
56: $\omega A \leftarrow \omega A + \text{angularIntegrationCoeff} * \left(\vec{colPos} - \vec{posA}\right) \times \vec{F_t}$ ▷ See below
57: **end if**
58: **return** $\vec{F}$
59: **end procedure**

Line	Extended comment
56:	This updates the value of the variable created in the original method because the memory address was passed down the chain of function calls.

product operations have been replaced by sets of dot product operations, which can be performed much faster.

Computing the force on a particle due to a collision with a triangle mesh uses the same contact model as for particle particle interactions. This is not physically correct because the way a sphere deforms when it hits another sphere must be different from the way a flat triangle of material deforms when it hits a sphere and it is this deformation and its restoring itself to original shape that causes the force on the sphere. However, modifications to the contact model would be minor and are unlikely to make a difference. Algorithm 5.7 is just a modification of the procedure collideSpheres in Algorithm 5.5. The total force a particle experiences from interactions with the mesh is the sum of the forces due to each triangle it is in contact with. In the mesh in Fig. 5.14, for example, the particle can be in contact with at most 3 faces, or 6 triangles.

The triangle meshes shown in this thesis can be more efficiently described simply by a 2D cross section in the plane defined by their central axis because they have no features dependent on the z component except the flat front and back faces. When such meshes are used in the simulation, one can take advantage of this by substituting Algorithm 5.6 with Algorithm 5.8.

5.2.8 Updating particle positions kernel

The mesh processing algorithms were included in the integration kernel rather than being in a separate kernel for a number of reasons. Threads in the integration kernel interact with the properties of only one particle per thread. Hence the kernel is run on the unsorted arrays where the velocities have been updated by the kernel detecting and processing particle particle collisions. When interacting with a mesh, each particle is only required to interact with the mesh data and this is stored in device constant memory so it is fair to work with the unsorted arrays. There are many improvements to the mesh handling technique implemented in this thesis that can be made and will have to be made when requirements for handling detailed and large meshes are made, but this simple technique was sufficient for our purposes until now.

Updating the particle positions using the explicit Euler integration scheme described in section 4.5 is straight forward. Each thread reads in its particle's velocity by means of a texture lookup and its particle's position by means of a coalesced memory read (the updated values must be written back to this array so read/write access is required) and the updated position value is written back. There is no concern with thread synchronisation because no other thread will access this data.

Algorithm 5.6 Closest point on a triangle to a point

procedure ClosestPointOnTriangleToPoint(Point p, Vertex a, Vertex b, Vertex c)
  $\vec{ab} \leftarrow \vec{b} - \vec{a}$
  $\vec{ap} \leftarrow \vec{p} - \vec{a}$
  $\vec{ac} \leftarrow \vec{c} - \vec{a}$
  $d1 \leftarrow \vec{ab}.\vec{ap}$
  $d2 \leftarrow \vec{ac}.\vec{ap}$
  if $d1 \leq 0$ **AND** $d2 \leq 0$ **then**
    return $\vec{a}$
  end if
  $\vec{bp} \leftarrow \vec{p} - \vec{b}$
  $d3 \leftarrow \vec{ab}.\vec{bp}$
  $d4 \leftarrow \vec{ac}.\vec{bp}$
  if $d3 \geq 0$ **AND** $d4 \leq d3$ **then**
    return $\vec{b}$
  end if
  $vc \leftarrow d1 * d4 - d3 * d2$
  if $vc \leq 0$ **AND** $d1 \geq 0$ **AND** $d1 \leq 0$ **then**
    return $\vec{a} + \frac{d1}{d1-d3}\vec{ab}$
  end if
  $\vec{cp} \leftarrow \vec{p} - \vec{c}$
  $d5 \leftarrow \vec{ab}.\vec{cp}$
  $d6 \leftarrow \vec{ac}.\vec{cp}$
  if $d6 \geq 0$ **AND** $d5 \leq d6$ **then**
    return $\vec{c}$
  end if
  $vb \leftarrow d5 * d2 - d1 * d6$
  if $vb \leq 0$ **AND** $d2 \geq 0$ **AND** $d6 \leq 0$ **then**
    return $\vec{a} + \frac{d2}{d2-d6}\vec{ac}$
  end if
  $va \leftarrow d3 * d6 - d5 * d4$
  if $va \leq 0$ **AND** $d4 - d3 \geq 0$ **AND** $d5 - d6 \geq 0$ **then**
    return $\vec{b} + \frac{d4-d3}{d4-d3+d5-d6}\left(\vec{c} - \vec{b}\right)$
  end if
  $denom \leftarrow \frac{1}{va+vb+vc}$
  return $\vec{a} + vb * denom * \vec{ab} + vc * denom * \vec{ac}$
end procedure

Algorithm 5.7 Detecting and processing collision between a particle and a triangle mesh

1: **procedure** COLLIDESPHERETRIANGLE(originalParticle,triangle)
2: $\vec{q} \leftarrow$ ClosestPointOnTriangleToPoint$(\vec{p}, \vec{a}, \vec{b}, \vec{c})$ ▷ See below
3: $dist \leftarrow \text{length}(\vec{p} - \vec{q})$
4: $\vec{F} \leftarrow \{0, 0, 0\}$
5: **if** $dist < r$ **then** ▷ if particles overlap
6: $\widehat{n} \leftarrow \frac{\vec{p}-\vec{q}}{dist}$ ▷ normal direction
7: $\vec{velq} \leftarrow \{q.y, -q.x, 0\} * millRotation$ ▷ See below
8: $\vec{V} \leftarrow \vec{v} - \vec{velq}$
9: $\vec{V_t} \leftarrow \vec{V} - \vec{V}.\widehat{n} + \vec{\omega} \times (\vec{q} - \vec{p})$
10: $\delta_n \leftarrow r - dist$
11: $\vec{F_n} \leftarrow \text{ppNormalForceCoeff} * \delta_n^{\frac{3}{2}}$ ▷ See Fig. 5.18 for example of coeffs.
12: $\vec{F_n} \leftarrow \vec{F_n} - \text{ppNormalDampingCoeff} * \delta_n^{\frac{1}{4}} * \left(\vec{V}.\widehat{n}\right) * \widehat{n}$
13: $\vec{F_t} \leftarrow -\text{ppTanForceCoeff} * \sqrt{\delta_n} * \vec{V_t}$ ▷ tangential force
14: $\vec{F_t} \leftarrow \vec{F_t} - \text{ppTanDampingCoeff} * \delta_n^{\frac{1}{4}} * \vec{V_t}$
15: **if** $\mu * F_n < F_t$ **then** ▷ friction
16: $\vec{F_t} \leftarrow -\mu F_n * \widehat{V_t}$
17: **end if**
18: $\vec{F} \leftarrow \vec{F_n} + \vec{F_t}$
19: $\omega \leftarrow \omega + \text{angularIntegrationCoeff} * (\vec{q} - \vec{p}) \times \vec{F_t}$
20: **end if**
21: **return** $\vec{F}$
22: **end procedure**

Line	Extended comment
2:	$\vec{p}$ is the position of the center of the particle being tested. Points $\{\vec{a}, \vec{b}, \vec{c}\}$ are the vertices of the triangle being tested.
7:	Triangles in the mesh are moving only with the angular velocity of the drum. The velocity of any point on the mill is perpendicular to the radial position of that point and equal in magnitude to the drumRotation * the distance to the axis of the drum.

Algorithm 5.8 Substitute algorithm for Algorithm 5.6 that takes advantage of axial symmetry

```
1: procedure ClosestPointOnMeshToPoint(Point p,Segment s)
2:     ab⃗ ← b⃗ − a⃗                          ▷ b⃗ and a⃗ are endpoints of segment
3:     t ← ((p⃗−a⃗).ab⃗) / (ab⃗.ab⃗)
4:     if t < 0 then
5:         t = 0
6:     else if t > 1 then
7:         t = 1
8:     end if
9:     q⃗ ← a⃗ + t ∗ ab⃗
10:    return {q.x, q.y, p.z}
11: end procedure
```

5.2.9 Data processing kernel

In order to make use of on-the-fly data processing to save on storage and post processing, the requirements from the simulation must be put into a data processing kernel to be launched at the end of a timestep as often as required. A common requirement from a DEM simulation is the solidicity/porosity plot for the simulated run and the velocity plots for the different components of velocity as well as the speed of the particles. The solidicity plot gives the probability of finding a particle in a volume inside the mill. A porosity plot is the inverse of this. It is convenient to decompose the space inside the mill for this purpose in exactly the same way that the world space is decomposed into grid cells for the purpose of the DEM. This is because then all one has to do is at a timestep when data is being processed for the output, increment a counter in each grid cell by the number of particles in that cell. Similarly for the velocity plots, one adds to the value in each grid cell the sum of the velocities (or the component one is interested in) of the particles in that cell. In order to maintain data alignment, it is good to maintain structures of sizes 1, 2, 4, 8 or 16 bytes, as explained in Section 5.1.2, and it was found convenient to use a 4-double in each cell made of the tuple $\{vX, vY, vZ, count\}$. These are respectively the sum of that component of velocity of all the particles that have been in a specific grid cell during a timestep when data was processed and the sum of the count of number of particles that have been in the cell during a timestep when data was processed. Algorithm 5.9 gives the pseudo-code for this kernel.

When concerned with radial effects, it is generally convenient to sum and average values along the axial direction. It is better to do this operation as part of post processing, however, for a number of reasons. Preserving the information along the axial direction comes at no additional cost, as long as there is memory available, and it is actually faster than having threads whose particles have the same X-Y grid cell coordinate writing to the same value because this would serialise the operation - multiple writes to the same

memory location are not possible simultaneously[18]. The question of the effect of cell size on the results of the simulation is also an interesting one, but this can be performed post-simulation for cell sizes that are a multiple of the cell size used by the particle grid. This is acceptable if the grid cell size is sufficiently small. If finer resolution were required, one could either implement a totally separate data processing grid, or one could subdivide the existing grid for data processing.

Algorithm 5.9 Data processing kernel

```
1:  procedure ProcessData
2:      hash ← FETCH(particleHashTex, index)                    ▷ See below
3:      start ← FETCH(cellStartTex, hash)                       ▷ random lookup, but cached
4:      if start = index then                                   ▷ See below
5:          v⃗ ← velPlot[hash]                                   ▷ Uncoalesced memory read
6:          end ← FETCH(cellEndTex, hash)                       ▷ random lookup, but cached
7:          for i = start to end − 1 do
8:              v⃗el ← FETCH(velArrayTex, i)                     ▷ Sorted velocity array bound to texture
9:              if vel.w = 0 then                               ▷ if particle is free
10:                 v⃗ ← {v.x + vel.x, v.y + vel.y, v.z + vel.z, v.w + 1}
11:             end if
12:         end for
13:     end if
14:     velPlot[hash] ← v⃗                                       ▷ Uncoalesced memory write
15: end procedure
```

Line	Extended comment
2:	*index* is the particleID that is being processed. The array of $\{ID, hash\}$ tuples that has been sorted is bound to a texture called $particleHashTex$. Actually, only the hash value is bound because the ID is not required.
4:	if current particle is first particle in cell continue - avoids double counting.

5.2.10 Copying data from the GPU

The decision as to how often to log data is not too serious as long as care is taken that data over as long a period as possible is taken and that the period when it is taken is representative of something. For instance, logging data before the mill reaches steady state gives skewed data that isn't useful. Also taking data over a period that includes such a random period can skew one's results deceptively. It was decided to launch the processData kernel every 300 timesteps. This gives an enormous number of timesteps logged so data averaging is good. It is also useful to log the position and velocities of the particles 100 times per second for use in animations, which are actually useful to give an idea of whether mechanisms are working as expected. Thus, every 10000 timesteps

[18]There are clever summing routines that take advantage of factors such as all values being loaded into the shared memory of a thread block. This greatly improves the efficiency of the data processing kernel

the unsorted position and velocity arrays are copied from GPU memory to a buffer in RAM. Each buffer can contain 5 million floats (there is a limit to how many elements an array can have in C++) or 20 million bytes (ca. 19MB) which was determined to be effective. This should be increased to the maximum possible when particle numbers in the simulation go up. When filled, the buffers are written to disk in binary format to make use of DMA (Direct Memory Access), which is an asynchronised write.

It is important to backup a simulation periodically during its execution incase some problem causes it to abort. Every 50000 timesteps the simulation is backed up by copying the particle property arrays and the array containing the processed simulation data from the GPU to RAM to disk in a single binary file. For the 303156 particle simulation, this backup file was 47MB. The grid does not need to be backed up because it is built from the particle positions.

Logging so many millions of timesteps can cause problems with precision or variable rollover. Since we are interested in a per particle average, every 500 000 timesteps the velPlot array, containing the processed data, is copied from the GPU and each cell's first three values are divided by the fourth value which is subsequently set to 1. This creates the effect of the simulation having seen only 1 particle in each cell with a velocity that is an average for that cell. The array is then copied back to the GPU. The information that is lost by this, however, is a time average. If required, this feature can be turned off with only a small chance of numerical inconsistencies as a consequence.

5.3 Summary

In this chapter the algorithms used, designed and implemented in this project have been laid out explicitly. Considerations made have been noted and the lucidity of each step gives a lot of confidence in the results. This is crucial in a computational physics code. The care to preserve elegance of the implementation of algorithms is often left out of physics codes. In a field like large-scale DEM modeling this is crucial as performance adds functionality rather than convenience. All the algorithms described in this chapter are neatly rolled into one or more CUDA kernels. This means extension to the program is simple and this is of great importance. It is foreseen that in the future an SPH (smooth particle hydrodynamics) or CFD (computational fluid dynamics) code will be coupled to this DEM. The structure of it allows for this extension.

6 Testing and Results

It is essential to validate the computational results of the simulation framework in this thesis. The aim of this work was to rewrite the DEM framework for the purpose of producing a tool that can be used to gain insight into the fundamental theory of the mechanisms inside dynamic particulate environments, specifically for the environment inside tumbling mills. Elementary particle tests were performed, and this is perhaps more crucial to the validation of the results than any other tests because it allows us to observe that the contact processing is behaving correctly. Testing the computational framework is not given in this chapter, as it was performed using unit tests throughout the development phase, but the algorithms developed and explained in the previous chapter take note of possible sources of computational errors (for example the implementation of infinitesimal rotations - see section 5.2.1) and have implemented solutions.

A study of the performance of this simulation is given in this chapter and the implications of it are explained. This is a major motivation behind this work. Lastly, an investigation into the idea that high friction spherical particles could be used to model shaped particles, was made and is shown in this section.

6.1 Elementary particle tests

Simulating a particle dropped from rest is the simplest scenario that illustrates that the contact model is working correctly. In section 4.3 Fig. 4.5, plotting the force experienced by a particle over a timestep showed the same behaviour observed by [ZW96]. This simulation was produced in a Mathematica implementation of the contact model because in that environment, unlike on a GPU, one has access to all variables at any stage in the process. Fig. 6.1 illustrates a drop test performed on the GPU. A period of 200000 timesteps was computed with position, velocity and angular velocity logged every timestep. This allows us to test that the ratio of drop height to bounce height returns the restitution coefficient, which in this case is $\epsilon = 0.658$, as expected. This ratio of incoming velocity to outgoing velocity from a collision is constant, as shown in Fig. 6.1(c), which gives a table of ϵ calculated for each bounce. As expected, there is no motion in the x-z plane and the angular velocity of the particle remains at zero since there is no tangential force. In Fig. 6.1(b) the almost vertical lines of disconnected dots observed indicate timesteps when the particle is in contact with the plate. During these timesteps the particle experiences a force and accelerates rapidly.

Varying the initial velocity in the y direction produces the unsurprising results in Fig. 6.2 and Fig. 6.3. When the horizontal component of the initial velocity of the particle was varied, some interesting observations were made. Fig. 6.4 and 6.5 show the particle launched with positive and negative velocities respectively and Fig. 6.6 shows the particle launched with greater horizontal velocity and from a greater height. Each time the ball touches the plate, its v_x is reduced. However, the particle's ω_z increases

as expected. One notices that at each successive bounce both these changes decrease until the particle is rolling. The contact model used in this test does not include rolling resistance, as the rolling resistance model presented is not based on physical properties directly and hence convenient to omit from elementary two particles tests. Thus, when a particle is rolling and its point of contact with another particle or with the mesh is stationary with respect to the other body in the contact, it experiences no force if there is no overlap in the normal direction ($\delta_n = 0$). A better solution to the problem of incorporating static and rolling friction into the contact model, is to add shape to the particles [L09].

There is much confidence in using the contact models described in this thesis. However an important test is to make sure that the maximum particle overlap stays small so that the theory used to derive the contact models is justifiable. Fig. 6.0 shows how this maximum particle overlap is related to the incoming velocity of the colliding particles. In a 300mm diameter mill rotating at 100% critical speed ($8.09 rad.s^{-1}$), the smallest particles would centrifuge to the mill shell and have a speed of $1.23 m.s^{-1}$. The formula for determining the critical speed of a mill with diameter D is

$$\omega_{\text{critical}} = \frac{42.3}{\sqrt{D}} * \frac{2\pi}{60} rad.s^{-1}$$

As can be seen from Fig. 6.0, the maximum particle overlap is well below 5% of the particle radius (3mm). This gives us confidence that we can model the forces experienced in the collision using the concept of particle overlap for velocities in the range we are concerned with.

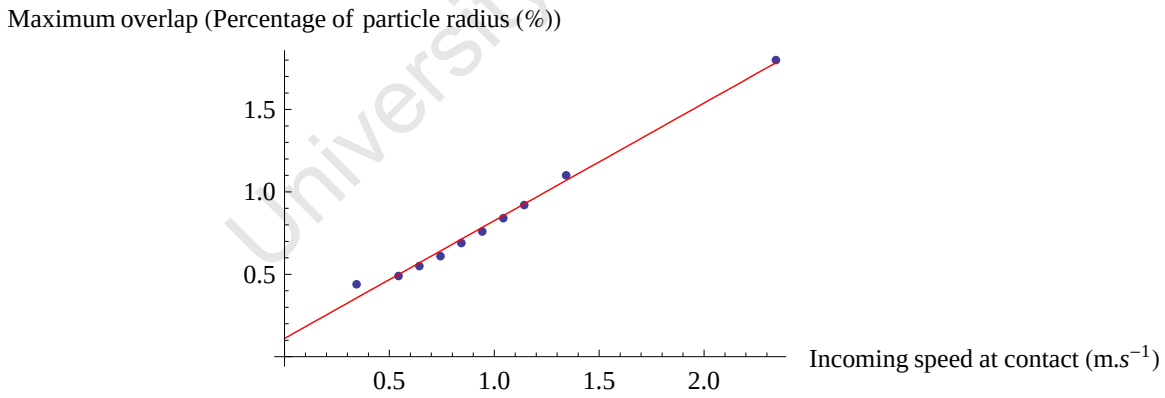
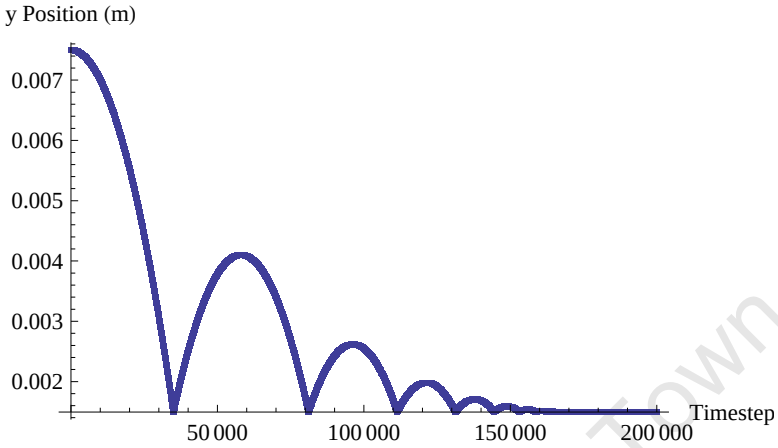
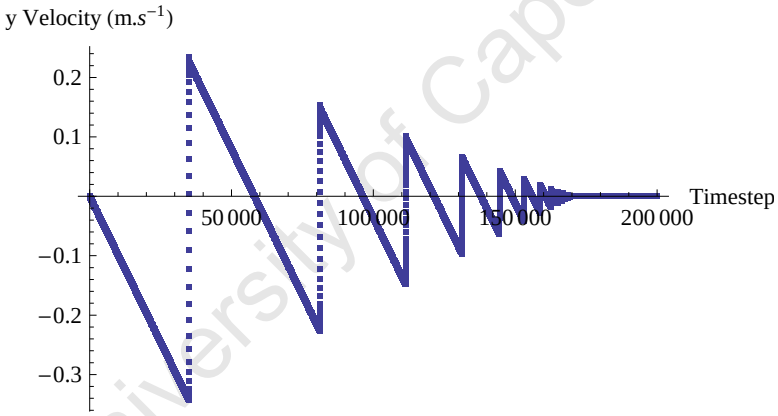


Fig 6.0: Illustrating that in the range of particle speeds inside a mill, the maximum particle overlap (δ_n) is small



(a) Vertical component of position (y component)

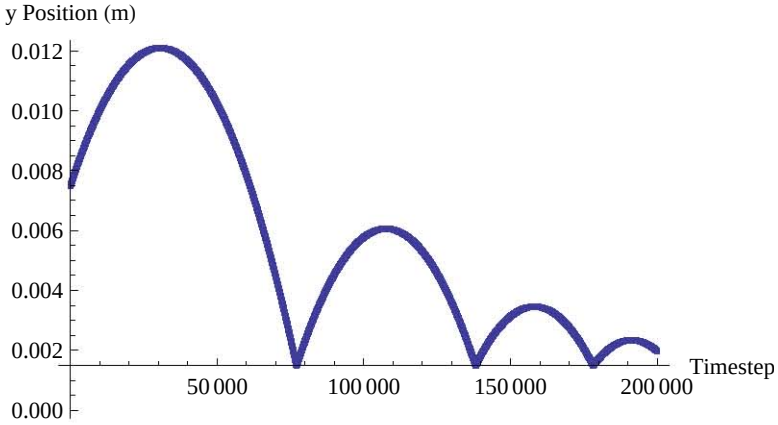


(b) Vertical component of velocity (y component)

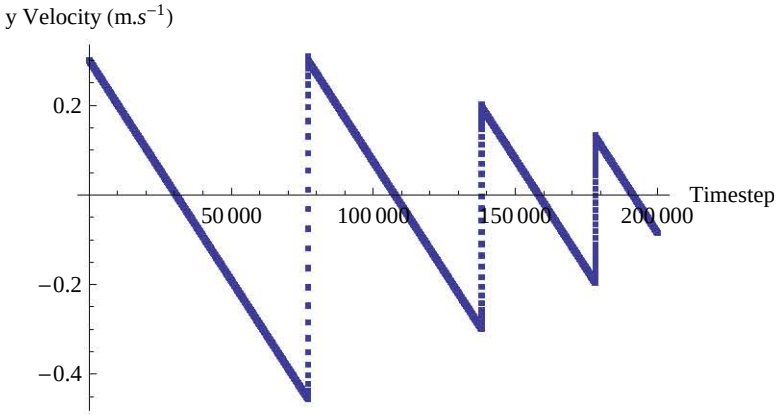
Bounce number	ϵ
1:	0.659
2:	0.658
3:	0.657

(c) Restitution coefficients calculated from sequential drop and bounce heights

Fig 6.1: Glass particle dropped from rest onto flat sheet



(a) Vertical component of position (y component)

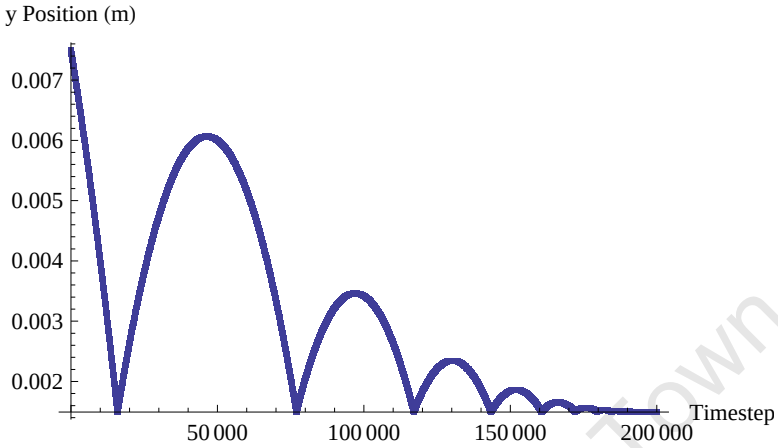


(b) Vertical component of velocity (y component)

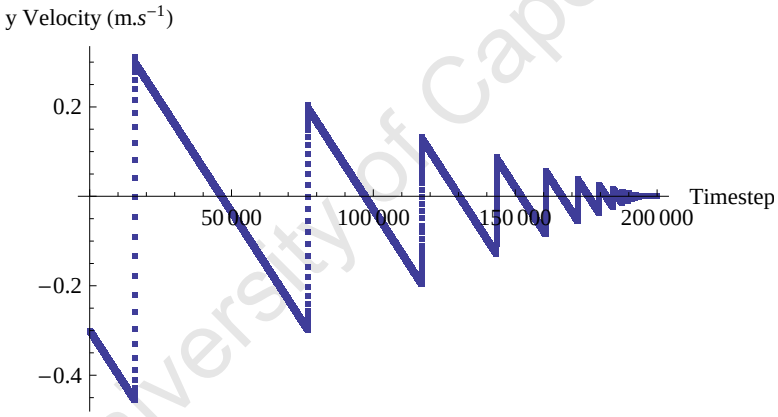
Bounce number	ϵ
1:	0.656
2:	0.656
3:	0.656

(c) Restitution coefficients calculated from sequential drop and bounce heights

Fig 6.2: Glass particle launched with v_0 up, falling onto flat sheet



(a) Vertical component of position (y component)

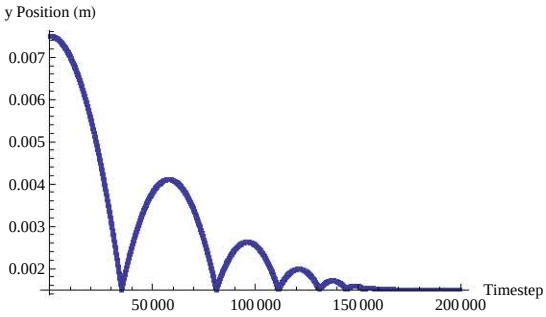


(b) Vertical component of velocity (y component)

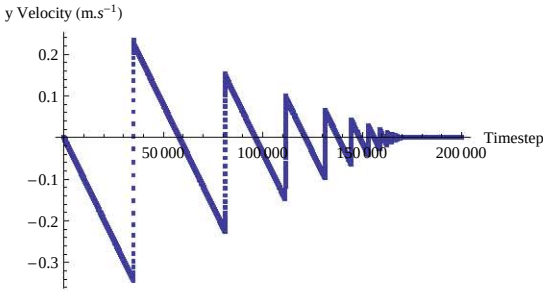
Bounce number	ϵ
1:	0.656
2:	0.656
3:	0.655

(c) Restitution coefficients calculated from sequential drop and bounce heights

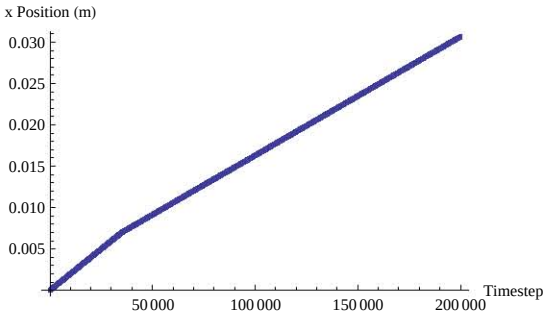
Fig 6.3: Glass particle launched with v_0 down, falling onto flat sheet



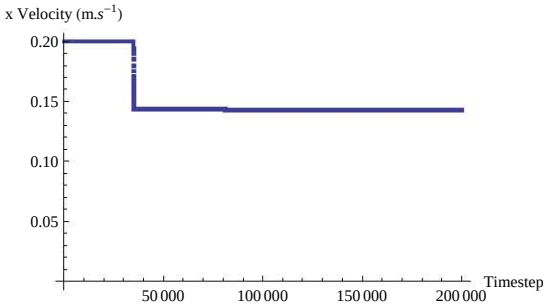
(a) Vertical component of position (y component)



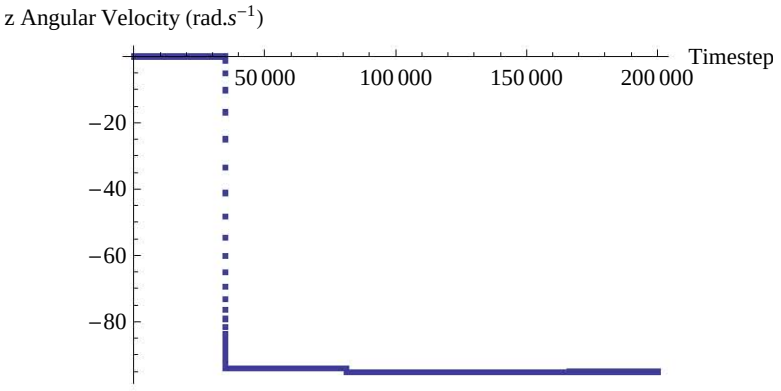
(b) Vertical component of velocity (v_y)



(c) Horizontal component of position (x component)

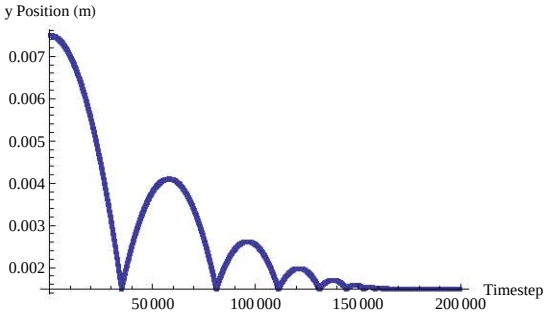


(d) Horizontal component of velocity (v_x)

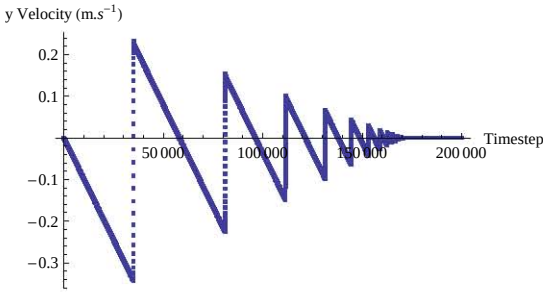


(e) z component of the angular velocity (ω_z)

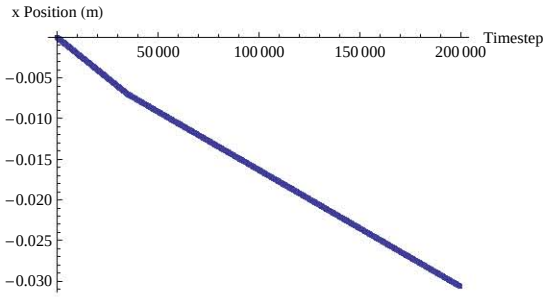
Fig 6.4: Glass particle launched with v_0 in x direction, falling onto flat sheet



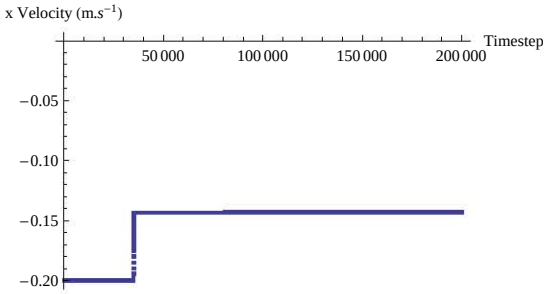
(a) Vertical component of position (y component)



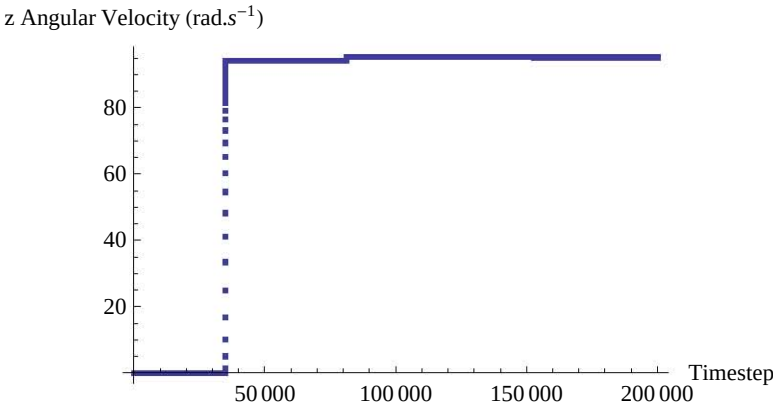
(b) Vertical component of velocity (v_y)



(c) Horizontal component of position (x component)

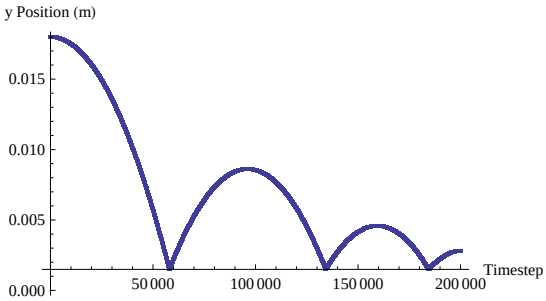


(d) Horizontal component of velocity (v_x)

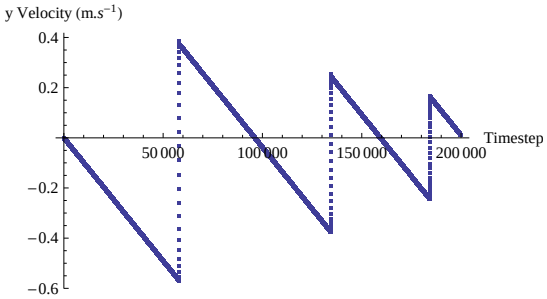


(e) z component of the angular velocity(ω_z)

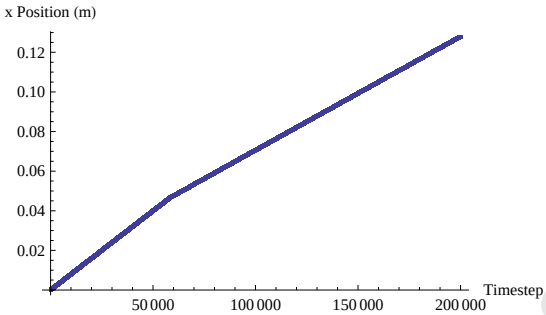
Fig 6.5: Glass particle launched with v_0 in negative x direction, falling onto flat sheet



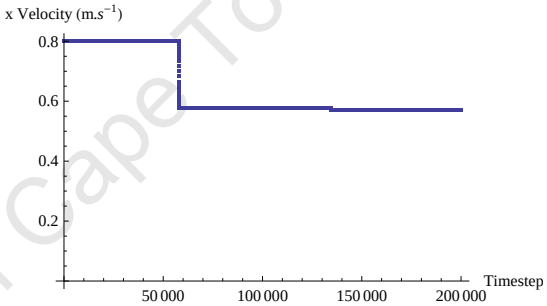
(a) Vertical component of position (y component)



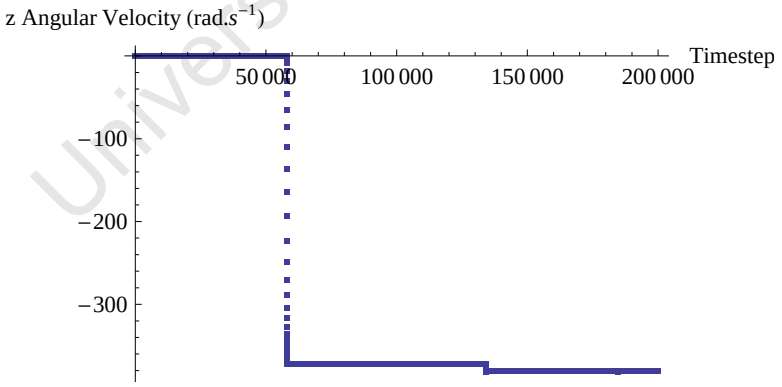
(b) Vertical component of velocity (v_y)



(c) Horizontal component of position (x component)



(d) Horizontal component of velocity (v_x)



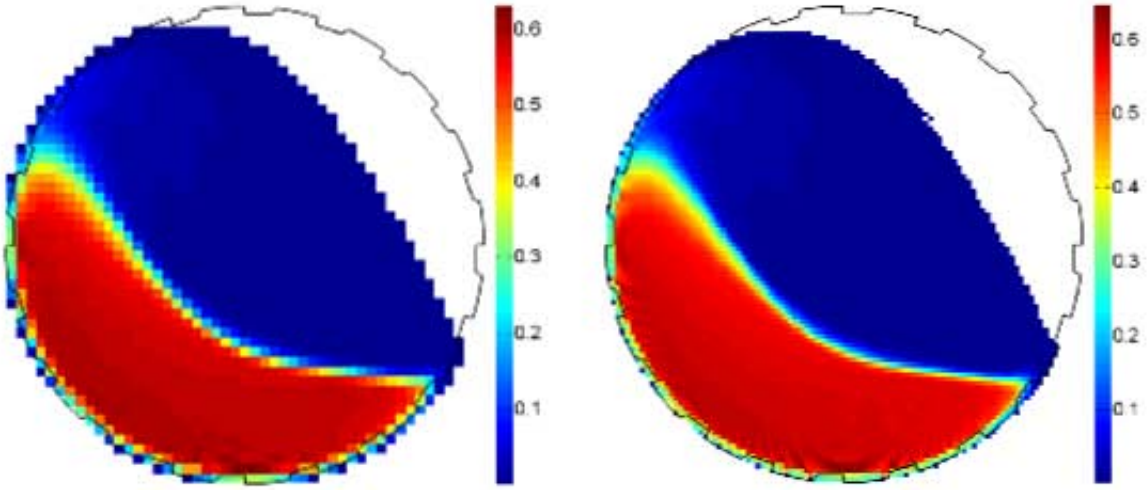
(e) z component of the angular velocity (ω_z)

Fig 6.6: Glass particle launched with v_0 in x direction from greater height, falling onto flat sheet

6.2 Tumbling mill analysis

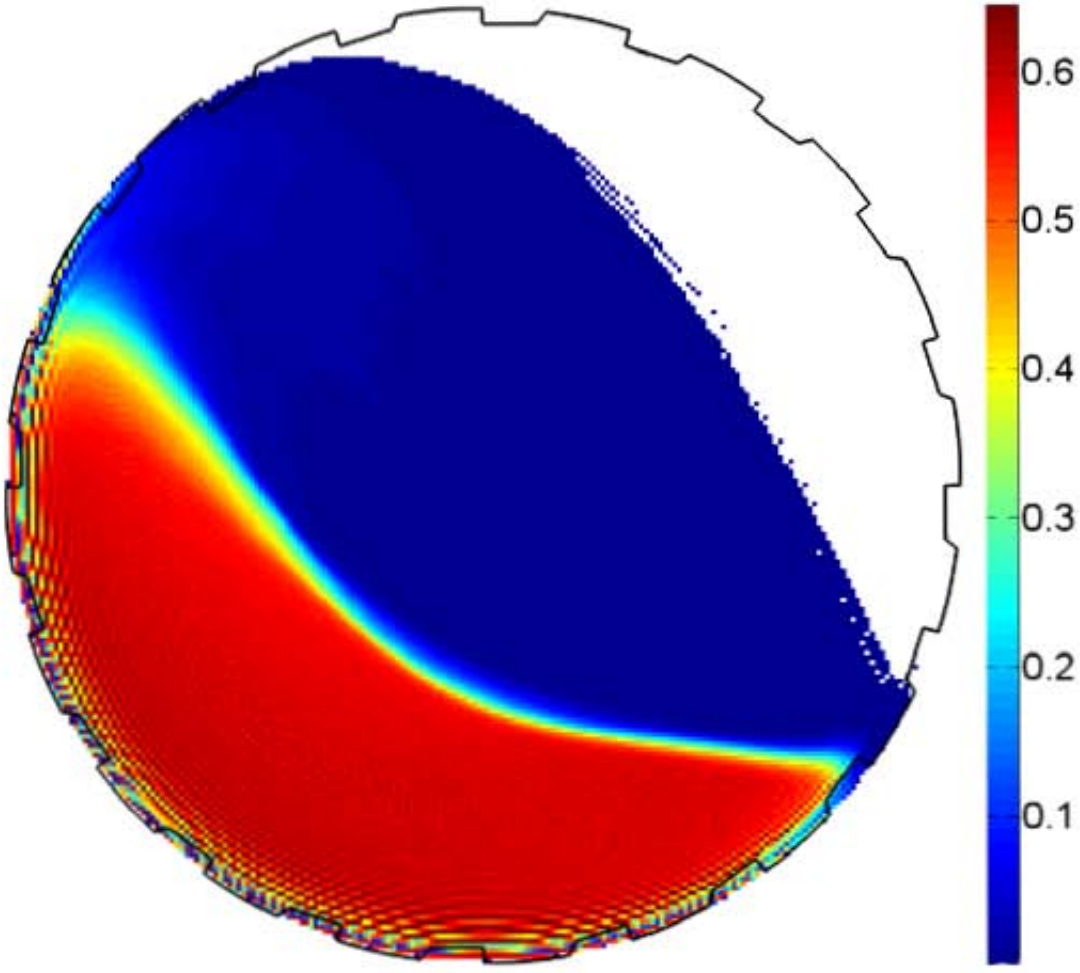
Fig 5.24 gives the physical properties of the system used. The friction values were determined through measurement using a 120 frames per second camera, and the material properties are based on the literature. Experimentation with high friction particles was also performed. The timestep was fixed at 10^{-6}s, which is not very efficient, but was shown to give stability and reliable results from the contact model. In each of the simulations below, at least 3 revolutions of the mill were simulated before data logging commenced to allow steady state to be reached. This occurs after about 2 revolutions of the mill. The mill were simulated at operation speeds of 60%, 70% and 80% of the critical speed of the mill. The end faces of the mill were treated as flat plates of the same material as the rest of the mill. The solidicity plot and velocity profile of each simulation was produced and experimentation was done with different data logging resolutions. Since we were not concerned with axial motion and all the data was binned by X-Y voxel, we could make the resolution of data arbitrarily high. As the solidicity data, which gives the probability of finding a particle at a particular point in the mill, is time averaged over at least a revolution of the mill, one can treat a particle as being in a voxel purely if its centre is in the voxel. This does however, limit the resolution of the data logging because once the volume of a particle gets much greater than the volume of a voxel, one is in danger of encountering solidicities of greater than 1. Time averaging over a longer period may alleviate this problem. We found that using voxel sizes of R, $2R$ and $4R$, with R being the particle radius, was sufficient. The coarser grids are useful in comparing to experimental setups that naturally give coarser resolutions. The respective dimensions of the grid were 256x256, 128x128 and 64x64.

The motion of the charge gets more aggressive as the operating speed increases, as expected. This is seen clearly comparing Fig. 6.8, Fig. 6.10 and Fig 6.12 are scaled velocity vector plots where the scaling is for esthetic reasons purely. The relative size of the vectors is significant as they indicate the relative velocities of particles in each voxel inside the mill. To aid interpretation these quiver plots have been coloured according to the legend attached to each plot. The color indicates the absolute velocity logged. The vectors display the flow field inside the mill. Fig. 6.8(c), Fig. 6.10(c) and Fig. 6.12(c) present the data logged at the higher resolution and from these plots, the effect of mill speed (and lifter profile) is most clear. The equilibrium surface can be clearly seen in the quiver plots where the net velocity of the particles goes to zero. The quiver plots clearly indicate the region in the mill where charge is changing direction. The splash effect at the toe of the mill is most easy to detect in Fig. 6.10(c). From the solidicity plots, it is seen that the solidicity in the bulk region of the mill is around 0.5-0.6, as expected. Increasing the data logging resolution gives much smoother data and this smoothness is not artificial. The particle counts in these plots is of the order of 10^9 and averaged over at least 1 revolution of the mill so using the fine grid really does extract more information from the grid. An interesting note is that simulation performance is not affected by the resolution of data logged.



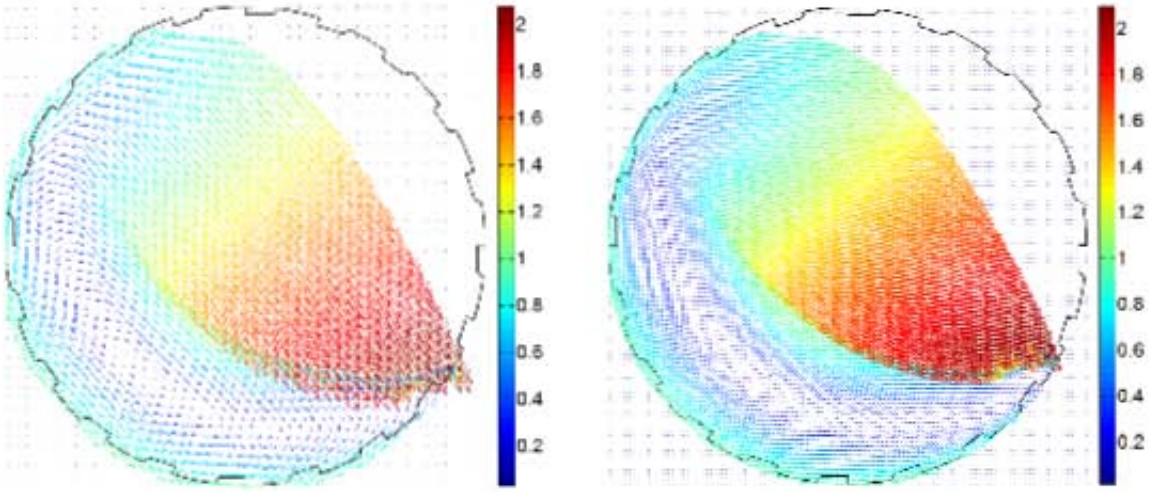
(a) 64x64 grid

(b) 128x128 grid

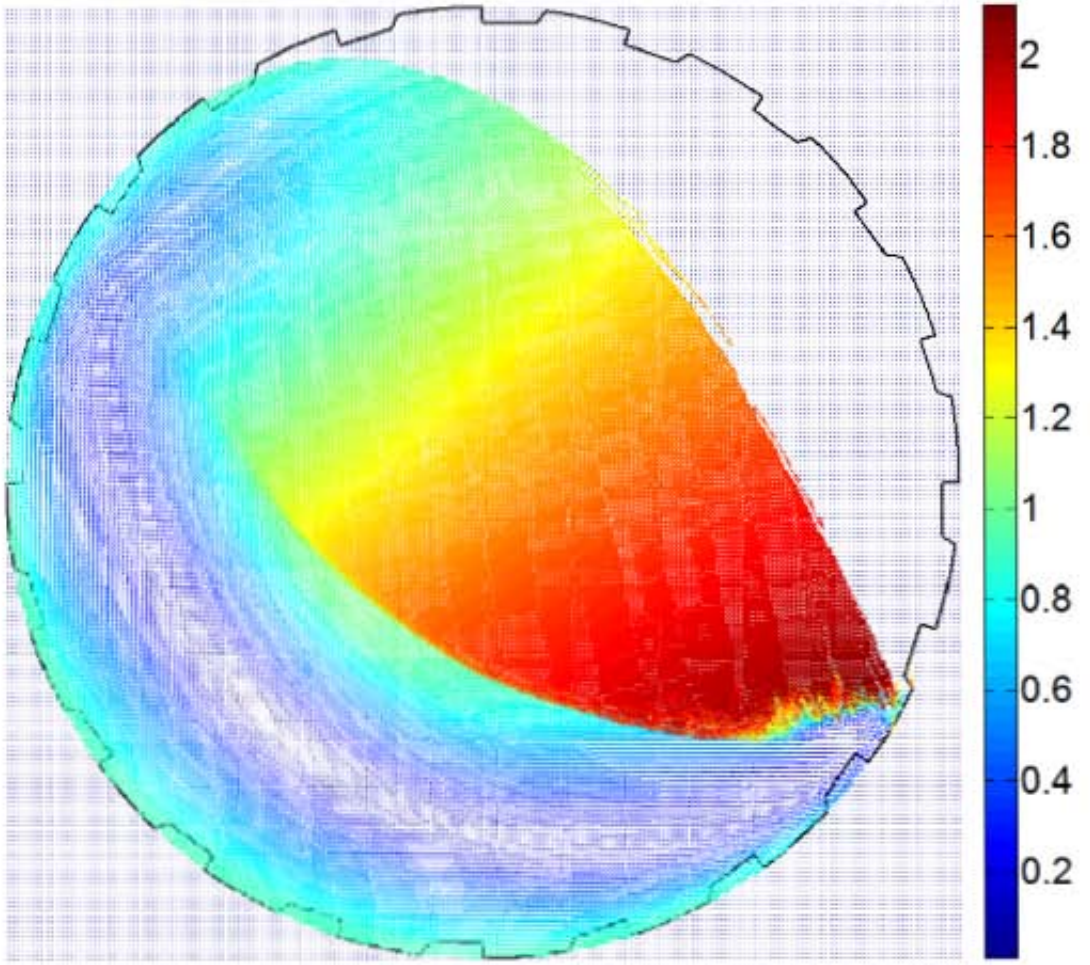


(c) 256x256 grid

Fig 6.7: Porosity plots at 80% critical speed

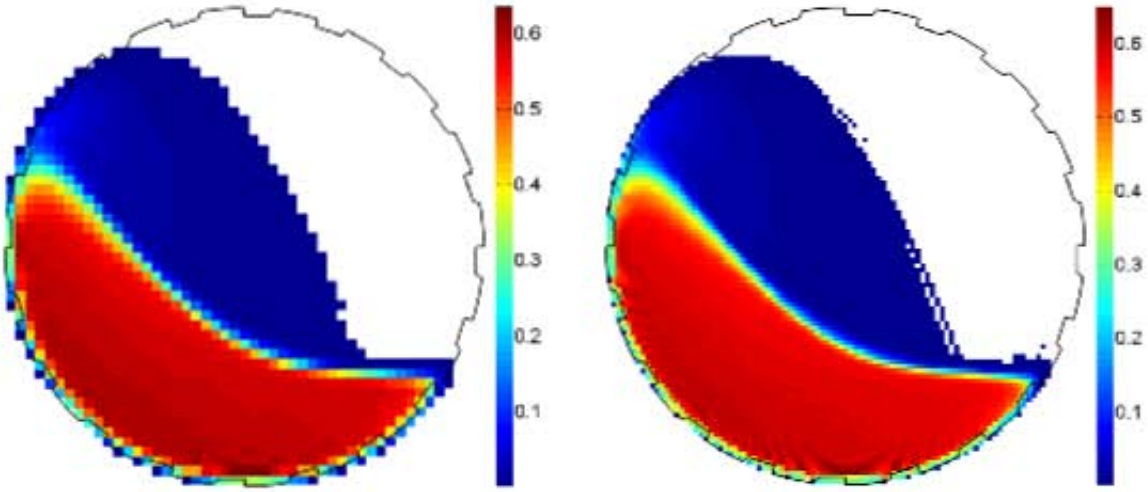


(a) 64x64 grid (b) 128x128 grid



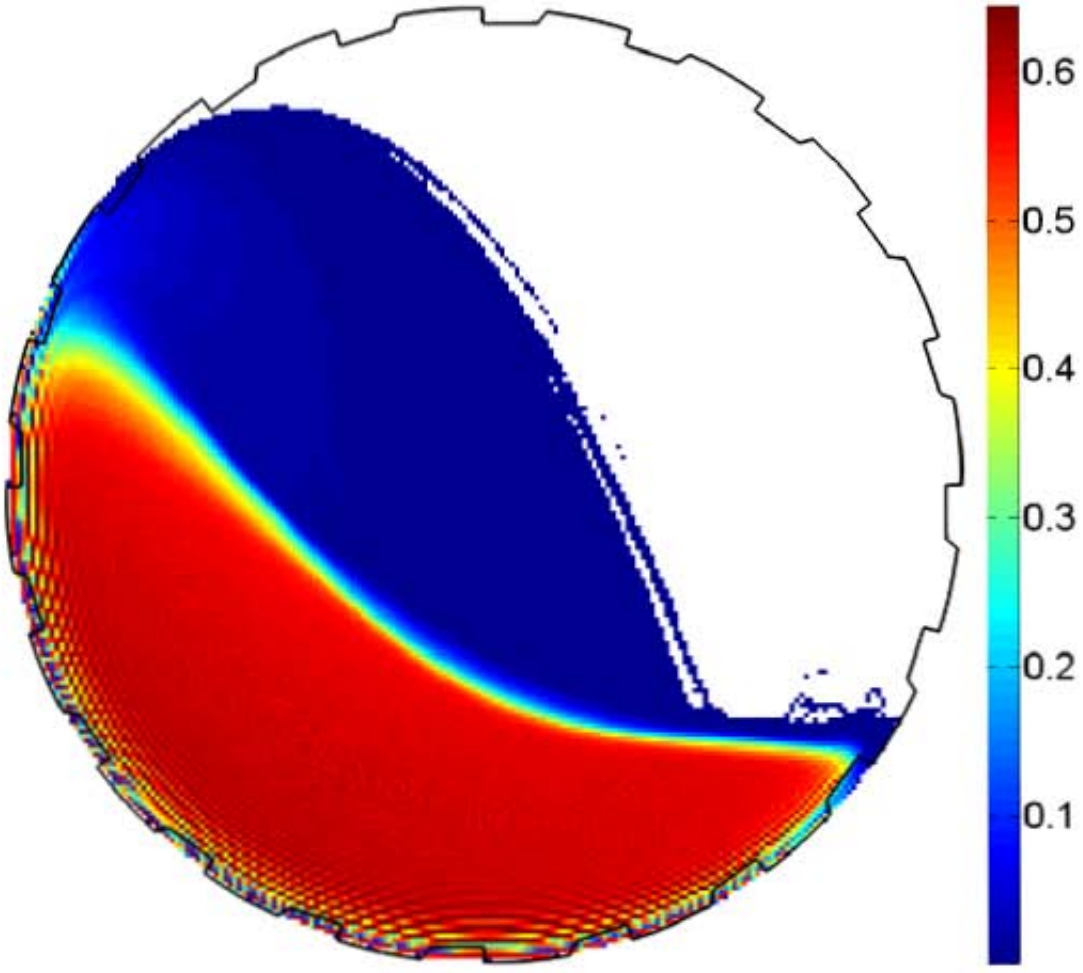
(c) 256x256 grid

Fig 6.8: Quiver plots at 80% critical speed



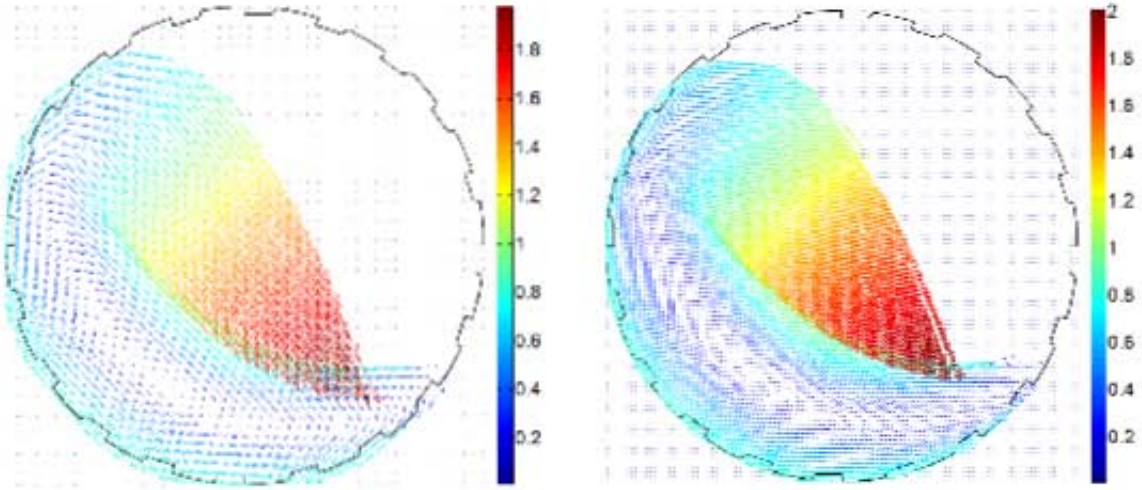
(a) 64x64 grid

(b) 128x128 grid

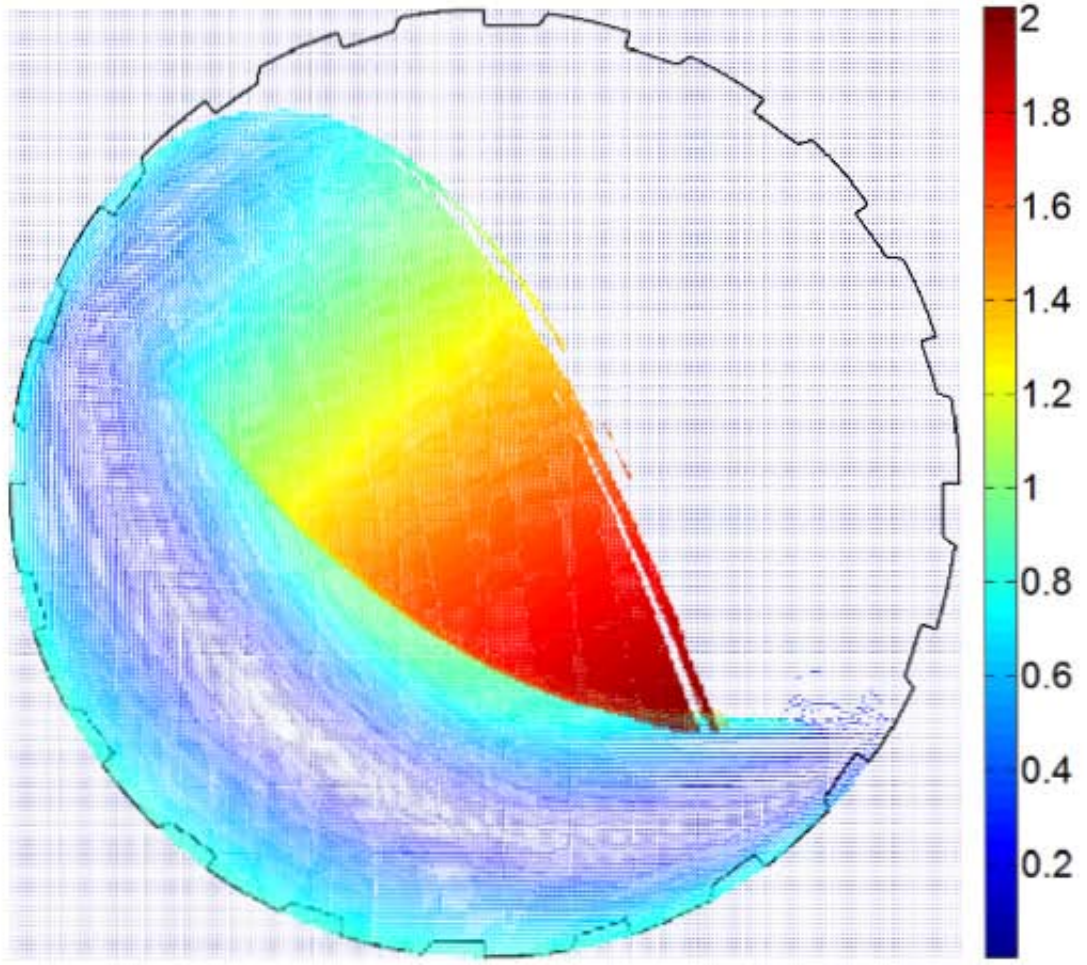


(c) 256x256 grid

Fig 6.9: Porosity plots at 70% critical speed

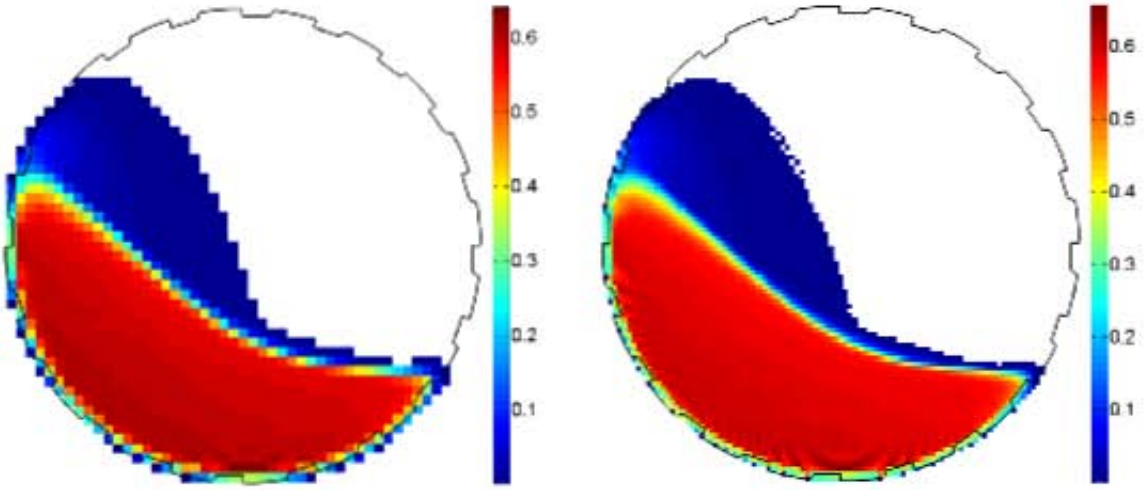


(a) 64x64 grid (b) 128x128 grid



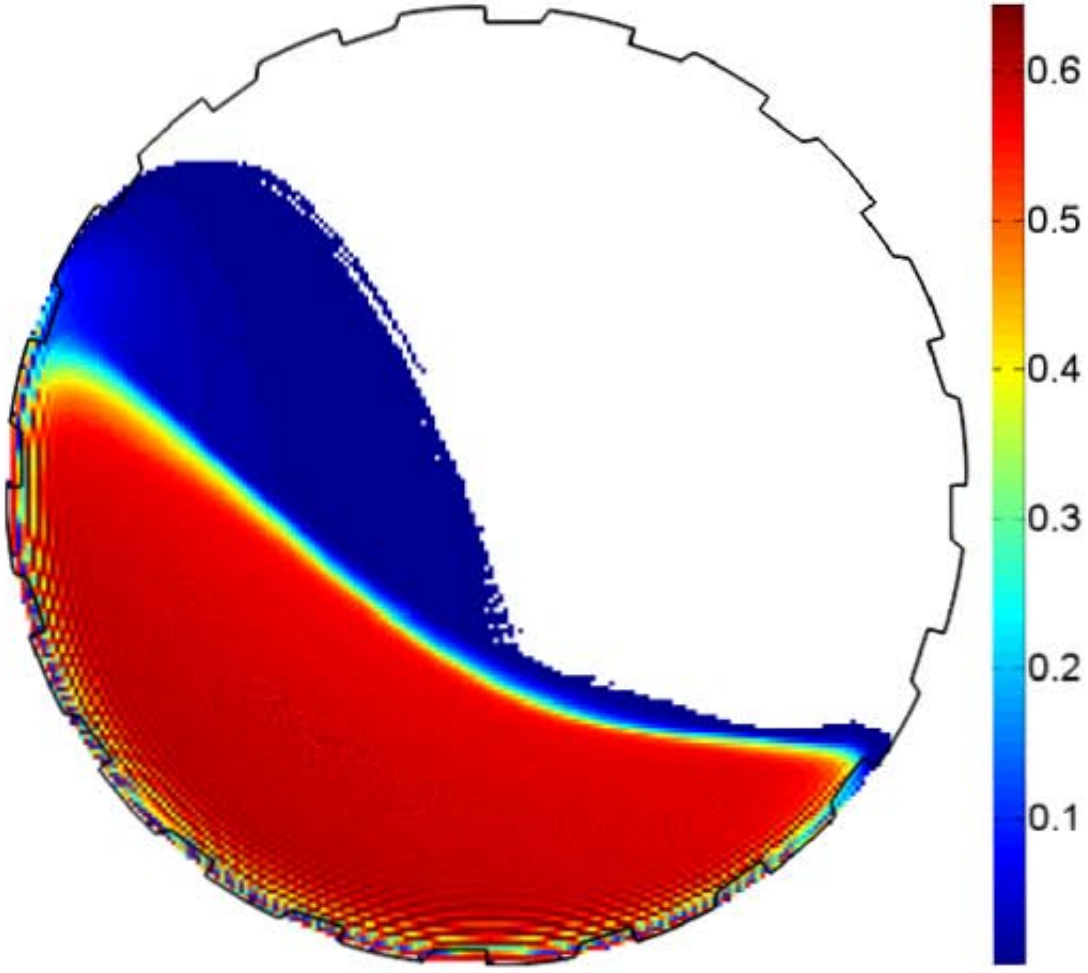
(c) 256x256 grid

Fig 6.10: Quiver plots at 70% critical speed



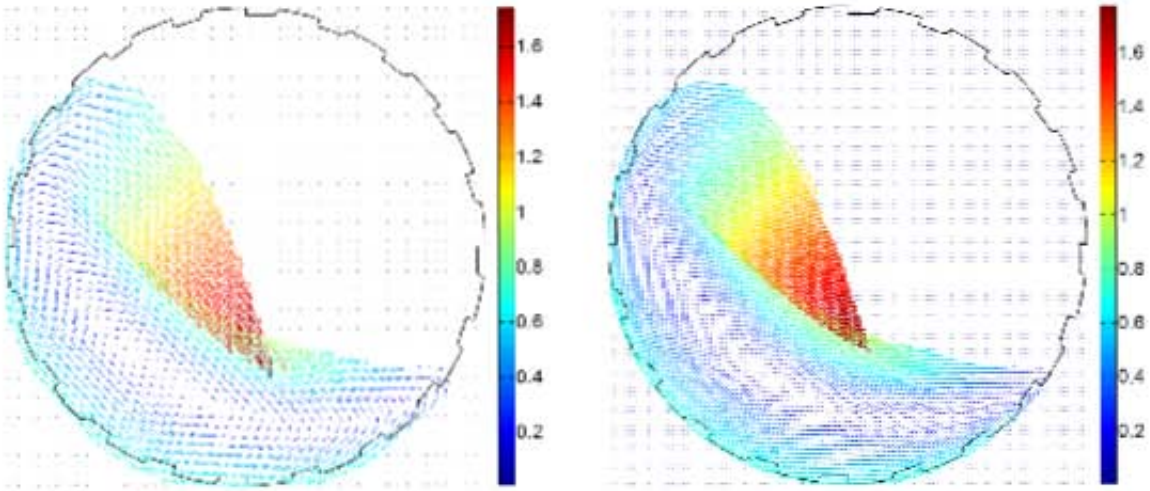
(a) 64x64 grid

(b) 128x128 grid



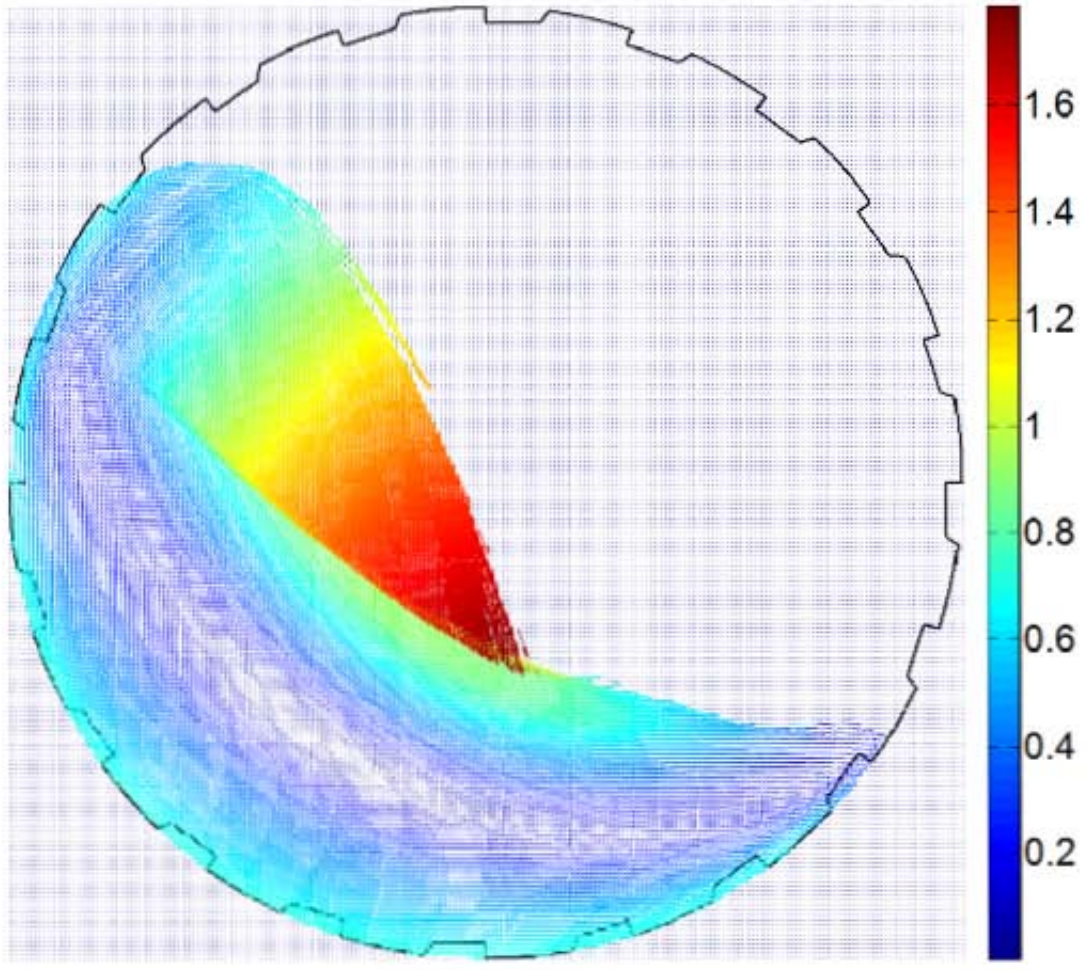
(c) 256x256 grid

Fig 6.11: Porosity plots at 60% critical speed



(a) 64x64 grid

(b) 128x128 grid



(c) 256x256 grid

Fig 6.12: Quiver plots at 60% critical speed

From the velocity plots, one can confirm that for mill speed of 60% critical, particles at the edge of the mill are traveling at ca. 0.7m.s^{-1}. The speed of the mill shell is given by $v = \omega * r$, where r is the radius of the mill, and $v = 4.85 * 0.15 = 0.73\text{m.s}^{-1}$ confirming that particle speeds are as expected. Similar confirmation for the 70% and 80% simulations can be made.

Porosity is defined in terms of solidicity by the following relation:

$$\phi = 1 - s.$$

where ϕ is porosity and s is solidicity defined by:

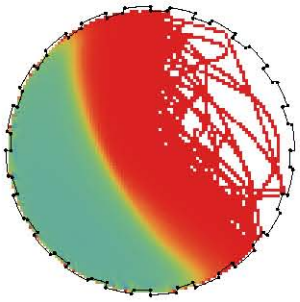
$$s = \frac{x * V_{particle}}{y * V_{voxel}}.$$

where x is the number of particles logged in the voxel and y is the number of timesteps data was logged.

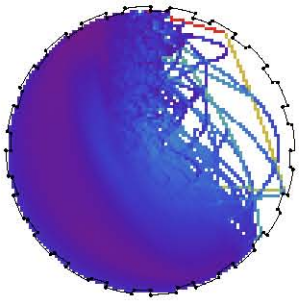
One observes a ring of very low values at the edges of the porosity plots, most distinctive in the 64x64 plots (Fig 6.7(a), 6.9(a) and 6.11(a)). This is due to the fact that solidicity is defined as the fraction of the volume of the particles in a voxel to the volume of the voxel and averaged over the number of timesteps for which data was logged. The volume of the voxel is the area of the grid cell multiplied by the length of the drum (in our case 0.27m), but at many timesteps part, or all, of a voxel may be filled with a lifter that is temporarily in that position. The other consideration is that since the mill is round and the voxels are rectangular prisms, part of the outermost voxels are outside the mill. Both these factors can be compensated for, and must be compensated for, when using properties of the data in sensitive theories such as granular flow modeling of the charge in the mill. However, for the purposes of this thesis, this correction is not necessary and the data has been left in raw form. It is important to note that this ring of low values is not an indication of an error in the simulation.

6.3 High friction particles to simulate shaped particles

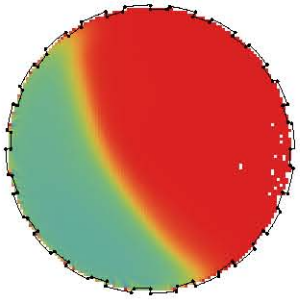
The parameter in the material properties to alter the dynamics of the charge most drastically has been found to be the friction coefficient governing the particle particle interaction. In an experiment using mustard seed shaped particles in a cylindrical drum [DRM+97] it was found that increasing the rotation speed of the drum produced an S-shape in the charge. Performing DEM on mustard seed shaped particles is difficult and we experimented with the idea that the effect of particle shape could be produced by simulating high friction particles. Fig. 6.13 shows simulation runs with increasing friction coefficient. Performing a DEM simulation of spherical particles with physically based friction coefficients in a cylindrical drum without lifters produces no effect even at ridiculously high mill speeds. The mill speed in these plots is only 50% critical speed and there is definite shape in the charge.



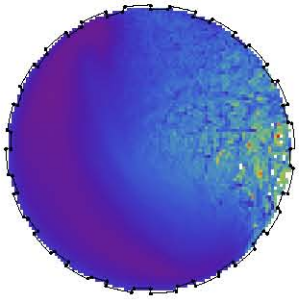
(a) Porosity plot - $\mu = 0.42$



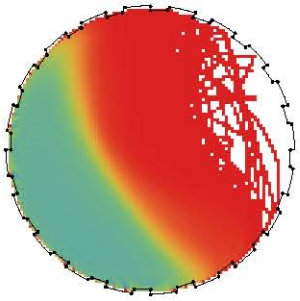
(b) Velocity plot - $\mu = 0.42$



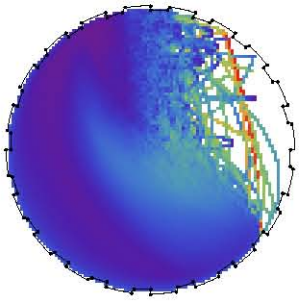
(c) Porosity plot - $\mu = 0.44$



(d) Velocity plot - $\mu = 0.44$



(e) Porosity plot - $\mu = 0.46$



(f) Velocity plot - $\mu = 0.46$

Fig 6.13: Simulation data from 300mm mill without lifters using particle representation of the mill with high friction particles.

6.4 Performance

Initially, the greatest effect on performance was seen to be caused by register spilling[19]. Profiling the collision detection and processing kernel revealed that once register spilling occurred, gigabytes of data were transferred between device memory, and memory local to each SM (streaming multiprocessor). L1 and L2 cache hit rates became low (ca 15%) due to cache ousting, increasing kernel duration many times. Once the code was optimised to operate within the 63 registers per thread limit, acceptable performance was achieved. The final version of the code required 58 registers per thread for the collide kernel, which is the only expensive kernel concerning registers. In Fig. 6.14. it is seen that global memory transfer to cache is just the size of the particle data. However, one notices that there are millions of requests to texture memory. Note that texture memory was used because the structure of texture memory allows for caching of memory stored in a 2D spatial locality. This assists with non-coalesced memory lookups. These occur when a particle is checking for collisions with particles in a neighbouring voxel in 1 dimension[20]. Unfortunately texture cache hit rate is 33.4%, as shown in Fig 6.14 and a large amount of data has to be transferred from global device memory. There are clever schemes using shared memory to alleviate this problem, and this will be addressed in future development of this code. Local memory to cache transfer is zero, indicating no register spilling occurred. Only a 30.82% occupancy was achieved. Occupancy of over 50% is able to hide memory latency. The limiting factor on occupancy was the number of registers per SM and the theoretical maximum was 33.33%. This is sensible, as with the large number of threads spawned, 1 per particle, there is a sufficient number of thread blocks spawned to not limit SM occupancy.

Performance was noticed to dramatically increase by 40% once steady state was reached. This is because thread divergence went down. Fig. 6.15 indicates a high efficiency at steady state with 3.7% of threads diverging and 5.5% of threads inactive. This gives clear indication of how well suited discrete element modeling is to GPU computing.

The simulation written in this thesis has a performance of about 13.795 seconds per 1000 timesteps at a particle count of 253125 at steady state on a Nvidia GTX560Ti graphics card. This translates to about 6.26 million timesteps per day. A comparison between simulations can be crudely done using the Cundall number:

$$C = N_t * \frac{N}{T},$$

where N_t is the number of timesteps processed, N is the particle count and T is the total simulation time [Cle09]. This is better stated as

$$C = \frac{t}{\Delta t} * \frac{N}{t/\Delta t/F} = N * F,$$

[19]Register spilling occurs when the number of registers required by a single thread exceeds the device registers per thread limit. This is caused by having too many live variables.

[20]This is an effect of storing particle information in 1 dimensional arrays

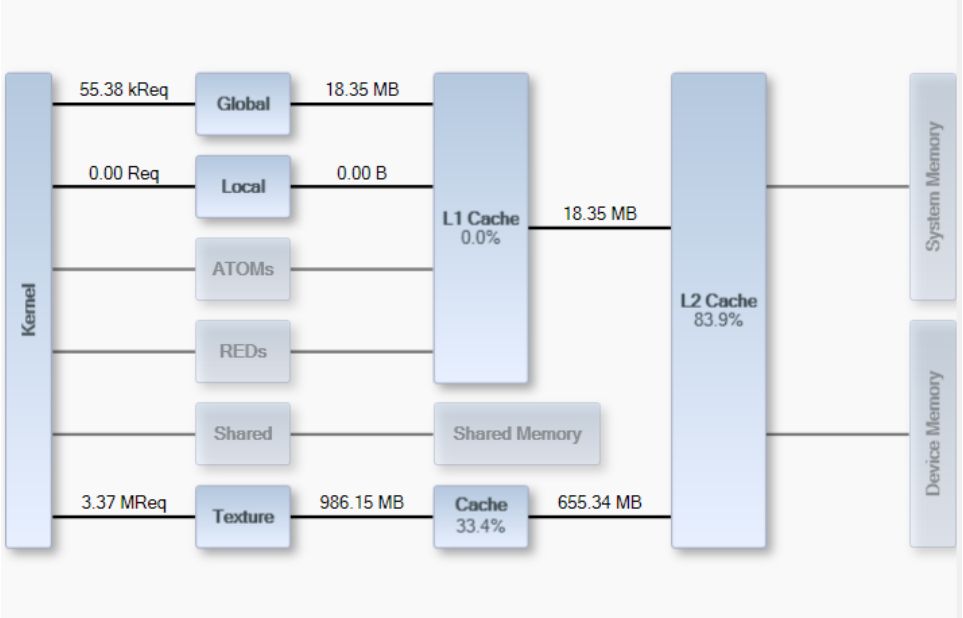


Fig 6.14: Profiling memory transactions - Parallel Nsight

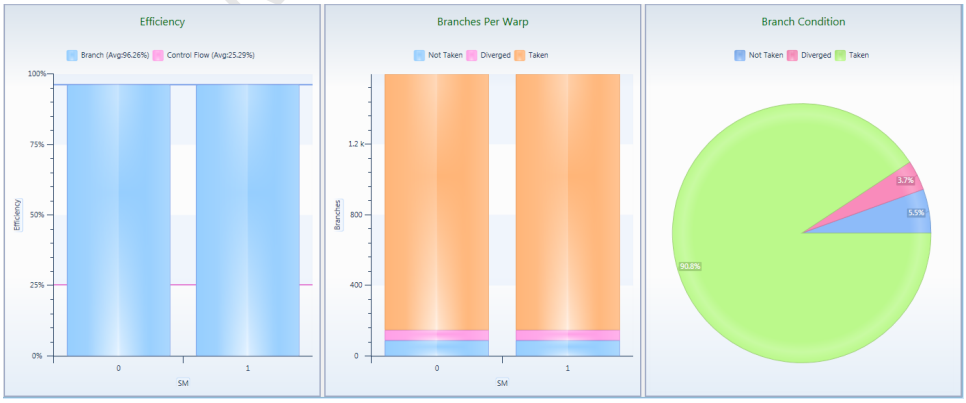


Fig 6.15: Profiling thread branching - Parallel Nsight

Author	Cundall number	Particle count
Longmore [L09]	$1.490 * 10^6$	$256000 * 4$
Harada et al. [HTK+07]	$2.08 * 10^6$	$16384 * 4$
Venetilo et al. [VC07]	$4.736 * 10^6$	128000
Ferrez [Fer01]	$2.023 * 10^4$	12000
This thesis	$1.8349 * 10^7$	253125

Fig 6.16: Comparison of Cundall numbers from various works.

where F is the number of frames that can be computed per second. Figure 6.16 gives a comparison to previous works. This is a very rough comparison. Ferrez implemented his work on a high-end symmetric multiprocessor (SMP) and it included a particle size distribution. Harad et al. simulated chess pieces composed of 4 particles of different shapes. Longmore simulated sand granules consisting of four spheres each. All of these simulations besides the one made by Ferrez are built on a GPU platform.

The focus of the work presented here was not performance, but the performance makes the tool useful. It is clear that many improvements can still be made to the algorithms and implementation presented. The CUDA framework limits one's control regarding this by abstracting one from GPU pipelining operations.

7 Conclusion and future development ideas

In the world of computer game development, discrete element modeling has been used to model physical elements as it is a way to simulate complex real-world systems by decomposing them into elements whose interactions are governed by established theory and have been shown to produce accurate results. All modern DEM simulations produced in this industry, where performance cannot be compromised, are built on GPU platforms. In this work, a GPU based scientific computing code has been built using machinery and computing theory from the computer games industry. A strict requirement on the physics has been maintained and elementary particle tests give confidence that the underlying mechanisms in the simulation are producing correct results.

In the investigation into effects inside tumbling mills, DEM has been used extensively. This work provides a high performance, flexible DEM tool for research into comminution theory. The performance demonstrated makes it suitable for research into the effects of design changes on tumbling mills and its flexibility allows it to be extended to test fundamental theories such as use of granular flow theory to characterise dynamics inside the mill. The ability to alter the contact model in whichever way is suitable has been demonstrated by the investigation into using high friction particles to produce the effects of shaped particles.

The control of the data processing and retrieval mechanisms in this code have allowed for on-the-fly data processing, which removes the burden of enormous quantities of data being produced by the simulation as well as the reducing the task of postprocessing. This factor also allows for the retrieval of individual contact information and tracking contacts, which otherwise produces an overwhelming amount of data.

7.1 Future work

The future development of this work is very exciting. The framework developed in this work is reasonably general. It includes modularity and is well suited to extension. The coupling of other codes, such as SPH or CFD codes, can be implemented at such a low-level that there needn't even be any additional communication through RAM! Such codes can be implemented as CUDA kernels, or combinations of kernels and with all the data already on the device, such extensions will be seamless. Such an extension would provide a means of simulating slurry flow inside a tumbling mill, which is a crucial aspect of the process of retrieving fines. This combines very nicely with theoretical research being conducted in the field [GTM10].

The rendering of the simulation is a component that has been neglected in this work. The development of an interface that would allow the user to dynamically control factors in the simulation would be of use. The inclusion of high quality lighting effects has been demonstrated on other projects to add little computational expense and would add a

nice facet to this project. It is definitely a strong consideration for a future development.

The introduction of shaped particles through the use of granules and cluster logic is a natural extension. Implementing a particle size distribution is also a necessary extension in the near future. Implementing a true rigid body DEM is not foreseen as a future work on this project. This would probably require the rework of many of the structures on which this program is based. The addition of contact and collision tracking is a slight extension to the existing data processing mechanisms, but this will be included as demand for specific data arises.

The extension of this work to support multiple GPU processors, or Tesla personal supercomputer systems is an extension that will allow this program to perform enormous DEM simulations ($> 10^7$ particles). GPU memory limitations are currently the limiting factor on simulation size. To take advantage of multiple GPU devices, the grid partitioning theory would have to be revised with careful consideration of the technical subtleties of such a platform.

References

[BTW87] P. Bak, C. Tang, K. Wiesenfeld. Self-organized criticality: An explanation of the 1/f noise. *Physical Review Letters*. 59(4):381-384. July 1987.

[CBI+03] P. W. Cox, S. Bakalis, H. Ismail, R. Forster, D. J. Parker, P. J. Fryer. Visualisation of three-dimensional flows in rotating cans using positron emission particle tracking (PEPT). *Journal of Food Engineering 60*. P. 229-240. 2003.

[CEI] Computational Engineering International. EnSight visualisation software. http://www.ceisoftware.com/.

[Cle09] P. Cleary. Ball motion, axial segregation and power consumption in a full scale two chamber cement mill. *Minerals Engineering*. (August 2009), 22 (9-10), P. 809-820. 2009.

[CM11] P. Cleary, S. Mead. Three dimensional avalanche modelling across irregular terrain using DEM: Comparison with experiment. *19th International Congress on Modelling and Simulation, Perth, Australia*. 2011.

[CS79] P. A. Cundall, O. D. L. Strack. A discrete numerical model for granular assemblies. *Geotechnique*. 29:47-65. 1977.

[Cle98] P. W. Cleary. Predicting charge motion, power draw, segregation and wear in ball mills using discrete element methods. *Minerals Engineering 11*. P. 1061-1080. 1998.

[DL11] M. J. Daniels, E. Lewis-Gray. Comminution efficiency attracts attention. *CEEC international Ltd. Bulletin*, October 2011. http://www.ceecthefuture.org/download-document/69-ausimm-nov-2011-comminution-efficiency-attracts-attention/Comminution efficiency_The Bulletin_Oct 2011.pdf. October 2011.

[DOE01] *Mining Industry of the future*. Office of industrial technologies (OIT), Office of Energy Efficiency and Renewable Energy, USA, doe/GO-102001-1157, www.oit.doe.gov/mining. February 2001.

[DRM+97] C. M. Dury, G. H. Ristow, J. L. Moss, M. Nakagawa. Boundary effects on the angle of repose in rotating cylinders. *Physical Review E, Volume 57, Number 4*. P. 4491-4497. 1997.

[EDEM] DEM Solutions. EDEM Simulation software. http://www.dem-solutions.com/

[Eri05] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann imprint, Elsevier Ltd. P. 136-142. 2005.

[Faz07] S. Fazekas. *Distinct Element Simulation of Granular Materials*. PhD thesis. Department of theoretical physics, Budapest university of technology and economics. Budapest. 2007.

[FB74] R. Finkel, J. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*. Volume 4, issue 1, P 1-9. 1974.

[Fer01] J. Ferrez. *Dynamic Triangulations for Efficient 3D Simulation of Granular Materials*. PhD thesis. Ecole Polytechnique Federale de Lausanne, 2001.

[FR81] M. D. Flavel, H. W. Rimmer. Particle breakage study in an impact breakage environment. *Reprint of paper presented at SME-AIME Annual Meeting, Chicago, USA*. SME Publications, Littleton, Ohio. 1981

[GBF03] E. Guendelman, R. Bridson, R. Fedkiw. Nonconvex rigid bodies with stacking. *ACM Transactions on Graphics*. Volume 22 Issue 3, P. 871-878. July 2003.

[GGH+98] E. Goles, G. Golzale, H. Herrmann, S. Martinez. Simple lattice model with inertia for sand piles. *Granular matter*. 1(3):137-140. 1998.

[Gre10] S. Green. *Particle Simulation using CUDA*. Nvidia. http://developer.download.nvidia.com/compute/DevZone/C/html/C/src/particles/doc/particles.pdf. May 2010.

[GTM10] I. Govender, G. B. Tupper, A. N. Mainza. Towards a mechanistic model for slurry transport in tumbling mills. *Minerals Engineering*. Volume 24. P. 230-235. 2010.

[GWL+03] N. Goodnight, C. Woolley, G. Lewin, D. Luebke, G. Humphreys. A multigrid solver for boundary value problems using programmable graphics hardware. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. P. 102-111. Eurographics Association Aire-la-Ville, Switzerland. Switzerland, 2003.

[Hal05] T. C. Hales. A proof of the Kepler Conjecture. *Annals of Mathematics 162*. P. 1065-1185. 2005.

[Har05] M. Harris. Fast fluid dynamics simulation on the GPU. *ACM SIGGRAPH 2005 Courses*. P. 220. Association for Computer Machinery. 2005.

[HC75] H. K. Hunt, F. R. E. Crossley. Coefficient of Restitution Interpreted as Damping in Vibroimpact. *ASME J Appl Mech*. P. 440-445. 1975.

[HKK07] T. Harada, S. Koshizuka, Y. Kawaguchi. Sliced data structure for particle-based simulations on GPUs. *5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*. P. 55-62. 2007.

[HL98] H. J. Herrmann, S. Luding. Modeling granular media on the computer. *Continuum Mechanics and Thermodynamics*. Volume 10, P. 189-23l.1998.

[HTK+07] T. Harada, M. Tanaka, S. Koshizuka, Y. Kawaguchi. Acceleration of rigid body simulation using graphics hardware. *Symposium on Interactive 3D Graphics and Games*. 2007.

[Kil02] J. Kilani. The business sense of energy management in mining. *Global mining initiative conference*. 2002.

[KLS+05] J. Kniss, A. Lefohn, R. Strzodka, S. Sengupta, J. D. Owens. Octree textures on graphics hardware. *ACM SIGGRAPH 2005 Sketches*. P. 16. Association for Computer Machinery. 2005.

[KSW04] P. Kipfer, M. Segal, R. Westermann. Uberflow: A GPU-based particle engine. *Graphics Hardware 2004*. P. 115-122. August 2004.

[L09] J. P. Longmore. *Towards Realistic Interactive Sand: A GPU-based Framework*. MSc thesis. University of Cape Town. 2009.

[Liu09] Y. Liu. Lecture 2 - Parallel Computing. *High Performance Computing with CUDA*. http://hpdm.ac.cn/PDF/Spring2009/CUDA_Parallel_computing.pdf. Graduate University of Chinese academy of Sciences, Research Center on Fictitious Economy and Data Science. 2009.

[Liu09(2)] Y. Liu. Lecture 4 - CUDA Memory. *High Performance Computing with CUDA*. http://hpdm.ac.cn/PDF/Spring2009/CUDA_memory.pdf. Graduate University of Chinese academy of Sciences, Research Center on Fictitious Economy and Data Science. 2009.

[LL86] L. Landau, E. Lifshitz. *Theory of Elasticity - Third Edition*. Volume 7. Butterworth-Heinemann imprint, Elsevier Ltd. 1986.

[Mir96] B. Mirtich. *Impulse-based dynamic simulation of rigid body systems*. MSc thesis. University of California Berkley. 1996.

[MR04] F. P. Di Maio, A. Di Renzo. Analytical solution for the problem of frictional-elastic collisions of spherical particles using the linear model. *Chemical Engineering Science (59)*. P. 3461-3475. 2004.

[MR04(2)] F. P. Di Maio, A. Di Renzo. Comparison of contact-force models for the simulation of collisions in DEM-based granular flow codes. *Chemical Engineering Science (59)*. P. 525-541. 2004.

[NG01] N. Nerone, S Gabbanelli. Surface fluctuations and inertia effects in sandpiles. *Granular Matter*3(1):117-120. 2001

[NV08] *Nvidia CUDA Programming Guide Version 2.1.* http://developer.download.nvidia.com/compute/cuda/2_1/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.1.pdf. 2008.

[NV10] *Nvidia CUDA Programming Guide Version 3.0.* http://developer.download.nvidia.com/compute/cuda/3_0/toolkit/docs/NVIDIA_CUDA_ProgrammingGuide.pdf. 2010.

[PDF+93] D. J. Parker, C. J. Broadbent, P. Fowles, M. R. Hawkesworth, P. McNeil. Positron emission particle tracking a technique for studying flow within engineering equipment. *Nuclear Instruments and Methods in Physics Research A326*. P. 592-607. 1993.

[PFC3D] Itasca PFC3D Version 4 Theory Manual.

[PTV+92] W. Press, S. Teukolsky, W. Vetterling, B. Flannery. *Numerical recipes in C: The art of scientific computing (2nd edition)*. Cambridge University Press. 1992.

[PWC+11] M. S. Powell, N. S. Weerasekara, S. Cole, R. D. Laroche, J. Favier. DEM modelling of liner evolution and its influence on grinding rate in ball mills. *Minerals Engineering 24*. P. 341-351. 2011.

[Sam84] H. Samet. The quadtree and related hierarchical data structures. *ACM Computer Survey*. Volume 16 Issue 2, P. 187-260. 1984.

[Sav79] S. B. Savage. Gravity flow of cohesionless granular materials in chutes and channels. *Journal of Fluid Mechanics*, Volume 92, P. 53-96. 1979.

[SHG09] N. Satish. M. Harris. M. Garland. Designing Efficient Sorting Algorithms for Manycore GPUs. *Proc. 23rd IEEE International Parallel & Distributed Processing Symposium*. May 2009.

[SK10] J. Sanders, E. Kandrot. CUDA by example: An Introduction to General-Purpose GPU Programming. Addison-Wesley, USA. 2010.

[Spe11] E. Specht. *The best known packings of equal circles in a circle (complete up to N = 1100)*. http://hydra.nat.uni-magdeburg.de/packing/cci/cci.html. November 2011.

[TTI92] Y. Tsuji, T. Tanaka, T. Ishida. Lagrangian numerical simulation of plug flow of cohesionless particles in a horizontal pipe. *Powder technology 71*. P. 239-250. 1992.

[VC07] J. Venetillo, W. Celes. GPU-based particle simulation with interciollisions. *The Visual Computer*. P. 851-860. 2007.

[W86] S. Wolfram. Random sequence generation by cellular automata. *Advances in Applied Mathematics, vol. 7*. P. 123-169. June 1986.

[WeiHCP] E. W. Weisstein. *Hexagonal Close Packing*. Mathworld - A Wolfram Web Resource. http://mathworld.wolfram.com/HexagonalClosePacking.html.

[WeiKC] E. W. Weisstein. *Kepler Conjecture*. Mathworld - A Wolfram Web Resource. http://mathworld.wolfram.com/KeplerConjecture.html.

[WC10] D. M. Williams, P. Castonguay, Stanford University. A Fast, Scalable High-Order Unstructured Compressible Flow Solver. *GPU Technology Conference*. San Jose, California, USA, 2010.

[ZB05] Y. Zhu, R. Bridson. Animating sand as a fluid. *ACM Trans. Graph.*. Volume 24 Issue 3, P. 965-972. 2005.

[ZW96] D. Zhang, W. J. Whiten. The calculation of contact forces between particles using spring and damping models. *Powder Technology 88*. P. 59-64. 1996.

[ZY03] H. P. Zhu, A. B. Yu. The effects of wall and rolling resistance on the couple stress of granular materials in vertical flow. *Physica A, Vol. 325*. P. 347-360. 2003.